Oracle Data Architecture & Database Optimization

Victor Zacchi

Master of Science Computing

Advanced Databases

D16128783

DT228B yr. 2

Lecturer: Brendan Tierney

Date: 08/12/2018

1. Scope **3**

2. The Data Set **3**

3. Entity Relationship Diagram **4**

3.1 Database Tables 5

4. Query and Database Optimization **7**

4.1 Query list 8

4.2 Query Response Times 11

4.2.1. First run 11

4.2.2. Second run - Increasing the size of our data. 13

4.2.3 Third run - Optimization 16

4.2.4 Recommendations 24

5. JSON Objects in Oracle **26**

1. Scope

The scope of this project is to build a data model from a non-normalised data source and demonstrate the various optimization techniques in the designed model and database queries. In section 3, we demonstrate the entity relationship and the tables created for this project. In section 4 we demonstrate the query results using the original size of this database which we then introduce new data (over 5M) demonstrating the performance of running the same query on a much larger database. In section 5, query and database optimizations are proposed and demonstrated. Section 6, we propose a new model including JSON objects and any optimization required. Lastly the conclusion in section 7.

2. The Data Set

The data set used contains Wine Reviews and 13 variables.

- **Country**: The country that the wine/winery is from.
- **Description**: Wine description.
- **Designation**: The vineyard within the winery where the grapes that made the wine are from
- **Points**: The number of points WineEnthusiast rated the wine on a scale of 1-100.
- **Price**: The cost for a bottle of the wine
- **Province**: The province or state that the wine is from.
- **Region_1**: The wine growing area in a province or state (ie Napa)
- **Region_2**: Sometimes there are more specific regions specified within a wine growing area (ie Rutherford inside the Napa Valley).
- **Taster_name**: taster_twitter_handle
- **Title**: The title of the wine review.
- **Variety**: The type of grapes used to make the wine (ie Pinot Noir)
- **Winery**: The winery that made the wine

This data set is available on the Kaggle website.

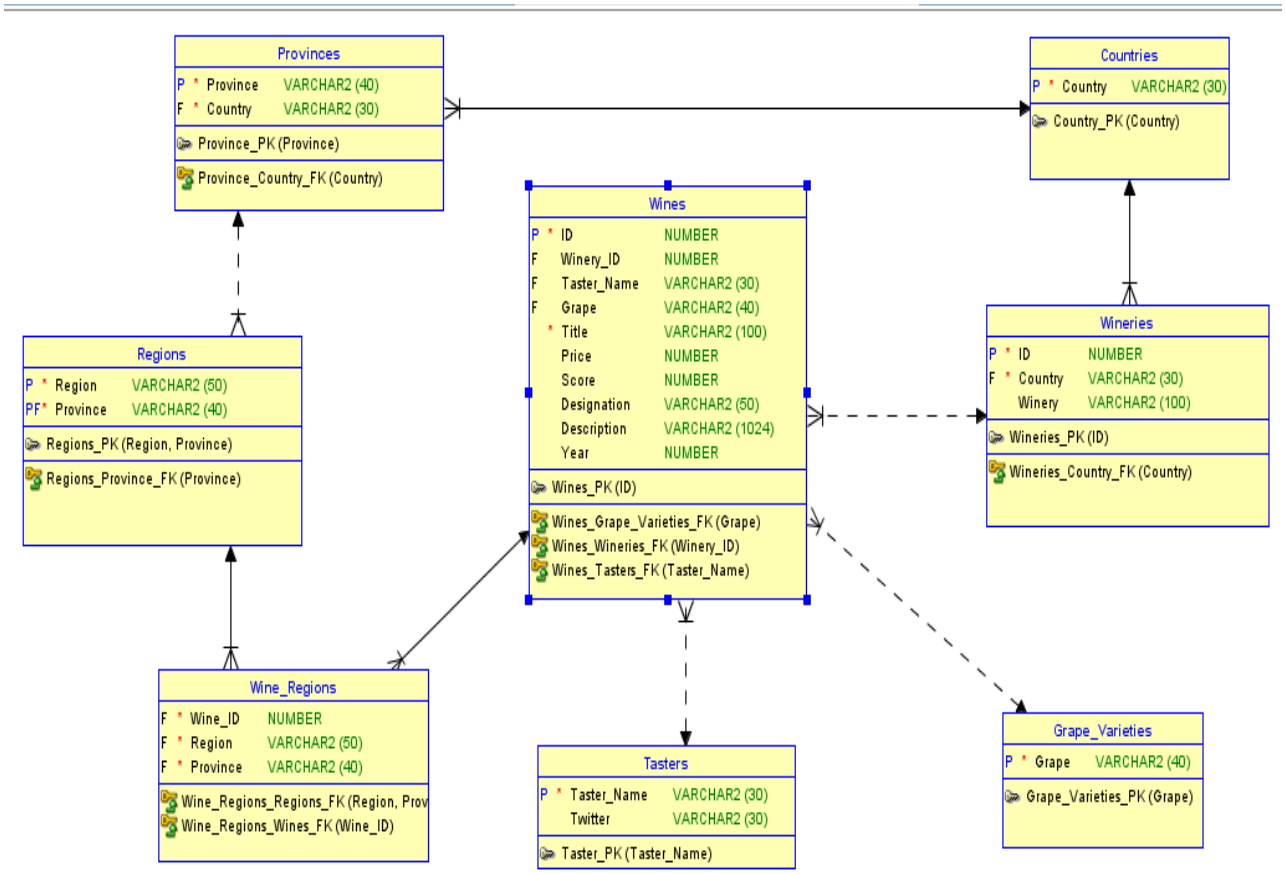
https://www.kaggle.com/zynicide/wine-reviews

3. Entity Relationship Diagram

A basic requirement in developing a database is to have a primary key for each table. We can specify a primary key in two ways:

- Use the data stored as the primary key itself. For example, use the Country name as the primary key of a Countries table. We call this kind of key a Natural Key.
- Use a field specifically created to be a primary key, with artificially generated values, for example by means of a sequence of self-incrementing numbers. This type of key is called a Surrogate key.

In general, a surrogate key is an integer value, with no meaning specific to the record it represents - its value is not derived from the data stored in the database. For demonstration purposes, it was decided to use both techniques.



3.1 Database Tables

COUNTRIES

- This table contains and stores only the country name as the primary key.
- 1:N relationship with the provinces table. Province record must have a country.
- 1:N relationship with the wineries table. A winery record must have a country.

PROVINCES

This dataset contained no duplicated provinces so, for this scenario, one column was created to hold the province name and set as primary key.

- The foreign key of the Countries table is mandatory.
- 0..1:N relationship with Regions. A province can have many regions. A region record can only be in one province.

```
SELECT PROVINCE, COUNT(*) FROM (
SELECT DISTINCT PROVINCE,COUNTRY FROM testdb.TEST_6)
GROUP BY PROVINCE
HAVING COUNT(*) > 1;
> no rows selected
```

REGIONS:*

- A region must be in a province.
- For the regions record to be unique we combined both the region name with the foreign key province, which together forms a unique value.

* There was a mistake in the creation of the Regions table. The province was made a primary and foreign key, making our table to be a 1:1 relationship with the province's table. This caused the province column to be linked to the Wine Regions table which is wrong.

WINERIES

The winery table contained some wineries with the same name but in different countries and therefore a surrogate key was used as the unique value.

```
SELECT DISTINCT WINERY,COUNTRY FROM testdb.TEST_6
WHERE COUNTRY IS NOT NULL AND WINERY IS NOT NULL
AND WINERY = 'Shannon'
ORDER BY WINERY;
```

WINERY	COUNTRY
Shannon	US
Shannon	South Africa

- ID sequence was created for the uniqueness of the winery records.
- Has a 0..1:N relationship with Wines table. A wine record may or may not have a winery, it is not mandatory. A winery may produce many different wines.
- The country is a foreign key and is mandatory. A winery must have a country.

WINES

This table is the main table in our database and is linked to four other tables, Grape Varieties, Tasters, Wineries, and Wine Regions. My original design had this table split into two tables, one for holding the wine record data and another for the reviews. However, I noticed that each wine record had one review only, and therefore, splitting the table would create a 1:1 relationship which did not seem necessary.

