Ćwiczenie 4.
Połączenia, struktury dodatkowe

W niniejszym ćwiczeniu przyjrzymy się, w jaki sposób realizowane są operacje połączeń w poleceniach SQL. Poznamy również dodatkowe struktury, których zastosowanie może podnieść wydajność realizacji operacji w bazie danych.

Podstawy operacji połączenia

W niniejszej części ćwiczenia będziemy analizowali plany zapytań z jedną operacją połączenia.

Uwaga! Przed rozpoczęciem niniejszego ćwiczenia zapoznaj się z materiałami dotyczącymi połączeń (prezentacja pt. „Optymalizacja poleceń SQL. Połączenia").

1. Wykonaj zapytanie z operacją połączenia naturalnego relacji `OPT_PRACOWNICY` z relacją `OPT_ZESPOLY`. Następnie wyświetl plan wykonania dla tego zapytania.

```
SELECT * FROM opt_pracownicy JOIN opt_zespoly USING(id_zesp);
```

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                | 10000 |   527K|    22   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |                | 10000 |   527K|    22   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   361K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("OPT_PRACOWNICY"."ID_ZESP"="OPT_ZESPOLY"."ID_ZESP")
```

   Algorytm, który został użyty do realizacji operacji połączenia relacji nosi nazwę *Hash Join*. Realizację połączenia w planie wskazuje operacja `HASH JOIN`. W tym przypadku zbiorem zewnętrznym są rekordy relacji `OPT_ZESPOLY` (jest mniejsza – to jest główna przesłanka do wybrania jej przez optymalizator do tej roli), relacja `OPT_PRACOWNICY` pełni rolę zbioru wewnętrznego. W planie ten zbiór, który jest wyżej w poddrzewie operacji `HASH JOIN`, jest zbiorem (relacją) zewnętrzną. Zwróć uwagę na umieszczoną w sekcji predykatów informację o warunku połączeniowym.

   Spróbuj zamienić miejscami relacje w zapytaniu. Czy ma to wpływ na postać planu wykonania?

   Wskazówki. Jeśli chcesz, aby optymalizator użył w planie algorytmu *Hash Join*, użyj wskazówki `USE_HASH(relacja_1 relacja_2)`. Z kolei jeśli chcesz zabronić użycia tego algorytmu, użyj wskazówki `NO_USE_HASH(relacja_1 relacja_2)`.

2. Zmodyfikuj w zapytaniu typ połączenia na zewnętrzne prawostronne. Co zmieniło się w planie wykonania?

```
SELECT * FROM opt_pracownicy RIGHT JOIN opt_zespoly USING(id_zesp);
```

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                | 10000 |   527K|    22   (0)| 00:00:01 |
|*  1 |  HASH JOIN OUTER   |                | 10000 |   527K|    22   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   361K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("OPT_ZESPOLY"."ID_ZESP"="OPT_PRACOWNICY"."ID_ZESP"(+))
```

Zwróć uwagę na zmianę nazwy operacji połączenia w planie oraz na postać warunku połączeniowego w sekcji predykatów.

3. Uzupełnij schemat ćwiczebny o następujące elementy: (1) dodaj do relacji `OPT_ZESPOLY` klucz główny o nazwie `OZ_PK` dla kolumny `ID_ZESP`, (2) dodaj do relacji `OPT_PRACOWNICY` klucz obcy o nazwie `OP_FK_OZ_1` na kolumnie `ID_ZESP`, połączony z kolumną `ID_ZESP` relacji `OPT_ZESPOLY`, (3) utwórz indeks `OP_ID_ZESP_IDX` dla relacji `OPT_PRACOWNICY`, kolumny `ID_ZESP`.

```
ALTER TABLE opt_zespoly ADD CONSTRAINT oz_pk PRIMARY KEY(id_zesp);

ALTER TABLE opt_pracownicy ADD CONSTRAINT op_fk_oz_1
  FOREIGN KEY(id_zesp) REFERENCES opt_zespoly(id_zesp);

CREATE INDEX op_id_zesp_idx ON opt_pracownicy(id_zesp);
```

Następnie ponownie zbierz statystyki dla relacji `OPT_PRACOWNICY` i `OPT_ZESPOLY`.

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

4. Uzupełnij zapytanie z połączeniem o warunek filtrujący pracowników zespołu o nazwie „Bazy Danych”. Czy coś zmieniło się w planie wykonania polecenia?

```
SELECT * FROM opt_pracownicy JOIN opt_zespoly USING(id_zesp)
WHERE nazwa = 'Bazy danych';
```

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                |  2000 |   105K|    22   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |                |  2000 |   105K|    22   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| OPT_ZESPOLY    |     1 |    17 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   361K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("OPT_PRACOWNICY"."ID_ZESP"="OPT_ZESPOLY"."ID_ZESP")
   2 - filter("OPT_ZESPOLY"."NAZWA"='Bazy danych')
```

5. Spróbujemy teraz „nakłonić” optymalizator, aby przy realizacji połączenia użył indeksów, które zbudowaliśmy. W tym celu zmniejszymy zakładany przez optymalizator koszt użycia indeksu.

