Лекция 1

Индексы
VACUUM
План запроса

Что уже известно?

- что такое БД, таблица
- ключи: первичные, внешние
- как писать запросы (SELECT, UPDATE, INSERT, DELETE)
- создавать, изменять, удалять таблицы (ещё попробуем на практике)

Что дальше?

- теперь нужно сделать так, чтобы запросы работали быстро
- СУБД просматривает таблицу строчка за строчкой
- если есть условие, то дополнительно тратятся мощности на сравнение значений
- можно ускорить запрос с помощью индексов

Примечание

- в лекциях и на практике будем использовать PostgreSQL
- у других СУБД могут быть некоторые отличия реализации, чего-то может не быть, а что-то наоборот

Популярность СУБД

364 systems in ranking, February 2021

Rank			DBMS	Database Model	Score		
Feb 2021	Jan 2021	Feb 2020			Feb 2021	Jan 2021	Feb 2020
1.	1.	1.	Oracle	Relational, Multi-model	1316.67	-6.26	-28.08
2.	2.	2.	MySQL	Relational, Multi-model	1243.37	-8.69	-24.28
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1022.93	-8.30	-70.81
4.	4.	4.	PostgreSQL	Relational, Multi-model	550.96	-1.27	+44.02
5.	5.	5.	MongoDB	Document, Multi-model	458.95	+1.73	+25.62
6.	6.	6.	IBM Db2	Relational, Multi-model	157.61	+0.44	-7.94
7.	7.	↑ 8.	Redis	Key-value, Multi-model	152.57	-2.44	+1.15
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	151.00	-0.25	-1.16
9.	9.	↑ 10.	SQLite	Relational	123.17	+1.28	-0.19
10.	10.	↑ 11.	Cassandra	Wide column	114.62	-3.46	-5.74

https://db-engines.com/en/ranking

Немного про индекс

- объект БД, используется для увеличения скорости поиска
- индекс не “бесплатен”, его нужно хранить и поддерживать в актуальном состоянии

Как создать индекс?

```
CREATE INDEX test1_id_index ON test1 (id);
```

- Название индекса может быть произвольным, но уникальным в рамках БД
- Удаление индекса

```
DROP INDEX test1_id_index;
```

Создание индекса

- создание индексов занимает время
- по умолчанию Postgres разрешает в это время делать запросы на выборку (SELECT), но блокирует операции записи (INSERT, UPDATE и DELETE)
- есть возможность разрешить запись параллельно с созданием индексов, при этом нужно учитывать ряд оговорок

После создания

после создания индекс нужно поддерживать

- это затраты на изменение, место для хранения и т.д.
- индексы, которые редко используются, должны быть удалены

Какие бывают индексы?

(применимо к PostgreSQL, но в других СУБД может быть похоже)

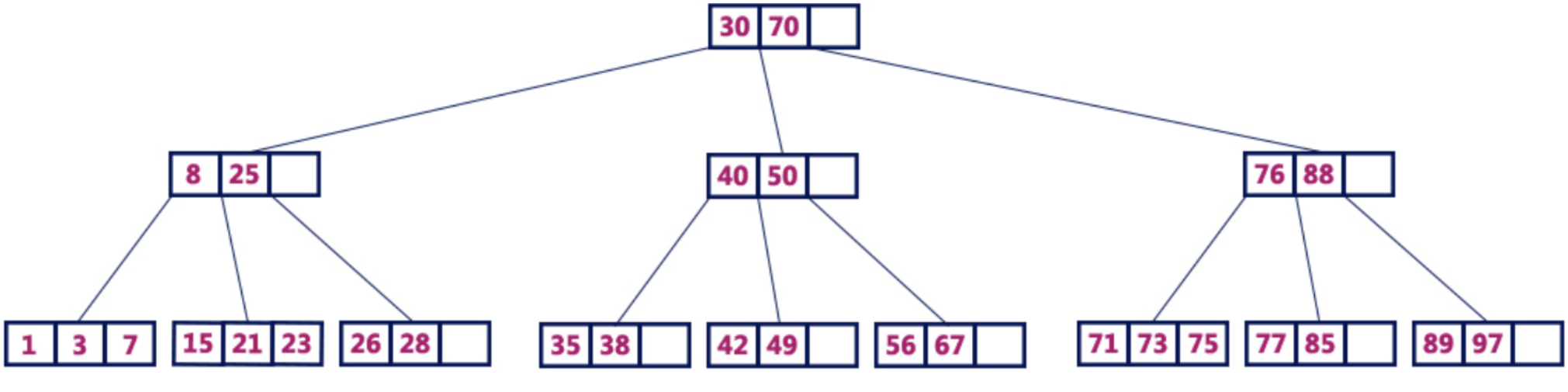
- B-дерево, хеш, GiST, SP-GiST, GIN (и не только)
- Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов.
- по умолчанию CREATE INDEX создаёт индексы типа B-дерево, эффективные в большинстве случаев