- The ID is a sequenced unique column which in this case was created for ease of use and demonstration purposes of using surrogate keys. For example, it seems that each wine *title* (after clearing out some duplicated records) was unique and therefore could have been used as primary key, although, it is quite lengthy and difficult to maintain. Therefore, the creation of a surrogate key was deemed appropriate.
- A title is mandatory.
- A wine record can have zero or one grape variety.
- A wine record can have zero or one taster.
- A wine record can have zero or one winery.
- A wine record can be in a region and sub-region. Therefore it has a 1:N relationship with Wine Regions table.

WINE REGIONS

This table work as a link table between Wines and Regions. This allowed a wine record to have more than one region (a region and sub-region, in this case).

- No specific primary keys needed.
- This table was used to break the N:N relationship between Wines and Regions.

TASTERS

This table holds information such as name and twitter account for each wine enthusiast. In this data set, there are no duplicated names or twitter account so a surrogate key was not needed here. Although, depending on taster name as a primary key could cause some issues such as spelling mistakes, different tasters with the same name e.g. John Smith, also altering the taster's name would not be allowed due to integrity constraint with the Wines table. However, for simplicity and to the scope of this project, the name as a primary key was deemed reasonable.

GRAPE VARIETIES

This table was used as a look-up table and to keep this data record separated from the main table. A surrogate key was not used as all grape types where unique.

4. Query and Database Optimization

In this section, firstly we demonstrate how the database, without using query optimization or indexing, reacts to certain queries when running on a relatively small dataset. Secondly, we increase the size of the database to over 30 times its current size and compare the running time and used resources. Finally, we explore the various database tuning and optimization techniques to improve the performance of our database.

4.1 Query list

There were four main queries used throughout this project.

Q1. Give me each wine records where it has a region and sub-region and the winery is located in the US.

```
SELECT
    wreg.region,
    w.title,
    w.score,
    wnr.country,
    wnr.winery
FROM
    wineries wnr,  wines w, wine_regions wreg
WHERE
    w.id = wreg.wine_id
    AND w.winery_id = wnr.id
    AND wnr.country = 'US'
    AND w.id IN (
        SELECT wine_id
        FROM wine_regions
        GROUP BY wine_id
        HAVING COUNT(wine_id) > );
```

Q2. This query fetches data from four different tables including country, region, wines, wineries and taster's details. Also merging looking for wines that have more than one region and scores between 93 and 89.

```
SELECT
    wreg.region,
    wnr.country,
    w.title,
    w.score,
    t.taster_name,
    t.twitter,
    wnr.winery
FROM
    wineries wnr, wines w, wine_regions wreg, tasters t
WHERE
    w.id = wreg.wine_id
```

```
    AND w.winery_id = wnr.id
    AND w.taster_name = t.taster_name
    AND w.id IN (
        SELECT wine_id
        FROM wine_regions
        GROUP BY wine_id
        HAVING COUNT(wine_id) > 1)
    AND t.twitter IS NOT NULL
    AND w.score <= 93
    AND w.score >= 89;
```

Q.3. This query is attempting to pull data from all of the tables and then doing an average calculation on the wine's price grouping by a few columns. The below query was an attempt to mimic the joins that would be needed if all of our tables were using surrogate keys as the primary key.

```
SELECT
    c.country,
    p.province,
    r.region,
    w.title,
    w.designation,
    w.score,
    gv.grape,
    wnr.winery,
    t.taster_name,
    t.twitter,
    round(AVG(w.price) - 1)
FROM
    countries c, provinces p, wineries wnr, grape_varieties gv, tasters t,
    wine_regions wr, regions r, wines w
WHERE
    w.winery_id = wnr.id
    AND wnr.country = c.country
    AND p.province = r.province
    AND r.region = wr.region
    AND r.province = wr.province
    AND wr.wine_id = w.id
    AND w.grape = gv.grape
    AND w.taster_name = t.taster_name
GROUP BY
```

```
    c.country, p.province, r.region, w.title, w.designation,
    w.score, gv.grape, wnr.winery, t.taster_name, t.twitter;
```

Q4. This query is a similar approach to the previous query, however, it is adding a few extra filters on the grapes, and where wines are in more than one region. A third filter was added to extract the year that was available on nearly every wine record in this dataset.

Title example extracted from dataset: “Fullerton 2015 Lichtenwalter Vineyard Pinot Noir (Ribbon Ridge)”.

```
SELECT  c.COUNTRY,
        p.PROVINCE,
        r.REGION,
        w.TITLE,
        w.DESIGNATION,
        w.SCORE,
        gv.GRAPE,
        wnr.winery,
        t.taster_name,
        t.twitter,
        round(avg(w.price) - 1)
FROM
    countries c, provinces p, wineries wnr, grape_varieties gv,
tasters t, wine_regions wr, regions r, wines w
WHERE
    w.winery_id = wnr.id
    AND wnr.country = c.country
    AND p.province = r.province
    AND r.region = wr.region
    AND r.province = wr.province
    AND wr.wine_id = w.id
    AND w.grape = gv.grape
    AND w.taster_name = t.taster_name
    AND w.score > 85
    AND upper(gv.grape) LIKE '%NOIR%'
    AND upper(w.title) LIKE '%2015%'
    AND w.id IN (
        SELECT wine_id
        FROM wine_regions
        GROUP BY wine_id
        HAVING COUNT(wine_id) > 1)
```

```
GROUP BY
    c.country, p.province, r.region, w.title, w.designation, w.score,
    gv.grape, wnr.winery, t.taster_name, t.twitter;
```

4.2 Query Response Times

4.2.1. First run

In order to turn on the Oracle database optimizer results, run the following command in your database.

```
Set autotrace on;
```

Q1. Results
Plan hash value: 1434536755

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		91	11284	669 (2)	00:00:09
* 1	HASH JOIN		91	11284	669 (2)	00:00:09
* 2	HASH JOIN		74	7770	463 (2)	00:00:06
3	NESTED LOOPS					
4	NESTED LOOPS		230	18630	440 (2)	00:00:06
5	VIEW	VW_NSO_1	4599	59787	209 (3)	00:00:03
* 6	FILTER					
7	HASH GROUP BY		12	22995	209 (3)	00:00:03
8	TABLE ACCESS FULL	WINE_REGIONS	133K	649K	205 (1)	00:00:03
* 9	INDEX UNIQUE SCAN	WINES_PK	1		0 (0)	00:00:01
10	TABLE ACCESS BY INDEX ROWID	WINES	1	68	1 (0)	00:00:01
* 11	TABLE ACCESS FULL	WINERIES	5375	125K	22 (0)	00:00:01
12	TABLE ACCESS FULL	WINE_REGIONS	133K	2468K	205 (1)	00:00:03

Q2. Results.

Plan hash value: 555838692

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
----	----------	----------	-------	-------	----------	--
0	SELECT STATEMENT		280	45360	672 (2)	00:00:09
* 1	HASH JOIN		280	45360	672 (2)	00:00:09
* 2	HASH JOIN		194	27742	467 (2)	00:00:06
* 3	HASH JOIN		194	23086	444 (2)	00:00:06
* 4	TABLE ACCESS FULL	TASTERS	16	416	3 (0)	00:00:01
5	NESTED LOOPS					
----	----------	----------	-------	-------	----------	--------
6	NESTED LOOPS		230	21390	440 (2)	00:00:06
7	VIEW	VW_NSO_1	4599	59787	209 (3)	00:00:03
* 8	FILTER					
9	HASH GROUP BY		12	22995	209 (3)	00:00:03
10	TABLE ACCESS FULL	WINE_REGIONS	133K	649K	205 (1)	00:00:03
* 11	INDEX UNIQUE SCAN	WINES_PK	1		0 (0)	00:00:01
* 12	TABLE ACCESS BY INDEX ROWID	WINES	1	80	1 (0)	00:00:01
13	TABLE ACCESS FULL	WINERIES	16934	396K	22 (0)	00:00:01
14	TABLE ACCESS FULL	WINE_REGIONS	133K	2468K	205 (1)	00:00:03

Q3. Results.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
----	----------	----------	-------	------	-------	----------	--------
0	SELECT STATEMENT		122K	25M		8534 (1)	00:01:43
1	HASH GROUP BY		122K	25M	27M	8534 (1)	00:01:43
* 2	HASH JOIN		122K	25M		2762 (1)	00:00:34
3	INDEX FAST FULL SCAN	REGIONS_PK	1233	35757		4 (0)	00:00:01
* 4	HASH JOIN		122K	21M		2757 (1)	00:00:34
5	TABLE ACCESS FULL	TASTERS	19	494		3 (0)	00:00:01
* 6	HASH JOIN		122K	18M		2753 (1)	00:00:34
7	TABLE ACCESS FULL	WINERIES	16934	396K		22 (0)	00:00:01
* 8	HASH JOIN		122K	16M	5464K	2730 (1)	00:00:33
9	TABLE ACCESS FULL	WINE_REGIONS	133K	3897K		205 (1)	00:00:03
* 10	TABLE ACCESS FULL	WINES	84809	8944K		1777 (1)	00:00:22

