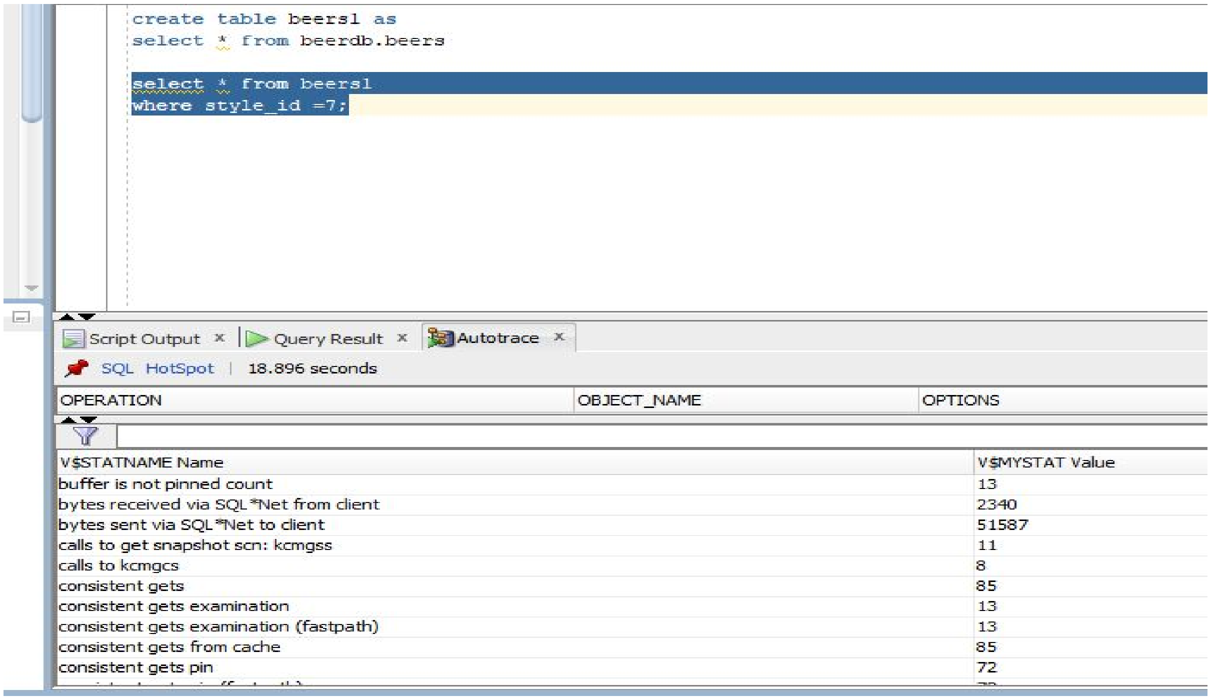
# Assignment 3 (Indexing EOTW)

## Idea 1

To begin, here are 2 simple queries to demonstrate the performance improvement as a result of indexing.

1) A query to select beers where `style_id` is 7 was just run.
When we don't index, we get the following results:

```
Consistent gets      85
Db block gets        0
```



```
create table beers1 as
select * from beerdb.beers

select * from beers1
where style_id =7;
```

18.896 seconds

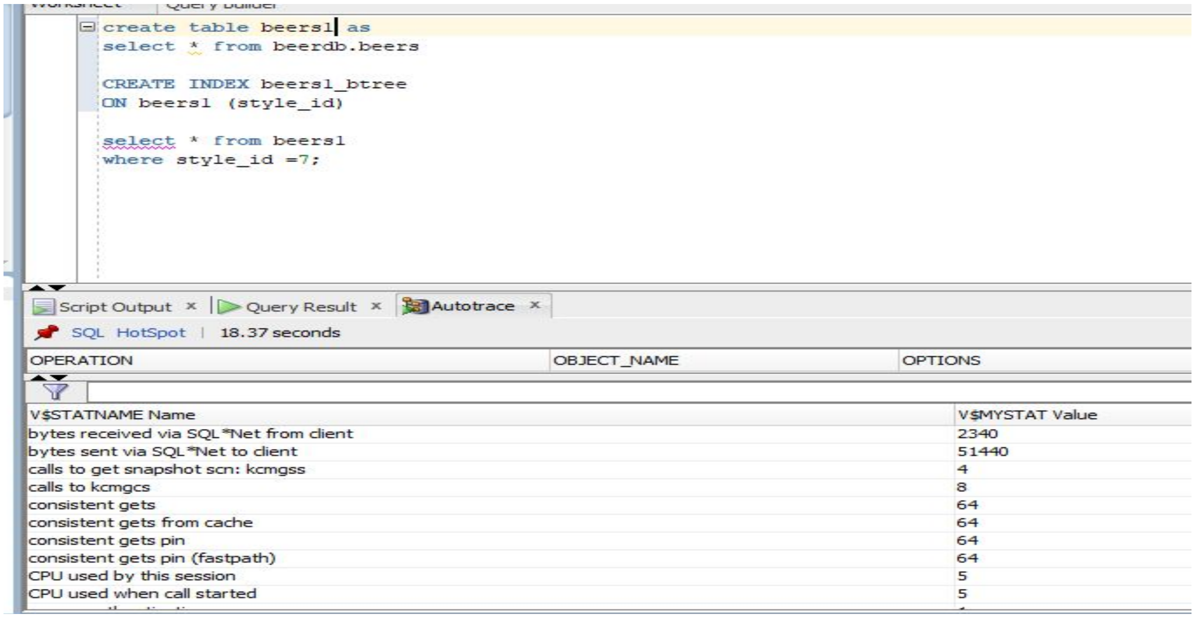
| OPERATION | OBJECT_NAME | OPTIONS |
|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 13 |
| bytes received via SQL*Net from client | 2340 |
| bytes sent via SQL*Net to client | 51587 |
| calls to get snapshot scn: kcmgss | 11 |
| calls to kcmgcs | 8 |
| consistent gets | 85 |
| consistent gets examination | 13 |
| consistent gets examination (fastpath) | 13 |
| consistent gets from cache | 85 |
| consistent gets pin | 72 |

After indexing:

```
Consistent gets      64
Db block gets        0
```

Therefore, after indexing, only 64 blocks are read to fetch the results, as compared to 85 without indexing, hence there is a noticeable improvement in performance.



```
create table beers1 as
select * from beerdb.beers

CREATE INDEX beers1_btree
ON beers1 (style_id)

select * from beers1
where style_id =7;
```

18.37 seconds

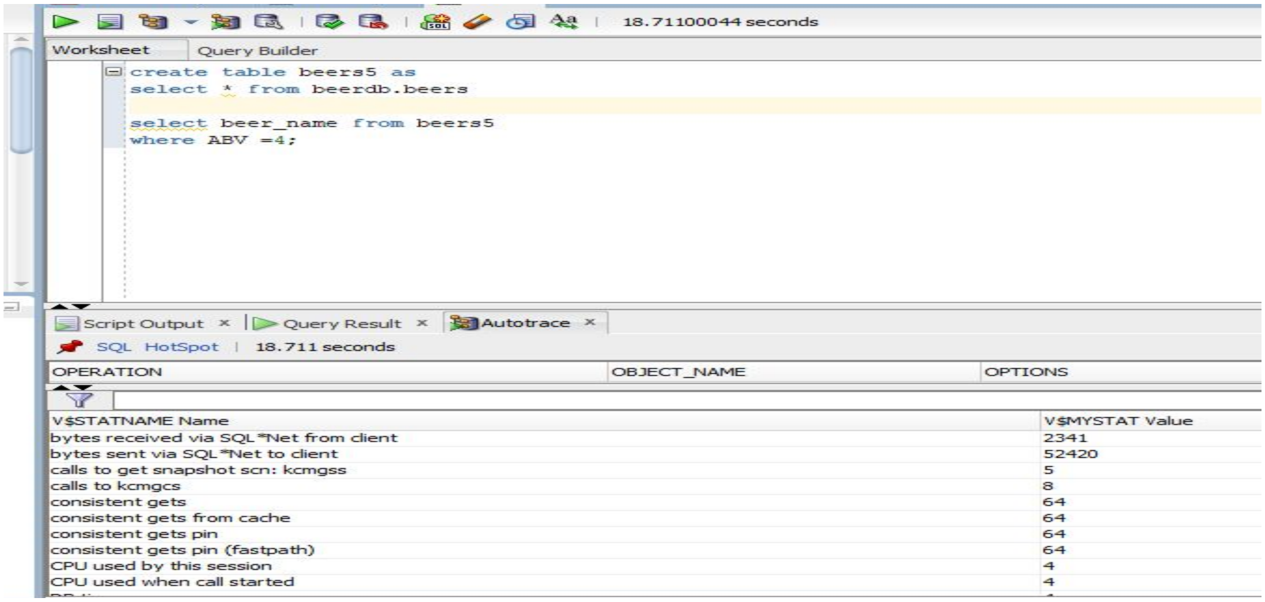
| OPERATION | OBJECT_NAME | OPTIONS |
|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2340 |
| bytes sent via SQL*Net to client | 51440 |
| calls to get snapshot scn: kcmgss | 4 |
| calls to kcmgcs | 8 |
| consistent gets | 64 |
| consistent gets from cache | 64 |
| consistent gets pin | 64 |
| consistent gets pin (fastpath) | 64 |
| CPU used by this session | 5 |
| CPU used when call started | 5 |

2) A query to select `beer_name` with `abv` = 4.

