ISM6218-901 Final Project
James Long
USF Spring 2020

Contents

Executive Summary

This project is the culmination of effort performed over the span of a 12-week semester. The project incorporates several earlier assignments along with additional work that builds upon and extends those assignments. The work is not organized chronologically due to the required format as specified in Table 1. To minimize confusion, I labeled previous assignments and added introductory comments where appropriate to aid the reader in understanding the sequence of events.

Table 1
Organization and Point Weightings

Topic Area	Description	Points
Database Design	Logical database design, normalization, integrity constraints, design considerations for loading bulk data	30
Query Writing	Queries, a stored procedure	20
Performance Tuning	Btree+ and bitmap indexing, table and column statistics, optimizer modes, table partitioning	25
Other Topics	Loading bulk data via Python	25

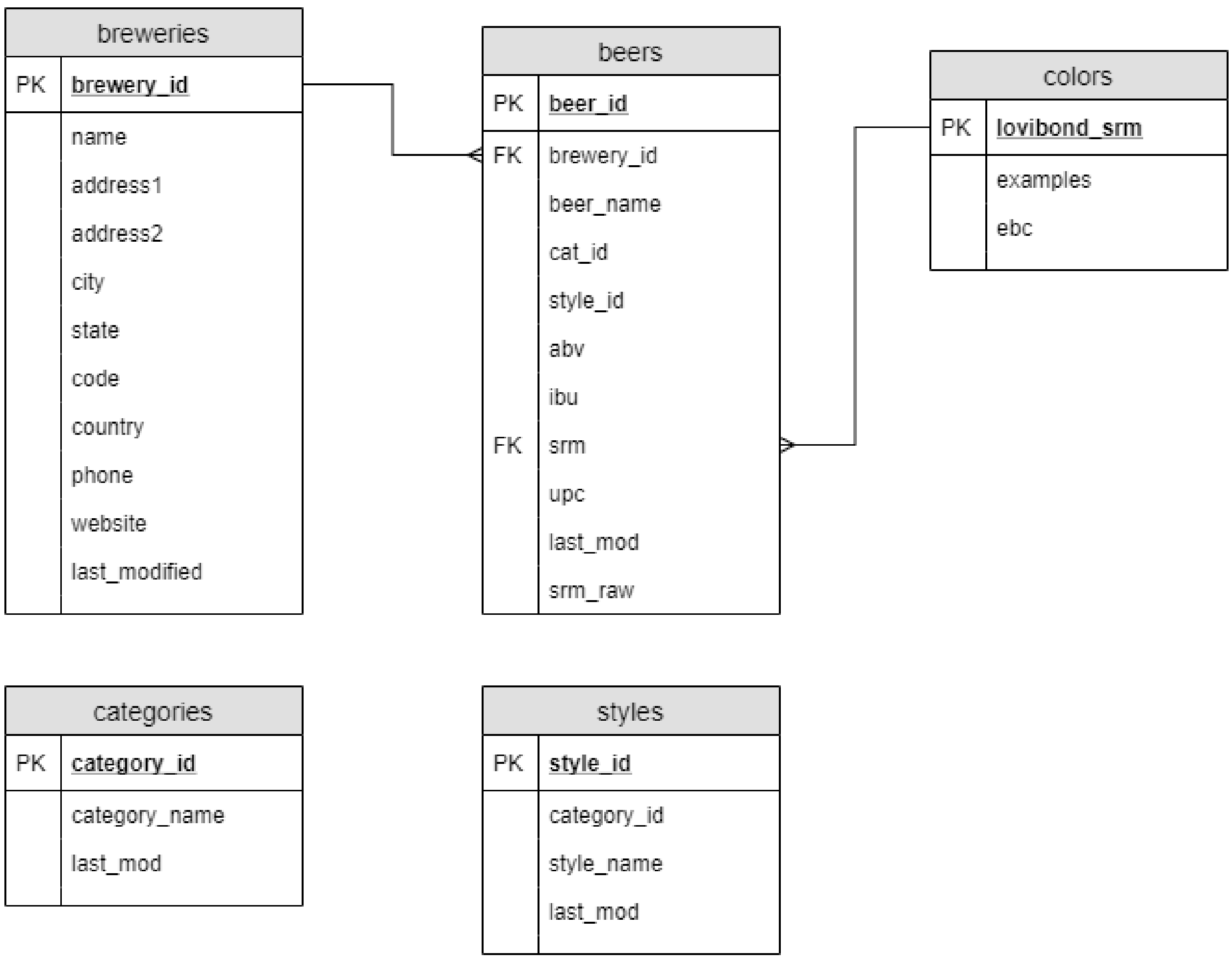
To maximize my learning outcomes, I chose to work alone this semester. Doing so forced me to complete 100% of the work as opposed to 25% in a group of four. While this benefited me, some of my solutions might be less polished than they would otherwise be had I the benefit of a peer group. Of course, this would not be the case if I were an experienced programmer and DBA. However, I am new to both, which is why I wanted to complete all the work for myself. The scope of work that I could undertake alone prevented me from fully exploring every obstacle I encountered. So, I left some exercises to future students of this course (search for “future students”). Hopefully, these will be of interest and provide some benefit to other students.

Of the choices given, I opted to remediate the beersdb schema rather than venture into the unknown. I began by copying the existing schema into my personal schema (DB870). Then I made several corrections before extending the design to accommodate competitions, sponsors, awards, and reviews (including numerical ratings and textual comments). The original data served as the basis for the first round of query and indexing experiments. I then loaded bulk data from the RateBeers dataset using a Python script. This new data served as the basis for the second round of query and performance experiments.

Database Design

Correct the Existing BeerDB Design (Assignment 2)

The following ERD depicts the beerdb schema as it currently exists on CDB9. This is slightly different than the ERD depicted on the assignment page in Canvas (which shows some FK constraints that are missing in reality).



The following SQL transactions were issued to replicate the beerdb tables and recreate the pre-existing key constraints within my personal schema (db870). All other constraints were "NOT NULL" and were preserved during the CTAS transactions.

```
CREATE TABLE
  beers
AS
  SELECT
    *
  FROM
```

```
      beerdb.beers;

CREATE TABLE
   breweries
AS
   SELECT
      *
   FROM
      beerdb.breweries;

CREATE TABLE
   categories
AS
   SELECT
      *
   FROM
      beerdb.categories;

CREATE TABLE
   colors
AS
   SELECT
      *
   FROM
      beerdb.colors;

CREATE TABLE
   styles
AS
   SELECT
      *
   FROM
      beerdb.styles;

ALTER TABLE
   beers
ADD CONSTRAINT
   beers_pk
PRIMARY KEY(beer_id);

ALTER TABLE
   breweries
ADD CONSTRAINT
   breweries_pk
PRIMARY KEY(brewery_id);

ALTER TABLE
   categories
```

```
ADD CONSTRAINT
   categories_pk
PRIMARY KEY(category_id);

ALTER TABLE
   colors
ADD CONSTRAINT
   colors_pk
PRIMARY KEY(lovibond_srm);

ALTER TABLE
   styles
ADD CONSTRAINT
   styles_pk
PRIMARY KEY(style_id);

ALTER TABLE
   beers
ADD CONSTRAINT
   breweries_fk
FOREIGN KEY
   (brewery_id)
REFERENCES
   breweries(brewery_id);

ALTER TABLE
   beers
ADD CONSTRAINT
   colors_fk
FOREIGN KEY
   (srm)
REFERENCES
   colors(lovibond_srm);
```

According to Craft Beer & Brewing (https://beerandbrewing.com/tools/color-calculator/), Lovibond is not the same thing as SRM. Using that website, I was able to determine the "lovibond_srm" values in the colors table are SRM values, not Lovibond values. So, the following change was made to correct the attribute name, enable natural joins between the beers and colors tables, and make the key relationship more obvious. The change did not break the existing key relationship.

```
ALTER TABLE
colors
RENAME COLUMN
lovibond_srm
TO
srm;
```

The following changes were made to eliminate NULL values in the srm attribute of the beers table and prevent new records with a NULL srm value.

```
INSERT INTO
   colors
VALUES
   (-1, 'unknown', -1);

UPDATE
   beers
SET
   srm = -1
WHERE
   srm IS NULL;

ALTER TABLE
   beers
ADD CONSTRAINT
   srm_not_null
CHECK ("SRM" IS NOT NULL);
```

Some records in the beers table had a value of -1 for style_id, but no such value existed in the styles table. Likewise, some records in the beers table had a value of -1 for category_id, but no such value existed in the categories table. I examined the relationship of style_id to category_id in the beers table as follows.

```
SELECT
   *
FROM
   beerdb.beers
WHERE
   cat_id = -1
   AND style_id <> -1;
```

No records were returned, which indicated all beers with a category_id of -1 also had a style_id of -1. Therefore, the following changes were made.

```
INSERT INTO
   categories
VALUES
   (-1, 'unknown', CURRENT_DATE);

INSERT INTO
   styles
VALUES
```

```
    (-1, -1, 'unknown', CURRENT_DATE);
```

The following foreign key constraints were then implemented to enable referential integrity. All attributes used as foreign keys in any table either already had a CHECK constraint to enforce NOT NULL or had a new such constraint defined as shown above (e.g., beers(srm)).

```
ALTER TABLE
    beers
ADD CONSTRAINT
    styles_fk
FOREIGN KEY
    (style_id)
REFERENCES
    styles(style_id);

ALTER TABLE
    styles
ADD CONSTRAINT
    categories_fk
FOREIGN KEY
    (category_id)
REFERENCES
    categories(category_id);
```

At this point, I realized the category_id attribute was stored redundantly in the beers and styles tables. This gives rise to potential data integrity problems:

- The user knows the category_id of a beer but not the style_id. The record is inserted into the beers table with a category_id of something other than -1 and a style_id of -1. Meanwhile, the styles table has only one record with style_id = -1, and the associated category_id is -1. We cannot force the user to also insert a new record into the styles table to define the new category_id / style_id pairing.
- The user knows the style_id of a beer but not the category_id. The user can run a query on the styles table to lookup the category_id associated with the known style_id. However, the user may instead choose to insert the record into the beers table with a style_id of something other than -1 and a category_id of -1. Meanwhile, the styles table has only one record with category_id = -1, and the associated style_id is -1. We cannot force the user to also insert a new record into the styles table to define the new category_id / style_id pairing.

This example illustrates why we normalize tables. The category_id of a given beer can be looked up via a join of the beers table with the styles table and then another join with the categories table. While the performance impact of two joins is obviously not ideal, I think the performance penalty of one or more CHECK constraints to ensure data integrity would be higher (assuming such constraints are even possible). So, I dropped the category_id attribute from the beers table.

```
-- cat_id NOT NULL
```

```
ALTER TABLE
   beers
DROP CONSTRAINT
   sys_c00108212;

ALTER TABLE
   beers
DROP COLUMN
   cat_id;
```

Without knowing which attribute was modified, the last_mod attribute is of little use. Furthermore, the attribute's name and format are inconsistent from table to table. Finally, this attribute tells us a fact about another non-key attribute, since the key value cannot change. By contrast, a date_created attribute would describe the key. So, I dropped the last_mod/last_modified attribute from all tables that contained it.

```
ALTER TABLE
   beers
DROP COLUMN
   last_mod;

ALTER TABLE
   categories
DROP COLUMN
   last_mod;

ALTER TABLE
   styles
DROP COLUMN
   last_mod;

ALTER TABLE
   breweries
DROP COLUMN
   last_modified;
```

The srm_raw attribute in the beers table is redundant with the srm attribute. The values do not all align, but most do. Those that do not align can be proven to have incorrect values in the srm_raw attribute according to the following query cross-referenced with the Craft Beer & Brewing website.

```
SELECT
   c.ebc,
   b.srm,
   b.srm_raw
FROM
   beers b
```

```
   INNER JOIN colors c
      ON b.srm = c.srm
WHERE
   b.beer_id = 5885
   OR b.beer_id = 5874;
```

So, I dropped the srm_raw attribute from the beers table.

```
ALTER TABLE
   beers
DROP COLUMN
   srm_raw;
```

The upc attribute in the beers table stores a value of 0 for all but six beers. Furthermore, only two values are shared by those six distinct beers. Thus, the upc attribute provides no useful information. So, I dropped the upc column.

```
ALTER TABLE
   beers
DROP COLUMN
   upc;
```

All remaining attributes in all tables are atomic except for examples in the colors table. So, I created a new table and migrated the examples attribute data. I used the example attribute as the primary key since an example can only belong to a single SRM value, thus requiring example to be unique. The Weissbier value appeared twice, so I did not migrate the instance associated with an SRM of 4, which I determined to be incorrect based on the results of an Internet search.

```
CREATE TABLE
   color_examples (
      example VARCHAR2(50 BYTE),
      srm NUMBER(2,0) NOT NULL,
      CONSTRAINT color_examples_pk PRIMARY KEY (example),
      CONSTRAINT color_fk FOREIGN KEY (srm) REFERENCES colors(srm)
      );

INSERT ALL
   INTO color_examples VALUES ('unknown', -1)
   INTO color_examples VALUES ('Pale Lager', 2)
   INTO color_examples VALUES ('Witbier', 2)
   INTO color_examples VALUES ('Pilsener', 2)
   INTO color_examples VALUES ('Berliner', 2)
   INTO color_examples VALUES ('Weisse', 2)
   INTO color_examples VALUES ('Maibock', 3)
   INTO color_examples VALUES ('Blonde Ale', 3)
   INTO color_examples VALUES ('American Pale Ale', 6)
```

```
   INTO color_examples VALUES ('India Pale Ale', 6)
   INTO color_examples VALUES ('Weissbier', 8)
   INTO color_examples VALUES ('Saison', 8)
   INTO color_examples VALUES ('English Bitter', 10)
   INTO color_examples VALUES ('ESB', 10)
   INTO color_examples VALUES ('Biere de Garde', 13)
   INTO color_examples VALUES ('Double IPA', 13)
   INTO color_examples VALUES ('Dark Lager', 17)
   INTO color_examples VALUES ('Vienna Lager', 17)
   INTO color_examples VALUES ('Marzen', 17)
   INTO color_examples VALUES ('Amber Ale', 17)
   INTO color_examples VALUES ('Brown Ale', 20)
   INTO color_examples VALUES ('Bock', 20)
   INTO color_examples VALUES ('Dunkel', 20)
   INTO color_examples VALUES ('Dunkelweizen', 20)
   INTO color_examples VALUES ('Irish Dry Stout', 24)
   INTO color_examples VALUES ('Doppelbock', 24)
   INTO color_examples VALUES ('Porter', 24)
   INTO color_examples VALUES ('Stout', 29)
   INTO color_examples VALUES ('Foreign Stout', 35)
   INTO color_examples VALUES ('Baltic Porter', 35)
   INTO color_examples VALUES ('Imperial Stout', 40)
SELECT * FROM dual;

ALTER TABLE
   colors
DROP COLUMN
   examples;
```

Brewers can own and operate multiple breweries. For example, the Molson Coors Beverage Company has seven breweries that offer tours (and possibly others that do not offer tours). Furthermore, a given beer can be produced at multiple breweries. Therefore, it is clear the intent of the breweries table is to track brewers, not breweries. So, I renamed the table brewers and renamed the brewery_id attribute brewer_id in the brewers and beers tables.

```
ALTER TABLE
   breweries
RENAME TO
   brewers;

ALTER TABLE
   brewers
RENAME COLUMN
   brewery_id
TO
   brewer_id;
```

```
ALTER TABLE
   beers
RENAME COLUMN
   brewery_id
TO
   brewer_id;
```

Given the high number of attributes named “name” that will be included in the extended design, I decided to rename all such attributes to be more descriptive. Only one attribute was affected in the current database schema.

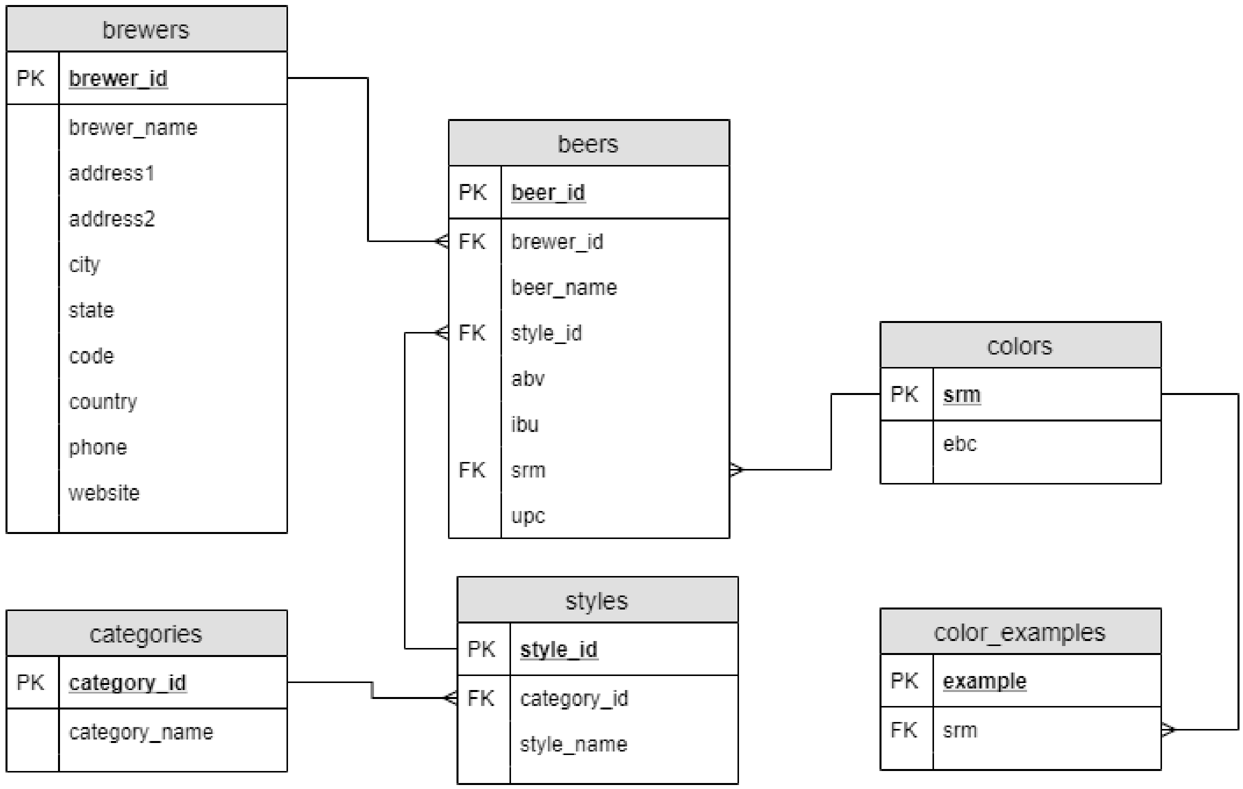
```
ALTER TABLE
   brewers
RENAME COLUMN
   name
TO
   brewer_name;
```

A beer’s name might not provide a unique identifier. For national or international beers, the name might suffice depending on state, national, and international trademark registrations. But this cannot be guaranteed. And for locally brewed beers, uniqueness is almost certainly not guaranteed. Thus, a composite natural key or synthetic key is required. As part of the extended design, the beers key must be used in the associative table beer_reviews, in the comments table, and in the beer_comp table. A composite key would increase storage requirements and could impact join performance. So, the short, numeric (synthetic) key that was already in place was kept.

Similarly, a brewer’s name might not provide a unique identifier for the same reasons that a beer’s name might not. This is less likely since the number of brewers is significantly lower than the number of beers. However, brewers periodically merge, reorganize, rebrand, and divest. Any of those events could precipitate a name change. Thus, the short, numeric (synthetic) key that was already in place was kept.

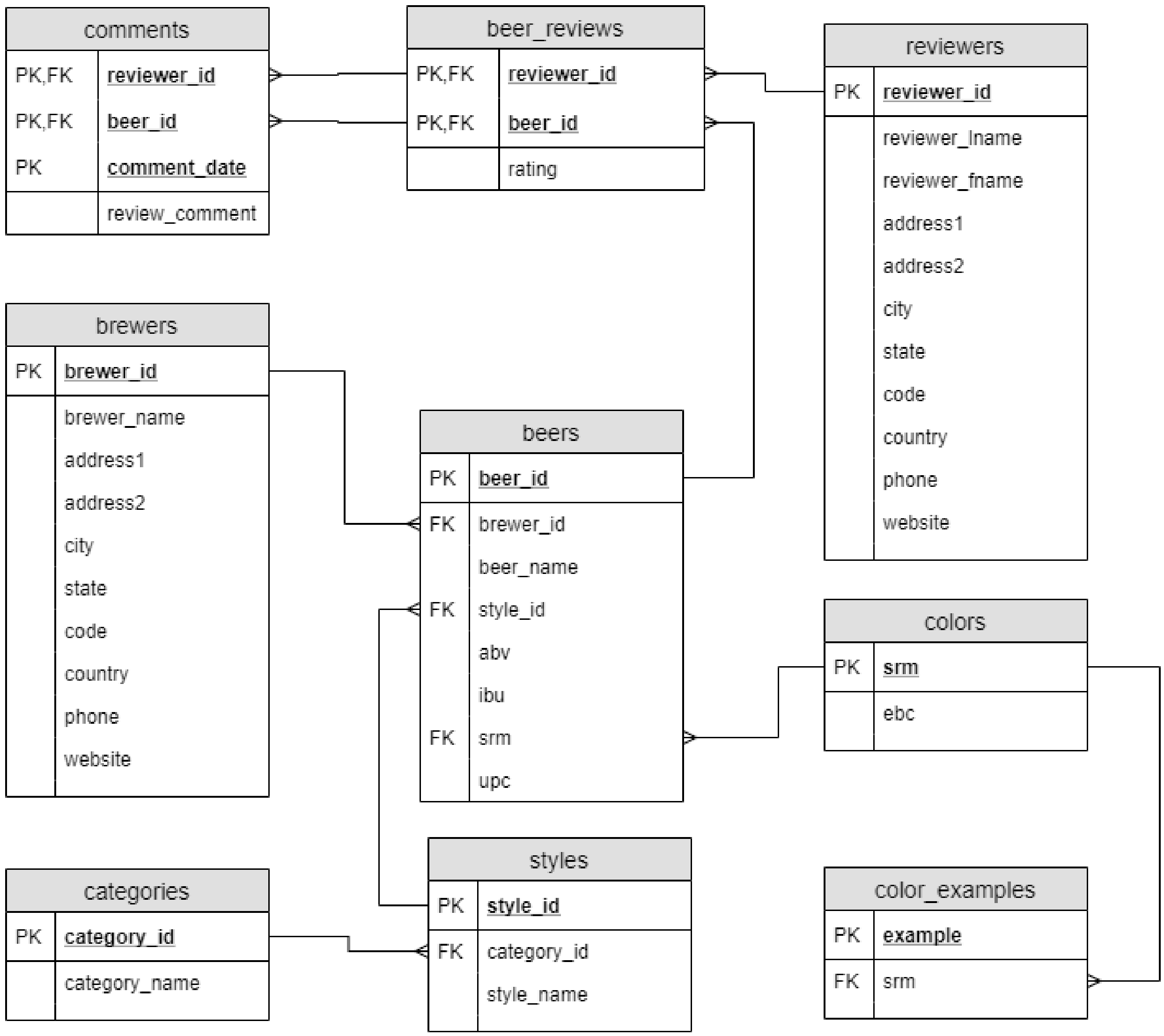
The style and category names are unique, but they are text strings up to 50 bytes long. Moreover, it is possible that additional names added in the future could be even longer. Therefore, using either name as a primary key could impact performance. Also, it is theoretically possible (but not likely) for a style or category name to change. So, the short, numeric (synthetic) keys that were already in place were kept.

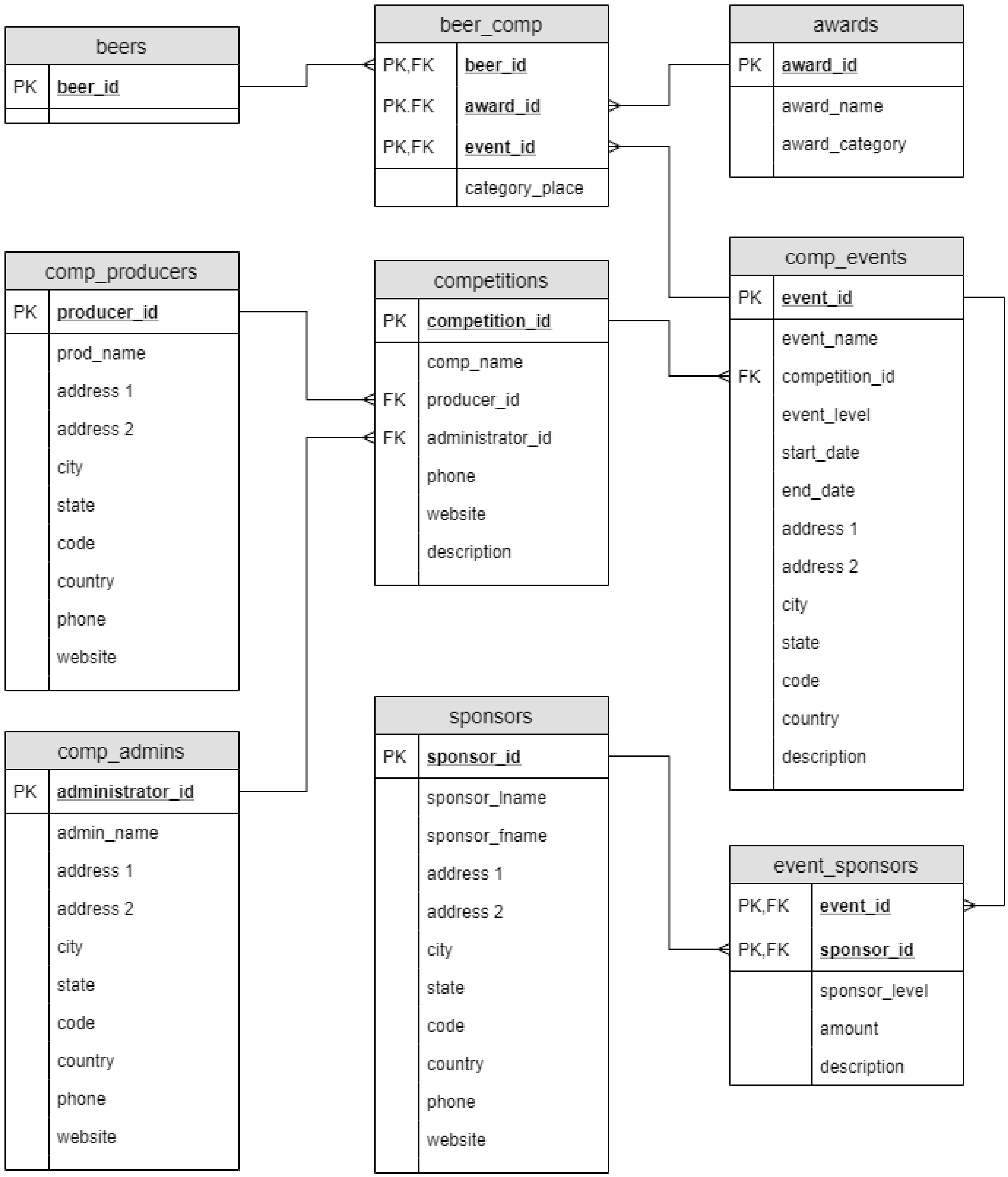
The resulting schema is in 3rd normal form. The updated ERD follows.



Update the Design to Fulfill the New Requirements (Assignment 2)

Next, I updated the design to incorporate the new requirements. The extended ERD follows. Other extension ideas for future students include tracking event judges and their credentials, tracking breweries and associating each with a brewer (especially those breweries that offer public tours), and tracking event revenues from sources other than sponsors (e.g., ticket sales).


comments
PK,FK reviewer_id
PK,FK beer_id
PK comment_date
review_comment
beer_reviews
PK,FK reviewer_id
PK,FK beer_id
rating
reviewers
PK reviewer_id
reviewer_lname
reviewer_fname
address1
address2
city
state
code
country
phone
website
brewers
PK brewer_id
brewer_name
address1
address2
city
state
code
country
phone
website
beers
PK beer_id
FK brewer_id
beer_name
FK style_id
abv
ibu
FK srm
upc
colors
PK srm
ebc
categories
PK category_id
category_name
styles
PK style_id
FK category_id
style_name
color_examples
PK example
FK srm



The beer_reviews table is an associative table, so the choice for primary key was made using conventional wisdom. The chosen rating is based on the Beer Connoisseur website (https://beerconnoisseur.com/how-we-score).

Beer reviewers are individuals, so the beer reviewer's name will not provide a unique identifier. Thus, a composite natural key or synthetic key is required. The reviewers key must be used in the associative table beer_reviews and in the comments table, which would increase storage requirements and could impact join performance. So, a short, numeric (synthetic) key was chosen. Some reviewers will be professional beer critics, so a website attribute (nullable) is included in the reviewers table.

Comments are tied to a specific review, so the obvious choice for primary key includes the composite primary key of the beer_reviews table plus the date attribute of the comment (which includes both date and time, so multiple comments can be submitted per day). Comments could be stored in the beer_reviews table, but the rating attribute (a small integer) would be redundantly stored. The alternative is to store comments in a separate table referencing the reviewer_id and beer_id in the beer_reviews table. This results in even more redundant data, but the only redundant data are key attributes. While this proper 3rd normal form, it might make sense (when implementing this design) to denormalize the comments table into the beer_reviews table (with date as part of the primary key) to reduce the total amount of redundant data and preclude the need for an extra join operation. However, this would introduce the possibility of inconsistent rating values across related records.

A competition's name might not be unique. Small, local competitions might be named after the city or county in which they take place (e.g., Springfield Beer Competition). Many cities and counties share the same name. Therefore, name and producer_id would be needed to form a natural key. However, that would require a composite foreign key in the comp_events table, which would increase storage requirements and could impact join performance. Also, a competition's name could change over time. So, a short, numeric (synthetic) key was chosen.

If two or more companies jointly produce a competition, I assume they form a Limited Liability Partnership (LLP) or other similar legal entity to facilitate the joint venture. Thus, each competition has a single producer. Of course, a producer may produce many different competitions.

A competition may be administered by a professional organization or by the producer. Either way, I assume a single administrator is associated with each competition. For producer-administered competitions, the administrator_id attribute is left empty (NULL).

Competition producers and administrators are assumed to be organizations in most (if not all) cases. As stated previously about brewers, organizations can be expected to change their names over time. Therefore, a short, numeric (synthetic) key was chosen for the comp_producers and comp_admins tables.

Events need to be stored separately from competitions because event names, dates, locations, sponsors, etc. can vary from event to event for a given competition. An event's name might be reused from year to year (e.g., Central Valley Annual Local Brew-off), so event names might not be unique. Thus, a natural key would need to be a composite of name and something like start_date. But the many-to-many relationship between comp_events and sponsors would require the composite key to be used in the associative table event_sponsors, which would increase storage requirements and could impact join performance. So, a short, numeric (synthetic) key was chosen. A description attribute (nullable) is included in case additional details are desirable (e.g., "25th anniversary of the competition"). I assume events do not have a website. Instead, they rely on the competition's website for all marketing and registrations. The event_level attribute should store a limited set of values such as local, state, regional, national, and international. To ensure only these values are used, a separate lookup table named event_levels could be created with attributes event_level_id and event_level_name (similar to the categories, styles, color_examples, and colors tables). Additionally, read-only access restrictions could be placed on all such tables to prevent proliferation of lookup values; only admins would have read-write access. These options are left to future classes to implement.

The event_sponsors table is an associative table, so the choice for primary key was made using conventional wisdom. Event sponsors may be grouped into levels (e.g., Platinum, Gold, etc.), so a sponsor_level attribute (nullable) was created to capture this information. An amount attribute (nullable) was also created to capture the dollar amount (or dollar value equivalent) of the sponsorship. A description attribute (nullable) was also created to capture any unique information that may be of value (e.g., “advertiser” or “loaned tables and chairs free of charge”).

A sponsor’s name might not provide a unique identifier. Sponsors might be individuals for small, local competition events. Thus, a composite natural key or synthetic key is required. The sponsors key must be used in the associative table event_sponsors, and corporate sponsors may change names over time. For the reasons stated previously, a short, numeric (synthetic) key was chosen. For non-individual sponsors, I assume the organization’s name will be stored in the sponsor_lname attribute, and the sponsor_fname attribute will be empty.