Q4. Results

Plan hash value: 4217040034

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
----	----------	----------	-------	-------	----------	---------
0	SELECT STATEMENT		11	2442	659 (2)	00:00:08
1	HASH GROUP BY		11	2442	659 (2)	00:00:08
2	NESTED LOOPS		11	2442	658 (2)	00:00:08
* 3	HASH JOIN		11	2123	658 (2)	00:00:08
4	NESTED LOOPS					
5	NESTED LOOPS		8	1304	452 (2)	00:00:06
* 6	HASH JOIN		8	1112	444 (2)	00:00:06
7	NESTED LOOPS					
8	NESTED LOOPS		8	904	440 (2)	00:00:06
9	VIEW	VW_NSO_1	4599	22995	209 (3)	00:00:03
* 10	FILTER					
11	HASH GROUP BY		1	22995	209 (3)	00:00:03
12	TABLE ACCESS FULL	WINE_REGIONS	133K	649K	205 (1)	00:00:03
* 13	INDEX UNIQUE SCAN	WINES_PK	1		0 (0)	00:00:01
* 14	TABLE ACCESS BY INDEX ROWID	WINES	1	108	1 (0)	00:00:01
15	TABLE ACCESS FULL	TASTERS	19	494	3 (0)	00:00:01
* 16	INDEX UNIQUE SCAN	WINERIES_PK	1		0 (0)	00:00:01
17	TABLE ACCESS BY INDEX ROWID	WINERIES	1	24	1 (0)	00:00:01
18	TABLE ACCESS FULL	WINE_REGIONS	133K	3897K	205 (1)	00:00:03
* 19	INDEX UNIQUE SCAN	REGIONS_PK	1	29	0 (0)	00:00:01

As shown in the above tables all our queries are doing a full scan on each joined table, which means that to find a specific record or a list or records these queries must scan through every row in every requested table to fetch the desired results. This is extremely inefficient for medium to large size databases. Q3 and Q4 seem to be doing a lot of work and could be very slow to return the desired records.

4.2.2. Second run - Increasing the size of our data.

Initially, we had 130,000 records which have been increased as follows.

- Table *TASTERS* added 1,100,000 new records.
- Table *WINERIES* added 1,200,000 new records.
- Table *WINES* added 5,000,000 new records.
- Table *WINE_REGIONS* 2,600,000 new records.

Q1. Results 2.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		834	78396		17146 (1)	00:03:26
* 1	HASH JOIN		834	78396		17146 (1)	00:03:26
* 2	HASH JOIN		1182	88650		12724 (1)	00:02:33
3	NESTED LOOPS						
4	NESTED LOOPS		2716	127K		11245 (1)	00:02:15
5	VIEW	VW_NSO_1	54311	689K		8255 (2)	00:01:40
* 6	FILTER						
7	HASH GROUP BY		136	477K	41M	8255 (2)	00:01:40
8	TABLE ACCESS FULL	WINE_REGIONS	2728K	23M		4411 (1)	00:00:53
* 9	INDEX UNIQUE SCAN	WINES_PK	1			1 (0)	00:00:01
10	TABLE ACCESS BY INDEX ROWID	WINES	1	35		2 (0)	00:00:01
* 11	TABLE ACCESS FULL	WINERIES	7256	191K		1478 (1)	00:00:18
12	TABLE ACCESS FULL	WINE_REGIONS	2728K	49M		4413 (1)	00:00:53

Q2. Results 2.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		3204	422K		18499 (1)	00:03:42
* 1	HASH JOIN		3204	422K		18499 (1)	00:03:42
* 2	HASH JOIN		2704	306K		14076 (1)	00:02:49
* 3	HASH JOIN		2704	235K		12595 (1)	00:02:32
4	NESTED LOOPS						
5	NESTED LOOPS		2716	164K		11245 (1)	00:02:15
6	VIEW	VW_NSO_1	54311	689K		8255 (2)	00:01:40
* 7	FILTER						
8	HASH GROUP BY		136	477K	41M	8255 (2)	00:01:40
9	TABLE ACCESS FULL	WINE_REGIONS	2728K	23M		4411 (1)	00:00:53
* 10	INDEX UNIQUE SCAN	WINES_PK	1			1 (0)	00:00:01
* 11	TABLE ACCESS BY INDEX ROWID	WINES	1	49		2 (0)	00:00:01
* 12	TABLE ACCESS FULL	TASTERS	1100K	28M		1346 (1)	00:00:17
13	TABLE ACCESS FULL	WINERIES	1216K	31M		1476 (1)	00:00:18
14	TABLE ACCESS FULL	WINE_REGIONS	2728K	49M		4413 (1)	00:00:53

Q3. Results 2.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		2715K	468M		189K (1)	00:38:00
1	HASH GROUP BY		2715K	468M	505M	189K (1)	00:38:00
* 2	HASH JOIN		2715K	468M		82141 (1)	00:16:26
3	INDEX FAST FULL SCAN	REGIONS_PK	1233	35757		4 (0)	00:00:01
* 4	HASH JOIN		2715K	393M	45M	82128 (1)	00:16:26
5	TABLE ACCESS FULL	WINERIES	1216K	31M		1476 (1)	00:00:18
* 6	HASH JOIN		2715K	323M	40M	60796 (1)	00:12:10
7	TABLE ACCESS FULL	TASTERS	1100K	28M		1345 (1)	00:00:17
* 8	HASH JOIN		2728K	254M	111M	43212 (1)	00:08:39
9	TABLE ACCESS FULL	WINE_REGIONS	2728K	80M		4415 (1)	00:00:53
* 10	TABLE ACCESS FULL	WINES	5108K	326M		14128 (1)	00:02:50

Q4. Results 2.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		1285	238K		17671 (1)	00:03:33
1	HASH GROUP BY		1285	238K		17671 (1)	00:03:33
2	NESTED LOOPS		1285	238K		17670 (1)	00:03:33
* 3	HASH JOIN		1286	202K		17670 (1)	00:03:33
4	NESTED LOOPS						
5	NESTED LOOPS		512	66560		13246 (1)	00:02:39
6	NESTED LOOPS		512	52736		12221 (1)	00:02:27
7	NESTED LOOPS		514	39064		11193 (2)	00:02:15
8	VIEW	VW_NSO_1	54311	477K		8255 (2)	00:01:40
* 9	FILTER						
10	HASH GROUP BY		7	477K	41M	8255 (2)	00:01:40
11	TABLE ACCESS FULL	WINE_REGIONS	2728K	23M		4411 (1)	00:00:53
* 12	TABLE ACCESS BY INDEX ROWID	WINES	1	67		2 (0)	00:00:01
* 13	INDEX UNIQUE SCAN	WINES_PK	1			1 (0)	00:00:01
14	TABLE ACCESS BY INDEX ROWID	TASTERS	1	27		2 (0)	00:00:01
* 15	INDEX UNIQUE SCAN	TASTER_PK	1			1 (0)	00:00:01
* 16	INDEX UNIQUE SCAN	WINERIES_PK	1			1 (0)	00:00:01
17	TABLE ACCESS BY INDEX ROWID	WINERIES	1	27		2 (0)	00:00:01
* 18	TABLE ACCESS FULL	WINE_REGIONS	2728K	80M		4415 (1)	00:00:53
* 19	INDEX UNIQUE SCAN	REGIONS_PK	1	29		0 (0)	00:00:01

As shown above, with our database now having millions of rows on at least four tables, the query became very slow and taking several seconds and sometimes minutes to return the desired result. The cost of CPU (17146, 18499, 189K, 17671) and data transferred bytes are

also quite high on all four queries, which means our queries will become more and more inefficient as it grows.

4.2.3 Third run - Optimization

In this scenario, we will demonstrate techniques to improve our queries.

Q1. Results 3.