First, the results before indexing:

```
Consistent gets        64
Db block gets          0
```



```
create table beers5 as
select * from beerdb.beers

select beer_name from beers5
where ABV =4;
```

18.711 seconds

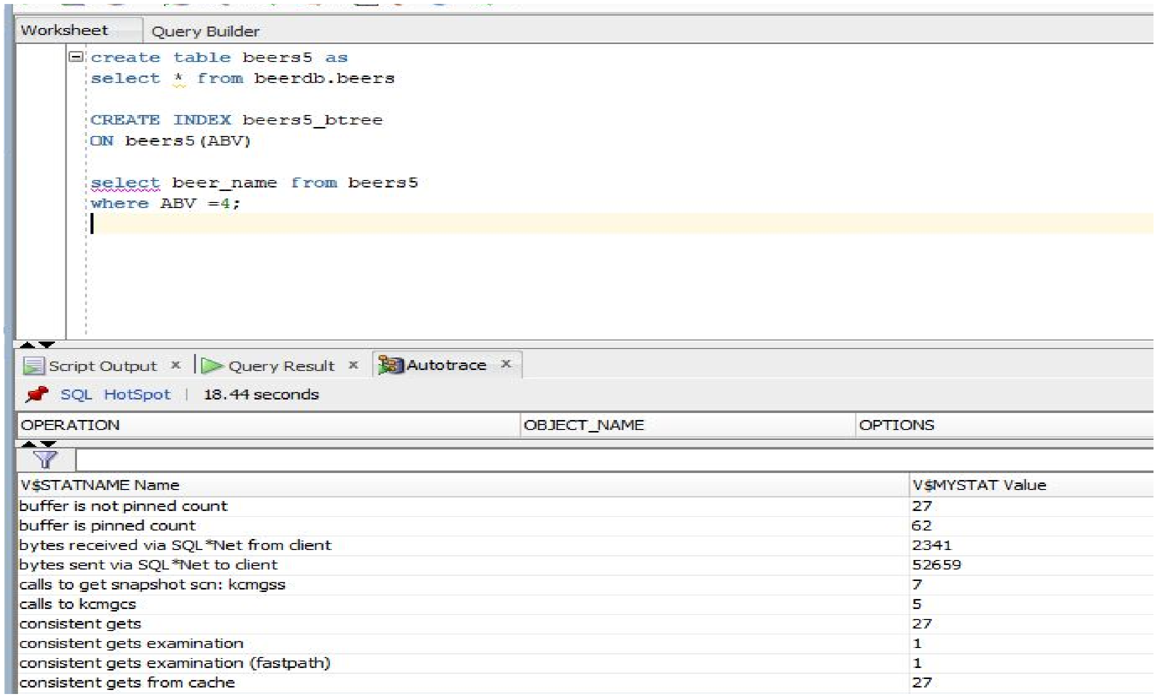
| OPERATION | OBJECT_NAME | OPTIONS |
|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2341 |
| bytes sent via SQL*Net to client | 52420 |
| calls to get snapshot scn: kcmgss | 5 |
| calls to kcmgcs | 8 |
| consistent gets | 64 |
| consistent gets from cache | 64 |
| consistent gets pin | 64 |
| consistent gets pin (fastpath) | 64 |
| CPU used by this session | 4 |
| CPU used when call started | 4 |

And then after indexing the `ABV` column:

```
Consistent gets        27
Db block gets          0
```

Hence, the performance has improved.



```
create table beers5 as
select * from beerdb.beers

CREATE INDEX beers5_btree
ON beers5(ABV)

select beer_name from beers5
where ABV =4;
```

| OPERATION | OBJECT_NAME | OPTIONS |
|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 27 |
| buffer is pinned count | 62 |
| bytes received via SQL*Net from client | 2341 |
| bytes sent via SQL*Net to client | 52659 |
| calls to get snapshot scn: kcmgss | 7 |
| calls to kcmgcs | 5 |
| consistent gets | 27 |
| consistent gets examination | 1 |
| consistent gets examination (fastpath) | 1 |
| consistent gets from cache | 27 |

Idea 2

How should you choose an index? And what makes for a good index?

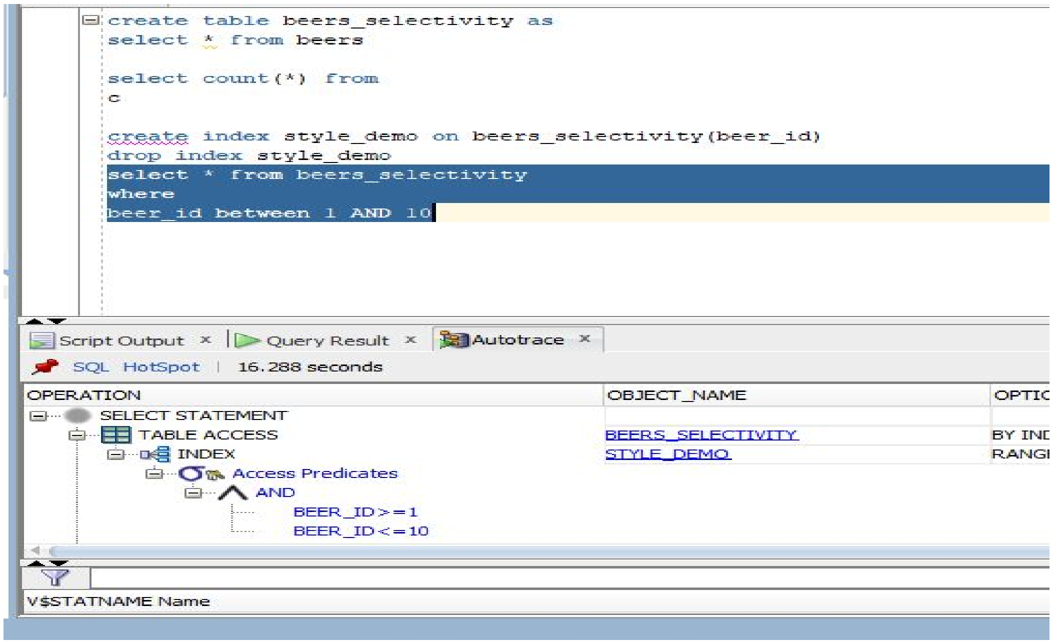
This experiment demonstrates selectivity. The idea of selectivity with regards to indexing is the number of distinct keys/total number of rows. A good index is one with a selectivity of 20%.

For example:

I have created an index on `beer_id` of `beers_selectivity` which I have derived from `beers` table using CTAS.

Is `beer_id` a good index or not?

Therefore, `beer_id` is actually distinct, so for every `beer_id`, there is a distinct value, making `beer_id` a perfect candidate for indexing.

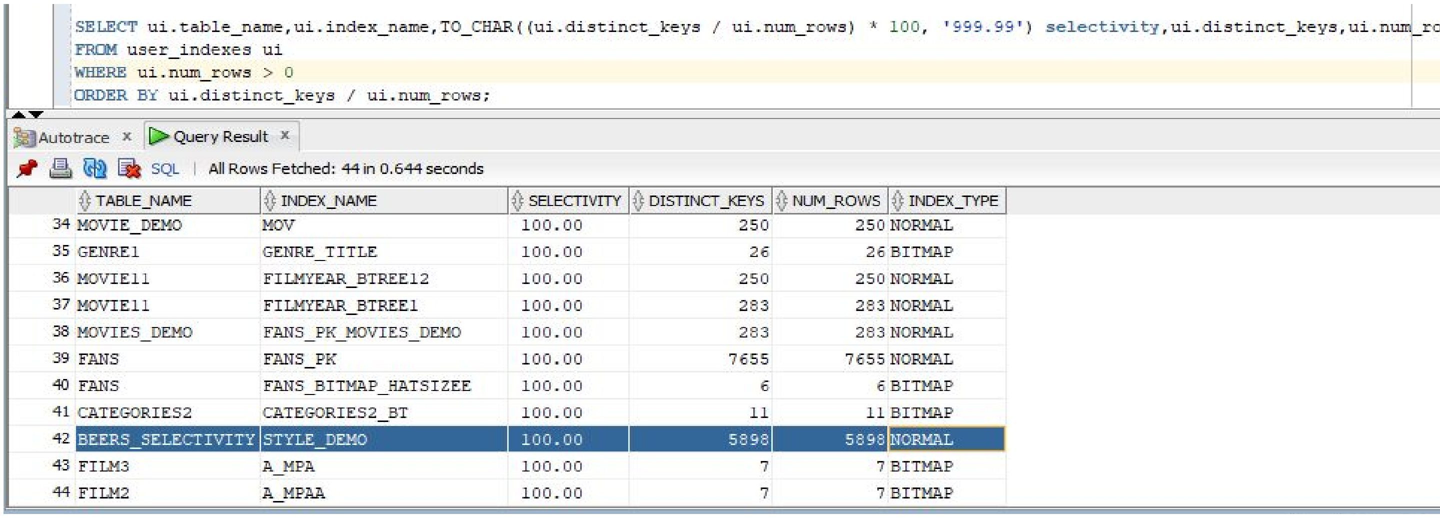


```
create table beers_selectivity as
select * from beers

select count(*) from
c

create index style_demo on beers_selectivity(beer_id)
drop index style_demo
select * from beers_selectivity
where
beer_id between 1 AND 10
```

