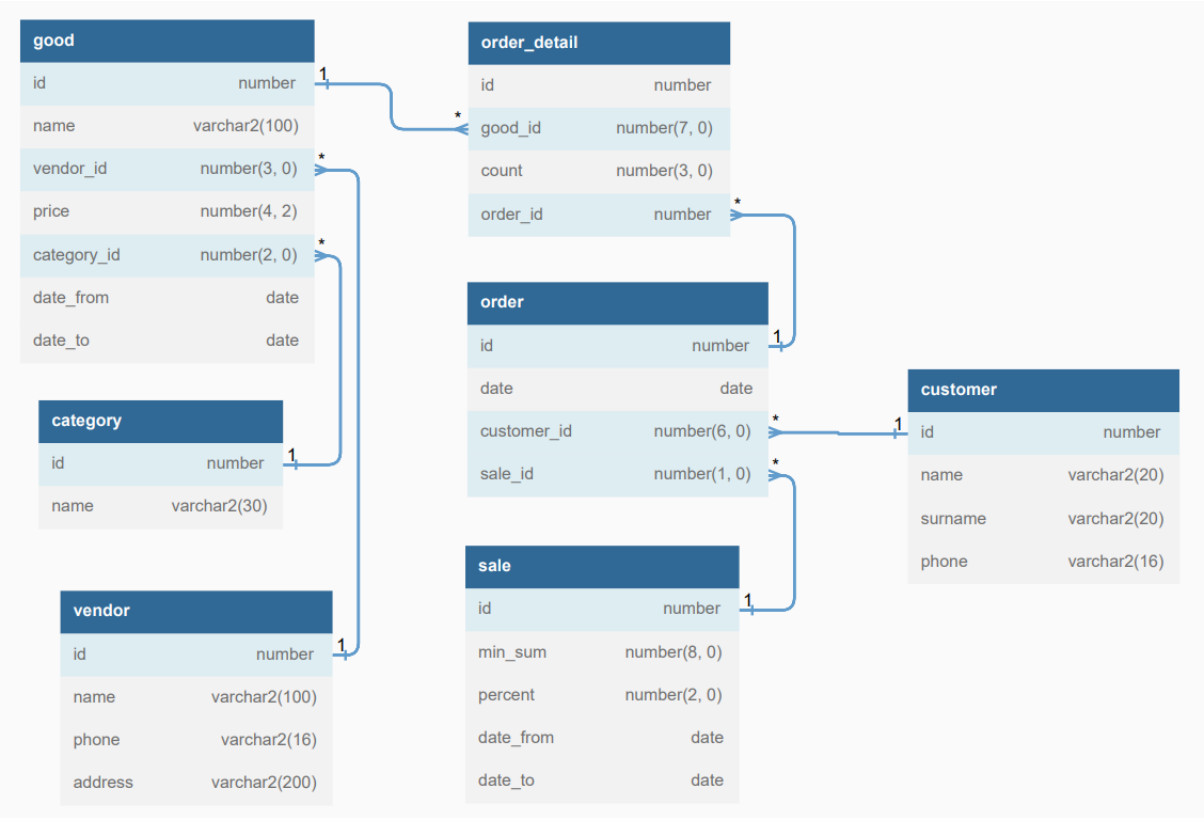
МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский ядерный университет «МИФИ»»

ЛАБОРАТОРНАЯ РАБОТА №2-3:
«Индексирование и кластеризация данных.»



Выполнила студентка группы Б19-515
Щербакова Александра

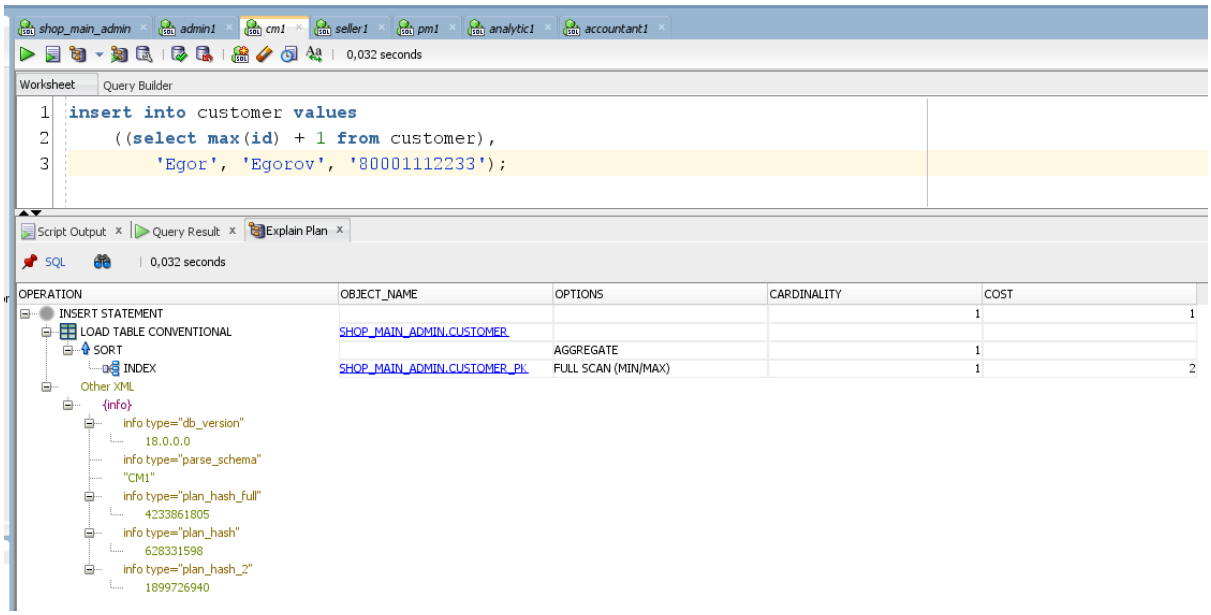
Москва, 2023 г.

1. План выполнения запросов.

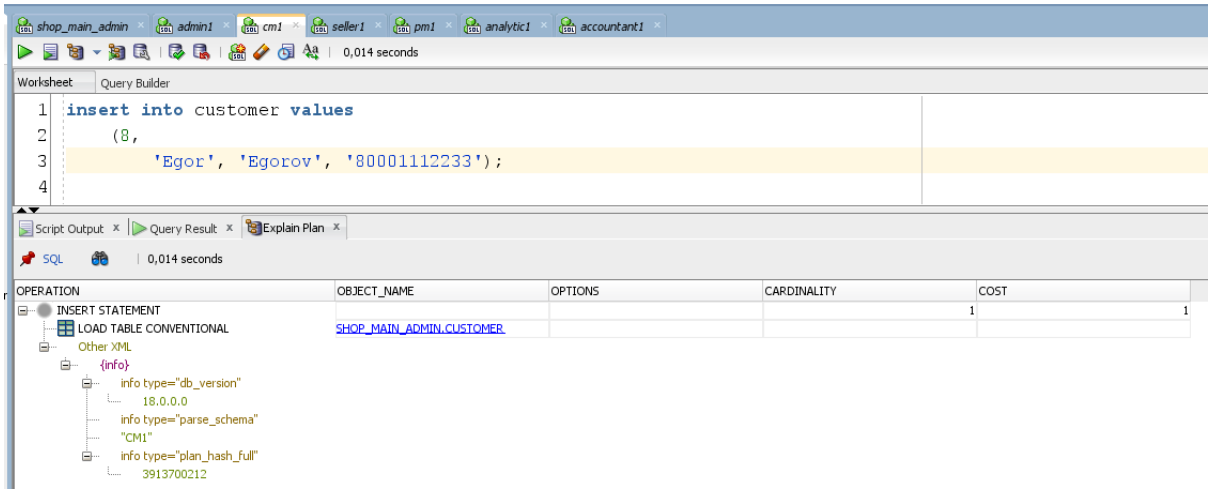
Рассмотрим 5 типичных для данной БД запросов – на чтение/добавление/изменение таблиц. Для каждого проанализируем план запроса, будем двигаться от простых запросов к более сложным. Код всех запросов в приложении А.