Improving the query:

1. The most restrictive condition “US” has been moved to the very end of the query tree.
2. Added ORDER BY wreg.REGION;
3. Created indexes:

```
A. create index winery_country_inx on wineries (UPPER(country) );
B. create index wineregions_ix_wine on wine_regions ( wine_id);
C. create index wineregions_ix_region on wine_regions ( region );
```

O1. Optimized query (Q1)

```
SELECT
    wreg.region,
    w.title,
    w.score,
    wnr.country,
    wnr.winery
FROM
    wineries wnr, wines w, wine_regions wreg
WHERE
    w.winery_id = wnr.id
    AND w.id = wreg.wine_id
    AND w.id IN (
        SELECT  wine_id
        FROM wine_regions
        GROUP BY wine_id
        HAVING COUNT(wine_id) > 1)
    AND upper(wnr.country) = 'US'
ORDER BY w.score;
```

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		445	42720		10520 (2)	00:02:07
1	SORT ORDER BY		445	42720		10520 (2)	00:02:07
2	NESTED LOOPS						
3	NESTED LOOPS		445	42720		10519 (2)	00:02:07
* 4	HASH JOIN		630	48510		8857 (2)	00:01:47
5	NESTED LOOPS						
6	NESTED LOOPS		2716	116K		8823 (2)	00:01:46
7	VIEW	VW_NSO_1	54311	477K		5886 (2)	00:01:11
* 8	FILTER						
9	HASH GROUP BY		136	477K	41M	5886 (2)	00:01:11
10	INDEX FAST FULL SCAN	WINEREGIONS_IX_WINE	2728	23M		2042 (1)	00:00:25
* 11	INDEX UNIQUE SCAN	WINES_PK	1			1 (0)	00:00:01
12	TABLE ACCESS BY INDEX ROWID	WINES	1	35		2 (0)	00:00:01
13	TABLE ACCESS BY INDEX ROWID	WINERIES	3869	124K		33 (0)	00:00:01
* 14	INDEX RANGE SCAN	WINERY_COUNTRY_INX	3869			12 (0)	00:00:01
* 15	INDEX RANGE SCAN	WINEREGIONS_IX_WINE	3			2 (0)	00:00:01
16	TABLE ACCESS BY INDEX ROWID	WINE_REGIONS	1	19		5 (0)	00:00:01

```
Statistics
----------------------------------------------------------

            504  CPU used by this session
            504  CPU used when call started
```

As shown above, the indexes created have reduced the full scan tables which have improved considerably our query Q1 results.

Q2. Results 3.

This query was doing four full scans.

1. Changed position of Winery - Wines - Wine Regions - Tasters
2. Added Order BY to an already indexed value wreg.REGION.
3. The most restricting condition, "Twitter is not null" was pushed to the end of the query.

Results:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		3248	431K		16129 (2)	00:03:14
* 1	HASH JOIN		3248	431K		16129 (2)	00:03:14
* 2	HASH JOIN		2704	308K		11706 (2)	00:02:21
* 3	HASH JOIN		2704	235K		10226 (2)	00:02:03
4	NESTED LOOPS						
5	NESTED LOOPS		2716	164K		8875 (2)	00:01:47
6	VIEW	VW_NSO_1	54311	689K		5886 (2)	00:01:11
* 7	FILTER						
8	HASH GROUP BY		136	477K	41M	5886 (2)	00:01:11
9	INDEX FAST FULL SCAN	WINEREGIONS_IX_WINE	2728K	23M		2042 (1)	00:00:25
* 10	INDEX UNIQUE SCAN	WINES_PK	1			1 (0)	00:00:01
* 11	TABLE ACCESS BY INDEX ROWID	WINES	1	49		2 (0)	00:00:01
* 12	TABLE ACCESS FULL	TASTERS	1100K	28M		1346 (1)	00:00:17
13	TABLE ACCESS FULL	WINERIES	1216K	32M		1476 (1)	00:00:18
14	TABLE ACCESS FULL	WINE_REGIONS	2728K	49M		4413 (1)	00:00:53

```
Statistics
----------------------------------------------------------

          683  CPU used by this session
          683  CPU used when call started
          690  DB time
```

Very little improvement was achieved here, the indexing on wineregions_ix_wine has helped a little but it is still doing a full scan on four tables. This is mostly because our query doesn't have many restrictions and is fetching data from all tables where the score is between 93 and 89, therefore, there isn't much escaping from doing full table scans. A good alternative for this type of queries is the creation of view tables (materialized views), which will be shown in Q3 results.

Q3. Results 3.

This query was quite costly and poorly written. This query demonstrated how much complexity surrogate keys could add to our system. To get the same results as showing in the below table, it needs many joins and therefore more CPU cost and consistent gets.

To improve it:

1. The query was rewritten removing any unnecessary table joins.
2. The use of indexes in this type of query seems to only create more work for the computer. It seems that, since there is no equal sign and many group by operations the query must go through a full scan to be able to bring the aggregated value requested by the AVG condition hence the high cost of CPU and Disk. Reduced query and results are shown below.

O3. Optimized query (Q3)

```
SELECT
    wnr.country,
    wr.province,
    wr.region,
    w.title,
    w.designation,
    w.score,
    w.grape,
    wnr.winery,
    t.taster_name,
    t.twitter,
    round(AVG(w.price) - 1)
FROM
    wineries wnr, tasters t, wine_regions wr, wines w
WHERE
    w.taster_name = t.taster_name
    AND wr.wine_id = w.id
    AND w.winery_id = wnr.id
GROUP BY
    wnr.country, wr.province, wr.region, w.title, w.designation,
    w.score, w.grape, wnr.winery, t.taster_name, t.twitter;
```

Results:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		2715K	396M		174K (1)	00:34:49
1	HASH GROUP BY		2715K	396M	433M	174K (1)	00:34:49
* 2	HASH JOIN		2715K	396M	46M	82183 (1)	00:16:27
3	TABLE ACCESS FULL	WINERIES	1216K	32M		1476 (1)	00:00:18
* 4	HASH JOIN		2715K	323M	40M	60793 (1)	00:12:10
5	TABLE ACCESS FULL	TASTERS	1100K	28M		1345 (1)	00:00:17
* 6	HASH JOIN		2728K	254M	111M	43209 (1)	00:08:39
7	TABLE ACCESS FULL	WINE_REGIONS	2728K	80M		4415 (1)	00:00:53
8	TABLE ACCESS FULL	WINES	5108K	326M		14124 (1)	00:02:50

```
Statistics
----------------------------------------------------------

          1075  CPU used by this session
          1075  CPU used when call started
          3914  DB time
```

To illustrate the inefficiency of over-indexing in this scenario, I have added the following new indexes:

```
create index wines_ix_price on wines (price);
create index wines_ix_grape on wines (grape);
```

Results after indexing:

```
Statistics
----------------------------------------------------------

          1268  CPU used by this session
          1268  CPU used when call started
         10267  DB time
         77341  consistent gets
        645890  bytes sent via SQL*Net to client
```

We couldn't reduce the full table scans easily for this query so let's try something else.

Using *Parallel* queries results:

```
statistics
-----------------------------------------------------------
        1089  CPU used by this session
        1089  CPU used when call started
```

A very slight improvement in CPU usage, but not enough. Let's try creating materialized views.

1. The total records returned by this query is 2,689,263 records
2. Created a materialized view for this query for 2.5 mil records which was created fairly quickly and increment for the next 500K.

```
CREATE MATERIALIZED VIEW avg_price_wines
PCTFREE 5 PCTUSED 60
STORAGE (INITIAL 2500000 NEXT 500K)
USING INDEX STORAGE (INITIAL 100K NEXT 100K)
REFRESH START WITH ROUND(SYSDATE)
NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY') + 15/24
AS SELECT ... <full query>
```

When querying the new materialized view table created avg_price_wines:

```
Plan hash value: 3603190773

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                 |  3647K|  1544M| 11499   (1)| 00:02:18 |
|   1 |  MAT_VIEW ACCESS FULL| AVG_PRICE_WINES |  3647K|  1544M| 11499   (1)| 00:02:18 |
-----------------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
              3  CPU used by this session
              3  CPU used when call started
              8  DB time
```

This was a lot faster than running the same query every time. This approach has proven to be very successful for this type of queries that have many joins, group by, and aggregates AVG, SUM, etc.

Q4. Results 3

This query was doing quite a lot, although, only doing one full scan on table WINE_REGIONS. Let's see if we can improve it further.

