



ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ

Кафедра №42 «Криптология и кибербезопасность»

*Федеральное государственное автономное образовательное
учреждение высшего образования*

«Национальный исследовательский ядерный университет «МИФИ»»

ЛАБОРАТОРНАЯ РАБОТА №2-3:

«Индексирование и кластеризация данных.»

Аверин Владислав

Группа: Б19-505

Октябрь, 2022

Содержание

1.	Типичные для БД запросы	4
2.	Индексы (B-деревья и BMAP)	6
3.	Хранение таблиц по индексу/в кластере	12
Выводы:		18

Цель работы

Ускорить выполнение запросов к базе данных за счёт оптимизации физической структуры данных и создания внешних индексов.

Ход работы

1. Если это ещё не было сделано для текущей схемы данных, разработать и проверить 5–10 запросов, которые, согласно легенде, являются типичными запросами к базе данных для разных категорий пользователей и регулярно используются при её эксплуатации;
2. Запросить у СУБД схему выполнения каждого из этих запросов. Для этого можно настроить функцию AUTOTRACE в SQL*Plus или использовать функцию контекстного меню EXPLAIN в SQL Developer;
3. При необходимости, создать дополнительные индексы на основе B-деревьев. Обосновать их полезность. Убедиться в том, что индексы используются при выполнении запросов (план выполнения должен измениться). Если такая необходимость отсутствует, специально разработать дополнительный запрос, для которого применение индекса целесообразно;
4. Аналогично, создать индекс на основе битовых карт, обосновать его полезность и убедиться, что он используется;
5. Существуют ли в схеме данных таблицы, которые целесообразно организовать по индексу? Изменить организацию этих таблиц; сравнить новый план выполнения запросов с её участием с прежним;
6. Существуют ли в схеме данных пары (тройки,...) таблиц, которые целесообразно хранить в кластере? Если они есть, поместить эти таблицы в кластер; сравнить новый план выполнения запросов с её участием с прежним;
7. Целесообразно ли применение hash-кластера для ускорения работы с одной или несколькими таблицами? Если да, поместить её (их) в hash-кластер и сравнить новый план выполнения запросов с её участием с прежним;
8. Оформить отчёт.

1. Типичные для БД запросы

С учетом специфики выполняемой лабораторной, желательно выполнять те запросы, где явно можно увидеть использование индексов. Но чтобы показать их не полную власть над всеми запросами (пока мы сами не оптимизируем работу), в начале приведем примеры типичных запросов, где стандартных созданных при формировании индексов не хватает.

Медперсонал: изменить значение атрибута med_rec у определенного сотрудника

```
UPDATE employeesPs
SET med_rec = 'updated'
WHERE employee_id = 18;
```

План выполнения:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
UPDATE STATEMENT			1	1
UPDATE	INFERNAL.EMPLOYEES			
INDEX	INFERNAL.EMPLOYEES_PK	UNIQUE SCAN	1	0
Access Predicates				
EMPLOYEE_ID=17				

EMPLOYEES_PK – это то название объекта, которое мы давали при создании таблицы в CONSTRAINT. То есть в данном случае индекс используется (что логично).

(Насчет UNIQUE SCAN и других методов доступа к данным будет описано далее.)

HR-отдел: найти всех сотрудников из одного отдела

```
SELECT * FROM EmployeesPs
WHERE department_id = 3;
```

План:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			3	3
TABLE ACCESS	INFERNAL.EMPLOYEES	FULL	3	3
Filter Predicates				
DEPARTMENT_ID=3				

Т.е. теперь СУБД просматривает все строки на наличие выполнения условия (т.к. на атрибут department_id индекса у нас нет).