Примечание: вручную индексы не создавались, но Oracle автоматические создает индекс при назначении столбца в качестве первичного ключа. По умолчанию применяется схема индексации B-дерево.

1) Добавление нового покупателя (client_manager).

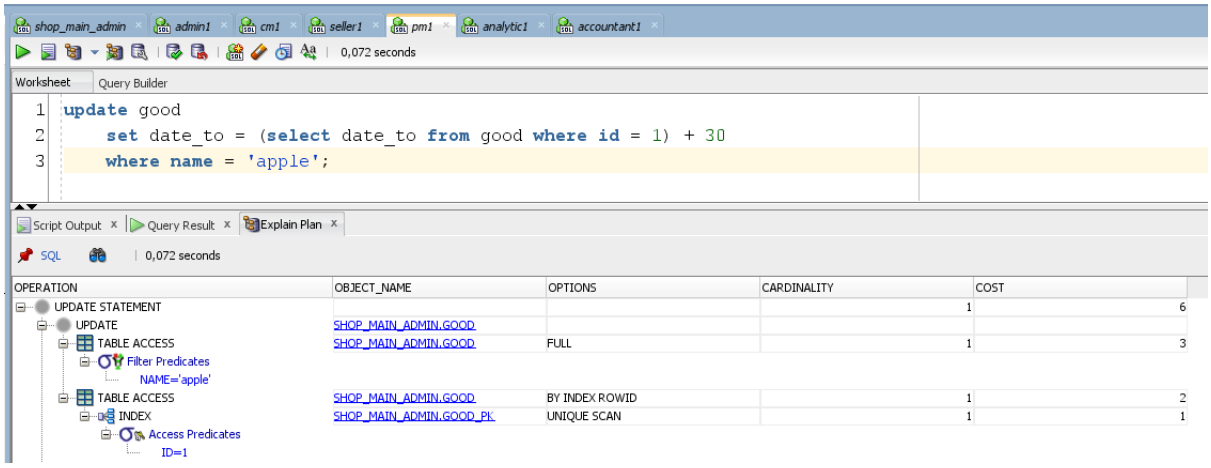


Для сравнения выполним тот же запрос, но id добавляемого покупателя захардкодим:



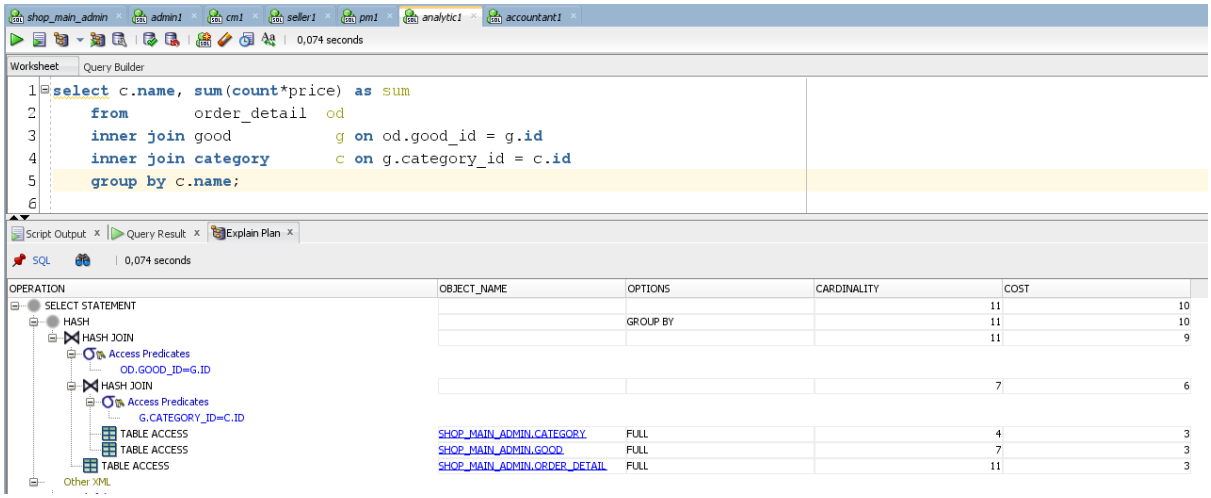
Видим, что в первом случае выполняется полное сканирование (сортировка) для поиска наибольшего значение id что увеличивает стоимость операции на 2.
Индекс используется только в первом случае (shop_main_admin.customer_pk – это первичный ключ таблицы, собственно, рассматриваемый индекс).

2) Изменить информацию о товаре, например, продлить срок действия цены (prch_manager)



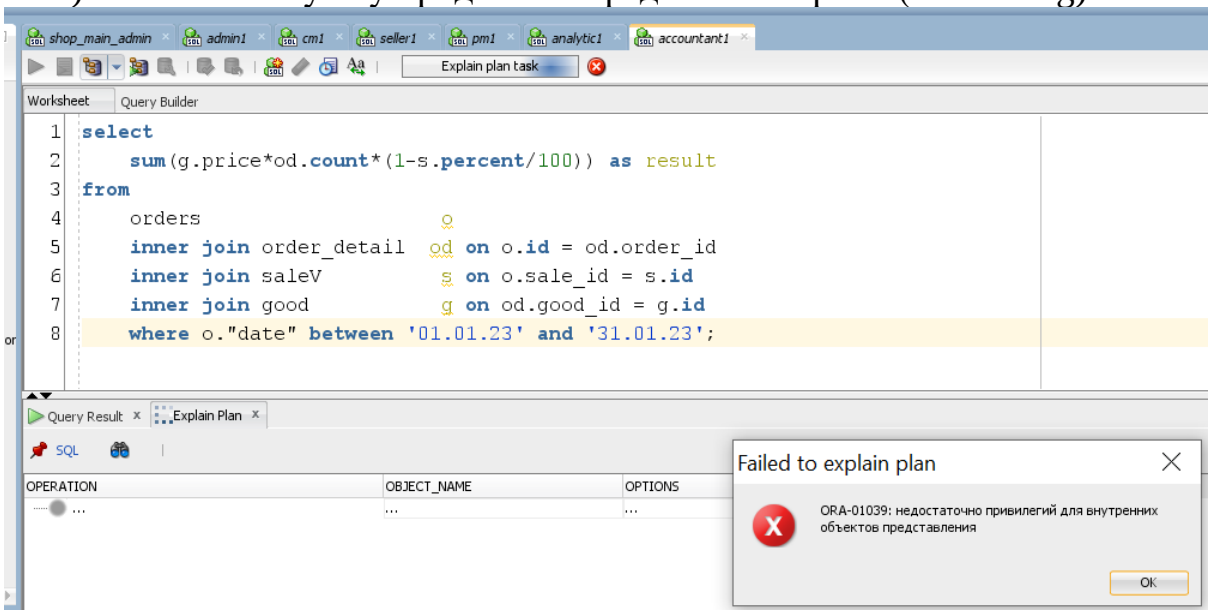
Сначала выполняется уникальное сканирование по индексу, затем доступ к таблице через операцию index rowid с использованием полученного rowed. Далее доступ через полное сканирование, так как столбец name не проиндексирован.

3) Вычислить сумму проданных товаров из каждой категории (analytic)

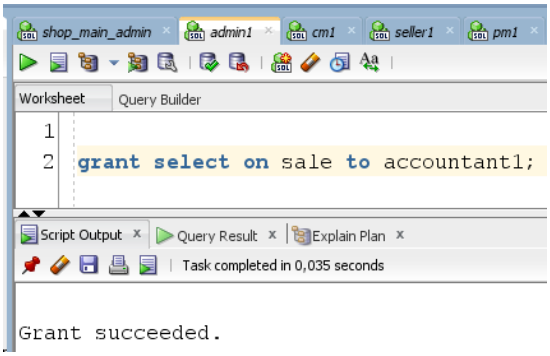


Видим трижды полное сканирование таблиц. Доступов по индексу нет, что логично исходя из текста запроса.