```
ALTER session SET optimizer_index_cost_adj = 20;
```

Powyższe polecenie ustawia koszt użycia indeksu do 20% kosztu rzeczywistego. Domyślna wartość parametru to `OPTIMIZER_INDEX_COST_ADJ` to 100, sterując tym parametrem możemy wpływać na „ochotę” optymalizatora do stosowania indeksów (oczywiście należy to czynić rozsądnie, w naszym przypadku ten zabieg służy do wyprodukowania planu z innym algorytmem realizacji operacji połączenia – dla danych o tak małym rozmiarze jak nasze system najczęściej użyje algorytmu *Hash Join*).

6. Wykonaj ponownie zapytanie z połączeniem i wyświetl jego plan wykonania.

```
SELECT * FROM opt_pracownicy JOIN opt_zespoly USING(id_zesp)
WHERE nazwa = 'Bazy danych';
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                | 2000 |   105K|    15   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                | 2000 |   105K|    15   (0)| 00:00:01 |
|   2 |   NESTED LOOPS               |                | 2000 |   105K|    15   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | OPT_ZESPOLY    |    1 |    17 |     3   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | OP_ID_ZESP_IDX | 2000 |       |     1   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| OPT_PRACOWNICY | 2000 | 74000 |    12   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   3 - filter("OPT_ZESPOLY"."NAZWA"='Bazy danych')
   4 - access("OPT_PRACOWNICY"."ID_ZESP"="OPT_ZESPOLY"."ID_ZESP")
```

Ustawiając zmniejszenie kosztu użycia indeksu „nakłoniliśmy” optymalizator do użycia algorytmu połączenia o nazwie *Nested Loops*. Algorytm ten staje się korzystny, jeśli optymalizator może użyć indeksu przy dostępie do jednej z łączonych relacji. Zanalizujmy otrzymany plan. Właściwą operację połączenia wykonuje operacja `NESTED LOOPS` o id=2. Biorą w niej udział dwa zbiory: zbiór zewnętrzny, zbudowany przez pełne przeglądnięcie relacji `OPT_ZESPOLY` (operacja z id=3, pojawia się zawsze jako pierwsza) z odfiltrowaniem rekordów wg predykatu `nazwa = 'Bazy danych'`, oraz zbiór wewnętrzny, utworzony przez zakresowe przeglądnięcie indeksu `OP_ID_ZESP_IDX` (operacja z id=4): dla bieżącego rekordu w zbiorze zewnętrznym wyszukiwane są wszystkie pasujące pod względem warunku połączeniowego rekordy zbioru wewnętrznego. Druga operacja `NESTED LOOPS` o id=1 wykonuje odczyt z relacji `OPT_PRACOWNICY` danych do uzupełnienia rekordów zbioru wynikowego na podstawie adresów rekordów.

Uwaga! Chociaż w planie operacja `NESTED LOOPS` pojawia się dwukrotnie, jest tu realizowana tylko jedna operacja połączenia.

Wskazówki. Jeśli chcesz, aby optymalizator użył w planie algorytmu *Nested Loops*, użyj wskazówki `USE_NL(relacja_1 relacja_2)`. Z kolei jeśli chcesz zabronić użycia tego algorytmu, użyj wskazówki `NO_USE_NL(relacja_1 relacja_2)`.

Przywróć standardowy koszt użycia indeksów.

```
ALTER session SET optimizer_index_cost_adj = 100;
```

7. Wykonaj poniższe zapytanie i wyświetl jego plan wykonania.

```
SELECT p1.nazwisko, p2.nazwisko
FROM opt_pracownicy p1 JOIN opt_pracownicy p2
ON p1.placa > p2.placa
WHERE p1.id_prac <> p2.id_prac AND p1.nazwisko LIKE 'Prac155%';
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows |Bytes| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 | 49371| 639K|    28   (8)| 00:00:01 |
|   1 |  MERGE JOIN                   |                 | 49371|1639K|    28   (8)| 00:00:01 |
|   2 |   SORT JOIN                   |                 |    10| 170 |     8  (20)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID|OPT_PRACOWNICY   |    10|  170|     7   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          |OP_NAZW_PLACA_IDX|    10|     |     2   (0)| 00:00:01 |
|*  5 |   FILTER                      |                 |      |     |            |          |
|*  6 |    SORT JOIN                  |                 | 10000| 166K|    20   (5)| 00:00:01 |
|   7 |     TABLE ACCESS FULL         |OPT_PRACOWNICY   | 10000| 166K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