16.288 seconds

```
OPERATION                                   OBJECT_NAME            OPTIC
SELECT STATEMENT
  TABLE ACCESS                              BEERS_SELECTIVITY      BY IND
    INDEX                                   STYLE_DEMO             RANGI
      Access Predicates
        AND
          BEER_ID>=1
          BEER_ID<=10
```

V$STATNAME Name



```
SELECT ui.table_name,ui.index_name,TO_CHAR((ui.distinct_keys / ui.num_rows) * 100, '999.99') selectivity,ui.distinct_keys,ui.num_ro
FROM user_indexes ui
WHERE ui.num_rows > 0
ORDER BY ui.distinct_keys / ui.num_rows;
```

All Rows Fetched: 44 in 0.644 seconds

| | TABLE_NAME | INDEX_NAME | SELECTIVITY | DISTINCT_KEYS | NUM_ROWS | INDEX_TYPE |
|---|---|---|---|---|---|---|
| 34 | MOVIE_DEMO | MOV | 100.00 | 250 | 250 | NORMAL |
| 35 | GENRE1 | GENRE_TITLE | 100.00 | 26 | 26 | BITMAP |
| 36 | MOVIE11 | FILMYEAR_BTREE12 | 100.00 | 250 | 250 | NORMAL |
| 37 | MOVIE11 | FILMYEAR_BTREE1 | 100.00 | 283 | 283 | NORMAL |
| 38 | MOVIES_DEMO | FANS_PK_MOVIES_DEMO | 100.00 | 283 | 283 | NORMAL |
| 39 | FANS | FANS_PK | 100.00 | 7655 | 7655 | NORMAL |
| 40 | FANS | FANS_BITMAP_HATSIZEE | 100.00 | 6 | 6 | BITMAP |
| 41 | CATEGORIES2 | CATEGORIES2_BT | 100.00 | 11 | 11 | BITMAP |
| 42 | BEERS_SELECTIVITY | STYLE_DEMO | 100.00 | 5898 | 5898 | NORMAL |
| 43 | FILM3 | A_MPA | 100.00 | 7 | 7 | BITMAP |
| 44 | FILM2 | A_MPAA | 100.00 | 7 | 7 | BITMAP |

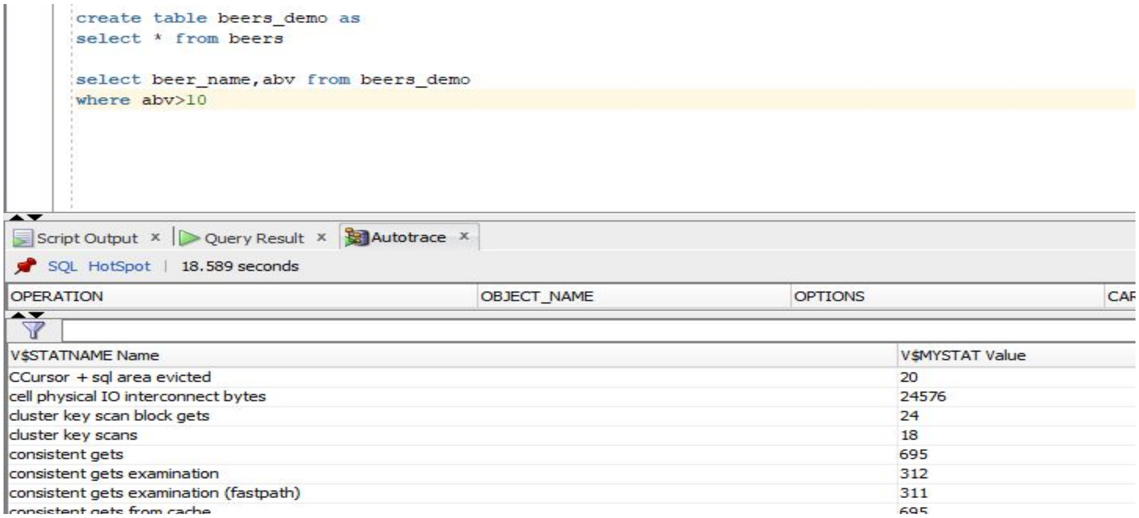
Any index that fails to have a selectivity below 20% should not be an index.

## Idea 3:
## How does indexing affect the performance of range, scan, and point queries?

1. Point queries: These were discussed in Idea 1 while demonstrating indexing.

2. Range queries:
We want to fetch the records of beers which is strong in alcohol concentration. Let's select all the beers having an `abv` above 10.

```
CREATE TABLE beers_demo as SELECT
    *
FROM beers SELECT
               beer_name,
               abv
           FROM
               beers_demo
           WHERE
               abv > 10
```

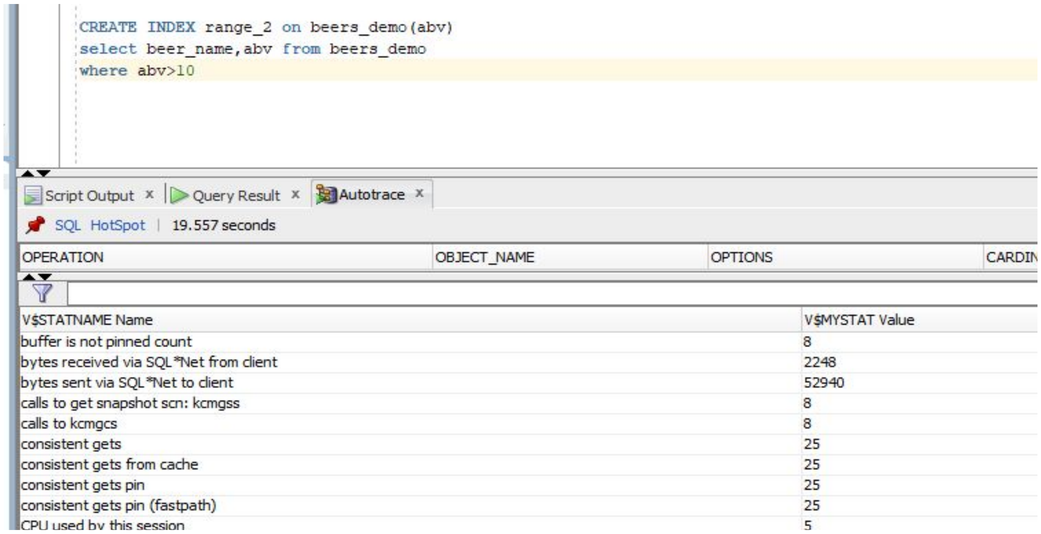


```
create table beers_demo as
select * from beers

select beer_name,abv from beers_demo
where abv>10
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| CCursor + sql area evicted | 20 |
| cell physical IO interconnect bytes | 24576 |
| cluster key scan block gets | 24 |
| cluster key scans | 18 |
| consistent gets | 695 |
| consistent gets examination | 312 |
| consistent gets examination (fastpath) | 311 |
| consistent gets from cache | 695 |

```
Consistent gets 695
Db block gets   0
```

After indexing, consistent gets are reduced to 25. Hence, it requires only 25 blocks to fetch the results.



```
CREATE INDEX range_2 on beers_demo(abv)
select beer_name,abv from beers_demo
where abv>10
```

| 19.557 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CARDIN |
|---|---|---|---|

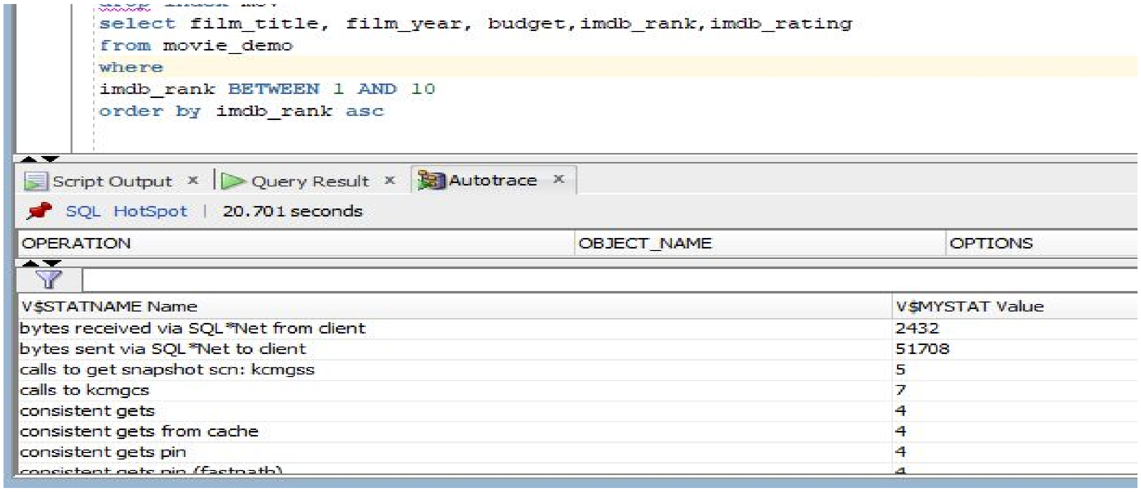
| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 8 |
| bytes received via SQL*Net from client | 2248 |
| bytes sent via SQL*Net to client | 52940 |
| calls to get snapshot scn: kcmgss | 8 |
| calls to kcmgcs | 8 |
| consistent gets | 25 |
| consistent gets from cache | 25 |
| consistent gets pin | 25 |
| consistent gets pin (fastpath) | 25 |
| CPU used by this session | 5 |

3. Indexing on range and sort queries:
The following query selects a few film details from films whose IMDb ranks are between 1 and 10, and the results are sorted by IMDb rank.