Бухгалтерия: найдем среднюю зарплату по участку

```
SELECT AVG(SALARY) AS AVG FROM EmployeesPs);
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	3
SORT		AGGREGATE	1	
TABLE ACCESS	INFERNAL.EMPLOYEES	FULL	35	3

Опять же, поиск идет по всей таблице без использования индекса.

Отдел руководства: найти открытые дела, за которым закреплен определенный сотрудник

```
SELECT AssignedCasesPs.* FROM AssignedCasesps, casesPs
WHERE
casesps.case_id = assignedcasesps.case_id
AND status_id = 1
AND employee_id = 2;
```

План:

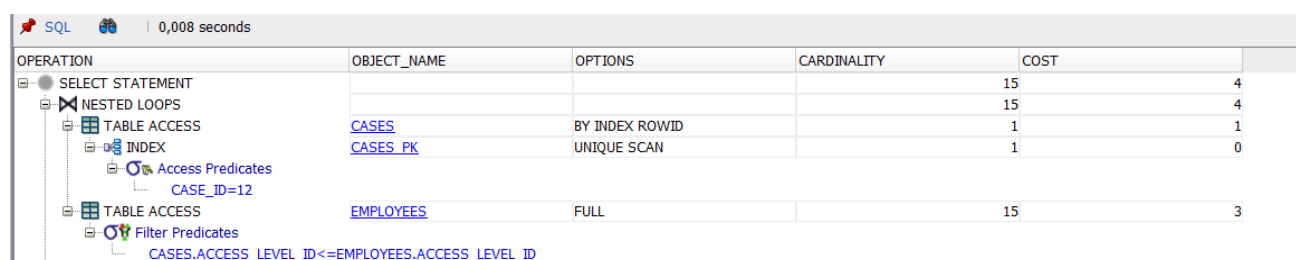
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			9	3
NESTED LOOPS			9	3
TABLE ACCESS	INFERNAL.CASES	FULL	21	3
Filter Predicates				
STATUS_ID=1				
INDEX	INFERNAL.ASSIGNEDCASES_PK	UNIQUE SCAN	1	0
Access Predicates				
AND				
EMPLOYEE_ID=2				
CASESPS.CASE_ID=ASSIGNEDCASESPS.CASE_ID				

Oracle ищет в AssignedCases строки с определенным employee_id по индексу, и делает полный перебор таблицы Cases для соединения по условию case_id и нужного status_id. Причем, поиск по индексу не занимает практически никакого времени, в отличие от того же TABLE FULL SCAN таблицы Cases. Поэтому неплохо было бы добавить новые индексы, но не бездумно на каждый из атрибутов всех таблиц, т.к. тогда БД начнет жутко тормозить при любом добавлении (все индексы будут балансироваться для добавления), а только те, что действительно смогут принести пользу.

2. Индексы (B-деревья и BMAP)

Насколько я понял, Oracle (как и в принципе все СУБД) по определению сразу создает для всех первичных ключей индексы, и индексы на основе B-деревьев (т.к. операция поиска по pk – основополагающая операция для баз данных). Поэтому практически все соединения выполняются максимально быстро, т.к. в подавляющем большинстве случаев соединения (не важно, какого типа) происходят по первичному ключу одной из таблиц. Моя БД не исключение: все потенциально типичные запросы проходят через первичные ключи (а следовательно, реализуются с помощью индексов). Самое простое решение – попробовать сделать соединение без использования первичного и внешнего ключей (гениально, знаю). В рассматриваемых таблицах Employees и Cases есть атрибут access_level_id – какой уровень допуска у сотрудника и требуемый уровень для доступа к делу соответственно. Попробуем найти всех сотрудников, которые могут быть допущены к работе с определенным делом:

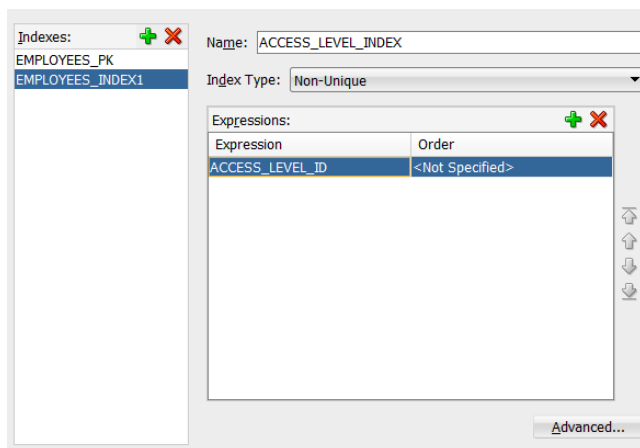
```
SELECT case_id, cases.access_level_id, Employees.employee_id, employees.access_level_id
FROM Cases, employees
WHERE case_id = 12 AND cases.access_level_id <= Employees.access_level_id;
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			15	4
NESTED LOOPS			15	4
TABLE ACCESS	CASES	BY INDEX ROWID	1	1
INDEX	CASES_PK	UNIQUE SCAN	1	0
Access Predicates				
CASE_ID=12				
TABLE ACCESS	EMPLOYEES	FULL	15	3
Filter Predicates				
CASES.ACCESS_LEVEL_ID<=EMPLOYEES.ACCESS_LEVEL_ID				

