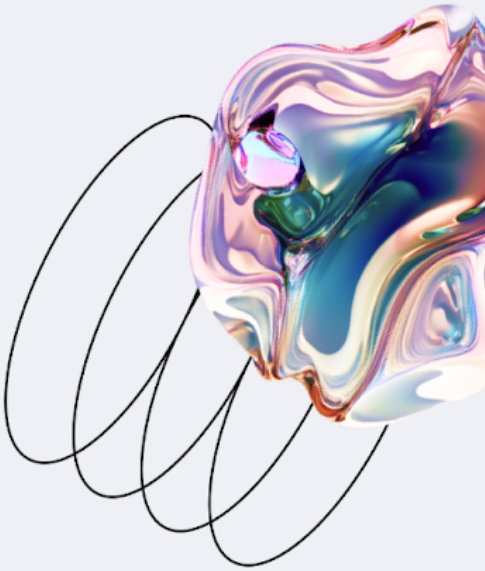
GeekBrains



Партицирование данных

ETL: автоматизация подготовки данных

Оглавление

Введение

Всем привет! Сегодня у нас четвертая лекция курса «ETL: автоматизация подготовки данных».

На прошлых уроках мы рассмотрели основные аспекты процесса ETL и поговорили о лучших практиках применения. Обсудили, что такое бизнес-аналитика, и подробно поговорили про таблицы фактов и измерений. А также узнали, как и зачем получать денормализованные таблицы из нормализованных.

Сегодня поговорим о партицировании данных и его видах. Посмотрим, как с помощью PostgreSQL создать партиции из базовой таблицы.

Термины, используемые в лекции

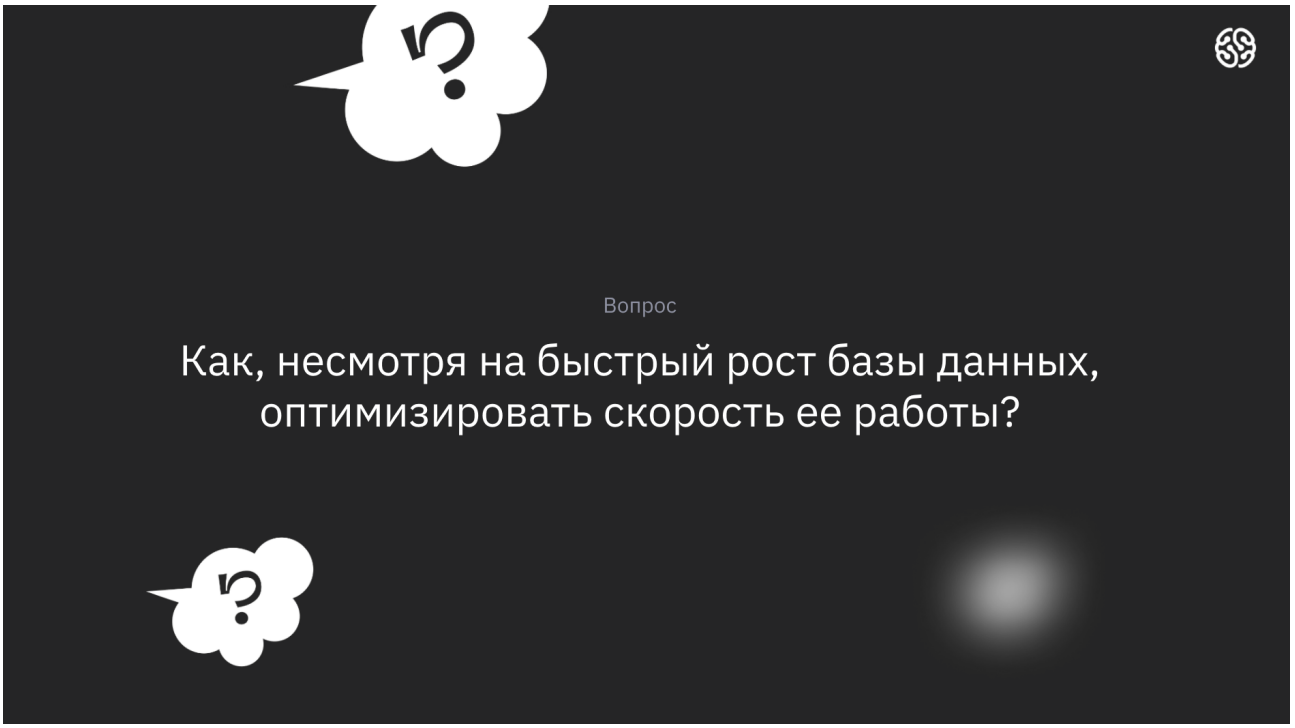
Горизонтальное партицирование (шардинг) — каждая часть (шард) представляет собой отдельное хранилище, но у всех общая схема. Части содержат набор данных: например, все заказы от группы клиентов.

