Индексы

Индексы

Разработчики приложения

• SQL как язык — таблицы, представления, транзакции, ограничения, запросы, процедуры и функции, работа с данными

Администраторы СУБД

• Хранилище, бэкапы и восстановление, индексы, настройки, высокая доступность

Индексы должны быть заботой разрабочиков приложений!

Мониторинг индексов остается администраторам.

Использование индекса при поиске

Индекс может использоваться для:

- фильтрации – **WHERE** expr opr value
- сортировки – **ORDER BY** expr [ASC|DESC]
- knn – **ORDER BY** expr opr value [ASC]

Индексы

PostgreSQL поддерживает несколько типов индексов:

- **B-дерево,**
- **HASH**
- **GiST**
- **SP-GiST**
- **GIN**
- **BRIN**
- и расширение bloom.

CREATE INDEX

Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы запросов. По умолчанию команда CREATE INDEX создаёт индексы–B-деревья, эффективные в большинстве случаев.

Динамическое создание индекса

```
CREATE OR REPLACE FUNCTION
create_index(table_name text, column_name text,

index_name text)

RETURNS void AS $$

BEGIN

 EXECUTE 'CREATE INDEX ' || index_name || ' ON
' || table_name || '(' ||

column_name || ')';

END;

$$ LANGUAGE plpgsql;
```

B-tree

B-tree (сбалансированное дерево) — это самый распространенный тип индекса в PostgreSQL. Он поддерживает все стандартные операции сравнения (>, <, >=, <=, =, <>) и может использоваться с большинством типов данных. B-tree индексы могут быть использованы для сортировки, ограничений уникальности и поиска по диапазону значений.

```
CREATE INDEX ix_example_btree ON example_table (column_name);
```

Hash

Hash-индексы предназначены для обеспечения быстрого доступа к данным **только по равенству.**

В силу многих ограничений и того факта, что они лишь недавно стали поддерживать транзакции, хэш-индексы не так широко применяются в большинстве промышленных баз данных PostgreSQL. Однако, поскольку они оптимизируют поиск единственной записи, то могут быть великолепны в производственных средах, в которых в значительной степени требуется быстрый поиск отдельной записи.

CREATE INDEX имя **ON** таблица **USING HASH** (столбец);

GiST (Generalized Search Tree)

GiST-индексы являются обобщенными и многоцелевыми, предназначены для работы с сложными типами данных, такими как **геометрические объекты(растояние,пересечение площадей), текст** и **массивы.** Они позволяют быстро выполнять поиск по пространственным, текстовым и иерархическим данным.

CREATE INDEX ix_example_gist ON example_table USING **gist** column_name;

https://habr.com/ru/companies/postgrespro/articles/444742/

SP-GiST (Space-Partitioned Generalized Search Tree)

SP-GiST более гибкий чем gist индекс предназначены для работы с непересекающимися и неравномерно распределенными данными(научная сфера). Они эффективны для поиска в геометрических и IP-адресных данных.

Индексы SP-GiST, как и GiST, поддерживают поиск ближайших соседей. Для классов операторов SP-GiST, поддерживающих упорядочивание по расстоянию.

```
CREATE INDEX ix_example_spgist ON example_table
USING spgist (inet(column_name));
```

GIN (Generalized Inverted Index)

GIN-индексы применяются для полнотекстового поиска и поиска по массивам, **JSON** и триграммам. Они обеспечивают высокую производительность при поиске в больших объемах данных. Но может привести к замендлению производительность при записи в БД.

```
CREATE INDEX ix_example_gin ON example_table USING gin column_name;
```

```
-- расширение входит в состав PostgreSQL

CREATE EXTENSION pg_trgm;
```

Расширение pg_trgm. Данное расширение предназначено для поиска текстовых документов по триграммам

GIN (Generalized Inverted Index)

GIN-индексы применяются для полнотекстового поиска и поиска по массивам, **JSON** и триграммам. Они обеспечивают высокую производительность при поиске в больших объемах данных. Но может привести к замендлению производительность при записи в БД.

```
CREATE INDEX ix_example_gin ON example_table USING gin column_name;
```

```
-- расширение входит в состав PostgreSQL

CREATE EXTENSION pg_trgm;
```

Расширение pg_trgm. Данное расширение предназначено для поиска текстовых документов по триграммам

Примеры .