Как мы видим, в таблице Employees (для соответствия строки из Cases, где для case_id есть индекс) используется FULL table access, т.е. перебираются все строки для поиска нужных по условию.

Теперь создадим индекс по Employees.access_level_id (из-за лени автора, был использован sql developer, не бейте):



Вызовем план еще раз, иии....

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				15 4
NESTED LOOPS				15 4
TABLE ACCESS	CASES	BY INDEX ROWID	1	1
INDEX	CASES_PK	UNIQUE SCAN	1	0
Access Predicates	CASE_ID=12			
TABLE ACCESS	EMPLOYEES	FULL		15 3
Filter Predicates	CASES.ACCESS_LEVEL_ID<=EMPLOYEES.ACCESS_LEVEL_ID			

Ничего. Он все также ищет нужную строку в Cases или в самом индексе – BY INDEX ROWID, UNIQUE SCAN говорит о том, что вернется одно (или ноль) строк, а FULL как раз показывает, что СУБД не использовала наш индекс... Почему? Потому что используется неравенство, вместо равенства (видимо, для таких операций, где для нужно дополнительно обрабатывать все листовые блоки дерева, как для равенств, система считает это медленнее, чем TABLE FULL SCAN, но что-то как-то сомнительно). Но если мы сделаем равенство, то появится уже RANGE SCAN с использованием нашего индекса (т.е. субд не перебирает все листовые блоки):

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				4 2
NESTED LOOPS				4 2
TABLE ACCESS	CASES	BY INDEX ROWID	1	1
INDEX	CASES_PK	UNIQUE SCAN	1	0
Access Predicates	CASE_ID=12			
TABLE ACCESS	EMPLOYEES	BY INDEX ROWID BATCHED		4 1
INDEX	ACCESS_LEVEL_INDEX	RANGE SCAN	4	0
Access Predicates				

(ROWID BATCHED как я понял просто более оптимизированный ROWID, если нужная информация уже есть в блоке индекса, то обращаться к самому ряду таблицы не надо)

Можно еще расширить это дело и вывести таких сотрудников вообще для всех дел (согласен, немного странный запрос сам по себе, но может, кто-то захочет его как подзапрос использовать):

```
SQL> SELECT case_id, casesps.access_level_id, Employeesps.employee_id, employeesps.access_level_id
  2  FROM Casesps, employeesps
  3  WHERE casesps.access_level_id = Employeesps.access_level_id;
```

План без использования индекса по employees.access_level_id:

```
Execution Plan
-----
Plan hash value: 4150962751

-----
| Id | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |            |    704 |  9152 |      8 (25)| 00:00:01 |
|  1 |  MERGE JOIN        |            |    704 |  9152 |      8 (25)| 00:00:01 |
|  2 |    SORT JOIN       |            |     35 |   210 |      4 (25)| 00:00:01 |
|  3 |      TABLE ACCESS FULL | EMPLOYEES |     35 |   210 |      3 (0)| 00:00:01 |
|*  4 |        SORT JOIN    |            |     47 |   329 |      4 (25)| 00:00:01 |
|  5 |          TABLE ACCESS FULL | CASES     |     47 |   329 |      3 (0)| 00:00:01 |
-----

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

   4 - access(INTERNAL_FUNCTION("CASES"."ACCESS_LEVEL_ID")<=INTERNAL_FUNC
        TION("EMPLOYEES"."ACCESS_LEVEL_ID"))
        filter(INTERNAL_FUNCTION("CASES"."ACCESS_LEVEL_ID")<=INTERNAL_FUNC
        TION("EMPLOYEES"."ACCESS_LEVEL_ID"))

SQL> _
```

Он снова выполняет FULL FULL SCAN.

