Ćwiczenie 2.
Metody dostępu do danych

W niniejszym ćwiczeniu przyjrzymy się metodom dostępu do tabel i indeksów używanych w planach wykonywania zapytań.

Uwaga! Przed rozpoczęciem ćwiczenia zapoznaj się z informacjami dotyczącymi indeksów w bazach danych (prezentacja pt. „Optymalizacja poleceń SQL. Indeksy") oraz opisem metod dostępu do danych (prezentacja pt. „Optymalizacja poleceń SQL. Metody dostępu do danych").

Zebranie statystyk dla optymalizatora

Przed rozpoczęciem ćwiczenia zbierz statystyki dla relacji `OPT_PRACOWNICY`, `OPT_ZESPOLY` i `OPT_ETATY`. Jeśli nie masz tych relacji w swoim schemacie, utwórz je posługując się poleceniami z początku Ćwiczenia 1. Zastąp wartość `<nazwa_schematu>` nazwą Twojego schematu w bazie danych.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_PRACOWNICY');
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_ZESPOLY');
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_ETATY');
END;
```

Statystyki to informacje opisujące dane składowane w bazie danych oraz struktury służące do przechowywania danych. Obecność statystyk jest niezbędna do poprawnej pracy optymalizatora systemu zarządzania bazą danych. Uwaga! Nie myl statystyk dla optymalizatora ze statystykami wykonania przedstawionymi w Ćwiczeniu 1.

Zarządzanie statystykami omówimy w kolejnym ćwiczeniu z optymalizacji.

Metody dostępu do tabeli

Przyjrzymy się teraz operacjom, jakie są używane w planie wykonania zapytania do odczytania danych relacji.

1. Sprawdź, czy relacja `OPT_PRACOWNICY` została wyposażona w indeksy. W tym celu wykonaj poniższe zapytanie do słownika bazy danych.

   ```
   SELECT index_name, index_type
   FROM user_indexes
   WHERE table_name = 'OPT_PRACOWNICY';
   ```

   Zapytanie powinno zwrócić pusty wynik – oznacza to, że relacja `OPT_PRACOWNICY` nie posiada żadnych indeksów, które mogłyby służyć do wyszukiwania danych w relacji.

2. Napisz i wykonaj polecenie SQL, które wyświetli nazwisko i płacę pracownika o identyfikatorze (kolumna `ID_PRAC`) równym 10. Polecenie może mieć następującą postać:

   ```
   SELECT nazwisko, placa
   FROM opt_pracownicy WHERE id_prac = 10;
   ```

   Następnie wyświetl plan wykonania tego polecenia: włącz dyrektywę `AUTOTRACE` z opcją `ON` i ponownie wykonaj zapytanie (jako skrypt – klawisz F5). Pierwsza część raportu pokazuje wynik oraz plan wykonania polecenia.

   ```
   NAZWISKO                                           PLACA
   -------------------------------------------------- ----------
   Prac10                                                381

   Plan hash value: 1276241977

   ------------------------------------------------------------------------------------
   | Id  | Operation         | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
   ------------------------------------------------------------------------------------
   |   0 | SELECT STATEMENT  |                |     1 |    17 |    19   (0)| 00:00:01 |
   |*  1 |  TABLE ACCESS FULL| OPT_PRACOWNICY |     1 |    17 |    19   (0)| 00:00:01 |
   ------------------------------------------------------------------------------------

   Predicate Information (identified by operation id):
   ---------------------------------------------------

      1 - filter("ID_PRAC"=10)
   ```

   Do realizacji zapytania użyto metody dostępu do danych tabeli o nazwie „pełne przeglądnięcie tabeli" oznaczonej w planie wykonania jako `TABLE ACCESS FULL` z identyfikatorem równym 1. Ideą tej metody jest przeglądnięcie wszystkich bloków bazy danych, w których przechowywane są rekordy relacji `OPT_PRACOWNICY` (10 000 rekordów). Pobrane rekordy zostają poddane operacji filtrowania – odrzucone zostają rekordy, które nie spełniają predykatu z klauzuli `WHERE` polecenia (patrz sekcja *Predicate Information…*). Rekordy pozostałe po filtrowaniu trafiają do zbioru wynikowego polecenia `SELECT` (operacja w planie z identyfikatorem 0). Całkowity koszt wykonania zapytania wg zaprezentowanego planu to 19 jednostek (koszt przy operacji z identyfikatorem 0).

Możemy sprawdzić, ile bloków bazy danych odczytała operacja „pełne przeglądnięcie tabeli”. W tym celu analizujemy drugą część raportu przedstawiającą statystyki wykonania.

```
    Statistics
-----------------------------------------------------------
               1  CPU used by this session
               1  CPU used when call started
               2  DB time
               3  Requests to/from client
              58  consistent gets
              58  consistent gets from cache
              58  consistent gets from cache (fastpath)
               3  non-idle wait count
               2  opened cursors cumulative
               1  opened cursors current
               1  pinned cursors current
              67  session logical reads
               4  user calls
```

Tu interesują nas następujące pozycje (uwaga – nie wszystkie znajdują się w uzyskanych przez nas statystykach ponieważ dla czytelności pomijane są pozycje z wartością 0):

- `consistent gets` – liczba odczytanych z bufora bazy danych bloków z danymi, które są aktualne z punktu widzenia transakcji zawierającej polecenie (najczęściej dla polecenia `SELECT` bez klauzuli `FOR UPDATE`),
- `db block gets` – liczba odczytanych z bufora bazy danych bloków z danymi, które są aktualne (najczęściej dla poleceń `INSERT`, `UPDATE`, `DELETE` i `SELECT FOR UPDATE`),
- `physical reads` – liczba bloków, które zostały odczytane z dysków (dla poleceń `INSERT`, `UPDATE`, `DELETE`, `SELECT` i `SELECT FOR UPDATE`).

Zsumowane wartości `consistent gets` i `db block gets` mówią nam, ile w sumie bloków bazy danych zostało odczytanych z bufora w celu realizacji polecenia. Dodanie wartości `physical reads` da nam informację o całkowitej liczbie bloków bazy danych, jakie zostały odczytane do realizacji polecenia (z bufora i z dysku).

Ponieważ w naszym przypadku pokazywana jest jedynie wartość dla statystyki `consistent gets`, zatem `db block gets` i `physical reads` mają wartość 0. Na tej podstawie wnioskujemy, że operacja „pełne przeglądnięcie tabeli” odczytała z bufora bazy danych 58 bloków (domyślny rozmiar bloku to 8192 B). Oznacza to, że rekordy relacji `OPT_PRACOWNICY` są przechowywane w 58 blokach bazy danych. Dane zostały odczytane z bufora (a nie z dysku), gdyż widocznie tam trafiły na skutek wcześniejszego wykonania analizowanego zapytania (generalna zasada: SZBD szuka danych najpierw w buforze bazy danych, jeśli ich tam nie ma, odczytuje dane z dysku, a następnie umieszcza je w buforze).

3. Wyłącz dyrektywę `AUTOTRACE` (użyj opcji `OFF`). Wykonaj ponownie zapytanie z punktu 2., odczytując dodatkowo adres rekordu (pseudokolumna `ROWID`), który przechowuje dane pracownika o identyfikatorze równym 10.