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

   4 - access("P1"."NAZWISKO" LIKE 'Prac155%')
       filter("P1"."NAZWISKO" LIKE 'Prac155%')
   5 - filter("P1"."ID_PRAC"<>"P2"."ID_PRAC")
   6 - access(INTERNAL_FUNCTION("P1"."PLACA")>INTERNAL_FUNCTION("P2"."PLACA"))
       filter(INTERNAL_FUNCTION("P1"."PLACA")>INTERNAL_FUNCTION("P2"."PLACA"))
```

W poleceniu zażądaliśmy połączenia zwrotnego nierównościowego relacji `OPT_PRACOWNICY`. W tej sytuacji optymalizator zdecydował się na użycie algorytmu *Sort Merge*. Zanalizujmy plan wykonania. Połączenie wg algorytmu *Sort Merge* jest realizowane przez operację `MERGE JOIN` o id=1. Biorą w niej udział dwa zbiory rekordów. Pierwszy zbiór jest odczytywany w następujący sposób: (1) zostaje wykonane zakresowe przeglądnięcie indeksu `OP_NAZW_PLACA_IDX` (id=4) celem obsłużenia warunku `nazwisko LIKE 'Prac155%'`, (2) adresy rekordów, uzyskane z

indeksu, posłużyły do odczytu rekordów z relacji OPT_PRACOWNICY za pomocą operacji dostępu do relacji na podstawie adresów rekordów (id=3). Zbiór drugi powstaje przez pełne przeglądnięcie relacji OPT_PRACOWNICY. Oba zbiory zostają posortowane wg wartości kolumn w warunku połączeniowym: operacje SORT JOIN o id=2 (pierwszy zbiór) i id=6 (drugi zbiór). Następnie realizowane jest połączenie rekordów z obu zbiorów przez jednokrotny skan obu zbiorów (operacja o id=1).

Wskazówki. Jeśli chcesz, aby optymalizator użył w planie algorytmu *Sort Merge*, użyj wskazówki USE_MERGE(relacja_1 relacja_2). Z kolei jeśli chcesz zabronić użycia tego algorytmu, użyj wskazówki NO_USE_MERGE(relacja_1 relacja_2).

8. W celu usprawnienia realizacji operacji połączenia można utworzyć tzw. bitmapowy indeks połączeniowy. Indeks ten może zostać zdefiniowany jedynie dla operacji równościowego połączenia dwóch lub więcej relacji. W indeksie dla każdej wartości klucza indeksu z jednej z łączonych relacji składowane są adresy rekordów drugiej z łączonych relacji, które pasują ze względu na warunek połączeniowy.

   Utworzymy teraz indeks, który będzie wspomagał wykonywanie poniższego zapytania.

```
SELECT COUNT(*)
FROM opt_pracownicy JOIN opt_zespoly USING(id_zesp)
WHERE nazwa = 'Bazy danych';
```

   W kluczu indeksu będą wartości kolumny NAZWA relacji OPT_ZESPOLY. Polecenie tworzące indeks przedstawia się następująco:

```
CREATE BITMAP INDEX op_oz_join_idx ON opt_pracownicy(nazwa)
FROM opt_pracownicy p, opt_zespoly z WHERE p.id_zesp = z.id_zesp;
```

   Zwróćmy uwagę, że indeks jest definiowany dla relacji OPT_PRACOWNICY, mimo że poindeksowaną kolumna (NAZWA) znajduje się w relacji OPT_ZESPOLY. W definicji indeksu należy zdefiniować operację połączenia, które indeks będzie wspierał; definicja musi korzystać ze składni niejawnej połączenia.

   Wyświetl teraz plan zapytania, dla którego utworzyliśmy indeks.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |     3 |     1   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |                |     1 |     3 |            |          |
|   2 |   BITMAP CONVERSION COUNT    |                |  2000 |  6000 |     1   (0)| 00:00:01 |
|*  3 |    BITMAP INDEX SINGLE VALUE| OP_OZ_JOIN_IDX |       |       |            |          |
-----------------------------------------------------------------------------------------------

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

   3 - access("OPT_PRACOWNICY"."SYS_NC00010$"='Bazy danych')
```

   Otrzymaliśmy typowy plan dla zapytania z użyciem indeksu bitmapowego. Operacja z id=3 odczytuje z indeksu mapę bitową, wskazującą lokalizację rekordów relacji OPT_ZESPOLY, w których wartość kolumny NAZWA to „Bazy danych”, wraz z lokalizacją rekordów relacji OPT_PRACOWNICY, które łączą się z rekordami relacji OPT_ZESPOLY na podstawie warunku połączeniowego. Operacja z id=2 zlicza „jedynki” w bitmapie, otrzymana liczba stanowi

odpowiedź na zapytania. Jak widzimy, w planie nie ma realizacji operacji połączenia a koszt wykonania zapytania wg tego planu znacznie spadł w porównaniu z planem bez użycia tego indeksu.

9. Usuń utworzony wcześniej bitmapowy indeks połączeniowy

```
DROP INDEX op_oz_join_idx;
```

Połączenia wielu relacji

Do tej pory obserwowaliśmy plany zapytań z jedną operacją połączenia. Przyjrzymy się teraz poleceniom z kilkoma operacjami połączenia.

1. Wykonaj i wyświetl plan poniższego zapytania.

```
SELECT p.nazwisko, z.nazwa, e.placa_min, e.placa_max
FROM opt_pracownicy p JOIN opt_zespoly z USING(id_zesp)
     JOIN opt_etaty e ON p.etat = e.nazwa;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                | 10000 |   517K|    25   (0)| 00:00:01 |
|*  1 |  HASH JOIN          |                | 10000 |   517K|    25   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL | OPT_ETATY      |    12 |   180 |     3   (0)| 00:00:01 |
|*  3 |   HASH JOIN         |                | 10000 |   371K|    22   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   205K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("P"."ETAT"="E"."NAZWA")
   3 - access("P"."ID_ZESP"="Z"."ID_ZESP")
```

Jako pierwsze zostaje wykonane połączenie relacji `OPT_ZESPOLY` i `OPT_PRACOWNICY` wg algorytmu *Hash Join* (operacja z id = 3). Następnie zbiór wynikowy tego połączenia jest łączony z relacją `OPT_ETATY` również przy użyciu algorytmu *Hash Join* (operacja z id = 1). Optymalizator stara się tak dobrać kolejność realizacji połączeń, aby na początku zminimalizować rozmiar przetwarzanych danych.