1. Re-wrote the query and removed any unnecessary joins.
2. Move most restrictive values further down the tree.
3. Also, it has been noticed previously that all wine records had the date imprinted in the Title column so we created a new column in the Wines table to store the Year and extracted it from the title with the following.

```
UPDATE wines
SET year = regexp_substr(title,'19[0-9][0-9]')
WHERE year IS NULL;

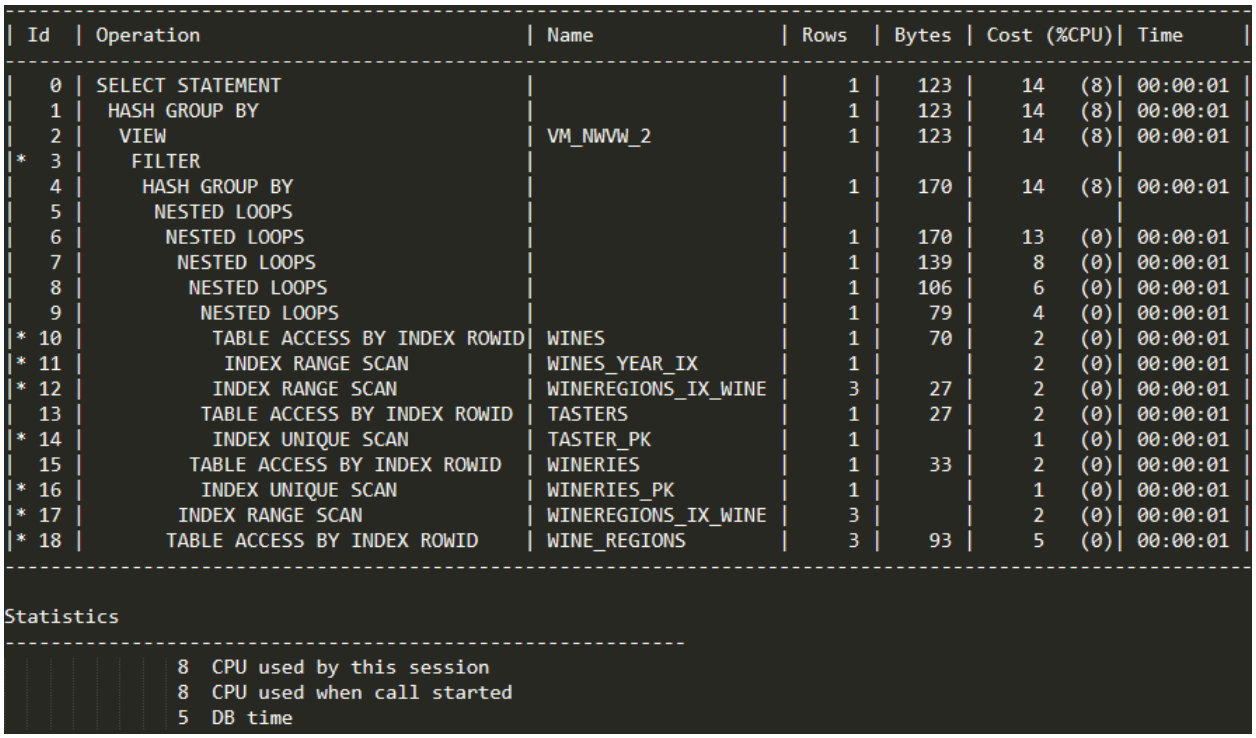
UPDATE wines
SET year = regexp_substr(title,'20[0-9][0-9]')
WHERE year IS NULL;
```

4. With this, there is no need to use the %% anymore and we can search by year using the equal operator.
5. Also added an index to the Year column, WINES_YEAR_IX.
6. Lastly, moved all the restrictive values further down.

O4. Optimized query (Q4):

```
SELECT
    wnr.country,
    wr.province,
    wr.region,
    w.title,
    w.designation,
    w.score,
    w.grape,
    wnr.winery,
    t.taster_name,
    t.twitter,
    round(AVG(w.price) - 1)
FROM
    wineries wnr, tasters t, wine_regions wr, wines w
WHERE
    w.winery_id = wnr.id
    AND wr.wine_id = w.id
    AND w.taster_name = t.taster_name
    AND w.id IN (
        SELECT wine_id
        FROM wine_regions
        GROUP BY wine_id
        HAVING COUNT(wine_id) > 1)
    AND wr.province <> '%France%'
    AND upper(w.grape) LIKE '%NOIR%'
    AND w.score > 85
    AND w.year = 2015
GROUP BY
    wnr.country, wr.province, wr.region, w.title, w.designation, w.score,
w.grape,
    wnr.winery, t.taster_name, t.twitter;
```

The result shows a significant change with the optimizations performed to this query and now using both WINEREGIONS_IX_WINE and WINES_YEAR_IX indexes the following is how it looks like by just re-running the same query Q4.



```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                     |     1 |   123 |    14   (8)| 00:00:01 |
|   1 |  HASH GROUP BY                     |                     |     1 |   123 |    14   (8)| 00:00:01 |
|   2 |   VIEW                             | VM_NWVW_2           |     1 |   123 |    14   (8)| 00:00:01 |
|*  3 |    FILTER                          |                     |       |       |            |          |
|   4 |     HASH GROUP BY                  |                     |     1 |   170 |    14   (8)| 00:00:01 |
|   5 |      NESTED LOOPS                  |                     |       |       |            |          |
|   6 |       NESTED LOOPS                 |                     |     1 |   170 |    13   (0)| 00:00:01 |
|   7 |        NESTED LOOPS                |                     |     1 |   139 |     8   (0)| 00:00:01 |
|   8 |         NESTED LOOPS               |                     |     1 |   106 |     6   (0)| 00:00:01 |
|   9 |          NESTED LOOPS              |                     |     1 |    79 |     4   (0)| 00:00:01 |
|* 10 |           TABLE ACCESS BY INDEX ROWID| WINES             |     1 |    70 |     2   (0)| 00:00:01 |
|* 11 |            INDEX RANGE SCAN        | WINES_YEAR_IX       |     1 |       |     2   (0)| 00:00:01 |
|* 12 |           INDEX RANGE SCAN         | WINEREGIONS_IX_WINE |     3 |    27 |     2   (0)| 00:00:01 |
|  13 |          TABLE ACCESS BY INDEX ROWID | TASTERS           |     1 |    27 |     2   (0)| 00:00:01 |
|* 14 |           INDEX UNIQUE SCAN        | TASTER_PK           |     1 |       |     1   (0)| 00:00:01 |
|  15 |         TABLE ACCESS BY INDEX ROWID | WINERIES           |     1 |    33 |     2   (0)| 00:00:01 |
|* 16 |          INDEX UNIQUE SCAN         | WINERIES_PK         |     1 |       |     1   (0)| 00:00:01 |
|* 17 |        INDEX RANGE SCAN            | WINEREGIONS_IX_WINE |     3 |       |     2   (0)| 00:00:01 |
|* 18 |       TABLE ACCESS BY INDEX ROWID  | WINE_REGIONS        |     3 |    93 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          8  CPU used by this session
          8  CPU used when call started
          5  DB time
```

Another great solution for this type of queries would be to have partition using the YEAR column that was now created which could also speed the process as the query hit the partition where the year of 2015 is located instead of scanning the entire table.

To demonstrate that it was created new a partitioned table WINES_2, and synchronized the records in WINES table and then redefined the original table using DBMS_REDEFINITION

```
BEGIN
  dbms_redefinition.finish_redef_table(
    uname      => USER,
    orig_table => 'WINES',
    int_table  => 'WINES_2');
END;
/
```

Now, the WINES table is partitioned as follows:

```
'WINES_OLDER_199   COUNT(*)
---------------- ----------
WINES_OLDER_1995          0
WINES_OLDER_2000    1500000
WINES_OLDER_2005     350000
```

```
WINES_OLDER_2010    1500000
WINES_OLDER_2015     750000
WINES_OVER_2016      879968
```

The price index was dropped and re-ran query Q4 which and it was quite fast.

```
Statistics
----------------------------------------------------------

            34  CPU used by this session
            34  CPU used when call started
           118  DB time
             4  Requests to/from client
          9271  session logical reads
```

4.2.4 Recommendations

Using Surrogate Keys

One of the advantages of using surrogate keys is that there is no need to enter values in this field when inserting data into a table, since the values are automatically generated - although in this case, the system can use more resources during insertion of the keys. you need to generate the key value. Also another issue, in this scenario, for example the table country, province, and grape_varieties, it would require extra joins in our query to be able to fetch results from these tables.

Natural Keys

Within the presented model, for natural keys we had rely on names such as taster name, country name, grape name, which could be a problem if a spelling mistake occurred we would not be able to change it easily due to foreign key constraints with other tables. Although for query purposes only this technique of using natural keys have avoided unnecessary joins with other tables consequently improving our queries performance.