B-tree частичный индекс

При поиске по IP вас обычно интересуют внешние подключения, IP-диапазон внутренней сети компании можно не включать в индекс.

Задана таблица:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Создать частичный индекс:

```
CREATE INDEX access_log_client_ip_ix ON access_log
(client_ip)

WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

Запрос с попаданием в индекс.

```
SELECT *
  FROM access_log
 WHERE url = '/index.html' AND client_ip = inet
'212.78.10.32';
```

HASH

```
CREATE TABLE shorturl (
    id serial primary key,
    key text not null,
    url text not null
);
```

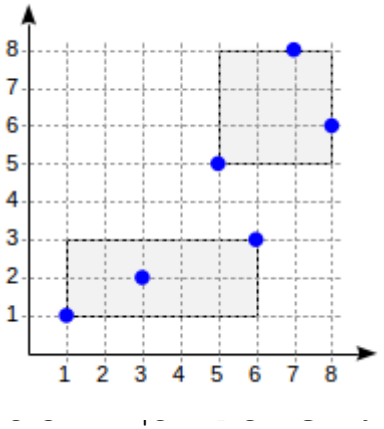
Хеш-индексы хранят 32-битный хеш-код, полученный из значения индексированного столбца, поэтому хеш-индексы работают только с простыми условиями равенства. Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором **=**. Создать такой индекс можно следующей командой:

```
CREATE INDEX shorturl_url_hash_index ON shorturl USING hash(url);
```

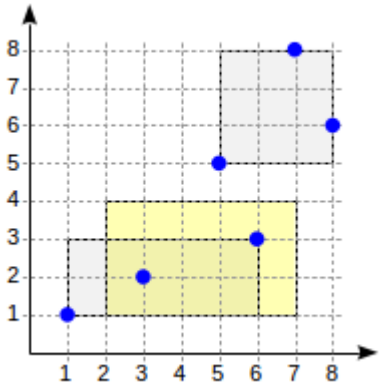
```
EXPLAIN (COSTS OFF) SELECT * FROM shorturl where url = 'https://www.supercool-url.com/756406'
```

Index Scan using shorturl_url_hash_index on shorturl

https://hakibenita.com/postgresql-hash-index



GiST




```
create table points(p point);
```

Добавим значений:

```
insert into points(p) values
  (point '(1,1)'), (point '(3,2)'), (point '(6,3)'),
  (point '(5,5)'), (point '(7,8)'), (point '(8,6)');
```

Создадим индекс:

```
create index on points using gist(p);
```

Посмотрим план выполнения поиска точек в заданном прямоугольнике:

```
explain(costs off) select * from points where p <@ box '(2,1),
(7,4)';
```

```
QUERY PLAN
-------------------------------------------
 Index Only Scan using points_p_idx on points
   Index Cond: (p <@ '(7,4),(2,1)'::box)
(2 rows)
```

Sp-GiST

Таблица точек:

```
create table points(p point);
```

Добавим значений:

```
insert into geo(p) values
   (POINT(583521.854408956, 4507077.862599085)),
   (POINT(583521.854408956, 4507077.862599085)),
   (POINT(583304.1823994748, 4506069.654048115)),
   (POINT(590250.10594797, 4518558.019924332)),
    ….............
 )
```

Создадим индекс:

```
create index pt_spgist_idx on geo using spgist(p);
CREATE INDEX;
```

Посмотрим план выполнения поиска точки:

```
explain (analyze on, buffers on) select p from geo
where p ~= '(590454.7399891173, 4519145.719617855)';
```

```
QUERY PLAN
-----------------------------------------------
Bitmap Heap Scan on pt_spgist_idx (cost=576.50..11992.42 rows=10227
width=16)
```

GIN (Generalized Inverted Index)

```
CREATE TABLE test_table (
 id SERIAL PRIMARY KEY,
 data JSON
);

INSERT INTO test_table (data) VALUES ('{"name":
"my_name", "age": 30}');

SELECT data->'name' as name, data->'age' as age FROM
test_table;


SELECT (data->>'name')::varchar as name, (data-
>>'age')::int as
age FROM test_table;
```

GIN

```
CREATE TABLE profiles_json (
 id serial primary key,
 emp_data JSONB
 );