Вертикальное партицирование — каждая часть содержит подмножество полей для элементов в хранилище данных. Поля делятся по шаблону их использования. Например, часто используемые поля размещаются в одной вертикальной секции, а редко используемые — в другой.

Партицирование (секционирование) данных

Проблема роста данных с каждым днем становится все более серьезной и актуальной. Например, база данных аудиторской компании может увеличиваться на 65 000 000 строк в год, то есть практически 180 000 строк в день. И это не самый быстрый прирост.

Какие варианты решения этой проблемы есть у нас?



Ответ: самый простой вариант — разбить стремительно растущие таблицы на части. В этом поможет партицирование или шардирование (разницу между понятиями рассмотрим позже).

> 💡 **Партицирование** — это метод разделения данных между таблицами или системами с целью ускорить запросы к данным и лучше организовать их. Данные можно разделить множеством способов в зависимости от их типа и способа получения.

Зачем нужно разделять данные?

Предположим, что у нас есть таблица с информацией об акциях компаний. В ней есть столбцы:

- company_id (Primary Key)
- company_name (String)
- location (String)
- stock_price (Float)
- percentage_change (Float)
- average_price (Float)

По мере того как мы добавляем данные в таблицу, ее размер растет. Допустим, текущий размер — около 50 000 000 строк, и мы продолжаем каждый день добавлять компании, запрашивать цены акций старых компаний и обновлять среднюю цену акций в реальном времени.

Проблема такой системы в том, что она не масштабируема: мы добавляем данные, а время, нужное для выполнения простого запроса, увеличивается. Вставка новой строки или ее обновление будут происходить дольше. Управлять такой системой сложно, она вызовет проблемы у бизнеса.

Решение — разделить большую таблицу на части (например, по диапазонам в 100 000 записей). Работать с ними можно будет быстрее, а управлять — легче.

Виды партицирования

Три распространенных стратегии партицирования данных:

- **Горизонтальное партицирование** (шардинг) — каждая часть (шард) представляет собой отдельное хранилище, но у всех общая схема. Части содержат набор данных: например, все заказы от группы клиентов.
- **Вертикальное партицирование** — каждая часть содержит подмножество полей для элементов в хранилище данных. Поля делятся по шаблону их использования. Например, часто используемые поля размещаются в одной вертикальной секции, а редко используемые — в другой.

- **Функциональное партицирование** — данные группируются в соответствии с их использованием в каждом связанном контексте в системе. Например, система для нужд электронной коммерции может хранить данные о счетах в одном разделе, а данные о товарных запасах — в другом.

Эти стратегии не исключают друг друга, при разработке схемы партицирования рекомендуем учитывать все три. Например, можно разделить данные на сегменты и затем использовать вертикальное секционирование, чтобы дальше разделить данные в каждом сегменте.

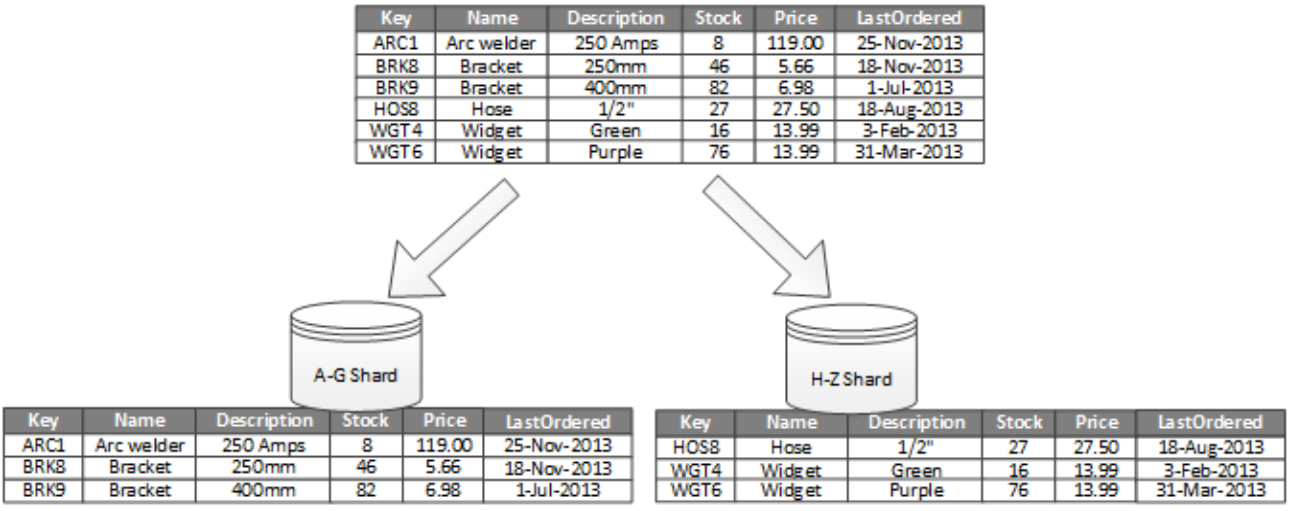
Горизонтальное партицирование

Другое название горизонтального партицирования — шардинг.