Indexing Guideline

- When indexed columns are modified, the DBMS shifts resource internally to keep these indexes updated and associated;
- The maintenance of indexes requires time and resources, so do not create indexes that will not be used effectively;

- When containing large amounts of duplicate data, indexes present more cost than benefits. As well as using indexes with attributes of little variation, such as "Country".
- Use Index on PK & FKs of a table.
- Add secondary index on columns that are involved in: selection or join criteria; ORDER BY; GROUP BY; and other operations involving sorting (such as UNION or DISTINCT).

Examples where the index will probably not help

- When using WHERE grape LIKE '%Noir%'.
- WHERE colA = colB / 2 or looking for the AVG, SUM, STANDDEV etc.

In-Memory tables

Unfortunately I was not able to use this technique due to some technical issues. Although it could be considered for companies where they have a comfortable amount of RAM to use. Using in-memory tables make it a lot faster to fetch data when compared to hard disks. The table Wine_Regions has 2.6 mil records and was very often used in our queries to link the Wines table to Regions table so we could have used it here. Alternatively, if RAM space is not an issue then would use it in the Wines table which is always used and has the most records.

Materialised Views

A materialised view is an actual table in the database that is updated always when an update occurs on some table used by the saved query. For this reason, the moment the user makes a query in this materialized view the result will be faster than if it were not materialized.

If space is not an issue, as results are stored in a new table and updated sporadically, materialised views can be very efficient for querying the database.

Database Partitioning

The main advantage - and the purpose - of partitioning is to provide performance advantages. It also enables better manageability for multiple applications. The purpose of partitioning is to split the database objects, such as tables, indexes, and other objects into manageable, smaller parts.

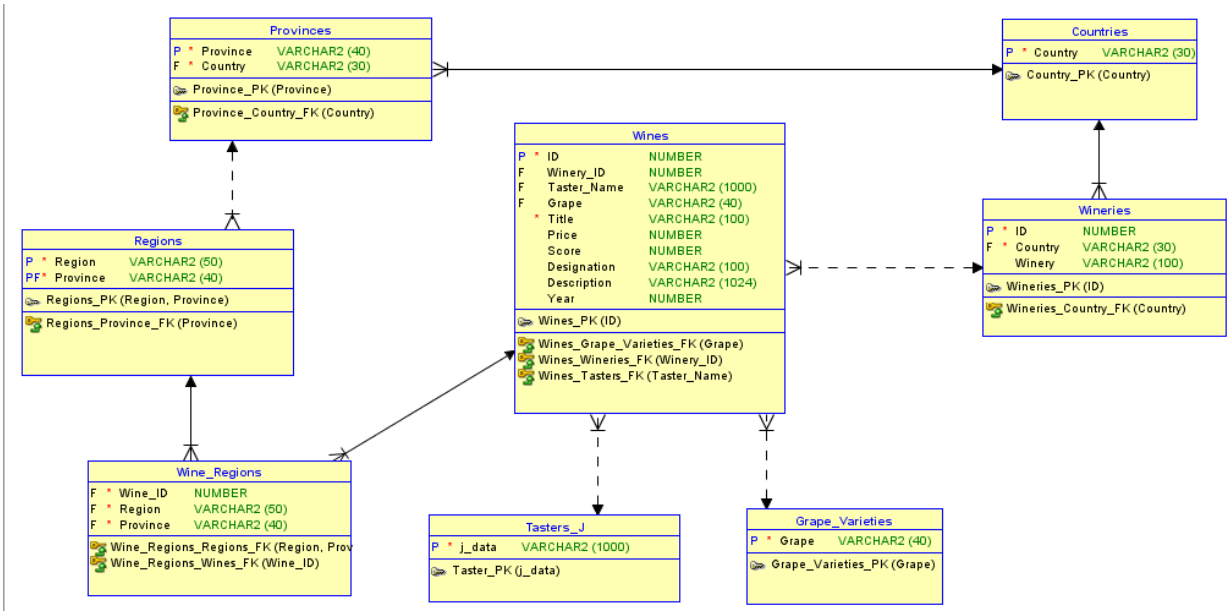
ORDER BY clause:

When you array fetch data from the database, the RDBM will write the first row in its entirety on the network. When it goes to write the second row it will only transmit column values that differ

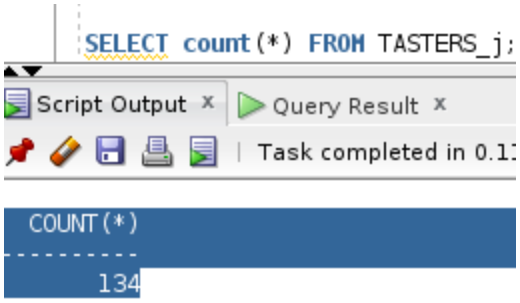
from the first row then repeat this process until the end. Therefore, the use of ORDER BY in our queries are very important so that the Oracle compression techniques are used to its fullest saving on transferring data across the network and consequently improving the query performance.

5. JSON Objects in Oracle

In order to test the new JSON features introduced to the Oracle 12c we have replaced the table TASTERS with a TASTERS_J that will be used for storing JSON objects.



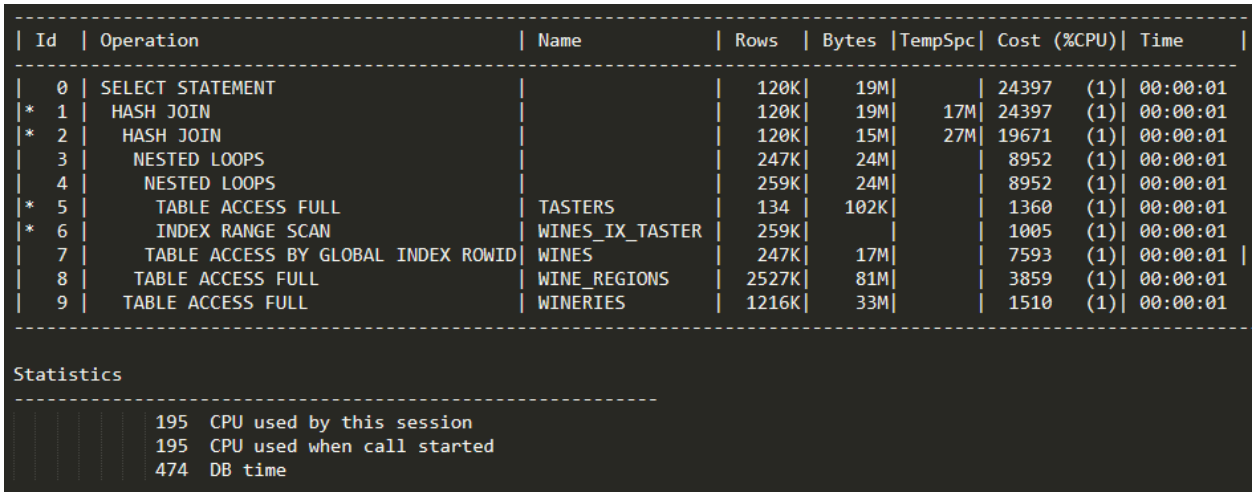
Created a table for TASTERS_J and added 134 tasters.