INSERT INTO profiles_json (emp_data) VALUES ('{"exp": 10,
"name": "foo", "past_exp": ["company1", "company2", "company3"],
"languages": ["English", "French"]}'),
('{"exp": 20, "name": "bar", "past_exp": ["company1",
"company2", "company3", "company4", "company5"], "languages":
["English","Spanish", "Hindi", "Chinese"]}');

SELECT   id as profile_id,  emp_data->>'name' as name,
 (emp_data->>'exp')::INT as experience
 FROM profiles_json
 WHERE (emp_data->>'exp')::INT > 10;

CREATE INDEX profile_data_idx ON profiles_json USING
gin(emp_data);

set enable_seqscan to off; //мило данных
EXPLAIN ANALYZE SELECT * FROM profiles_json WHERE emp_data ?
'name';
Bitmap Index Scan on profile_data_idx
```

BTREE индексирование значений

```
CREATE INDEX profile_data_name_idx2 ON profiles_json ( (emp_
data->>'name') );
```

```
EXPLAIN ANALYZE SELECT * FROM profiles_json WHERE
emp_data ->>
'name' = 'foo';

QUERY PLAN
-----------------------------------------------------------
Index Scan using profile_data_name_idx2 on profiles_json
```

BRIN (Block Range INdex)

```
CREATE TABLE testtab
   (id int NOT NULL PRIMARY KEY,
  date TIMESTAMP NOT NULL,
 level INTEGER,
   msg TEXT);

INSERT INTO testtab (id, date, level, msg)
SELECT g, CURRENT_TIMESTAMP + ( g || 'minute' ) ::
interval, random() * 6, md5(g::text)
FROM generate_series(1,8000000) as g;
INSERT 0 8000000

create index testtab_date_brin_idx  on testtab using
brin (date);

