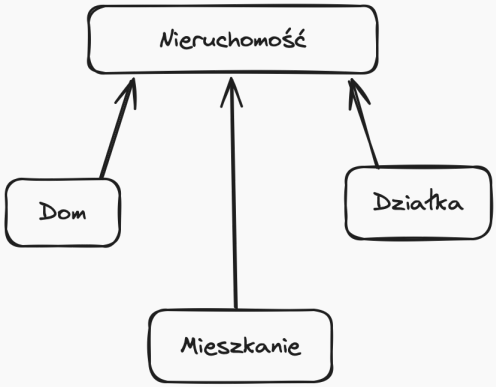
Wprowadzenie

O czym rozmawiamy dzisiaj?

- Wiele aspektów *projektowania baz danych* skupia się na zapewnieniu struktury, która:
  - z jednej strony dostarcza odpowiedniej siły ekspresji aby zapisać wszystkie informacje o modelowanych w bazie obiektach,
  - a z drugiej unika szkodliwych zjawisk, takich jak nadmiarowość (redundancja), które mogą prowadzić do powstania anomalii.
- Jednakże jest też ta „druga strona medalu", związana z wydajnością.
- Wpływ na wydajność możemy mieć na etapach:
  - projektowania struktury bazy,
  - projektowania samych zapytań,
  - stosowania dodatkowych mechanizmów, takich jak indeksowanie.

Obszary strojenia bazy

- Sam projekt bazy może wpływać na wydajność wykonywania zapytań, np. przy modelowaniu encji stanowiących uogólnienie/doszczegółowienie („dziedziczenie" z obiektowych języków programowania, generalizacja w UML).
- Dziś skupimy się jednak bardziej na optymalizacji działania bazy już istniejącej, o ustalonej strukturze.
- Nie da się zoptymalizować *samej struktury* bazy, jeżeli nie znamy scenariuszy jej użycia – wykonywanych zapytań.



Rysunek 1: Przykład encji stanowiącej generalizację innych

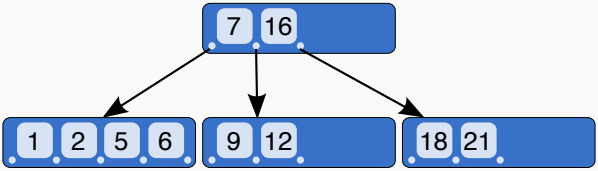
Podstawy indeksowania

Czym jest indeks?

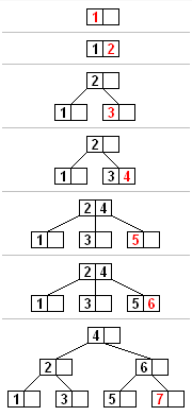
- Indeks jest *strukturą danych* przyspieszającą wyszukiwanie rekordów w tabelach.
- Indeks zawsze zdefiniowany jest dla określonych atrybutów, zwanych *kluczem indeksowania* (ang. *index key*) lub *kluczem wyszukiwania* (ang. *search key*).
- Pokrycie wartości:
  - indeks *gęsty* (ang. *dense*) zawiera wszystkie wartości występujące w kolumnach wchodzących w skład *klucza indeksowania*
  - indeks *rzadki* (ang. *sparse*) zawiera tylko niektóre wartości.

Przykład indeksu: B-drzewo

- Każdy węzeł może mieć maksymalnie n potomków (mówimy o drzewie *rzędu* n).
- Jest to drzewo *samorównoważące się*.



Rysunek 2: B-drzewo (źródło: Wikipedia)



Rysunek 3: Operacja wstawiania węzła

Dlaczego nie stworzyć mnóstwa indeksów?

- Jak widać, indeksy przyspieszają *wyszukiwanie*, ale wymagają *przebudowywania* przy prawie każdej operacji modyfikacji danych (`INSERT`, `UPDATE`, `DELETE`, ...).
- Tak jak wiele rzeczy w informatyce (i nie tylko), stosowanie indeksów jest więc kwestią pewnego *kompromisu*.
- Przy projektowaniu indeksów należy więc wziąć pod uwagę *charakter bazy* i *scenariusze jej użycia*:
  - W bazie obsługującej duży portal z wiadomościami operacje wyszukiwania treści artykułów lub komentarzy zdarzają się bardzo często (tysiące zapytań na sekundę), a operacje modyfikacji treści rzadziej – tu indeksowanie może być korzystne.
  - Ale w bazie zbierającej na bieżąco dane operacyjne z kas w supermarkecie, używanej potem zazwyczaj tylko do wygenerowania w nocy raportu dobowego, tworzenie indeksów może spowolnić operacje dodawania nowych rekordów, a potrzeby związane z wyszukiwaniem danych i tak nie są krytyczne czasowo.