The beer_comp table is an associative table, so the choice for primary key was made using conventional wisdom. A beer may win zero, one, or many awards across categories and/or within a single category at a single competition event. My design accommodates all these scenarios. The beer_comp table includes award_id as part of the primary key. If a beer does not win an award, an award_id value of -1 is used, which has an award_name of “no award” and an award_category of “no award category”. For such a beer, the category_place attribute in the beer_comp table represents the beer’s place within the primary category in which the beer competed. For winning beers, the category_place attribute represents the beer’s placement in the associated award_category. If a beer wins multiple awards in a single category, the award_id and award_name will differ but the award_category and category_place will be the same. If a beer wins multiple awards across categories, the award_id, award_name, award_category, and category_place will all differ. The award grade (e.g., gold, silver, 1st place, 2nd place, etc.) is assumed to be reflected in the award’s name. The award level (e.g., local, state, regional, etc.) is captured in the comp_events table instead of the awards table because the level applies to all awards at the event (i.e., level tell us a fact about the event). The level is not captured in the competitions table because a competition may hold numerous events at different levels (local, state, regional, etc.) leading up to a national or international final event.

The award_name attribute is likely to hold many instances of “gold medal”, “first place”, and similar such names. Likewise, award_category is likely to contain many common values since different competitions are likely to categorize beers in a similar manner. Therefore, a short, numeric (synthetic) key was chosen. This also improves join performance with the beer_comp table. The award_category is set to NOT NULL. For an award that does not pertain to a category, this attribute can be set to “overall”.

All name attributes are set to NOT NULL.

In newly created tables, all primary keys that can be auto-generated are auto-generated. In pre-existing tables, no changes were made to the way primary keys are generated because such a change would require dropping existing primary key columns and re-creating them as ID columns.

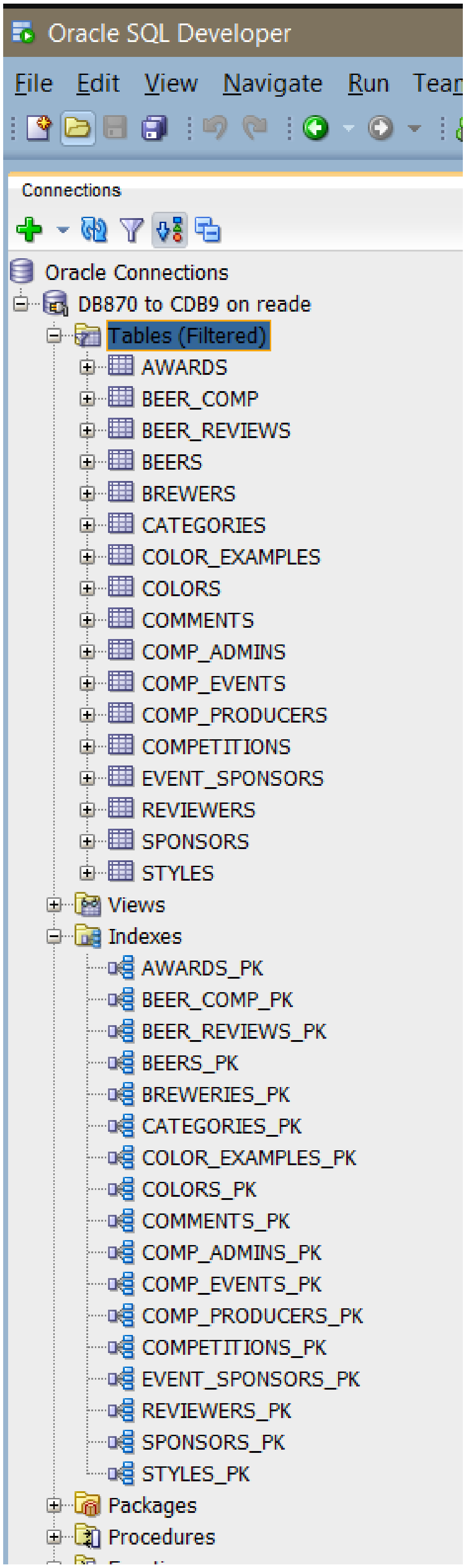
The design was implemented with the following code.

```
CREATE TABLE
   reviewers (
      reviewer_id NUMBER GENERATED by default on null as IDENTITY,
      reviewer_lname VARCHAR2(50 BYTE) NOT NULL,
      reviewer_fname VARCHAR2(50 BYTE) NOT NULL,
      address1 VARCHAR2(255 BYTE),
      address2 VARCHAR2(255 BYTE),
      city VARCHAR2(255 BYTE),
      state VARCHAR2(255 BYTE),
      code VARCHAR2(25 BYTE),
      country VARCHAR2(255 BYTE),
      phone VARCHAR2(50 BYTE),
      website VARCHAR2(255 BYTE),
      CONSTRAINT reviewers_pk PRIMARY KEY (reviewer_id)
      );

CREATE TABLE
   beer_reviews (
      reviewer_id NUMBER,
      beer_id NUMBER,
      rating NUMBER(3,0) NOT NULL,
      CONSTRAINT beer_reviews_pk PRIMARY KEY (reviewer_id, beer_id),
      CONSTRAINT reviewers_fk FOREIGN KEY (reviewer_id) REFERENCES reviewers(reviewer_id),
      CONSTRAINT beers_fk FOREIGN KEY (beer_id) REFERENCES beers(beer_id)
      );

CREATE TABLE
   comments (
      reviewer_id NUMBER,
      beer_id NUMBER,
      comment_date DATE,
      review_comment VARCHAR2(255 BYTE) NOT NULL,
      CONSTRAINT comments_pk PRIMARY KEY (reviewer_id, beer_id, comment_date),
      CONSTRAINT beer_reviews_fk FOREIGN KEY (reviewer_id, beer_id) REFERENCES
          beer_reviews(reviewer_id, beer_id)
      );

CREATE TABLE
   comp_producers (
      producer_id NUMBER GENERATED by default on null as IDENTITY,
      prod_name VARCHAR2(255 BYTE) NOT NULL,
      address1 VARCHAR2(255 BYTE),
      address2 VARCHAR2(255 BYTE),
      city VARCHAR2(255 BYTE),
      state VARCHAR2(255 BYTE),
      code VARCHAR2(25 BYTE),
      country VARCHAR2(255 BYTE),
      phone VARCHAR2(50 BYTE),
```

```
      website VARCHAR2(255 BYTE),
      CONSTRAINT comp_producers_pk PRIMARY KEY (producer_id)
      );

CREATE TABLE
   comp_admins (
      administrator_id NUMBER GENERATED by default on null as IDENTITY,
      admin_name VARCHAR2(255 BYTE) NOT NULL,
      address1 VARCHAR2(255 BYTE),
      address2 VARCHAR2(255 BYTE),
      city VARCHAR2(255 BYTE),
      state VARCHAR2(255 BYTE),
      code VARCHAR2(25 BYTE),
      country VARCHAR2(255 BYTE),
      phone VARCHAR2(50 BYTE),
      website VARCHAR2(255 BYTE),
      CONSTRAINT comp_admins_pk PRIMARY KEY (administrator_id)
      );

CREATE TABLE
   competitions (
      competition_id NUMBER GENERATED by default on null as IDENTITY,
      comp_name VARCHAR2(255 BYTE) NOT NULL,
      producer_id NUMBER,
      administrator_id NUMBER,
      phone VARCHAR2(50 BYTE),
      website VARCHAR2(255 BYTE),
      description VARCHAR2(255 BYTE),
      CONSTRAINT competitions_pk PRIMARY KEY (competition_id),
      CONSTRAINT comp_producers_fk FOREIGN KEY (producer_id) REFERENCES
         comp_producers(producer_id),
      CONSTRAINT comp_admins_fk FOREIGN KEY (administrator_id) REFERENCES
         comp_admins(administrator_id)
      );

CREATE TABLE
   comp_events (
      event_id NUMBER GENERATED by default on null as IDENTITY,
      event_name VARCHAR2(255 BYTE) NOT NULL,
      competition_id NUMBER,
      event_level VARCHAR2(255 BYTE) NOT NULL,
      start_date DATE NOT NULL,
      end_date DATE NOT NULL,
      address1 VARCHAR2(255 BYTE),
      address2 VARCHAR2(255 BYTE),
      city VARCHAR2(255 BYTE),
      state VARCHAR2(255 BYTE),
      code VARCHAR2(25 BYTE),
```

```
        country VARCHAR2(255 BYTE),
        description VARCHAR2(255 BYTE),
        CONSTRAINT comp_events_pk PRIMARY KEY (event_id),
        CONSTRAINT competitions_fk FOREIGN KEY (competition_id) REFERENCES
          competitions(competition_id)
        );

CREATE TABLE
    sponsors (
        sponsor_id NUMBER GENERATED by default on null as IDENTITY,
        sponsor_lname VARCHAR2(255 BYTE) NOT NULL,
        sponsor_fname VARCHAR2(255 BYTE),
        address1 VARCHAR2(255 BYTE),
        address2 VARCHAR2(255 BYTE),
        city VARCHAR2(255 BYTE),
        state VARCHAR2(255 BYTE),
        code VARCHAR2(25 BYTE),
        country VARCHAR2(255 BYTE),
        phone VARCHAR2(50 BYTE),
        website VARCHAR2(255 BYTE),
        CONSTRAINT sponsors_pk PRIMARY KEY (sponsor_id)
        );

CREATE TABLE
    event_sponsors (
        event_id NUMBER,
        sponsor_id NUMBER,
        sponsor_level VARCHAR2(255 BYTE),
        amount NUMBER,
        description VARCHAR2(255 BYTE),
        CONSTRAINT event_sponsors_pk PRIMARY KEY (event_id, sponsor_id),
        CONSTRAINT comp_events_fk FOREIGN KEY (event_id) REFERENCES comp_events(event_id),
        CONSTRAINT sponsors_fk FOREIGN KEY (sponsor_id) REFERENCES sponsors(sponsor_id)
        );

CREATE TABLE
    awards (
        award_id NUMBER GENERATED by default on null as IDENTITY,
        award_name VARCHAR2(255 BYTE) NOT NULL,
        award_category VARCHAR2(255 BYTE) NOT NULL,
        CONSTRAINT awards_pk PRIMARY KEY (award_id)
        );

ALTER TABLE
    awards
DISABLE CONSTRAINT
    awards_pk;
```

```
INSERT INTO
   awards
VALUES
   (-1, 'no award', 'no award category');

ALTER TABLE
   awards
ENABLE CONSTRAINT
   awards_pk;

CREATE TABLE
   beer_comp (
      beer_id NUMBER,
      award_id NUMBER,
      event_id NUMBER,
      category_place NUMBER,
      CONSTRAINT beer_comp_pk PRIMARY KEY (beer_id, award_id, event_id),
      CONSTRAINT beer_fk FOREIGN KEY (beer_id) REFERENCES beers(beer_id),
      CONSTRAINT awards_fk FOREIGN KEY (award_id) REFERENCES awards(award_id),
      CONSTRAINT comp_event_fk FOREIGN KEY (event_id) REFERENCES comp_events(event_id)
      );
```


Oracle SQL Developer
File Edit View Navigate Run Tea
Connections
Oracle Connections
DB870 to CDB9 on reade
Tables (Filtered)
AWARDS
BEER_COMP
BEER_REVIEWS
BEERS
BREWERS
CATEGORIES
COLOR_EXAMPLES
COLORS
COMMENTS
COMP_ADMINS
COMP_EVENTS
COMP_PRODUCERS
COMPETITIONS
EVENT_SPONSORS
REVIEWERS
SPONSORS
STYLES
Views
Indexes
AWARDS_PK
BEER_COMP_PK
BEER_REVIEWS_PK
BEERS_PK
BREWERIES_PK
CATEGORIES_PK
COLOR_EXAMPLES_PK
COLORS_PK
COMMENTS_PK
COMP_ADMINS_PK
COMP_EVENTS_PK
COMP_PRODUCERS_PK
COMPETITIONS_PK
EVENT_SPONSORS_PK
REVIEWERS_PK
SPONSORS_PK
STYLES_PK
Packages
Procedures

Design Considerations for Loading Bulk Data

Initially, I used the data pre-loaded in the beerdb database. In various parts of this project, I also loaded data manually with SQL INSERT statements to facilitate experiments. However, that solution proved inadequate in some respects. So, between assignments three and four, I loaded data from the Stanford SNAP / RateBeer dataset (available at https://www.ratebeer.com/api.asp) into the beers, brewers, styles, reviewers, beer_reviews, and comments tables. Basic statistics about the RateBeer dataset are available at https://snap.stanford.edu/data/web-ratebeer.html and shown below.

Number of reviews	2,924,127
Number of users	40,213
Number of beers	110,419
Users with > 50 reviews	4,798
Median # of words per review	54
Timespan	Apr 2000 - Nov 2011

A sample of the data provided for each beer follows.

beer/name: John Harvards Simcoe IPA
beer/beerId: 63836
beer/brewerId: 8481
beer/ABV: 5.4
beer/style: India Pale Ale (IPA)
review/appearance: 4/5
review/aroma: 6/10
review/palate: 3/5
review/taste: 6/10
review/overall: 13/20
review/time: 1157587200
review/profileName: hopdog
review/text: On tap at the Springfield, PA location. Poured a deep and cloudy orange (almost a copper) color with a small sized off white head. Aromas or oranges and all around citric. Tastes of oranges, light caramel and a very light grapefruit finish. I too would not believe the 80+ IBUs - I found this one to have a very light bitterness with a medium sweetness to it. Light lacing left on the glass.

Since beer names are not guaranteed to be unique, some of the provided beer names might have been duplicates of beer names pre-populated in the beers table. Also, the beer_name attribute in the beers table did not have a pre-defined UNIQUE constraint, which indicated duplicate beer names might have already existed in the beers table. However, each (brewer_id, beer_name) tuple should be unique. So, I attempted to implement that constraint on the beers table. Eight duplicates were found, so I identified and removed them as follows. The duplicate records with the least amount of valid data were deleted.

```
SELECT
```

```
   brewer_id,
   beer_name,
   COUNT(*)
FROM
   beers
GROUP BY
   brewer_id,
   beer_name
   HAVING COUNT(*) > 1
ORDER BY
   brewer_id;

SELECT
   b.beer_id,
   b.brewer_id,
   br.brewer_name,
   br.country,
   b.beer_name,
   b.style_id,
   b.abv,
   b.ibu,
   b.srm,
   b.upc
FROM
   beers b
   INNER JOIN brewers br
      ON b.brewer_id=br.brewer_id
WHERE
   b.brewer_id IN (49,236,1391,1417)
   AND b.beer_name IN ('Guinness Draught','Avalanche Amber','Dos Perros','Hefeweizen',
                       'Onward Stout','Pale Ale','Sly Rye Porter','Watershed IPA')
ORDER BY
   b.beer_name;

DELETE FROM
   beers
WHERE
   beer_id IN (3721,4457,4473,4462,4460,4459,4458,5896);
```

Then I implemented the UNIQUE constraint on beers(brewer_id, beer_name) as follows.

```
ALTER TABLE
   beers
ADD CONSTRAINT
   brewer_beer_unq
UNIQUE (brewer_id, beer_name);
```

Some of the provided beer names contained “(” and “)” representing “(“ and “)” respectively. Other encodings may have been present as well. I did not convert these strings. Cleaning up the beer_names values was left as an exercise for future students.

The RateBeer dataset provided a brewer_id field but no other information about the brewers. Nonetheless, the provided brewer_id field served to prevent the conflation of distinct beers sharing a duplicate name. Therefore, I removed the NOT NULL constraint on brewers(brewer_name). In the future, RateBeer.com might make available more information about brewers, at which time the brewer table can be populated by matching records on brewer_id. Afterward, the NOT NULL constraint can be re-implemented on brewer_name. My load script was written such that any records in the RateBeer dataset that did not have a brewerId value did not get inserted into the database. However, my full scan of the file indicated no such records existed.

```
-- brewer_name NOT NULL
ALTER TABLE
   brewers
DROP CONSTRAINT
   sys_c00108215;
```

Some of the provided beerId values could have been duplicates of pre-populated beer_id values in the beers table. The RateBeer dataset claimed to have less than 111,000 unique beerIds. However, there was no way to know if the beerIds were numbered sequentially and contiguously starting at 0. To resolve this potential issue, I temporarily disabled the beer_id foreign key constraint on the beer_comp and beer_reviews tables. Then I updated all beer_id values in the beers table by adding 500,000 to each record. Then I repeated the update on the beer_comp table. Finally, I re-enabled the beer_id foreign key constraint on the beer_comp and beer_reviews tables. My solution avoided collisions as new records were added and made it very easy to distinguish new from pre-existing records using the beer_id range. In the event a RateBeer record had no beerId, the record was not inserted into the database. However, my full scan of the file indicated no such records existed.

```
ALTER TABLE
   beer_comp
DISABLE CONSTRAINT
   beer_fk;

ALTER TABLE
   beer_reviews
DISABLE CONSTRAINT
   beers_fk;

UPDATE
   beers
SET
   beer_id = (beer_id + 500000);

UPDATE
```

```
   beer_comp
SET
   beer_id = (beer_id + 500000);

ALTER TABLE
   beer_reviews
ENABLE CONSTRAINT
   beers_fk;

ALTER TABLE
   beer_comp
ENABLE CONSTRAINT
   beer_fk;
```

Similarly, some of the provided brewerId values could have been duplicates of pre-populated brewer_id values in the brewers table. I implemented the same solution as for beer_ids.

```
ALTER TABLE
   beers
RENAME CONSTRAINT
   breweries_fk
TO
   brewers_fk;

ALTER TABLE
   beers
DISABLE CONSTRAINT
   brewers_fk;

UPDATE
   brewers
SET
   brewer_id = (brewer_id + 500000);

UPDATE
   beers
SET
   brewer_id = (brewer_id + 500000);

ALTER TABLE
   beers
ENABLE CONSTRAINT
   brewers_fk;
```

Some of the provided abv values were "-" indicating an unknown abv. Since unknown abv values were already represented in the beers table as 0, I converted all "-" values to 0 upon insertion into the beers table.

The RateBeer dataset provided a style field containing a style name but no category information. Furthermore, many (perhaps all) of the provided style names differed from the style names already in the beerdb styles table. And for any matching values, there was no way to know if the category mapping pre-established in beerdb was valid for the new records (since style names can be legitimately duplicated across categories). So, I inserted all RateBeer style names as new records in the styles table with new (generated) style_ids mapped to a category_id of -1. The styles_id attribute proved to be too small, so I had to increase the size.

```
ALTER TABLE
   styles
MODIFY
   style_id NUMBER;
```

The RateBeer dataset provided review data for each beer including separate ratings for appearance, aroma, palate, taste, and overall. To complicate matters, each rating used a different scale. So, I normalized the rating scales to 100 and modified the beer_reviews table as follows. None of the new attributes was given a NOT NULL constraint, but the previously defined NOT NULL constraint on rating (now overall_rating) was retained. In the event a RateBeer record had no overall_rating, the review and comment portions of the record were not inserted into the database. However, my full scan of the file indicated no such records existed.

```
ALTER TABLE
   beer_reviews
RENAME COLUMN rating
TO overall_rating;

ALTER TABLE
   beer_reviews
ADD (
   appearance_rating NUMBER(3,0),
   aroma_rating NUMBER(3,0),
   palate_rating NUMBER(3,0),
   taste_rating NUMBER(3,0)
   );
```

The time stamp of each review was a large integer representing a date between April 2000 and November 2011. Decoding and transforming those values into a more user-friendly format was left as an exercise for future students. I loaded this field unmodified into comments(comment_date). To do so, I first had to modify the data type of the comment_date attribute as follows.

```
ALTER TABLE
   comments
```

```
MODIFY
   comment_date NUMBER;
```

In the event a RateBeer record had no comment_date, the record was not inserted into the comments table, but other portions of the record were inserted into appropriate tables. However, my full scan of the file indicated no such records existed. By contrast, many RateBeer records existed with no data in the review/text field. Obviously, such records were not inserted into the comments table.

Most of the review comments exceeded 255 bytes, and some exceeded 4000 bytes. So, I increased the size of comments(review_comment) by changing the data type to LONG.

```
ALTER TABLE
   comments
MODIFY
   review_comment LONG;

ALTER TABLE
   comments
MODIFY
   (review_comment NOT NULL);
```

The reviewers were identified by a profileName field instead of first and last name. I mapped the profileName field into reviewers(reviewer_lname) and dropped the NOT NULL constraint on reviewers(reviewer_fname). In the event a RateBeer record had no profileName, the review and comment portions of the record were not inserted into the database. However, my full scan of the file indicated no such records existed. Since real names can be duplicated (even first/last pairings), I did not implement any UNIQUE constraints on any combination of the name attributes.

```
-- fname NOT NULL
ALTER TABLE
   reviewers
DROP CONSTRAINT
   sys_c00108390;
```

All records inserted into the beers table were given a value of -1 (unknown) for srm. Since a value of 0 was already used in the beers table to represent an unknown ibu value, I assigned an ibu value of 0 to all inserted records.

I did not log SQL errors during the load process. With so many records, losing a few due to errors is not a problem. Programatically identifying any missing records and inserting them into the database might be a good exercise for future students. Similarly, some records in the RateBeer dataset contained invalid characters. I replaced those with “?” prior to insertion into the database. Programatically finding and replacing those characters with the correct characters might be an interesting exercise for future students. However, deciphering what the correct characters are would likely entail a manual review of each affected record.

Prior to loading the new dataset, I made all indexes unusable on the affected tables to expedite the load process. This action resulted in errors related to primary key indexes. I did not have high confidence in the RateBeer dataset or my code, so I did not want to disable integrity constraints. Thus, I re-enabled all indexes associated with primary or foreign key constraints on the affected tables. The following indexes remained unusable until after the load completed.

```
ALTER INDEX beer_name_btree UNUSABLE;
ALTER INDEX brewer_name_btree UNUSABLE;
ALTER INDEX style_name_btree UNUSABLE;
```

I also renamed the breweries_pk index to match the new name of the table.

```
ALTER INDEX breweries_pk
RENAME TO brewers_pk;
```

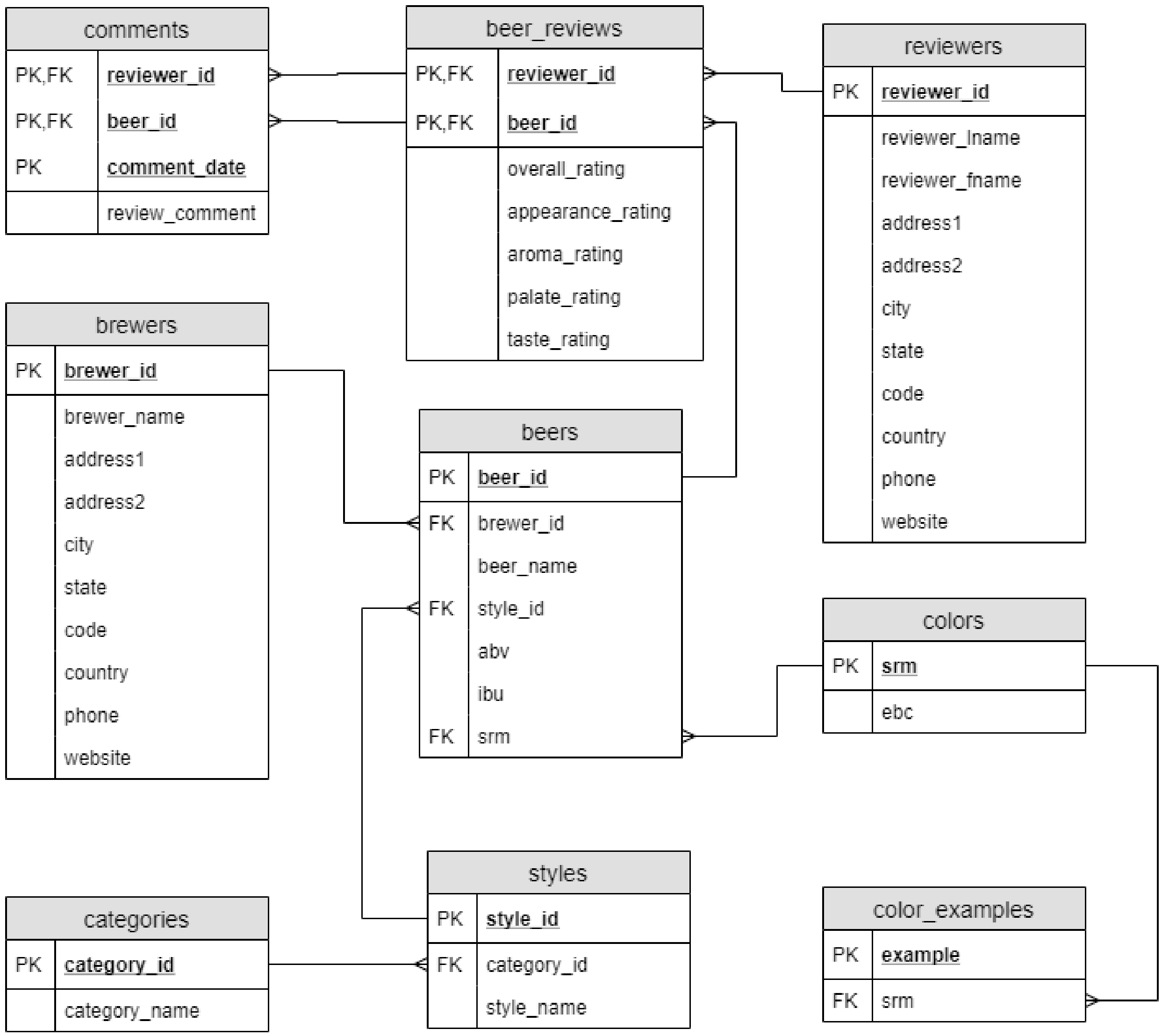
After the load process, I refreshed the statistics for the impacted tables.

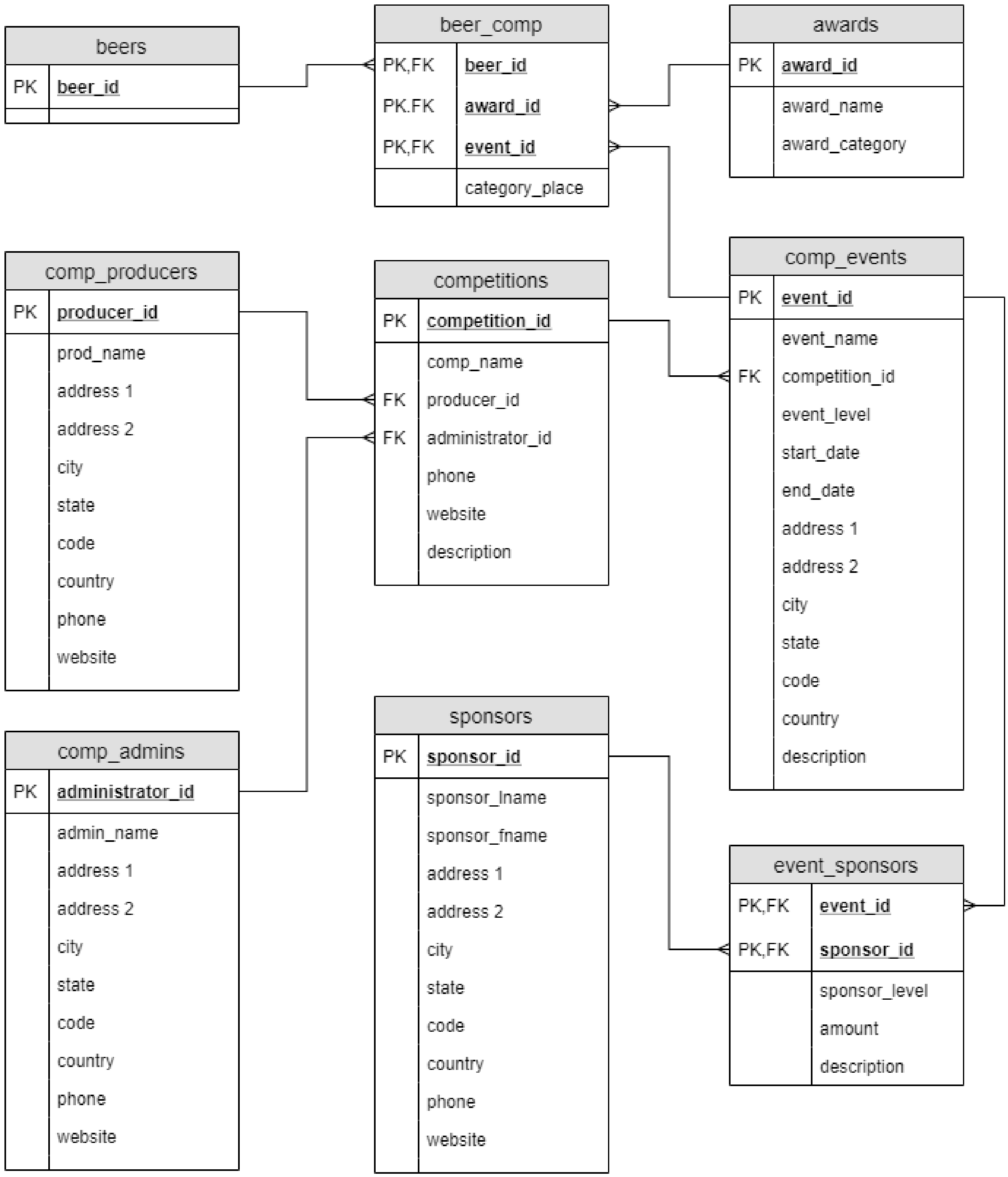
```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'beer_reviews');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comments');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'reviewers');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'brewers');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'styles');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'beers');
```

Then I rebuilt the unusable indexes.

```
ALTER INDEX beer_name_btree REBUILD;
ALTER INDEX brewer_name_btree REBUILD;
ALTER INDEX style_name_btree REBUILD;
```

The updated ERD follows.


comments
PK,FK reviewer_id
PK,FK beer_id
PK comment_date
review_comment
beer_reviews
PK,FK reviewer_id
PK,FK beer_id
overall_rating
appearance_rating
aroma_rating
palate_rating
taste_rating
reviewers
PK reviewer_id
reviewer_lname
reviewer_fname
address1
address2
city
state
code
country
phone
website
brewers
PK brewer_id
brewer_name
address1
address2
city
state
code
country
phone
website
beers
PK beer_id
FK brewer_id
beer_name
FK style_id
abv
ibu
FK srm
colors
PK srm
ebc
styles
PK style_id
FK category_id
style_name
categories
PK category_id
category_name
color_examples
PK example
FK srm


beers
PK beer_id
beer_comp
PK,FK beer_id
PK.FK award_id
PK,FK event_id
category_place
awards
PK award_id
award_name
award_category
comp_producers
PK producer_id
prod_name
address 1
address 2
city
state
code
country
phone
website
competitions
PK competition_id
comp_name
FK producer_id
FK administrator_id
phone
website
description
comp_events
PK event_id
event_name
FK competition_id
event_level
start_date
end_date
address 1
address 2
city
state
code
country
description
comp_admins
PK administrator_id
admin_name
address 1
address 2
city
state
code
country
phone
website
sponsors
PK sponsor_id
sponsor_lname
sponsor_fname
address 1
address 2
city
state
code
country
phone
website
event_sponsors
PK,FK event_id
PK,FK sponsor_id
sponsor_level
amount
description

Query Writing

Write Two Interesting Queries (Assignment 2)

1. For each level of competition (local, state, etc.), list the top five competition producers in terms of sponsorship revenue. Sort results by event level alphabetically, and from highest to lowest revenue within each level.

```
-- insert dummy data to validate query results
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer1');
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer2');
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer3');
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer4');
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer5');
INSERT INTO
	comp_producers (prod_name)
VALUES
	('producer6');
INSERT INTO
	competitions (comp_name, producer_id)
VALUES
	('comp1', 1);
INSERT INTO
	competitions (comp_name, producer_id)
VALUES
	('comp2', 4);
INSERT INTO
	competitions (comp_name, producer_id)
VALUES
	('comp3', 5);
INSERT INTO
	competitions (comp_name, producer_id)
```

```
VALUES
	('comp4', 6);
INSERT INTO
	competitions (comp_name, producer_id)
VALUES
	('comp5', 7);
INSERT INTO
	competitions (comp_name, producer_id)
VALUES
	('comp6', 8);

INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event1', 1, 'local', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event2', 1, 'local', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event3', 4, 'local', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event4', 4, 'local', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event5', 5, 'state', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event6', 5, 'state', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event7', 6, 'state', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event8', 6, 'state', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event9', 2, 'regional', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
```

```sql
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event10', 2, 'regional', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event11', 3, 'regional', CURRENT_DATE, CURRENT_DATE);
INSERT INTO
	comp_events (event_name, competition_id, event_level, start_date, end_date)
VALUES
	('event12', 3, 'regional', CURRENT_DATE, CURRENT_DATE);

INSERT INTO
	sponsors (sponsor_lname)
VALUES
	('corp_sponsor1');

INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(2, 1, 100);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(3, 1, 150);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(4, 1, 200);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(5, 1, 250);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(6, 1, 300);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(7, 1, 350);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
	(8, 1, 400);
INSERT INTO
	event_sponsors (event_id, sponsor_id, amount)
VALUES
```

```
            (9, 1, 450);
INSERT INTO
            event_sponsors (event_id, sponsor_id, amount)
VALUES
            (14, 1, 500);
INSERT INTO
            event_sponsors (event_id, sponsor_id, amount)
VALUES
            (15, 1, 550);
INSERT INTO
            event_sponsors (event_id, sponsor_id, amount)
VALUES
            (16, 1, 600);
INSERT INTO
            event_sponsors (event_id, sponsor_id, amount)
VALUES
            (17, 1, 650);

-- run query
(SELECT
   ce.event_level,
   cp.prod_name as producer,
   SUM(es.amount) as revenue
FROM
   comp_producers cp
   INNER JOIN competitions c
      ON cp.producer_id = c.producer_id
   INNER JOIN comp_events ce
      ON c.competition_id = ce.competition_id
   INNER JOIN event_sponsors es
      ON ce.event_id = es.event_id
WHERE
   ce.event_level LIKE 'local'
GROUP BY
   ce.event_level,
   cp.prod_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
   ce.event_level,
   cp.prod_name as producer,
   SUM(es.amount) as revenue
FROM
   comp_producers cp
   INNER JOIN competitions c
      ON cp.producer_id = c.producer_id
   INNER JOIN comp_events ce
      ON c.competition_id = ce.competition_id
```

```
    INNER JOIN event_sponsors es
        ON ce.event_id = es.event_id
WHERE
    ce.event_level LIKE 'state'
GROUP BY
    ce.event_level,
    cp.prod_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
    ce.event_level,
    cp.prod_name as producer,
    SUM(es.amount) as revenue
FROM
    comp_producers cp
    INNER JOIN competitions c
        ON cp.producer_id = c.producer_id
    INNER JOIN comp_events ce
        ON c.competition_id = ce.competition_id
    INNER JOIN event_sponsors es
        ON ce.event_id = es.event_id
WHERE
    ce.event_level LIKE 'regional'
GROUP BY
    ce.event_level,
    cp.prod_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
    ce.event_level,
    cp.prod_name as producer,
    SUM(es.amount) as revenue
FROM
    comp_producers cp
    INNER JOIN competitions c
        ON cp.producer_id = c.producer_id
    INNER JOIN comp_events ce
        ON c.competition_id = ce.competition_id
    INNER JOIN event_sponsors es
        ON ce.event_id = es.event_id
WHERE
    ce.event_level LIKE 'national'
GROUP BY
    ce.event_level,
    cp.prod_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
```

```
   ce.event_level,
   cp.prod_name as producer,
   SUM(es.amount) as revenue
FROM
   comp_producers cp
   INNER JOIN competitions c
      ON cp.producer_id = c.producer_id
   INNER JOIN comp_events ce
      ON c.competition_id = ce.competition_id
   INNER JOIN event_sponsors es
      ON ce.event_id = es.event_id
WHERE
   ce.event_level LIKE 'international'
GROUP BY
   ce.event_level,
   cp.prod_name
FETCH FIRST 5 ROWS ONLY)
UNION
SELECT
   null,
   null,
   null
FROM
   dual
WHERE
   null IS NOT NULL
ORDER BY
   1 ASC,
   3 DESC;
```

I had difficulty with the sort operation related to syntax (parentheses required here but not there), the fetch limit requirement per select clause, and the sort requirement across all select clauses. I ended up doing some kludgy stuff with dual to make the sort work properly. There might be a more elegant way to accomplish this.