explain analyze select * from public.testtab where date
between '2019-08-08 14:40:47.974791' and '2019-08-08
14:50:47.974791';
QUERY PLAN  Bitmap Index Scan on testtab_date_brin_idx
```

https://www.percona.com/blog/brin-index-for-postgresql-dont-forget-the-benefits/

Механизм индексирования

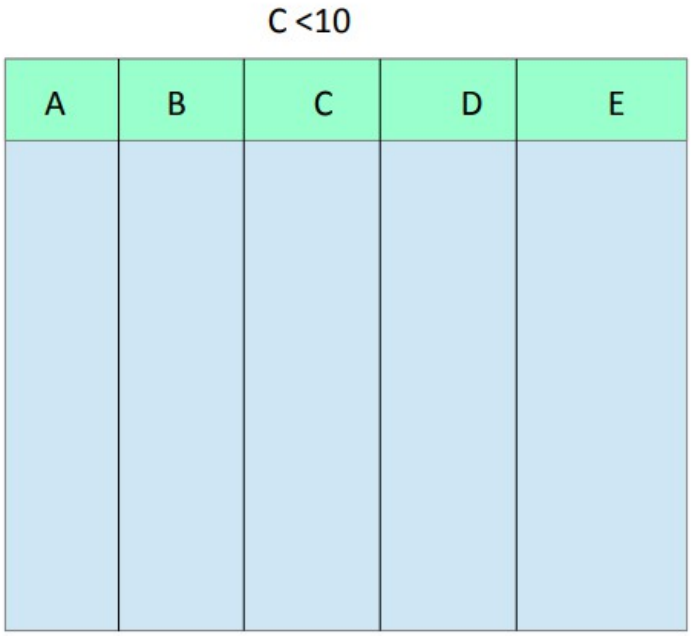
Механизм индексирования позволяет PostgreSQL одинаково работать с самыми разными методами доступа, учитывая их возможности.

Основные способы сканирования

- Seq Scan – Последовательное сканирование
- Index Scan – Индексное сканирование