2. Za pomocą wskazówek możesz wpływać na wybór algorytmów do realizacji poszczególnych połączeń. Np. chcielibyśmy, aby relacje `OPT_PRACOWNICY` i `OPT_ZESPOLY` zostały połączone przy użyciu algorytmy *Nested Loops*, natomiast przy łączeniu relacji `OPT_ETATY` był użyty algorytm *Sort Merge*.

```
SELECT /*+ USE_NL(p z) USE_MERGE(e) */
       p.nazwisko, z.nazwa, e.placa_min, e.placa_max
FROM opt_pracownicy p JOIN opt_zespoly z USING(id_zesp)
     JOIN opt_etaty e ON p.etat = e.nazwa;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                | 10000 |   517K|    99   (4)| 00:00:02 |
|   1 |  MERGE JOIN         |                | 10000 |   517K|    99   (4)| 00:00:02 |
|   2 |   SORT JOIN         |                | 10000 |   371K|    95   (3)| 00:00:02 |
|   3 |    NESTED LOOPS     |                | 10000 |   371K|    93   (0)| 00:00:02 |
|   4 |     TABLE ACCESS FULL| OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|*  5 |     TABLE ACCESS FULL| OPT_PRACOWNICY |  2000 | 42000 |    18   (0)| 00:00:01 |
|*  6 |   SORT JOIN         |                |    12 |   180 |     4  (25)| 00:00:01 |
|   7 |    TABLE ACCESS FULL | OPT_ETATY      |    12 |   180 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   5 - filter("P"."ID_ZESP"="Z"."ID_ZESP")
   6 - access("P"."ETAT"="E"."NAZWA")
       filter("P"."ETAT"="E"."NAZWA")
```

Widzimy, że nasze życzenie zostało spełnione. Jednak plan wyraźnie się pogorszył – jego koszt jest czterokrotnie większy od kosztu planu zaproponowanego przez optymalizator.

3. Kolejną cechą planu, którą możemy zmodyfikować używając wskazówek, jest kolejność wykonywania operacji połączenia w planie. Wykonajmy poniższe polecenie i wyświetlmy jego plan.

```
SELECT /*+ LEADING(p e) */
       p.nazwisko, z.nazwa, e.placa_min, e.placa_max
FROM opt_pracownicy p JOIN opt_zespoly z USING(id_zesp)
     JOIN opt_etaty e ON p.etat = e.nazwa;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                | 10000 |   517K|    25   (0)| 00:00:01 |
|*  1 |  HASH JOIN          |                | 10000 |   517K|    25   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL | OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|*  3 |   HASH JOIN         |                | 10000 |   351K|    22   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   205K|    19   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| OPT_ETATY      |    12 |   180 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("P"."ID_ZESP"="Z"."ID_ZESP")
   3 - access("P"."ETAT"="E"."NAZWA")
```

Wskazówka `LEADING` spowodowała, że jako pierwsze zostały połączone relacje `OPT_PRACOWNICY` i `OPT_ETATY`, a relacja `OPT_ZESPOLY` została połączona z wynikiem pierwszej operacji połączenia.

Zmodyfikuj wskazówkę `LEADING` w taki sposób, aby najpierw były łączone relacje `OPT_ZESPOLY` i `OPT_ETATY`. Uwaga! Między tymi relacjami nie ma warunku połączeniowego! Co zauważyłaś/eś w otrzymanym planie?

4. Zmodyfikujmy nieco nasze zapytanie w taki sposób, aby w klauzuli `FROM` jako pierwszy były łączone relacje `OPT_PRACOWNICY` i `OPT_ETATY`. Wyświetl plan wykonania.

```
SELECT p.nazwisko, z.nazwa, e.placa_min, e.placa_max
FROM opt_pracownicy p JOIN opt_etaty e ON p.etat = e.nazwa
     JOIN opt_zespoly z USING(id_zesp);
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                | 10000 |   517K|    25   (0)| 00:00:01 |
|*  1 |  HASH JOIN          |                | 10000 |   517K|    25   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL | OPT_ETATY      |    12 |   180 |     3   (0)| 00:00:01 |
|*  3 |   HASH JOIN         |                | 10000 |   371K|    22   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   205K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

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

   1 - access("P"."ETAT"="E"."NAZWA")
   3 - access("P"."ID_ZESP"="Z"."ID_ZESP")
```