Шардинг — это повтор (копирование) схемы и разделение данных на основе ключа шардинга на отдельный экземпляр таблицы данных для распределения нагрузки.

У каждой распределенной таблицы есть один ключ сегмента. Он может содержать любое количество столбцов. Ключ шардинга таблицы определяет, в каком разделе хранится данная строка таблицы. Когда вы выполняете запрос INSERT, вычисляется хеш-функция значений в столбце или столбцах, которые составляют ключ сегмента, который создает номер шарда, в котором должна храниться строка. Затем операция INSERT выполняется в соответствующей части БД.

В примере ниже данные инвентаризации товаров разделены на сегменты на основании ключа товара. Каждый сегмент содержит данные для непрерывного диапазона ключей сегментов (A-G и H-Z) в алфавитном порядке.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Источник: learn.microsoft.com

Горизонтальное партицирование может помочь повысить производительность БД. Например:

- **Уменьшить размер индекса.** Поскольку таблицы разделены и распределены по нескольким серверам, общее количество строк в каждой таблице в каждой БД уменьшается. Соответственно, уменьшается и размер индекса, что повышает производительность поиска.
- **Распределить БД по нескольким машинам.** Каждый шард может быть размещен на отдельном оборудовании, а несколько шардов — на нескольких компьютерах. Можно распределять базу данных по большому количеству машин, значительно повышая производительность.
- **Сегментировать данные по географии.** Если сегмент базы данных основан на реальной сегментации данных (например, европейские и американские клиенты), можно автоматически вывести соответствующее членство в сегменте и запросить только соответствующий шард.

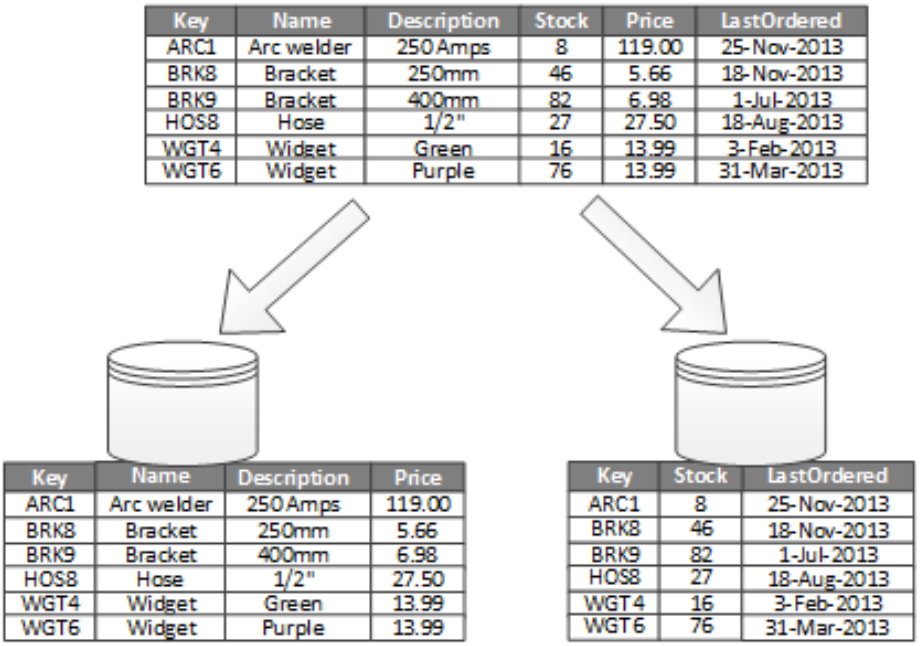
При этом шардинг следует использовать только тогда, когда все другие варианты оптимизации не подходят, потому что вместе с преимуществами есть и потенциальные проблемы:

- **Сложность SQL.** Количество ошибок может увеличиться, потому что разработчикам приходится писать более сложный SQL для обработки логики сегментирования.
- **ПО может дать сбой.** Дополнительное программное обеспечение, которое разделяет, балансирует, координирует и обеспечивает целостность, может дать сбой.
- **Единая точка отказа.** Повреждение одного сегмента из-за проблем с сетью, аппаратным обеспечением или системой приводит к отказу всей таблицы.
- **Сложность отказоустойчивого сервера.** У отказоустойчивых серверов должны быть копии групп шардов.
- **Сложность резервных копий.** Резервные копии БД отдельных сегментов должны быть согласованы с резервными копиями других сегментов.
- **Операционная сложность.** Добавление и удаление индексов, добавление и удаление столбцов, изменение схемы становится намного сложнее.