- Bitmap Heap Scan – сканирование по битовой карте
- Index Only Scan – Покрывающие индексы

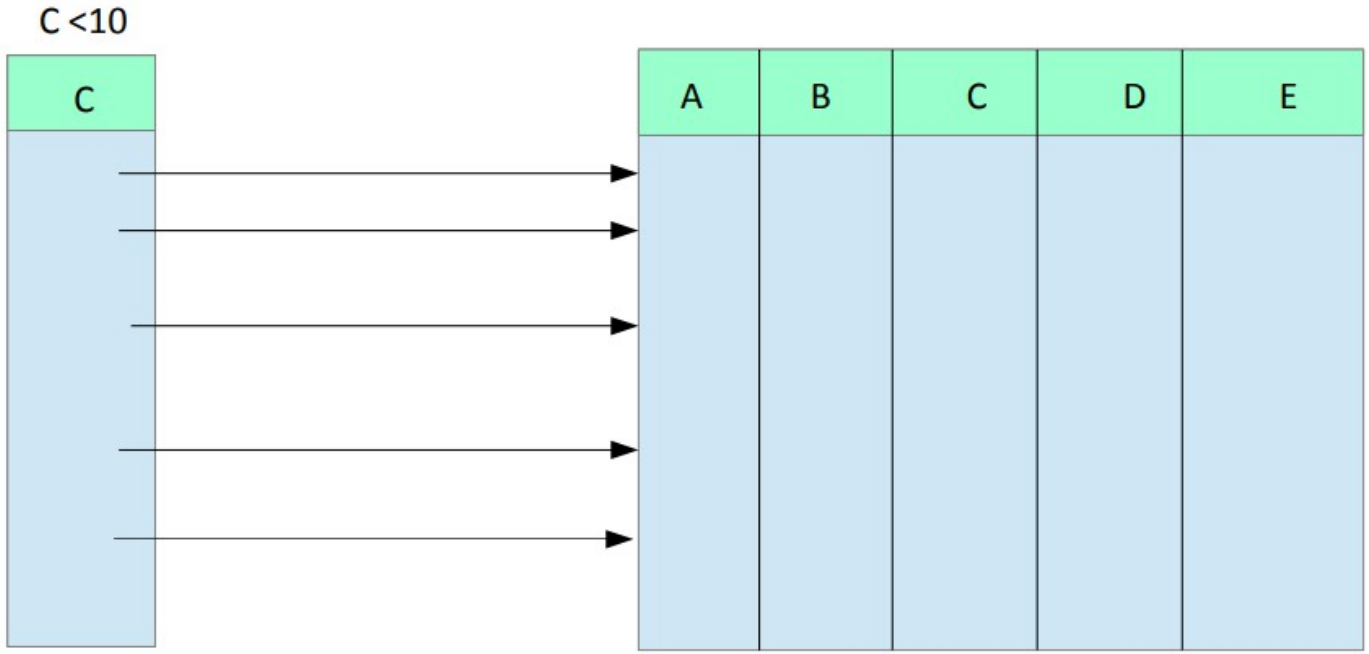
Sequential Scan

- Читаем последовательно таблицу и фильтруем записи по предикату



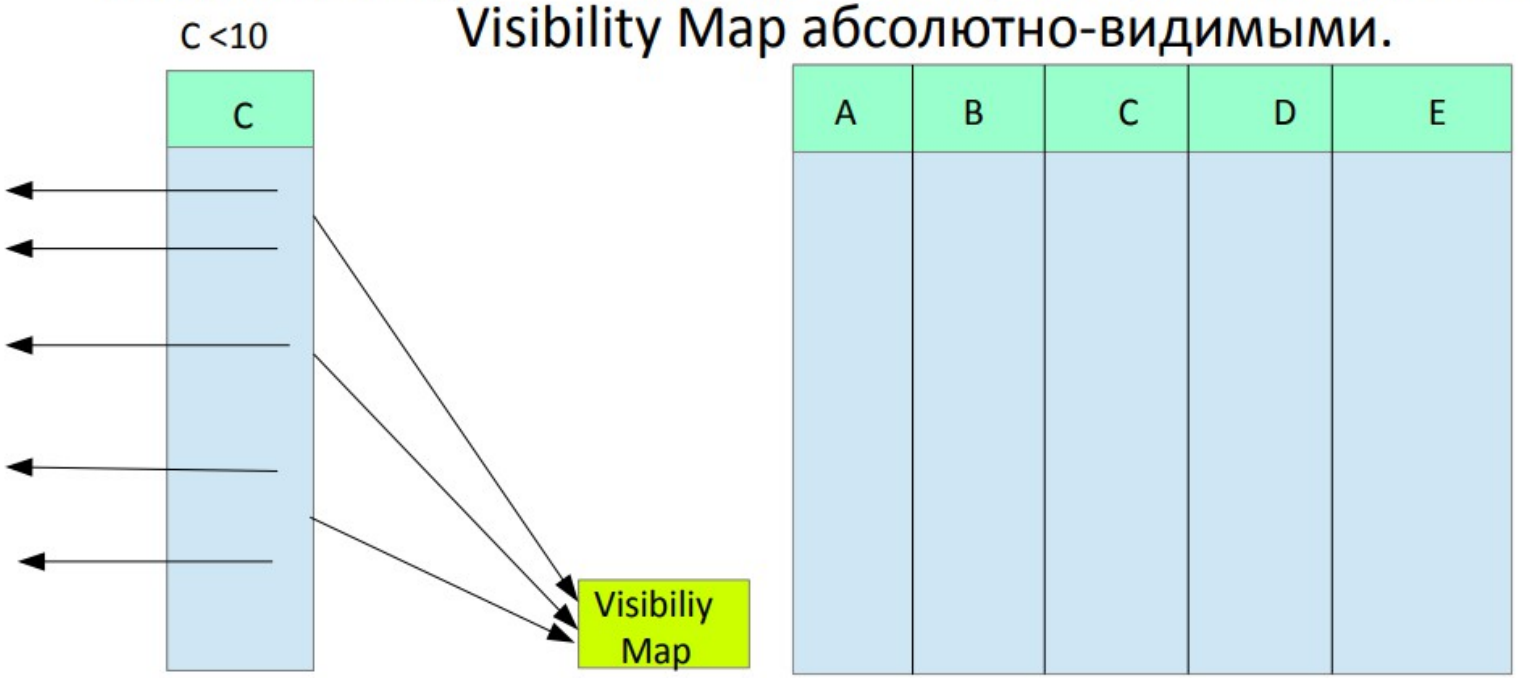
Простейший индекс (Index Scan)

- Читаем последовательно колонку, читаем таблицу (только нужные записи), выигрыш за счет меньшего размера колонки



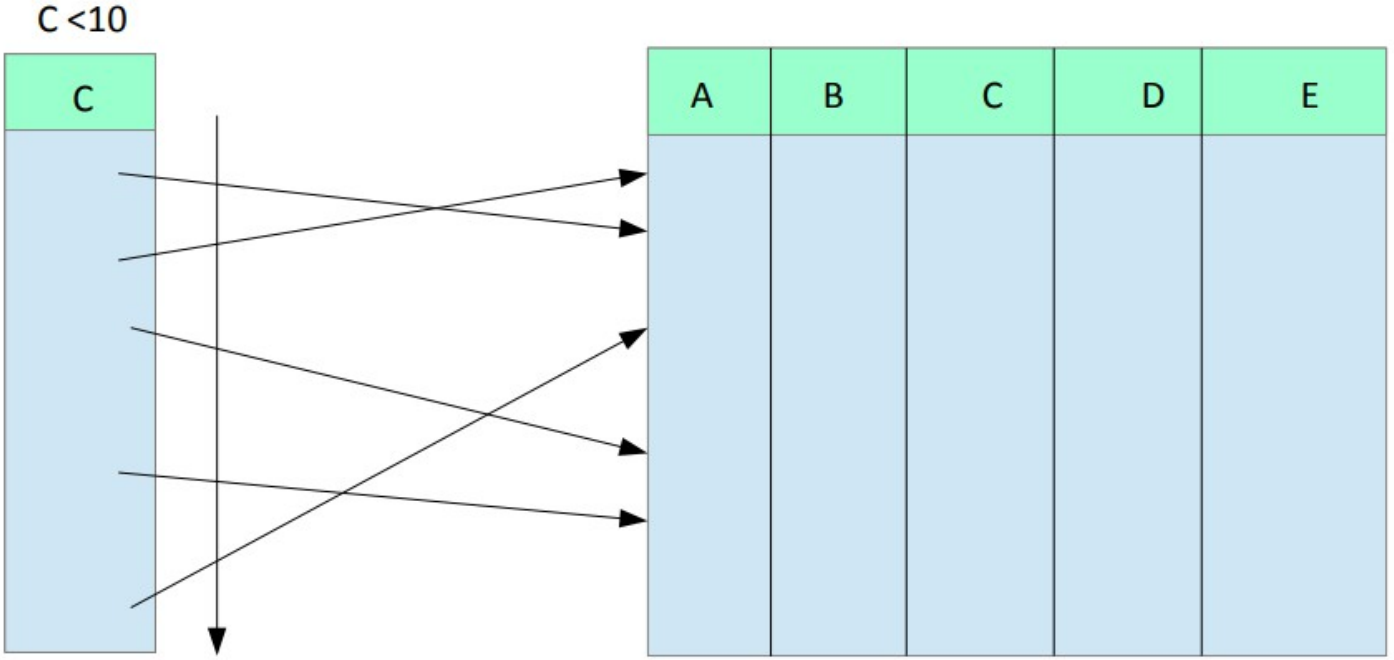
Простейший индекс (Index-only Scan)