B-деревья

позволяют сравнивать данные, т.е. могут задействоваться при операциях <, <=, =, >=, >

- могут использоваться в операторах, которые подразумевают использование операторов сравнения: BETWEEN и IN
- могут использоваться при условиях IS NULL и IS NOT NULL

B-дерево 4 порядка



B-дерево 4 порядка содержит максимум 3 значения ключа и максимум 4 потомка для каждого узла.

Хеш-индексы

Хеш-индексы работают только с простыми условиями равенства, т.е. используется только с операцией =

Создать такой индекс можно следующей командой: CREATE INDEX имя ON таблица USING HASH (столбец);

Хеш-индекс

- -обновление hash-индекса очень быстрое в сравнении с btree (начиная с Postgres 10)
- сложность операции O(1) (быстрее, чем O(logN) у btree)
- лучшее использование: для уникальных полей в join-условиях
- не используется в условиях IS NULL и IS NOT NULL

Прочие: GiST, SP-GiST, GIN

- GiST: геометрические типы, сетевые адреса, диапазоны
- GIN: массивы, jsonb и tsvector
- также используются для полнотестового поиска

Индексы

- индекс на первичный ключ обычно создаётся автоматически

```
cs_server=# CREATE TABLE test (id int PRIMARY KEY, value int);
CREATE TABLE
cs_server=# \d test
                Table "public.test"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 value  | integer |           |          |
Indexes:
    "test_pkey" PRIMARY KEY, btree (id)
```

Индексы

- создавать индексы на внешние ключи - хорошая практика
- многие БД не требуют создание таких индексов, однако это может ускорить производительность
- SELECT * FROM table1 INNER JOIN table2 ON table1.id=table2.external_id;

• С чего начать?

- Первые кандидаты в индекс - это внешние ключи и поля из условия where

```
SELECT * FROM task
WHERE is_active=True AND task_status='ready'
  AND status_changing_timestamp<'2021-02-08
  12:00' AND creation_timestamp>'2021-01-01';
```

Индексы

- не стоит создавать индексы на все столбцы. В какой-то момент размер индекса может сравняться с размером данных
- удалять индекы, которые больше не используются

Когда полезен индекс?

- у столбца/столбцов достаточно уникальные значения (чем больше уникальных значений, тем лучше)
- нужно часто делать выборку по определённому дипазону значений

Когда индекс не очень полезен?

- выбирается довольно большой процент (>10-20%) строк таблицы
- нет дополнительного места под индекс
- нужно максимизировать производительность на insert (а не select)
- не нужно выбирать в качестве индекса поле, которое часто обновляется

Индекс от функции

- если в запросе используется функция от поля, нужно делать индекс по функции от поля, а не по самому полю
- например, у нас есть поле Integer, в котором хранится время в unix-timestamp
- Есть запрос:

  select data from task where to_timestamp(creation_timestamp) > '2021-01-01';
- to_timestamp() - функция, которая преобразует количество секунд с 01-01-1970 во вполне человеческий вид

Индекс от функции

- если висит индекс на creation_timestamp, это никак не повлияет на скорость выполнения запроса
- нужно делать индекс именно на to_timestamp(creation_timestamp).
- Какие ещё могут быть варианты функций? UPPER(name), разные функции, определённые пользователем
- SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');

  Result: 2001-01-01 00:00:00

Составной индекс

- можно создавать индекс на несколько столбцов
- Допустим есть таблица

  tel_book(id, name, age, phoneno)
- например, индекс можно сделать на столбцы (age, phoneno)
- И тогда запросы

  ‘WHERE age = xx AND phoneno = xxxxxxxxx’

  ‘WHERE age = xx’

  будут работать быстрее

Составной индекс

- при индексе (age, phoneno) запрос вида

  WHERE phoneno = xxxxxxxxx

  всё ещё будет перебирать все строки в таблице, потому что индекс сделан на (age, phoneno), а не (phoneno, age).