```
SELECT
    film_title,
    film_year,
    budget,
    imdb_rank,
    imdb_rating
FROM
    movie_demo
WHERE
    imdb_rank BETWEEN 1 AND 10
ORDER BY
    imdb_rank ASC
```

Before indexing, we get the following:

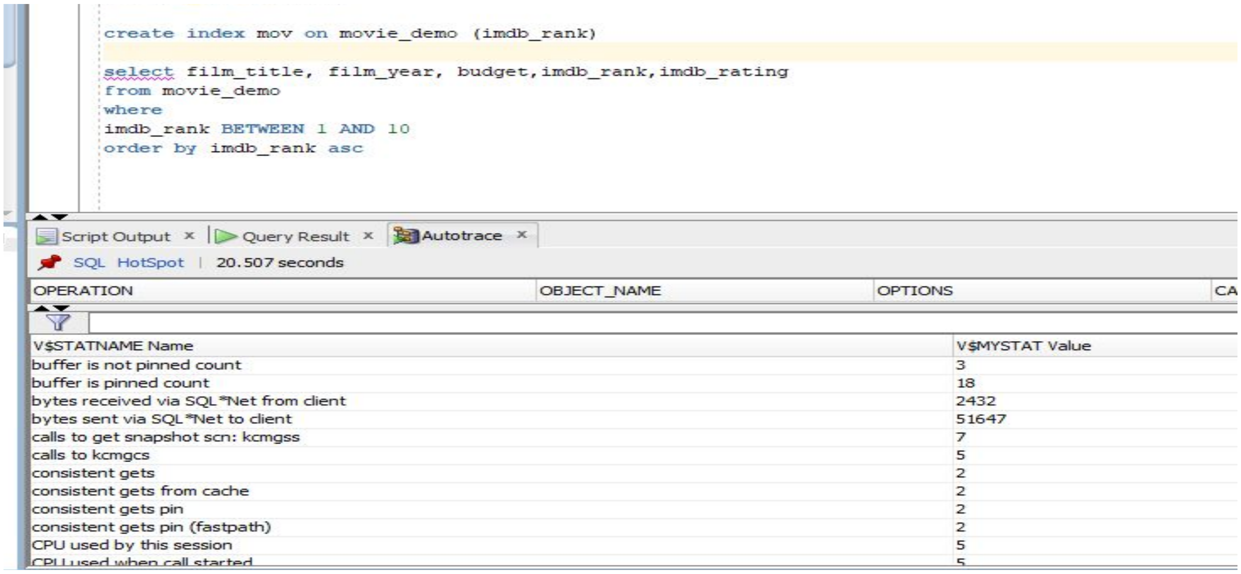
```
Consistent gets 4
Db block gets   0
```



```
select film_title, film_year, budget,imdb_rank,imdb_rating
from movie_demo
where
imdb_rank BETWEEN 1 AND 10
order by imdb_rank asc
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2432 |
| bytes sent via SQL*Net to client | 51708 |
| calls to get snapshot scn: kcmgss | 5 |
| calls to kcmgcs | 7 |
| consistent gets | 4 |
| consistent gets from cache | 4 |
| consistent gets pin | 4 |
| consistent gets pin (fastpath) | 4 |

After indexing, consistent reads gets reduced to 2 with no db block gets



```
create index mov on movie_demo (imdb_rank)

select film_title, film_year, budget,imdb_rank,imdb_rating
from movie_demo
where
imdb_rank BETWEEN 1 AND 10
order by imdb_rank asc
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 3 |
| buffer is pinned count | 18 |
| bytes received via SQL*Net from client | 2432 |
| bytes sent via SQL*Net to client | 51647 |
| calls to get snapshot scn: kcmgss | 7 |
| calls to kcmgcs | 5 |
| consistent gets | 2 |
| consistent gets from cache | 2 |
| consistent gets pin | 2 |
| consistent gets pin (fastpath) | 2 |
| CPU used by this session | 5 |
| CPU used when call started | 5 |

4. Indexing effects on grouping query:

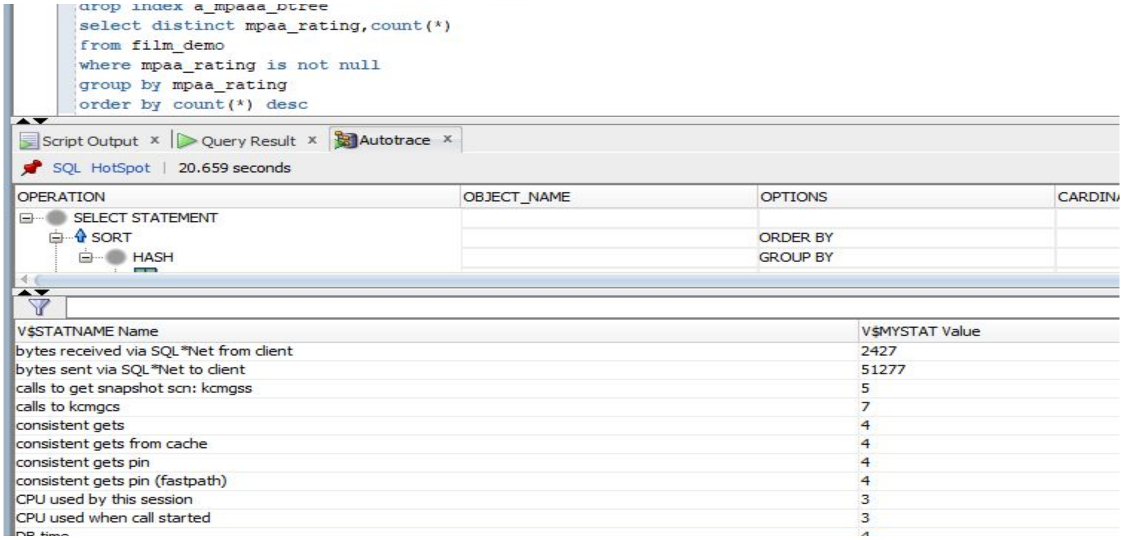
The following query counts the number of number of movies with each possible MPAA rating and groups them by rating.

```
SELECT DISTINCT
    mpaa_rating,
    COUNT(*)
FROM
    film_demo
WHERE
    mpaa_rating IS NOT NULL
GROUP BY
    mpaa_rating
```

```
ORDER BY
    COUNT(*) DESC
```

Before creating an index:

```
Consistent gets 4
Db block gets   0
```



```
drop index a_mpaaa_btree
select distinct mpaa_rating,count(*)
from film_demo
where mpaa_rating is not null
group by mpaa_rating
order by count(*) desc
```

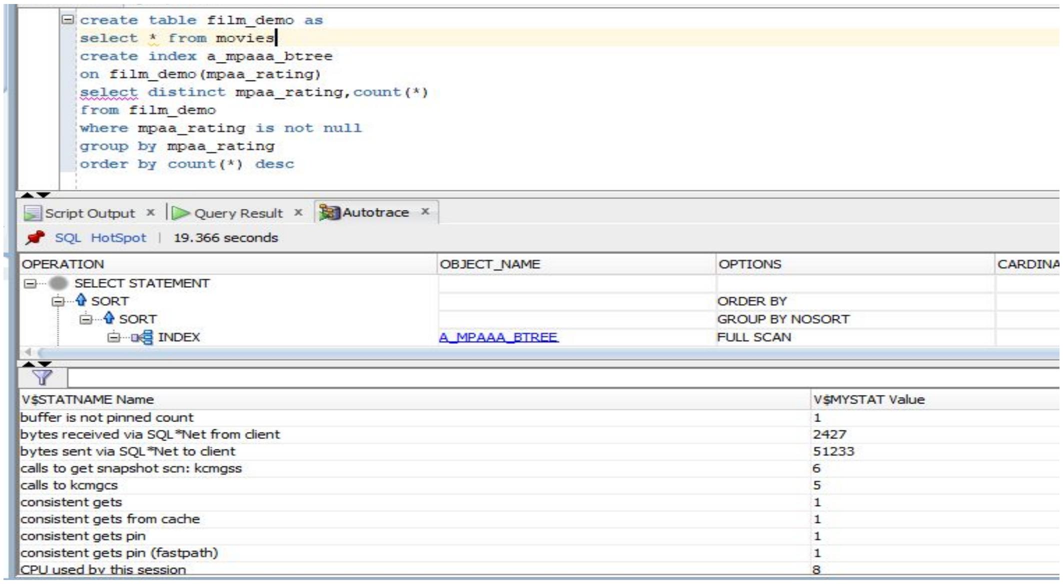
20.659 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CARDINA |
|---|---|---|---|
| SELECT STATEMENT | | | |
| SORT | | ORDER BY | |
| HASH | | GROUP BY | |

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2427 |
| bytes sent via SQL*Net to client | 51277 |
| calls to get snapshot scn: kcmgss | 5 |
| calls to kcmgcs | 7 |
| consistent gets | 4 |
| consistent gets from cache | 4 |
| consistent gets pin | 4 |
| consistent gets pin (fastpath) | 4 |
| CPU used by this session | 3 |
| CPU used when call started | 3 |

After creating an index on `mpaa_rating`, the performance improved.