- Читаем последовательно колонку, находим нужные значения и напрямик выдаем наружу, если страницы таблицы помечены в Visibility Map абсолютно-видимыми.



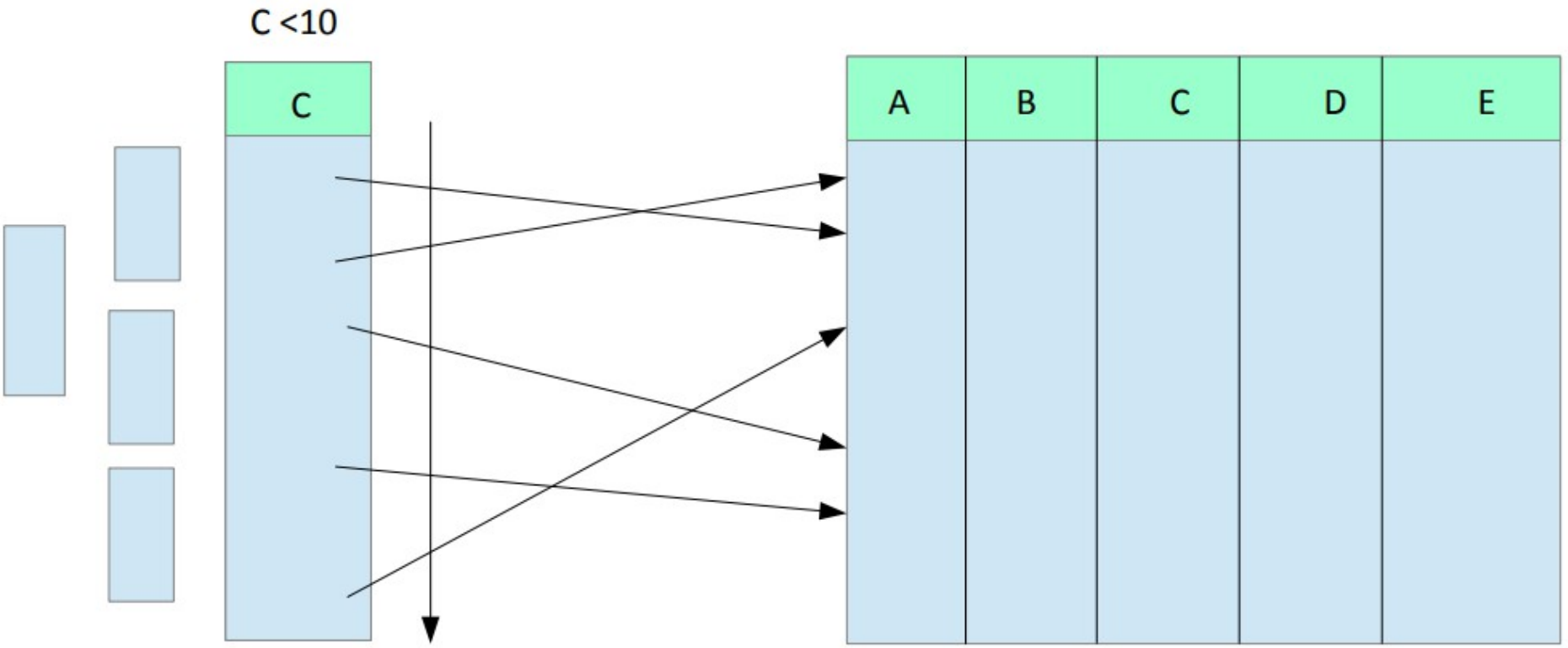
Простейший индекс++

- Упорядочиваем — получаем быстрый поиск, ускоряем ORDER BY, но случайное чтение таблицы.



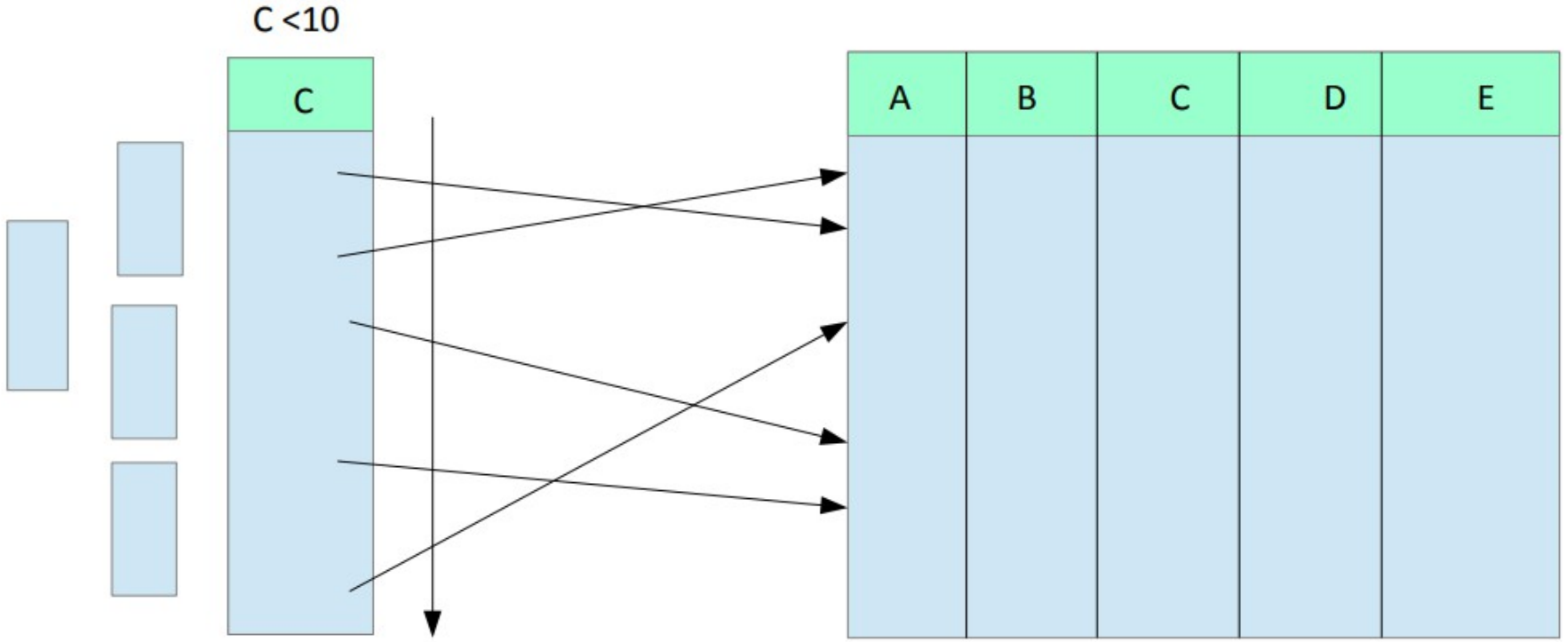
B-tree индекс

- Строим дерево — уменьшаем чтение индекса при быстром поиске



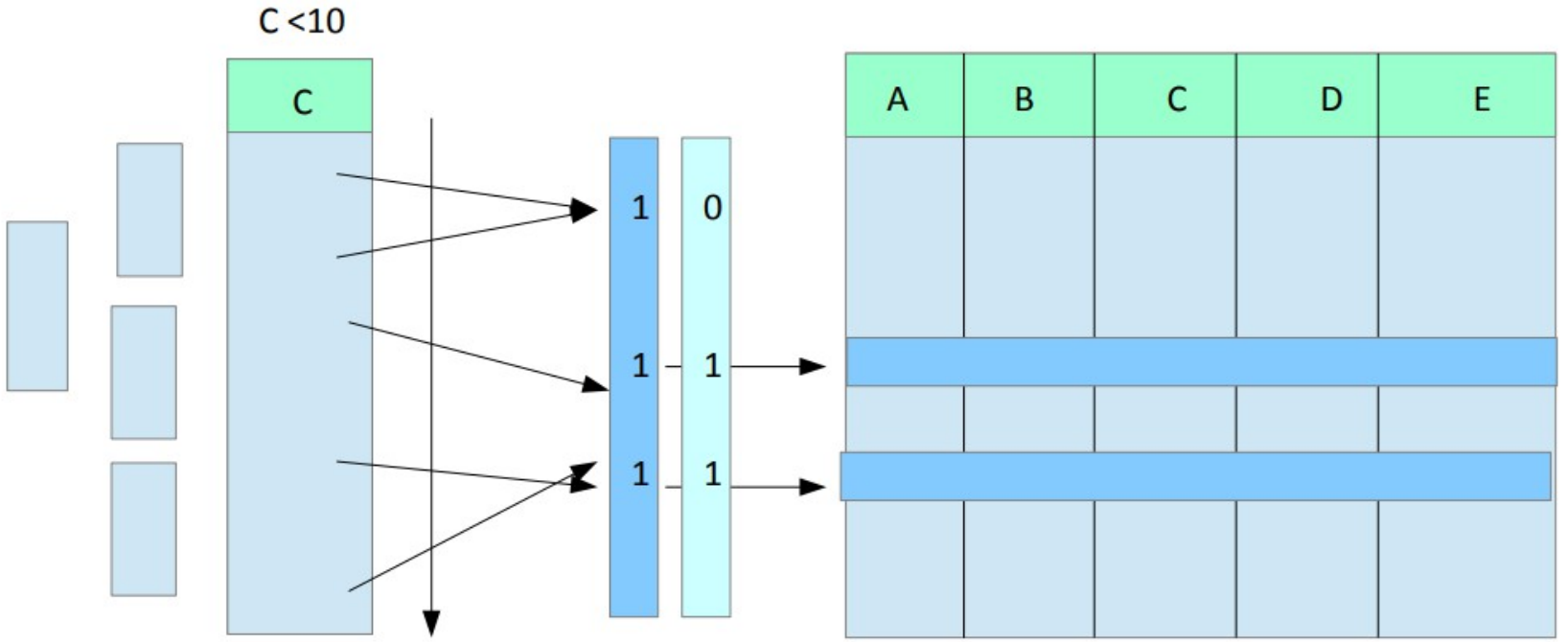
GiST,GIN,SP-GiST индексы

- Дерево — шаблон с API, поддержка произвольных типов данных



Bitmap index scan