Вертикальное партицирование

Вертикальное партицирование чаще всего применяют, чтобы сократить количество операций ввода-вывода и расходов на производительность, связанных с обращением к часто используемым элементам.

На схеме ниже показан пример вертикального партицирование. В нем разные свойства элемента данных хранятся в разных секциях. В одной секции — данные, которые используются чаще всего: название товара, его описание и цена. В другой — данные о товарных запасах: их объем и дата последнего заказа.



Источник: learn.microsoft.com

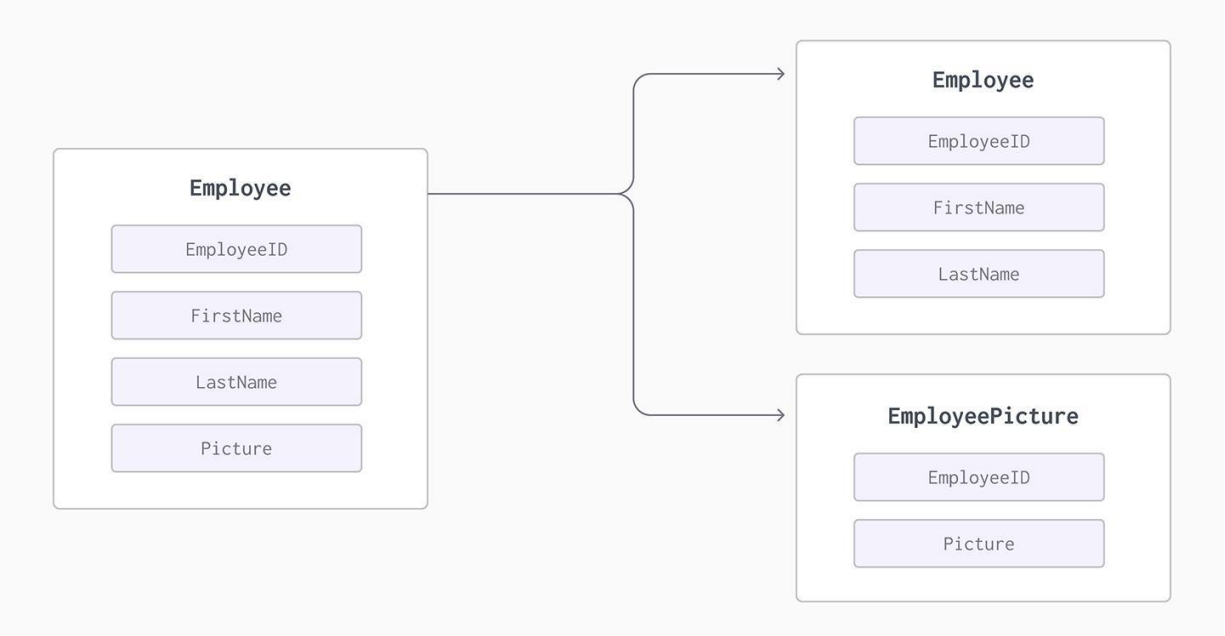
В примере приложение регулярно запрашивает имя, описание и цену продукта при отображении подробных сведений о продукте для клиента. Данные об объеме запасов и дате последнего заказа хранятся в отдельной секции, так как эти два элемента часто используются вместе.

Преимущества вертикального секционирования:

- Можно разделить относительно редко меняющиеся данные (название товара, его описание, цену) и часто меняющиеся данные (объем товарных запасов и дата последнего заказа). Редко меняющиеся данные можно кэшировать в памяти.
- Конфиденциальные данные можно хранить в отдельной секции с дополнительными средствами защиты.

- Вертикальное партицирование способно сократить количество необходимых операций одновременного доступа к данным.
- Вертикальное партицирование выполняется на уровне сущности в хранилище данных, частично нормализуя сущность и преобразуя ее из широкого элемента в набор узких элементов. Это идеально для хранилищ данных со столбцами (HBase и Cassandra). Если данные в коллекции столбцов меняются редко, можно использовать хранилища со столбцами в SQL Server.

Ниже — еще один пример вертикального партицирования: личные данные отделяют от фотографии профиля, чтобы совершать запросы данных быстрее (строковые данные занимают гораздо меньше места, чем фотографии).