Составной индекс

- порядок столбцов в индексе важен, потому что он сначала упорядочивается по первому столбцу и уже внутри - по второму
- в большинстве случаев индекс по одному столбцу будет работать достаточно хорошо
- СУБД сама может объединять индексы при необходимости

Частичный индекс

- индекс строится только по подмножеству строк

  CREATE INDEX index_name ON table_name (column_name) WHERE condition;

- например, чаще выбираются только активные пользователи, тогда индекс можно сделать только на них

  CREATE INDEX active_users ON users (rating) WHERE is_active = true;

Обслуживание БД

(относится только к PostgreSQL)

- VACUUM
- VACUUM ANALYZE
- VACUUM FULL

Зачем что-то делать?

- при обычных операциях кортежи, удалённые или устаревшие в результате обновления, физически не удаляются из таблицы
- они сохраняются до тех пор, пока не будет выполнена команда vacuum
- существует pg_autovacuum - команды VACUUM и ANALYZE запускаются автоматически после достижения определённого предела изменений

Обычная очистка VACUUM

- VACUUM table_name; // VACUUM;
- работает быстро, но освобождает лишь часть места
- вычищает не только ненужные версии строк, но и ссылки на них из всех индексов
- в таблице просматриваются только те страницы, в которых происходила какая-то активность
- обновляется карта свободного пространства, чтобы отразить появившееся свободное места в страницах

VACUUM ANALYZE;

ANALYZE - служит для обновления информации о распределении данных в таблице

- сначала выполняется очистка, а затем анализ
- стоит использовать для таблицы после большого количества вставок/удалений

Полная очистка (vacuum full)

- если таблица или индекс выросли в размерах, то обычная очистка освободит место внутри существующих страниц, в них появятся "дыры", которые будут использованы для вставки новых строк
- число страниц не изменится, и с точки зрения операционной системы файлы будут занимать столько же места, сколько и занимали до очистки

Полная очистка (vacuum full)

Это плохо, потому что:

- замедляется полное сканирование таблицы (или индекса)
- файлы занимают лишнее место на диске и в резервных копиях
- в дереве индекса может появиться “лишний” уровень, который будет замедлять индексный доступ

Полная очистка (vacuum full)

Если доля полезной информации стала мала, можно выполнить полную очистку таблицы.

- индексы перестраиваются с нуля, а данные упаковываются максимально компактно
- сначала перестраиивается таблица, а затем каждый из её индексов
- для каждого объекта создаются новые файлы, а старые в конце удаляются
- в процессе требуется дополнительное место
- нельзя использовать часто, потому что полностью блокируется любая работа с таблицей (даже на select) на всё время очистки

Анализ плана запроса EXPLAIN (ANALYZE)

- в том или ином виде анализ плана запроса (команды EXPLAIN и ANALYZE) есть почти в каждой СУБД

- мы рассматриваем анализ плана запроса в СУБД PostgreSQL

Что такое план запроса?

- запрос поступает в СУБД, планировщик строит план запроса (на самом деле несколько, но в итоге выбирает самый быстрый), по этому плану выполняется запрос.

- EXPLAIN помогает понять, что и где можно улучшить

1. Создание таблицы

Создаётся таблица, заполняется случайными значениями

```
CREATE TABLE foo (c1 integer, c2 text);
INSERT INTO foo
SELECT i, md5(random()::text)
FROM generate_series(1, 1000000) AS i;
```

2. Первый запрос

```
EXPLAIN SELECT * FROM foo;
                        QUERY PLAN
```

- Seq Scan on foo (cost=0.00..18584.82
  rows=1025082 width=36)

- Seq Scan - таблица читается последовательно, блок за блоком
- Количество строк отличается от реально добавленных, но это вопрос сбора статистики

Значения

- Cost - это не время, а некое понятие для оценки затратность операции. За 1 единицу cost принимается время чтения одной страницы в 8Кб
- Первое значение - 0.00 — затраты на получение первой строки.
- Второе — 18334.00 — затраты на получение всех строк
- Rows - приблизительное количество строк
- Width - средний размер одной строки в байтах (она может меняться)

ANALYZE

- Для обновления статистики нужно запустить команду ANALYZE
- Для команды ANALYZE можно указывать отдельную таблицу, а можно не указывать.

```
ANALYZE foo;

EXPLAIN SELECT * FROM foo;

                QUERY PLAN

 Seq Scan on foo  (cost=0.00..18334.00
                     rows=1000000 width=37)
```

Что делает команда ANALYZE?