2. For each level of competition (local, state, etc.), list the top five beers in terms of the number of awards won. Sort results by event level alphabetically, and from highest to lowest award count within each level.

```
-- insert dummy data to validate query results
INSERT INTO
   awards (award_name, award_category)
VALUES
   ('gold', 'pilsner');
INSERT INTO
   awards (award_name, award_category)
VALUES
```

```
  ('silver', 'pilsner');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('bronze', 'pilsner');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('gold', 'lager');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('silver', 'lager');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('bronze', 'lager');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('gold', 'ale');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('silver', 'ale');
INSERT INTO
  awards (award_name, award_category)
VALUES
  ('bronze', 'ale');
INSERT INTO
  beer_comp (beer_id, award_id, event_id)
VALUES
  (10, 1, 2);
INSERT INTO
  beer_comp (beer_id, award_id, event_id)
VALUES
  (10, 2, 3);
INSERT INTO
  beer_comp (beer_id, award_id, event_id)
VALUES
  (20, 2, 2);
INSERT INTO
  beer_comp (beer_id, award_id, event_id)
VALUES
  (20, 1, 3);
INSERT INTO
  beer_comp (beer_id, award_id, event_id)
VALUES
```

```
    (20, 1, 4);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (30, 4, 6);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (30, 5, 7);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (40, 5, 6);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (40, 4, 7);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (40, 4, 8);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (50, 7, 14);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (50, 8, 15);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (60, 8, 14);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (60, 7, 15);
INSERT INTO
    beer_comp (beer_id, award_id, event_id)
VALUES
    (60, 7, 16);

-- run query
(SELECT
    ce.event_level,
    b.beer_name,
    COUNT(bc.award_id) as awards
FROM
```

```
    beers b
    INNER JOIN beer_comp bc
        ON b.beer_id = bc.beer_id
    INNER JOIN comp_events ce
        ON bc.event_id = ce.event_id
WHERE
    ce.event_level LIKE 'local'
    AND bc.award_id <> -1
GROUP BY
    ce.event_level,
    b.beer_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
    ce.event_level,
    b.beer_name,
    COUNT(bc.award_id) as awards
FROM
    beers b
    INNER JOIN beer_comp bc
        ON b.beer_id = bc.beer_id
    INNER JOIN comp_events ce
        ON bc.event_id = ce.event_id
WHERE
    ce.event_level LIKE 'state'
    AND bc.award_id <> -1
GROUP BY
    ce.event_level,
    b.beer_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
    ce.event_level,
    b.beer_name,
    COUNT(bc.award_id) as awards
FROM
    beers b
    INNER JOIN beer_comp bc
        ON b.beer_id = bc.beer_id
    INNER JOIN comp_events ce
        ON bc.event_id = ce.event_id
WHERE
    ce.event_level LIKE 'regional'
    AND bc.award_id <> -1
GROUP BY
    ce.event_level,
    b.beer_name
FETCH FIRST 5 ROWS ONLY)
```

```
UNION
(SELECT
   ce.event_level,
   b.beer_name,
   COUNT(bc.award_id) as awards
FROM
   beers b
   INNER JOIN beer_comp bc
      ON b.beer_id = bc.beer_id
   INNER JOIN comp_events ce
      ON bc.event_id = ce.event_id
WHERE
   ce.event_level LIKE 'national'
   AND bc.award_id <> -1
GROUP BY
   ce.event_level,
   b.beer_name
FETCH FIRST 5 ROWS ONLY)
UNION
(SELECT
   ce.event_level,
   b.beer_name,
   COUNT(bc.award_id) as awards
FROM
   beers b
   INNER JOIN beer_comp bc
      ON b.beer_id = bc.beer_id
   INNER JOIN comp_events ce
      ON bc.event_id = ce.event_id
WHERE
   ce.event_level LIKE 'international'
   AND bc.award_id <> -1
GROUP BY
   ce.event_level,
   b.beer_name
FETCH FIRST 5 ROWS ONLY)
UNION
SELECT
   null,
   null,
   null
FROM
   dual
WHERE
   null IS NOT NULL
ORDER BY
   1 ASC,
   3 DESC;
```

Additional Queries

The following queries were run after loading bulk data into the schema (see the Other Topics section).

1. Display the count of beer reviews that do not have an associated comment.

```
-- count 1710949
SELECT
   COUNT(*)
FROM
   beer_reviews rev
   LEFT JOIN comments c
      ON rev.reviewer_id=c.reviewer_id
      AND rev.beer_id=c.beer_id
WHERE
   c.reviewer_id IS NULL
   AND c.beer_id IS NULL;
```

2. Display the count of beer reviews that have more than one associated comment.

```
-- count 0
SELECT
   COUNT(*)
FROM
   (SELECT
      DISTINCT c1.reviewer_id,
      c1.beer_id
   FROM
      comments c1
      INNER JOIN comments c2
         ON c1.reviewer_id=c2.reviewer_id
         AND c1.beer_id=c2.beer_id
   WHERE
      c1.comment_date<>c2.comment_date);
```

3. Display the count of beer reviews that have exactly one associated comment.

```
-- count 1094588
SELECT
   COUNT(*)
FROM
   beer_reviews rev
   INNER JOIN comments c
```

```
        ON rev.reviewer_id=c.reviewer_id
        AND rev.beer_id=c.beer_id
WHERE
    (rev.reviewer_id, rev.beer_id) NOT IN
        (SELECT
            DISTINCT c1.reviewer_id,
            c1.beer_id
        FROM
            comments c1
            INNER JOIN comments c2
                ON c1.reviewer_id=c2.reviewer_id
                AND c1.beer_id=c2.beer_id
        WHERE
            c1.comment_date<>c2.comment_date);
```

Note the total of the counts returned from the previous three queries is 2805537, which is the exact number of rows in the beer_reviews table.

4. Display the brewer ID and (if available) brewer name of every brewer that has an average overall rating of 80 or higher and an average ABV of 4% or higher. For the purpose of calculating average ABV, exclude beers with an ABV of 0. Sort the results by average rating highest to lowest and then by average ABV highest to lowest.

```
-- 108 rows returned
SELECT
    br.brewer_id,
    br.brewer_name,
    ROUND(AVG(rev.overall_rating),1) AS Avg_Rating,
    ROUND(AVG(b.abv),2) AS Avg_ABV
FROM
    brewers br
    INNER JOIN beers b
        ON br.brewer_id=b.brewer_id
    INNER JOIN beer_reviews rev
        ON b.beer_id=rev.beer_id
WHERE
    b.abv > 0
GROUP BY
    br.brewer_id,
    br.brewer_name
    HAVING ROUND(AVG(rev.overall_rating),1) >= 80
    AND ROUND(AVG(b.abv),2) >= 4
ORDER BY
    Avg_Rating DESC,
    Avg_ABV DESC;
```

Why are none of the brewer names displayed? Because the brewer name is not available for beers imported from the RateBeer dataset, and ratings are available only for beers imported from the RateBeer dataset.

5. Determine if any beers imported from the RateBeer dataset had a style name that was already in the styles table. If so, display the category and style names along with the count of beers for each style name. As a point of reference, include the count of all other imported beers in a single row. Sort by beer count highest to lowest. Hints: recall that all pre-existing beers were reassigned beer ID values over 500000, and all style names in the RateBeer dataset were mapped to category_id = -1 if the style name did not already exist in the styles table.

```
-- 6 rows returned
SELECT
   DISTINCT ca.category_name,
   s.style_name,
   COUNT(*) AS Beer_Count
FROM
   categories ca
   INNER JOIN styles s
      ON ca.category_id=s.category_id
   INNER JOIN beers b
      ON s.style_id=b.style_id
WHERE
   b.beer_id < 500000
   AND ca.category_id <> -1
GROUP BY
   ca.category_name,
   s.style_name
UNION ALL
SELECT
   'Unknown' AS category_name,
   'Aggregated' AS style_name,
   COUNT(*) AS Beer_Count
FROM
   categories ca
   INNER JOIN styles s
      ON ca.category_id=s.category_id
   INNER JOIN beers b
      ON s.style_id=b.style_id
WHERE
   b.beer_id < 500000
   AND ca.category_id = -1
ORDER BY
   Beer_Count DESC;
```

The total number of beers counted in the previous query was 108194. The total number of pre-existing beers in the beers table after cleaning the original data was 5890. Those numbers sum to 114084, which is the total number of beers in the beers table.

6. Display all beers with an average overall rating that is equal to the highest overall rating given by any reviewer to any beer within the same style. Sort results by style name alphabetically and then by beer name alphabetically.

While working on this query, I discovered a data anomaly when I noticed some numbers did not add up properly. The updated styles table had 226 total rows. The original styles table had 141. Thus, 85 new styles were added during the RateBeer import. Imported beers were also associated to five of the pre-existing styles, for a total of 90 styles used by imported beers. However, the following query returned only 89 rows.

```
SELECT
   DISTINCT style_id
FROM
   beers
WHERE
   beer_id < 500000;
```

During the import, several unrecognized characters were encountered in the RateBeer dataset. My Python script replaced them with “?”. However, one style name (Kölsch) occurred twice with a different number of unrecognized characters. Consequently, two versions of this style name were imported, K?lsch and K???lsch. The strange thing is that no beers were associated to K???lsch. I have no explanation for this.

```
SELECT
   DISTINCT style_id,
   style_name
FROM
   styles
WHERE
   category_id = -1
   AND style_id <> -1
MINUS
SELECT
   DISTINCT s.style_id,
   style_name
FROM
   styles s
   INNER JOIN beers b
      ON s.style_id=b.style_id
WHERE
   b.beer_id < 500000
   AND s.category_id = -1;
```

```
SELECT
   *
FROM
   styles
WHERE
   style_name LIKE 'K%';

SELECT
   beer_id,
   beer_name,
   b.style_id,
   s.style_name
FROM
   beers b
   INNER JOIN styles s
      ON b.style_id=s.style_id
WHERE
   b.style_id = 226
   OR b.style_id = 144;
```

To resolve this issue, I deleted the K???lsch style name from the styles table.

```
DELETE FROM
   styles
WHERE
   style_id = 226;

EXECUTE dbms_stats.gather_table_stats('DB870', 'styles');
```

Now back to the query at hand.

```
-- 53 rows returned
WITH
   style_ratings
AS
   (SELECT
      b.style_id,
      MAX(rev.overall_rating) AS max_rating
   FROM
      beers b
      INNER JOIN beer_reviews rev
         ON b.beer_id=rev.beer_id
   GROUP BY
      b.style_id
   ORDER BY
```

```
      style_id ASC)
SELECT
   s.style_name,
   b.beer_name,
   sr.max_rating AS rating
FROM
   styles s
   INNER JOIN beers b
      ON s.style_id=b.style_id
   INNER JOIN style_ratings sr
      ON b.style_id=sr.style_id
   INNER JOIN (SELECT
                     b.beer_id,
                     ROUND(AVG(rev.overall_rating),0) AS avg_rating
                  FROM
                     beers b
                  INNER JOIN beer_reviews rev
                     ON b.beer_id=rev.beer_id
                 GROUP BY
                     b.beer_id
                 ORDER BY
                     beer_id ASC) rat
      ON b.beer_id=rat.beer_id
WHERE
   rat.avg_rating=sr.max_rating
ORDER BY
   style_name ASC,
   beer_name ASC;
```

A Stored Procedure

Goal: Identify the top five beers within each style based on average overall rating. Include ties even if doing so returns more than five beers per style. Display the style id, style name, average overall rating of the beer, beer id, beer name, and brewer id. (Brewer name would be nice but is not available.) Sort the results by style name alphabetically and then by rating highest to lowest.

An inelegant solution would use the following query as the base building block for a very long UNION series (currently 89 styles, so 89 base blocks). Sorting could be done with a "dummy" block at the end as was done with the "two interesting queries" from Assignment 2. However, each style_id would need to be coded manually into the WHERE clause of each block, and prior knowledge of every style_id in the beer_reviews table would be needed. Obviously, such code would be onerous to maintain and brittle in the presence of data updates.

```
WITH
   beer_ratings
AS
```

```
  (SELECT
    b.style_id,
    b.beer_id,
    ROUND(AVG(rev.overall_rating),0) AS avg_rating
  FROM
    beers b
    INNER JOIN beer_reviews rev
      ON b.beer_id=rev.beer_id
  GROUP BY
    b.style_id,
    b.beer_id
  ORDER BY
    style_id ASC,
    avg_rating DESC)
SELECT
  br.style_id,
  s.style_name,
  br.avg_rating,
  br.beer_id,
  b.beer_name,
  b.brewer_id
FROM
  beer_ratings br
  INNER JOIN beers b
    ON br.beer_id=b.beer_id
  INNER JOIN styles s
    ON b.style_id=s.style_id
WHERE
  br.style_id=13
ORDER BY
  avg_rating DESC
FETCH FIRST 5 ROWS WITH TIES;
```

A more elegant solution is the following stored procedure. It would be easier to maintain and more robust in the presence of data updates. However, I could not figure out how to sort the results as required and display column headers. Perhaps future students can tackle these challenges.

```
create or replace PROCEDURE
  top_beers
IS
  beer_lst DBMS_OUTPUT.CHARARR;
  CURSOR
    style_cur
  IS
    SELECT
      DISTINCT b.style_id
    FROM
```

```
          beers b
          INNER JOIN beer_reviews rev
            ON b.beer_id=rev.beer_id
        ORDER BY
          style_id ASC;
BEGIN
  DBMS_OUTPUT.ENABLE (100000);
  FOR rec IN style_cur
    LOOP
      DECLARE
        CURSOR
          loop_cur
        IS
          SELECT
            br.style_id,
            s.style_name,
            br.avg_rating,
            br.beer_id,
            b.beer_name,
            b.brewer_id
          FROM
            (SELECT
              b.style_id,
              b.beer_id,
              ROUND(AVG(rev.overall_rating),0) AS avg_rating
            FROM
              beers b
            INNER JOIN beer_reviews rev
              ON b.beer_id=rev.beer_id
            GROUP BY
              b.style_id,
              b.beer_id
            ORDER BY
              style_id ASC,
              avg_rating DESC) br
            INNER JOIN beers b
              ON br.beer_id=b.beer_id
            INNER JOIN styles s
              ON b.style_id=s.style_id
          WHERE
            br.style_id=rec.style_id
          ORDER BY
            avg_rating DESC
          FETCH FIRST 5 ROWS WITH TIES;
      BEGIN
      FOR style_rec IN loop_cur
        LOOP
```

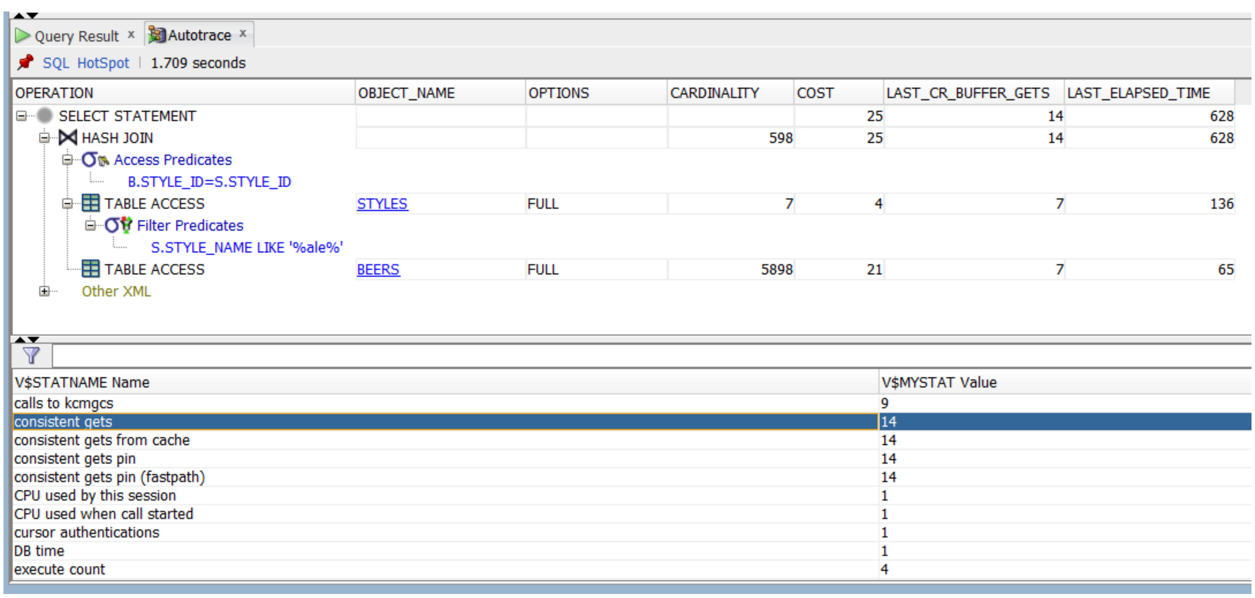
```
                DBMS_OUTPUT.PUT_LINE(style_rec.style_id || ' : ' || style_rec.style_name || ' : ' ||
style_rec.avg_rating || ' : ' || style_rec.beer_id || ' : ' || style_rec.beer_name || ' : ' ||
style_rec.brewer_id);
            END LOOP;
        END;
    END LOOP;
END;

-- 665 rows returned
EXECUTE top_beers;
```

Performance Tuning

Basic Indexing & Query Performance (Assignment 3)

Using my implemented design from Assignment 2, I ran some basic queries with only the default indexes. The first query did a full scan on each joined table and returned 1025 rows.

```
SELECT
    b.beer_name,
    s.style_name
FROM
    beers b
    INNER JOIN styles s
        ON b.style_id=s.style_id
WHERE
    style_name LIKE '%ale%';
```



Query Result × | Autotrace ×

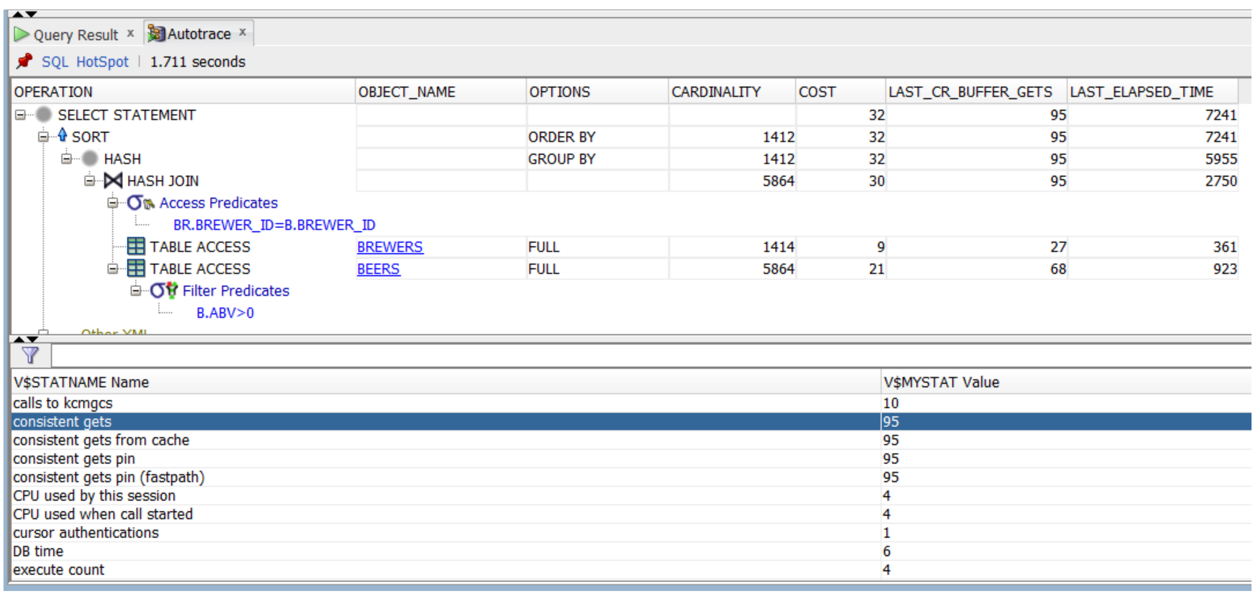
SQL HotSpot | 1.709 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				25	14	628
HASH JOIN			598	25	14	628
Access Predicates						
B.STYLE_ID=S.STYLE_ID						
TABLE ACCESS	STYLES	FULL	7	4	7	136
Filter Predicates						
S.STYLE_NAME LIKE '%ale%'						
TABLE ACCESS	BEERS	FULL	5898	21	7	65
Other XML						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	9
consistent gets	14
consistent gets from cache	14
consistent gets pin	14
consistent gets pin (fastpath)	14
CPU used by this session	1
CPU used when call started	1
cursor authentications	1
DB time	1
execute count	4

The second query also did a full scan on each joined table. It returned 754 rows.

```
SELECT
   br.brewer_name,
   MIN(b.abv) AS Min_ABV,
   MAX(b.abv) AS Max_ABV,
   ROUND(AVG(b.abv),2) AS Avg_ABV
FROM
   brewers br
   INNER JOIN beers b
      ON br.brewer_id=b.brewer_id
WHERE
   b.abv > 0
GROUP BY
   br.brewer_name
ORDER BY
   Avg_ABV DESC;
```



Query Result | Autotrace

SQL HotSpot | 1.711 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				32	95	7241
SORT		ORDER BY	1412	32	95	7241
HASH		GROUP BY	1412	32	95	5955
HASH JOIN			5864	30	95	2750
Access Predicates						
BR.BREWER_ID=B.BREWER_ID						
TABLE ACCESS	BREWERS	FULL	1414	9	27	361
TABLE ACCESS	BEERS	FULL	5864	21	68	923
Filter Predicates						
B.ABV>0						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	10
consistent gets	95
consistent gets from cache	95
consistent gets pin	95
consistent gets pin (fastpath)	95
CPU used by this session	4
CPU used when call started	4
cursor authentications	1
DB time	6
execute count	4

Since searches often include name attributes, it might be advantageous to create some indexes on frequently used name attributes. Beer drinkers tend to be interested in specific styles, so I started with the style_name attribute. Logically, all (category_name, style_name) tuples should be unique. However, I discovered a duplicate style_name record in the North American Lager category.

```
SELECT
    *
FROM
    styles s1
    INNER JOIN styles s2
        ON s1.style_name=s2.style_name
WHERE
    s1.style_id <> s2.style_id;
```

Query Result × Autotrace ×

SQL | All Rows Fetched: 4 in 0.03 seconds

	STYLE_ID	CATEGORY_ID	STYLE_NAME	STYLE_ID_1	CATEGORY_ID_1	STYLE_NAME_1
1	46	3	Porter	25	2	Porter
2	25	2	Porter	46	3	Porter
3	103	8	American-Style Amber Lager	98	8	American-Style Amber Lager
4	98	8	American-Style Amber Lager	103	8	American-Style Amber Lager

No beers were associated with this style, so I was able to delete the duplicate record without impacting other tables.

```
DELETE FROM
   styles
WHERE
   style_id = 103;
```

Then I created a unique constraint to prevent this incident from recurring. Oracle automatically created an index (also named cat_style_unq) on the constraint.

```
ALTER TABLE
   styles
ADD CONSTRAINT
   cat_style_unq
UNIQUE (category_id, style_name);
```

Next, I created a B+-tree index on style_name with default settings.

```
CREATE INDEX
   style_name_btree
ON
   styles (style_name);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'styles');
```

It makes sense that beer and brewer names would be used in many searches, so I also created B+-tree indexes on each of them using default settings.

```
CREATE INDEX
```

```
   beer_name_btree
ON
   beers (beer_name);

CREATE INDEX
   brewer_name_btree
ON
   brewers (brewer_name);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'beers');

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'brewers');
```

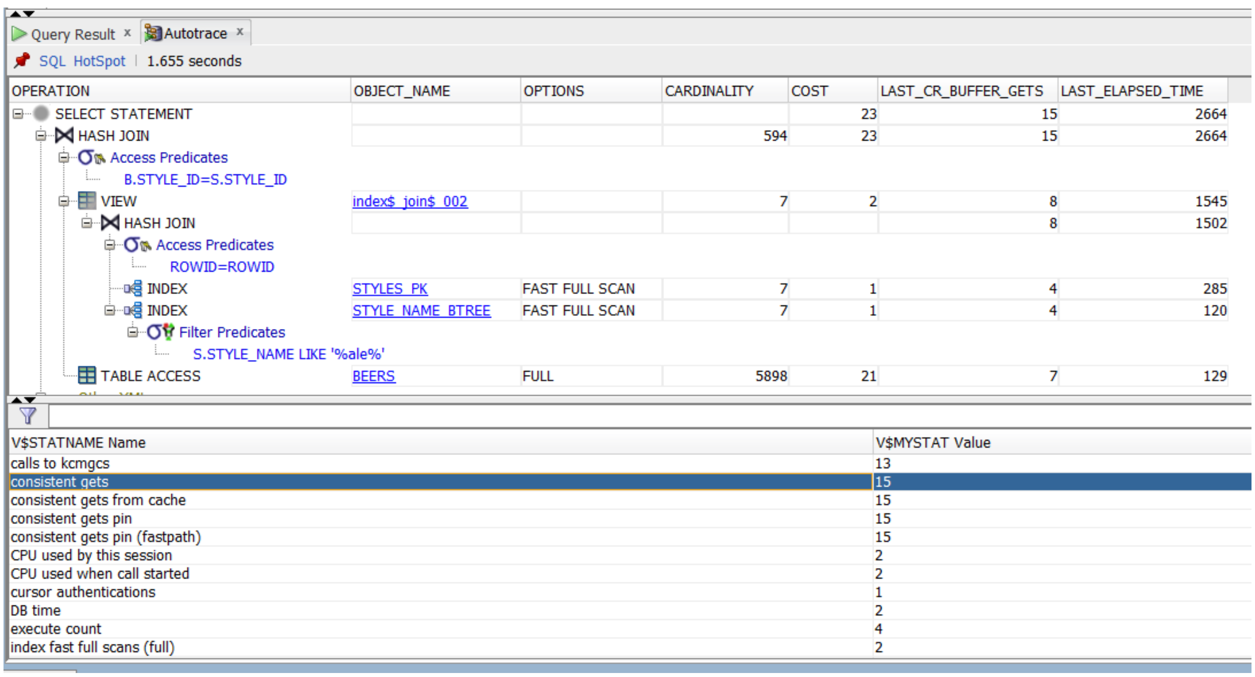
Before doing any more experimentation, I created the usual indexes on foreign keys. However, I did not create indexes for beers(srm), color_examples(srm), or styles(category_id) because the number of foreign key values is very low and likely to stay very low. Thus, a full scan of the referenced tables would not be costly, which means the optimizer might not use the indexes.

```
CREATE INDEX
   brewer_fk_btree
ON
   beers (brewer_id);

CREATE INDEX
   style_fk_btree
ON
   beers (style_id);

CREATE INDEX
   comp_fk_btree
ON
   comp_events (competition_id);

CREATE INDEX
   prod_fk_btree
ON
   competitions (producer_id);

CREATE INDEX
   admin_fk_btree
ON
   competitions (administrator_id);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'beers');

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comp_events');
```

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'competitions');

Next, I re-ran the simple queries from above. The first query used the new style_name index because I filtered on style_name. The new styles foreign key index was not used, but the styles primary key index was used. The consistent_gets increased by one, perhaps related to the extra object access (two indexes and one table versus two tables).

```
SELECT
   b.beer_name,
   s.style_name
FROM
   beers b
   INNER JOIN styles s
      ON b.style_id=s.style_id
WHERE
   style_name LIKE '%ale%';
```