```
Consistent gets 1
Db block gets   0
```



```
create table film_demo as
select * from movies
create index a_mpaaa_btree
on film_demo(mpaa_rating)
select distinct mpaa_rating,count(*)
from film_demo
where mpaa_rating is not null
group by mpaa_rating
order by count(*) desc
```

19.366 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CARDINA |
|---|---|---|---|
| SELECT STATEMENT | | | |
| SORT | | ORDER BY | |
| SORT | | GROUP BY NOSORT | |
| INDEX | A_MPAAA_BTREE | FULL SCAN | |

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 1 |
| bytes received via SQL*Net from client | 2427 |
| bytes sent via SQL*Net to client | 51233 |
| calls to get snapshot scn: kcmgss | 6 |
| calls to kcmgcs | 5 |
| consistent gets | 1 |
| consistent gets from cache | 1 |
| consistent gets pin | 1 |
| consistent gets pin (fastpath) | 1 |
| CPU used by this session | 8 |

Idea 4

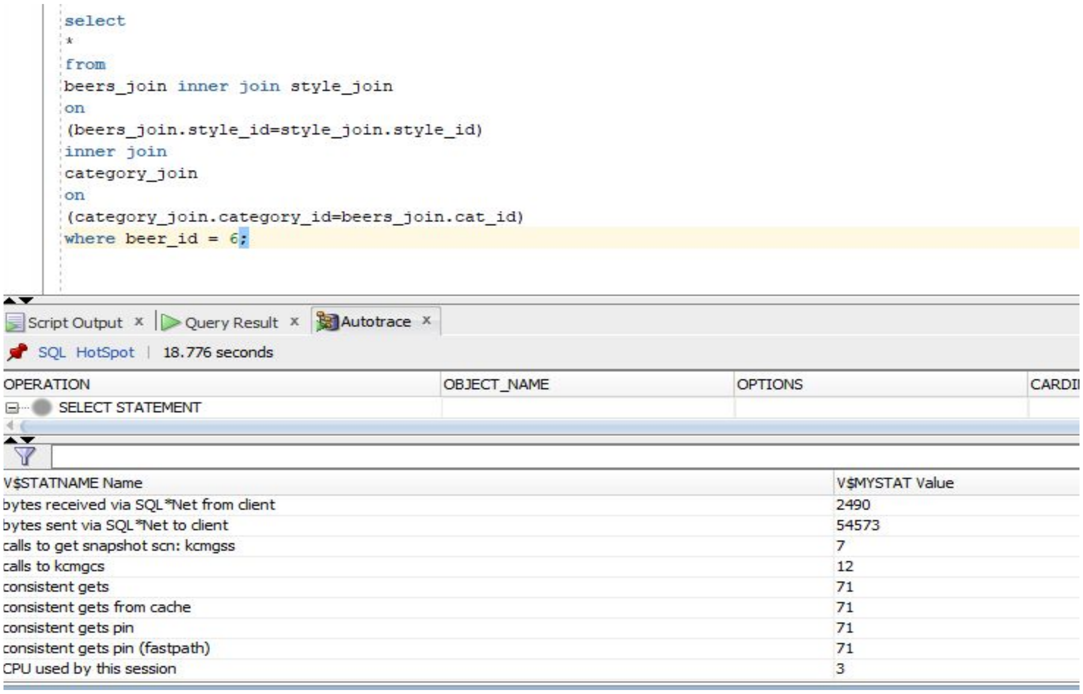
How does indexing affect joins?

This experiment will demonstrate simple joins using a `where` condition, and then discuss performance effects when indexing comes into play.

```
SELECT
    *
FROM
    beers_join
    INNER JOIN style_join
        ON ( beers_join.style_id = style_join.style_id )
    INNER JOIN category_join
        ON ( category_join.category_id = beers_join.cat_id )
WHERE
    beer_id = 6;
```

Without indexing:

```
Consistent gets 71
Db block gets   0
```



```
select
*
from
beers_join inner join style_join
on
(beers_join.style_id=style_join.style_id)
inner join
category_join
on
(category_join.category_id=beers_join.cat_id)
where beer_id = 6;
```

| OPERATION | OBJECT_NAME | OPTIONS | CARDI |
|---|---|---|---|
| SELECT STATEMENT | | | |

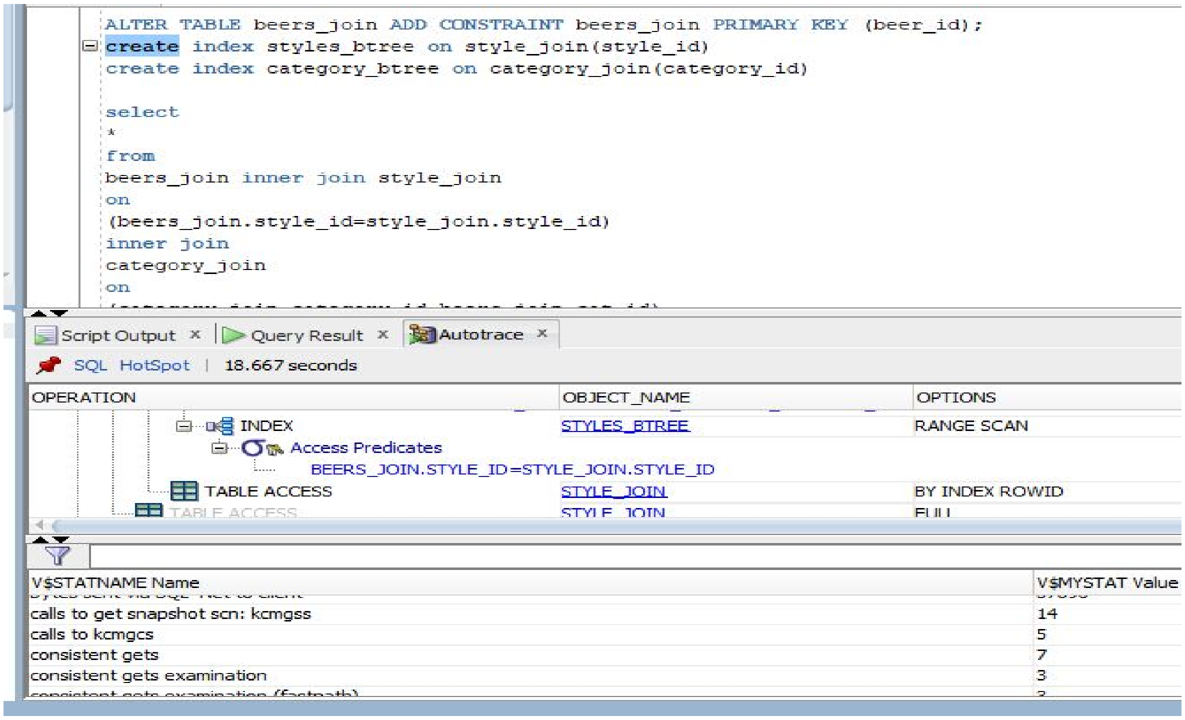
| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2490 |
| bytes sent via SQL*Net to client | 54573 |
| calls to get snapshot scn: kcmgss | 7 |
| calls to kcmgcs | 12 |
| consistent gets | 71 |
| consistent gets from cache | 71 |
| consistent gets pin | 71 |
| consistent gets pin (fastpath) | 71 |
| CPU used by this session | 3 |

After indexing, there is a huge performance increase.

