Problem Set 1 Solutions

1 Answers

All questions were worth 1 point.

1. Write a query (using the `SELECT` statement) that will compute times and ids when any sensor's light reading was above 550. Show both the query and the first few lines of the result.

```
SELECT result_time, nodeid
FROM expt_table
WHERE light > 550
```

```
        result_time         | nodeid | light
----------------------------+--------+-------
 2004-08-21 06:41:25.088768 |      2 |   555
 2004-08-21 06:41:54.299207 |      2 |   558
 2004-08-21 06:42:23.596589 |      2 |   558
 2004-08-21 06:42:52.294125 |      2 |   554
 2004-08-21 06:52:37.536493 |      2 |   554
 2004-08-21 06:53:06.853934 |      2 |   554
 2004-08-21 06:53:35.833396 |      2 |   558
 2004-08-21 06:54:05.198795 |      2 |   566
 2004-08-21 06:54:34.632188 |      2 |   572
 2004-08-21 06:55:04.167547 |      2 |   577
 2004-08-21 06:55:33.071017 |      2 |   580
 2004-08-21 06:56:01.919605 |      2 |   589
 2004-08-21 06:56:31.61888  |      2 |   595
 2004-08-21 06:57:01.299197 |      2 |   604
 2004-08-21 06:57:29.572791 |      2 |   618
 2004-08-21 06:57:59.416131 |      2 |   627
 2004-08-21 06:58:28.323607 |      2 |   631
 2004-08-21 06:58:57.629101 |      2 |   640
 2004-08-21 06:59:27.309336 |      2 |   645
```

2. Write a query that will compute the average light reading at sensor 1 between 6 PM and 9 PM (inclusive of 6:00:00 PM and 9:00:00 PM). Show the query and the result.

   A basic aggregate query. The "::" operator does typecasting in Postgres – here we use it to convert `result_time`, which includes a day and year, to a `time` type which is just the time of day.

```
SELECT AVG(light)
FROM expt_table
WHERE nodeid = 1 AND
result_time::time >= '6:00:00 PM' AND
result_time::time <= '9:00:00 PM'
```

```
        avg
---------------------
 165.7606060606060606
```

3. Write a single query that computes the average temperature and light reading at every sensor between 6 PM and 9 PM, but exclude any sensors whose maximum voltage was greater than 418 during that time period. Show both the query and the result.

   This query required a grouped-aggregate. The "HAVING" clause allows us to select only aggregate records that meet a specific condition. Note that in this case, no sensor had a voltage greater than 418; if you run the query until 9:59:00 PM, you would see that sensor 2 is excluded.

```
SELECT nodeid, AVG(light)
FROM expt_table
WHERE result_time::time >= '6:00:00 PM' AND
      result_time::time <= '9:00:00 PM'
GROUP BY nodeid
HAVING MAX(voltage) <= 418

 nodeid |         avg
--------+---------------------
      3 | 271.8764367816091954
      1 | 165.7606060606060606
      2 | 237.2383720930232558


SELECT nodeid, AVG(light)
FROM expt_table
WHERE result_time::time >= '6:00:00 PM' AND
      result_time::time <= '9:59:00 PM'
GROUP BY nodeid
HAVING MAX(voltage) <= 418

 nodeid |         avg
--------+---------------------
      3 | 274.4742489270386266
      1 | 155.5114155251141553
```

4. Write a query that computes the average calibrated temperature readings from sensor 2 during each hour, inclusive, between 6 PM and 9 PM (i.e., your answer should consist of 4 rows of calibrated temperatures.)

   A combination of a join between the `expt_table` and the `calib_temp` table and an aggregate. Note the confusing wording of the question – I intended orignally for the answer to consist of 4 rows representing 6-7 PM, 7-8 PM, 8-9 PM, and 9-10 PM, which is what I've shown here. Many of you provided and answer with just three rows, which was also fine.

```
SELECT EXTRACT('hour' FROM result_time) AS hour, AVG(calib) AS temp
FROM expt_table, calib_temp
WHERE raw=temp AND
      nodeid = 2 AND
      EXTRACT('hour' FROM result_time) BETWEEN 18 and 21
GROUP BY EXTRACT('hour' FROM result_time)
ORDER BY EXTRACT('hour' FROM result_time);

hour  |      hour |        temp
------+---------------------
   18 | 25.2436974789915966
   19 | 25.4324324324324324
   20 | 25.9912280701754386
   21 | 26.0000000000000000
```

5. Write a query that computes all the epochs during which the results from sensors 1 and 2 arrived more than 1 second apart. Show the query and the result. Note that you can use the difference (minus) operator on timestamps in Postgres, and that the string '1 second' refers to a period of 1 second.