Indeksy a SQL

Indeks tworzymy przy pomocy polecenia `CREATE TABLE`, podając co najmniej:

- unikalną nazwę indeksu,
- tabelę,
- atrybut lub atrybuty wchodzące w skład klucza indeksowania.

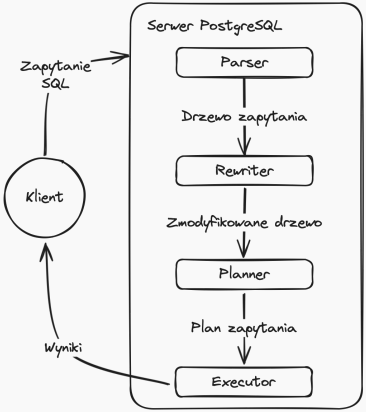
```
CREATE INDEX employees_surname_name_idx
  ON employees (surname, name);
```

Dodatkowo można określić inne parametry, takie jak porządek sortowania, zawężenie wartości czy kodowanie znaków – o tym później.

Przetwarzanie zapytań w PostgreSQL

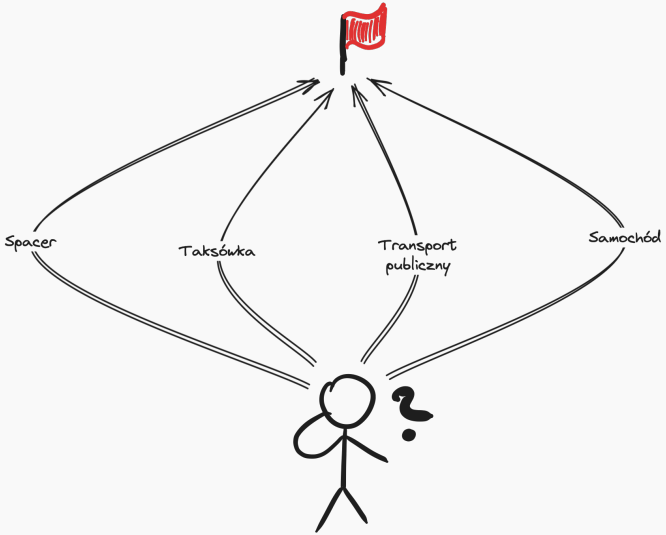
Droga zapytania w PostgreSQL

1. Klient nawiązuje połączenie z serwerem SZBD i przekazuje zapytanie SQL.
2. *Parser* sprawdza poprawność zapytania i buduje *drzewo zapytania* (ang. *query tree*).
3. System *rewrite* przepisuje zapytanie korzystając z reguł zapisanych w katalogu systemowym. Na tym etapie również zapytania do *widoków* są przetwarzane na zapytania do fizycznych tabel.
4. *Planner/optimizer* wyznacza wszystkie możliwe *ścieżki* wykonania zapytania, dla każdej szacując *koszt*. Ścieżka „najtańsza" przekazywana jest jako plan do wykonania.
5. *Executor* rekurencyjnie wykonuje wszystkie kroki planu, wykonując m.in. filtrowanie, złączanie i sortowanie danych.



Rysunek 4: Przetwarzanie zapytania w PostgreSQL

Problemy z planowaniem



Budowanie planu

- Jakie decyzje podejmuje planer?
  - Sposób wyszukiwania rekordów: skanowanie sekwencyjne vs. wykorzystanie indeksów
  - Wykorzystanie funkcji sortujących
  - Sposób wykonywania złączeń: *nested loop*, *merge join*, *hash join*
  - Kolejność wykonywania operacji
- W oparciu o jakie kryteria?
  - Istnienie odpowiednich indeksów w tabeli
  - Możliwość wykorzystania indeksów przy danym zapytaniu
  - Liczba rekordów do przejrzenia
  - Szacowana liczba rekordów w zbiorze wynikowym

Szacowanie kosztów operacji

Koszt operacji szacowany jest jako kombinacja liniowa liczby przeglądanych obiektów (stron dyskowych, wierszy) oraz *stałych szacowania*, z których najważniejsze to:

- `seq_page_cost` – szacowany koszt sekwencyjnego odczytu strony z dysku (domyślnie 1.0),
- `random_page_cost` – szacowany koszt niesekwencyjnego odczytu strony z dysku (domyślnie 4.0),
- `cpu_tuple_cost` – koszt przetworzenia wiersza (rekordu) przez procesor (domyślnie 0.01),
- `cpu_index_tuple_cost` – koszt przetworzenia wpisu w indeksie (domyślnie 0.05),
- `cpu_operator_cost` – koszt przetworzenia operatora lub funkcji (np. sprawdzenie, czy wiersz spełnia predykat `WHERE`; domyślnie 0.0025).