Источник: singlestore.com

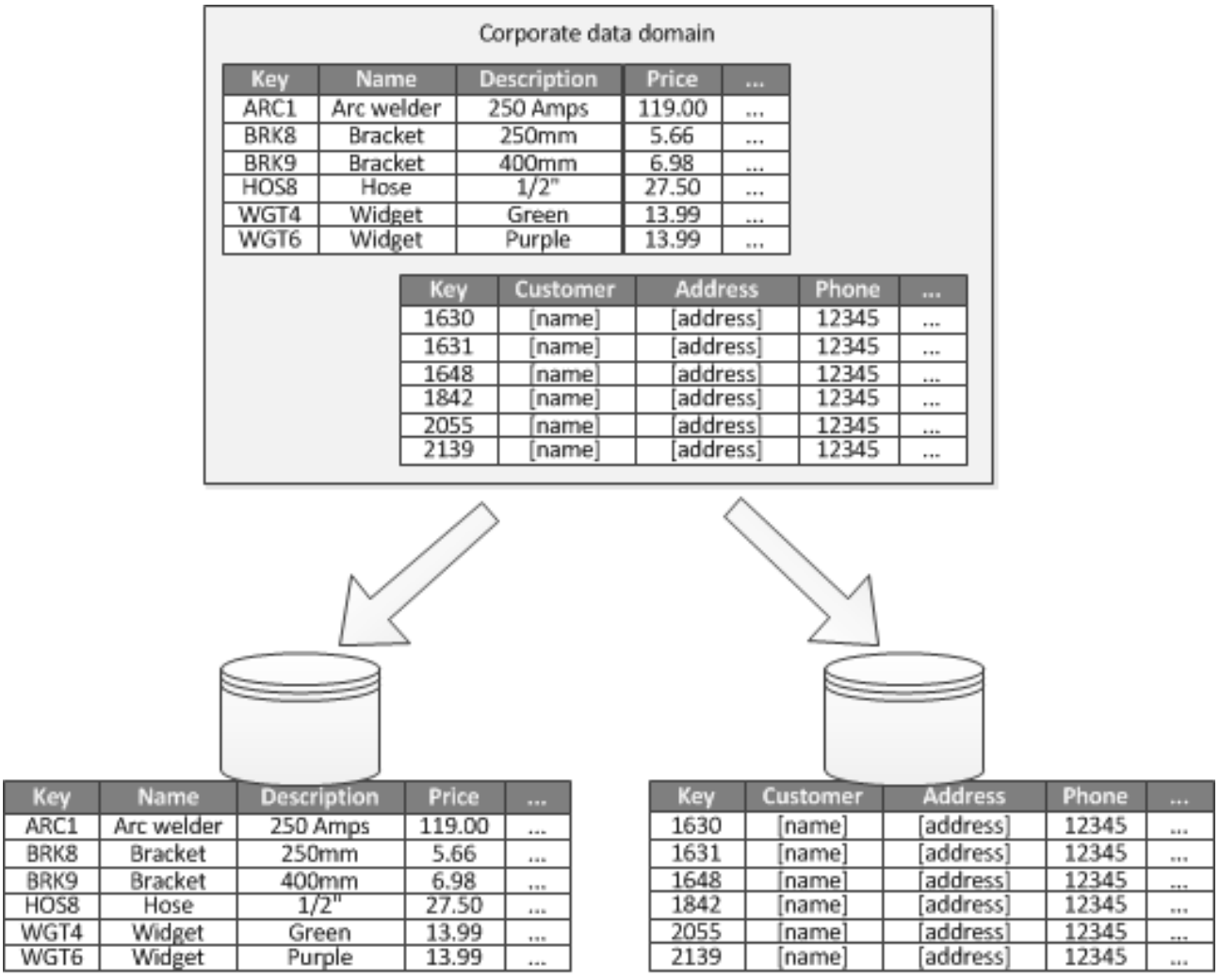
Когда следует делать вертикальное партицирование таблиц:

- Размер таблицы более 2 ГБ. Такие таблицы всегда следует рассматривать в качестве кандидатов на партицирование.
- Таблицы, содержащие исторические данные, в которых новые данные добавляются в самый новый раздел. Типичный пример — историческая таблица, в которой обновляются данные только за текущий месяц, а остальные 11 месяцев доступны только для чтения.
- Содержимое таблицы нужно распределить по разным типам устройств хранения.

Функциональное партицирование

Когда можно идентифицировать связанный контекст для каждой бизнес-области приложения, функциональное партицирование позволяет повысить уровень изоляции и ускорить доступ к данным. Кроме того, функциональное партицирование используют для разделения данных, предназначенных для чтения и записи и только для чтения.

На схеме ниже — пример функционального секционирования: данные инвентаризации отделены от данных клиентов.