Query Result × | Autotrace ×
SQL HotSpot | 1.655 seconds

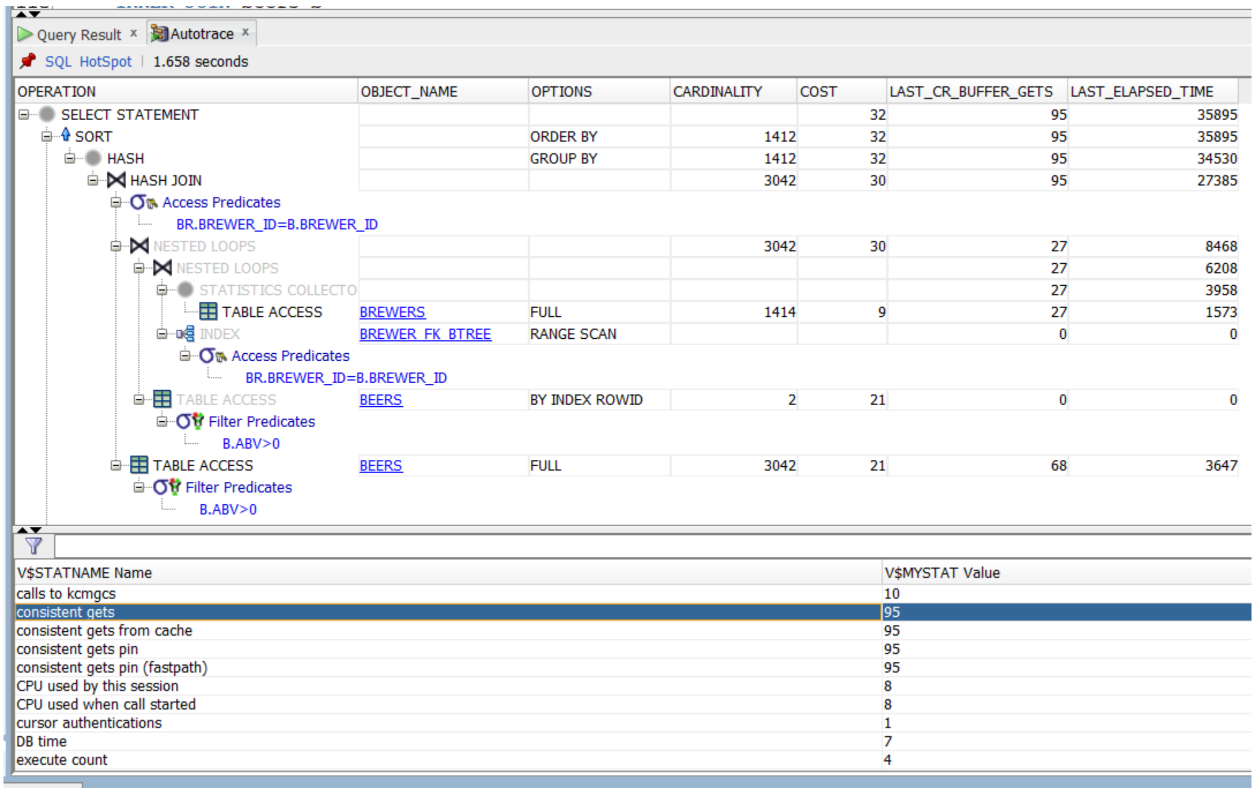
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				23	15	2664
HASH JOIN			594	23	15	2664
Access Predicates						
B.STYLE_ID=S.STYLE_ID						
VIEW	index$_join$_002		7	2	8	1545
HASH JOIN					8	1502
Access Predicates						
ROWID=ROWID						
INDEX	STYLES_PK	FAST FULL SCAN	7	1	4	285
INDEX	STYLE_NAME_BTREE	FAST FULL SCAN	7	1	4	120
Filter Predicates						
S.STYLE_NAME LIKE '%ale%'						
TABLE ACCESS	BEERS	FULL	5898	21	7	129

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	13
consistent gets	15
consistent gets from cache	15
consistent gets pin	15
consistent gets pin (fastpath)	15
CPU used by this session	2
CPU used when call started	2
cursor authentications	1
DB time	2
execute count	4
index fast full scans (full)	2

The second query did not use the new brewer_name index because the query required all brewers to be included in the results. The new brewer foreign key index was used. The consistent_gets remained the same.

```
SELECT
   br.brewer_name,
   MIN(b.abv) AS Min_ABV,
   MAX(b.abv) AS Max_ABV,
```

```
    ROUND(AVG(b.abv),2) AS Avg_ABV
FROM
    brewers br
    INNER JOIN beers b
        ON br.brewer_id=b.brewer_id
WHERE
    b.abv > 0
GROUP BY
    br.brewer_name
ORDER BY
    Avg_ABV DESC;
```


Query Result × | Autotrace ×

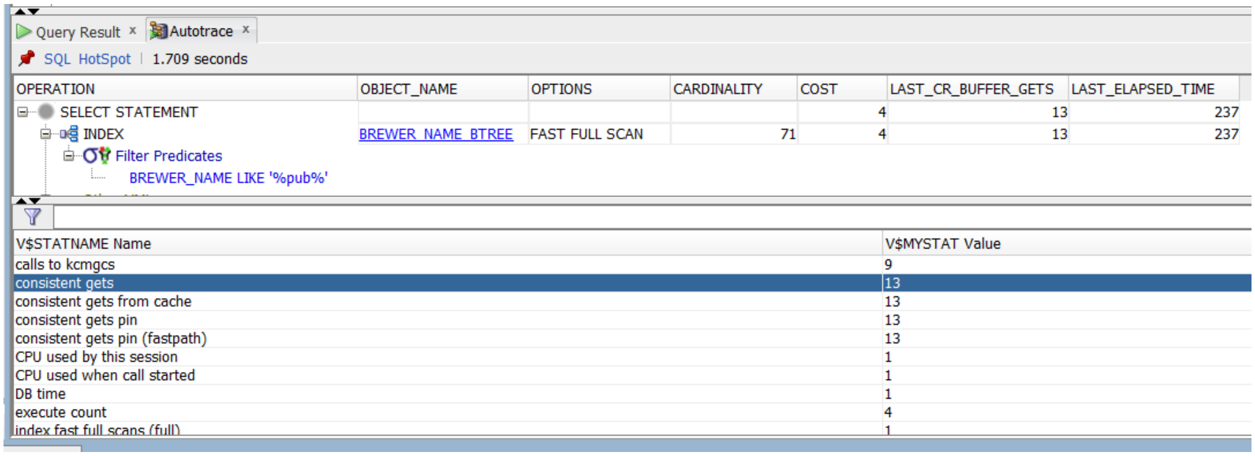
SQL HotSpot | 1.658 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				32	95	35895
SORT		ORDER BY	1412	32	95	35895
HASH		GROUP BY	1412	32	95	34530
HASH JOIN			3042	30	95	27385
Access Predicates						
BR.BREWER_ID=B.BREWER_ID						
NESTED LOOPS			3042	30	27	8468
NESTED LOOPS					27	6208
STATISTICS COLLECTO					27	3958
TABLE ACCESS	BREWERS	FULL	1414	9	27	1573
INDEX	BREWER_FK_BTREE	RANGE SCAN			0	0
Access Predicates						
BR.BREWER_ID=B.BREWER_ID						
TABLE ACCESS	BEERS	BY INDEX ROWID	2	21	0	0
Filter Predicates						
B.ABV>0						
TABLE ACCESS	BEERS	FULL	3042	21	68	3647
Filter Predicates						
B.ABV>0						

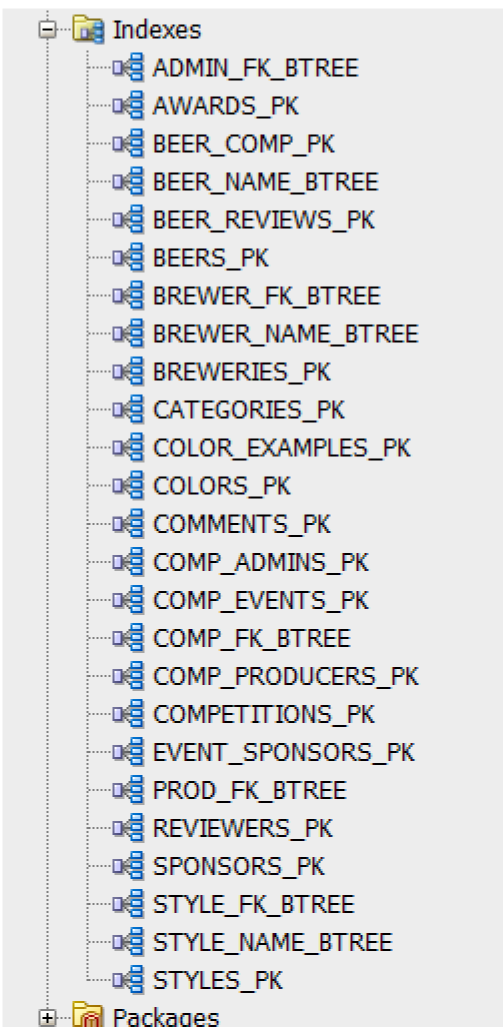
V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	10
consistent gets	95
consistent gets from cache	95
consistent gets pin	95
consistent gets pin (fastpath)	95
CPU used by this session	8
CPU used when call started	8
cursor authentications	1
DB time	7
execute count	4

Filtering on brewer_name to find pubs with inhouse microbreweries demonstrates the use of the brewer_name index. Only 25 rows are returned.

```
SELECT
    br.brewer_name
FROM
    brewers br
WHERE
    brewer_name LIKE '%pub%';
```



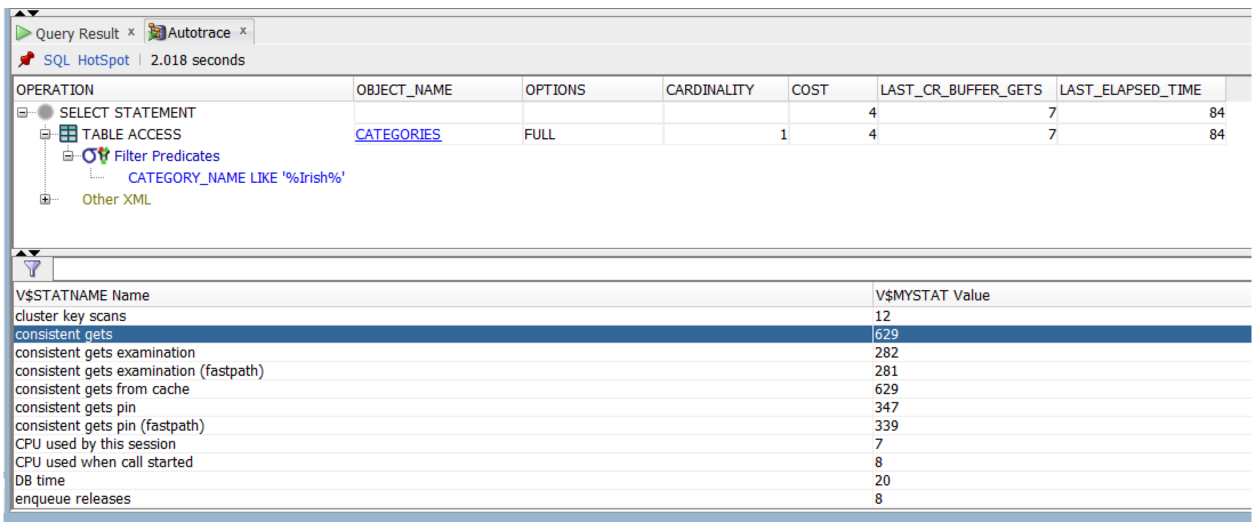
Here's what I have so far. All the new indexes end with BTREE to make it easy to spot them.



Intermediate Indexing & Query Performance (Assignment 3)

Bitmap indexes could be useful for attributes such as comp_events(event_level) with five levels and event_sponsors(sponsor_level) with a currently unknown but expected to be small number of values (such as titanium, platinum, gold, silver, and bronze). In time, those tables will each hold many instances. Another possibility is categories(category_name) with 12 values, but the number of instances likely will not grow beyond the current 12. Thus, we have high cardinality (all unique values) combined with a low number of values. That should make an interesting experiment. Since the categories tables is pre-populated and (indirectly) referenced from the pre-populated beers table, I have data with which to work. I started with the categories table and worked backward to the beers table.

```
SELECT
   *
FROM
   categories
WHERE
   category_name LIKE '%Irish%';
```



Query Result × | Autotrace ×

SQL HotSpot | 2.018 seconds

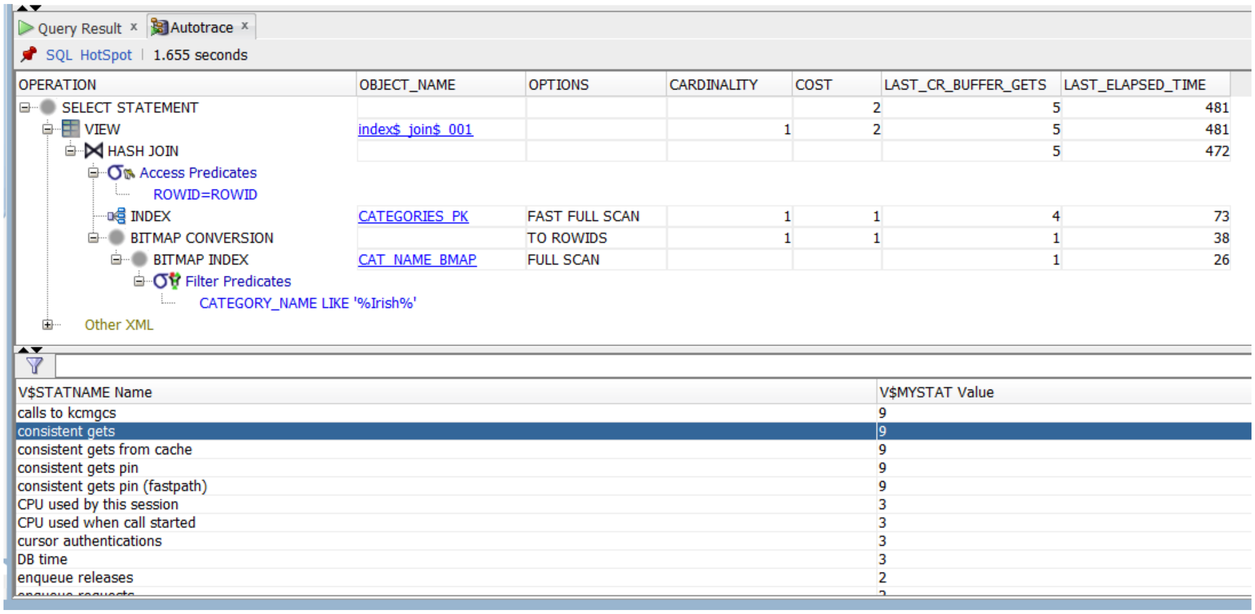
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				4	7	84
TABLE ACCESS	CATEGORIES	FULL	1	4	7	84
Filter Predicates						
CATEGORY_NAME LIKE '%Irish%'						
Other XML						

V$STATNAME Name	V$MYSTAT Value
cluster key scans	12
consistent gets	629
consistent gets examination	282
consistent gets examination (fastpath)	281
consistent gets from cache	629
consistent gets pin	347
consistent gets pin (fastpath)	339
CPU used by this session	7
CPU used when call started	8
DB time	20
enqueue releases	8

The query did a full table scan, and the consistent_gets came in at 629. I do not understand how such a small table could require that many I/Os. The Last_CR_Buffer_Gets column indicates much lower I/O. Anyway, I created the index as follows.

```
CREATE BITMAP INDEX
   cat_name_bmap
ON
   categories(category_name);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'categories');

SELECT
```

```
    *
FROM
    categories
WHERE
    category_name LIKE '%Irish%';
```



Query Result | Autotrace

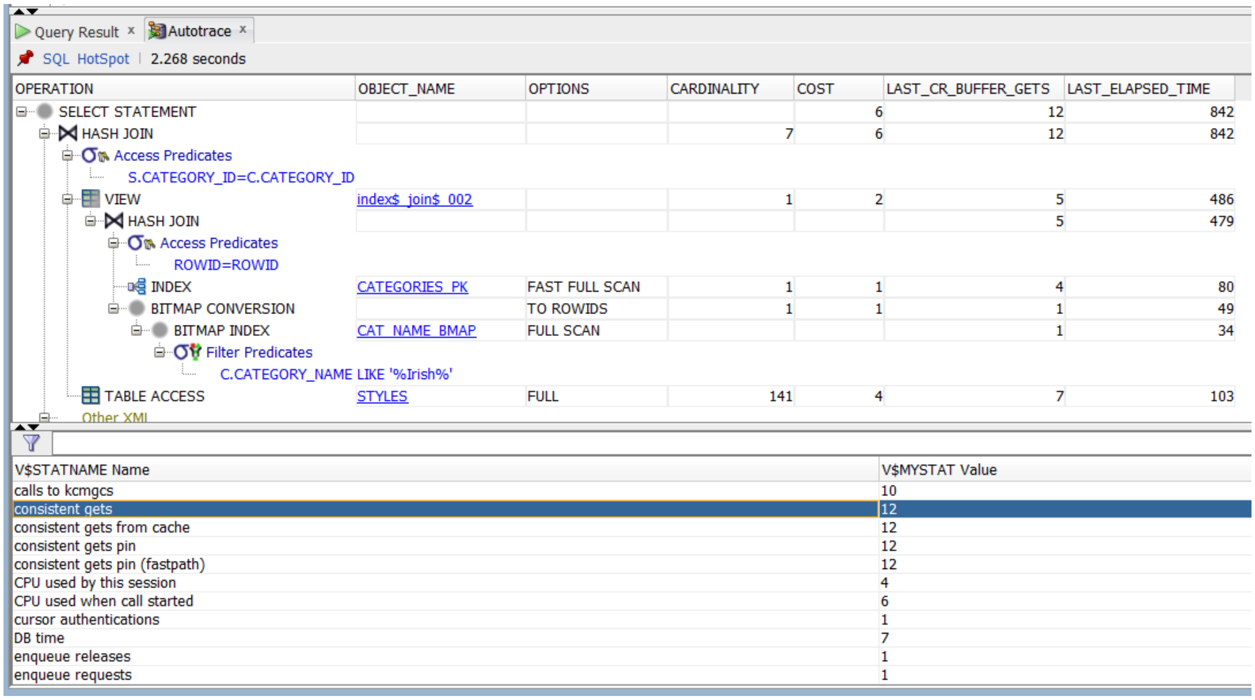
SQL HotSpot | 1.655 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				2	5	481
VIEW	index$_join$_001		1	2	5	481
HASH JOIN					5	472
Access Predicates						
ROWID=ROWID						
INDEX	CATEGORIES_PK	FAST FULL SCAN	1	1	4	73
BITMAP CONVERSION		TO ROWIDS	1	1	1	38
BITMAP INDEX	CAT_NAME_BMAP	FULL SCAN			1	26
Filter Predicates						
CATEGORY_NAME LIKE '%Irish%'						
Other XML						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	9
consistent gets	9
consistent gets from cache	9
consistent gets pin	9
consistent gets pin (fastpath)	9
CPU used by this session	3
CPU used when call started	3
cursor authentications	3
DB time	3
enqueue releases	2

The query used the new index, and the consistent_gets dropped to just nine. The Last_CR_Buffer_Gets dropped from seven to five. Let’s see what happens when I query the styles table and then the beers table.

```
SELECT
    style_name
FROM
    styles s
    INNER JOIN categories c
        ON s.category_id = c.category_id
WHERE
    category_name LIKE '%Irish%';
```



Query Result × Autotrace ×

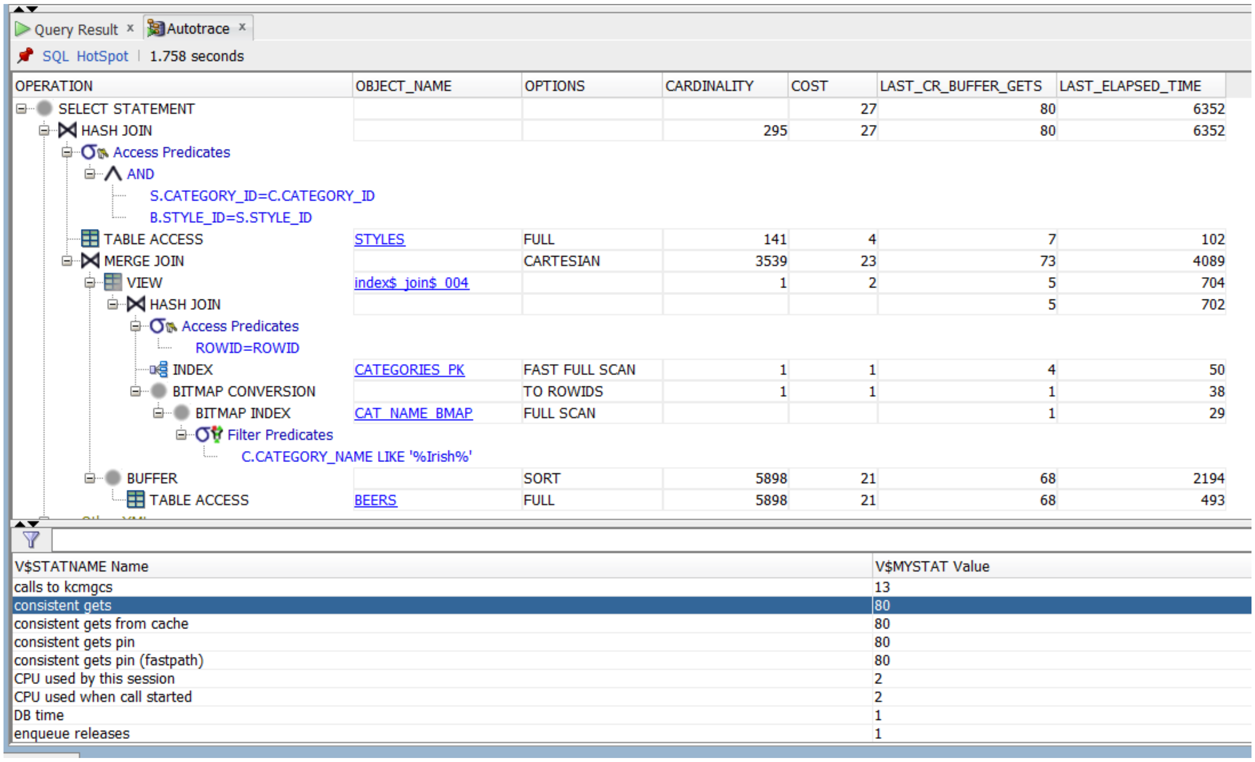
SQL HotSpot | 2.268 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				6	12	842
HASH JOIN			7	6	12	842
Access Predicates						
S.CATEGORY_ID=C.CATEGORY_ID						
VIEW	index$_join$_002		1	2	5	486
HASH JOIN					5	479
Access Predicates						
ROWID=ROWID						
INDEX	CATEGORIES_PK	FAST FULL SCAN	1	1	4	80
BITMAP CONVERSION		TO ROWIDS	1	1	1	49
BITMAP INDEX	CAT_NAME_BMAP	FULL SCAN			1	34
Filter Predicates						
C.CATEGORY_NAME LIKE '%Irish%'						
TABLE ACCESS	STYLES	FULL	141	4	7	103
Other XML						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	10
consistent gets	12
consistent gets from cache	12
consistent gets pin	12
consistent gets pin (fastpath)	12
CPU used by this session	4
CPU used when call started	6
cursor authentications	1
DB time	7
enqueue releases	1
enqueue requests	1

The index is used again with a low consistent_gets count of 12. The index-related Last_CR_Buffer_Gets remained at five.

```
SELECT
   beer_name
FROM
   beers b
   INNER JOIN styles s
      ON b.style_id=s.style_id
   INNER JOIN categories c
      ON s.category_id = c.category_id
WHERE
   category_name LIKE '%Irish%';
```



Query Result × Autotrace ×

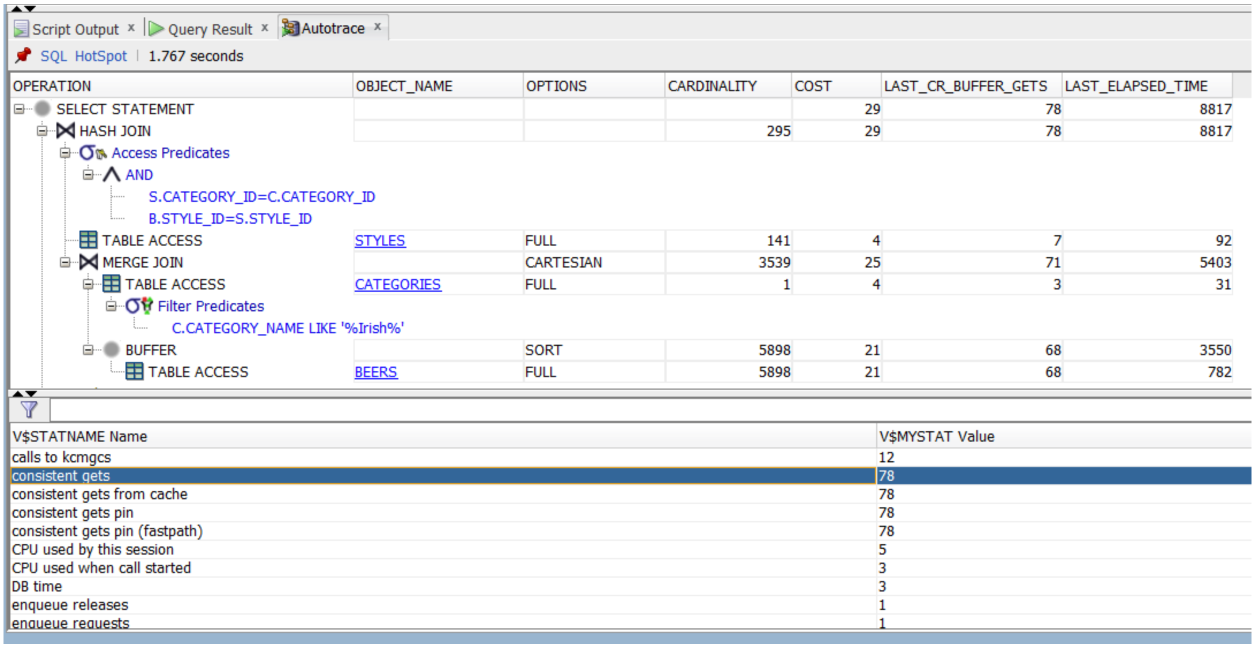
SQL HotSpot | 1.758 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				27	80	6352
HASH JOIN			295	27	80	6352
Access Predicates						
AND						
S.CATEGORY_ID=C.CATEGORY_ID						
B.STYLE_ID=S.STYLE_ID						
TABLE ACCESS	STYLES	FULL	141	4	7	102
MERGE JOIN		CARTESIAN	3539	23	73	4089
VIEW	index$_join$_004		1	2	5	704
HASH JOIN					5	702
Access Predicates						
ROWID=ROWID						
INDEX	CATEGORIES_PK	FAST FULL SCAN	1	1	4	50
BITMAP CONVERSION		TO ROWIDS	1	1	1	38
BITMAP INDEX	CAT_NAME_BMAP	FULL SCAN			1	29
Filter Predicates						
C.CATEGORY_NAME LIKE '%Irish%'						
BUFFER		SORT	5898	21	68	2194
TABLE ACCESS	BEERS	FULL	5898	21	68	493

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	13
consistent gets	80
consistent gets from cache	80
consistent gets pin	80
consistent gets pin (fastpath)	80
CPU used by this session	2
CPU used when call started	2
DB time	1
enqueue releases	1

The index is used again, but the consistent_gets count rose to 80. However, the index-related Last_CR_Buffer_Gets remained at five. Now let's try it without the bitmap index.

```
DROP INDEX
        cat_name_bmap;

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'categories');

SELECT
  beer_name
FROM
  beers b
  INNER JOIN styles s
    ON b.style_id=s.style_id
  INNER JOIN categories c
    ON s.category_id = c.category_id
WHERE
  category_name LIKE '%Irish%';
```


Script Output × | Query Result × | Autotrace ×

SQL HotSpot | 1.767 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				29	78	8817
HASH JOIN			295	29	78	8817
Access Predicates						
AND						
S.CATEGORY_ID=C.CATEGORY_ID						
B.STYLE_ID=S.STYLE_ID						
TABLE ACCESS	STYLES	FULL	141	4	7	92
MERGE JOIN		CARTESIAN	3539	25	71	5403
TABLE ACCESS	CATEGORIES	FULL	1	4	3	31
Filter Predicates						
C.CATEGORY_NAME LIKE '%Irish%'						
BUFFER		SORT	5898	21	68	3550
TABLE ACCESS	BEERS	FULL	5898	21	68	782

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	12
consistent gets	78
consistent gets from cache	78
consistent gets pin	78
consistent gets pin (fastpath)	78
CPU used by this session	5
CPU used when call started	3
DB time	3
enqueue releases	1
enqueue requests	1

The consistent_gets dropped by two. The five index-related Last_CR_Buffer_Gets were replaced by three for a full table scan of categories. In a more realistic scenario (with a large table), the bitmap index would undoubtedly be more efficient than a full table scan. Of course, that scenario would imply a low cardinality attribute. So, we have confirmed a low number of values is, by itself, insufficient to guarantee a bitmap index efficiency advantage. However, I cannot state low cardinality is required for bitmap indexes to be advantageous because (1) my experiment did not prove that and (2) the OTN article (https://www.oracle.com/technical-resources/articles/sharma-indexes.html) provided in module five provides evidence to the contrary.