   The easiest way to answer this query is with a "self-join" – a join between two instances of `expt_table`. The answer below includes both `result_time` and `epoch` columns so to verify that the answer is correct. Note that you had to find times where node 1's data preceded node 2's and cases where node 2's data preceded node 1's.

```
SELECT  e1.result_time,
e2.result_time,
e1.nodeid,
e2.nodeid,
e1.epoch,
e2.epoch
FROM expt_table AS e1, expt_table AS e2
WHERE ((e1.result_time - e2.result_time) > '1 seconds' OR
       (e1.result_time - e2.result_time) < '-1 second') AND
      e1.nodeid = 1 AND
      e2.nodeid = 2 AND
      e1.epoch = e2.epoch;
```

```
        result_time         |        result_time         | nodeid | nodeid | epoch | epoch
----------------------------+----------------------------+--------+--------+-------+-------
 2004-08-21 00:14:49.434821 | 2004-08-21 00:14:48.414879 |      1 |      2 |   667 |   667
 2004-08-21 01:19:15.565386 | 2004-08-21 01:19:10.017423 |      1 |      2 |   799 |   799
 2004-08-21 01:40:08.092629 | 2004-08-21 01:40:13.150497 |      1 |      2 |   842 |   842
 2004-08-21 05:08:47.252727 | 2004-08-21 05:08:52.365847 |      1 |      2 |  1270 |  1270
 2004-08-21 09:31:08.613878 | 2004-08-21 09:31:03.811451 |      1 |      2 |  1808 |  1808
 2004-08-21 11:59:16.512238 | 2004-08-21 11:59:20.991415 |      1 |      2 |  2112 |  2112
 2004-08-21 14:47:28.038216 | 2004-08-21 14:47:27.001914 |      1 |      2 |  2457 |  2457
 2004-08-21 15:11:20.605471 | 2004-08-21 15:11:26.253382 |      1 |      2 |  2506 |  2506
 2004-08-21 17:03:29.018229 | 2004-08-21 17:03:28.01174  |      1 |      2 |  2736 |  2736
 2004-08-21 22:17:01.519419 | 2004-08-21 22:16:56.49269  |      1 |      2 |  3379 |  3379
```

6. Write a query that determines epochs during which one or two of the sensors did not return results. Show your query and the first few results, sorted by epoch number. You may wish to use a nested query – that is, a SELECT statement within the FROM clause of another SELECT statement.

   This question didn't really require a nested query – it can be computed with a simple grouped aggregate.

```
SELECT epoch
FROM  expt_table
GROUP BY epoch
HAVING COUNT(*) < 3
ORDER BY epoch;
```

```
 epoch
-------
   639
   653
   683
   712
   715
   725
   727
   729
   732
   734
```

7. Write a query that produces a temperature reading for each of the three sensors during any epoch in which any sensor produced a reading. If a sensor is missing a value during a given epoch, your result should report the value of this sensor as the most recent previously reported value. If there is no such value (e.g., the first value for a particular sensor is missing), you should return the special value 'null'. You may wish to read about the CASE and OUTER JOIN SQL statements.

   This query is substantially more complicated than the others. The easiest way to answer it is to decompose it into parts and then combine those parts together. First, we can build a sub-result that contains a one entry for every sensor during every epoch when at least one sensor reported using an unrestricted join – in other words, a cross-product:

```
SELECT nodeid,epoch FROM
    (SELECT DISTINCT epoch FROM expt_table) AS es,
    (SELECT DISTINCT nodeid FROM expt_table) AS ns) AS e2;
```

   Then, we can combine this with the list of sensor readings, substituting `null`s for any pair of `nodeid`s and `epoch`s in the cross-product that doesn't also appear in the original list of readings. We use an outer join to do this:

```
SELECT e2.nodeid, e2.epoch,
FROM expt_table AS e1
FULL OUTER JOIN
  (SELECT nodeid,epoch FROM
    (SELECT DISTINCT epoch FROM expt_table) AS es,
    (SELECT DISTINCT nodeid FROM expt_table) AS ns) AS e2
  ON (e2.nodeid = e1.nodeid and e2.epoch = e1.epoch)
ORDER BY e2.epoch, e2.nodeid;
```

   Finally, we need to replace the `null`s in this list with the most recent non-`null` entry for this result. We can use the `CASE` statement to replace a particular value with the results of a subquery (alternatively, SQL provides the `COALESCE` statement for the special case of replacing a `null` value with some other result.)