Источник: learn.microsoft.com

Эта стратегия секционирования поможет уменьшить количество конфликтов доступа к данным в разных частях системы.

Методы партицирования

Теперь обсудим методы партицирования. Выделим четыре основных:

1. **Партицирование на основе хеша** (hash based) — вы никак не управляете партицированием, просто указываете, по какому полю строить хеш и сколько «подтаблиц» создавать. Так гораздо быстрее происходит выборка по указанному полю. В некоторых случаях можно достичь равномерного разброса и ускорения записи данных.

2. **Партицирование на основе списка** (list based) — по точному списку значений. Разбивать на партиции необходимо либо исходя из соображений оптимизации выборки (чаще), либо из соображений оптимизации записи (реже).

   Идеальный вариант — разбить таблицу на максимально возможное количество партиций так, что бы 90% всех выборок происходило в пределах одной партиции. Если у вас сложная логика выборки (например, объекты расположенные в северных кварталах города, ID которых идут в разнобой), иногда есть смысл перечислять их принудительно.

3. **Партицирование на основе диапазона** (range based) — по диапазону значений. Например, партицирование по дате. Записи за один месяц (год, день, час в зависимости от потребности) разбиваются на отдельные партиции.

4. **Комбинированное партицирование** объединяет два или более методов партицирования для столбца. Метод предлагает преимущество разделения в нескольких измерениях. Данные сначала разбиваются на разделы одним способом, а выходные разделы снова разбиваются процедурой.

   Типы комбинированного партицирования:

   - Комбинация «диапазон — диапазон»
   - Комбинация «диапазон — хэш»
   - Комбинация «диапазон — список»
   - Комбинация «список — диапазон»
   - Комбинация «список — хэша»
   - Комбинация «список — список»

Данные сначала разбиваются по первому методу из названия, а затем по второму. В некоторых случаях такая же процедура используется для партиций.

Преимущества и недостатки партицирования

Преимущества:

- Улучшает производительность запросов.
- Легче управлять данными.
- Позволяет получить доступ к большой части одного раздела.
- Система обладает высокой доступностью и масштабируемостью.
- Задачи администрирования становятся проще.
- Гибкое размещение индекса.
- Меньшее дерево индекса или сканирование таблицы при запросе в одной партиции.
- Запросы могут обращаться к разным партициям параллельно.

Недостатки:

- Автоматизация изменений в разделенной таблице может быть затруднена.
- Операции только с метаданными могут быть заблокированы операциями DML до тех пор, пока не будет получена блокировка модификации схемы.
- Файловые группы и файлы должны управляться, если разделы размещены в отдельных файловых группах.
- Несколько БД или экземпляров не могут быть объединены.
- Партицированные таблицы плохо работают с инструментами, связанными с ORM.

Партицирование на примере PostgreSQL

Создадим простую таблицу Posts, которая будет хранить посты. Таблицы такого типа встречаются во множестве проектов.

```
CREATE TABLE posts(id bigint not null,
                   category_id int not null,
                   author character varying not null,
                   rate int not null,
                   title character varying);
```

У постов в таблице будет id, категория, автор, оценка и заголовок.

Сделаем вертикальное партицирование, то есть разобьем таблицу по признакам. Для этого нужно будет выполнить два действия:

1. Указать у партиции (например, posts_1), что она будет наследоваться таблицей posts. Posts — базовая таблица, содержащая всю структуру. Создавая партицию, будем указывать, что она наследуется базовой таблицей.

   У наследованной таблицы будут все колонки родителя — той базовой таблицы, которую мы указали. Также у нее могут быть свои колонки, которые мы дополнительно добавим. Она будет полноценной таблицей, но унаследованной от родителя, и в ней не будет ограничений, индексов и триггеров от родителя — это очень важно. Если вы на базовой таблице насоздаете индексы и унаследуете ее, то в унаследованной таблице индексов, ограничений и триггеров не будет.

2. Поставить ограничения. Это будет проверка, что в таблицу будут попадать данные только с таким признаком.

```
CREATE TABLE posts_1(
  CHECK (category_id = 1)
  ) INHERITS (posts)
```

Эта команда создает таблицу, в которую попадут посты только первой категории.

Типы проверок для партицирования таблиц:

- Строгое значение (CHECK (category_id = 1))
- Список значений (CHECK (category_id IN (1,3,7,10))
- Диапазон значений (CHECK (rate > 10 AND rate < 100))

Остановимся подробнее, потому что проверка поддерживает оператор BETWEEN (наверняка вы его знаете).

```
CREATE TABLE posts_rate_10_200 (
    CHECK (rate BETWEEN 10 AND 200 )
) INHERITS (posts)

CREATE TABLE posts_rate_200_300 (
    CHECK (rate BETWEEN 200 AND 300 )
) INHERITS (posts)
```

Так можно сделать, но это плохая практика.