```
Consistent gets 7
Db block gets   0
```

Here are the indexes that were added:

```
ALTER TABLE beers_join ADD CONSTRAINT beers_join PRIMARY KEY (
beer_id );

    CREATE INDEX styles_btree ON
        style_join (
            style_id
        )
create index category_btree ON category_join ( category_id )
```



```
ALTER TABLE beers_join ADD CONSTRAINT beers_join PRIMARY KEY (beer_id);
create index styles_btree on style_join(style_id)
create index category_btree on category_join(category_id)

select
*
from
beers_join inner join style_join
on
(beers_join.style_id=style_join.style_id)
inner join
category_join
on
```

18.667 seconds

```
OPERATION                                   OBJECT_NAME         OPTIONS
        INDEX                               STYLES_BTREE        RANGE SCAN
          Access Predicates
            BEERS_JOIN.STYLE_ID=STYLE_JOIN.STYLE_ID
      TABLE ACCESS                          STYLE_JOIN          BY INDEX ROWID
    TABLE ACCESS                            STYLE_JOIN          FULL
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| calls to get snapshot scn: kcmgss | 14 |
| calls to kcmgcs | 5 |
| consistent gets | 7 |
| consistent gets examination | 3 |

## Idea 5

### What is the difference between single and composite indexes, and when is the best time to use each?

This experiment will demonstrate the usage of indexes when an application is required to fetch more than one attribute. For instance, certain applications will retrieve the first name and last name at the same time. Creating an index only on the first name won’t improve the performance much.

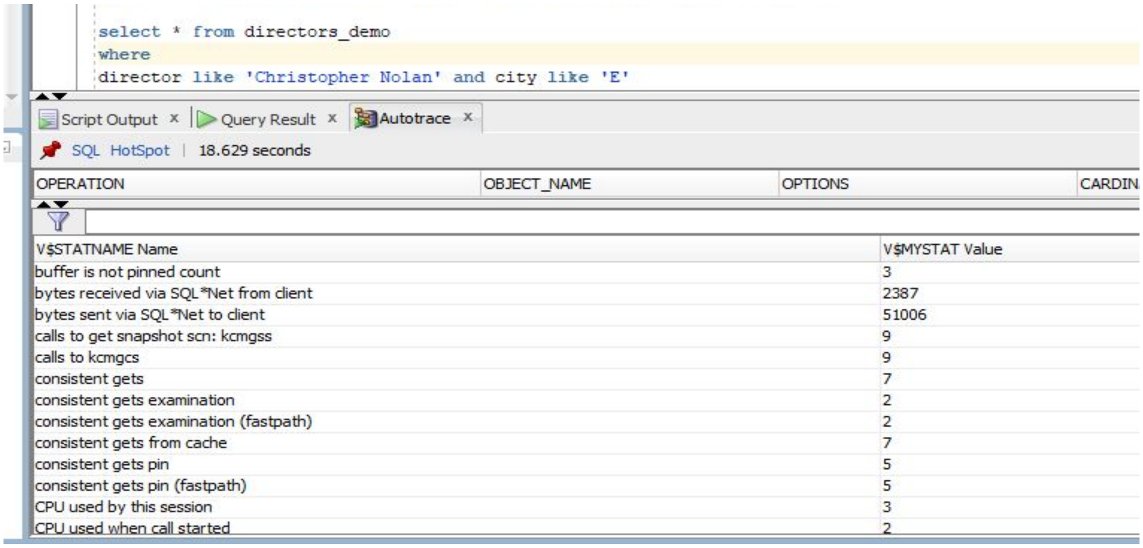
This experiment is used to demonstrate the effect of having multiple indexes.

A `directors_demo` table has been created, and it has columns like `film_id` and `director` name. For the purposes of this experiment, I have added another column which holds the city in which the film was located and where the director directed that movie.
We can assume that all city names are made solely out of letters.

```
CREATE TABLE directors_demo
    AS
        SELECT
            *
        FROM
            directors

ALTER TABLE directors_demo ADD city VARCHAR2(15);

UPDATE directors_demo
SET
    city = DECODE(mod(ROWNUM, 10), 0, 'A', 1, 'B', 2, 'C', 3, 'D', 4,
'E',5, 'F',6, 'G',7, 'H',8, 'J', 'K');

COMMIT;

SELECT
    *
FROM
    directors_demo
WHERE
    director LIKE 'Christopher Nolan'
    AND city like 'E'
```

Before indexing:

```
Consistent gets 7
Db block gets   0
```

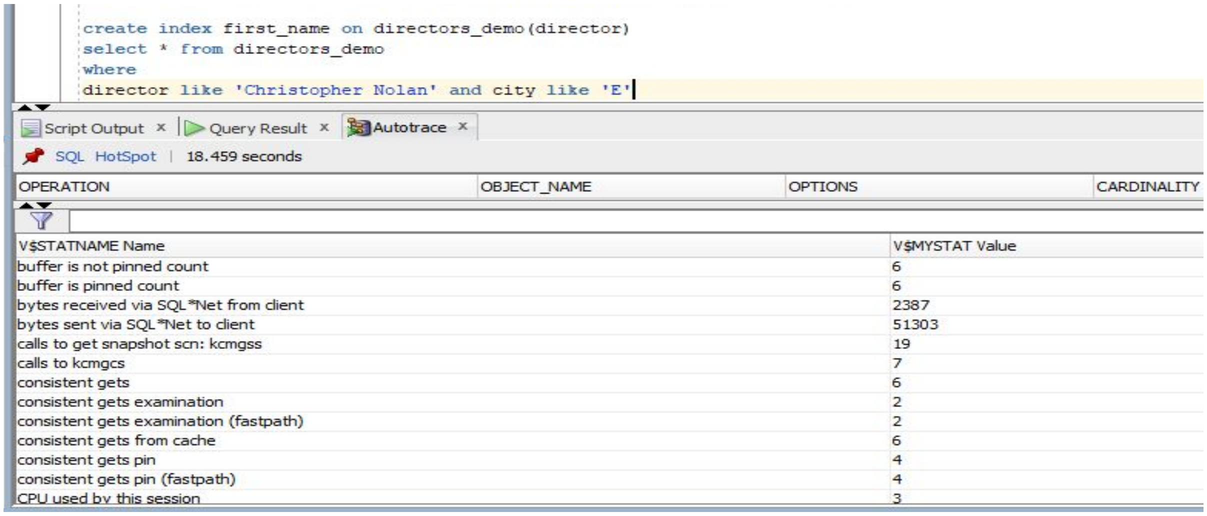


```
select * from directors_demo
where
director like 'Christopher Nolan' and city like 'E'
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 3 |
| bytes received via SQL*Net from client | 2387 |
| bytes sent via SQL*Net to client | 51006 |
| calls to get snapshot scn: kcmgss | 9 |
| calls to kcmgcs | 9 |
| consistent gets | 7 |
| consistent gets examination | 2 |
| consistent gets examination (fastpath) | 2 |
| consistent gets from cache | 7 |
| consistent gets pin | 5 |
| consistent gets pin (fastpath) | 5 |
| CPU used by this session | 3 |
| CPU used when call started | 2 |

After adding a single index on the director's name, consistent gets were reduced to 6. However, this didn't have much of an impact on performance.

```
Db block gets   0
```

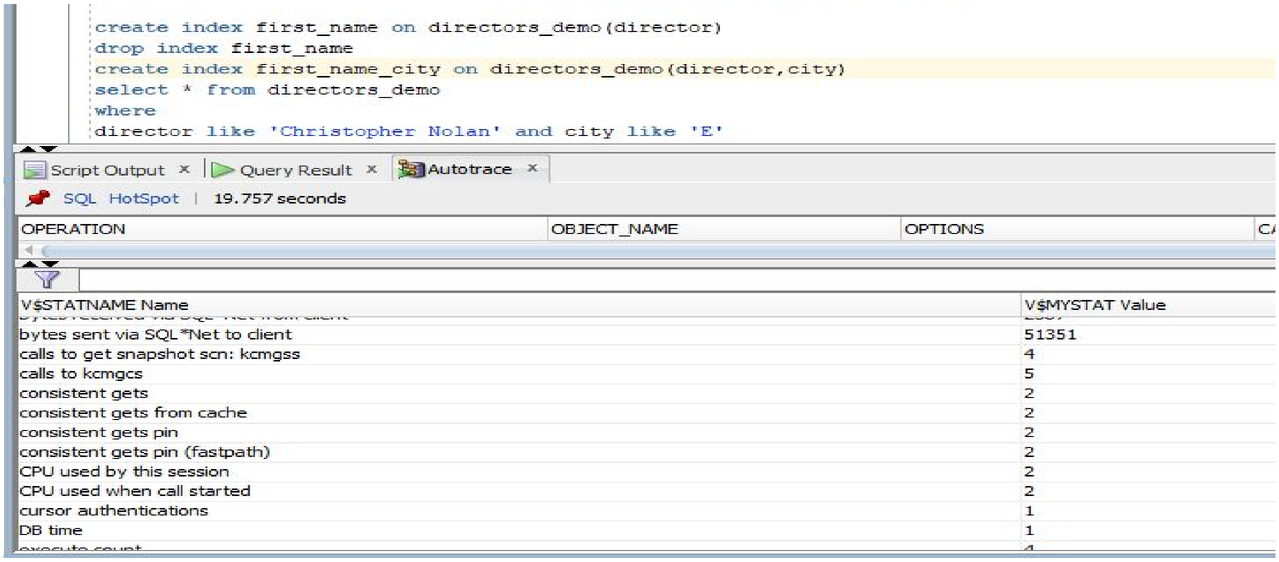


```
create index first_name on directors_demo(director)
select * from directors_demo
where
director like 'Christopher Nolan' and city like 'E'
```

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| buffer is not pinned count | 6 |
| buffer is pinned count | 6 |
| bytes received via SQL*Net from client | 2387 |
| bytes sent via SQL*Net to client | 51303 |
| calls to get snapshot scn: kcmgss | 19 |
| calls to kcmgcs | 7 |
| consistent gets | 6 |
| consistent gets examination | 2 |
| consistent gets examination (fastpath) | 2 |
| consistent gets from cache | 6 |
| consistent gets pin | 4 |
| consistent gets pin (fastpath) | 4 |
| CPU used by this session | 3 |

After adding multiple indexes on the city name and director's name, consistent gets were reduced to 2, proving that the performance was improved greatly.