```
SELECT nazwisko, placa, ROWID
FROM opt_pracownicy WHERE id_prac = 10;
```

Adres rekordu wskazuje dokładną lokalizację rekordu na nośniku fizycznym. Zawiera:

- numer obiektu bazodanowego, do którego należy rekord (relacja lub perspektywa),
- numer pliku, w którym składowany jest rekord,
- numer bloku bazodanowego pliku, w którym znajduje się rekord,
- pozycję rekordu w bloku.

Adres rekordu jest unikalny w całej bazie danych.

4. Zmodyfikuj zapytanie z punktu 2., tak aby wyszukiwało rekord opisujący pracownika z identyfikatorem 10 przy użyciu odczytanego wcześniej adresu rekordu. Polecenie powinno wyglądać podobnie do przedstawionego poniżej (użyj adresu rekordu, jaki otrzymałaś/eś w wyniku wykonania swojego zapytania).

```
SELECT nazwisko, placa
FROM opt_pracownicy WHERE ROWID = 'AABGclAAHAAAMdHAC7';
```

5. Włącz dyrektywę `AUTOTRACE` z opcją `ON` i wykonaj ponownie powyższe zapytanie. Plan wykonania powinien wyglądać tak, jak przedstawiono poniżej.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                |     1 |    25 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY USER ROWID| OPT_PRACOWNICY |     1 |    25 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Tym razem system nie musiał odczytywać wszystkich bloków, w których relacja `OPT_PRACOWNICY` przechowuje swoje rekordy – w zapytaniu wprost podaliśmy adres bloku zawierającego poszukiwany rekord. Operacja „dostęp do tabeli na podstawie adresu rekordu podanego przez użytkownika” (w planie `TABLE ACCESS BY USER ROWID`) odczytuje dokładnie jeden blok (z bufora bądź dysku) o adresie podanym wprost w poleceniu i pobiera z tego bloku dane wskazanego rekordu. Jest to najbardziej wydajna operacja dostępu do danych – dowodem tego jest jej koszt równy 1 (koszt minimalny).

Jeśli zanalizujemy statystyki wykonania to rzeczywiście widzimy, że system musiał odczytać tylko jeden blok.

```
   Statistics
----------------------------------------------------------
              3  Requests to/from client
              1  consistent gets
              1  consistent gets from cache
              1  consistent gets from cache (fastpath)
              3  non-idle wait count
              2  opened cursors cumulative
              1  opened cursors current
              1  pinned cursors current
              1  session logical reads
              4  user calls
```

Omawiana operacja nie jest często spotykana – bardzo rzadko użytkownik zna adres rekordu, który chce odczytać z relacji. Jednak warto o niej wspomnieć – w następnych punktach poznamy podobną operację, używaną znacznie częściej.

Metody dostępu do indeksów b-drzewo

Teraz będziemy analizować plany wykonania zapytań, w których używane są indeksy.

1. W zapytaniu z poprzedniej części szukaliśmy danych pracownika o wskazanym identyfikatorze. Spróbujemy teraz wpłynąć na poprawę efektywności wykonywania tego zapytania przez utworzenie odpowiedniego indeksu.

   Wyłącz dyrektywę `AUTOTRACE`. Następnie wykonaj polecenie, które utworzy nieunikalny indeks b-drzewo o nazwie `OP_ID_PRAC_IDX`. Klucz tego indeksu ma zawierać wartości kolumny `ID_PRAC` relacji `OPT_PRACOWNICY`. Polecenie tworzące indeks przedstawiono poniżej.

   ```
   CREATE INDEX op_id_prac_idx
      ON opt_pracownicy(id_prac);
   ```

   Następnie sprawdź w słowniku bazy danych informacje o utworzonym indeksie.

   ```
   SELECT index_name, index_type, uniqueness
   FROM user_indexes
   WHERE table_name = 'OPT_PRACOWNICY';

   SELECT column_name, column_position
   FROM user_ind_columns
   WHERE index_name = 'OP_ID_PRAC_IDX'
   ORDER BY column_position;
   ```

2. Wykonaj ponownie zapytanie, które szuka danych pracownika o identyfikatorze równym 10. Wyświetl plan wykonania zapytania i statystyki wykonania.

   W pierwszej kolejności zanalizujmy plan wykonania.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    17 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| OPT_PRACOWNICY |     1 |    17 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | OP_ID_PRAC_IDX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ID_PRAC"=10)
```

   Plan wykonania składa się teraz z trzech operacji. Jako pierwsza wykonywana jest operacja o nazwie „zakresowe przeglądnięcie indeksu” (w planie: `INDEX RANGE SCAN`) z identyfikatorem równym 2. Operacja ta, korzystając z indeksu `OP_ID_PRAC_IDX`, wyszukuje w indeksie wartości klucza równe 10, następnie zwraca adresy rekordów, jakie są związane ze znalezionymi w indeksie pozycjami. Operacja ta realizuje predykat `ID_PRAC=10` z klauzuli `WHERE` zapytania (patrz sekcja *Predicate Information…*). Zauważmy, że teraz predykat został oznaczony słowem *access* – oznacza to, że dane zostały wyszukane przy użyciu indeksu i nie wymagają dodatkowego filtrowania (w zapytaniu z poprzedniej części, które zostało zrealizowane przez pełne przeglądnięcie relacji, predykat był oznaczony słowem *filter* – uzyskane z relacji dane musiały zostać poddane filtrowaniu).