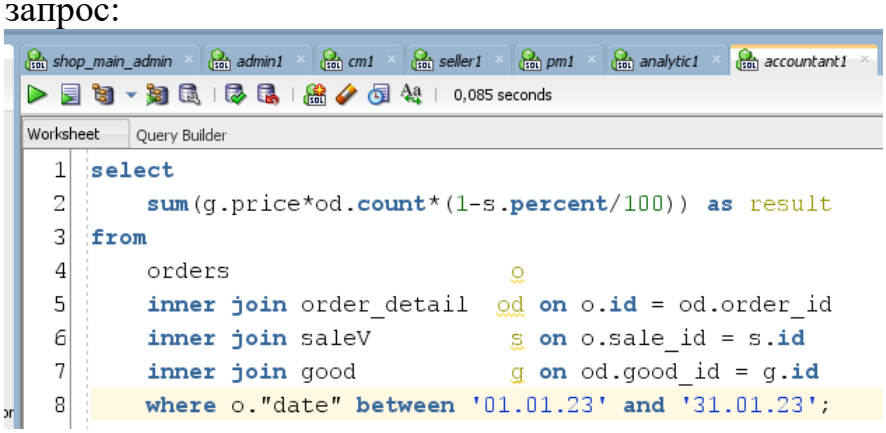
4) Вычислить сумму продаж за определенное время (accounting)



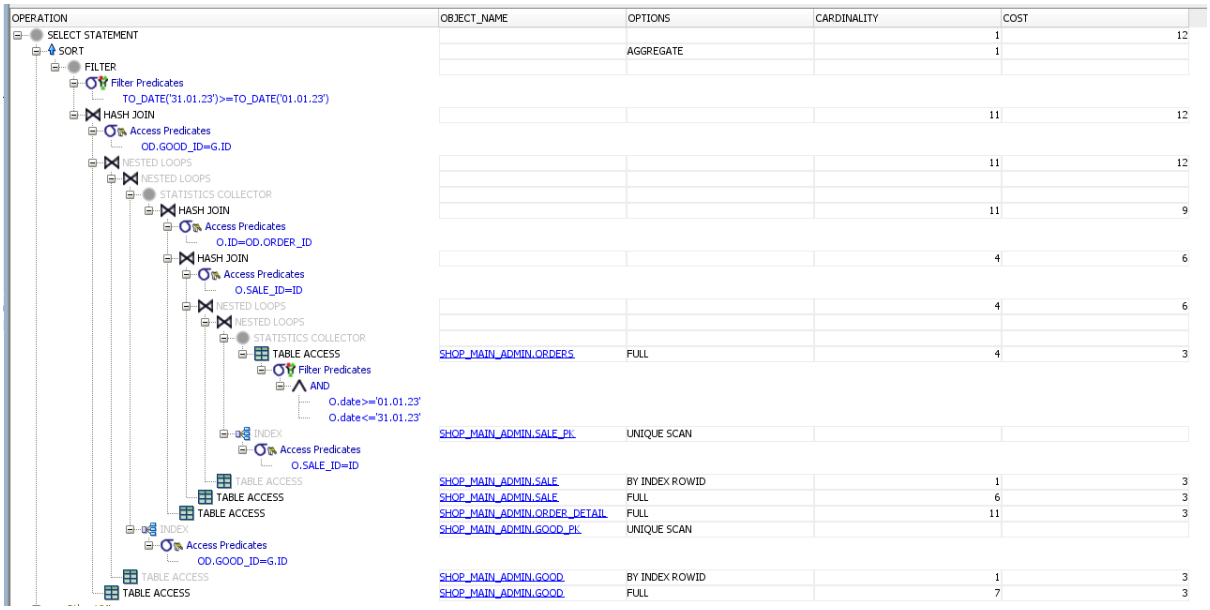
Посмотреть план запроса не получилось из-за недостатка привилегий. Проблему можно решить, либо выдав пользователю привилегии на всю таблицу sale (вместо представления saleV), либо открыв план запроса от имени админа.
Пойдем первым путем:



Итак, сам запрос:



И его план:

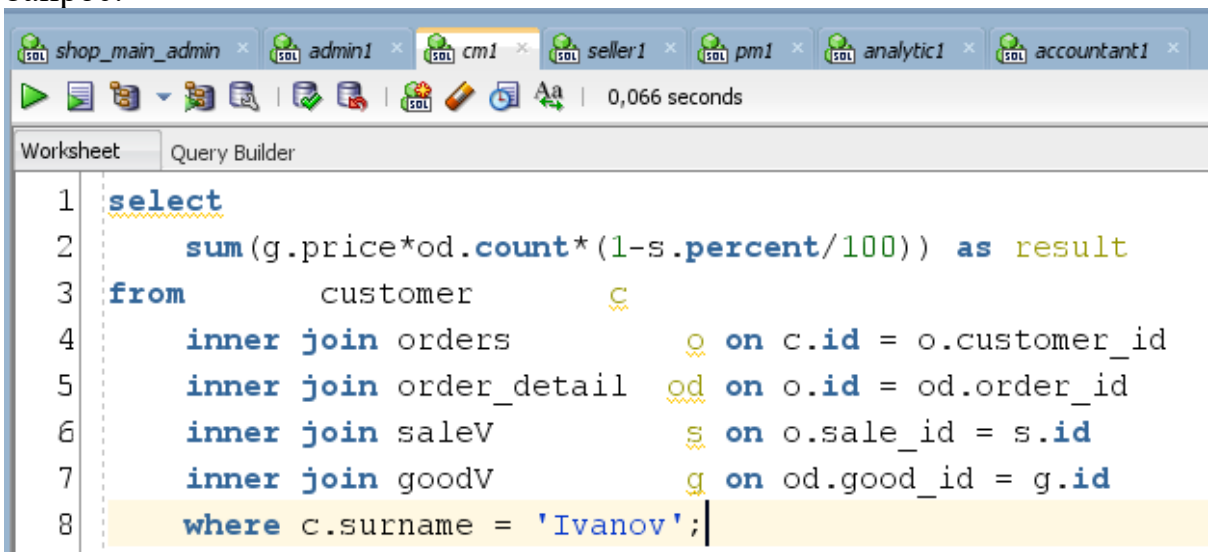


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	12
SORT		AGGREGATE	1	
FILTER				
Filter Predicates				
TO_DATE('31.01.23')>=TO_DATE('01.01.23')				
HASH JOIN			11	12
Access Predicates				
OD.GOOD_ID=G.ID				
NESTED LOOPS			11	12
NESTED LOOPS				
STATISTICS COLLECTOR				
HASH JOIN			11	9
Access Predicates				
O.ID=OD.ORDER_ID				
HASH JOIN			4	6
Access Predicates				
O.SALE_ID=ID				
NESTED LOOPS			4	6
NESTED LOOPS				
STATISTICS COLLECTOR				
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDERS	FULL	4	3
Filter Predicates				
AND				
O.date>='01.01.23'				
O.date<='31.01.23'				
INDEX	SHOP_MAIN_ADMIN.SALE_PK	UNIQUE SCAN		
Access Predicates				
O.SALE_ID=ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	BY INDEX ROWID	1	3
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	FULL	6	3
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDER_DETAIL	FULL	11	3
INDEX	SHOP_MAIN_ADMIN.GOOD_PK	UNIQUE SCAN		
Access Predicates				
OD.GOOD_ID=G.ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	BY INDEX ROWID	1	3
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	FULL	7	3