Можно, потому что PostgreSQL поддерживает такое действие. В первую партицию попадают данные между 100 и 200, а во вторую — между 200 и 300. В какую из этих партиций попадет запись с рейтингом 200? Неизвестно, как повезет.

Так лучше не делать. Нужно указывать строгое значение: в первую партицию будут попадать значения больше 100 и меньше либо равно 200, во вторую — больше 200 (но не 200) и меньше либо равно 300.

```
CREATE TABLE posts_rate_10_200 (
    CHECK (rate > 10 AND rate ⩽ 200 )
) INHERITS (posts)

CREATE TABLE posts_rate_200_300
    CHECK (rate > 200 AND rate ⩽ 300 )
) INHERITS (posts)
```

Запомните это и четко прописывайте все условия проверки, иначе есть риск не узнать, в какую из партиций попадут данные.

Также не стоит создавать партиции по разным полям: например, чтобы в первую партицию попадали записи с category_id=1, а во вторую — с рейтингом 100.

```
CREATE TABLE posts_1 (
  CHECK (category_id = 1)
) INHERITS (posts)

 CREATE TABLE posts_1 (
  CHECK (rate = 100)
) INHERITS (posts)
```

Неизвестно, в какую из партиций попадет эта запись. Партицировать стоит по одному признаку, по какому-то одному полю — это очень важно.

Давайте рассмотрим нашу партицию. Это таблица posts_1, в которую будут попадать только записи с категорией, равной 1. Таблица будет унаследована от базовой таблицы posts.

На базовую таблицу мы должны добавить некоторое правило, чтобы, когда мы будем работать с основной таблицей news, вставка на запись с category_id = 1 попала именно в ту партицию, а не в основную. Мы указываем простое правило, называем его как хотим, говорим, что когда данные будут вставляться в news с category_id = 1, вместо этого мы будем вставлять их в news_1. Это правило создается на базовой таблице.

```
CREATE RULE posts_insert_to_1 AS ON INSERT TO posts
WHERE ( category_id = 1)
DO INSTEAD INSERT INTO posts_1 VALUES (NEW.*)
```

Давайте сделаем несколько частей таблицы, чтобы посмотреть, как все работает на деле. Для начала добавим еще одну партицию и правила для базовой таблицы.

```
CREATE RULE posts_insert_to_1 AS ON INSERT TO posts
WHERE ( category_id = 1)
DO INSTEAD INSERT INTO posts_1 VALUES (NEW.*);

CREATE RULE posts_insert_to_2 AS ON INSERT TO posts
WHERE (category_id = 2)
DO INSTEAD INSERT INTO posts_2 VALUES (NEW.*);
```

После того, как мы создали правила добавления новых строк, рассмотрим пример добавления данных.

```sql
INSERT INTO posts
(id, category_id, title, author, rate)
VALUES
(1, 1, 'Post #1', 'Ilon', 1);

INSERT INTO posts
(id, category_id, title, author, rate)
VALUES
(2, 2, 'Post #2', 'Vova', 10);

INSERT INTO posts
(id, category_id, title, author, rate)
VALUES
(3, 3, 'Post #3', 'Boris', 1);
```

Выберем всю дату из полученной таблицы SELECT * FROM posts.

```
 id | category_id | author | rate |  title
----+-------------+--------+------+---------
  3 |           3 | Boris  |    1 | Post #3
  1 |           1 | Ilon   |    1 | Post #1
  2 |           2 | Vova   |   10 | Post #2
(3 rows)
```

У нас нет партиции для category_id = 3, но это не проблема, потому что данные, для которых нет созданного правила в базовой таблице, просто помещаются в эту базовую таблицу. Проверим это, сделав выборки из наших партиций.

```
postgres=# SELECT * FROM posts_1;
 id | category_id | author | rate |  title
----+-------------+--------+------+---------
  1 |           1 | Ilon   |    1 | Post #1
(1 row)

postgres=# SELECT * FROM posts_2;
 id | category_id | author | rate |  title
----+-------------+--------+------+---------
  2 |           2 | Vova   |   10 | Post #2
(1 row)

postgres=# SELECT * FROM posts_3;
ERROR:  relation "posts_3" does not exist
LINE 1: SELECT * FROM posts_3;
```

Видно, что таблицы posts_3 не существует, а из таблиц posts_1 и posts_2 дата выбирается корректно. Получается, партиции нет, а дата есть. Хоть партицирование