Zbiór („zakres”) adresów rekordów, odczytany z indeksu przez operację z identyfikatorem 2., zostaje przekazany do drugiej operacji o nazwie „dostęp do tabeli na podstawie adresów rekordów uzyskanych z indeksu” (w planie: `TABLE ACCESS BY INDEX ROWID`), oznaczonej identyfikatorem 1. Operacja ta odczytuje dane rekordów relacji, których adresy zostały jej przekazane jako parametr (jest to identyczna operacja jak poznana wcześniej operacja „dostęp do tabeli na podstawie adresu rekordu podanego przez użytkownika”, tylko że teraz adresy rekordów pochodzą z indeksu, a nie z samego tekstu zapytania).

Spójrzmy na koszt planu: wynosi on 2 jednostki (wartość w kolumnie *Cost* przy operacji z identyfikatorem 0). Koszt realizacji tego samego zapytania przy użyciu pełnego przeglądnięcia tabeli, a więc bez indeksu, wynosił 19 jednostek. Widzimy zatem dużą poprawę efektywności realizacji zapytania.

Spójrzmy teraz na statystyki wykonania analizowanego zapytania.

```
   Statistics
----------------------------------------------------------
              4  Requests to/from client
              3  consistent gets
              1  consistent gets - examination
              3  consistent gets from cache
              2  consistent gets from cache (fastpath)
              3  non-idle wait count
              2  opened cursors cumulative
              1  opened cursors current
              1  pinned cursors current
              3  session logical reads
              4  user calls
...
```

Przy realizacji zapytania wg planu z indeksem system musiał odczytać jedynie 3 bloki bazy danych. Można podejrzewać, że dwa bloki to odczyt indeksu a jeden to odczyt danych relacji (w planie widzimy, że w kolumnie *Rows* przy metodzie zakresowego przeglądnięcie indeksu jest liczba 1, co oznacza, że z indeksu odczytano jeden adres rekordu - stąd wniosek, że odczyt relacji sprowadził się do odczytu tylko jednego bloku).

3. Zauważmy, że kolumna `ID_PRAC` jest de facto kluczem głównym relacji `OPT_PRACOWNICY`. Spróbujmy wykorzystać ten fakt i jeszcze poprawić efektywność realizacji naszego zapytania.

   Wyłącz dyrektywę `AUTOTRACE`. Następnie usuń indeks `OP_ID_PRAC_IDX`.

   ```
   DROP INDEX op_id_prac_idx;
   ```

   Następnie zdefiniuj w relacji `OPT_PRACOWNICY` klucz główny o nazwie `OP_PK` dla kolumny `ID_PRAC`.

   ```
   ALTER TABLE opt_pracownicy ADD CONSTRAINT op_pk PRIMARY KEY(id_prac);
   ```

   Sprawdź w słowniku bazy danych, jakie teraz indeksy posiada relacja `OPT_PRACOWNICY`.

   ```
   SELECT index_name, index_type, uniqueness
   FROM user_indexes
   WHERE table_name = 'OPT_PRACOWNICY';
   ```

Widzimy, że został niejawnie utworzony indeks o nazwie równej nazwie zdefiniowanego klucza głównego. Jest to indeks unikalny o strukturze b-drzewa. Sprawdź, jakie kolumny relacji OPT_PRACOWNICY znajdują się w kluczu tego indeksu.

```
SELECT column_name, column_position
FROM user_ind_columns
WHERE index_name = 'OP_PK'
ORDER BY column_position;
```

Uwaga! Nie powinniśmy jawnie tworzyć indeksów unikalnych (mimo że jest dostępne polecenie CREATE UNIQUE INDEX), należy zawsze utworzyć ograniczenie (klucz główny lub unikalny), a to pociągnie za sobą zdefiniowanie odpowiednich indeksów.

4. Sprawdźmy teraz, jaki ma wpływ na realizację naszego zapytania obecność indeksu unikalnego. W tym celu włącz dyrektywę AUTOTRACE i wykonaj ponownie zapytanie o pracownika z identyfikatorem równym 10.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    17 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| OPT_PRACOWNICY |     1 |    17 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | OP_PK          |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ID_PRAC"=10)
```

Jak widzimy, plan uległ małej zmianie: operacja zakresowego przeglądnięcie indeksu została zastąpiona przez operację o nazwie „unikalne przeglądnięcie indeksu” (w planie: INDEX UNIQUE SCAN). Operacja ta pojawi się w planie, jeśli w zapytaniu użyto predykatu równościowego z kolumną, na której założono indeks unikalny.

5. Sprawdź, co się zmieni w planie, jeśli zmienimy warunek na nierównościowy, np. ID_PRAC < 10.

6. Utwórz kolejny indeks o nazwie OP_NAZWISKO_IDX. Ma on za zadanie wspomóc przeszukiwanie relacji OPT_PRACOWNICY wg nazwisk.

```
CREATE INDEX op_nazwisko_idx ON opt_pracownicy(nazwisko);
```

Ponownie zażądaj zebrania statystyk dla relacji OPT_PRACOWNICY.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_PRACOWNICY');
END;
```

Przy tworzeniu indeksu statystyki są dla niego gromadzone automatycznie. Jednak warto po utworzeniu indeksu zebrać na nowo statystyki dla poindeksowanej relacji – zostaną one uzupełnione o dokładniejsze statystyki dla kolumn w kluczu indeksu.