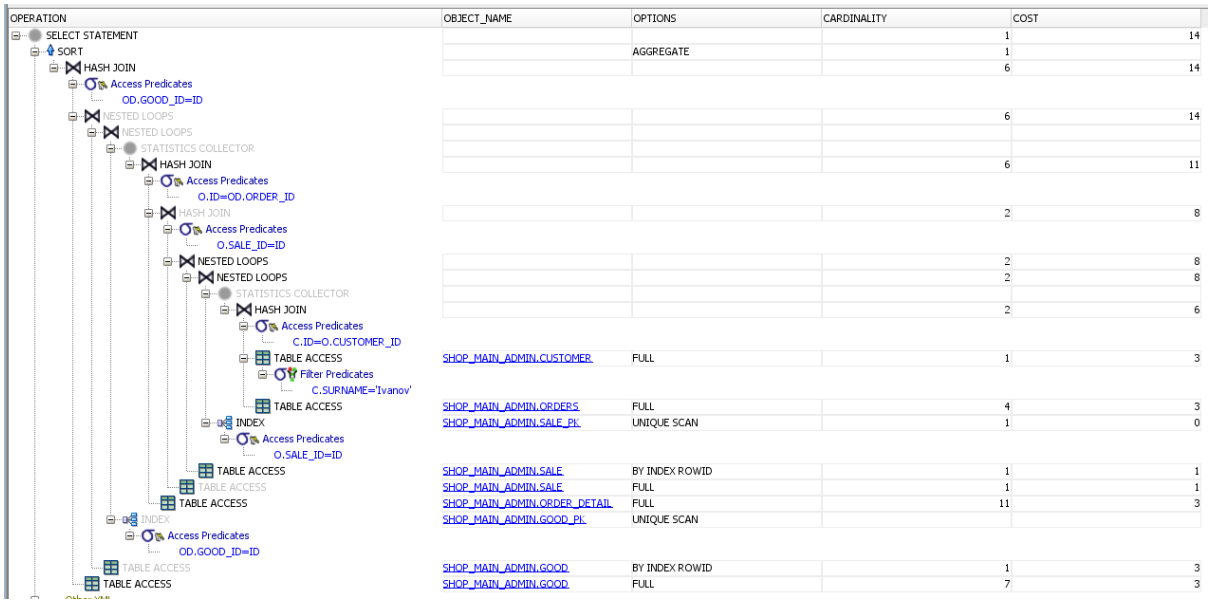
Видим дважды использование индексов в операциях index *table_name* unique scan, соответствующие строчкам 6 и 7 запроса.
Странно, что не показаны значения cardinality и cost для строк index unique scan. С чем это связано – не смогла разобраться.

5) Вычислить сумму покупок клиента (client_manager)

Запрос:



План запроса:

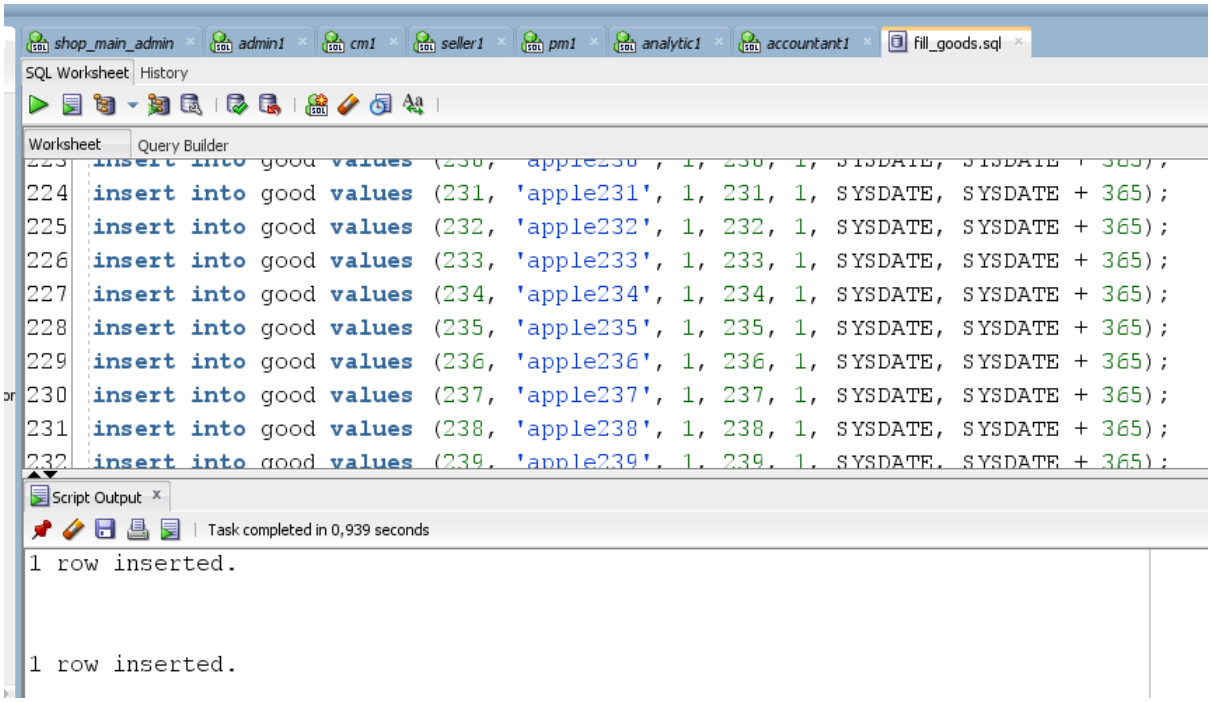


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	14
SORT		AGGREGATE	1	
HASH JOIN			6	14
Access Predicates				
OD.GOOD_ID=ID				
NESTED LOOPS			6	14
NESTED LOOPS				
STATISTICS COLLECTOR				
HASH JOIN			6	11
Access Predicates				
O.ID=OD.ORDER_ID				
HASH JOIN			2	8
Access Predicates				
O.SALE_ID=ID				
NESTED LOOPS			2	8
NESTED LOOPS			2	8
STATISTICS COLLECTOR				
HASH JOIN			2	6
Access Predicates				
C.ID=O.CUSTOMER_ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.CUSTOMER	FULL	1	3
Filter Predicates				
C.SURNAME='Ivanov'				
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDERS	FULL	4	3
INDEX	SHOP_MAIN_ADMIN.SALE_PK	UNIQUE SCAN	1	0
Access Predicates				
O.SALE_ID=ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	BY INDEX ROWID	1	1
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	FULL	1	1
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDER_DETAIL	FULL	11	3
INDEX	SHOP_MAIN_ADMIN.GOOD_PK	UNIQUE SCAN		
Access Predicates				
OD.GOOD_ID=ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	BY INDEX ROWID	1	3
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	FULL	7	3

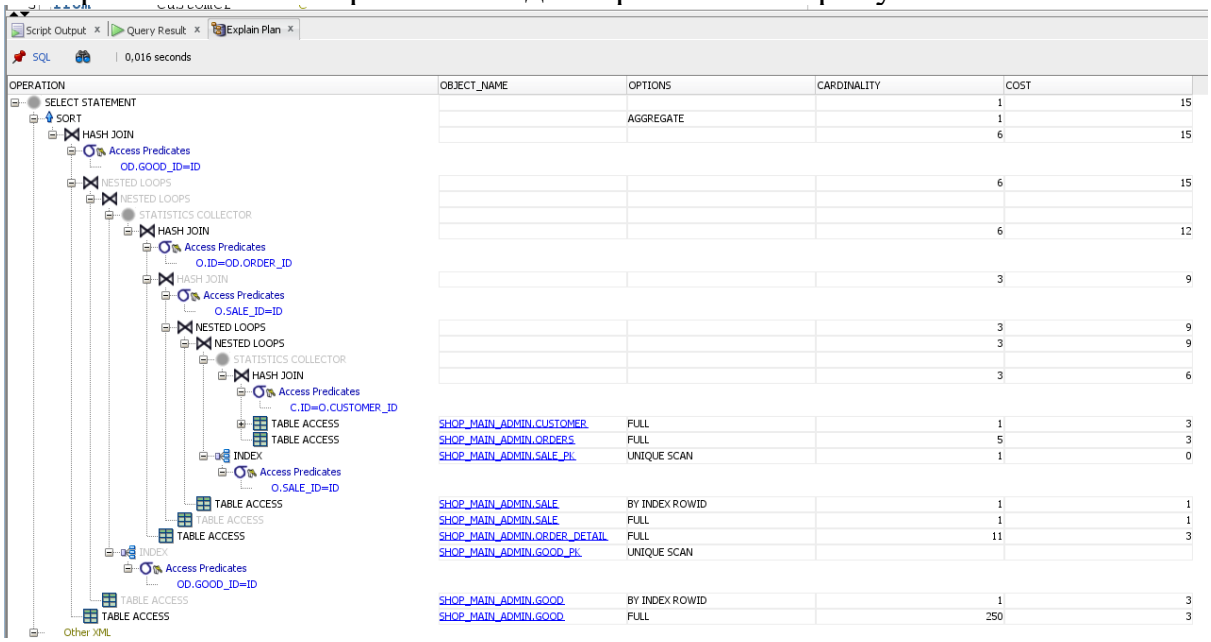
Как и в прошлом запросе, видим два использования индексов в операциях index *table_name* unique scan. Значения cardinality и cost для этой операции 1 и 0 соответственно, что делает ее самой дешевой из всех операций данного запроса.