При включении индекса обратно, он использует уже INDEX FAST FULL SCAN (то же, что и INDEX FULL SCAN, но без обращения к самой таблице, т.к. нужные данные (access_level_id) уже есть в листе дерева):

Execution Plan

Plan hash value: 1983416076

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		704	9152	7 (29)	00:00:01
1	MERGE JOIN		704	9152	7 (29)	00:00:01
2	SORT JOIN		35	210	3 (34)	00:00:01
3	VIEW	index\$_join\$_002	35	210	2 (0)	00:00:01
* 4	HASH JOIN					
5	INDEX FAST FULL SCAN	ACCESS_LEVEL_INDEX	35	210	1 (0)	00:00:01
6	INDEX FAST FULL SCAN	EMPLOYEES_PK	35	210	1 (0)	00:00:01
* 7	SORT JOIN		47	329	4 (25)	00:00:01
8	TABLE ACCESS FULL	CASES	47	329	3 (0)	00:00:01

Predicate Information (identified by operation id):

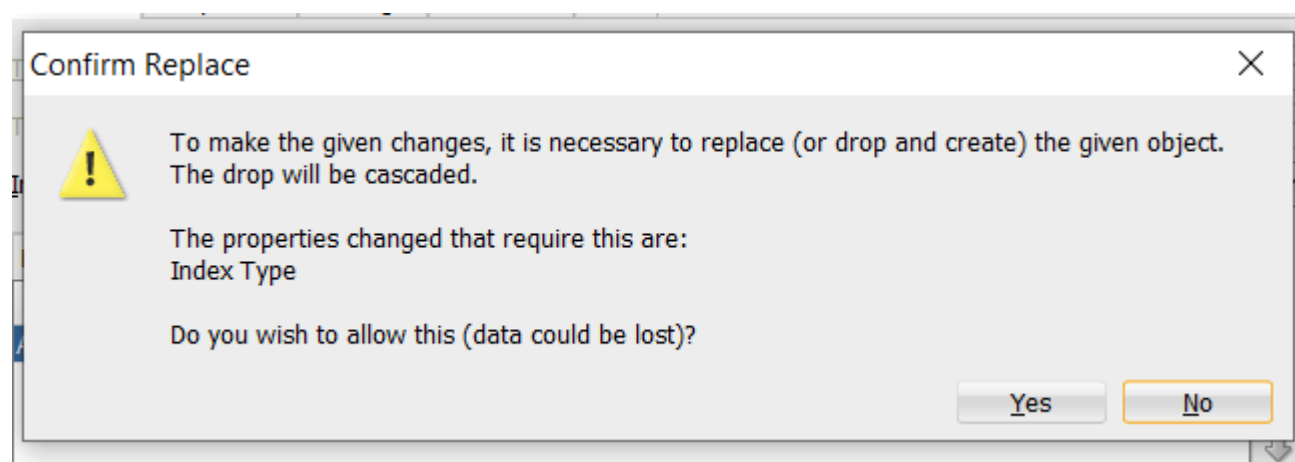
4 - access(ROWID=ROWID)
 7 - access(INTERNAL_FUNCTION("CASES"."ACCESS_LEVEL_ID")<=INTERNAL_FUNCTION("EMPLOYEE S"."ACCESS_LEVEL_ID"))
 filter(INTERNAL_FUNCTION("CASES"."ACCESS_LEVEL_ID")<=INTERNAL_FUNCTION("EMPLOYEE S"."ACCESS_LEVEL_ID"))

SQL>

Таким образом, прирост производительности наблюдается, но он достаточно мал (т.к. сами таблицы были размером не больше 40 строк, что достаточно мало).

Индексы на основе битовых карт.

