Assignment 7: Database Design Patterns, Part 1

This week you will explore two basic design patterns used in relational database schema design. The first part focuses on how to manage user authentication without storing passwords in an insecure way. The second part is a brief tour of OLAP databases and star schemas.

A template archive is provided with this assignment, so that you can fill in the answers between the required annotations. When you complete the assignment, repackage this archive with the name **`cs121hw7-`*`username`***, and submit it on the course website.

Part A: Secure Password Storage (30 points)

It is unfortunately all too common for database schemas to include a severe design defect: storing user passwords in plain text in the database. This is a catastrophic flaw for several reasons. First, once the database is compromised, the attacker can use users' passwords to access the system through normal channels. Second, most users use the same passwords for many different systems; once a user's password is known, that password can be used to compromise other systems that the user has access to.

A simple solution is to apply a cryptographically secure hash function to passwords. Instead of storing the password itself, a *hash* of the password is stored. Authentication simply involves applying the hash to the entered password, and then comparing the hash values. Unfortunately, this approach is susceptible to *dictionary attacks*; users are very predictable, and by computing the hash codes of the most common passwords (e.g. "abc123", "password", and "123456" are in the top 10 most common passwords), an attacker with access to the database can simply compare the stored hash codes to a dictionary of hash codes computed from common passwords. This won't compromise all user accounts, but almost always it will compromise at least a few accounts.

To prevent dictionary attacks, one can prepend a *salt* value to the password before applying the hash function; if the salt is large enough then the potential for dictionary attacks is eliminated. Of course, there is still the issue of users picking bad passwords, but several techniques can prevent brute-force attacks, such as imposing a time delay on authentication failure, and enforcing strict policies on password choices such as expiring passwords after a period of time, and disallowing users from using any common password.

For this section, you will implement a secure password storage mechanism using both hashing and salting. Note that such functionality is normally implemented in the application sitting on top of the database, or even in the web client. You, however, will be implementing these operations as stored routines in the database, simply to avoid the complexity of having to interface with the database from another programming language.

(The reason for implementing these operations in other parts of the system is to limit where the plaintext password is actually visible. For example, a more secure authentication mechanism for a client-server system would be for the server to pass the salt to the client once the username is known; then the client can salt and hash the password and send it back to the server. The password itself is simply never communicated from the client. Other similar mechanisms exist as well.)

All of your work for this problem will be in the `passwords.sql` file. Include your completed version of this file in your submission. Additionally, a helper function called **`make_salt`** is

provided in **make-salt.sql**, which is able to generate a random salt of up to 20 characters. You can use it like this:

```
SELECT make_salt(10) AS salt;
```

Scoring: Part 1 is 4pts, parts 2 and 3 are 8pts, part 4 is 10pts. Total is 30 points.

1a) Create a table called **user_info** that will store the data for this password mechanism. There are three string values the table needs to store:

- Usernames, in a column named **username**, will be up to 20 characters long
- Salt values, in a column named **salt**
- The hashed value of the password, called **password_hash**

We will be using the SHA-2 function to generate the cryptographic hash. (MySQL also has the MD5 and SHA-1 hash functions, but both of these are now considered too weak to be secure.) You can try out the SHA-2 hash function in MySQL like this:

```
SELECT SHA2('letmein', 256);
```

This function can generate hashes that are 228 bits, 256 bits, 384 bits, or 512 bits. We will use 256-bit hashes. Note that the hash is returned as a sequence of hexadecimal characters, so you will need to choose a suitable string length to store the hexadecimal version of the 256-bit hash. Make your **password_hash** column a fixed-size column that is exactly the right size, no larger and no smaller.

You should use a salt of at least 6 characters.

1b) Create a stored procedure **sp_add_user(new_username, password)**. This procedure is very simple:

- Generate a new salt.
- Add a new record to your **user_info** table with the username, salt, and salted password.

Make sure that you prepend the salt to the password before hashing the string. You can use the MySQL **CONCAT(s1, s2, ...)** function for this.

Don't worry about handling the situation where the username is already taken; assume that the application has already verified whether the username is available.

1c) Create a stored procedure **sp_change_password(username, new_password)**. This procedure is virtually identical to the previous procedure, except that an existing user record will be updated, rather than adding a new record.

Make sure to generate a new salt value in this function!

1d) Create a function (not a procedure) called **authenticate(username, password)**, which returns a **BOOLEAN** value of **TRUE** or **FALSE**, based on whether a valid username and password have been provided. The function should return **TRUE** iff:

- The username actually appears in the **user_info** table, and
- When the specified password is salted and hashed, the resulting hash matches the hash stored in the database.

If the username is not present, or the hashed password doesn't match the value stored in the database, authentication fails and **`FALSE`** should be returned.

Note that we don't distinguish between an invalid username and an invalid password; again, this is to keep attackers from identifying valid usernames from outside the system.