2. Попытка обоснования полезности индексов.

1) Тест 1 – с индексом / без индекса для первичного ключа.
   В примерах выше таблицы заполнены данными максимум на 20 строк, так что разница в производительности между поиском по индексу и полным сканированием таблицы не видна. Попробуем заполнить таблицу good до 250 строк, сгенерировав однотипные значения на питоне.

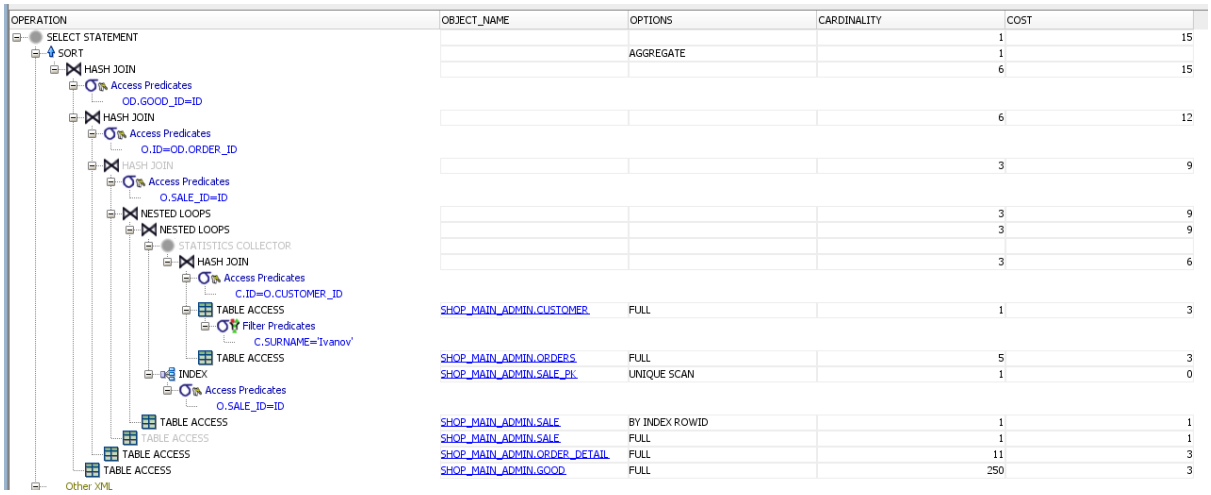


Теперь выполним запрос #5 с индексированным атрибутом id:



В самое нижней строке видим новое значение cardinality = 250, однако cost =3 не изменилось. Видимо, нужен бОльший размер таблиц, чтобы увидеть здесь разницу.

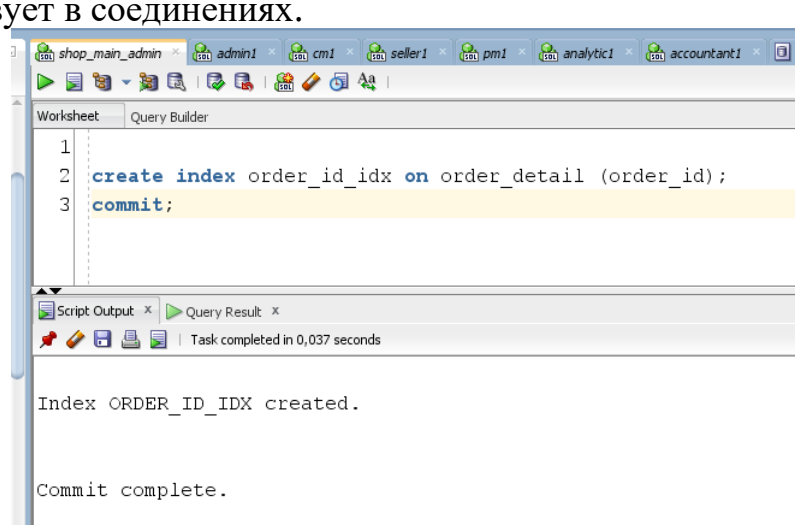
Теперь удалим индекс и еще раз посмотрим на план запроса.



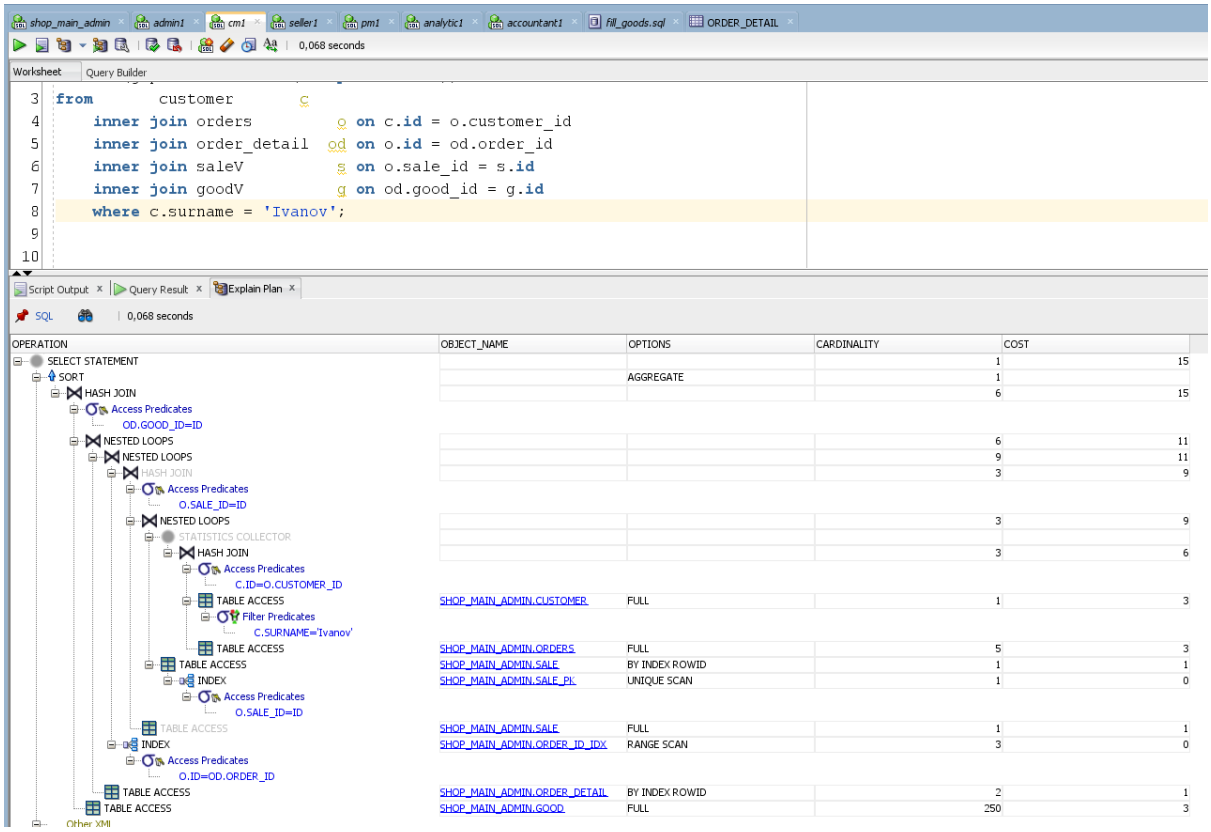
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	15
SORT		AGGREGATE	1	
HASH JOIN			6	15
Access Predicates				
OD.GOOD_ID=ID				
HASH JOIN			6	12
Access Predicates				
O.ID=OD.ORDER_ID				
HASH JOIN			3	9
Access Predicates				
O.SALE_ID=ID				
NESTED LOOPS			3	9
NESTED LOOPS			3	9
STATISTICS COLLECTOR				
HASH JOIN			3	6
Access Predicates				
C.ID=O.CUSTOMER_ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.CUSTOMER	FULL	1	3
Filter Predicates				
C.SURNAME='Ivanov'				
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDERS	FULL	5	3
INDEX	SHOP_MAIN_ADMIN.SALE_PK	UNIQUE SCAN	1	0
Access Predicates				
O.SALE_ID=ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	BY INDEX ROWID	1	1
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	FULL	1	1
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDER_DETAIL	FULL	11	3
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	FULL	250	3
Other XML				

Две нижние строки с таблицей good просто пропали.