Następnie sprawdź wykorzystanie tego indeksu w planach wykonania następujących zapytań:

```
SELECT * FROM opt_pracownicy WHERE nazwisko = 'Prac155';
SELECT * FROM opt_pracownicy WHERE nazwisko LIKE 'Prac155%';
SELECT * FROM opt_pracownicy WHERE nazwisko LIKE '%Prac155%';
```

W którym zapytaniu indeks nie został użyty? Dlaczego?

7. Wyświetl plan kolejnego zapytania:

```
SELECT *
FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac155%' OR nazwisko LIKE 'Prac255%';
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |    20 |   720 |     8   (0)| 00:00:01 |
|   1 |  CONCATENATION                |               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY |    10 |   360 |     4   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           |OP_NAZWISKO_IDX|    10 |       |     2   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY |    10 |   360 |     4   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN           |OP_NAZWISKO_IDX|    10 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("NAZWISKO" LIKE 'Prac155%')
       filter("NAZWISKO" LIKE 'Prac155%')
   5 - access("NAZWISKO" LIKE 'Prac255%')
       filter("NAZWISKO" LIKE 'Prac255%' AND LNNVL("NAZWISKO" LIKE 'Prac155%'))
```

Plan wykonania nieco się skomplikował. SZBD „nie lubi” zapytań z warunkami połączonymi spójnikiem logicznym `OR` – użycie indeksu w takich zapytaniach nie jest możliwe. Tutaj jednak indeks został użyty: widzimy w planie operacje zakresowego przeglądnięcia indeksu `OP_NAZWISKO_IDX`. Jest to możliwe dzięki transformacji oryginalnego zapytania na zapytanie z operatorem `UNION ALL` w postaci:

```
SELECT * FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac155%'
UNION ALL
SELECT * FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac255%';
```

Mamy teraz de facto do czynienia z dwoma zapytaniami, warunki w tych zapytaniach mogą być obsłużone przez odpowiedni indeks. Jeśli spojrzymy na informacje o predykatach w planie, zobaczymy dodatkowy warunek `LNNVL("NAZWISKO" LIKE 'Prac155%')`, którego nie ma w oryginalnym zapytaniu. Służy on do eliminacji ew. duplikatów, które mogłyby się pojawić przy wykonaniu drugiego zapytania. Wyniki obu zapytań są łączone w jeden zbiór wynikowy przez operację `CONCATENATION` (z identyfikatorem równym 1).

8. Zanalizujmy plan jeszcze jednego zapytania:

```
SELECT * FROM opt_pracownicy
WHERE nazwisko IN ('Prac155','Prac255');
```

Plan wygląda tym razem następująco:

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     2 |    72 |     4   (0)| 00:00:01 |
|   1 |  INLIST ITERATOR              |               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY |     2 |    72 |     4   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           |OP_NAZWISKO_IDX|     2 |       |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("NAZWISKO"='Prac155' OR "NAZWISKO"='Prac255')
```

Nowa operacja o nazwie `INLIST ITERATOR` pojawia się w sytuacji, gdy w zapytaniu użyto operatora `IN`. Ta operacja może również się pojawić w zapytaniu z warunkami połączonymi operatorem `OR` (jest alternatywą dla operacji `CONCATENATION` – optymalizator wybierze tą operację, której zastosowanie będzie obarczone niższym kosztem). System, korzystając z indeksu dla kolumny `NAZWISKO` relacji `OPT_PRACOWNICY`, pobiera adresy rekordów spełniających warunek. Następnie na podstawie adresów odczytuje rekordy relacji `OPT_PRACOWNICY`. Te dwie operacje są powtarzane dla każdej wartości w zbiorze wartości przy operatorze `IN` w zapytaniu.

9. Zanalizujemy teraz plan wykonania zapytania z warunkami połączonymi spójnikiem logicznym `AND`. Zapytanie wygląda następująco:

```
SELECT nazwisko, placa, id_prac
FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac155%' AND placa > 1000;
```

Plan wykonania zapytania przedstawia się następująco:

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     5 |    85 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| OPT_PRACOWNICY |     5 |    85 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN            | OP_NAZWISKO_IDX|    10 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("PLACA">1000)
   2 - access("NAZWISKO" LIKE 'Prac155%')
       filter("NAZWISKO" LIKE 'Prac155%')
```

Jak widzimy, jest to znany nam już plan z operacją zakresowego przeglądnięcia indeksu. System pobiera z indeksu `OP_NAZWISKO_IDX` adresy rekordów, dla których wartość kolumny `NAZWISKO` spełnia predykat `NAZWISKO LIKE 'Prac155%'` (predykat oznaczony jako *access*, czyli realizowany z użyciem indeksu, dodatkowo jako *filter* gdyż dane, uzyskane z indeksu, muszą być dodatkowo odfiltrowane); jest to realizowane przez operację z identyfikatorem równym 2. Zauważmy, że optymalizator oszacował liczbę adresów rekordów, spełniających ten predykat, na 10. Następnie na podstawie uzyskanych adresów system odczytuje rekordy z relacji `OPT_PRACOWNICY`. Rekordy te są filtrowane przez predykat `PLACA > 1000` (predykat

oznaczony jako *filter*); optymalizator oszacował, że odfiltrowane zostanie 5 rekordów z 10. Te działania stanowią treść operacji o identyfikatorze 1.

Jeśli często wykonujemy takie zapytanie, korzystnie jest utworzyć tzw. indeks złożony, w kluczu którego znajdują się wartości obu kolumn z predykatów połączonych spójnikiem `AND`.