- читает случайные строки в таблице (300*default_statistics_target)
- default_statistics_target - поле в настройках PostgreSQL (по умолчанию 100)
- вычисляется разная статистическая информация
- процент NULL-значений, средний размер строки, количество уникальных значений и т.д.
- наиболее популярные значения и их частота

width

Seq Scan on foo (cost=0.00..18334.00

rows=1000000 width=37)

- width - средний размер (в байтах) строки в конечном наборе данных
- чем меньше, тем лучше
- средний width - это среднее значение суммы всех столбцов

EXPLAIN (ANALYZE)

- EXPLAIN не выполняет запрос, он только производит планирование
- А вот если запустить запрос как EXPLAIN (ANALYZE), то запрос реально выполнится и выведет дополнительную информацию
- реальное время выполнения в милисекундах (для первой строки и для всех строк)
- реальное количество строк
- количество циклов

EXPLAIN (ANALYZE)

EXPLAIN (ANALYZE) SELECT * FROM foo;

QUERY PLAN

- Seq Scan on foo

(cost=0.00..18334.00 rows=1000000 width=37)

(actual time=0.036..50.697 rows=1000000 loops=1)

Planning Time: 1.257 ms

Execution Time: 68.454 ms

EXPLAIN (ANALYZE)

- Чаще всего лучше наверняка знать, как выполняется запрос, чтобы узнать реальное время и реальное число строк.
- Нужно быть осторожным с EXPLAIN (ANALYZE) и командами модификации, потому что запрос реально выполняется и данные изменяются. Поэтому, чтобы данные оставались неизменными, их нужно оборачивать в транзакции.

Добавим условие WHERE

```
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
                        QUERY PLAN
 Seq Scan on foo  (cost=0.00..20834.00
                      rows=999579 width=37)
   Filter: (c1 > 500)
```

- появляется новая строчка Filter

EXPLAIN с WHERE

- стоимость выше
- меньше строк
- больше работы из-за фильтрации
- так как индекса нет, то СУБД приходится сканировать всю таблицу, и сравнивать столбец
- стоимость выше, так как появились дополнительные расходы на сравнение значения в столбце, это дольше, чем просто прочитать строку

Поможет ли индекс?

```
CREATE INDEX ON foo(c1);
EXPLAIN SELECT * FROM foo WHERE c1 > 500;
                QUERY PLAN
 Seq Scan on foo  (cost=0.00..20834.00
                     rows=999529 width=37)
Filter: (c1 > 500)
```

- Индекс не помог

Почему так получилось?

- Многим кажется, что индекс всегда помогает, если использовать его на поле в WHERE. Однако, это может быть и не так.
- В текущей таблице 1кк строк, а пропускаются условием лишь 500.
- СУБД "дешевле" прочитать всю таблицу и отфильтровать (использовать CPU), чем использовать индекс.

Запрос в реальном запуске

EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;

QUERY PLAN

Seq Scan on foo (cost=0.00..20834.00 rows=999502 width=37) (actual time=0.049..111.614 rows=999500 loops=1)

Filter: (c1 > 500)

Rows Removed by Filter: 500

Planning Time: 0.220 ms

Execution Time: 132.433 ms

Новые данные

Rows Removed by Filter: 500

- Новая строчка - Rows removed by Filter: 500
- (отображается только при ANALYZE)
- приходится прочитать 99.95% таблицы

Заставить использовать индекс

```
SET enable_seqscan TO off;
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
SET enable_seqscan TO on;
```

(Лучше никогда не отключать возможность последовательного сканирования, потому что в случаях, когда нет индекса, это единственная возможность добыть данные)

Принудительный индекс

Index Scan using foo_c1_idx on foo
(cost=0.42..36800.71 rows=999502 width=37)
(actual time=0.040..131.403 rows=999500
loops=1)

Index Cond: (c1 > 500)

Planning Time: 0.081 ms

Execution Time: 149.221 ms (а было 132ms)

Принудительный индекс

- работает, но медленно
- планировщик бы прав
- здесь индекс не даёт никакого выигрыша

Теперь поменяем условие

EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 < 500;

Index Scan using foo_c1_idx on foo (cost=0.42..25.12 rows=497 width=37) (actual time=0.016..0.088 rows=499 loops=1)

Index Cond: (c1 < 500)

Planning Time: 0.236 ms

Execution Time: 0.145 ms

Теперь выбирается индекс

Планировщик выбрал индекс - Index Scan

- новая строчка Index Cond

  Index Cond: (c1 < 500)
- планировщик выбирает сканирование по индексу, потому что читается малое количество строк (только 499 из 1кк)

Лекция окончена

- Вопросы?
- Для практического занятия нужно будет поставить PostgreSQL 12 (или любой другой версии)
- https://www.postgresql.org/download/
- Главное после установки запомнить пароль для пользователя postgres :)