Inne stałe szacowania dotyczą m.in. przetwarzania rozproszonego (uruchamianie współbieżnych procesów), rozmiaru dostępnego buforu (*cache*), kompilacji JIT, itd.

Monitorowanie wykonania zapytań w PostgreSQL

Polecenie `EXPLAIN`

- Wyświetla plan wykonania zapytania, wraz z oszacowaniem kosztów każdego z jego etapów, liczby zwracanych wierszy i rozmiaru rekordu
- Użycie jest bardzo proste – wystarczy poprzedzić zapytanie słowem kluczowym `EXPLAIN`:

```
EXPLAIN SELECT * FROM employees;
```

- Przedstawiany jest ten plan, który został wybrany jako *optymalny* przez planer.

EXPLAIN – przykładowy wynik

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

- Zwracane dla każdego węzła planu oszacowania:
  - koszt rozruchu (ang. *start-up cost*) – czas potrzebny do rozpoczęcia zwracania wyników,
  - koszt całkowity (ang. *total cost*) – czas zakończenia przetwarzania i zwrócenia wszystkich wierszy,
  - liczba wierszy w zwracanych przez węzeł planu,
  - szerokość pojedynczego wiersza (w bajtach).
- Jednostki kosztu są arbitralne, lecz można je z grubsza przełożyć na liczbę stron odczytywanych z dysku (gdyż `seq_page_cost` to domyślnie 1.0).

EXPLAIN z warunkiem WHERE

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

- Koszt samego skanowania sekwencyjnego się nie zmienił
- Rzeczywista liczba zwracanych rekordów to 7000
- Różnica kosztu (483 zamiast 458) wynika z konieczności sprawdzania warunku dla każdego z 10000 rekordów (pojedyncze sprawdzenie „kosztuje” `cpu_operator_cost`, domyślnie 0.0025):

$$458 + 10000 \cdot 0.0025 = 458 + 25 = 483$$

- Zwróćmy uwagę że to zapytanie zwraca istotną część (70%) rekordów

EXPLAIN z bardziej restrykcyjnym warunkiem

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

- W tym wypadku pobieramy zaledwie ok. 1% rekordów, a więc planer uznał, że „opłaca się" skorzystać z indeksu
- Dwie fazy:
  - „Dolna" operacja wyszukuje rekordy spełniające warunek przy pomocy indeksu
  - „Górna" operacja fizycznie pobiera je z tabeli, ale korzystając z dostępu swobodnego, nie sekwencyjnego, który jest znacznie „droższy"

EXPLAIN, dwa warunki

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

- Koszt się nie obniżył, trzeba przejrzeć tę samą liczbę rekordów
- Sprawdzenie warunki dla `stringu1` podniosło go, ale bardzo nieznacznie

EXPLAIN, proste skanowanie indeksu

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

- Takie skanowanie zwraca wiersze w kolejności takiej jak określono w indeksie, ale koszt pobierania jest wyższy
- Często stosowane przy pobieraniu pojedynczych rekordów lub z klauzulą `ORDER BY`

EXPLAIN, sortowanie

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;

                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

- Sortowanie wykonywane jest jako osobny krok

EXPLAIN, sortowanie przy pomocy indeksu

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Limit  (cost=521.06..538.05 rows=100 width=244)
   ->  Incremental Sort  (cost=521.06..2220.95 rows=10000 width=244)
         Sort Key: four, ten
         Presorted Key: four
         ->  Index Scan using index_tenk1_on_four on tenk1  (cost=0.29..1510.08 rows=10000 width=244)
```

- Wyniki są już posortowane według pierwszego klucza (`four`), więc wystarczy sortowanie przyrostowe (wg. `ten`)

EXPLAIN, koniunkcja dwóch warunków

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

```
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

EXPLAIN, wpływ LIMIT

```
EXPLAIN SELECT * FROM tenk1
  WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

- Koszt operacji nadrzędnej jest niższy niż podrzędnej
- Operacji pobierania będzie niewiele, więc korzystanie z indeksu się opłaca

EXPLAIN ANALYZE

- Dodanie słowa kluczowego `ANALYZE` powoduje *wykonanie* zapytania oraz zebranie danych dotyczących *rzeczywistych* czasów wykonania:

  ```
  EXPLAIN ANALYZE SELECT * FROM employees;
  ```

- Przy każdym węźle planu zostanie podany czas wykonania (*actual time*)
- Czas rzeczywisty jest w milisekundach, a oszacowanie kosztu w arbitralnych jednostkach – a więc te wartości trudno ze sobą porównywać
- Najlepiej sprawdzać, czy szacowana liczba wierszy jest zbieżna z rzeczywistą

Problem planu optymalnego

- Nawet jeżeli indeks *mógłby* być wykorzystany do danego zapytania, planer może zdecydować, że skanowanie sekwencyjne będzie bardziej optymalne – na przykład przy małej liczbie rekordów w tabeli.
- Możemy wtedy mieć złudne wrażenie, że indeks nie jest odpowiedni, bo nie jest wykorzystywany w planie.
- Możemy wtedy „wymusić" użycie indeksu zmieniając jeden z parametrów konfiguracji planera:

```
SET enable_seqscan TO off;
```

- Nie da się całkowicie zabronić skanowania sekwencyjnego, ale ta zmiana sztucznie zawyża koszt takiej operacji aby „zachęcić" bazę do szukania alternatywnych dróg wykonania zapytania.

Rodzaje indeksów w PostgreSQL

B-drzewa

- Najpopularniejszy rodzaj indeksów „ogólnego przeznaczenia”
- Można stosować dla kolumn, których wartości posiadają relację porządkującą...
- ...czyli wtedy, gdy możemy je porównywać przy pomocy operatorów:
  `<` `<=` `=` `>=` `>`
- Może zwracać dane w postaci uporządkowanej

Hash

- Wyliczają i przechowują 32-bitowy skrót (ang. *hash*) wartości
- Funkcje haszujące nie utrzymują kolejności wartości, więc taki indeks może być wykorzystany tylko do zapytań korzystających z operatora =

```
CREATE INDEX name ON table USING hash (column);
```

GiST

- Jest to cała infrastruktura pozwalająca na implementację strategii indeksujących dla specyficznych typów danych
- Przykładowo, standardowo dostępna jest strategia dla danych geometrycznych 2D, z obsługą dla następujących operatorów:

```
<<    &<    &>    >>    <<|    &<|
|&>    |>>    @>    <@    ~=    &&
```

- Więcej informacji: dokumentacja

GIN

- Indeksy odwrócone, stosowne do indeksowania danych nieatomicznych, takich jak tablice (ang. *array*)
- Indeks odwrócony ma osobne wpisy dla wszystkich wartości elementarnych
- Przykładowo, standardowo dostępna jest strategia dla tablic, z obsługą dla operatorów: `<@` `@>` `=` `&&`
- Więcej informacji: dokumentacja

BRIN (Block Range INdexes)

- Przechowują podsumowania wartości zapisanych w kolejnych fizycznych blokach tabel
- Dobrze sprawdzają się wtedy, gdy wartości są dobrze skorelowane z fizycznym położeniem wierszy w tabeli
- Operują na zakresach wartości – na przykład dla typów o liniowym porządku sortowania, indeksowane dane to minimalna i maksymalna wartość w danym bloku
- Więcej informacji: dokumentacja

Indeksowanie w praktyce

Indeksy a dopasowanie wzorców

- Indeksy oparte o b-drzewa mogą być użyte również w zapytaniach wykorzystujących operatory dopasowania wzorców (`LIKE`, `~`), ale tylko wtedy gdy wzorzec jest stałą i jest zakotwiczony na początku ciągu:
  - `col LIKE 'foo%'`
  - `col ~ '^foo'`
  - ~~`col LIKE '%bar'`~~
- Mogą być także wykorzystane do operatorów *case-insensitive* (`ILIKE`, `~*`), ale tylko wtedy gdy wzorzec rozpoczyna się od znaków innych niż litery.
- Jeżeli baza korzysta z *locale* innego niż C, indeks musi zostać utworzony z odpowiednią klasą operatorów (o tym za chwilę).

Indeksy wielokolumnowe

Czasami zapytania dotyczą wartości więcej niż jednego atrybutu:

```
SELECT * FROM employees WHERE
  name = 'John' AND surname = 'Smith';
```

W takim przypadku przydatny może być indeks wielokolumnowy:

```
CREATE INDEX employees_surname_name_idx
  ON employees (surname, name);
```

Uwaga: W większości sytuacji indeksy jednokolumnowe są wystarczające i mogą być łączone w jednym zapytaniu; indeksy powyżej 3 kolumn praktycznie nie dają przyspieszenia.

Łączenie indeksów

- Indeks wielokolumnowy może być użyty tylko wtedy, gdy warunki dotyczące poszczególnych kolumn połączone są operatorem `AND`, np. dla indeksu na kolumnach `(x, y)` warunek musi mieć postać: `WHERE x = 3 AND y = 4`
- PostgreSQL może łączyć wiele indeksów (lub ten sam indeks kilka razy) dla warunków połączonych operatorem `AND` lub `OR`.
- W pamięci tworzone są *bitmapy*, które łączone są przy pomocy odpowiednich spójników logicznych, zgodnie ze strukturą klauzuli `WHERE`.

Indeksy a sortowanie

- Indeks oparty o B-drzewa może zwracać dane w porządku zgodnym z porządkiem klucza indeksowania.
- Dane mogą być pobrane w sposób sekwencyjny (szybciej) i posortowane w osobnym kroku, lub poprzez dostęp swobodny (wolniej) i nie wymagać już sortowania – ta decyzja należy do planera.
- Domyślnie dane z kolumn sortowane są rosnąco, z wartościami `NULL` na końcu.
- Porządek sortowania można określić przy tworzeniu indeksu:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Indeksy unikalne

- Indeks może wymagać, aby wartości w kolumnie (lub ich kombinacje w przypadku indeksów wielokolumnowych) były *unikalne*:

  ```
  CREATE UNIQUE INDEX ...
  ```

- Wartości `NULL` są traktowane jako różne, chyba że użyto opcji `NULLS NOT DISTINCT`.
- Indeks unikalny tworzony jest automatycznie dla kolumn określonych jako klucz główny.

Indeksy na wyrażeniach

- Dotychczas rozważaliśmy tylko indeksy, których klucz zawiera jedną lub więcej kolumn. Zamiast nich mogą jednak pojawić się dowolne *wyrażenia* SQL.
- Często stosuje się indeksy dla kolumn tekstowych przetworzonych funkcjami takimi jak np. `lower`:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

- Jeżeli wyrażenie to coś innego niż proste wywołanie funkcji, należy je ująć w nawiasy:

```
CREATE INDEX people_names ON people
  ((first_name || ' ' || last_name));
```

Indeksy częściowe

- Indeksy mogą dotyczyć tylko części rekordów – do polecenia `CREATE INDEX` można dodać klauzulę `WHERE` analogicznie jak np. w zapytaniach `SELECT`.
- Indeks taki może być wykorzystany tylko jeżeli klauzula `WHERE` zapytania jest analogiczna lub węższa od klauzuli użytej do budowy indeksu.
- Indeksy częściowe stosujemy przede wszystkim aby uniknąć indeksowania często występujących wartości.

```
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');
```

Klasy i rodziny operatorów

- Przy tworzeniu indeksu można określić dla każdej z kolumn *klasę operatorów*:

```
CREATE INDEX test_index ON test_table
  (col varchar_pattern_ops);
```

- Jeżeli baza korzysta z *locale* innego niż standardowe (C), wykorzystanie indeksów przy dopasowaniu wzorców w danych tekstowych wymaga zastosowania klasy `text_pattern_ops`, `varchar_pattern_ops` lub `bpchar_pattern_ops` (zależnie od typu kolumny).

Indeksy a porządek znaków

- Przy definiowaniu tabel, dla atrybutów można określić porządek znaków (ang. *collation*).
- Indeksy tworzone są automatycznie dla tak określonego porządku.
- W zapytaniach *collation* może jednak być inny, np.:

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

- Wtedy możemy dodać indeks dla takiego porządku:

```
CREATE INDEX test1c_content_y_index
  ON test1c (content COLLATE "y");
```

Inne zagadnienia optymalizacji

Sterowanie złączeniami

W przypadku złączeń wykonywanych np. tak:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

planer może wybrać dowolną kolejność ich wykonania, ale przy wielu tabelach wymaga to przeglądu dużej liczby kombinacji.

Możemy więc sterować złączeniem, tak aby wymusić (lub podpowiedzieć) odpowiednią kolejność ich wykonania – a co za tym idzie, skrócić sam *czas planowania*:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c
  WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Optymalizacja ładowania danych

Wskazówki optymalizacji ładowania danych do bazy:

- Wyłączenie automatycznego zatwierdzania
- Wykorzystanie polecenia `COPY`
- Tymczasowe usunięcie indeksów i/lub więzów klucza obcego
- Optymalizacja parametrów użycia pamięci i buforów
- Wykonanie polecenia `ANALYZE` po imporcie