Moving on, I wanted to experiment with a composite index versus a pair of single indexes. Perhaps a beer lover wants to attend a beer competition in his/her area. A search based on city and state would be useful. For this experiment, I needed more data, so I inserted records as follows.

```
UPDATE
   comp_events
SET
   city = 'Miami',
   state = 'FL'
WHERE
   event_id IN (2,3);
UPDATE
   comp_events
SET
   city = 'Key West',
   state = 'FL'
WHERE
   event_id IN (4,5);
UPDATE
```

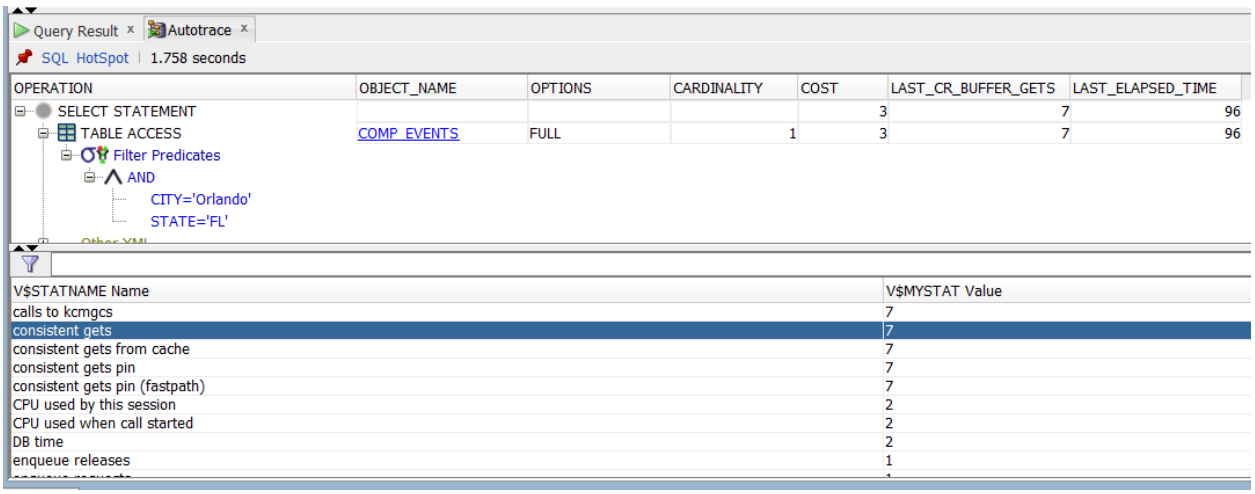
```
   comp_events
SET
   city = 'Orlando',
   state = 'FL'
WHERE
   event_id IN (6,7);
UPDATE
   comp_events
SET
   city = 'Fort Myers',
   state = 'FL'
WHERE
   event_id IN (8,9);
UPDATE
   comp_events
SET
   city = 'Sarasota',
   state = 'FL'
WHERE
   event_id IN (14,15);
UPDATE
   comp_events
SET
   city = 'Tampa',
   state = 'FL'
WHERE
   event_id IN (16,17);

INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event13', 1, 'regional', CURRENT_DATE, CURRENT_DATE, 'Gainesville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event14', 1, 'regional', CURRENT_DATE, CURRENT_DATE, 'Gainesville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event15', 1, 'state', CURRENT_DATE, CURRENT_DATE, 'Jacksonville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event16', 1, 'state', CURRENT_DATE, CURRENT_DATE, 'Jacksonville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event17', 2, 'state', CURRENT_DATE, CURRENT_DATE, 'Boca Raton', 'FL');
```

```
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event18', 2, 'state', CURRENT_DATE, CURRENT_DATE, 'Boca Raton', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event19', 2, 'national', CURRENT_DATE, CURRENT_DATE, 'Orlando', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event20', 2, 'national', CURRENT_DATE, CURRENT_DATE, 'Orlando', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event21', 3, 'national', CURRENT_DATE, CURRENT_DATE, 'Tallahassee', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event22', 3, 'national', CURRENT_DATE, CURRENT_DATE, 'Tallahassee', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event23', 3, 'national', CURRENT_DATE, CURRENT_DATE, 'Lakeland', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event24', 3, 'national', CURRENT_DATE, CURRENT_DATE, 'Lakeland', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event25', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Lake City', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event26', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Lake City', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event27', 4, 'national', CURRENT_DATE, CURRENT_DATE, 'Hollywood', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event28', 4, 'national', CURRENT_DATE, CURRENT_DATE, 'Hollywood', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event29', 4, 'local', CURRENT_DATE, CURRENT_DATE, 'Melbourne', 'FL');
```

```
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event30', 4, 'local', CURRENT_DATE, CURRENT_DATE, 'Melbourne', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event31', 5, 'national', CURRENT_DATE, CURRENT_DATE, 'Tampa', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event32', 5, 'national', CURRENT_DATE, CURRENT_DATE, 'Tampa', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event33', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Orlando', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event34', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Orlando', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event35', 5, 'national', CURRENT_DATE, CURRENT_DATE, 'Jacksonville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event36', 5, 'national', CURRENT_DATE, CURRENT_DATE, 'Jacksonville', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event37', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'Ocala', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event38', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'Ocala', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event39', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'West Palm Beach', 'FL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event40', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'West Palm Beach', 'FL');
```

Then I ran the following targeted query.

```
SELECT
   *
FROM
   comp_events
WHERE
   city = 'Orlando'
   AND state = 'FL';
```


Query Result × | Autotrace ×

SQL HotSpot | 1.758 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				3	7	96
TABLE ACCESS	COMP_EVENTS	FULL	1	3	7	96
Filter Predicates						
AND						
CITY='Orlando'						
STATE='FL'						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	7
consistent gets	7
consistent gets from cache	7
consistent gets pin	7
consistent gets pin (fastpath)	7
CPU used by this session	2
CPU used when call started	2
DB time	2
enqueue releases	1

Of course, a full table scan was required because neither city nor state were indexed attributes. Then I created the following B+-tree indexes and ran the query again.

```
CREATE INDEX
   ce_city_btree
ON
   comp_events(city);
CREATE INDEX
   ce_state_btree
ON
   comp_events(state);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comp_events');

SELECT
   *
FROM
   comp_events
WHERE
   city = 'Orlando'
   AND state = 'FL';
```

Query Result × | Autotrace ×

SQL HotSpot | 1.756 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				3	3	73
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	6	3	3	73
Filter Predicates						
STATE='FL'						
INDEX	CE_CITY_BTREE	RANGE SCAN	6	1	1	34
Access Predicates						
CITY='Orlando'						

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	5
consistent gets	3
consistent gets from cache	3
consistent gets pin	3
consistent gets pin (fastpath)	3
CPU used by this session	4
CPU used when call started	4
DB time	4
enqueue releases	1
enqueue requests	1

The city index was used, but the state index was not. So, I added some events in other states to see if the state index would be used.

```
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event41', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event42', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Dallas', 'TX');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event43', 2, 'regional', CURRENT_DATE, CURRENT_DATE, 'Chicago', 'IL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event44', 2, 'regional', CURRENT_DATE, CURRENT_DATE, 'San Diego', 'CA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event45', 3, 'local', CURRENT_DATE, CURRENT_DATE, 'Huntsville', 'AL');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event46', 3, 'local', CURRENT_DATE, CURRENT_DATE, 'Jackson', 'MS');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event47', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Cheyenne', 'WY');
INSERT INTO
```

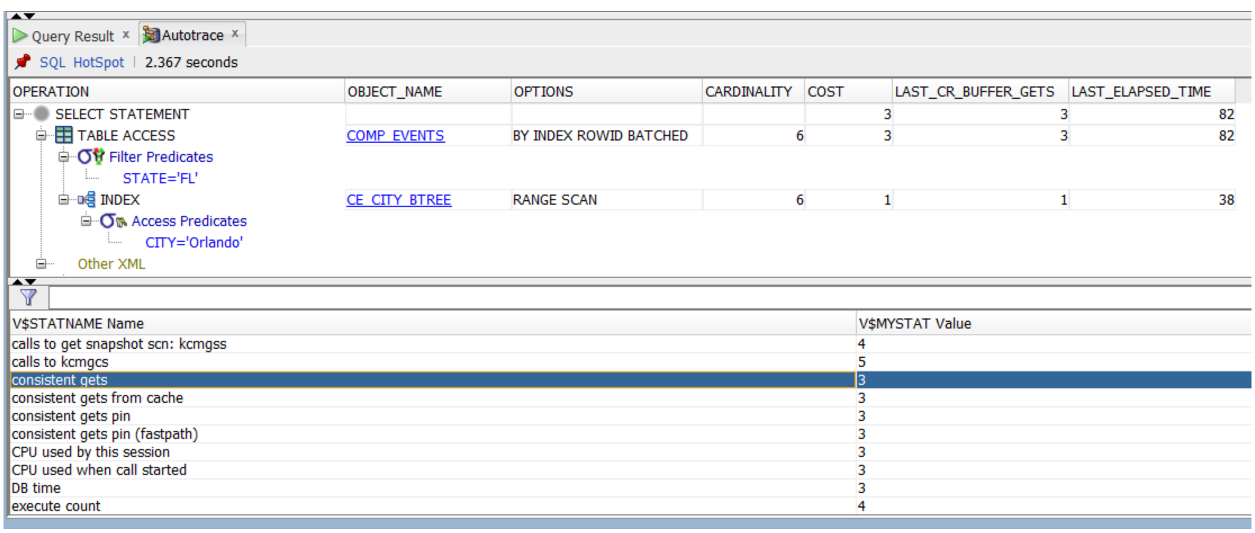
```
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event48', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Tulsa', 'OK');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event49', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Anchorage', 'AK');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event50', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Roanoke', 'VA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event51', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Denver', 'CO');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event52', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Boise', 'ID');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event53', 2, 'regional', CURRENT_DATE, CURRENT_DATE, 'Reno', 'NV');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event54', 2, 'regional', CURRENT_DATE, CURRENT_DATE, 'Tucson', 'AZ');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event55', 3, 'local', CURRENT_DATE, CURRENT_DATE, 'Albuquerque', 'NM');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event56', 3, 'local', CURRENT_DATE, CURRENT_DATE, 'Seattle', 'WA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event57', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Portland', 'OR');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event58', 4, 'state', CURRENT_DATE, CURRENT_DATE, 'Billings', 'MT');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event59', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Bismarck', 'ND');
INSERT INTO
```

```sql
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event60', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Rapid City', 'SD');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event61', 6, 'national', CURRENT_DATE, CURRENT_DATE, 'Lincoln', 'NE');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event62', 6, 'national', CURRENT_DATE, CURRENT_DATE, 'Wichita', 'KS');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event63', 6, 'regional', CURRENT_DATE, CURRENT_DATE, 'Provo', 'UT');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event64', 6, 'regional', CURRENT_DATE, CURRENT_DATE, 'Little Rock', 'AR');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event65', 6, 'local', CURRENT_DATE, CURRENT_DATE, 'Shreveport', 'LA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event66', 6, 'local', CURRENT_DATE, CURRENT_DATE, 'St. Louis', 'MO');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event67', 6, 'state', CURRENT_DATE, CURRENT_DATE, 'Des Moines', 'IA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event68', 6, 'state', CURRENT_DATE, CURRENT_DATE, 'Rochester', 'MN');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event69', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'Green Bay', 'WI');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event70', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'Fort Wayne', 'IN');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event71', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
```

```
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event72', 2, 'national', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event73', 3, 'regional', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event74', 4, 'regional', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event75', 5, 'local', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event76', 6, 'local', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event77', 6, 'state', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event78', 6, 'state', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event79', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event80', 6, 'international', CURRENT_DATE, CURRENT_DATE, 'New York', 'NY');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event81', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event82', 2, 'national', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
   comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
   ('event83', 3, 'regional', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
```

```
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event84', 4, 'regional', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event85', 5, 'local', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event86', 1, 'local', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event87', 2, 'state', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event88', 3, 'state', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event89', 4, 'international', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event90', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Atlanta', 'GA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event91', 1, 'national', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event92', 2, 'national', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event93', 3, 'regional', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event94', 4, 'regional', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event95', 5, 'local', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
```

```
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event96', 1, 'local', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event97', 2, 'state', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event98', 3, 'state', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event99', 4, 'international', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');
INSERT INTO
    comp_events (event_name, competition_id, event_level, start_date, end_date, city, state)
VALUES
    ('event100', 5, 'international', CURRENT_DATE, CURRENT_DATE, 'Pittsburgh', 'PA');

SELECT
    *
FROM
    comp_events
WHERE
    city = 'Orlando'
    AND state = 'FL';
```

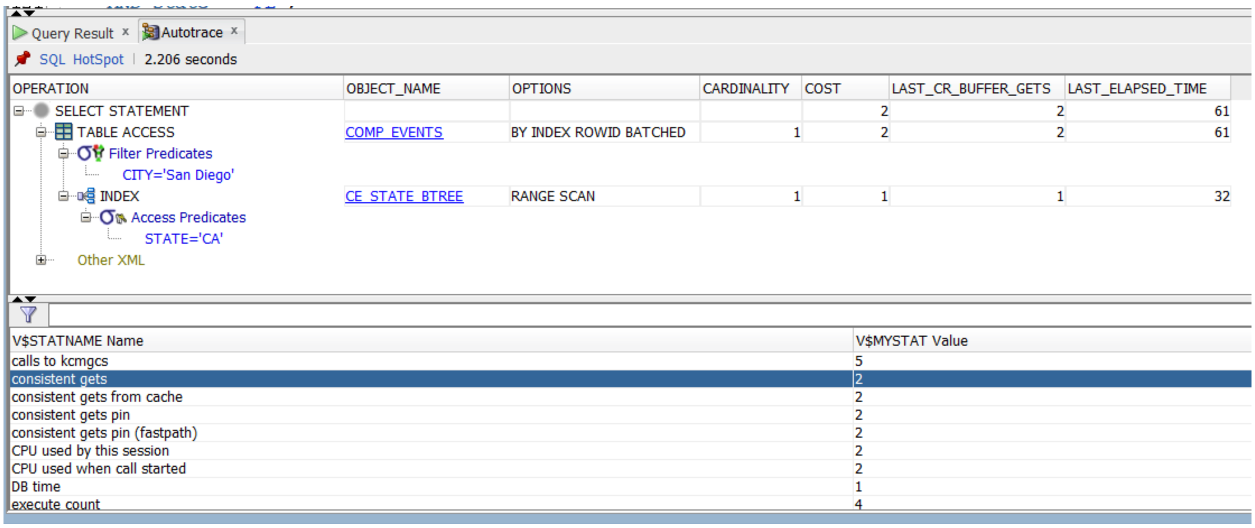


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				3	3	82
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	6	3	3	82
Filter Predicates						
STATE='FL'						
INDEX	CE_CITY_BTREE	RANGE SCAN	6	1	1	38
Access Predicates						
CITY='Orlando'						
Other XML						

V$STATNAME Name	V$MYSTAT Value
calls to get snapshot scn: kcmgss	4
calls to kcmgcs	5
consistent gets	3
consistent gets from cache	3
consistent gets pin	3
consistent gets pin (fastpath)	3
CPU used by this session	3
CPU used when call started	3
DB time	3
execute count	4

Same result, so I changed the query parameters.

```
SELECT
   *
FROM
   comp_events
WHERE
   city = 'San Diego'
   AND state = 'CA';
```



Query Result × | Autotrace ×

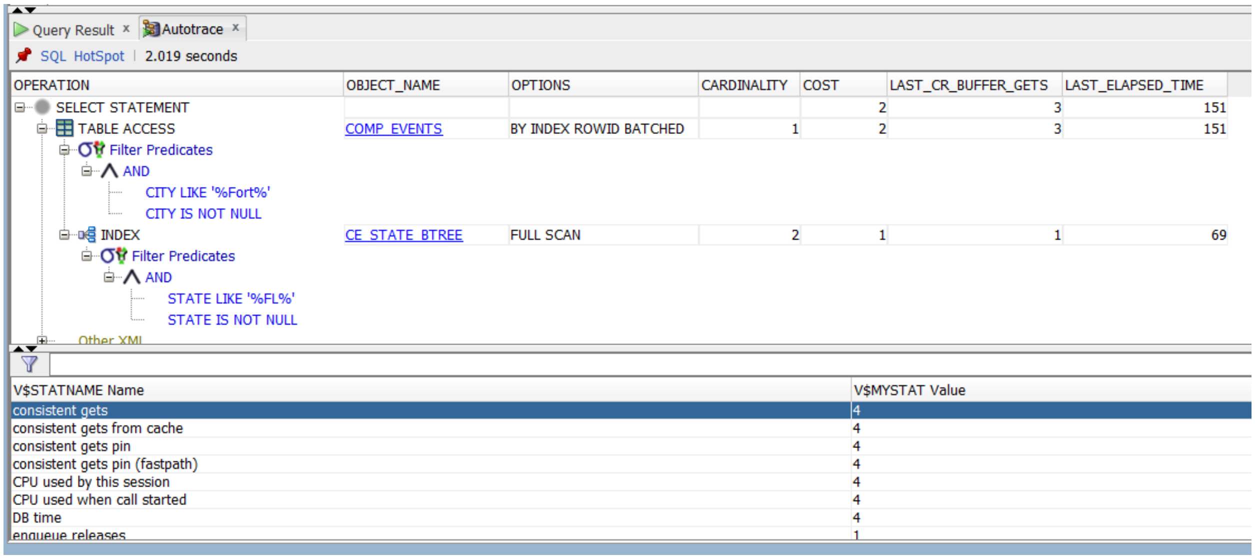
SQL HotSpot | 2.206 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				2	2	61
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	1	2	2	61
Filter Predicates						
CITY='San Diego'						
INDEX	CE_STATE_BTREE	RANGE SCAN	1	1	1	32
Access Predicates						
STATE='CA'						
Other XML						

V$STATNAME Name	V$MYSTAT Value
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
DB time	1
execute count	4

The state index was used, but not the city index. Next, I tried a wildcard query.

```
SELECT
   *
FROM
   comp_events
WHERE
   city LIKE '%Fort%'
   AND state LIKE '%FL%';
```



Query Result × | Autotrace ×

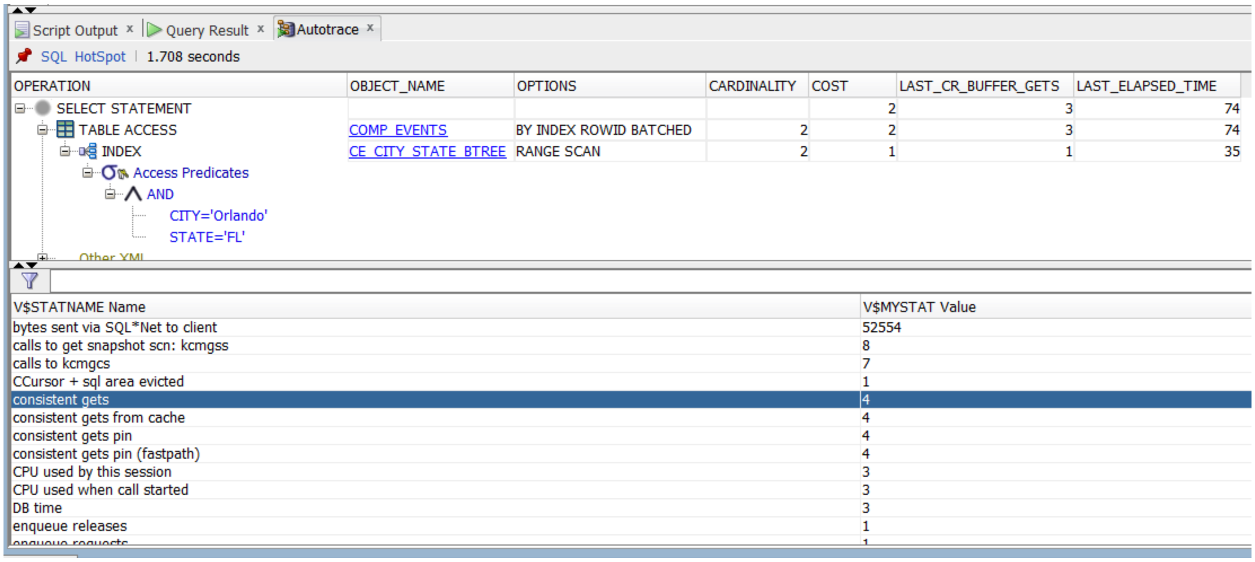
SQL HotSpot | 2.019 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				2	3	151
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	1	2	3	151
Filter Predicates						
AND						
CITY LIKE '%Fort%'						
CITY IS NOT NULL						
INDEX	CE_STATE_BTREE	FULL SCAN	2	1	1	69
Filter Predicates						
AND						
STATE LIKE '%FL%'						
STATE IS NOT NULL						
Other XML						

V$STATNAME Name	V$MYSTAT Value
consistent gets	4
consistent gets from cache	4
consistent gets pin	4
consistent gets pin (fastpath)	4
CPU used by this session	4
CPU used when call started	4
DB time	4
enqueue releases	1

The results were the same except for the type of scan done on the state index. Next, I dropped the city and state indexes and created a composite index. Then I ran the same three queries.

```
DROP INDEX
   ce_city_btree;
DROP INDEX
   ce_state_btree;

CREATE INDEX
   ce_city_state_btree
ON
   comp_events(city, state);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comp_events');

SELECT
   *
FROM
   comp_events
WHERE
   city = 'Orlando'
   AND state = 'FL';
```



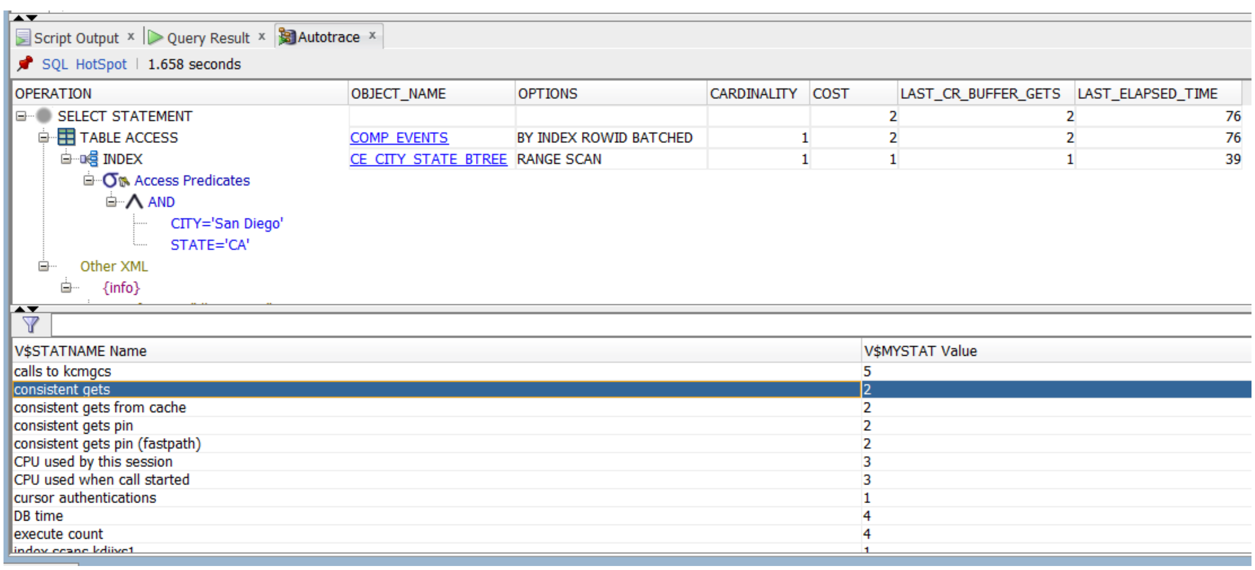
Script Output × | Query Result × | Autotrace ×

SQL HotSpot | 1.708 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				2	3	74
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	2	2	3	74
INDEX	CE_CITY_STATE_BTREE	RANGE SCAN	2	1	1	35
Access Predicates						
AND						
CITY='Orlando'						
STATE='FL'						
Other XML						

V$STATNAME Name	V$MYSTAT Value
bytes sent via SQL*Net to client	52554
calls to get snapshot scn: kcmgss	8
calls to kcmgcs	7
CCursor + sql area evicted	1
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
DB time	3
enqueue releases	1

SELECT
    *
FROM
    comp_events
WHERE
    city = 'San Diego'
    AND state = 'CA';



Script Output × | Query Result × | Autotrace ×

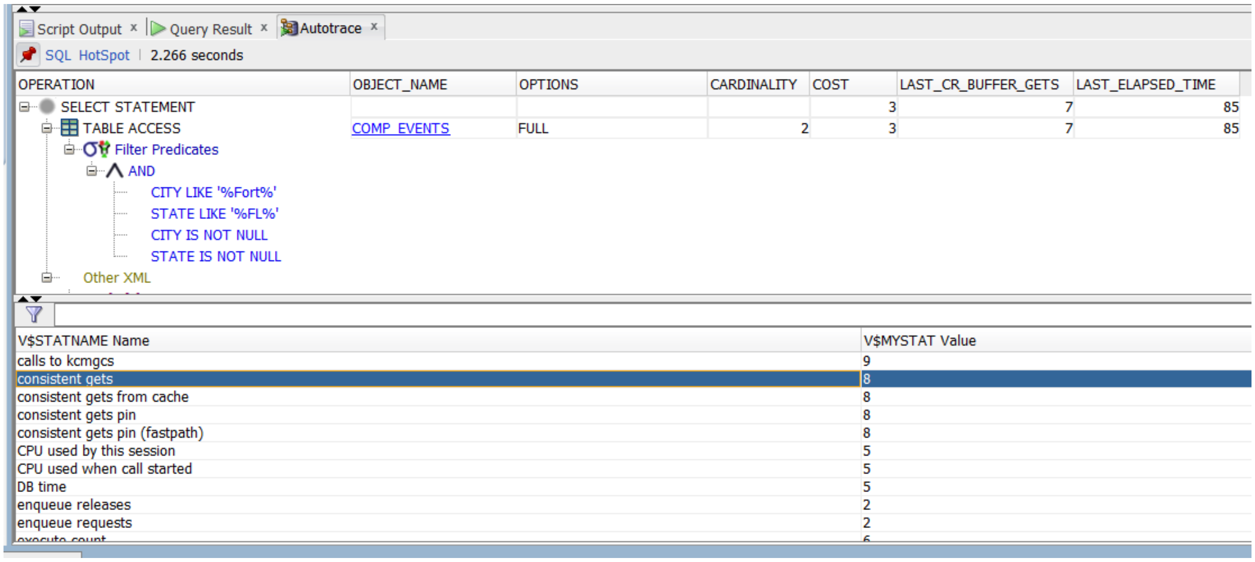
SQL HotSpot | 1.658 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				2	2	76
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID BATCHED	1	2	2	76
INDEX	CE_CITY_STATE_BTREE	RANGE SCAN	1	1	1	39
Access Predicates						
AND						
CITY='San Diego'						
STATE='CA'						
Other XML						
{info}						

V$STATNAME Name	V$MYSTAT Value
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	3
CPU used when call started	3
cursor authentications	1
DB time	4
execute count	4

SELECT
    *
FROM
    comp_events

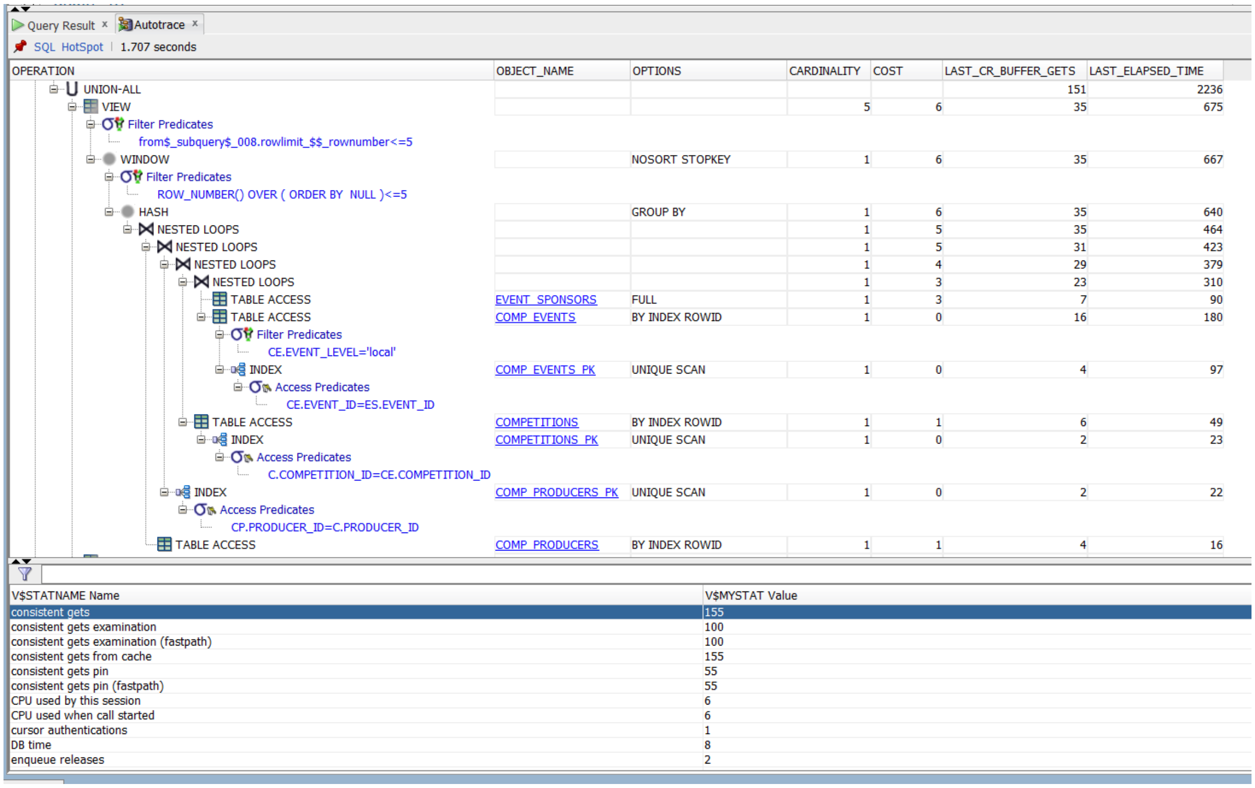
```
WHERE
   city LIKE '%Fort%'
   AND state LIKE '%FL%';
```



The first two queries used the composite index. The third query did not, which I suppose was due to the less targeted nature of the wildcard query. This seems conceptually consistent with the execution of the wildcard query using separate indexes in the sense that the type of index scan went from range to full. My take-away is that composite indexes are best used when you know a certain type of targeted query will be run repeatedly. Otherwise, stick to separate indexes so you have a better chance of the indexes being used.

With the composite index, the consistent_gets jumped from two for the second targeted query to eight for the wildcard query. In absolute terms, that is not a significant difference. However, in relative terms, that is a very significant increase, which speaks to the potential benefits of indexes on large tables. In all the queries on the comp_events table, the consistent_gets and Last_CR_Buffer_Gets were very small, most likely due to the small size of the table. I suppose I would get more interesting results from a much larger table.

For my final experiment, I wanted to optimize my complex union query (the first one) from the last assignment. I thought a bitmap index on comp_events(event_level) would help. Likewise, as the database grows, a B+-tree on comp_producers(prod_name) might help too. For brevity's sake, the query is not repeated here. Click here to review it.