The performance will be tested using the following query:

```
SELECT  wnr.COUNTRY,
        wr.PROVINCE,
        wr.REGION,
        w.TITLE,
        w.DESIGNATION,
        w.SCORE,
        w.GRAPE,
        wnr.WINERY,
        t.TASTER_NAME,
        t.TWITTER
FROM    WINERIES wnr, TASTERS t, WINE_REGIONS wr, WINES w
WHERE   w.TASTER_NAME = t.TASTER_NAME
AND     wr.WINE_ID = w.ID
AND     w.WINERY_ID = wnr.ID
AND     t.TWITTER = '@vboone';
```

All the previous indexes were dropped and the old table tasters has been reduced to 134 tasters. When running using the relational TASTERS table:



```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name            | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                 |   120K|    19M|       | 24397   (1)| 00:00:01 |
|*  1 |  HASH JOIN                          |                 |   120K|    19M|    17M| 24397   (1)| 00:00:01 |
|*  2 |   HASH JOIN                         |                 |   120K|    15M|    27M| 19671   (1)| 00:00:01 |
|   3 |    NESTED LOOPS                     |                 |   247K|    24M|       |  8952   (1)| 00:00:01 |
|   4 |     NESTED LOOPS                    |                 |   259K|    24M|       |  8952   (1)| 00:00:01 |
|*  5 |      TABLE ACCESS FULL              | TASTERS         |   134 |   102K|       |  1360   (1)| 00:00:01 |
|*  6 |      INDEX RANGE SCAN               | WINES_IX_TASTER |   259K|       |       |  1005   (1)| 00:00:01 |
|   7 |     TABLE ACCESS BY GLOBAL INDEX ROWID| WINES         |   247K|    17M|       |  7593   (1)| 00:00:01 |
|   8 |    TABLE ACCESS FULL                | WINE_REGIONS    |  2527K|    81M|       |  3859   (1)| 00:00:01 |
|   9 |   TABLE ACCESS FULL                 | WINERIES        |  1216K|    33M|       |  1510   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          195  CPU used by this session
          195  CPU used when call started
          474  DB time
```

When running using the new TASTERS_J table:

```
SELECT
    wnr.country,
    wr.province,
    wr.region,
```

```
    w.title,
    w.designation,
    w.score,
    w.grape,
    wnr.winery,
    tj.j_data.name, --json
    tj.j_data.twitter --json
FROM
    tasters_j tj, wineries wnr, wine_regions wr, wines w
WHERE
    w.taster_name = tj.j_data.name
    AND wr.wine_id = w.id
    AND w.winery_id = wnr.id
    AND tj.j_data.twitter = '@vboone' --JSON data
ORDER BY
    wnr.country;
```

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name         | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |              |  1325M|    20T|       |  3001M  (1)| 32:34:02 |       |       |
|*  1 |  HASH JOIN                 |              |  1325M|    20T|   110M|  3001M  (1)| 32:34:02 |       |       |
|   2 |   TABLE ACCESS FULL        | WINE_REGIONS |  2527K|    81M|       |  3859   (1)| 00:00:01 |       |       |
|   3 |   MERGE JOIN               |              |  2725M|    42T|       |   810M  (1)| 08:47:36 |       |       |
|   4 |    SORT JOIN               |              |    13G|   204T|   198T|   810M  (1)| 08:47:32 |       |       |
|   5 |     MERGE JOIN CARTESIAN   |              |    13G|   204T|       |    16M  (1)| 00:10:45 |       |       |
|   6 |      NESTED LOOPS          |              | 10945 |   176M|       |  3653   (1)| 00:00:01 |       |       |
|   7 |       TABLE ACCESS FULL    | TASTERS_J    |   134 |  2209K|       |     3   (0)| 00:00:01 |       |       |
|*  8 |       JSONTABLE EVALUATION |              |       |       |       |            |          |       |       |
|   9 |      BUFFER SORT           |              |  1216K|    33M|       |    16M  (1)| 00:10:45 |       |       |
|  10 |       TABLE ACCESS FULL    | WINERIES     |  1216K|    33M|       |  1508   (1)| 00:00:01 |       |       |
|* 11 |    SORT JOIN               |              |  4979K|   360M|   915M|   106K  (1)| 00:00:05 |       |       |
|  12 |     PARTITION RANGE ALL    |              |  4979K|   360M|       | 18356   (1)| 00:00:01 |     1 |     6 |
|  13 |      TABLE ACCESS FULL     | WINES        |  4979K|   360M|       | 18356   (1)| 00:00:01 |     1 |     6 |
-----------------------------------------------------------------------------------------------------------------------

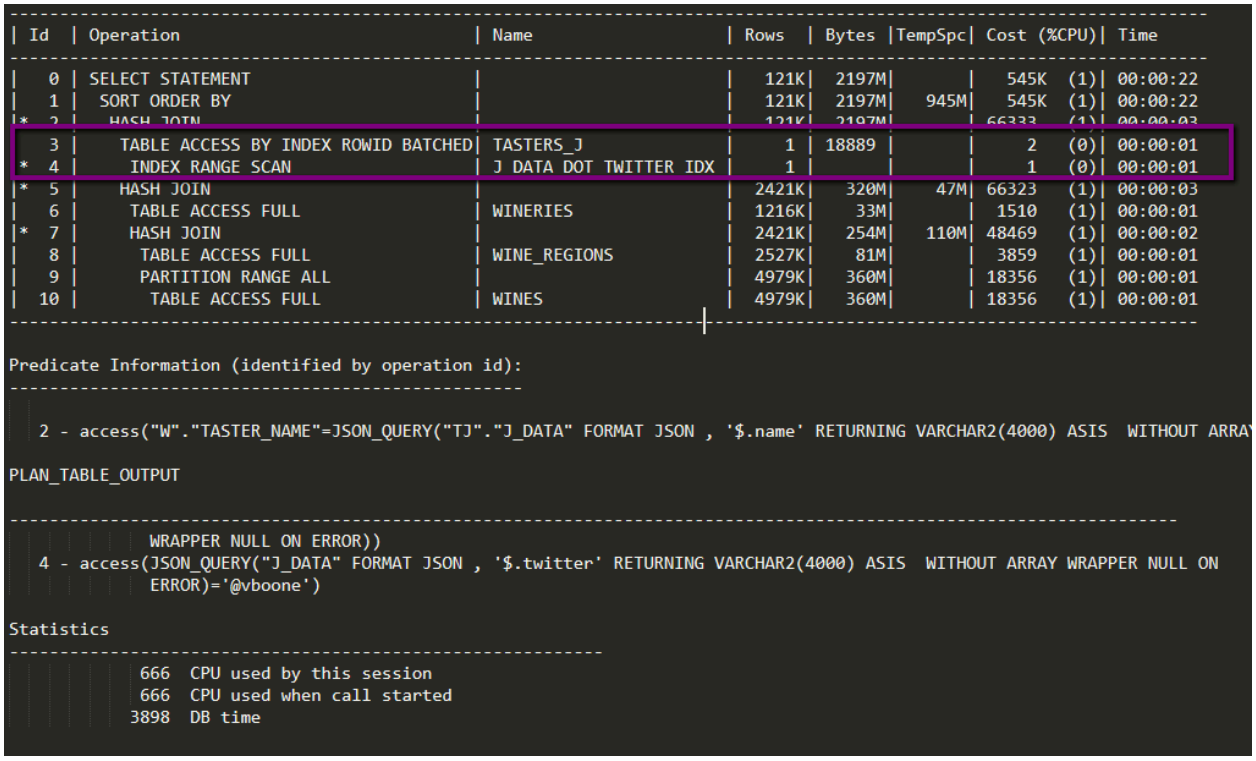
Statistics
-----------------------------------------------------------
          902  CPU used by this session
          902  CPU used when call started
         8003  DB time
```

Seems to have jumped up quite considerably when joining with a JSON object. Let's see if this can be improved.

Creating index on JSON table at the twitter object:

```
CREATE INDEX j_data_dot_twitter_idx ON tasters_j tj( tj.j_data.twitter);
```

The indexing has provided a good improvement onto the JSON table:



```
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                 | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                      |   121K|  2197M|       |   545K  (1)| 00:00:22 |
|   1 |  SORT ORDER BY                      |                      |   121K|  2197M|   945M|   545K  (1)| 00:00:22 |
|*  2 |   HASH JOIN                         |                      |   121K|  2197M|       | 66333   (1)| 00:00:03 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| TASTERS_J          |     1 | 18889 |       |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN                | J DATA DOT TWITTER IDX |     1 |       |       |     1   (0)| 00:00:01 |
|*  5 |    HASH JOIN                        |                      |  2421K|   320M|    47M| 66323   (1)| 00:00:03 |
|   6 |     TABLE ACCESS FULL               | WINERIES             |  1216K|    33M|       |  1510   (1)| 00:00:01 |
|*  7 |     HASH JOIN                       |                      |  2421K|   254M|   110M| 48469   (1)| 00:00:02 |
|   8 |      TABLE ACCESS FULL              | WINE_REGIONS         |  2527K|    81M|       |  3859   (1)| 00:00:01 |
|   9 |      PARTITION RANGE ALL            |                      |  4979K|   360M|       | 18356   (1)| 00:00:01 |
|  10 |       TABLE ACCESS FULL             | WINES                |  4979K|   360M|       | 18356   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------

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

   2 - access("W"."TASTER_NAME"=JSON_QUERY("TJ"."J_DATA" FORMAT JSON , '$.name' RETURNING VARCHAR2(4000) ASIS  WITHOUT ARRAY