```
Db block gets   0
```



```
create index first_name on directors_demo(director)
drop index first_name
create index first_name_city on directors_demo(director,city)
select * from directors_demo
where
director like 'Christopher Nolan' and city like 'E'
```

SQL HotSpot | 19.757 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CA |
|---|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | [illegible] |
| bytes sent via SQL*Net to client | 51351 |
| calls to get snapshot scn: kcmgss | 4 |
| calls to kcmgcs | 5 |
| consistent gets | 2 |
| consistent gets from cache | 2 |
| consistent gets pin | 2 |
| consistent gets pin (fastpath) | 2 |
| CPU used by this session | 2 |
| CPU used when call started | 2 |
| cursor authentications | 1 |
| DB time | 1 |
| execute count | 4 |

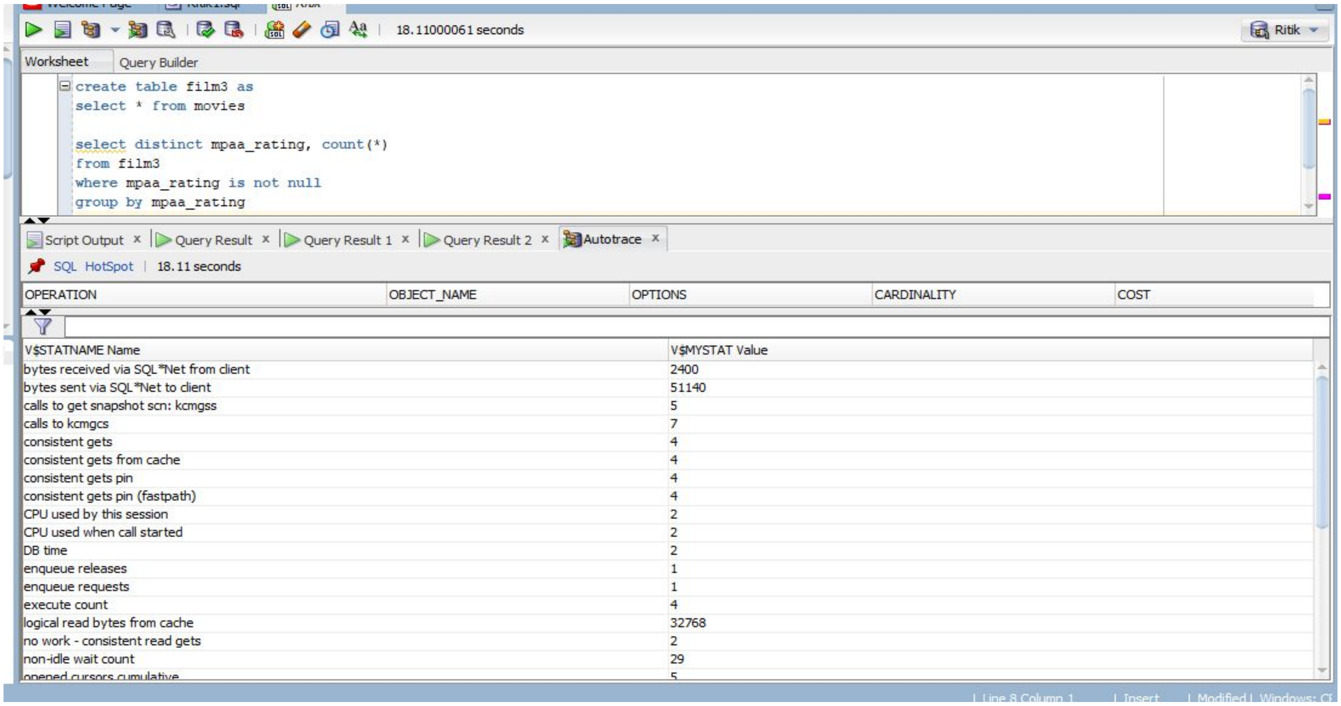
# Idea 6

## What’s the difference between a bitmap and a B-tree index?

First, a simple bitmap index.

Before index:

```
Consistent gets 4
Db block gets   0
```



18.11000061 seconds

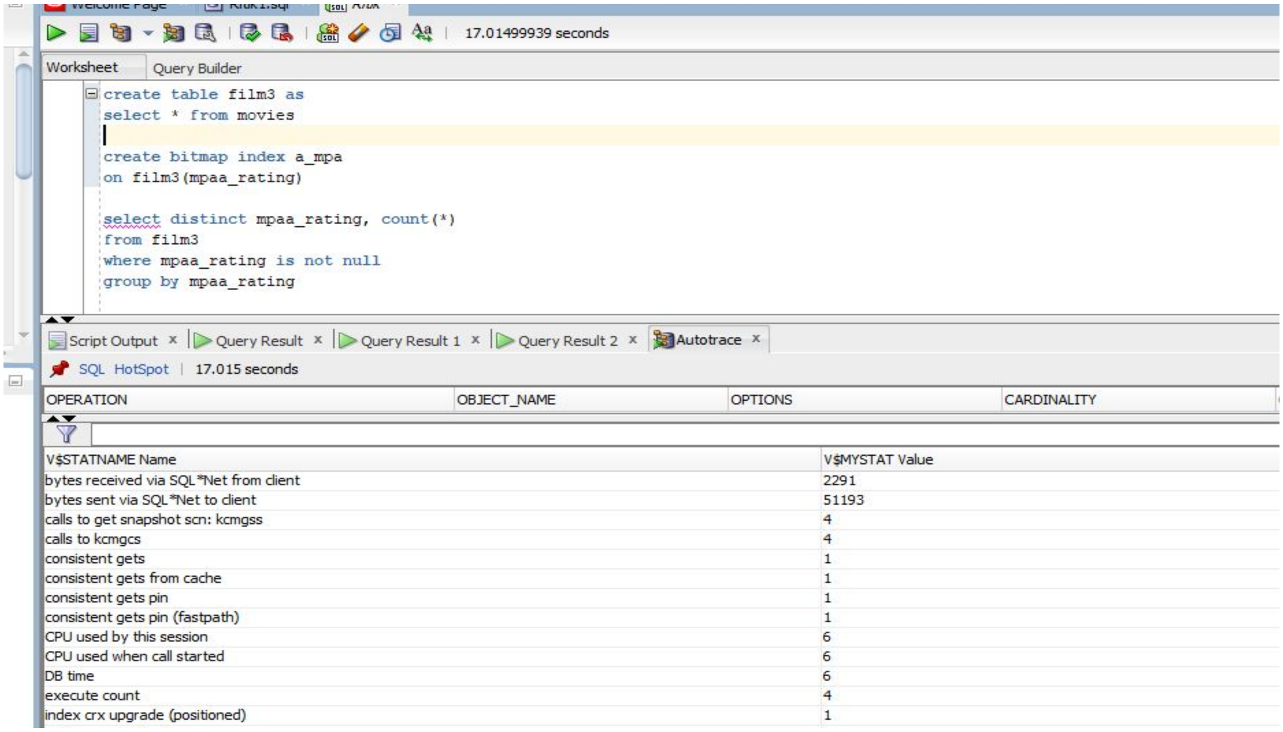
```
create table film3 as
select * from movies

select distinct mpaa_rating, count(*)
from film3
where mpaa_rating is not null
group by mpaa_rating
```

18.11 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CARDINALITY | COST |
|---|---|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2400 |
| bytes sent via SQL*Net to client | 51140 |
| calls to get snapshot scn: kcmgss | 5 |
| calls to kcmgcs | 7 |
| consistent gets | 4 |
| consistent gets from cache | 4 |
| consistent gets pin | 4 |
| consistent gets pin (fastpath) | 4 |
| CPU used by this session | 2 |
| CPU used when call started | 2 |
| DB time | 2 |
| enqueue releases | 1 |
| enqueue requests | 1 |
| execute count | 4 |
| logical read bytes from cache | 32768 |
| no work - consistent read gets | 2 |
| non-idle wait count | 29 |
| opened cursors cumulative | 5 |

| Line 8 Column 1 | Insert | Modified | Windows: CI