В силу специфики концепции битовых карт использовать их для навигации по первичным ключам не рационально (вместо линейного размера матрицы с учетом неизменности кол-ва значений атрибута, получим квадратичный – $n*n$), поэтому в данном случае ACCESS_LEVEL_INDEX также можно настроить на использования битовой карты:



Выполним тот же запрос с соединением двух таблиц:

```
SELECT case_id, cases.access_level_id, Employees.employee_id, employees.access_level_id
FROM Cases, employees
WHERE cases.access_level_id <= Employees.access_level_id;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				7
MERGE JOIN			704	7
VIEW			704	7
JOIN		JOIN	35	3
index\$_join\$_002			35	2
Access Predicates				
ROWID=ROWID				
BITMAP CONVERSION		TO ROWIDS	35	1
BITMAP INDEX	ACCESS_LEVEL_INDEX	FULL SCAN		
INDEX	EMPLOYEES_PK	FAST FULL SCAN	35	1
JOIN		JOIN	47	4
Access Predicates				
INTERNAL_FUNCTION(CASES.ACCESS_LEVEL_ID)<=INTERNAL_FUNCTION(EMPLOYEES.ACCESS_LEVEL_ID)				
Filter Predicates				
INTERNAL_FUNCTION(CASES.ACCESS_LEVEL_ID)<=INTERNAL_FUNCTION(EMPLOYEES.ACCESS_LEVEL_ID)				
TABLE ACCESS	CASES	FULL	47	3

Насколько я понимаю, он точно так же, соединяет каждую строку из Cases (TABLE FULL SCAN) со строками, которые находит через BITMAP INDEX. Причем точно так же, обращения к самому блоку таблицы не идет, вся нужная информация (employee_id и access_level_id) уже есть в карте.

Note: интересно, что если попытаться найти ту информацию, которой в битовой карте явно нет (т.е. если необходимо обращение к памяти):

```
SELECT case_id, cases.access_level_id, Employees.first_name
FROM Cases, employees
WHERE cases.access_level_id <= Employees.access_level_id;
```

то Oracle будет не будет использовать данный индекс, а будет делать просто полное соединение с одновременным отбором подходящих по условию строк:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				8
MERGE JOIN			704	8
JOIN		JOIN	35	4
TABLE ACCESS	EMPLOYEES	FULL	35	3
JOIN		JOIN	47	4
Access Predicates				
INTERNAL_FUNCTION(CASES.ACCESS_LEVEL_ID)<=INTERNAL_FUNCTION(EMPLOYEES.ACCESS_LEVEL_ID)				
Filter Predicates				
INTERNAL_FUNCTION(CASES.ACCESS_LEVEL_ID)<=INTERNAL_FUNCTION(EMPLOYEES.ACCESS_LEVEL_ID)				
TABLE ACCESS	CASES	FULL	47	3

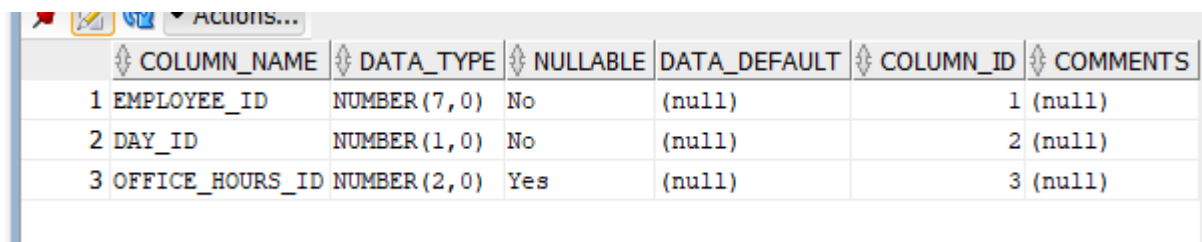
(P.S. Мне чем-то эти индексы напомнили типы хранения графов в памяти: их тоже можно хранить как список или в матричном представлении. И в зависимости от вида графа тот или иной способ будет эффективнее по скорости, но проигрывает в размере. Как, впрочем, и всегда: нужно искать баланс между производительностью и требуемой памятью)

3. Хранение таблиц по индексу/в кластере

Для реализации таблицы через индекс она должна максимально удовлетворять следующим условиям:


- Редкое обновление (добавление и удаление строк);
- Большой размер (кол-во строк);
- Хорошо бы чтобы еще и все данные ряда умещались в одном блоке (чтобы не выделять CHAINED BLOCK), но с этим вроде бы проблем быть не должно

Под данное описание очень хорошо подходят таблицы измерений, которые создавались для нормализации таблицы; из всех таких таблиц наиболее большая это WorkShedule – в ней на данный момент 135 строк, с составным первичным ключом (из двух атрибутов), и изменяться она должна (официально) только тогда, когда сотрудник увольняется/нанимается.



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	EMPLOYEE_ID	NUMBER(7,0)	No	(null)	1	(null)
2	DAY_ID	NUMBER(1,0)	No	(null)	2	(null)
3	OFFICE_HOURS_ID	NUMBER(2,0)	Yes	(null)	3	(null)

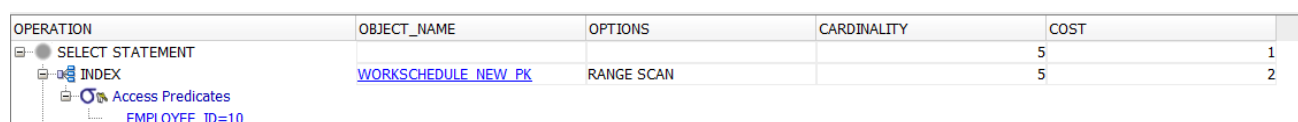
Собственно, тогда обычный select с условием поиска по определенному сотруднику в первом случае будет использовать свой индекс для данного ключа:



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				5 2
TABLE ACCESS	WORKSCHEDULE	BY INDEX ROWID BATCHED		5 2
INDEX	WORKSCHEDULE_PK	RANGE SCAN		5 1

Access Predicates
EMPLOYEE_ID=10

А в случае с индекс-таблицей:



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				5 1
INDEX	WORKSCHEDULE_NEW_PK	RANGE SCAN		5 2

Access Predicates
EMPLOYEE_ID=10

Т.е. таблица сама является индексом.

Хранение таблиц в кластере.

Изначально у меня была идея создать кластер из трех таблиц. Но не с одним и тем же ключом, а чтобы реализовать двойное соединение сотрудник-дело, организованное через дополнительную таблицу Assigned Cases. Но насколько я понял, так просто это не сделать. Может, и возможно создать кластер из двух кластеров (если каждое соединение организовать отдельно), но как-то мне не нравится то, что в таком случае будет происходить внутри. Поэтому обойдемся созданием одного кластера из этих двух: соединение AssignedCases с Cases.

Ключом кластера будет, как ни странно, case_id.

Во-первых, привилегий на CREATE CLUSTER мы не выдавали, так надо выдать:

```
GRANT CREATE CLUSTER TO Infernal;
```

Кластер:

```
CREATE CLUSTER cases_cl (case_id NUMBER (7,0))  
SIZE 1024  
TABLESPACE users;
```

```
Cluster CASES_CL created.
```

(SIZE 1024 по идее должно хватить, чтобы вместить всю информацию, с учетом полей VARCHAR2(100) и (400))

Создаем обе таблицы заново, с учетом уникальности имен индексов и связи с кластером (остальное создание взято из lab1-1):

```
CREATE TABLE Cases_new  
(  
    case_id NUMBER(7,0),  
    case_name VARCHAR2(100) NOT NULL,  
    status_id NOT NULL,  
    access_level_id NOT NULL,  
    description VARCHAR2(400) NOT NULL,  
    start_date DATE,  
    close_date DATE,  
    CONSTRAINT Cases_pk_new PRIMARY KEY (case_id),  
    CONSTRAINT Cases_fk_status_new  
        FOREIGN KEY (status_id)  
        REFERENCES StatusStates(status_id),  
    CONSTRAINT Cases_fk_access_new  
        FOREIGN KEY (access_level_id)  
        REFERENCES AccessLevels(access_level_id)  
)  
CLUSTER cases_cl (case_id);
```

Table CASES_NEW created.