PLAN_TABLE_OUTPUT

-------------------------------------------------------------------------------------------------------------------
              WRAPPER NULL ON ERROR))
   4 - access(JSON_QUERY("J_DATA" FORMAT JSON , '$.twitter' RETURNING VARCHAR2(4000) ASIS  WITHOUT ARRAY WRAPPER NULL ON
              ERROR)='@vboone')

Statistics
-----------------------------------------------------------
            666  CPU used by this session
            666  CPU used when call started
           3898  DB time
```

6. Appendix

```
CREATE TABLE countries (
    country   VARCHAR2(30) NOT NULL
);

ALTER TABLE countries ADD CONSTRAINT country_pk PRIMARY KEY ( country );

CREATE TABLE provinces (
    province   VARCHAR2(40) NOT NULL,
    country    VARCHAR2(30) NOT NULL
);

ALTER TABLE provinces ADD CONSTRAINT province_pk PRIMARY KEY ( province );

CREATE TABLE wineries (
    id         NUMBER NOT NULL,
    winery     VARCHAR2(100),
    country    VARCHAR2(30) NOT NULL
```

```
);

ALTER TABLE wineries ADD CONSTRAINT wineries_pk PRIMARY KEY ( id );

ALTER TABLE provinces
    ADD CONSTRAINT province_country_fk FOREIGN KEY ( country )
        REFERENCES countries ( country );

ALTER TABLE wineries
    ADD CONSTRAINT wineries_country_fk FOREIGN KEY ( country )
        REFERENCES countries ( country );


CREATE TABLE regions (
    region     VARCHAR2(50) NOT NULL,
    province   VARCHAR2(40) NOT NULL
);

ALTER TABLE regions ADD CONSTRAINT regions_pk PRIMARY KEY ( region,
                                                            province );

ALTER TABLE regions
    ADD CONSTRAINT regions_province_fk FOREIGN KEY ( province )
        REFERENCES provinces ( province );

CREATE TABLE grape_varieties (
    grape   VARCHAR2(30) NOT NULL
);

ALTER TABLE grape_varieties ADD CONSTRAINT grape_varieties_pk PRIMARY KEY (
grape );


CREATE TABLE tasters (
    taster_name   VARCHAR2(30) NOT NULL,
    twitter       VARCHAR2(30)
);

ALTER TABLE tasters ADD CONSTRAINT taster_pk PRIMARY KEY ( taster_name );

CREATE TABLE wines (
    id            NUMBER NOT NULL,
```

```
    winery_id     NUMBER,
    taster_name   VARCHAR2(30),
    grape         VARCHAR2(40),
    title         VARCHAR2(100) NOT NULL,
    price         NUMBER,
    score         NUMBER,
    designation   VARCHAR2(50),
    description   VARCHAR2(1024)
);

ALTER TABLE wines ADD CONSTRAINT wines_pk PRIMARY KEY ( id );

ALTER TABLE wines
    ADD CONSTRAINT wines_grape_varieties_fk FOREIGN KEY ( grape )
        REFERENCES grape_varieties ( grape );

ALTER TABLE wines
    ADD CONSTRAINT wines_tasters_fk FOREIGN KEY ( taster_name )
        REFERENCES tasters ( taster_name );

ALTER TABLE wines
    ADD CONSTRAINT wines_wineries_fk FOREIGN KEY ( winery_id )
        REFERENCES wineries ( id );


CREATE TABLE wine_regions (
    wine_id    NUMBER NOT NULL,
    region     VARCHAR2(50) NOT NULL,
    province   VARCHAR2(40) NOT NULL
);

ALTER TABLE wine_regions
    ADD CONSTRAINT wine_regions_regions_fk FOREIGN KEY ( region,
                                                         province )
        REFERENCES regions ( region,
                             province );

ALTER TABLE wine_regions
    ADD CONSTRAINT wine_regions_wines_fk FOREIGN KEY ( wine_id )
        REFERENCES wines ( id );
```

```
desc wine_regions;

-----------------------------------------
delete from regions;
DROP TABLE wine_regions;

-- POPULATE COUNTRIES TABLE
INSERT INTO COUNTRIES
SELECT DISTINCT COUNTRY FROM TESTDB.TEST_6
WHERE COUNTRY IS NOT NULL;

-- WINERIES TABLE
INSERT INTO WINERIES(WINERY,COUNTRY)
(SELECT DISTINCT WINERY,COUNTRY FROM TESTDB.TEST_6 WHERE COUNTRY IS NOT
NULL AND WINERY IS NOT NULL);

-- POPULATE PROVINCES TABLE
INSERT INTO PROVINCES(PROVINCE,COUNTRY)
(SELECT DISTINCT PROVINCE,COUNTRY FROM TESTDB.TEST_6 WHERE COUNTRY IS NOT
NULL AND PROVINCE IS NOT NULL);

-- POPULATE REGIONS TABLE
INSERT INTO REGIONS(REGION,PROVINCE)
(SELECT DISTINCT REGION_1,PROVINCE FROM TESTDB.TEST_6 WHERE flag is not
null AND REGION_1 IS NOT NULL AND PROVINCE IS NOT NULL);

-- POPULATE VARIETIES TABLE
INSERT INTO GRAPE_VARIETIES(GRAPE)
(SELECT DISTINCT VARIETY FROM TESTDB.TEST_6 WHERE VARIETY IS NOT NULL);

-- POPULATE TASTERS TABLE
INSERT INTO TASTERS(TASTER_NAME, TWITTER)
(SELECT DISTINCT TASTER_NAME, TASTER_TWITTER_HANDLE FROM TESTDB.TEST_6
WHERE TASTER_NAME IS NOT NULL);

--POPULATE WINES TABLE
INSERT INTO WINES(winery_id, taster_name, grape, title, price, description,
designation, score)
(SELECT WINERY_ID, TASTER_NAME, VARIETY, TITLE, PRICE, DESCRIPTION,
DESIGNATION, POINTS FROM TESTDB.TEST_6 WHERE TITLE IS NOT NULL);

--POPULATE WINE_REGIONS LINK TABLE
```

```
INSERT INTO WINE_REGIONS(WINE_ID, REGION, PROVINCE)
(
    SELECT w.ID, t.REGION_1, t.PROVINCE
    FROM TESTDB.TEST_6 t, WINES w
    WHERE t.TITLE  = w.TITLE
    AND t.DESCRIPTION = w.DESCRIPTION
    AND t.FLAG is not null
    AND t.REGION_1 IS NOT NULL
);
--POPULATE WINE_REGIONS LINK TABLE (Regions_2)
INSERT INTO WINE_REGIONS(WINE_ID, REGION, PROVINCE)
(
    SELECT w.ID, t.REGION_2, t.PROVINCE
    FROM TESTDB.TEST_6 t, WINES w
    WHERE t.TITLE  = w.TITLE
    AND t.DESCRIPTION = w.DESCRIPTION
    AND t.FLAG is null
    AND t.REGION_2 IS NOT NULL);

select count(*)from testdb.test_6 where region_2 is not null;


delete from wines where id in (
    select id from wines where title in (
        select title from wines
        group by title
        having count(*)>1));
--

select distinct region_2 from testdb.test_6;

INSERT INTO REGIONS(REGION,PROVINCE)
(SELECT DISTINCT REGION_2,PROVINCE FROM TESTDB.TEST_6 WHERE region_2 =
'Washington Other');

INSERT INTO REGIONS(REGION,PROVINCE)
(SELECT DISTINCT REGION_2,PROVINCE FROM TESTDB WHERE region_2 = 'New York
Other');

SELECT DISTINCT REGION_2,PROVINCE
FROM TESTDB.TEST_6
where region_2 is not null;
```

```
-------------------------

insert /*+ APPEND */ into tasters
select 'Test'||rownum, '@twitterx'||rownum
from dual
connect by level <= 1000000;

select * from wineries;

insert /*+ APPEND */ into wineries
select '216934'||rownum, 'Winery'||rownum, 'Italy'
from dual
connect by level <= 1000000;

select * from wines;

insert /*+ APPEND */ into wines
select '4108101'||rownum, '330', 'Jeff Jenssen', 'Pinot Noir', 'Wine
fakeb'||rownum, '67', '93', null, null
from dual
connect by level <= 1000000;

select * from wine_regions;
insert /*+ APPEND */ into wine_regions
select '3108101'||rownum, 'Atlantique', 'France Other'
from dual
connect by level <= 1000000;

insert into tasters (TASTER_NAME, TWITTER)
VALUES ('Victor Angelo', '@d16128783');
```