Na początku usuniemy istniejący indeks dla kolumny `NAZWISKO` relacji `OPT_PRACOWNICY`.

```
DROP INDEX op_nazwisko_idx;
```

Następnie tworzymy indeks złożony o nazwie `OP_NAZW_PLACA_IDX` relacji `OPT_PRACOWNICY`. Ważna jest kolejność kolumn w kluczu. Zwykle jako pierwsza w kluczu powinna zostać umieszczona kolumna, która jest częściej używana w zapytaniach. Założymy, że w naszym przypadku będzie to kolumna `NAZWISKO`. Druga kolumna w kluczu to kolumna `PLACA`.

```
CREATE INDEX op_nazw_placa_idx ON
   opt_pracownicy(nazwisko, placa);
```

Sprawdź teraz w słowniku bazy danych informacje o utworzonym indeksie.

```
SELECT index_name, table_name, index_type, uniqueness
FROM user_indexes
WHERE index_name = 'OP_NAZW_PLACA_IDX';

SELECT column_name, column_position
FROM user_ind_columns
WHERE index_name = 'OP_NAZW_PLACA_IDX'
ORDER BY column_position;
```

Ponownie zażądaj zebrania statystyk dla relacji `OPT_PRACOWNICY`.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_PRACOWNICY');
END;
```

10. Wyświetl plan zapytania z warunkami połączonymi spójnikiem `AND`.

```
SELECT nazwisko, placa, id_prac
FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac155%' AND placa > 1000;
```

Plan jest niemal identyczny jak poprzednio.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |    5 |    85 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY    |    5 |    85 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           |OP_NAZW_PLACA_IDX|    5 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("NAZWISKO" LIKE 'Prac155%' AND "PLACA">1000 AND "PLACA" IS NOT NULL)
       filter("NAZWISKO" LIKE 'Prac155%' AND "PLACA">1000)
```

Zauważmy jednak, że teraz oba warunki zostały zrealizowany w ramach operacji o identyfikatorze równym 2, czyli z użyciem indeksu (oznaczonej jako *access*). Przez to system musiał pobrać

z relacji mniej rekordów. W naszych zapytaniach zysk jest niewielki, ale w rzeczywistych zastosowaniach użycie indeksu złożonego może przynieść dużo korzyści.

11. Sprawdź, jak będą wyglądały plany zapytań, w których nie użyjemy wszystkich kolumn z klucza indeksu złożonego.

```
SELECT nazwisko, placa, id_prac
FROM opt_pracownicy WHERE nazwisko LIKE 'Prac155%';

SELECT nazwisko, placa, id_prac
FROM opt_pracownicy WHERE placa < 200;
```

Wniosek: indeks złożony będzie używany również w przypadku użycia w zapytaniu podzbioru kolumn z klucza indeksu, jednak muszą to być kolumny z części wiodącej klucza (z lewej strony). W drugim zapytaniu użyto warunku na kolumnie `PLACA`, która jest drugą kolumną w kluczu indeksu `OP_NAZW_PLACA_IDX`, a więc nie jest w części wiodącej. Stąd brak operacji z użyciem indeksu w planie wykonania dla tego zapytania.

12. Ciekawym przypadkiem jest odpowiedź na zapytanie z użyciem indeksu, ale bez dostępu do tabeli. Wyświetlmy plan poniższego zapytania.

```
SELECT nazwisko, placa
FROM opt_pracownicy
WHERE nazwisko LIKE 'Prac155%';
```

Wszystkie informacje, których użytkownik zażądał, znajdują się w indeksie `OP_NAZW_PLACA_IDX`.

```
-------------------------------------------------------------------------------------------
| Id  | Operation        | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                   |    10 |   130 |     2   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| OP_NAZW_PLACA_IDX |    10 |   130 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("NAZWISKO" LIKE 'Prac155%')
       filter("NAZWISKO" LIKE 'Prac155%')
```

System, wykorzystując operację zakresowego przeglądnięcia indeksu, odczytuje wartości kluczy indeksu `OP_NAZW_PLACA_IDX`, które spełniają predykat w klauzuli `WHERE` zapytania. W kluczu, oprócz wartości kolumny `NAZWISKO`, znajdują się również wartości kolumny `PLACA`. Nie ma potrzeby odczytu danych z rekordów relacji – wykonanie zapytania kończy się na dostępie do indeksu.

13. Co się stanie, jeśli odwrócimy sytuację z poprzedniego zapytania: wszystkie informacje, których żąda użytkownik, znajdują się w indeksie, ale predykat będzie się odwoływał do kolumny z części niewiodącej klucza? Znajdźmy plan poniższego zapytania.

```
SELECT nazwisko FROM opt_pracownicy WHERE placa < 1000;
```

Indeks nie powinien zostać użyty. Jednak w planie pojawia się dostęp do indeksu.

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                   |  4593 | 59709 |    11   (0)| 00:00:01 |
|*  1 |  INDEX FAST FULL SCAN| OP_NAZW_PLACA_IDX |  4593 | 59709 |    11   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("PLACA"<1000)
```

System nie może wykonać tego zapytania korzystając z operacji zakresowego przeglądnięcia indeksu. Zamiast tego korzysta z indeksu przy pomocy operacji analogicznej do pełnego przeglądnięcia tabeli, mianowicie operacji o nazwie „pełne szybkie przeglądnięcie indeksu” (w planie: `FAST FULL INDEX SCAN`). Ideą tej operacji jest odczyt wszystkich liści indeksu (w naszym przypadku indeksu `OP_NAZW_PLACA_IDX`) przy pomocy specjalnego rodzaju odczytu, mianowicie odczytu wieloblokowego. W liściach indeksu b-drzewo znajdują się wartości kluczy (w naszym przypadku wartości kolumn `NAZWISKO` i `PLACA` relacji `OPT_PRACOWNICY`) oraz adresy rekordów relacji `OPT_PRACOWNICY`, czyli wszystko, co jest potrzebne do wykonania naszego zapytania. System po odczycie wartości kluczy z liści indeksu dokonuje ich odfiltrowania przez predykat z klauzuli `WHERE` zapytania (zauważ, że predykat został oznaczony jako *filter* – został zrealizowany po odczycie danych z indeksu, jeśli byłby realizowany w trakcie odczytu indeksu, byłby oznaczony jako *access*). Do zbioru wynikowego trafiają wartości pozostałych po filtrowaniu kluczy indeksu (oczywiście adresy rekordów są pomijane i nie trafiają do zbioru wynikowego). Nie jest wykonywana żadna operacja dostępu do danych relacji. Właśnie w takich sytuacjach, gdy zapytanie można w pełni obsłużyć przez indeks, ale nie ma możliwości zastosowania zakresowego przeglądnięcia indeksu (czyli nawigacji po drzewie indeksu w celu wyszukania wartości pasujących kluczy), wykorzystywane jest szybkie pełne przeglądnięcie indeksu.

14. Jest jeszcze jedna operacja, która może zostać zastosowana w sytuacji braku możliwości użycia zakresowego przeglądnięcia indeksu. Utwórzmy złożony indeks b-drzewo, w kluczu którego znajdą się wartości kolumn `PLEC` i `PLACA` relacji `OPT_PRACOWNICY`.