```
CREATE TABLE AssignedCases_new
(
    employee_id,
    case_id NUMBER(7,0),
    CONSTRAINT AssignedCases_pk_new PRIMARY KEY (employee_id, case_id),
    CONSTRAINT AssignedCases_fk_employee_new
        FOREIGN KEY (employee_id)
        REFERENCES Employees(employee_id),
    CONSTRAINT AssignedCases_fk_case_new
        FOREIGN KEY (case_id)
        REFERENCES Cases_new(case_id)
)
CLUSTER cases_cl (case_id);
```

Table ASSIGNEDCASES_NEW created.

Причем, понадобилось явно указать тип case_id несмотря на то, что это внешний ключ; иначе Oracle не давал создать таблицу:

```
Error report -
ORA-01753: определение столбца несовместимо с определением кластер-столбца
01753. 00000 - "column definition incompatible with clustered column definit
```

Экспортируем и импортируем все данные из старых таблиц в новые и выполним запрос на соединение:

.....

А, или не импортируем :)

```
ORA-02032: нельзя использовать кластеризованные таблицы, пока не создан кластер-индекс
```

Значит, создадим (хотя непонятно, почему он автоматически не создается)

```
CREATE INDEX cases_cl_idx ON CLUSTER cases_cl;
```

Index CASES_CL_IDX created.

Для обычного соединения:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				
HASH JOIN				157
Access Predicates				4
ASSIGNEDCASES.CASE_ID=CASES.CASE_ID				
TABLE ACCESS	CASES	FULL	47	3
INDEX	ASSIGNEDCASES_PK	FULL SCAN	157	1

Для соединения кластеризованных таблиц:

SELECT STATEMENT				157	7
HASH JOIN				157	7
Access Predicates					
ASSIGNEDCASES_NEW.CASE_ID=CASES_NEW.CASE_ID					
INDEX	ASSIGNEDCASES_PK_NEW	FAST FULL SCAN	157	2	
TABLE ACCESS	CASES_NEW	FULL	50	5	

Ожидания не оправдались, оно работает медленнее.... Хотя во втором случае и идет FAST FULL SCAN за счет того, что вся информация сразу хранится в индексе.

Проверено: даже если выполнить двойное соединение дополнительно с таблицей Employees, то оно все равно будет работать медленнее при работе с кластером.

Однако, если сделать выборочный запрос (с определенным id), используя тем самым INDEX ROWID, то эффективность обоих методов одинакова, то в случае с кластеризацией выигрыш более существенен, чем при полном просмотре таблиц (в чем, как мы знаем, кластеры не могут помочь):

Не в кластере:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			7	4
NESTED LOOPS			7	4
NESTED LOOPS			35	4
TABLE ACCESS	CASES	BY INDEX ROWID	1	1
INDEX	CASES_PK	UNIQUE SCAN	1	0
Access Predicates	CASES.CASE_ID=12			
TABLE ACCESS	EMPLOYEES	FULL	35	3
INDEX	ASSIGNEDCASES_PK	UNIQUE SCAN	1	0
Access Predicates	AND			
	ASSIGNEDCASES.EMPLOYEE_ID=EMPLOYEES.EMPLOYEE_ID			
	ASSIGNEDCASES.CASE_ID=12			

В кластере:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			7	5
NESTED LOOPS			7	5
NESTED LOOPS			35	5
TABLE ACCESS	CASES_NEW	BY INDEX ROWID	1	2
INDEX	CASES_PK_NEW	UNIQUE SCAN	1	1
Access Predicates				
	CASES_NEW.CASE_ID=12			
TABLE ACCESS	EMPLOYEES	FULL	35	3
INDEX	ASSIGNEDCASES_PK_NEW	UNIQUE SCAN	1	0
Access Predicates				
AND				
	ASSIGNEDCASES_NEW.EMPLOYEE_ID=EMPLOYEES.EMPLOYEE_ID			
	ASSIGNEDCASES_NEW.CASE_ID=12			

Hash-кластер:

Самым простым примером использования будет создание hash-кластера для таблицы Employees. Поставим ограничение, что в участке не могут работать более 1000 человек, что уже многовато. Создадим кластер и копию таблицы Employees:

```
CREATE CLUSTER employees_hash_cluster (  
    employee_id NUMBER (7,0),  
    department_id NUMBER(2,0),  
    post_id NUMBER(2,0),  
    access_level_id Number(2,0),  
    first_name VARCHAR2(30),  
    second_name VARCHAR2(30),  
    patronymic VARCHAR2(30),  
    age NUMBER(2, 0)  
)  
HASH IS employee_id HASHKEYS 1024  
SIZE 1024 SINGLE TABLE  
TABLESPACE users;
```