Query Result × Autotrace ×

SQL HotSpot | 1.707 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
UNION-ALL					151	2236
VIEW			5	6	35	675
Filter Predicates						
from$_subquery$_008.rowlimit_$$_rownumber<=5						
WINDOW		NOSORT STOPKEY	1	6	35	667
Filter Predicates						
ROW_NUMBER() OVER (ORDER BY NULL)<=5						
HASH		GROUP BY	1	6	35	640
NESTED LOOPS			1	5	35	464
NESTED LOOPS			1	5	31	423
NESTED LOOPS			1	4	29	379
NESTED LOOPS			1	3	23	310
TABLE ACCESS	EVENT_SPONSORS	FULL	1	3	7	90
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID	1	0	16	180
Filter Predicates						
CE.EVENT_LEVEL='local'						
INDEX	COMP_EVENTS_PK	UNIQUE SCAN	1	0	4	97
Access Predicates						
CE.EVENT_ID=ES.EVENT_ID						
TABLE ACCESS	COMPETITIONS	BY INDEX ROWID	1	1	6	49
INDEX	COMPETITIONS_PK	UNIQUE SCAN	1	0	2	23
Access Predicates						
C.COMPETITION_ID=CE.COMPETITION_ID						
INDEX	COMP_PRODUCERS_PK	UNIQUE SCAN	1	0	2	22
Access Predicates						
CP.PRODUCER_ID=C.PRODUCER_ID						
TABLE ACCESS	COMP_PRODUCERS	BY INDEX ROWID	1	1	4	16

V$STATNAME Name	V$MYSTAT Value
consistent gets	155
consistent gets examination	100
consistent gets examination (fastpath)	100
consistent gets from cache	155
consistent gets pin	55
consistent gets pin (fastpath)	55
CPU used by this session	6
CPU used when call started	6
cursor authentications	1
DB time	8
enqueue releases	2

The consistent_gets are 155 with all the indexes I implemented so far. Looking at the autotrace results, it appears the primary key index of each table except event_sponsors was scanned to facilitate the joins. Then the rowids from the index scans were used to facilitate table access. A full scan was done on event_sponsors presumably due to the aggregation function in the SELECT clause. The new indexes were implemented as follows.

```
CREATE BITMAP INDEX
   event_level_bmap
ON
   comp_events(event_level);
CREATE INDEX
   prod_name_btree
ON
   comp_producers(prod_name);

EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comp_events');
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('db870', 'comp_producers');
```

Query Result × | Autotrace ×

SQL HotSpot | 2.018 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
UNION-ALL					151	2617
VIEW			5	6	35	871
Filter Predicates						
from$_subquery$_008.rowlimit_$$_rownumber<=5						
WINDOW		NOSORT STOPKEY	1	6	35	863
Filter Predicates						
ROW_NUMBER() OVER (ORDER BY NULL)<=5						
HASH		GROUP BY	1	6	35	838
NESTED LOOPS			1	5	35	478
NESTED LOOPS			1	5	31	429
NESTED LOOPS			1	4	29	386
NESTED LOOPS			1	3	23	316
TABLE ACCESS	EVENT_SPONSORS	FULL	1	3	7	90
TABLE ACCESS	COMP_EVENTS	BY INDEX ROWID	1	0	16	181
Filter Predicates						
CE.EVENT_LEVEL='local'						
INDEX	COMP_EVENTS_PK	UNIQUE SCAN	1	0	4	76
Access Predicates						
CE.EVENT_ID=ES.EVENT_ID						
TABLE ACCESS	COMPETITIONS	BY INDEX ROWID	1	1	6	48
INDEX	COMPETITIONS_PK	UNIQUE SCAN	1	0	2	17
Access Predicates						
C.COMPETITION_ID=CE.COMPETITION_ID						
INDEX	COMP_PRODUCERS_PK	UNIQUE SCAN	1	0	2	22
Access Predicates						
CP.PRODUCER_ID=C.PRODUCER_ID						
TABLE ACCESS	COMP_PRODUCERS	BY INDEX ROWID	1	1	4	18

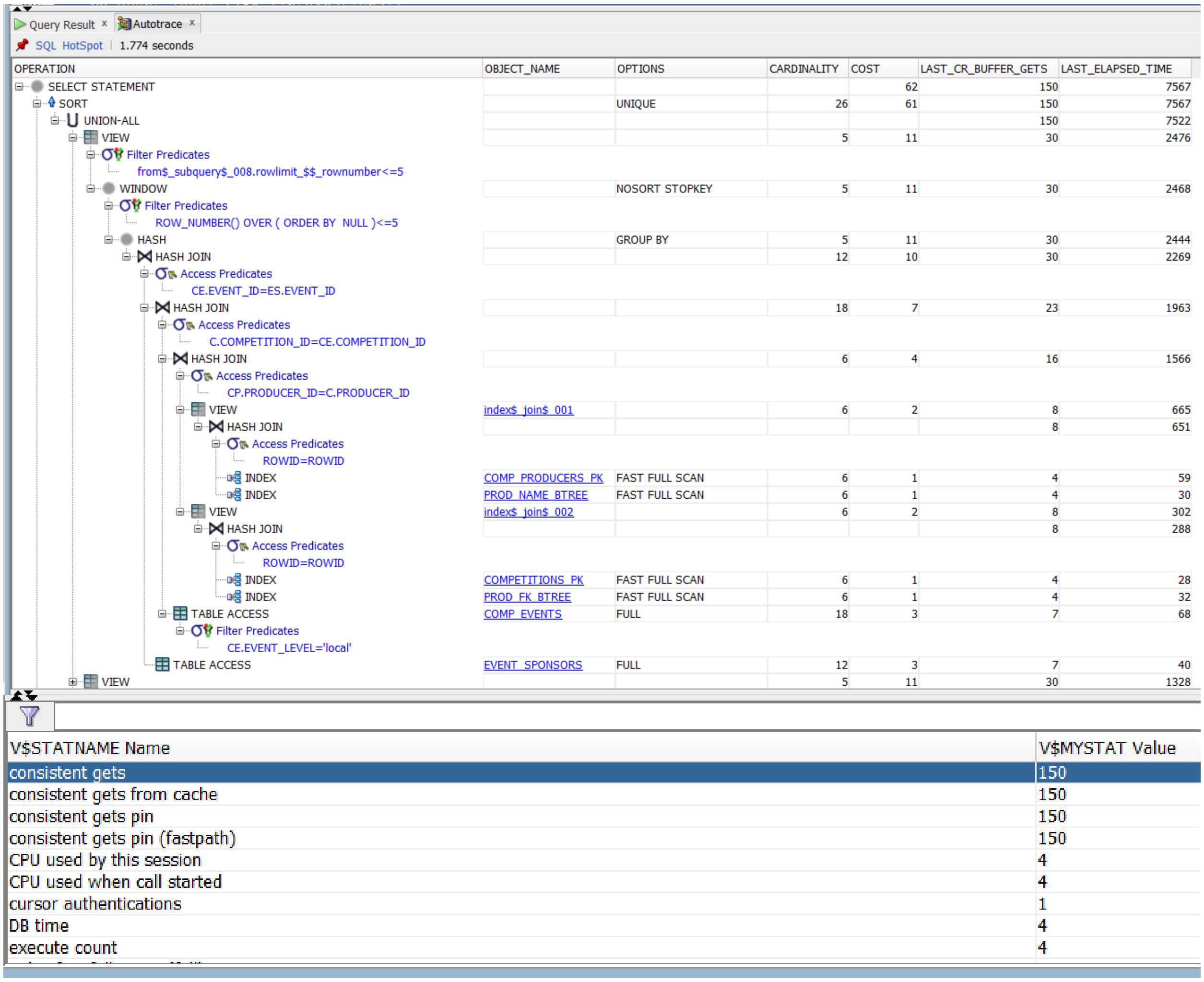
V$STATNAME Name	V$MYSTAT Value
consistent gets	155
consistent gets examination	100
consistent gets examination (fastpath)	100
consistent gets from cache	155
consistent gets pin	55
consistent gets pin (fastpath)	55
CPU used by this session	6
CPU used when call started	6
cursor authentications	1
DB time	7
enqueue releases	1

The consistent_gets remained 155 with the new indexes. I did not expect the prod_name index to be used since the table only has six records, but I thought the event_level index would be used (based on the results of the category_name experiment). Making comp_events_pk a covering index for event_level would eliminate the table access, but that would potentially preclude use of the index for many other queries. That seems like a bad idea for a primary key index. Maybe as the table grows, the bitmap index would become advantageous.

Just for kicks, I re-ran the gather_table_stats statements with sample size at 100%. This time I also included the event_sponsors table, which had not been analyzed since its creation.

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('db870', 'comp_events', estimate_percent => 100);
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('db870', 'competitions', estimate_percent => 100);
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('db870', 'comp_producers', estimate_percent => 100);
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ('db870', 'event_sponsors', estimate_percent => 100);
```

When I re-ran the query, I got interesting autotrace results.



Query Result | Autotrace

SQL HotSpot | 1.774 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				62	150	7567
SORT		UNIQUE	26	61	150	7567
UNION-ALL					150	7522
VIEW			5	11	30	2476
Filter Predicates						
from$_subquery$_008.rowlimit_$$_rownumber<=5						
WINDOW		NOSORT STOPKEY	5	11	30	2468
Filter Predicates						
ROW_NUMBER() OVER (ORDER BY NULL)<=5						
HASH		GROUP BY	5	11	30	2444
HASH JOIN			12	10	30	2269
Access Predicates						
CE.EVENT_ID=ES.EVENT_ID						
HASH JOIN			18	7	23	1963
Access Predicates						
C.COMPETITION_ID=CE.COMPETITION_ID						
HASH JOIN			6	4	16	1566
Access Predicates						
CP.PRODUCER_ID=C.PRODUCER_ID						
VIEW	index$_join$_001		6	2	8	665
HASH JOIN					8	651
Access Predicates						
ROWID=ROWID						
INDEX	COMP_PRODUCERS_PK	FAST FULL SCAN	6	1	4	59
INDEX	PROD_NAME_BTREE	FAST FULL SCAN	6	1	4	30
VIEW	index$_join$_002		6	2	8	302
HASH JOIN					8	288
Access Predicates						
ROWID=ROWID						
INDEX	COMPETITIONS_PK	FAST FULL SCAN	6	1	4	28
INDEX	PROD_FK_BTREE	FAST FULL SCAN	6	1	4	32
TABLE ACCESS	COMP_EVENTS	FULL	18	3	7	68
Filter Predicates						
CE.EVENT_LEVEL='local'						
TABLE ACCESS	EVENT_SPONSORS	FULL	12	3	7	40
VIEW			5	11	30	1328

V$STATNAME Name	V$MYSTAT Value
consistent gets	150
consistent gets from cache	150
consistent gets pin	150
consistent gets pin (fastpath)	150
CPU used by this session	4
CPU used when call started	4
cursor authentications	1
DB time	4
execute count	4

Unlike last time, the prod_name_btree index was used (surprisingly) as was the prod_fk_btree index, but the comp_events_pk index was not. Like last time, the competitions_pk and comp_producers_pk indexes were used. The consistent_gets dropped by five, while the Last_CR_Buffer_Gets dropped by one. Even though the performance did not improve much, this experiment illustrates the potential variance among execution plans. I am not sure if the execution plan changed as a result of increasing the sample size or gathering stats on the event_sponsors table or both.

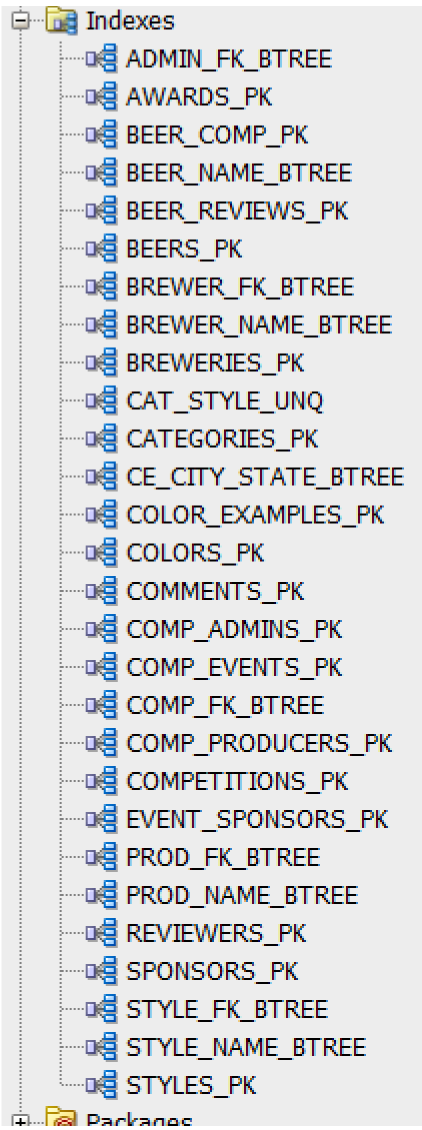
Next, I changed all LIKE operators to =. The execution plan was identical, and the I/O cost was nearly identical. This makes sense considering the equality returned multiple matches just as the LIKE did. Had the event_level attribute been subject to a UNIQUE constraint, the equality would have returned just one result per SELECT clause, and the execution plan probably would have been different.

I wondered about the relationship of indexes to set operations, union in particular. In chapter 15 of the textbook (7th edition), query processing is discussed. Little guidance is offered about optimizing unions (disjunctives). If I understand correctly, the authors indicate optimization is performed within each SELECT statement to retrieve pointers optimally (same as non-set operations). The only difference is that the pointers are aggregated prior to record retrieval from the file(s). Likewise, the lecture slides did not provide any further insight. So, there does not seem to be any union-specific indexing technique.

I decided to drop the bitmap index on event_level since it was never used. I kept the B+-tree index on prod_name since that attribute will likely be used as a predicate in future queries as the database grows, and my experiment results indicate it will be used at times.

```
DROP INDEX
   event_level_bmap;
```

This left the database with no active bitmap indexes.



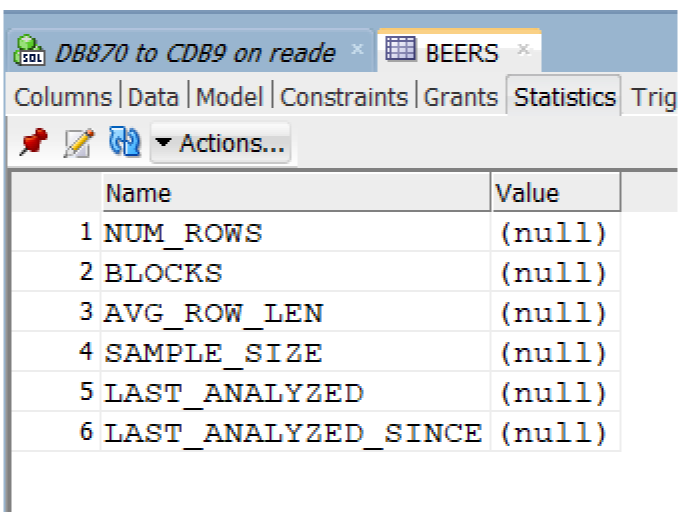
Column Statistics (Assignment 4)

Between assignments three and four, I loaded bulk data into my schema. The “Other Topics” section of this document contains the details of that effort. The following table summarizes the results. The comments table has a LONG attribute that contains textual comments that exceed the maximum size of VARCHAR2.

Table	**Total Records**	**Pre-existing Records**	**New Records**
beers	114,084	5,890	108,194
brewers	8,789	1,414	7,375
styles	226	141	85
reviewers	29,097	0	29,097
beer_reviews	2,805,537	0	2,805,537
comments	2,775,682	0	2,775,682

For my first experiment, I cleared all statistics in my schema and ran some queries to establish a performance baseline.

EXECUTE dbms_stats.delete_schema_stats('DB870');




```
SELECT
   b.beer_name,
   b.beer_id,
   avg(rev.overall_rating) as avg_rating
FROM
   beers b
   INNER JOIN beer_reviews rev
      ON b.beer_id = rev.beer_id
GROUP BY
```

```sql
    b.beer_name,
    b.beer_id
ORDER BY
    avg_rating DESC,
    beer_name ASC;


SELECT
    b.beer_name,
    b.beer_id,
    count(c.review_comment) as count
FROM
    beers b
    INNER JOIN comments c
        ON b.beer_id = c.beer_id
GROUP BY
    b.beer_name,
    b.beer_id
ORDER BY
    count DESC,
    beer_name ASC;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	7812	14090	14790543
SORT		ORDER BY	1939621	50	7812	14090	14790543
HASH		GROUP BY	1939621	107406	7812	14090	14638173
HASH JOIN			1939621	2775682	7715	14090	12759564
Access Predicates							
C.BEER_ID=B.BEER_ID							
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	9792
INDEX	COMMENTS_PK	FAST FULL SCAN	1939620	2775682	3464	12796	4273223
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	78644
consistent gets direct	0
consistent gets examination	58768
consistent gets examination (fastpath)	58766
consistent gets from cache	78644
consistent gets pin	19876
consistent gets pin (fastpath)	19693

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	7812	14090	14074578
SORT		ORDER BY	1939621	50	7812	14090	14074578
HASH		GROUP BY	1939621	107406	7812	14090	13932128
HASH JOIN			1939621	2775682	7715	14090	12007238
Access Predicates							
B.BEER_ID=C.BEER_ID							
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	11485
INDEX	COMMENTS_PK	FAST FULL SCAN	1939620	2775682	3464	12796	3120219
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	111416
consistent gets direct	0
consistent gets examination	61422
consistent gets examination (fastpath)	61422
consistent gets from cache	111416
consistent gets pin	49994
consistent gets pin (fastpath)	49994

Curiously, the primary key index of the comments table was used by both queries even though the first query did not join the comments table. Next, I gathered statistics on all tables in the schema and re-ran the same queries in the same order to determine how much performance improved.

EXECUTE dbms_stats.gather_schema_stats('DB870');

SQL HotSpot | 14.818 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	7812	14090	13272668
SORT		ORDER BY	1939621	50	7812	14090	13272668
HASH		GROUP BY	1939621	107406	7812	14090	13130734
HASH JOIN			1939621	2775682	7715	14090	11317817
Access Predicates							
B.BEER_ID=C.BEER_ID							
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	17785
INDEX	COMMENTS_PK	FAST FULL SCAN	1939620	2775682	3464	12796	2650655
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	14090
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	14090
consistent gets pin	14090
consistent gets pin (fastpath)	14084

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	7812	14090	13100337
SORT		ORDER BY	1939621	50	7812	14090	13100337
HASH		GROUP BY	1939621	107406	7812	14090	12961788
HASH JOIN			1939621	2775682	7715	14090	11166085
Access Predicates							
B.BEER_ID=C.BEER_ID							
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	12997
INDEX	COMMENTS_PK	FAST FULL SCAN	1939620	2775682	3464	12796	2582262
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	14090
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	14090
consistent gets pin	14090
consistent gets pin (fastpath)	14090

As we can see, the last_elapsed_time decreased slightly for both queries, while the consistent_gets dropped significantly for both queries. The execution plans did not change.

Next, I experimented with column group statistics using the ratings columns in the beer_reviews table. First, I cleared table statistics and ran a query to get a baseline.

EXECUTE dbms_stats.delete_table_stats('DB870', 'beer_reviews');

```
SELECT
   b.beer_name,
   b.beer_id,
   rev.taste_rating,
   rev.aroma_rating
FROM
   beers b
   INNER JOIN beer_reviews rev
      ON b.beer_id = rev.beer_id
```

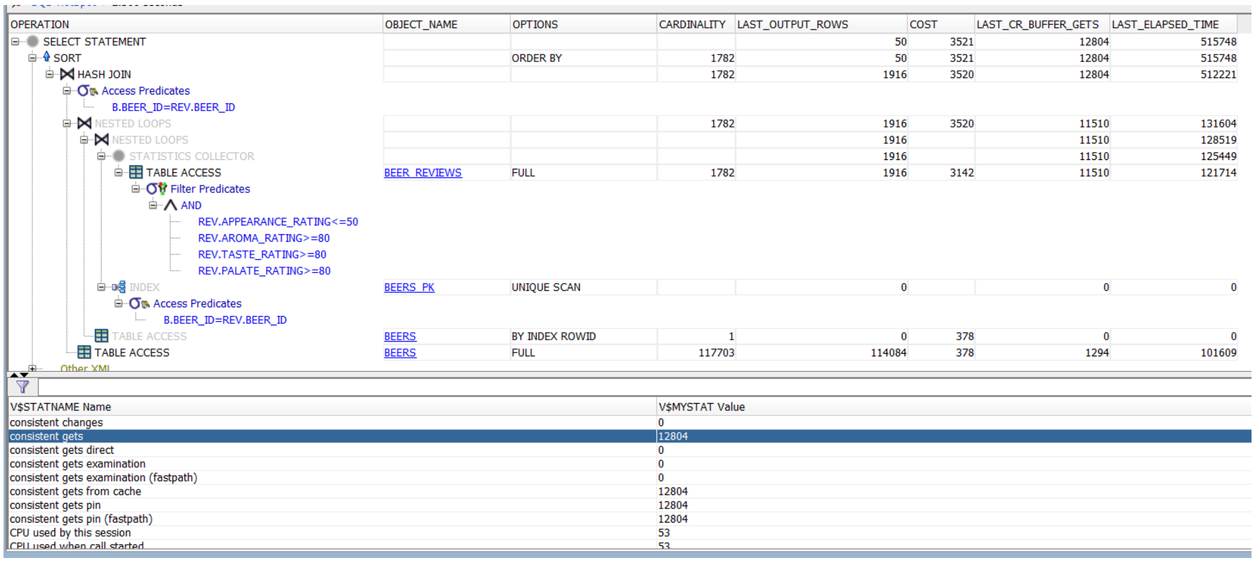
```
WHERE
   rev.taste_rating >= 80
   AND rev.aroma_rating >= 80
   AND rev.palate_rating >= 80
   AND rev.appearance_rating <= 50
ORDER BY
   taste_rating DESC,
   aroma_rating DESC,
   beer_name ASC;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	3521	12804	659828
SORT		ORDER BY	1782	50	3521	12804	659828
HASH JOIN			1782	1916	3520	12804	655106
Access Predicates							
B.BEER_ID=REV.BEER_ID							
NESTED LOOPS			1782	1916	3520	11510	171534
NESTED LOOPS				1916		11510	167339
STATISTICS COLLECTOR				1916		11510	163359
TABLE ACCESS	BEER_REVIEWS	FULL	1782	1916	3142	11510	157921
Filter Predicates							
AND							
REV.APPEARANCE_RATING<=50							
REV.AROMA_RATING>=80							
REV.TASTE_RATING>=80							
REV.PALATE_RATING>=80							
INDEX	BEERS_PK	UNIQUE SCAN		0		0	0
Access Predicates							
B.BEER_ID=REV.BEER_ID							
TABLE ACCESS	BEERS	BY INDEX ROWID	1	0	378	0	0
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	132572
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	12804
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	12804
consistent gets pin	12804
consistent gets pin (fastpath)	12804
CPU used by this session	67
CPU used when call started	67
CR blocks created	0

Then, I gathered column group statistics and re-ran the query to determine how much performance improves.

```
SELECT
   dbms_stats.create_extended_stats('DB870', 'beer_reviews', '(overall_rating, appearance_rating,
aroma_rating, palate_rating, taste_rating)')
FROM
   dual;

EXECUTE dbms_stats.gather_table_stats('DB870', 'beer_reviews');
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	3521	12804	515748
SORT		ORDER BY	1782	50	3521	12804	515748
HASH JOIN			1782	1916	3520	12804	512221
Access Predicates							
B.BEER_ID=REV.BEER_ID							
NESTED LOOPS			1782	1916	3520	11510	131604
NESTED LOOPS				1916		11510	128519
STATISTICS COLLECTOR				1916		11510	125449
TABLE ACCESS	BEER_REVIEWS	FULL	1782	1916	3142	11510	121714
Filter Predicates							
AND							
REV.APPEARANCE_RATING<=50							
REV.AROMA_RATING>=80							
REV.TASTE_RATING>=80							
REV.PALATE_RATING>=80							
INDEX	BEERS_PK	UNIQUE SCAN		0		0	0
Access Predicates							
B.BEER_ID=REV.BEER_ID							
TABLE ACCESS	BEERS	BY INDEX ROWID	1	0	378	0	0
TABLE ACCESS	BEERS	FULL	117703	114084	378	1294	101609
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	12804
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	12804
consistent gets pin	12804
consistent gets pin (fastpath)	12804
CPU used by this session	53
CPU used when call started	53

As we can see, last_elapsed_time dropped moderately, while consistent_gets and the execution plan remained unchanged. That was not what I was expecting to see. Several online articles indicated the optimizer can benefit from column group statistics when several columns are used together in the predicate. I suppose the specific nature of the relationships between the columns in the predicate could influence the results, but I was not sure what to do next. So, I moved on.

Bitmap Indexes (Assignment 4)

Since the ratings columns in beer_reviews all have low cardinality with a small number of values, I thought they would be good candidates for another bitmap index experiment.

```
CREATE BITMAP INDEX
   ovr_bmap
ON
   beer_reviews(overall_rating);

CREATE BITMAP INDEX
   apr_bmap
ON
   beer_reviews(appearance_rating);

CREATE BITMAP INDEX
   arr_bmap
ON
   beer_reviews(aroma_rating);

CREATE BITMAP INDEX
   par_bmap
ON
```

```
   beer_reviews(palate_rating);

CREATE BITMAP INDEX
   tar_bmap
ON
   beer_reviews(taste_rating);

EXECUTE dbms_stats.gather_table_stats('DB870', 'beer_reviews');
```

I tested with the same query from the previous experiment.

```
SELECT
   b.beer_name,
   b.beer_id,
   rev.taste_rating,
   rev.aroma_rating
FROM
   beers b
   INNER JOIN beer_reviews rev
      ON b.beer_id = rev.beer_id
WHERE
   rev.taste_rating >= 80
   AND rev.aroma_rating >= 80
   AND rev.palate_rating >= 80
   AND rev.appearance_rating <= 50
ORDER BY
   taste_rating DESC,
   aroma_rating DESC,
   beer_name ASC;
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	1575	3126	477062
SORT		ORDER BY	6087	50	1575	3126	477062
HASH JOIN			6087	1916	1573	3126	473246
Access Predicates							
B.BEER_ID=REV.BEER_ID							
NESTED LOOPS			6087	1916	1573	1832	54902
NESTED LOOPS				1916		1832	51742
STATISTICS COLLECTOR				1916		1832	48681
TABLE ACCESS	BEER_REVIEWS	BY INDEX ROWID BATCHED	6087	1916	1195	1832	45491
BITMAP CONVERSION		TO ROWIDS		1916		246	37593
BITMAP AND				1		246	35431
BITMAP MERGE				12		33	5865
BITMAP INDEX	APR_BMAP	RANGE SCAN		62		33	293
Access Predicates							
REV.APPEARANCE_RATING<=50							
Filter Predicates							
REV.APPEARANCE_RATING<=50							
BITMAP MERGE				19		66	7617
BITMAP INDEX	ARR_BMAP	RANGE SCAN		129		66	435
Access Predicates							
REV.AROMA_RATING>=80							
Filter Predicates							
REV.AROMA_RATING>=80							
BITMAP MERGE				20		71	8148
BITMAP INDEX	TAR_BMAP	RANGE SCAN		140		71	517
Access Predicates							
REV.TASTE_RATING>=80							
Filter Predicates							
REV.TASTE_RATING>=80							
BITMAP MERGE				24		76	6757
BITMAP INDEX	PAR_BMAP	RANGE SCAN		149		76	506
Access Predicates							
REV.PALATE_RATING>=80							
Filter Predicates							
REV.PALATE_RATING>=80							
INDEX	BEERS_PK	UNIQUE SCAN		0		0	0
Access Predicates							
B.BEER_ID=REV.BEER_ID							
TABLE ACCESS	BEERS	BY INDEX ROWID	1	0	378	0	0
TABLE ACCESS	BEERS	FULL	114084	114084	378	1294	112351

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	3126
consistent gets direct	0
consistent gets examination	4
consistent gets examination (fastpath)	4
consistent gets from cache	3126
consistent gets pin	3122
consistent gets pin (fastpath)	3122
CPU used by this session	53
CPU used when call started	53
CR blocks created	0

As we can see, last_elapsed_time dropped moderately, while consistent_gets dropped significantly. The execution plan changed quite a bit and used four of the new bitmap indexes.

Optimizer Modes (Assignment 4)

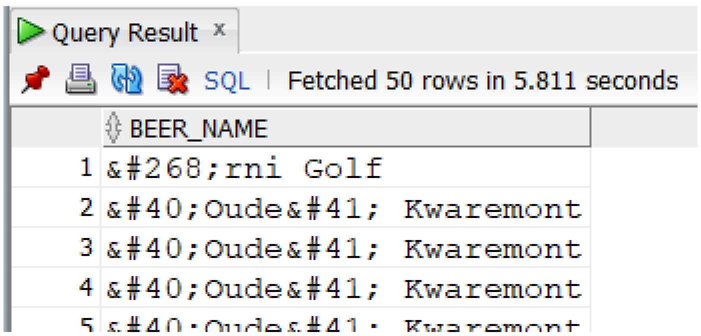
Next, I experimented with session-level optimizer modes. I first tried to query substrings in the review_comment attribute of the comments table and discovered that is not possible with the LONG data type. So, I tried to convert review_comment from LONG to CLOB. Unfortunately, a tablespace error occurred.

ORA-01652: unable to extend temp segment by 1024 in tablespace STUDENTS
01652. 00000 - "unable to extend temp segment by %s in tablespace %s"
*Cause: Failed to allocate an extent of the required number of blocks for
a temporary segment in the tablespace indicated.

So, I used the comment_date attribute instead.

ALTER SESSION SET OPTIMIZER_MODE = ALL_ROWS;

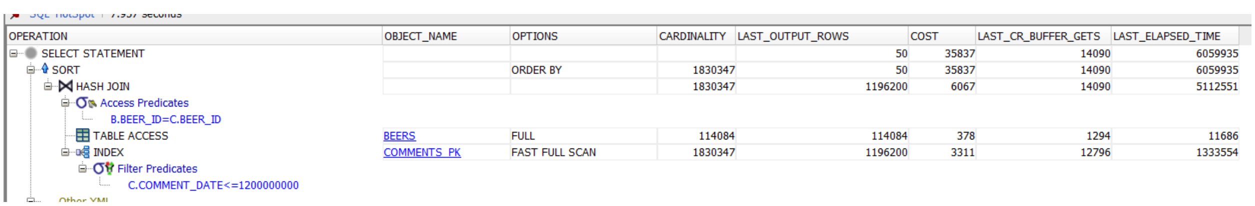
```
SELECT
   b.beer_name
FROM
   beers b
   INNER JOIN comments c
      ON b.beer_id = c.beer_id
WHERE
   c.comment_date <= 1200000000
ORDER BY
   b.beer_name ASC;
```



Query Result

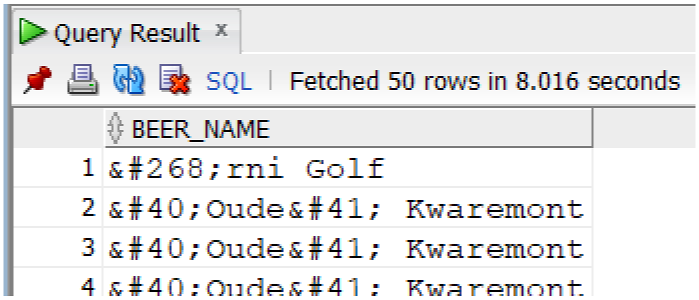
SQL | Fetched 50 rows in 5.811 seconds

	BEER_NAME
1	Črni Golf
2	(Oude) Kwaremont
3	(Oude) Kwaremont
4	(Oude) Kwaremont
5	(Oude) Kwaremont



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	35837	14090	6059935
SORT		ORDER BY	1830347	50	35837	14090	6059935
HASH JOIN			1830347	1196200	6067	14090	5112551
Access Predicates							
B.BEER_ID=C.BEER_ID							
TABLE ACCESS	BEERS	FULL	114084	114084	378	1294	11686
INDEX	COMMENTS_PK	FAST FULL SCAN	1830347	1196200	3311	12796	1333554
Filter Predicates							
C.COMMENT_DATE<=1200000000							

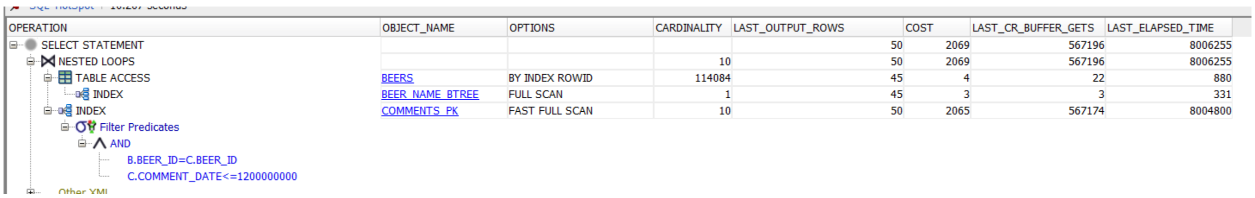
```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;

SELECT
   b.beer_name
FROM
   beers b
   INNER JOIN comments c
      ON b.beer_id = c.beer_id
WHERE
   c.comment_date <= 1200000000
ORDER BY
   b.beer_name ASC;
```



Query Result

SQL | Fetched 50 rows in 8.016 seconds

	BEER_NAME
1	Črni Golf
2	(Oude) Kwaremont
3	(Oude) Kwaremont
4	(Oude) Kwaremont



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	2069	567196	8006255
NESTED LOOPS			10	50	2069	567196	8006255
TABLE ACCESS	BEERS	BY INDEX ROWID	114084	45	4	22	880
INDEX	BEER_NAME_BTREE	FULL SCAN	1	45	3	3	331
INDEX	COMMENTS_PK	FAST FULL SCAN	10	50	2065	567174	8004800
Filter Predicates							
AND							
B.BEER_ID=C.BEER_ID							
C.COMMENT_DATE<=1200000000							

I was confused by these results at first. In FIRST_ROWS_10 mode, the optimizer used the comments_pk and beer_name_btree indexes, accessed the beers table BY INDEX ROWID, and avoided doing a HASH JOIN. In ALL_ROWS mode, only the comments_pk index was used, and a FULL SCAN was done on the beers table, and a HASH JOIN was employed. The FIRST_ROWS_10 execution plan should be faster since more indexes were used, access BY INDEX ROWID generally takes less time than a FULL SCAN, and a HASH JOIN entails startup cost. Yet, the FIRST_ROWS_10 query took longer. I hypothesize that, due to the excessively large size of the inner result set from the comments_pk index, the time required for the NESTED LOOP JOIN more than offset the time required for the FULL SCAN of the relatively small beers table and the HASH JOIN startup cost. Since I was one for three in Assignment 4, I decided to move on to partitioning where I felt confident that I could get "friendly" results.

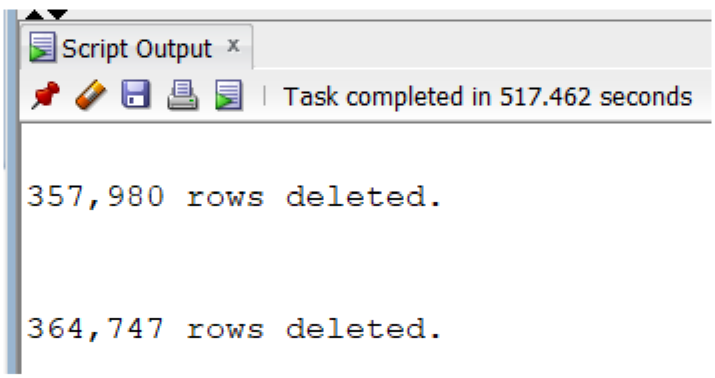
Table Partitioning (Assignment 4)

I found a way to partition an unpartitioned table while keeping the table online. I tried it on the comments table using the following code:

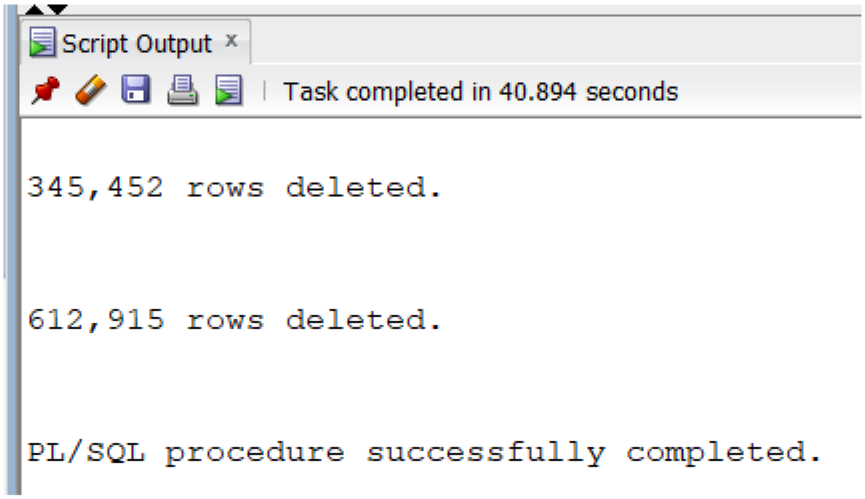
```
ALTER TABLE
   comments
MODIFY
   PARTITION BY RANGE (comment_date) INTERVAL (100000000)
   (PARTITION p1 VALUES LESS THAN (1000000000),
    PARTITION p2 VALUES LESS THAN (1100000000),
    PARTITION p3 VALUES LESS THAN (1200000000),
    PARTITION p4 VALUES LESS THAN (1300000000),
    PARTITION p5 VALUES LESS THAN (1400000000))
   ONLINE;
```

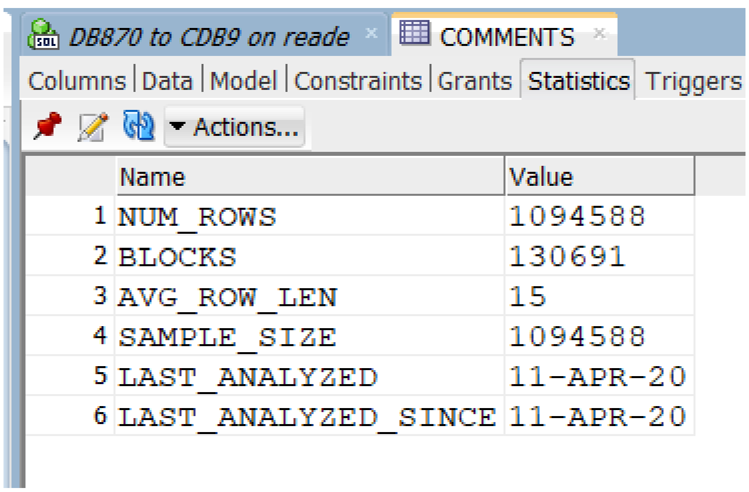
Unfortunately, I got an error. Further research revealed this method only works in Oracle 12c R2, and the course server appears to be running Oracle 12c R1. An alternative was to create a partitioned table, copy the data from the comments table into the new table, enable any lost constraints on the new table, disable downstream constraints, drop the original comments table, rename the new table “comments”, re-enable downstream constraints, redefine any lost indexes on the new comments table, and refresh the new table’s stats. Since I had already twice received tablespace errors indicating limited available storage space, this method would require deletion of a significant amount of data from the existing comments table prior to moving any data to the new table. This solution was not ideal, but I chose to move forward because I wanted to see the benefits of partitioning in action on a very large table. Since I did not know the distribution of rows across dates, I proceeded with caution.