   Putting it all together, we get:

```
SELECT e2.nodeid, e2.epoch,
 (CASE WHEN
    e1.temp IS null THEN
       (SELECT temp FROM expt_table
        WHERE nodeid = e2.nodeid AND
        epoch =
           (SELECT max(epoch) FROM expt_table
            WHERE epoch < e2.epoch AND
            nodeid = e2.nodeid)
       )
    ELSE
       e1.temp
   END)
FROM expt_table AS e1
FULL OUTER JOIN
  (SELECT nodeid,epoch FROM
    (SELECT DISTINCT epoch FROM expt_table) AS es,
    (SELECT DISTINCT nodeid FROM expt_table) AS ns) AS e2
  ON (e2.nodeid = e1.nodeid and e2.epoch = e1.epoch)
ORDER BY e2.epoch, e2.nodeid;

 nodeid | epoch | temp
--------+-------+------
      1 |   637 |  505
```

```
 2 |   637 |  512
 3 |   637 |  355
 1 |   638 |  505
 2 |   638 |  512
 3 |   638 |  355
 1 |   639 |  505
 2 |   639 |  512
 3 |   639 |  356
 1 |   640 |  505
 2 |   640 |  512
 3 |   640 |  356
 1 |   641 |  505
 2 |   641 |  512
 3 |   641 |  354
 1 |   642 |  505
 2 |   642 |  512
 3 |   642 |  352
 1 |   643 |  505
 2 |   643 |  512
 3 |   643 |  356
 1 |   644 |  505
 2 |   644 |  512
 3 |   644 |  353
 1 |   645 |  505
 2 |   645 |  512
 3 |   645 |  351
 1 |   646 |  505
 2 |   646 |  512
 3 |   646 |  350
```

8. Write a query that determines epochs during which all three sensors did not return any results. Note that this is a deceptively hard query to write – you may need to make some assumptions about the frequency of missing epochs.

   The trick here was to realize that there's no built-in way to get SQL to build a list of all numbers between 1 and n – if there were, we could simply take the difference between such a list over the range `min(epoch)` to `max(epoch)` and the actual list of reported epochs.

   Instead, I assumed that there were no missing gaps of size greater than 4, and then built a list of all epochs that appeared `UNION`ed with the list of all epochs that appeared minus one, `UNION`ed with the same list minus two, and so on up to four. Then, I removed the list of epochs that acutally appeared (using the `EXCEPT` statement) to get a list of all missing epochs.

```
SELECT epoch-i AS missing_epochs
FROM expt_table, (SELECT 1 AS i UNION
                  SELECT 2 AS i UNION
                  SELECT 3 AS i UNION
                  SELECT 4 AS i) AS t
WHERE epoch-i > (SELECT min(epoch) FROM expt_table)
EXCEPT
SELECT epoch FROM expt_table;

 missing_epochs
----------------
            655
            656
           1048
```

Alternatively, if you were willing to report the number of missing epochs, without specifically enumerating them, you could take the difference between the first missing epoch and the next available epoch.

```
SELECT  DISTINCT(epoch) as "Missed Epoch",
        epoch -
                (SELECT MAX(epoch)
                 FROM expt_table c
                 WHERE c.epoch < a.epoch) as "Number of Earlier Missing Epochs" -1
FROM (SELECT epoch-1 AS epoch
      FROM expt_table
      EXCEPT
      SELECT epoch
      FROM  expt_table) AS a
WHERE epoch > (SELECT min(epoch) from expt_table);

 Missed Epoch | Number of Earlier Missing Epochs
--------------+-----------------------------------
          656 |                                 1
         1048 |                                 0
```

Reporting answers in this way was suggested by Daniel Abadi.