```
CREATE TABLE Employees_new  
(  
    employee_id NUMBER(7,0),  
    department_id NUMBER(2,0) NOT NULL,  
    post_id NUMBER(2,0) NOT NULL,  
    access_level_id NUMBER(2,0) NOT NULL,  
    first_name VARCHAR2(30) NOT NULL,  
    second_name VARCHAR2(30) NOT NULL,  
    patronymic VARCHAR2(30),  
    age NUMBER(2, 0) NOT NULL,  
    employment_date DATE NOT NULL,  
    CONSTRAINT Employees_pk_new PRIMARY KEY (employee_id),  
    CONSTRAINT Employees_fk_department_new  
        FOREIGN KEY (department_id)  
        REFERENCES Departments(department_id),  
    CONSTRAINT Employees_fk_post_new  
        FOREIGN KEY (post_id)  
        REFERENCES Posts(post_id),  
    CONSTRAINT Employees_fk_access_new  
        FOREIGN KEY (access_level_id)  
        REFERENCES AccessLevels(access_level_id)  
)  
CLUSTER employees_hash_cluster (employee_id, department_id, post_id, access_level_id, first_name, second_name, patronymic, age);
```

И выполним двойное соединение:

```
SELECT * FROM Assignedcases, cases, employees  
WHERE employees.employee_id = 12 AND assignedcases.case_id = cases.case_id AND assignedcases.employee_id = employees.employee_id;  
  
SELECT * FROM Assignedcases, cases, employees_new  
WHERE employees_new.employee_id = 12 AND assignedcases.case_id = cases.case_id AND assignedcases.employee_id = employees_new.employee_id;
```

Однако вопреки моим ожиданиям, поиск подходящей по условию строки в случае с кластером так же, как и в кластеризованных таблицах, происходит медленнее, чем с обычным B-tree индексом:

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	
SELECT STATEMENT				4	4
NESTED LOOPS				4	4
NESTED LOOPS				47	4
TABLE ACCESS	EMPLOYEES	BY INDEX ROWID		1	1
INDEX	EMPLOYEES_PK	UNIQUE SCAN		1	0
Access Predicates	EMPLOYEES.EMPLOYEE_ID=12				
TABLE ACCESS	CASES	FULL		47	3
INDEX	ASSIGNEDCASES_PK	UNIQUE SCAN		1	0
Access Predicates	ASSIGNEDCASES.EMPLOYEE_ID=12				
AND	ASSIGNEDCASES.CASE_ID=CASES.CASE_ID				

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	
SELECT STATEMENT				4	5
NESTED LOOPS				4	5
NESTED LOOPS				47	5
TABLE ACCESS	EMPLOYEES_NEW	BY INDEX ROWID		1	2
INDEX	EMPLOYEES_PK_NEW	UNIQUE SCAN		1	1
Access Predicates	EMPLOYEES_NEW.EMPLOYEE_ID=12				
TABLE ACCESS	CASES	FULL		47	3
INDEX	ASSIGNEDCASES_PK	UNIQUE SCAN		1	0
Access Predicates	ASSIGNEDCASES.EMPLOYEE_ID=12				
AND	ASSIGNEDCASES.CASE_ID=CASES.CASE_ID				

Возможно, это связано со слишком малым размером таблиц, и использовать такую оптимизацию не рационально. Либо Hash-кластеризация для одиночной таблицы используется в каких-то других случаях. Хотя, вроде бы мы и должны были выиграть при операции чтения по индексу (которой у hash не будет).

Выводы:

В результате данной лабораторной работы были изучены базовые механизмы работы поиска в СУБД: индексы. Рассмотрены их физическая и логическая структуры, добавлены дополнительные индексы на основе битовых карт и В-деревьев, а также проведены практические эксперименты с созданием кластеров таблиц, из которых был сделан вывод, что кластеризация в данной БД не сможет в достаточной мере ускорить ее (по крайней мере, пока она находится в исходном состоянии).

Интересное наблюдение: при создании индексов вообще на все атрибуты таблицы Cases и добавлении новой строки, СУБД действительно практически на одну секунду залагала, а сам коммит длился не как обычно 0,001 сек, а намного больше. Поэтому индексация (ровно как и кластеризация) не является панацеей, и своими действиями мы можем сделать только хуже (что я, судя по всему и сделал при создании кластеров :))