Widzimy, że optymalizator nie zmienił planu w stosunku do zapytania, w którym w klauzuli `FROM` jako pierwsze były łączone relacje `OPT_PRACOWNICY` i `OPT_ZESPOLY`. Jeśli chcemy, aby relacje

były łączone w takiej kolejności, w jakiej zostały umieszczone w klauzuli `FROM` polecenia, możemy użyć wskazówki `ORDERED`.

```
SELECT /*+ ORDERED */
       p.nazwisko, z.nazwa, e.placa_min, e.placa_max
FROM opt_pracownicy p JOIN opt_etaty e ON p.etat = e.nazwa
     JOIN opt_zespoly z USING(id_zesp);

-------------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                | 10000 |   517K|    25   (0)| 00:00:01 |
|*  1 |  HASH JOIN          |                | 10000 |   517K|    25   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL | OPT_ZESPOLY    |     5 |    85 |     3   (0)| 00:00:01 |
|*  3 |   HASH JOIN         |                | 10000 |   351K|    22   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   205K|    19   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| OPT_ETATY      |    12 |   180 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   1 - access("P"."ID_ZESP"="Z"."ID_ZESP")
   3 - access("P"."ETAT"="E"."NAZWA")
```

Jak widać w planie, teraz jako pierwsze łączone są relacje `OPT_PRACOWNICY` i `OPT_ETATY`.

Transformacje

Przyjrzymy się teraz przykładowym sytuacjom, w których optymalizator przekształca zapytanie do lepszej postaci.

1. Wykonaj poniższe polecenie: szukamy w nim identyfikatorów pracowników wraz z identyfikatorami zespołów, do których należą.

```
SELECT id_prac, id_zesp
FROM opt_pracownicy JOIN opt_zespoly USING(id_zesp);
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                | 10000 | 70000 |    19   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 | 70000 |    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("OPT_PRACOWNICY"."ID_ZESP" IS NOT NULL)
```

W planie wykonania tego zapytania nie pojawia się operacja połączenia! Optymalizator wyeliminował ją jako zbędną. Oczywiście mógł to zrobić sam użytkownik na etapie konstrukcji polecenia.

2. Wykonajmy z kolei takie zapytanie.

```
SELECT nazwisko
FROM opt_pracownicy
WHERE id_zesp IN
  (SELECT id_zesp FROM opt_zespoly WHERE nazwa = 'Bazy Danych');
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                |  2000 | 58000 |    22   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |                |  2000 | 58000 |    22   (0)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| OPT_ZESPOLY    |     1 |    17 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| OPT_PRACOWNICY | 10000 |   117K|    19   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("ID_ZESP"="ID_ZESP")
   2 - filter("NAZWA"='Bazy Danych')
```

Tu z kolei w zapytaniu nie ma połączenia, natomiast w planie wykonania pojawia się operacja `HASH JOIN`, czyli realizacja połączenia relacji `OPT_PRACOWNICY` i `OPT_ZESPOLY`. Jest to przykład transformacji, realizowanej automatycznie przez optymalizator, zastępującej zapytanie z podzapytaniem zapytaniem z połączeniem.

W obu powyższych przypadkach możemy zapobiec transformacjom dodając do zapytań wskazówkę `NO_QUERY_TRANSFORMATION`. Użyj ją w obu zapytaniach i wyświetl ich plany wykonania.

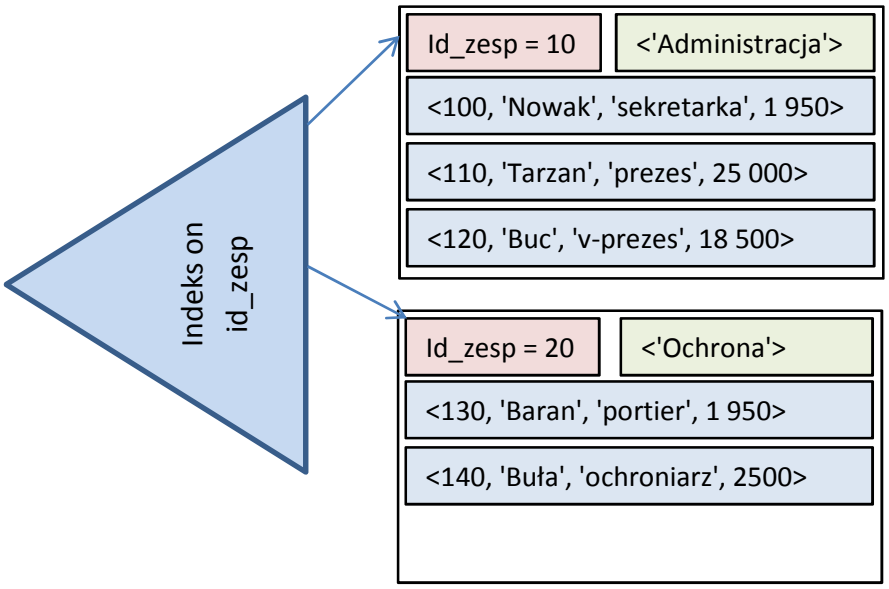
Dodatkowe struktury danych

W niniejszej części ćwiczenia przyjrzymy się dodatkowym strukturom do składowania danych, mianowicie klastrom i relacjom zorganizowanym jak indeksy.

Klastry indeksowe

Klaster to struktura umożliwiająca przechowywanie krotek wielu relacji. Rekordy różnych relacji o takich samych wartościach wybranego atrybutu są przechowywane w tych samych blokach dyskowych. Zastosowanie klastrów może przyśpieszyć wykonanie równościowego połączenia relacji umieszczonych w klastrze oraz operacje dostępu lub grupowania odwołujące się do wspólnego atrybutu klastra.