2) Тест 2 – с индексом / без индекса для не первичного ключа
   Попробуем обнаружить разницу другим путем - добавим индекс в атрибут order_detail.order_id, так как этот столбец очень часто участвует в соединениях.



Выполним все тот же запрос #5:



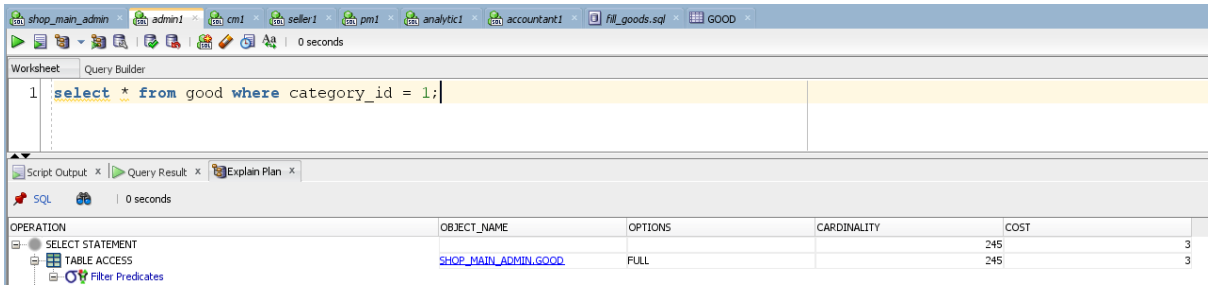
```
from       customer       c
    inner join orders         o  on c.id = o.customer_id
    inner join order_detail   od on o.id = od.order_id
    inner join saleV          s  on o.sale_id = s.id
    inner join goodV          g  on od.good_id = g.id
    where c.surname = 'Ivanov';
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	15
SORT		AGGREGATE	1	
HASH JOIN			6	15
Access Predicates OD.GOOD_ID=ID				
NESTED LOOPS			6	11
NESTED LOOPS			9	11
HASH JOIN			3	9
Access Predicates O.SALE_ID=ID				
NESTED LOOPS			3	9
STATISTICS COLLECTOR				
HASH JOIN			3	6
Access Predicates C.ID=O.CUSTOMER_ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.CUSTOMER	FULL	1	3
Filter Predicates C.SURNAME='Ivanov'				
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDERS	FULL	5	3
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	BY INDEX ROWID	1	1
INDEX	SHOP_MAIN_ADMIN.SALE_PK	UNIQUE SCAN	1	0
Access Predicates O.SALE_ID=ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.SALE	FULL	1	1
INDEX	SHOP_MAIN_ADMIN.ORDER_ID_IDX	RANGE SCAN	3	0
Access Predicates O.ID=OD.ORDER_ID				
TABLE ACCESS	SHOP_MAIN_ADMIN.ORDER_DETAIL	BY INDEX ROWID	2	1
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	FULL	250	3

Наконец видим разницу: в строке shop_main_admin.order_detail без индекса было полное сканирование таблицы (cardinality = 11, cost = 3), а с индексом эта операция разбилась на две – index range scan (cardinality = 3, cost = 0) и table access by index rowid (cardinality = 2, cost = 1).

3) Тест 3 – с индексом в столбце, выборка большого/маленького количества данных.

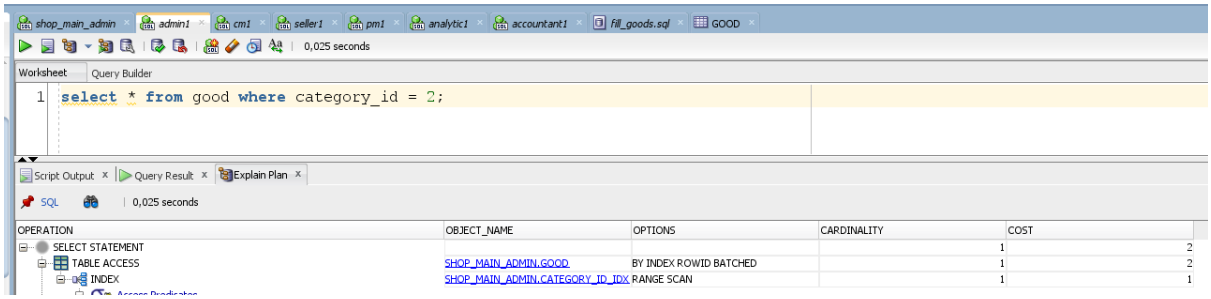
Добавим индекс столбцу good.category_id.
Таблица good на данный момент выглядит так: 245 видов яблок (category 1) и один огурец (category 2).
При выборке большого процента данных oracle автоматически применяет table access full.



```
select * from good where category_id = 1;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			245	3
TABLE ACCESS	SHOP_MAIN_ADMIN.GOOD	FULL	245	3
Filter Predicates				

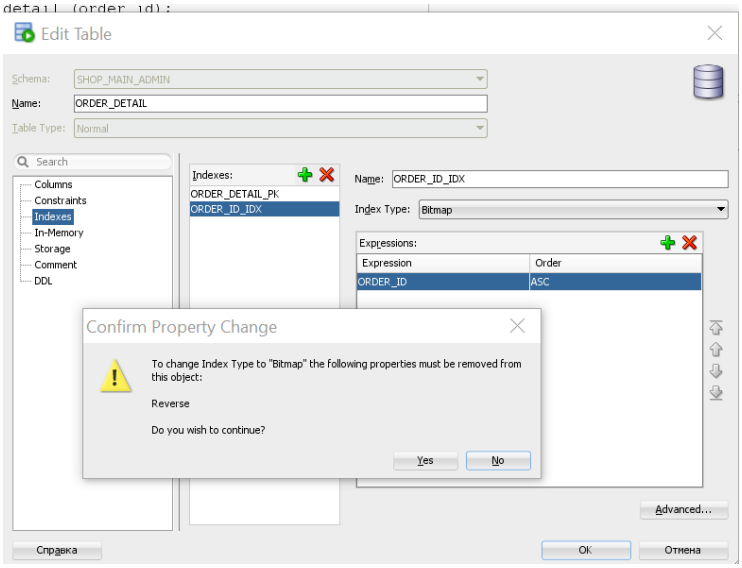
А при выборке малого процента данных – поиск по индексу.