```
CREATE INDEX op_plec_placa_idx ON opt_pracownicy (plec, placa);
```

Zauważmy, że pierwsza kolumna w klucz utworzonego indeksu ma jedynie dwie wartości: „K” i „M”.

Ponownie zażądaj zebrania statystyk dla relacji `OPT_PRACOWNICY`.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu>',
      tabname => 'OPT_PRACOWNICY');
END;
```

Następnie wyświetlmy plan poniższego zapytania.

```
SELECT /*+ INDEX_SS(p) */ plec, nazwisko FROM opt_pracownicy p
WHERE placa > 1500;
```

W zapytaniu, w klauzuli SELECT, pojawiła się tzw. wskazówka, która „zmusi" optymalizator do budowy planu z zastosowaniem operacji „przeglądnięcie indeksu z pominięciem kolumn" (w planie: INDEX SKIP SCAN). Wskazówkom dla optymalizatora przyjrzymy się w kolejnych częściach warsztatu, tu musieliśmy jej użyć aby optymalizator wyprodukował plan z omawianą operacją – dla naszego zapytania i naszych danych optymalizator nie użyłby omawianej operacji, koszt planu z jej użyciem jest większy niż koszt planu z pełnym przeglądnięciem tabeli.

Zanalizujmy plan wykonania polecenia.

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 | 2640 | 39600 |   541   (0)| 00:00:07 |
|   1 |  TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY    | 2640 | 39600 |   541   (0)| 00:00:07 |
|*  2 |   INDEX SKIP SCAN            |OP_PLEC_PLACA_IDX| 2640 |       |    35   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("PLACA">1500)
       filter("PLACA">1500)
```

W naszym zapytaniu nie możemy użyć metody zakresowego przeglądnięcia indeksu OP_PLEC_PLACA_IDX, ponieważ predykat w zapytaniu nie odwołuje się do części wiodącej klucza. Zamiast tego system przekształca oryginalny indeks OP_PLEC_PLACA_IDX, rozbijając go na zbiór nowych indeksów, o kluczach zawierających jedynie wartości kolumny PLACA (czyli pomijając wartości kolumny PLEC), liczba tych nowych indeksów jest równa liczbie różnych wartości kolumny PLEC z klucza indeksu OP_PLEC_PLACA_IDX. Jest to ideą operacji „przeglądnięcie indeksu z pominięciem kolumn". Uwaga! Nowe indeksy nie zostają trwale zapisane w bazie danych, służą do realizacji tylko tego zapytania i zaraz po jego zakończeniu są usuwane. Następnie system wykonuje zapytanie przeglądając po kolei utworzone indeksy i uzyskując z nich adresy rekordów. Powyżej omówione działania stanowią treść operacji z identyfikatorem równym 2 w planie.

Uzyskane z operacji przeglądnięcia indeksu z pominięciem kolumn adresy rekordów służą dalej operacji z identyfikatorem 1 do odczytu rekordów relacji OPT_PRACOWNICY (operacja dostępu do relacji z użyciem adresów rekordów).

Widzimy, że dla naszych danych koszt wykonania zapytania wg tego planu jest wielokrotnie wyższy od kosztu wykonania tego zapytania wg planu z pełnym przeglądnięciem tabeli (19 jednostek): dlatego „zmusiliśmy" optymalizator użyciem wskazówki do budowy takiego planu – działając samodzielnie optymalizator nie zaproponowałby nam planu z tak wysokim kosztem.

Dobrym kandydatem dla operacji przeglądnięcia indeksu z pominięciem kolumn jest indeks złożony, którego kolumna wiodąca posiada małą dziedzinę wartości.

15. Jeśli w zapytaniu konstruujemy predykat z kolumną, dla której utworzono indeks, starajmy się unikać stosowania wyrażeń na poindeksowanej kolumnie.

    Wyświetlmy plan poniższego zapytania.

```
SELECT placa, etat FROM opt_pracownicy
WHERE nazwisko = 'Prac155';
```

    Jak się zapewne spodziewałaś/eś, przy realizacji tego zapytania korzystne jest użycie indeksu `OP_NAZW_PLACA_IDX`.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    |Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    22 |     3  (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY   |     1 |    22 |     3  (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           |OP_NAZW_PLACA_IDX|     1 |       |     2  (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("NAZWISKO"='Prac155')
```

    Spróbuj ponownie wykonać zapytanie, tym razem jednak warunek ma być niezależny od wielkości liter, jakimi zapisane jest nazwisko szukanego pracownika.

```
SELECT placa, etat FROM opt_pracownicy
WHERE UPPER(nazwisko) = 'PRAC155';
```

    Zapytanie staje się bardziej elastyczne, jednak jego wykonanie będzie mniej efektywne (porównaj koszt poniższego planu z kosztem planu zapytania bez funkcji `UPPER`).

```
-------------------------------------------------------------------------------------
| Id  | Operation         | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                |   100 |  2200 |    19   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| OPT_PRACOWNICY |   100 |  2200 |    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(UPPER("NAZWISKO")='PRAC155')
```

    Jeśli koniecznie musimy użyć w zapytaniu wyrażenia, ratunkiem może okazać się indeks funkcyjny. W kluczu tego indeksu znajdują się wartości nie kolumn a wyrażeń, realizowanych na kolumnach relacji.