Przykładową strukturę klastra indeksowego, przechowującego rekordy z relacji `OPT_ZESPOLY` i `OPT_PRACOWNICY` przedstawiono na poniższym rysunku.



1. Utworzymy klaster indeksowy, który będzie składował rekordy, skopiowane z relacji `OPT_PRACOWNICY` i `OPT_ZESPOLY`, połączone na podstawie wartości kolumny `ID_ZESP`.

   W pierwszym kroku definiujemy klaster.

   ```
   CREATE CLUSTER opt_kl_ind_zesp_prac(id_zesp NUMBER)
   INDEX size 4k;
   ```

   Klaster indeksowy nosi `nazwę OPT_KL_IND_ZESP_PRAC` o rozmiarze bloku równym 4 KB. W kolejnym kroku definiujemy indeks klastra.

   ```
   CREATE INDEX opt_zesp_prac_idx ON CLUSTER opt_kl_ind_zesp_prac;
   ```

   Teraz utworzymy relacje `OPT_PRACOWNICY_KL` i `OPT_ZESPOLY_KL`, których rekordy będą składowane w klastrze. Relacje te będą kopią relacji, odpowiednio, `OPT_PRACOWNICY` i `OPT_ZESPOLY`.

```
CREATE TABLE opt_pracownicy_kl CLUSTER opt_kl_ind_zesp_prac (id_zesp)
  AS SELECT * FROM opt_pracownicy;

CREATE TABLE opt_zespoly_kl CLUSTER opt_kl_ind_zesp_prac (id_zesp)
  AS SELECT * FROM opt_zespoly;
```

Na końcu zgromadzimy statystyki dla obu relacji.

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

2. Odczytaj ze słownika bazy danych informacje o relacjach w Twoim schemacie.

```
SELECT table_name, cluster_name
FROM USER_TABLES;

TABLE_NAME                     CLUSTER_NAME
------------------------------ ------------------------------
OPT_PRACOWNICY
...
OPT_PRACOWNICY_KL              OPT_KL_IND_ZESP_PRAC
...
OPT_ZESPOLY
OPT_ZESPOLY_KL                 OPT_KL_IND_ZESP_PRAC
...
```

3. Wykonaj zapytanie, które znajdzie liczbę rekordów w relacji `OPT_PRACOWNICY_KL`, następnie wyświetl plan wykonania zapytania. Czy widzisz jakąś różnicę w stosunku do wcześniej oglądanych planów zapytań do relacji `OPT_PRACOWNICY`?

```
SELECT COUNT(*) FROM opt_pracownicy_kl;

-------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     1 |    17   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |                   |     1 |            |          |
|   2 |   TABLE ACCESS FULL| OPT_PRACOWNICY_KL | 10000 |    17   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

4. Wykonaj teraz zapytanie, które znajdzie liczbę rekordów wyniku połączenia naturalnego relacji `OPT_PRACOWNICY_KL` i `OPT_ZESPOLY_KL`. Wyświetl plan wykonania.

```
SELECT COUNT(*)
FROM opt_zespoly_kl JOIN opt_pracownicy_kl USING(id_zesp);

-------------------------------------------------------------------------------------------
| Id  | Operation              | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                   |     1 |     6 |    22   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE        |                   |     1 |     6 |            |          |
|   2 |   NESTED LOOPS         |                   | 10000 | 60000 |    22   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL   | OPT_ZESPOLY_KL    |     5 |    15 |    17   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS CLUSTER| OPT_PRACOWNICY_KL |  2000 |  6000 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - filter("OPT_ZESPOLY_KL"."ID_ZESP"="OPT_PRACOWNICY_KL"."ID_ZESP")
```

W planie pojawiła się operacja pełnego przeglądu tabeli w klastrze (w planie: `TABLE ACCESS CLUSTER`). Jest to charakterystyczna operacja dostępu do relacji, której rekordy umieszczone są w klastrze.

Operacja połączenia tabel `OPT_PRACOWNICY_KL` i `OPT_ZESPOLY_KL` w zdefiniowanym klastrze, będzie najbardziej wydajna w zbiorze różnych metod wykonywania operacji połączeń. Krotki tych relacji są na bieżąco fizycznie łączone w pliku klastrowym.

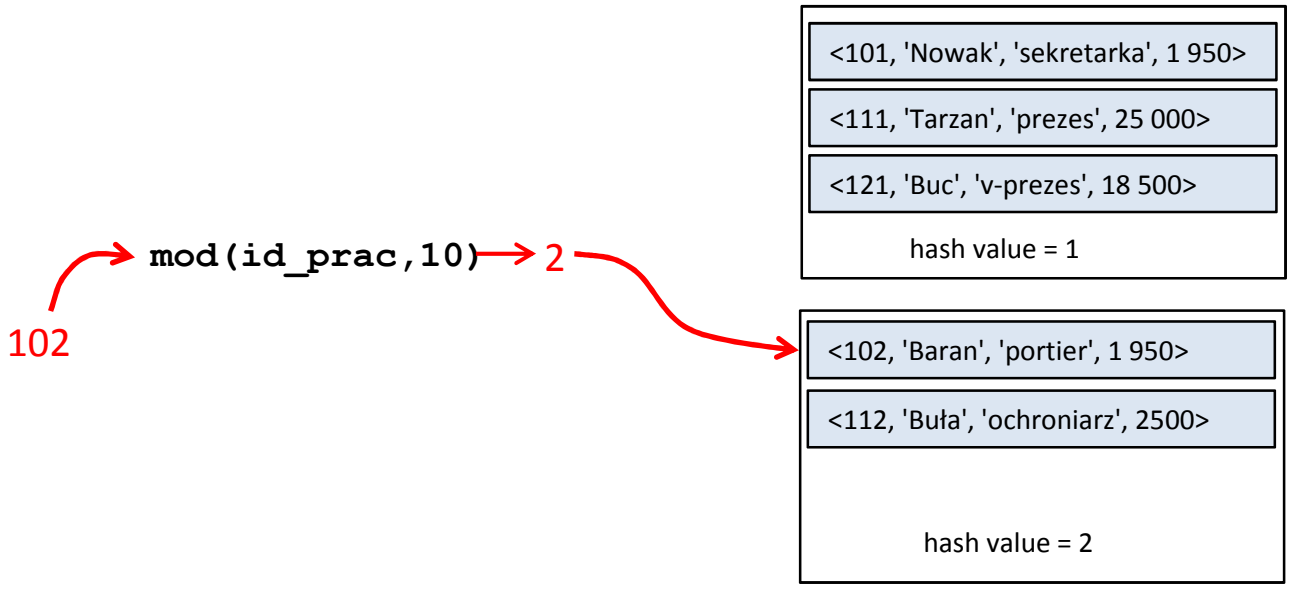
Klastry haszowe

Dla pewnej klasy zapytań klastry (pliki) haszowe oferują najszybszy z możliwych dostępów za pomocą skojarzonej z plikiem funkcji haszowej. Polecenie tworzenia klastra haszowego przedstawiono poniżej.