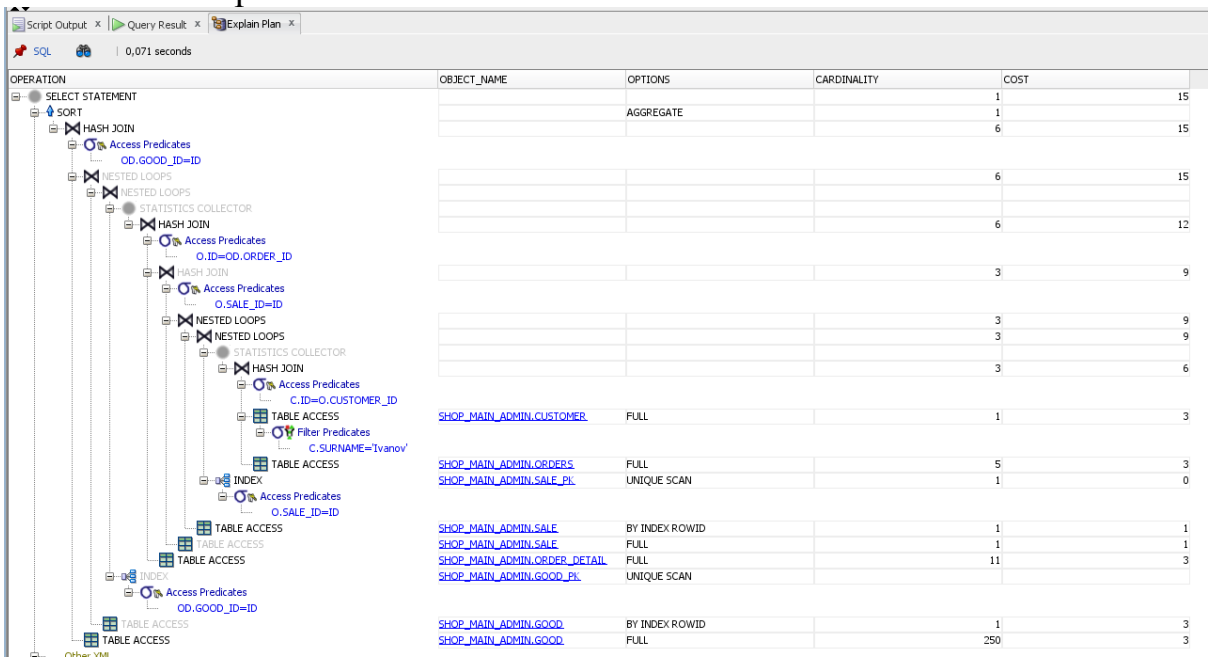
Выводы:
+ выгодно индексировать столбцы, участвующие в многотабличных операциях соединения;
+ выгодно индексировать столбцы, из которых типичные запросы извлекают маленький процент данных;
- злоупотреблять индексированием для очень маленьких таблиц не стоит, так как это будет невыгодно по памяти, и скорости работы
- для столбцов, которые часто обновляются, также не стоит применять индексирование из-за накладных расходов на изменение b-дерева

3. Индексы на основе битовых карт.

1) Тест 1 - сложный запрос с соединениями.
   В теории говорится, что индексы на основе битовых карт подходят для столбцов, содержащих сравнительно мало различных значений. Столбец order_detail.order_id должен содержать примерно на порядок меньше значений, чем всего строк в таблице order_detail. Изменим тип индекса на битовую карту:



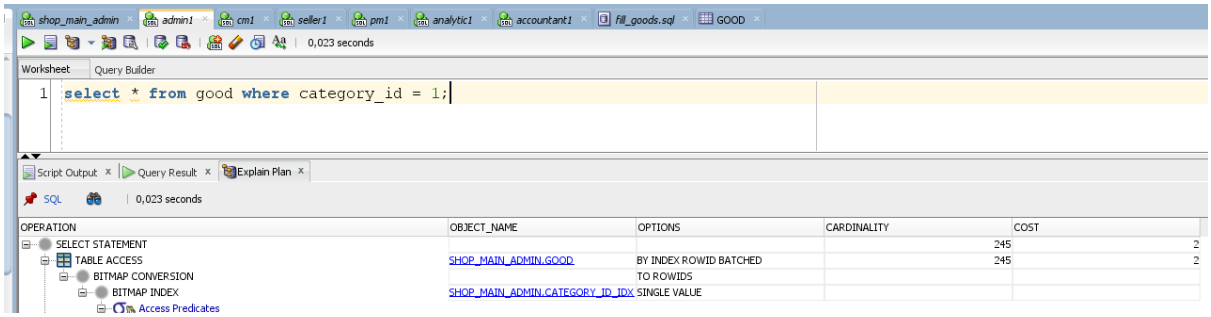
Все тот же запрос #5:



План запроса совпадает с планом, когда на атрибуте order_detail.order_id не было индекса вообще. Почему так – непонятно.

2) Тест 2 – простой запрос.

Изменим индекс good.category_id на bitmap и сравним с результатами теста 3 из части 3.

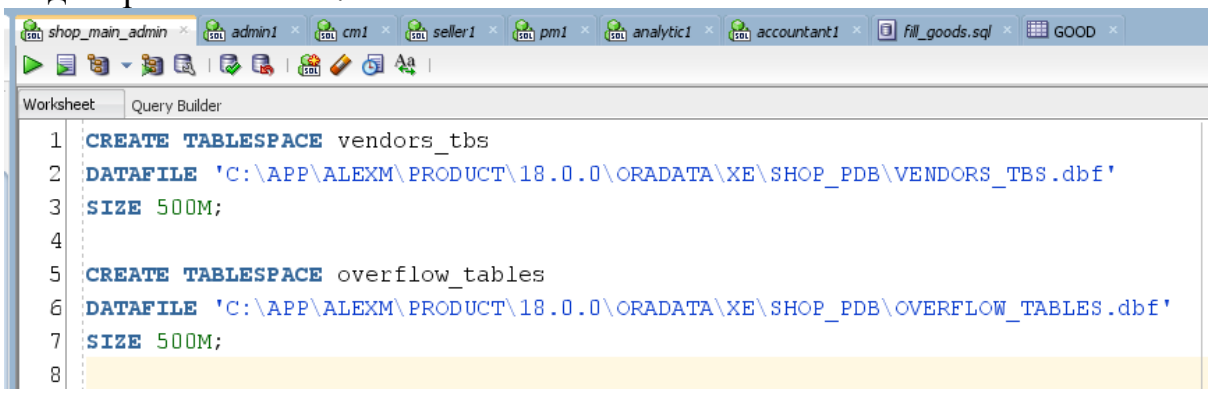


Операция access table full заменилась на три операции: вычисление одного значения индекса, перевод в rowids и вывод найденных строк. Заметим, что стоимость уменьшилась на 1.

4. Индекс-таблицы

Индекс-таблицы целесообразно применять, если выполняются следующие условия:
- таблица редко обновляется;
- в таблице очень много строк;
- часто выполняются запросы на основе первичного ключа;
- таблица не содержит данные long и lob.

Таблица vendor подходит по критериям. Создадим ее копию, но организованную по индексу (для этого также надо создать два tablespace). Код в приложении В.



```
CREATE TABLE vendors_new(
  id number(4, 0),
  name varchar2(100) NOT NULL,
  phone varchar2(16) NOT NULL,
  address varchar2(200),
  CONSTRAINT vendors_new_pk PRIMARY KEY (id))
  ORGANIZATION INDEX TABLESPACE vendors_tbs
  PCTTHRESHOLD 25
  INCLUDING name
  OVERFLOW TABLESPACE overflow_tables;
```

Вставим в таблицу 250 строк при помощи генерации строк на питоне:

```
select * from vendors_new;
```

All Rows Fetched: 250 in 0,006 seconds

	ID	NAME	PHONE	ADDRESS
217	217	vendor217	phone217	Moscow
218	218	vendor218	phone218	Moscow
219	219	vendor219	phone219	Moscow
220	220	vendor220	phone220	Moscow
221	221	vendor221	phone221	Moscow
222	222	vendor222	phone222	Moscow
223	223	vendor223	phone223	Moscow

Выполним запрос: найти кол-во позиций товаров, поставляемых каждым вендором.