    Zdefiniuj indeks funkcyjny o nazwie `OP_FUN_IDX`, który będzie mógł być użyty przy wykonaniu powyższego zapytania.

```
CREATE INDEX op_fun_idx ON opt_pracownicy(UPPER(nazwisko));
```

Następnie ponownie wykonaj zapytanie z funkcją `UPPER` i wyświetl jego plan.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |   100 |  3600 |     9   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| OPT_PRACOWNICY |   100 |  3600 |     9   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | OP_FUN_IDX     |    40 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(UPPER("NAZWISKO")='PRAC155')
```

Jaka operacja została użyta przy dostępie do indeksu funkcyjnego? Jaki predykat został przez tą operację obsłużony?

16. Spróbujemy teraz zanalizować jeszcze jedną ciekawą operację z użyciem indeksów, tzw. połączenie indeksów. Wykonaj poniższe zapytanie i wyświetl jego plan.

```
SELECT nazwisko
FROM opt_pracownicy
WHERE id_prac < 500 AND nazwisko LIKE 'Prac15%';
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     6 |    78 |     4   (0)| 00:00:01 |
|*  1 |  VIEW              | index$_join$_001  |     6 |    78 |     4   (0)| 00:00:01 |
|*  2 |   HASH JOIN        |                   |       |       |            |          |
|*  3 |    INDEX RANGE SCAN| OP_NAZW_PLACA_IDX |     6 |    78 |     3  (34)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN| OP_PK             |     6 |    78 |     3  (34)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("ID_PRAC"<500 AND "NAZWISKO" LIKE 'Prac15%')
   2 - access(ROWID=ROWID)
   3 - access("NAZWISKO" LIKE 'Prac15%')
   4 - access("ID_PRAC"<500)
```

Idealnym indeksem do obsługi tego zapytania byłby indeks złożony z kluczem (`ID_PRAC`, `NAZWISKO`), jednak takiego indeksu w naszym schemacie nie ma. Zamiast tego mamy osobny indeks dla `ID_PRAC` (o nazwie `OP_PK`) i osobny dla `NAZWISKO` (o nazwie `OP_NAZW_PLACA_IDX`, indeks złożony z kolumną `PLACA` w części niewiodącej). System mógł użyć planu, w którym korzysta z jednego z wymienionych indeksów do obsługi jednego z warunków (`OP_PK` dla warunku `id_prac < 500` albo `OP_NAZW_PLACA_IDX` dla warunku `nazwisko LIKE 'Prac15%'`), następnie sięga do relacji `OPT_PRACOWNICY` na podstawie adresów rekordów, uzyskanych z indeksu, w ostatnim kroku filtrując dane relacji wg drugiego z warunków. Tymczasem optymalizator wybrał plan alternatywny: używa obu indeksów, wykonując na nich zakresowe przeglądnięcie, stąd uzyskuje dwa zbiory danych: zbiór pierwszy, będący wynikiem operacji o id = 4, w którym znajdują się adresy rekordów spełniających warunek `id_prac < 500`, oraz zbiór drugi, będący wynikiem operacji o id = 3, w którym znajdują się adresy rekordów spełniających warunek `nazwisko like 'Prac15%'`. Pamiętajmy, że wraz z adresami rekordów z indeksu operacje odczytują również wartości kluczy, czyli w zbiorze pierwszym są wartości kolumny `ID_PRAC`, a w zbiorze drugim wartości kolumn `NAZWISKO` i `PLACA`. W kolejnym kroku (id = 2) system wykonuje operację połączenia tych dwóch zbiorów, traktując je jak zbiory rekordów: warunkiem połączeniowym jest warunek równościowy porównujący adresy rekordów

z obu zbiorów (patrz predykat nr 2 w sekcji *Predicate Information*). W wyniku operacji połączenia otrzymujemy „rekordy” spełniające oba warunki. Do zbioru wynikowego system przesyła jedynie wartości z klucza z indeksu `OP_NAZW_PLACA_IDX`, a właściwie tylko jego część, mianowicie `NAZWISKO`. Nie ma konieczności dostępu do relacji `OPT_PRACOWNICY` zaadresowanej w poleceniu – wszystkie informacje udało się uzyskać z indeksów. Taką operację nazywamy *połączeniem indeksów* (ang. index join).

Metody dostępu do indeksów bitmapowych

Indeksy b-drzewo najlepiej zachowują się w sytuacji, gdy rozmiar dziedziny indeksowanej kolumny relacji jest duży. Teraz przyjrzymy się indeksom bitmapowym, których zastosowanie to indeksowanie kolumn o małej dziedzinie wartości.

1. Wyłącz dyrektywę `AUTOTRACE`. Wykonaj zapytania, które pokażą rozmiary dziedzin kolumn `PLEC` i `ETAT` relacji `OPT_PRACOWNICY`.

   ```
   SELECT DISTINCT plec FROM opt_pracownicy;

   SELECT DISTINCT etat FROM opt_pracownicy;
   ```

   Jak widzimy, rozmiary dziedzin tych kolumn są małe. Użycie do poindeksowania wymienionych powyżej kolumn indeksów b-drzewo nie byłoby właściwe. Dlatego użyjemy indeksów bitmapowych.

2. Wykonaj poniższe polecenia, które utworzą indeksy bitmapowe dla kolumn `PLEC` i `ETAT`.

   ```
   CREATE BITMAP INDEX op_etat_bmp_idx ON opt_pracownicy(etat);

   CREATE BITMAP INDEX op_plec_bmp_idx ON opt_pracownicy(plec);
   ```

   Następnie odczytaj ze słownika bazy danych informacje o utworzonych indeksach.

   ```
   SELECT index_name, table_name, index_type, uniqueness
   FROM user_indexes
   WHERE index_name IN ('OP_ETAT_BMP_IDX', 'OP_PLEC_BMP_IDX');