```
CREATE CLUSTER nazwa_klastra (
atrybut_haszowy typ_danych)
SIZE rozmiar_kubełka
SINGLE TABLE HASHKEYS liczba_kubełków HASH IS funkcja;
```

Wartości `size` i `hashkeys` muszą wynikać z przewidywanej liczby rekordów tabeli, rozmiaru pojedynczych rekordów i rozkładu wartości atrybutu haszowego. Faktyczna liczba kubełków w pliku haszowym jest równa najmniejszej z liczb pierwszych większej od podanej wartości `hashkeys`. Domyślną funkcją haszującą jest funkcja modulo.

Poniższy rysunek przedstawia dostęp do rekordów relacji `OPT_PRACOWNICY`, umieszczonej w klastrze haszowym.



1. Utworzymy klaster haszowy z relacją zawierającą kopię rekordów relacji `OPT_PRACOWNICY`. Klaster będzie wspomagał przeszukiwanie relacji ze względu na wartość kolumny `ID_PRAC`. Wykonaj poniższe polecenie.

```
CREATE CLUSTER opt_kl_hasz_prac (id_prac NUMBER)
SIZE 1K SINGLE TABLE HASHKEYS 1025;
```

Teraz tworzymy w klastrze relację OPT_PRACOWNICY_HASZ.

```
CREATE TABLE opt_pracownicy_hasz CLUSTER opt_kl_hasz_prac(id_prac)
  AS SELECT * FROM opt_pracownicy;
```

Na końcu zbierz statystyki dla relacji OPT_PRACOWNICY_HASZ.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(
     ownname=>'<nazwa schematu>', tabname => 'OPT_PRACOWNICY_HASZ');
END;
```

2. Wykonaj zapytanie, które z relacji OPT_PRACOWNICY_HASZ odczyta dane pracownika o identyfikatorze 1000. Wyświetl i zanalizuj plan wykonania.

```
SELECT nazwisko FROM opt_pracownicy_hasz
WHERE id_prac = 1000;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     1 |    13 |     0   (0)|
|*  1 |  TABLE ACCESS HASH| OPT_PRACOWNICY_HASZ |     1 |    13 |            |
-------------------------------------------------------------------------------

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

   1 - access("ID_PRAC"=1000)
```

Zwróć uwagę na użytą odmianę operacji pełnego przeglądnięcia tabeli, umieszczonej w klastrze haszowym.

Relacje zorganizowane jak indeks

SZBD Oracle umożliwia składowanie rekordów relacji bezpośrednio w strukturze indeksu drzewiastego, zbudowanego na zbiorze wartości klucza głównego relacji. Jest to tzw. relacja zorganizowana jak indeks i jest alternatywą dla pary obiektów: relacja + indeks na kluczu głównym. W takiej strukturze w liściach drzewa składowane są rekordy (zamiast adresów rekordów jak w indeksie). Taka organizacja wymusza posortowanie rekordów relacji według klucza głównego. Relacja zorganizowana jako indeks jest implementacją indeksu rzadkiego.