```
create table film3 as
select * from movies

create bitmap index a_mpa
on film3(mpaa_rating)

select distinct mpaa_rating, count(*)
from film3
where mpaa_rating is not null
group by mpaa_rating
```

| OPERATION | OBJECT_NAME | OPTIONS | CARDINALITY |
|---|---|---|---|

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2291 |
| bytes sent via SQL*Net to client | 51193 |
| calls to get snapshot scn: kcmgss | 4 |
| calls to kcmgcs | 4 |
| consistent gets | 1 |
| consistent gets from cache | 1 |
| consistent gets pin | 1 |
| consistent gets pin (fastpath) | 1 |
| CPU used by this session | 6 |
| CPU used when call started | 6 |
| DB time | 6 |
| execute count | 4 |
| index crx upgrade (positioned) | 1 |

After index:

```
Consistent gets 1
```

Now let's take a look at the performance difference between B-tree and bitmap indexes.

Based on the previous experiments, we know that bitmap indexes work well with low cardinality attributes and B-tree indexes work well with high cardinality attributes.
We will prove this fact and, since bitmaps are mainly used for data warehousing applications, we will use a reporting query using `group by` that will have low cardinality columns to prove that bitmaps are the best choice when indexing is done on those columns.

We have created a `streaming_service` table which has movies and their `streaming_platform` and `streaming_partners`, along with if they are released or not.

This is how the table is created and updated:

```
CREATE TABLE streaming_service
    AS
        SELECT
            film_id,
            film_title
        FROM
            movies
```

```
ALTER TABLE streaming_service ADD streaming_platform VARCHAR2(15)
ALTER TABLE streaming_service ADD streaming_partner VARCHAR2(15)
ALTER TABLE streaming_service ADD
    released VARCHAR2 ( 5 )
UPDATE streaming_service
SET
    streaming_platform = DECODE(mod(ROWNUM, 4), 0, 'Netflix', 1,
'Amazon', 2, 'Hulu', 3, 'CBS', 4);

COMMIT;

UPDATE streaming_service
SET
    streaming_partner = DECODE(mod(ROWNUM, 3), 0, 'HBO', 1,
'originals',2, 'Sling', 3);

COMMIT;

UPDATE streaming_service SET
    released = DECODE(mod(ROWNUM, 2), 0, 'Y', 1, 'N', 2);

COMMIT;
```

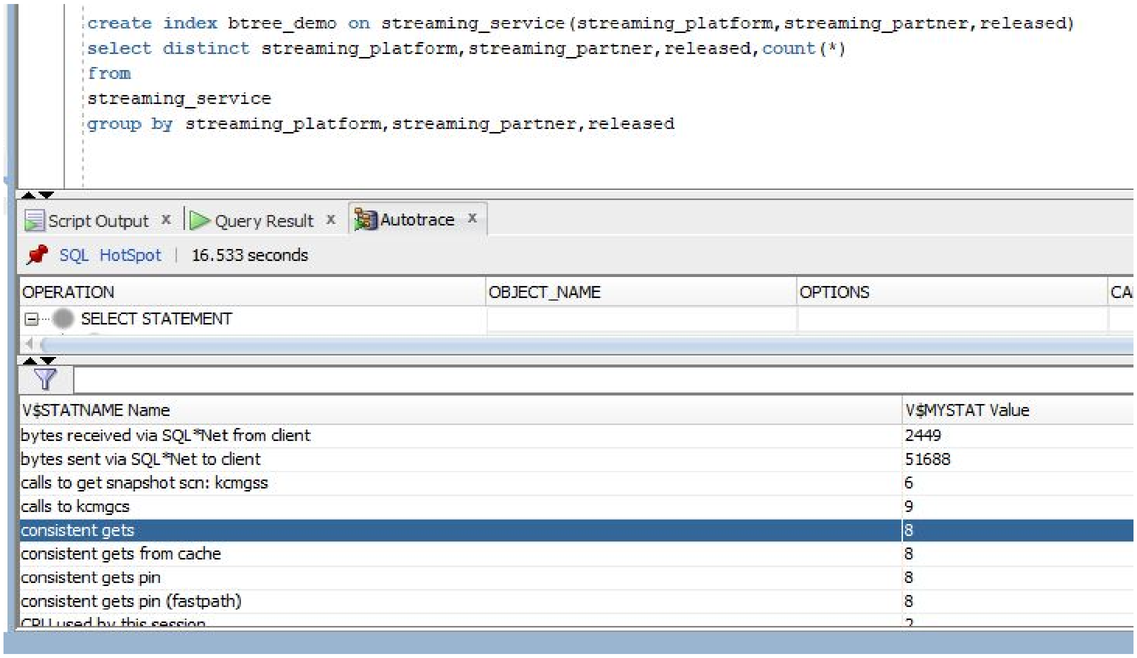
Using a B-tree index:

```
    CREATE INDEX btree_demo ON
        streaming_service (
            streaming_platform,
            streaming_partner,
            released
        )
```

This query returns the number of distinct platforms, partners, and movies released

```
SELECT DISTINCT
    streaming_platform,
    streaming_partner,
    released,
    COUNT(*)
FROM
    streaming_service
GROUP BY
```

```
    streaming_platform,
    streaming_partner,
    released
```



```
create index btree_demo on streaming_service(streaming_platform,streaming_partner,released)
select distinct streaming_platform,streaming_partner,released,count(*)
from
streaming_service
group by streaming_platform,streaming_partner,released
```

16.533 seconds

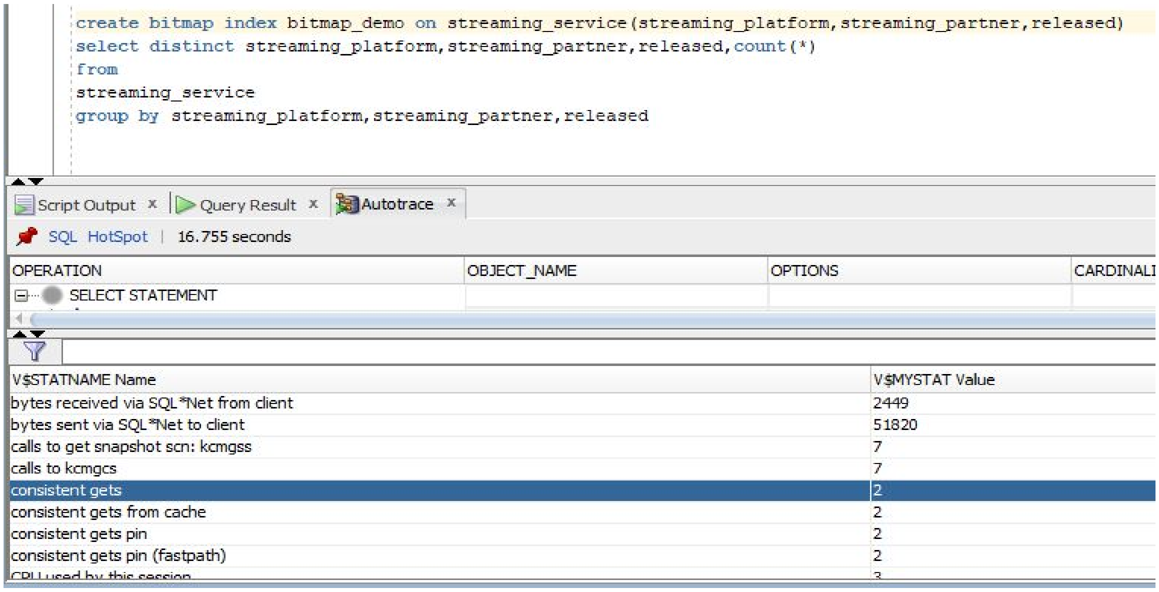
| OPERATION | OBJECT_NAME | OPTIONS | CA |
|---|---|---|---|
| SELECT STATEMENT | | | |

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2449 |
| bytes sent via SQL*Net to client | 51688 |
| calls to get snapshot scn: kcmgss | 6 |
| calls to kcmgcs | 9 |
| consistent gets | 8 |
| consistent gets from cache | 8 |
| consistent gets pin | 8 |
| consistent gets pin (fastpath) | 8 |
| CPU used by this session | 2 |

```
Consistent gets 8
Db block gets   8
```

Using a bitmap index:

```
    CREATE BITMAP INDEX bitmap_demo ON
        streaming_service (
            streaming_platform,
            streaming_partner,
            released
        )
```



```
create bitmap index bitmap_demo on streaming_service(streaming_platform,streaming_partner,released)
select distinct streaming_platform,streaming_partner,released,count(*)
from
streaming_service
group by streaming_platform,streaming_partner,released
```

16.755 seconds

| OPERATION | OBJECT_NAME | OPTIONS | CARDINALI |
|---|---|---|---|
| SELECT STATEMENT | | | |

| V$STATNAME Name | V$MYSTAT Value |
|---|---|
| bytes received via SQL*Net from client | 2449 |
| bytes sent via SQL*Net to client | 51820 |
| calls to get snapshot scn: kcmgss | 7 |
| calls to kcmgcs | 7 |
| consistent gets | 2 |
| consistent gets from cache | 2 |
| consistent gets pin | 2 |
| consistent gets pin (fastpath) | 2 |
| CPU used by this session | 3 |

Using a bitmap index, both values are reduced:

```
Consistent gets 2
Db block gets   2
```

This proves that bitmap indexes have an edge over B-tree indexes in terms of performance when attributes have low cardinality values.