Запрос для таблиц vendor и vendors_new (обе таблицы заполнены до 250 строк):

```
select v.name, count(g.vendor_id) as num_of_goods
from vendor  v
    inner join good g on v.id = g.vendor_id
    where g.vendor_id = v.id
group by v.name;
```

All Rows Fetched: 5 in 0,001 seconds

	NAME	NUM_OF_GOODS
1	chocolate vendor 2	1
2	water vendor 1	2
3	fruit vendor	245
4	fruit and vegetables vendor	1
5	chocolate vendor 1	1

```
select v.name, count(g.vendor_id) as num_of_goods
from vendors_new  v
    inner join good g on v.id = g.vendor_id
    where g.vendor_id = v.id
group by v.name;
```

All Rows Fetched: 5 in 0,001 seconds

	NAME	NUM_OF_GOODS
1	vendor2	1
2	vendor4	1
3	vendor3	1
4	fruit vendor	245
5	vendor5	2

План запроса для vendor:

OBJECT_NAME	OPTIONS	CARDINALITY	COST
		5	7
	GROUP BY	5	7
		250	6
SHOP_MAIN_ADMIN.VENDOR	BY INDEX ROWID	5	2
SHOP_MAIN_ADMIN.VENDOR_PK	FULL SCAN	5	1
	JOIN	250	4
SHOP_MAIN_ADMIN.GOOD	FULL	250	3

План запроса для vendors_new:

OBJECT_NAME	OPTIONS	CARDINALITY	COST
		5	10
	GROUP BY	5	10
		5	9
		5	9
SYS.VW_GBF_7		5	4
	GROUP BY	5	4
SHOP_MAIN_ADMIN.GOOD	FULL	250	3
VENDORS_NEW_PK	UNIQUE SCAN	1	0
VENDORS_NEW_PK	FAST FULL SCAN	1	0

Видим, что таблица без индексов отработала быстрее. В теории на бОльшем объеме данных индекс-таблица должна выигрывать: на скрине видим операции unique scan и fast full scan, которые должны быть предпочтительнее.

5. Кластеризация

Для кластеризации выгодно выбирать таблицы с совпадающими столбцами, которые часто приходится соединять. Скорость доступа связанных таблиц возрастет, но при этом уменьшится производительность оператора insert.

В рассматриваемой модели базы данных нет таблиц с совпадающими столбцами, а в задании лабораторной работы не сказано создавать новые таблицы, если существующие не подходят для кластеризации. Что ж, придется пропустить 6 пункт.

6. Хеш-кластер.

В хеш-кластер целесообразно помещать таблицу(ы), если выборки в основном производятся по одному и тому же столбцу. Простейший случай – одна таблица. Среди имеющихся таблиц для этого подойдет, например, vendor.

Создадим новое табличное пространство, хеш-кластер и новую таблицу vendors_hash. Скопируем в нее все строки из vendor. Код в приложении С.

```
CREATE CLUSTER vendor_hash_cluster (
    id number(4, 0),
    name varchar2(100),
    phone varchar2(16),
    address varchar2(200)
    )
hash is id HASHKEYS 1024
size 1024 single table
tablespace hash_cluster;

CREATE TABLE vendors_hash (
  id number(4, 0),
  name varchar2(100) NOT NULL,
  phone varchar2(16) NOT NULL,
  address varchar2(200),
  CONSTRAINT vendors_hash_pk PRIMARY KEY (id)
)
CLUSTER vendor_hash_cluster (id, name, phone, address);
```

Выполним запрос из части 4 отчета для обеих таблиц и сравним планы запросов.
Сам запрос:

```
select v.name, count(g.vendor_id) as num_of_goods
from vendor  v
    inner join good g on v.id = g.vendor_id
    where g.vendor_id = v.id
group by v.name;



select v.name, count(g.vendor_id) as num_of_goods
from vendors_hash  v
    inner join good g on v.id = g.vendor_id
    where g.vendor_id = v.id
group by v.name;
```

План для vendor:

OBJECT_NAME	OPTIONS	CARDINALITY	COST
		250	7
	GROUP BY	250	7
		250	6
SHOP_MAIN_ADMIN.VENDOR	FULL	250	3
SHOP_MAIN_ADMIN.GOOD	FULL	250	3

План для vendors_hash:

OBJECT_NAME	OPTIONS	CARDINALITY	COST
		1	4
	GROUP BY	1	4
		1	3
		250	3
SHOP_MAIN_ADMIN.GOOD	FULL	250	3
VENDORS_HASH_PK	UNIQUE SCAN	1	0
VENDORS_HASH	BY INDEX ROWID	1	0

Запрос в хеш-кластере работает быстрее за счет операции index uqinue scan.

7. Заключение

В данной работе рассмотрены методы оптимизации работы с таблицами в БД: индексы на основе B-деревьев и битовых карт, индекс-таблицы, хеш-кластеры. Изучены теоретические основы методов, а также экспериментальным путем получены рекомендации по их применению:

1) Все перечисленные методы оптимизации целесообразно применять только на больших таблицах, которые редко обновляются. Эти методы ускоряют выборки select, но замедляют операции вставки/удаления из-за накладных расходов.
2) Индексы может быть выгодно применять для столбцов, которые часто участвуют в многотабличных операциях соединения. Причем B-дерево используется, если извлекается малый процент данных, а BitMap, если большой (столбец содержит мало различных значений).
3) Индекс-таблица и хеш-кластер применяются, чтобы ускорить работу с таблицей, которая используется в основном для выборок по одному столбцу.

Приложение А. Типичные запросы.

```
-- 1) Добавить нового клиента (client_manager)
insert into customer values
        ((select max(id) + 1 from customer),
                'Egor', 'Egorov', '80001112233');

-- 2) Изменить информацию о товаре, например, продлить срок действия
цены (prch_manager)
update good
        set date_to = (select date_to from good where id = 1) + 30
        where name = 'apple';

-- 3) Вычислить сумму проданных товаров из каждой категории (analytic)
select c.name, sum(count*price) as sum
   from      order_detail  od
   inner join good          g on od.good_id = g.id
   inner join category      c on g.category_id = c.id
   group by c.name;

-- 4) Вычислить сумму продаж за определенное время (accounting)
select
   sum(g.price*od.count*(1-s.percent/100)) as result
from
   orders                  o
   inner join order_detail  od on o.id = od.order_id
   inner join saleV         s on o.sale_id = s.id
   inner join good          g on od.good_id = g.id
   where o."date" between '01.01.23' and '31.01.23';

-- 5) Вычислить сумму покупок клиента (client_manager)
select
        sum(g.price*od.count*(1-s.percent/100)) as result
from      customer      c
   inner join orders        o on c.id = o.customer_id
   inner join order_detail  od on o.id = od.order_id
   inner join saleV         s on o.sale_id = s.id
   inner join goodV         g on od.good_id = g.id
   where c.surname = 'Ivanov';
```

Приложение В. Создание индекс-таблицы.

```
-- create iot
CREATE TABLESPACE vendors_tbs
DATAFILE
'C:\APP\ALEXM\PRODUCT\18.0.0\ORADATA\XE\SHOP_PDB\VENDORS
_TBS.dbf'
SIZE 500M;

CREATE TABLESPACE overflow_tables
DATAFILE
'C:\APP\ALEXM\PRODUCT\18.0.0\ORADATA\XE\SHOP_PDB\OVERFLO
W_TABLES.dbf'
SIZE 500M;


CREATE TABLE vendors_new(
  id number(4, 0),
  name varchar2(100) NOT NULL,
  phone varchar2(16) NOT NULL,
  address varchar2(200),
  CONSTRAINT vendors_new_pk PRIMARY KEY (id))
  ORGANIZATION INDEX TABLESPACE vendors_tbs
  PCTTHRESHOLD 25
  INCLUDING name
  OVERFLOW TABLESPACE overflow_tables;
```

Приложение С. Создание хеш-кластера.

```
-- create hash-cluster
CREATE TABLESPACE hash_cluster
DATAFILE
'C:\APP\ALEXM\PRODUCT\18.0.0\ORADATA\XE\SHOP_PDB\HASH_CL
USTER.dbf'
SIZE 500M;

CREATE CLUSTER vendor_hash_cluster (
        id number(4, 0),
        name varchar2(100),
        phone varchar2(16),
        address varchar2(200)
        )
hash is id HASHKEYS 1024
size 1024 single table
tablespace hash_cluster;

CREATE TABLE vendors_hash (
  id number(4, 0),
  name varchar2(100) NOT NULL,
  phone varchar2(16) NOT NULL,
  address varchar2(200),
  CONSTRAINT vendors_hash_pk PRIMARY KEY (id)
)
CLUSTER vendor_hash_cluster (id, name, phone, address);

insert into vendors_hash select * from vendor;
```