```
DELETE FROM
   comments
WHERE
   comment_date < 1100000000;

DELETE FROM
   comments
WHERE
   comment_date > 1300000000;
```




```
DELETE FROM
   comments
WHERE
   comment_date < 1150000000;

DELETE FROM
   comments
WHERE
   comment_date > 1250000000;

EXECUTE dbms_stats.gather_table_stats('DB870', 'beer_reviews');
```

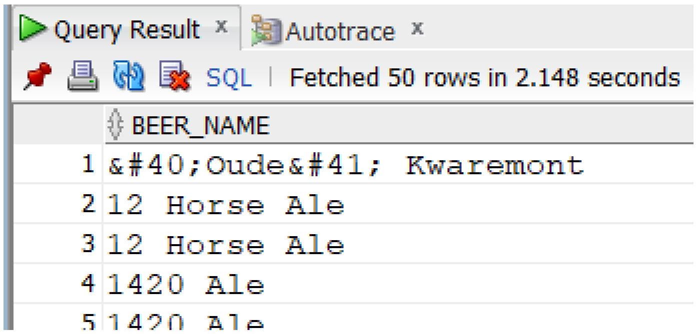


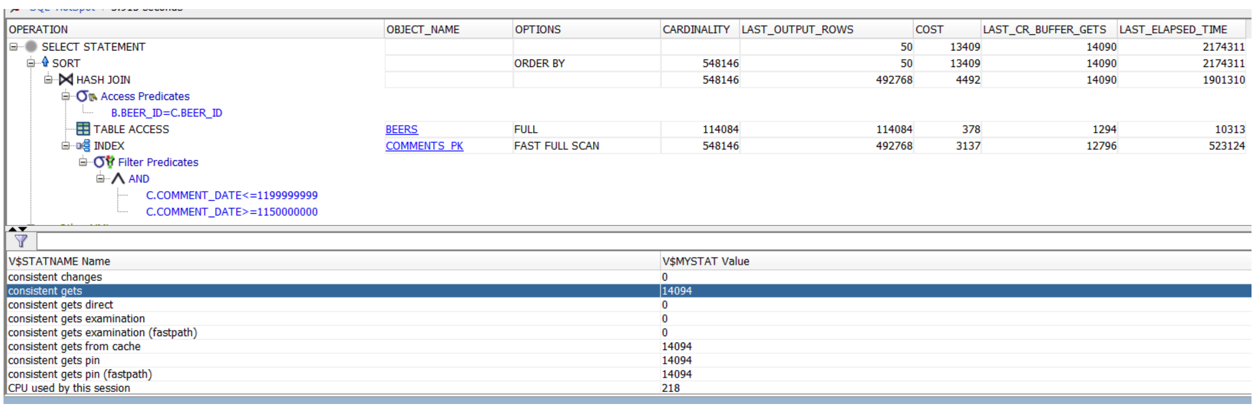


	Name	Value
1	NUM_ROWS	1094588
2	BLOCKS	130691
3	AVG_ROW_LEN	15
4	SAMPLE_SIZE	1094588
5	LAST_ANALYZED	11-APR-20
6	LAST_ANALYZED_SINCE	11-APR-20

That left nearly 1.1 million rows between date values 1150000000 and 1250000000. I ran a baseline query prior to partitioning.

```
SELECT
   b.beer_name
FROM
   beers b
   INNER JOIN comments c
      ON b.beer_id = c.beer_id
WHERE
   c.comment_date BETWEEN 1150000000 AND 1199999999
ORDER BY
   b.beer_name ASC;
```





Then I tried to create a new table from the comments table.

```
CREATE TABLE
   pcomments
   PARTITION BY RANGE (comment_date) INTERVAL (25000000)
   (PARTITION p1 VALUES LESS THAN (1175000000),
   PARTITION p2 VALUES LESS THAN (1200000000),
   PARTITION p3 VALUES LESS THAN (1225000000),
   PARTITION p4 VALUES LESS THAN (1250000000))
AS
   SELECT
      *
   FROM
      comments;
```

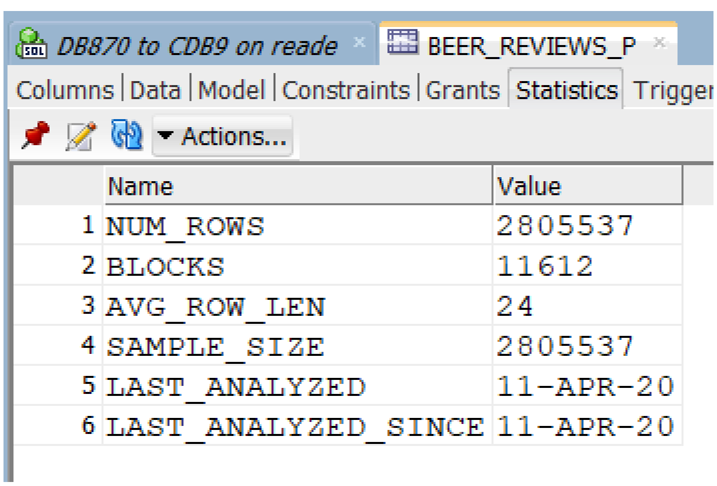
Sadly, I got the following error.

ORA-00997: illegal use of LONG datatype
00997. 00000 - "illegal use of LONG datatype"

At this point, I tried very hard to forget I ever created the comments table. Two valuable lessons were learned. First, plan accordingly and create your partitions up front. Second, never use the LONG data type; use xLOB instead.

Moving on, I decided to set my sights on the beer_reviews table. It too was very large in terms of row count, but the block count was much smaller.

```
CREATE TABLE
   beer_reviews_p
   PARTITION BY RANGE (overall_rating) INTERVAL (10)
   (PARTITION p01 VALUES LESS THAN (20),
   PARTITION p02 VALUES LESS THAN (30),
   PARTITION p03 VALUES LESS THAN (40),
   PARTITION p04 VALUES LESS THAN (50),
   PARTITION p05 VALUES LESS THAN (60),
   PARTITION p06 VALUES LESS THAN (70),
   PARTITION p07 VALUES LESS THAN (80),
   PARTITION p08 VALUES LESS THAN (90),
   PARTITION p09 VALUES LESS THAN (100),
   PARTITION p10 VALUES LESS THAN (110))
AS
   SELECT
      *
   FROM
      beer_reviews;

EXECUTE dbms_stats.gather_table_stats('DB870', 'beer_reviews_p');
```



	Name	Value
1	NUM_ROWS	2805537
2	BLOCKS	11612
3	AVG_ROW_LEN	24
4	SAMPLE_SIZE	2805537
5	LAST_ANALYZED	11-APR-20
6	LAST_ANALYZED_SINCE	11-APR-20

DB870 to CDB9 on reade | BEER_REVIEWS_P

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Ir

Actions...

	PARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE
1	P01	11-APR-20	46633	206	46633	20
2	P02	11-APR-20	51451	225	51451	30
3	P03	11-APR-20	81050	344	81050	40
4	P04	11-APR-20	151178	627	151178	50
5	P05	11-APR-20	319710	1321	319710	60
6	P06	11-APR-20	632664	2599	632664	70
7	P07	11-APR-20	824554	3391	824554	80
8	P08	11-APR-20	538818	2228	538818	90
9	P09	11-APR-20	146228	611	146228	100
10	P10	11-APR-20	13251	60	13251	110

```
ALTER TABLE
   beer_reviews_p
ADD CONSTRAINT beer_reviews_p_pk PRIMARY KEY (reviewer_id, beer_id);

ALTER TABLE
   beer_reviews_p
ADD CONSTRAINT reviewers_p_fk FOREIGN KEY (reviewer_id) REFERENCES reviewers(reviewer_id);

ALTER TABLE
   beer_reviews_p
ADD CONSTRAINT beers_p_fk FOREIGN KEY (beer_id) REFERENCES beers(beer_id);
```

The NOT NULL constraint on the overall_rating attribute carried over to the new table automatically.

DB870 to CDB9 on reade | BEER_REVIEWS_P

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

	CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION	R_OWNER	R_TABLE_NAME	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
1	SYS_C00109406	Check	"OVERALL_RATING" IS NOT NULL	(null)	(null)	(null)	(null)	ENABLED
2	BEERS_P_FK	Foreign_Key	(null)	DB870	BEERS	BEERS_PK	NO ACTION	ENABLED
3	REVIEWERS_P_FK	Foreign_Key	(null)	DB870	REVIEWERS	REVIEWERS_PK	NO ACTION	ENABLED
4	BEER_REVIEWS_P_PK	Primary_Key	(null)	(null)	(null)	(null)	(null)	ENABLED

DB870 to CDB9 on reade | BEER_REVIEWS_P

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

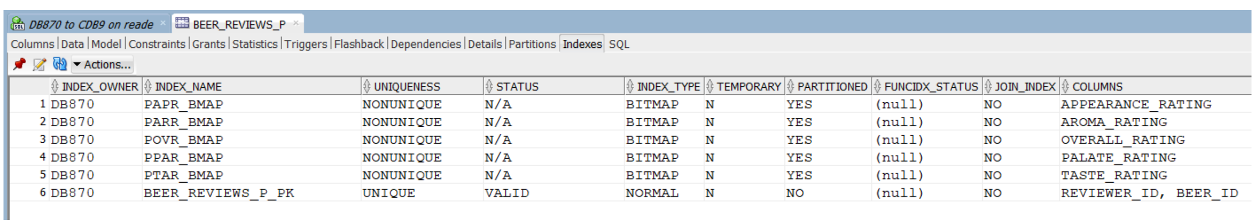
	INDEX_OWNER	INDEX_NAME	UNIQUENESS	STATUS	INDEX_TYPE	TEMPORARY	PARTITIONED	FUNCIDX_STATUS	JOIN_INDEX	COLUMNS
1	DB870	BEER_REVIEWS_P_PK	UNIQUE	VALID	NORMAL	N	NO	(null)	NO	REVIEWER_ID, BEER_ID

Refresh: 0

	INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	COLUMN_LENGTH	CHAR_LENGTH	DESCEND	COLUMN_EXPRESSION
1	DB870	BEER_REVIEWS_P_PK	DB870	BEER_REVIEWS_P	REVIEWER_ID	1	22	0	ASC	(null)
2	DB870	BEER_REVIEWS_P_PK	DB870	BEER_REVIEWS_P	BEER_ID	2	22	0	ASC	(null)

The only remaining step was to configure the bitmap indexes for the new table.

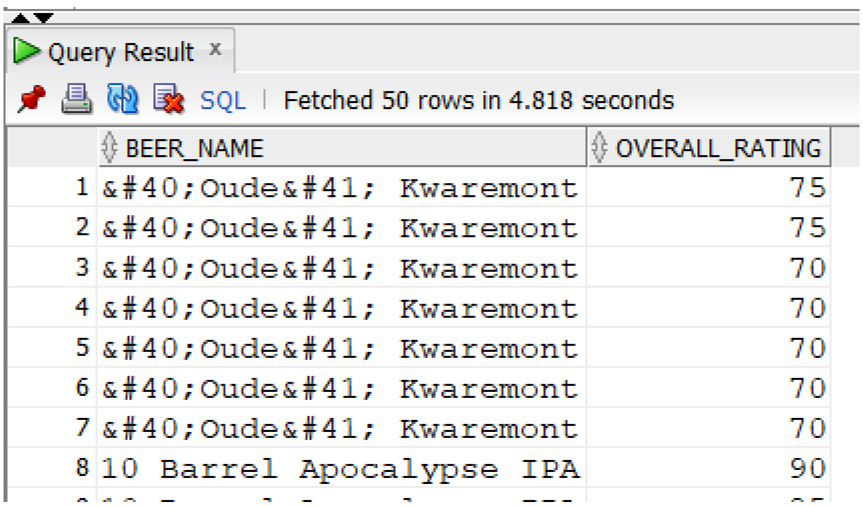
```
CREATE BITMAP INDEX
   povr_bmap
ON
   beer_reviews_p(overall_rating) LOCAL;

CREATE BITMAP INDEX
   papr_bmap
ON
   beer_reviews_p(appearance_rating) LOCAL;

CREATE BITMAP INDEX
   parr_bmap
ON
   beer_reviews_p(aroma_rating) LOCAL;

CREATE BITMAP INDEX
   ppar_bmap
ON
   beer_reviews_p(palate_rating) LOCAL;

CREATE BITMAP INDEX
   ptar_bmap
ON
   beer_reviews_p(taste_rating) LOCAL;

EXECUTE dbms_stats.gather_table_stats('DB870', 'beer_reviews_p');
```



	INDEX_OWNER	INDEX_NAME	UNIQUENESS	STATUS	INDEX_TYPE	TEMPORARY	PARTITIONED	FUNCIDX_STATUS	JOIN_INDEX	COLUMNS
1	DB870	PAPR_BMAP	NONUNIQUE	N/A	BITMAP	N	YES	(null)	NO	APPEARANCE_RATING
2	DB870	PARR_BMAP	NONUNIQUE	N/A	BITMAP	N	YES	(null)	NO	AROMA_RATING
3	DB870	POVR_BMAP	NONUNIQUE	N/A	BITMAP	N	YES	(null)	NO	OVERALL_RATING
4	DB870	PPAR_BMAP	NONUNIQUE	N/A	BITMAP	N	YES	(null)	NO	PALATE_RATING
5	DB870	PTAR_BMAP	NONUNIQUE	N/A	BITMAP	N	YES	(null)	NO	TASTE_RATING
6	DB870	BEER_REVIEWS_P_PK	UNIQUE	VALID	NORMAL	N	NO	(null)	NO	REVIEWER_ID, BEER_ID

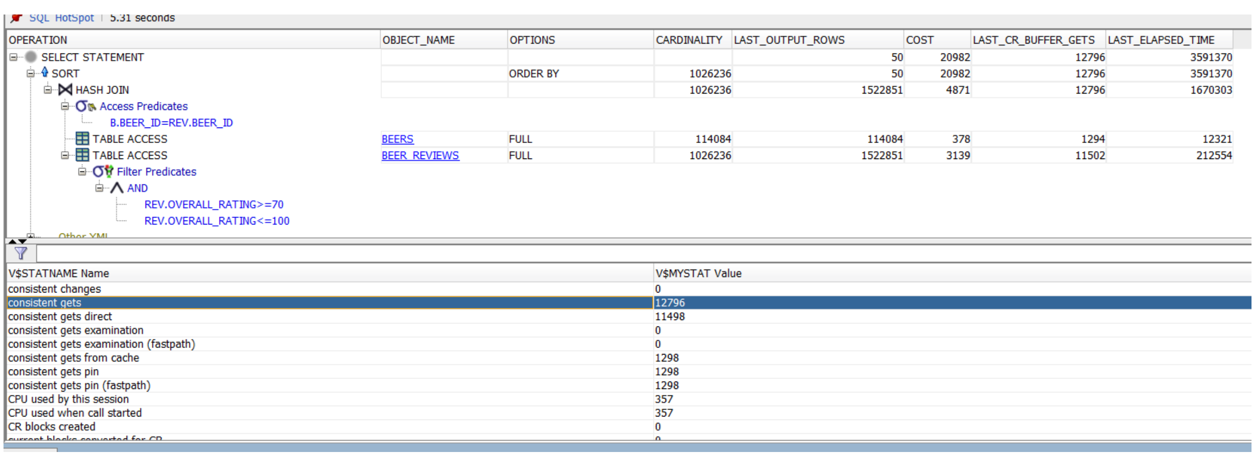
At this point, I gave some thought to previously planned next steps. In a production environment, it would make sense to backup and then drop (or simply rename) the old table and then rename the new table to assume the old table's name. However, this is a lab environment, and we have plenty of space since I deleted over half the rows in the comments table. So, I decided to keep the old table, and I left the comments table unchanged (i.e., still pointing to the beer_reviews table for its FK constraints). With this schema, we can run queries against the old and new review tables for comparison. So, that is what I did next.

SELECT
   b.beer_name,
   rev.overall_rating
FROM
   beers b
   INNER JOIN beer_reviews rev
      ON b.beer_id = rev.beer_id
WHERE
   rev.overall_rating BETWEEN 70 AND 100
ORDER BY
   b.beer_name ASC,
   rev.overall_rating DESC;



Query Result
SQL | Fetched 50 rows in 4.818 seconds

	BEER_NAME	OVERALL_RATING
1	(Oude) Kwaremont	75
2	(Oude) Kwaremont	75
3	(Oude) Kwaremont	70
4	(Oude) Kwaremont	70
5	(Oude) Kwaremont	70
6	(Oude) Kwaremont	70
7	(Oude) Kwaremont	70
8	10 Barrel Apocalypse IPA	90



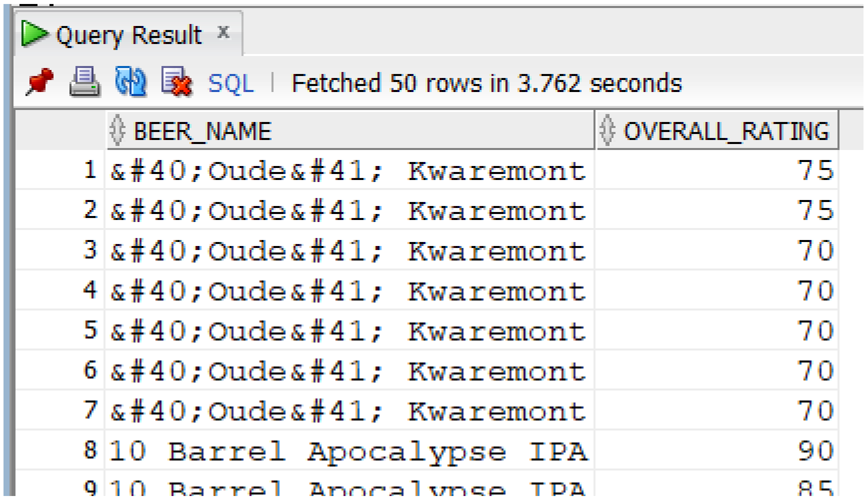
SQL HotSpot | 5.31 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_OUTPUT_ROWS	COST	LAST_CR_BUFFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT				50	20982	12796	3591370
SORT		ORDER BY	1026236	50	20982	12796	3591370
HASH JOIN			1026236	1522851	4871	12796	1670303
Access Predicates							
B.BEER_ID=REV.BEER_ID							
TABLE ACCESS	BEERS	FULL	114084	114084	378	1294	12321
TABLE ACCESS	BEER_REVIEWS	FULL	1026236	1522851	3139	11502	212554
Filter Predicates							
AND							
REV.OVERALL_RATING>=70							
REV.OVERALL_RATING<=100							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	12796
consistent gets direct	11498
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	1298
consistent gets pin	1298
consistent gets pin (fastpath)	1298
CPU used by this session	357
CPU used when call started	357
CR blocks created	0

SELECT
   b.beer_name,
   rev.overall_rating
FROM

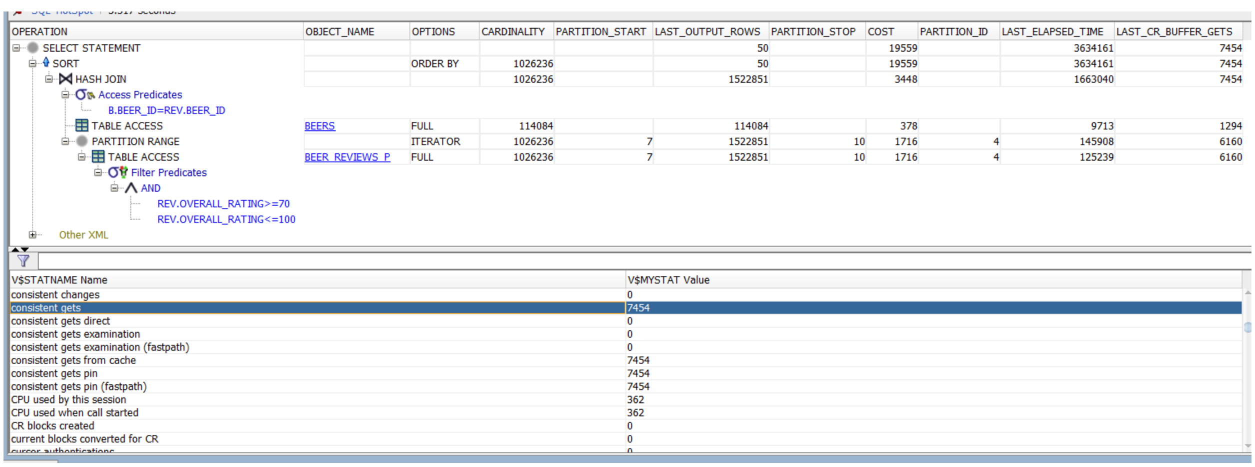
```
    beers b
    INNER JOIN beer_reviews_p rev
        ON b.beer_id = rev.beer_id
WHERE
    rev.overall_rating BETWEEN 70 AND 100
ORDER BY
    b.beer_name ASC,
    rev.overall_rating DESC;
```



Query Result

Fetched 50 rows in 3.762 seconds

	BEER_NAME	OVERALL_RATING
1	(Oude) Kwaremont	75
2	(Oude) Kwaremont	75
3	(Oude) Kwaremont	70
4	(Oude) Kwaremont	70
5	(Oude) Kwaremont	70
6	(Oude) Kwaremont	70
7	(Oude) Kwaremont	70
8	10 Barrel Apocalypse IPA	90
9	10 Barrel Apocalypse IPA	85



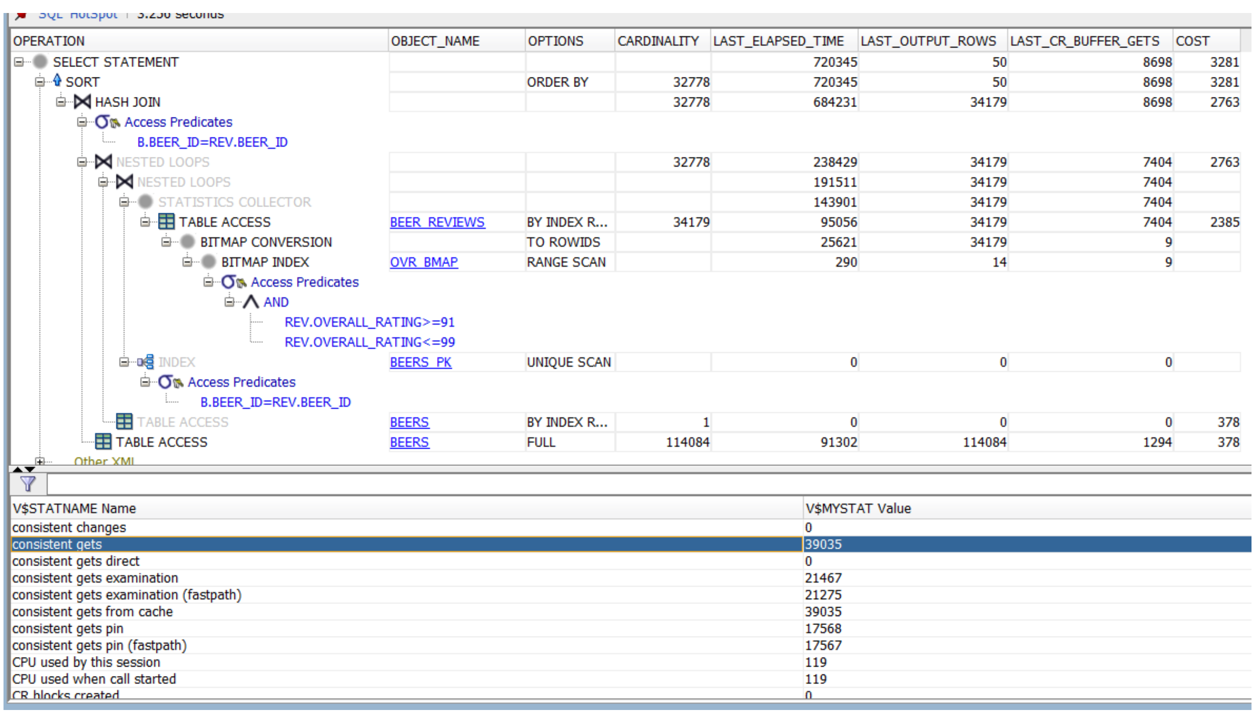
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	PARTITION_START	LAST_OUTPUT_ROWS	PARTITION_STOP	COST	PARTITION_ID	LAST_ELAPSED_TIME	LAST_CR_BUFFER_GETS
SELECT STATEMENT					50		19559		3634161	7454
SORT		ORDER BY	1026236		50		19559		3634161	7454
HASH JOIN			1026236		1522851		3448		1663040	7454
Access Predicates										
B.BEER_ID=REV.BEER_ID										
TABLE ACCESS	BEERS	FULL	114084		114084		378		9713	1294
PARTITION RANGE		ITERATOR	1026236	7	1522851	10	1716	4	145908	6160
TABLE ACCESS	BEER_REVIEWS_P	FULL	1026236	7	1522851	10	1716	4	125239	6160
Filter Predicates										
AND										
REV.OVERALL_RATING>=70										
REV.OVERALL_RATING<=100										
Other XML										

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	7454
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	7454
consistent gets pin	7454
consistent gets pin (fastpath)	7454
CPU used by this session	362
CPU used when call started	362
CR blocks created	0
current blocks converted for CR	0

As we can see, the execution plan for the partitioned table made use of the partitions. The partitioned table performed better with significantly fewer consistent_gets and a much faster time to return results (3.762 seconds versus 4.818 seconds). However, the last_elapsed_time was slightly higher for the partitioned table. According to Oracle documentation, that column represents the “Elapsed time (in microseconds) corresponding to this operation, during the last execution.” So, I am confused how it can be higher for the partitioned table. I decided to try a more selective query.

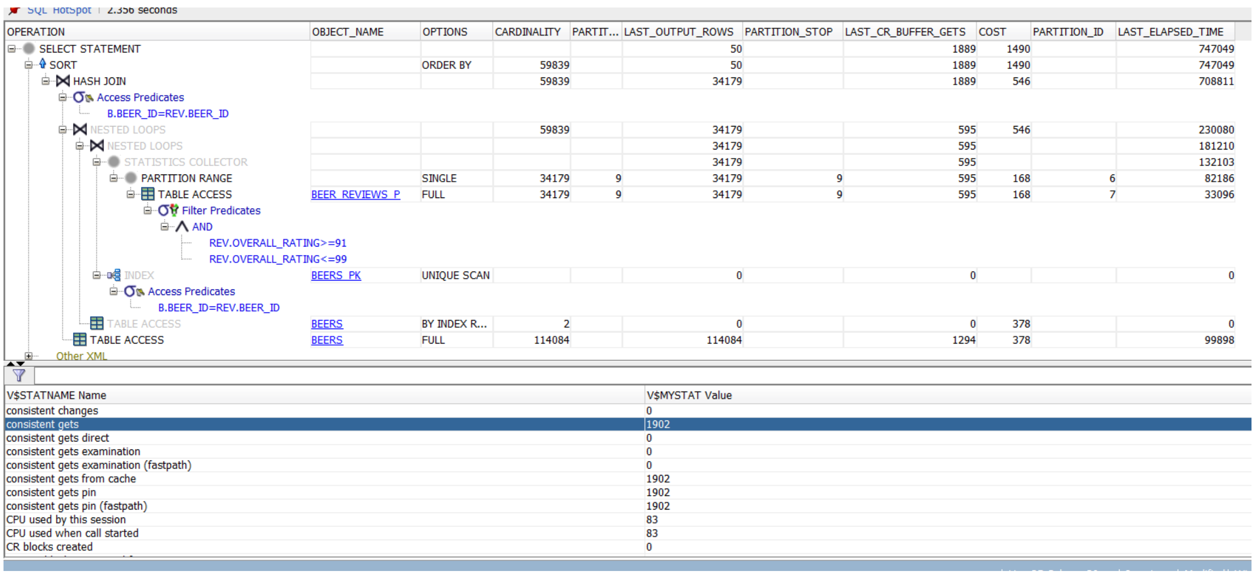
```sql
SELECT
   b.beer_name,
   rev.overall_rating
FROM
   beers b
   INNER JOIN beer_reviews rev
      ON b.beer_id = rev.beer_id
WHERE
   rev.overall_rating BETWEEN 91 AND 99
ORDER BY
   b.beer_name ASC,
   rev.overall_rating DESC;
```