- Результат Index scan сортируем, строим в памяти bitmap и читаем таблицу последовательно. Можно комбинировать индексы.



Управление использованием индексов

- Cost-based планер обычно умнее, но:
  - Иногда он ошибается и хочется разобраться в причинах
  - Хочется «пощупать» разные индексы
    - индексы долго строить
    - индексы реально используются
- Инструментарий
  - EXPLAIN, explain.depesz.com
  - Генераторы данных (generate_series())
  - contrib/pgbench

EXPLAIN

- Показать выбраный план выполнения запроса
- ANALYZE - Выполнить запрос и показать результирующий план
- COSTS ON|OFF, стоимости операций
- BUFFERS ON|OFF - прочитанные страницы
  - SHARED READ,HIT — с диска, из разделяемой памяти
- TIMING ON|OFF — времена выполнения

Syntax:
EXPLAIN [(option [, ...])] statement
EXPLAIN [ANALYZE] [VERBOSE] statement

where option can be one of:

ANALYZE [boolean]
VERBOSE [boolean]
COSTS [boolean]
BUFFERS [boolean]
TIMING [boolean]
FORMAT { TEXT | XML | JSON | YAML }

Азы EXPLIAN

EXPLAIN SELECT * FROM table1;

QUERY PLAN

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

https://postgrespro.ru/docs/postgresql/15/using-explain

Спасибо за внимание!