Postać polecenia tworzącego relację zorganizowaną jak indeks przedstawiono poniżej.

```
CREATE TABLE nazwa_tabeli
(specyfikacja kolumn i więzów integralności)
ORGANIZATION INDEX
PCTTHRESHOLD <procent>
OVERFLOW TABLESPACE <przestrzeń tabel>
INCLUDING <atrybut>;
```

Klauzula `PCTTHRESHOLD` definiuje procent objętości liści struktury, przeznaczony do składowania pojedynczego rekordu. Domyślną i maksymalną wartością jest 50%, co oznacza co najmniej dwa rekordy na liść. Rekordy o wielkości przekraczającej 50% objętości liścia są dzielone na dwie części. Jedna część, która musi zawierać co najmniej cały klucz, pozostaje w strukturze. Wartości pozostałych atrybutów są lokowane w dodatkowym obszarze poza strukturą.

Klauzula `OVERFLOW TABLESPACE` wskazuje przestrzeń tabel, w której zostaną umieszczone części rekordów, nie mieszczące się w liściach struktury .

Klauzula `INCLUDING` atrybut wskazuje alternatywny sposób podziału zbyt długich rekordów. Parametr definiuje ostatni atrybut, w kolejności wynikającej z definicji tabeli, który musi się znaleźć w liściach struktury.

1. Utwórz relację o nazwie OPT_PRACOWNICY_IOT, która będzie zorganizowana jak indeks. Struktura relacji ma składać się z następujących kolumn:

   - `ID_PRAC NUMBER`, klucz podstawowy o nazwie `OP_IOT_PK`,
   - `NAZWISKO VARCHAR2(50)`,
   - `PLACA NUMBER`,
   - `PLACA_DOD NUMBER`,
   - `ETAT VARCHAR2(10)`.

   Kolumny do `PLACA` włącznie mają być przechowywane w liściach struktury, wartości pozostałych kolumn mają być składowane w Twojej domyślnej przestrzeni tabel (jej nazwę odczytasz zapytaniem `SELECT default_tablespace FROM user_users`).

   ```
   CREATE TABLE OPT_PRACOWNICY_IOT
   ...
   ORGANIZATION INDEX
   ...;
   ```

Wstaw do utworzonej relacji kopię rekordów z relacji OPT_PRACOWNICY.

```
INSERT INTO opt_pracownicy_iot
SELECT id_prac, nazwisko, placa, placa_dod, etat FROM opt_pracownicy;
```

Zbierz statystyki dla nowej relacji.

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

2. Odczytaj ze słownika bazy danych informacje o relacjach w Twoim schemacie.

```
SELECT table_name, IOT_TYPE, iot_name
FROM USER_TABLES;

TABLE_NAME                     IOT_TYPE     IOT_NAME
------------------------------ ------------ ------------------------------
OPT_PRACOWNICY
...
OPT_PRACOWNICY_IOT             IOT
...
SYS_IOT_OVER_289412            IOT_OVERFLOW OPT_PRACOWNICY_IOT
...
```

Zwróć uwagę na rekord opisujący relację, która w kolumnie IOT_TYPE ma wartość „IOT_OVERFLOW”. Jest to relacja, przechowują wartości tych kolumn relacji OPT_PRACOWNICY_IOT, które są składowane poza strukturą indeksową. Spróbuj wykonać zapytanie do tej relacji.

3. Wykonaj zapytanie, które odczyta wszystkie dane relacji OPT_PRACOWNICY_IOT. Wyświetl i zanalizuj jego plan.

```
SELECT * FROM opt_pracownicy_iot;

-------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           | 10000 |   283K|  9487   (0)| 00:01:54 |
|   1 |  INDEX FAST FULL SCAN| OP_IOT_PK | 10000 |   283K|  9487   (0)| 00:01:54 |
-------------------------------------------------------------------------------
```

Jaka metoda została użyta? Czy jest to metoda dostępu do relacji czy do indeksu?

4. Wykonaj kolejne zapytanie, które odczyta z relacji OPT_PRACOWNICY_IOT. dane pracownika o identyfikatorze równym 1000 Wyświetl i zanalizuj jego plan.

```
SELECT * FROM opt_pracownicy_iot
WHERE id_prac = 1000;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |    29 |     2   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| OP_IOT_PK |     1 |    29 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("ID_PRAC"=1000)
```

Co tym razem pojawiło się w planie wykonania?

5. Relacja `OPT_PRACOWNICY_IOT`, mimo że sama ma strukturę indeksu, może również zostać wyposażona w dodatkowe indeksy, umożliwiające szybkie jej przeszukiwanie ze względu na wartości innych kolumn niż te z klucza podstawowego.

   Utwórz indeks, który ułatwi przeszukiwanie relacji `OPT_PRACOWNICY_IOT` ze względu na wartości kolumny `NAZWISKO`.

```
CREATE INDEX op_pr_iot_nazwisko ON opt_pracownicy_iot (nazwisko);
```

   Wykonaj zapytanie, które znajdzie rekord opisujący pracownika o nazwisku „Prac155”. Wyświetl i zanalizuj plan wykonania tego polecenia.

```
SELECT * FROM opt_pracownicy_iot
WHERE nazwisko = 'Prac155';

-------------------------------------------------------------------------------------------
| Id  | Operation         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                    |     1 |    29 |     2   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| OP_IOT_PK          |     1 |    29 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN| OP_PR_IOT_NAZWISKO |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

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

Jakie operacje zostały umieszczone w planie wykonania?