и применено к базовой таблице, основная таблица все равно существует. Она настоящая, может хранить данные, с помощью оператора ONLY можно выбрать данные только из этой таблицы, и мы можем найти, что эта запись здесь спряталась.

```
postgres=# SELECT * FROM ONLY posts;
 id | category_id | author | rate |  title
----+-------------+--------+------+---------
  3 |           3 | Boris  |    1 | Post #3
(1 row)
```

Все наши манипуляции работают, потому что мы проводили вставку в основную таблицу, используя правила, но, кроме этого, мы можем производить вставку напрямую в партицию, только должно соблюдаться условие записи в нее, иначе БД выдаст ошибку.

```
postgres=# INSERT INTO posts_1
postgres-# (id, category_id, title, author, rate)
postgres-# VALUES
postgres-# (1, 1, 'Post #10', 'Dimon', 100);
INSERT 0 1
postgres=#
postgres=#
postgres=# INSERT INTO posts_1
postgres-# (id, category_id, title, author, rate)
postgres-# VALUES
postgres-# (1, 4, 'Post #10', 'Dimon', 100);
ERROR:  new row for relation "posts_1" violates check constraint "posts_1_category_id_check"
DETAIL:  Failing row contains (1, 4, Dimon, 100, Post #10).
```

Также никто нам не запрещает использовать многострочную вставку. Мы можем вставлять несколько записей одновременно, и они сами распределятся с помощью правил нужной партиции. То есть мы можем вообще не заморачиваться, просто работать с нашей таблицей, как работали раньше. Приложение продолжит работать, но при этом данные будут попадать в партиции, все будет красиво разложено по полочкам без нашего участия.

```sql
INSERT INTO posts_1
(id, category_id, title, author, rate)
VALUES
(5, 1, 'Post #11', 'Irina', 1000),
(6, 2, 'Post #12', 'Valya', 1),
(7, 3, 'Post #21', 'Lena', 10);

```

Напомним, что данные мы можем выбирать как из партиции, так и из базовой таблицы.

```
postgres=# SELECT * FROM posts WHERE category_id =1;
 id | category_id | author | rate |  title
----+-------------+--------+------+----------
  1 |           1 | Ilon   |    1 | Post #1
  1 |           1 | Dimon  |  100 | Post #10
  5 |           1 | Irina  | 1000 | Post #11
(3 rows)

postgres=# SELECT * FROM posts_1;
 id | category_id | author | rate |  title
----+-------------+--------+------+----------
  1 |           1 | Ilon   |    1 | Post #1
  1 |           1 | Dimon  |  100 | Post #10
  5 |           1 | Irina  | 1000 | Post #11
(3 rows)
```

Если мы запустим эксплейнер запроса, то увидим, что в запросе через базовую таблицу seq scan проходит всю таблицу целиком, потому что данные могут все равно туда попадать, и будет скан по партиции. Если мы будем указывать условия нескольких категорий, он будет сканировать только те таблицы, на которые есть условия. Так работает оптимизатор — это правильно и так действительно быстрее.

```
1 EXPLAIN ANALYZE
2 SELECT * FROM posts WHERE category_id = 1;
3
4 ...
5    →  Seq Scan on posts  ...
6    →  Seq Scan on posts_1  ...
7
8 EXPLAIN ANALYZE SELECT * FROM posts_1;
9
10 ...
11   →  Seq Scan on posts_1   ...
```

Так же будут работать update и delete. Мы можем обновить основную таблицу, можем слать обновления напрямую в партиции. Так же будет работать и удаление. Нужно создать соответствующие правила, как мы создавали для добавления, только вместо insert написать update или delete.

В завершение поговорим об индексах. Индексы, созданные на основной таблице, не наследуются в дочерней таблице партиции. Это грустно, но придется дублировать все индексы, все ограничения, все триггеры на все таблицы. Есть утилиты для дублирования индексов, они open source, выбор конкретной зависит от того, какую платформу вы используете.

Заключение

Сегодня мы поговорили о партицировании данных. Рассмотрели виды партицирования и их особенности. Научились партицировать данные.

Напомним, что партицирование применяется на одном инстансе — это тот же самый инстанс базы, где лежала бы большая таблица, но мы раздробили ее на мелкие части. Мы можем не менять наше приложение — оно так же будет работать с основной таблицей: мы сможем вставлять туда данные, редактировать и удалять. Все работает так же, но в среднем в 3–4 раза быстрее.

Шардинг или горизонтальное партицирование — практически то же, что и вертикальное, только каждый шард находится на отдельном сервере.

Что можно почитать еще?

1. «Бизнес-процессы. Языки моделирования, методы, инструменты», Франк Шёнталер, Готфрид Фоссен, Андреас Обервайс, Томас Карле.
2. «Путь аналитика. Практическое руководство IT-специалиста», Андрей Перерва, Вера Иванова.