```sql
SELECT
   b.beer_name,
   rev.overall_rating
FROM
   beers b
   INNER JOIN beer_reviews_p rev
      ON b.beer_id = rev.beer_id
WHERE
   rev.overall_rating BETWEEN 91 AND 99
ORDER BY
   b.beer_name ASC,
   rev.overall_rating DESC;
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	LAST_ELAPSED_TIME	LAST_OUTPUT_ROWS	LAST_CR_BUFFER_GETS	COST
SELECT STATEMENT				720345	50	8698	3281
SORT		ORDER BY	32778	720345	50	8698	3281
HASH JOIN			32778	684231	34179	8698	2763
Access Predicates							
B.BEER_ID=REV.BEER_ID							
NESTED LOOPS			32778	238429	34179	7404	2763
NESTED LOOPS				191511	34179	7404	
STATISTICS COLLECTOR				143901	34179	7404	
TABLE ACCESS	BEER_REVIEWS	BY INDEX R...	34179	95056	34179	7404	2385
BITMAP CONVERSION		TO ROWIDS		25621	34179	9	
BITMAP INDEX	OVR_BMAP	RANGE SCAN		290	14	9	
Access Predicates							
AND							
REV.OVERALL_RATING>=91							
REV.OVERALL_RATING<=99							
INDEX	BEERS_PK	UNIQUE SCAN		0	0	0	
Access Predicates							
B.BEER_ID=REV.BEER_ID							
TABLE ACCESS	BEERS	BY INDEX R...	1	0	0	0	378
TABLE ACCESS	BEERS	FULL	114084	91302	114084	1294	378
Other XML							

V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	39035
consistent gets direct	0
consistent gets examination	21467
consistent gets examination (fastpath)	21275
consistent gets from cache	39035
consistent gets pin	17568
consistent gets pin (fastpath)	17567
CPU used by this session	119
CPU used when call started	119
CR blocks created	0



SQL HotSpot | 2.356 seconds

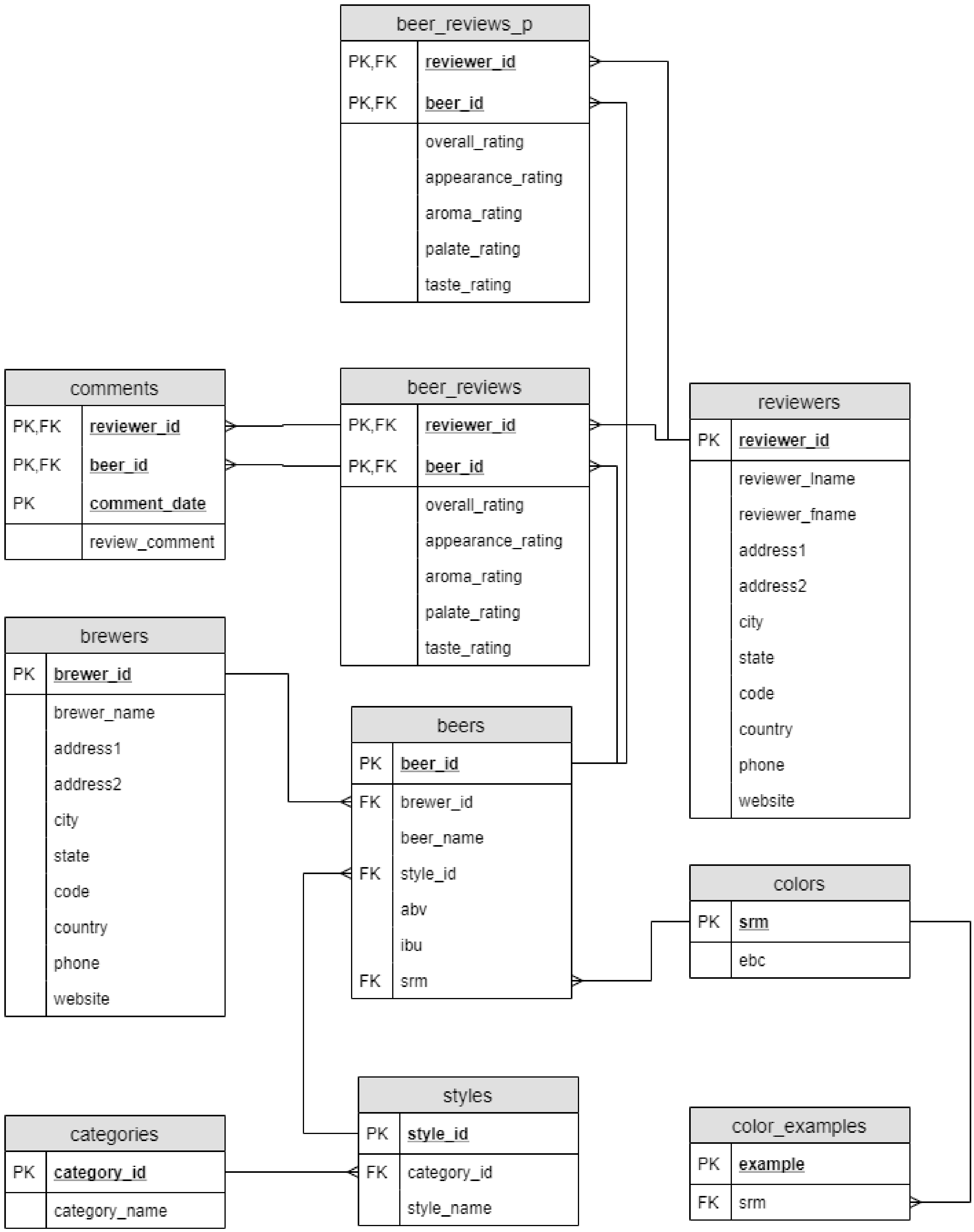
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	PARTIT...	LAST_OUTPUT_ROWS	PARTITION_STOP	LAST_CR_BUFFER_GETS	COST	PARTITION_ID	LAST_ELAPSED_TIME
SELECT STATEMENT					50		1889	1490		747049
SORT		ORDER BY	59839		50		1889	1490		747049
HASH JOIN			59839		34179		1889	546		708811
Access Predicates										
B.BEER_ID=REV.BEER_ID										
NESTED LOOPS			59839		34179		595	546		230080
NESTED LOOPS					34179		595			181210
STATISTICS COLLECTOR					34179		595			132103
PARTITION RANGE		SINGLE	34179	9	34179	9	595	168	6	82186
TABLE ACCESS	BEER_REVIEWS_P	FULL	34179	9	34179	9	595	168	7	33096
Filter Predicates										
AND										
REV.OVERALL_RATING>=91										
REV.OVERALL_RATING<=99										
INDEX	BEERS_PK	UNIQUE SCAN			0		0			0
Access Predicates										
B.BEER_ID=REV.BEER_ID										
TABLE ACCESS	BEERS	BY INDEX R...	2		0		0	378		0
TABLE ACCESS	BEERS	FULL	114084		114084		1294	378		99898
Other XML										

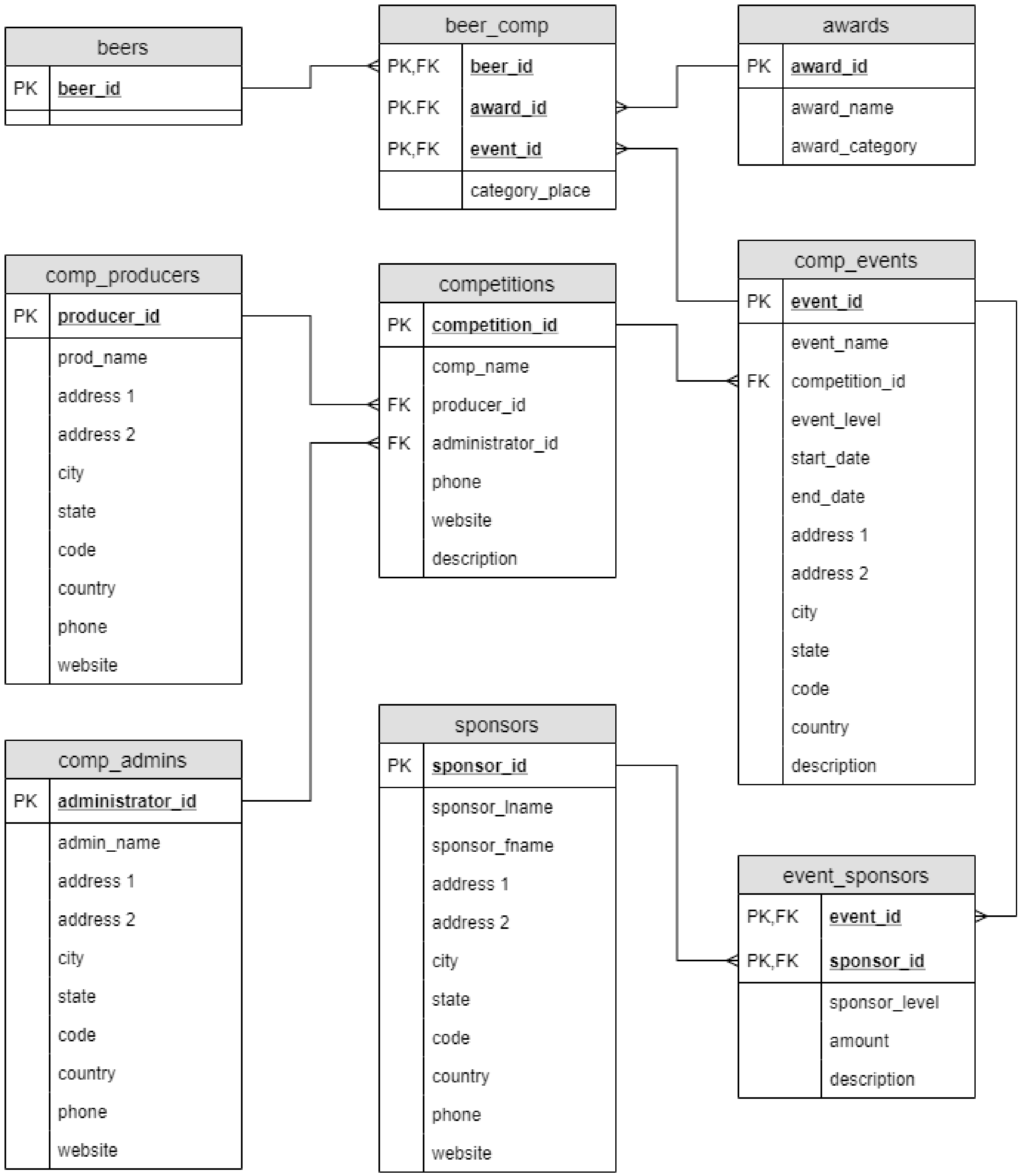
V$STATNAME Name	V$MYSTAT Value
consistent changes	0
consistent gets	1902
consistent gets direct	0
consistent gets examination	0
consistent gets examination (fastpath)	0
consistent gets from cache	1902
consistent gets pin	1902
consistent gets pin (fastpath)	1902
CPU used by this session	83
CPU used when call started	83
CR blocks created	0

I ran each query a dozen times. The first query averaged 0.088 seconds to return results. The second query (partitioned) averaged 0.07 seconds. That equates to a 20% performance improvement over the non-partitioned query. Again, we see the last_elapsed_time is higher for the partitioned table on the last run of the query. I did not average that statistic across all runs, so it might have been lower on average for the partitioned table. However, it seems strange that it would ever be higher for the partitioned table given the large performance advantage of the partitioned table in terms of time to return results to the client. In any event, I counted this as a win, which brought my score to two for four in Assignment 4.

Note the access path to the beer_reviews table was the ovr_bmap index, which significantly increased performance versus a FULL SCAN, thereby offsetting some of the partitioning advantage. The access path to the beer_reviews_p table was a FULL SCAN of partition p9 (owing to the relatively small size of p9 I suppose).

The updated ERD follows and shows the addition of the partitioned table.


beer_reviews_p
PK,FK reviewer_id
PK,FK beer_id
overall_rating
appearance_rating
aroma_rating
palate_rating
taste_rating
comments
PK,FK reviewer_id
PK,FK beer_id
PK comment_date
review_comment
beer_reviews
PK,FK reviewer_id
PK,FK beer_id
overall_rating
appearance_rating
aroma_rating
palate_rating
taste_rating
reviewers
PK reviewer_id
reviewer_lname
reviewer_fname
address1
address2
city
state
code
country
phone
website
brewers
PK brewer_id
brewer_name
address1
address2
city
state
code
country
phone
website
beers
PK beer_id
FK brewer_id
beer_name
FK style_id
abv
ibu
FK srm
colors
PK srm
ebc
styles
PK style_id
FK category_id
style_name
categories
PK category_id
category_name
color_examples
PK example
FK srm



The updated list of indexes follows.

- Indexes
  - ADMIN_FK_BTREE
  - APR_BMAP
  - ARR_BMAP
  - AWARDS_PK
  - BEER_COMP_PK
  - BEER_NAME_BTREE
  - BEER_REVIEWS_P_PK
  - BEER_REVIEWS_PK
  - BEERS_PK
  - BREWER_BEER_UNQ
  - BREWER_FK_BTREE
  - BREWER_NAME_BTREE
  - BREWERS_PK
  - CAT_STYLE_UNQ
  - CATEGORIES_PK
  - CE_CITY_STATE_BTREE
  - COLOR_EXAMPLES_PK
  - COLORS_PK
  - COMMENTS_PK
  - COMP_ADMINS_PK
  - COMP_EVENTS_PK
  - COMP_FK_BTREE
  - COMP_PRODUCERS_PK
  - COMPETITIONS_PK
  - EVENT_SPONSORS_PK
  - OVR_BMAP
  - PAPR_BMAP
  - PAR_BMAP
  - PARR_BMAP
  - POVR_BMAP
  - PPAR_BMAP
  - PROD_FK_BTREE
  - PROD_NAME_BTREE
  - PTAR_BMAP
  - REVIEWERS_PK
  - SPONSORS_PK
  - STYLE_FK_BTREE
  - STYLE_NAME_BTREE
  - STYLES_PK
  - TAR_BMAP
- Packages

Other Topics

Loading Bulk Data via Python

For my other topic, I chose to experiment with Python, cx_Oracle, and Oracle Instant Client to load the RateBeer dataset into the database. I used Spyder 3.3.6 and Python 3.7.4 (both installed by Anaconda3) with cx_Oracle 7.3.0 and Oracle Instant Client (Basic Light) 19.5.0.0.0. I had other options for loading the RateBeer dataset, but I saw this as an opportunity to improve my limited Python skills. Since I am new to coding, experienced coders will likely see many potential improvements in my code. Nonetheless, here it is.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 17 22:40:00 2020

@author: James Long
"""

import cx_Oracle
import re

# Connect as user "DB870" with password <redacted>
# running on <redacted>
conn = cx_Oracle.connect("DB870", <redacted>, <redacted>)
cur = conn.cursor()

#rbfh = open('ratebeer_sample.txt')    # test dataset, first 5 records
rbfh = open('ratebeer_db.txt')    # full dataset
#rbfh = open('ratebeer_db_trimmed.txt')    # partial dataset to speed disconnect recovery

# if a restart is needed due to disconnect before the script completes, you can
# comment the following lines down to (but not including) the error logging block
cur.execute('''
        SELECT
          beer_id
        FROM
          beers
        '''
        )
beer_id_lst = list(cur.fetchall())
cur.execute('''
        SELECT
          brewer_id,
          beer_name
        FROM
          beers
        '''
```

```
        )
beer_unq_lst = list(cur.fetchall())

cur.execute('''
        SELECT
          style_name,
          style_id
        FROM
          styles
        '''
        )
style_lst = list(cur.fetchall())
style_dict = {}
for tup in style_lst:
  style_dict[tup[0]] = tup[1]
cur.execute('''
        SELECT
          style_id
        FROM
          styles
        '''
        )
style_lst = list(cur.fetchall())
new_style_id = int(max(style_lst)[0])
del style_lst

cur.execute('''
        SELECT
          reviewer_lname,
          reviewer_id
        FROM
          reviewers
        '''
        )
reviewer_lst = list(cur.fetchall())
reviewer_dict = {}
for tup in reviewer_lst:
  reviewer_dict[tup[0]] = tup[1]
del reviewer_lst

cur.execute('''
        SELECT
          brewer_id
        FROM
          brewers
        '''
        )
brewer_lst = list(cur.fetchall())
```

```
cur.execute('''
        SELECT
          reviewer_id,
          beer_id,
          comment_date
        FROM
          comments
        '''
        )
comment_lst = list(cur.fetchall())

cur.execute('''
        SELECT
          reviewer_id,
          beer_id
        FROM
          beer_reviews
        '''
        )
beer_review_lst = list(cur.fetchall())

# error logging
reviewer_err_lst = list()
reviewer_err_cnt = 0
brewer_err_lst = list()
brewer_err_cnt = 0
style_err_lst = list()
style_err_cnt = 0
beer_err_lst = list()
beer_err_cnt = 0
beer_review_err_lst = list()
beer_review_err_cnt = 0
comment_err_lst = list()
comment_err_cnt = 0

print()
print('******* BEGIN *******')
print()

for line in rbfh:
  if len(line) == 1:
    continue
  line = line.encode('ascii','replace').decode('ascii')
  if line.startswith('?'):
    line = line.translate(line.maketrans('', '', '?'))
  if line.startswith('beer/name'):
    try:
```

```
            beer_name = str(re.findall('^beer/name: (.+)', line)[0])
            continue
        except:
            beer_name = 0
            continue
    elif line.startswith('beer/beerId'):
        try:
            beer_id = int(re.findall('^beer/beerId: (.+)', line)[0])
            continue
        except:
            beer_id = 0
            continue
    elif line.startswith('beer/brewerId'):
        try:
            brewer_id = int(re.findall('^beer/brewerId: (.+)', line)[0])
            continue
        except:
            brewer_id = -1
            continue
    elif line.startswith('beer/ABV'):
        try:
            abv = float(re.findall('^beer/ABV: (.+)', line)[0])
            continue
        except:
            abv = 0
            continue
    elif line.startswith('beer/style'):
        try:
            style_name = str(re.findall('^beer/style: (.+)', line)[0])
            continue
        except:
            style_name = 'unknown'
            continue
    elif line.startswith('review/appearance'):
        try:
            appearance_rating = str(re.findall('^review/appearance: (.+)', line)[0])
            appearance_rating = str(round((float(appearance_rating[0])/float(appearance_rating[2]))*100,0))
            continue
        except:
            appearance_rating = None
            continue
    elif line.startswith('review/aroma') and len(line) == 19:
        try:
            aroma_rating = str(re.findall('^review/aroma: (.+)', line)[0])
            aroma_rating = str(round((float(aroma_rating[0])/float(aroma_rating[2:]))*100,0))
            continue
        except:
            aroma_rating = None
```

```
            continue
        elif line.startswith('review/aroma') and len(line) == 20:
            try:
                aroma_rating = str(re.findall('^review/aroma: (.+)', line)[0])
                aroma_rating = str(round((float(aroma_rating[:2])/float(aroma_rating[3:]))*100,0))
                continue
            except:
                aroma_rating = None
                continue
        elif line.startswith('review/palate'):
            try:
                palate_rating = str(re.findall('^review/palate: (.+)', line)[0])
                palate_rating = str(round((float(palate_rating[0])/float(palate_rating[2]))*100,0))
                continue
            except:
                palate_rating = None
                continue
        elif line.startswith('review/taste') and len(line) == 19:
            try:
                taste_rating = str(re.findall('^review/taste: (.+)', line)[0])
                taste_rating = str(round((float(taste_rating[0])/float(taste_rating[2:]))*100,0))
                continue
            except:
                taste_rating = None
                continue
        elif line.startswith('review/taste') and len(line) == 20:
            try:
                taste_rating = str(re.findall('^review/taste: (.+)', line)[0])
                taste_rating = str(round((float(taste_rating[:2])/float(taste_rating[3:]))*100,0))
                continue
            except:
                taste_rating = None
                continue
        elif line.startswith('review/overall') and len(line) == 21:
            try:
                overall_rating = str(re.findall('^review/overall: (.+)', line)[0])
                overall_rating = str(round((float(overall_rating[0])/float(overall_rating[2:]))*100,0))
                continue
            except:
                overall_rating = -1
                continue
        elif line.startswith('review/overall') and len(line) == 22:
            try:
                overall_rating = str(re.findall('^review/overall: (.+)', line)[0])
                overall_rating = str(round((float(overall_rating[:2])/float(overall_rating[3:]))*100,0))
                continue
            except:
                overall_rating = -1
```

```
        continue
elif line.startswith('review/time'):
    try:
        comment_date = int(re.findall('^review/time: (.+)', line)[0])
        continue
    except:
        comment_date = -1
        continue
elif line.startswith('review/profileName'):
    try:
        reviewer_lname = str(re.findall('^review/profileName: (.+)', line)[0])
        continue
    except:
        reviewer_lname = 0
        continue
elif line.startswith('review/text'):
    try:
        review_comment = str(re.findall('^review/text: (.+)', line)[0])
    except:
        review_comment = None

if reviewer_lname != 0:
    reviewer_id = reviewer_dict.get(reviewer_lname,0)
    if reviewer_id == 0:
        try:
            cur.execute('''
                    INSERT INTO
                        reviewers(reviewer_lname)
                    VALUES
                        (:reviewer_lname)
                    ''',
                    [reviewer_lname]
                    )
            conn.commit();
            cur.execute('''
                    SELECT
                        reviewer_id
                    FROM
                        reviewers
                    WHERE
                        reviewer_lname = :reviewer_lname
                    ''',
                    [reviewer_lname]
                    )
            reviewer_dict[reviewer_lname] = cur.fetchone()[0]
            reviewer_id = reviewer_dict.get(reviewer_lname)
        except Exception as nam_err:
            reviewer_err_lst.append(reviewer_lname)
```

```
            reviewer_err_cnt = reviewer_err_cnt + 1
            print('reviewer error',reviewer_err_cnt)
            print(nam_err)

    if (brewer_id,) not in brewer_lst:
        try:
            cur.execute('''
                    INSERT INTO
                        brewers(brewer_id)
                    VALUES
                        (:brewer_id)
                    ''',
                    [brewer_id]
                    )
            conn.commit();
            brewer_lst.append((brewer_id,))
        except Exception as brew_err:
            brewer_err_lst.append(brewer_id)
            brewer_err_cnt = brewer_err_cnt + 1
            print('brewer error',brewer_err_cnt)
            print(brew_err)

    style_id = style_dict.get(style_name,0)
    if style_id == 0:
        new_style_id = new_style_id + 1
        style_id = new_style_id
        try:
            cur.execute('''
                    INSERT INTO
                        styles(style_id,category_id,style_name)
                    VALUES
                        (:style_id,-1,:style_name)
                    ''',
                    [style_id, style_name]
                    )
            conn.commit();
            style_dict[style_name] = style_id
        except Exception as sty_err:
            style_err_lst.append(style_id)
            style_err_cnt = style_err_cnt + 1
            print('style error',style_err_cnt)
            print(sty_err)

    if beer_name == 0:
        continue
    if beer_id == 0:
        continue
    if (brewer_id, beer_name) in beer_unq_lst:
```

```
        continue
    if (beer_id,) not in beer_id_lst:
        try:
            cur.execute('''
                    INSERT INTO
                        beers(beer_id, brewer_id, beer_name, style_id, abv, ibu, srm)
                    VALUES
                        (:beer_id, :brewer_id, :beer_name, :style_id, :abv, 0, -1)
                    ''',
                    [beer_id, brewer_id, beer_name, style_id, abv]
                    )
            conn.commit();
            beer_id_lst.append((beer_id,))
        except Exception as beer_err:
            beer_err_lst.append(beer_id)
            beer_err_cnt = beer_err_cnt + 1
            print('beer error',beer_err_cnt)
            print(beer_err)

    if reviewer_lname == 0:
        continue
    if overall_rating == -1:
        continue
    if (reviewer_id, beer_id) not in beer_review_lst:
        try:
            cur.execute('''
                    INSERT INTO
                        beer_reviews(reviewer_id, beer_id, overall_rating, appearance_rating, aroma_rating,
palate_rating, taste_rating)
                    VALUES
                        (:reviewer_id, :beer_id, :overall_rating, :appearance_rating, :aroma_rating,
:palate_rating, :taste_rating)
                    ''',
                    [reviewer_id, beer_id, overall_rating, appearance_rating, aroma_rating, palate_rating,
taste_rating]
                    )
            conn.commit();
            beer_review_lst.append((reviewer_id, beer_id))
        except Exception as rev_err:
            beer_review_err_lst.append((reviewer_id, beer_id))
            beer_review_err_cnt = beer_review_err_cnt + 1
            print('beer review error',beer_review_err_cnt)
            print(rev_err)

    if comment_date == -1:
        continue
    if (reviewer_id, beer_id, comment_date) not in comment_lst:
        try:
```

```
        len(review_comment)
        try:
          cur.execute('''
                  INSERT INTO
                    comments(reviewer_id, beer_id, comment_date, review_comment)
                  VALUES
                    (:reviewer_id, :beer_id, :comment_date, :review_comment)
                  ''',
                  [reviewer_id, beer_id, comment_date, review_comment]
                  )
          conn.commit();
          comment_lst.append((reviewer_id, beer_id, comment_date))
        except Exception as com_err:
          comment_err_lst.append((reviewer_id, beer_id, comment_date))
          comment_err_cnt = comment_err_cnt + 1
          print('comment error',comment_err_cnt)
          print(com_err)
      except:
        continue

conn.commit()
cur.close()
```

Before starting, I performed a full scan of the file using the following code.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 23 22:40:00 2020

@author: James Long
"""

import re

review_lst = list()
rec_cnt = 0
rev_err = 0
beer_err = 0

rbfh = open('RateBeer_db.txt')

for line in rbfh:
  if len(line) == 1:
    continue
  line = line.encode('ascii','replace').decode('ascii')
  if line.startswith('?'):
    line = line.translate(line.maketrans('', '', '?'))
```

```
    if line.startswith('beer/beerId'):
        try:
            beer_id = str(re.findall('^beer/beerId: (.+)', line)[0])
            continue
        except:
            beer_id = None
            beer_err = beer_err + 1
            continue
    elif line.startswith('review/profileName'):
        rec_cnt = rec_cnt + 1
        try:
            reviewer_lname = str(re.findall('^review/profileName: (.+)', line)[0])
            if (reviewer_lname, beer_id) not in review_lst:
                review_lst.append((reviewer_lname, beer_id))
            continue
        except:
            reviewer_lname = None
            rev_err = rev_err + 1
            continue

print()
print('unique reviews =',len(review_lst))
print('total records =',rec_cnt)
```

The scan counted 2,924,163 total records. That is 36 more than the 2,924,127 reviews reported by Stanford SNAP. The scan also counted 2,855,232 unique reviews. The lower unique review count indicates some reviewers reviewed the same beer more than once. Since the requirements were to store a single review per beer per reviewer, I did not insert duplicate reviews. (Doing so was not possible based on the composite key for the beer_reviews table.) In case of duplicate reviews, the first one encountered in the dataset was inserted. Without writing additional code to log these duplicate records and compare their timestamps, I cannot say whether the first, intermediate, or last review was inserted. This might be an interesting exercise for future students. The beer_err and rev_err counts were both zero, which indicated no records were missing beerId or profileName values, and no such values were corrupt.

I cached certain data from the database upon startup to reduce total run time by eliminating millions of roundtrips to the server. Caching also made resumption of new record insertion quicker whenever the script had to be restarted. As it happens, my Internet connection dropped several times while the script was running, and I had to restart the script. So, these code changes really saved a lot of time. However, once my script reached around 1.7 million inserted records, restarting took so long that my database connection timed out while my script was reading through the locally cached records to locate the first uninserted record. To get around this, I wrote the following code to trim the first x million records from the dataset file (where x was set for each restart as close as possible to the actual number of inserted rows in the beer_reviews table). I then used the trimmed file to resume record insertion.

```
# -*- coding: utf-8 -*-
"""
```

Created on Tue Mar 31 18:12:20 2020

@author: james
"""

```
oldfh = open('ratebeer_db.txt')

newfh = open('ratebeer_db_trimmed.txt', mode='w')

linecnt = 0

for line in oldfh:
    linecnt = linecnt + 1
    if linecnt <= 28000000:
        continue
    else:
        newfh.write(line)

newfh.close()
```

With my early code, the BREWER_BEER_UNQ constraint was violated 48 times during insertion of new records. Since that constraint is based on the (brewer_id, beer_name) tuple in the beers table, two possible explanations exist. First, if the records generating these errors were indeed duplicates in the RateBeer dataset, then each would have a beerId matching the beer_id of a previously inserted record. Thus, these errors could not have occurred due to the beer_id check condition already in my code at that time. However, it is possible that a data entry error occurred when the RateBeer dataset was created, and a new beerId was assigned to certain records that already existed, thereby creating duplicate records. If this were the case, the choice would come down to inserting such records into the database to capture as much data as possible versus dropping such records to preserve the data integrity of the database. Given the total number of records, I decided it was better to drop some data than to introduce unreliable or misleading data into the database.

The other possibility is that each record represents a distinct beer (as indicated by the unique beerId) that shares a common name with another distinct beer produced by the same brewer. Such beers could be distinguishable (for the benefit of consumers) by some other characteristic, such as the color of the can/bottle or a sub-name that was excluded from the RateBeer dataset. For example, Summertime Ale - Lemon and Summertime Ale - Lime might both have been entered as Summertime Ale. If this were the case, the choice would come down to modifying such records with distinguishing data prior to inserting them into the database versus dropping such records versus dropping the BREWER_BEER_UNQ constraint. Dropping the BREWER_BEER_UNQ constraint would permit other illegitimate cases of non-unique (brewer_id, beer_name) records to be inserted, so I decided against that option. And since I did not know the actual feature that distinguished these beers, I had no way to reliably modify the records (e.g., by appending the missing name data to the beer name). Thus, I decided to drop such records.

The following debug data were collected for the 48 errors in case future students want to investigate this further. The fact that each offending beerId occurs more than once suggests that the root cause was not entry of an erroneous beerId. Rather, the associated brewerId and/or name values are likely the cause.

Error Count	beerId
1	19336
2	19336
3	19336
4	19336
5	19336
6	19336
7	19336
8	19336
9	19336
10	19336
11	19336
12	19336
13	19336
14	19336
15	102105
16	102105
17	102105
18	102105
19	102105
20	102105
21	102105
22	102105
23	102105
24	102105
25	102105
26	102105
27	102105
28	102105
29	102105
30	102105
31	102105
32	102105
33	102105
34	102105
35	102105
36	102105
37	102105
38	102105
39	102105
40	102105
41	102105
42	102105

43	102105
44	102105
45	102105
46	102105
47	102105
48	102105

Once my code was able to insert several tens of thousands of records with no errors, I dropped all newly inserted records from the database and re-ran my script to ensure only clean data was inserted. The script ran for 8+ days excluding downtime due to disconnects. In total, the effort took three weeks. Unfortunately, I was unable to complete the load process because the STUDENTS tablespace filled up near the very end of my script.

ORA-01653: unable to extend table DB870.COMMENTS by 8192 in tablespace STUDENTS

The following table shows the number of records in each of the affected tables upon completion of the load process.

Table	Total Records	Pre-existing Records	New Records
beers	114,084	5,890	108,194
brewers	8,789	1,414	7,375
styles	226	141	85
reviewers	29,097	0	29,097
beer_reviews	2,805,537	0	2,805,537
comments	2,775,682	0	2,775,682

The total comment count is lower than the total review count because some reviews did not have a comment. While it was possible that some reviews had more than one comment, that turned out not to be the case as indicated by the following query, which returned zero rows.

```
SELECT
   c1.beer_id,
   c1.reviewer_id,
   c1.comment_date
FROM
   comments c1
   INNER JOIN comments c2
      ON c1.beer_id=c2.beer_id
      AND c1.reviewer_id=c2.reviewer_id
WHERE
   c1.comment_date<>c2.comment_date;
```

If the review/time values in the RateBeer dataset represented distinct days without a time stamp (i.e., date only), then my code was only able to capture multiple comments by a single reviewer about a single beer if the comments were made on separate days. Multiple comments submitted on the same day by the same reviewer about the same beer would have the same review/time value. Thus, my code would have inserted only the first of such comments encountered in the dataset. There is no way to know whether the inserted comment was the first, intermediate, and last comment made on that day about that beer by that reviewer. However, if the time stamps were also embedded in the review/time values, then same-day comments would have unique review/time values, and my code captured all comments by the same reviewer about the same beer.

I used the following SQL statement in Oracle SQL Developer to monitor the Python session.

```
SELECT
   program,
   event,
   p2text,
   wait_class,
   state,
   wait_time_micro,
   time_since_last_wait_micro
FROM
   v$session
WHERE
   username = 'DB870';
```