   SELECT index_name, column_name, column_position
   FROM user_ind_columns
   WHERE index_name IN ('OP_ETAT_BMP_IDX', 'OP_PLEC_BMP_IDX')
   ORDER BY index_name, column_position;
   ```

   Zwróć uwagę na wartość w kolumnie `INDEX_TYPE`.

   Zażądaj zebrania statystyk dla relacji `OPT_PRACOWNICY`.

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS(ownname => '<nazwa_schematu',
         tabname => 'OPT_PRACOWNICY');
   END;
   ```

3. Wykonaj zapytanie, które znajdzie liczbę kobiet wśród pracowników na etacie „DYREKTOR”. Zapytanie może wyglądać jak poniższe.

```
SELECT COUNT(*) FROM opt_pracownicy
WHERE plec = 'K' AND etat = 'DYREKTOR';
```

Następnie wyświetl plan wykonania tego zapytania.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     1 |    11 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |               |     1 |    11 |            |          |
|   2 |   BITMAP CONVERSION COUNT     |               |     9 |    99 |     2   (0)| 00:00:01 |
|   3 |    BITMAP AND                 |               |       |       |            |          |
|*  4 |     BITMAP INDEX SINGLE VALUE|OP_ETAT_BMP_IDX|       |       |            |          |
|*  5 |     BITMAP INDEX SINGLE VALUE|OP_PLEC_BMP_IDX|       |       |            |          |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("ETAT"='DYREKTOR')
   5 - access("PLEC"='K')
```

W planie wykonania pojawiają się dwie operacje `BITMAP INDEX SINGLE VALUE`. Pierwsza z operacji, z identyfikatorem równym 4, realizuje pobranie z indeksu bitmapowego założonego na kolumnie `ETAT` relacji `OPT_PRACOWNICY`, bitmapy dla wartości „DYREKTOR”. Druga z operacji, o identyfikatorze równym 5, realizuje analogiczne zadanie dla indeksu bitmapowego kolumny `PLEC` i wartości „K”. Następnie obie bitmapy zostają połączone w jedną bitmapę przy pomocy operacji `BITMAP AND` – „jedynki” w tej bitmapie odpowiadają rekordom relacji `OPT_PRACOWNICY` z wartościami „DYREKTOR” i „K” w kolumnach, odpowiednio, `ETAT` i `PLEC`. Kolejna operacja, `BITMAP CONVERSION COUNT` (identyfikator równy 2) liczy „jedynki” w tej bitmapie, wynik tego liczenia to wynik całego zapytania. Jak widzimy, plan nie korzysta z relacji `OPT_PRACOWNICY`. Dodatkowo koszt wykonania zapytania wg takiego planu jest niski (2 jednostki).

4. Zmodyfikujmy nieco nasze zapytanie. Dodajmy do niego nowy warunek ze spójnikiem `OR`.

```
SELECT COUNT(*) FROM opt_pracownicy
WHERE plec = 'K' AND (etat = 'DYREKTOR' OR etat = 'PROFESOR');
```

Pamiętamy, że indeksy b-drzewo nie „lubią” warunków z `OR`. Jak się to ma w przypadku indeksów bitmapowych? Zanalizujmy plan wykonania tego zapytania.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          |Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |    1 |    11 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE               |               |    1 |    11 |            |          |
|   2 |   BITMAP CONVERSION COUNT     |               |   41 |   451 |     3   (0)| 00:00:01 |
|   3 |    BITMAP AND                 |               |      |       |            |          |
|*  4 |     BITMAP INDEX SINGLE VALUE |OP_PLEC_BMP_IDX|      |       |            |          |
|   5 |     BITMAP OR                 |               |      |       |            |          |
|*  6 |      BITMAP INDEX SINGLE VALUE|OP_ETAT_BMP_IDX|      |       |            |          |
|*  7 |      BITMAP INDEX SINGLE VALUE|OP_ETAT_BMP_IDX|      |       |            |          |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("PLEC"='K')
   6 - access("ETAT"='DYREKTOR')
   7 - access("ETAT"='PROFESOR')
```

Jak widzimy, indeksy nadal są używane. Warunki połączone spójnikiem `OR` są realizowane w analogiczny sposób, jak te ze spójnikiem `AND`.

5. W indeksie bitmapowym znajdują się mapy bitowe zamiast adresów rekordów. Co się zatem stanie, jeśli w zapytaniu trzeba będzie sięgnąć do danych relacji? Zmodyfikujmy jedno z poprzednio wykonanych zapytań w następujący sposób:

```
SELECT nazwisko FROM opt_pracownicy
WHERE plec = 'K' AND etat = 'DYREKTOR';
```

i wyświetlmy jego plan wykonania.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |   455 |  9100 |    19   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID |OPT_PRACOWNICY |   455 |  9100 |    19   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS|               |       |       |            |          |
|   3 |    BITMAP AND                |               |       |       |            |          |
|*  4 |     BITMAP INDEX SINGLE VALUE|OP_ETAT_BMP_IDX|       |       |            |          |
|*  5 |     BITMAP INDEX SINGLE VALUE|OP_PLEC_BMP_IDX|       |       |            |          |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("ETAT"='DYREKTOR')
   5 - access("PLEC"='K')
```

Jak widzimy, po utworzeniu bitmapy, której „jedynki” odpowiadają rekordom opisującym kobiety na etacie „DYREKTOR” (operacje z identyfikatorami 4 i 5), zostaje zrealizowana operacja `BITMAP CONVERSION TO ROWIDS` (z identyfikatorem równym 2). Jej ideą jest przekształcenie „jedynek” w bitmapie na adresy rekordów w relacji `OPT_PRACOWNICY`. Zbiór adresów rekordów jest przekazany do znanej nam już operacji dostępu do tabeli na podstawie adresów rekordów uzyskanych z indeksu (z identyfikatorem 1), które znajduje rekordy relacji i odczytuje z nich wartości kolumny `NAZWISKO`.

6. Sprawdź, jak zachowuje się indeks bitmapowy w sytuacji poszukiwania wartości pustych. Zmień zapytanie w taki sposób, aby wyszukiwało pracowników będących kobietami z pustym etatem. Zanalizuj plan wykonania tego zapytania.