Once you have completed these operations, make sure you test them to verify that they are working properly. For example:

```
CALL sp_add_user('alice', 'hello');
CALL sp_add_user('bob', 'goodbye');

SELECT authenticate('carl', 'hello');       -- Should return 0/FALSE
SELECT authenticate('alice', 'goodbye');    -- Should return 0/FALSE
SELECT authenticate('alice', 'hello');      -- Should return 1/TRUE
SELECT authenticate('bob', 'goodbye');      -- Should return 1/TRUE

CALL sp_change_password('alice', 'greetings');

SELECT authenticate('alice', 'hello');      -- Should return 0/FALSE
SELECT authenticate('alice', 'greetings');  -- Should return 1/TRUE
SELECT authenticate('bob', 'greetings');    -- Should return 0/FALSE
```

Part B: OLAP and Star Schemas

A star schema follows a completely different pattern from the database schemas we have worked with so far. A star schema is *very* denormalized, in order to maximize the performance of queries over very large amounts of data.

A star schema consists of several "dimension tables," and a very small number of "fact tables." Fact tables contain summary information that we would like to compute from our data warehouse, but aggregated over the smallest resolutions that we are interested in supporting. The dimension tables, as the name would indicate, specify the dimensions that our queries will vary over, when generating results from our fact tables.

The data we have to work with for this assignment is web log data from NASA web servers from 1995.[1] Raw web-log data is very expensive to compute results from, so we will design a simple reporting schema for this data so that we can generate reports over various time intervals. This dataset is *huge* compared to what we have worked with so far; it's about 3.5 million records. However, this is still pretty tiny compared to many databases in everyday use. The entire data set is around 360MB, which completely fits into the main memory of most computers.

Step 0: Getting Situated

For this assignment, you will be creating a sequence of operations to convert the raw log file into a dimensional database. This includes operations like creating the star schema, populating dimension tables and fact tables from the raw log data, and so forth. Since you probably won't get it exactly right the very first time, it is recommended that you work with a "processing script file" that you can run multiple times against your database. This can be especially useful if you drop and recreate the fact and dimension tables at the beginning of your script, so that the auto-increment columns are reset each time you try to do another test run.

[1] http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

You shouldn't drop or reload the raw log table unless you absolutely have to, since the data takes a lot of time to load. So, don't put those operations into your processing script, but do include them in your final homework submission.

Step 1: Load the Raw Log Data

Before you can load the log data, you will need a table to load it into. You can download the file **`make-weblog.sql`** and use it to create the schema for the raw logs to go into. Note that the table is named **`raw_web_log`**; this will be important.

Data is normally bulk-loaded straight from the filesystem on the server where the database is running, since this provides the fastest import path into the database. However, if you are using the shared database server on **`checksum.cms`** then you will have to use the **`mysqlimport`** utility from a remote computer. This utility is available on the CS cluster.

The **`mysqlimport`** utility can be invoked in a very similar way to the MySQL command-line client:

```
mysqlimport -h checksum.cms.caltech.edu -u username -p \
      --delete --local username_db datafile1 ...
```

The details are as follows:

- The **`-h`** argument specifies the host to connect to, as before.
- The **`-u`** and **`-p`** arguments specify your username, and tell **`mysqlimport`** to ask you for your password.
- The **`--delete`** argument tells **`mysqlimport`** to delete the existing contents of the table before loading data into it. (This is appropriate for our needs in this lab, but it isn't always appropriate, of course.)
- The **`--local`** argument tells **`mysqlimport`** to load the data file from the client's local filesystem, rather from the server's filesystem.
- The ***`username`*`_db`** argument is the name of your database.

Then, one or more data files are listed. The important thing to note is that the data file's name dictates what table the data is loaded into! The **`mysqlimport`** utility will chop off all extensions, and use the remaining filename as the name of the table to load. For example, if your filename is "**`loan.dat`**", the utility would load this data into the "**`loan`**" table.

The log data for this database is on the CS cluster at this path:

```
/cs/courses/cs121/raw_web_log.dat
```

(If you don't have a CS cluster account, or you want to load the data from your own machine, the data is at this URL: http://users.cms.caltech.edu/~donnie/dbcourse/raw_web_log.dat)

All told, the import process should take around 30 seconds to complete, if you are loading this from the CS cluster. (If you are loading the data over a slower connection, e.g. a wireless connection, the load may go significantly slower.) If you have to load the data multiple times, and the **`raw_web_log`** table has to be cleaned out first, this will increase the total load time by a few seconds.

Step 2: Getting To Know The Data *(part a: 1pt; parts b-d: 3pts each; total is 10pts)*

Once you have successfully loaded the database, write and run these queries. **Put your answers for this section into `rawqueries.sql`.**

(For the sake of your sanity, the queries for parts a-c should each complete in under 2 minutes, but the solution for part d may go over that time by a bit. There are no indexes on this table…)

2a) Return a count of the total number of rows in the **`raw_web_log`** table.

2b) For each IP address in the log data, compute the total number of requests from that IP address. Your query should return the top 20 IP addresses along with how many requests each one made. Order the results in descending order of request-counts. Give the request-count a meaningful name.

2c) Write a query that computes, for each resource:

- The total number of requests for that resource. (This will simply be the number of log records that specify the resource.)
- The total number of bytes served for that resource. (This is in the **`bytes_sent`** column.)

Your query should return the top 20 resources, ordered by decreasing "total bytes served," and should include the resource, the total requests for that resource, and the total bytes served for that resource. Make sure to give all columns meaningful names. (Hint: You should be able to write this as a single **`SELECT`** statement. It is perfectly fine to compute several different aggregate results in a single **`SELECT`**.)

2d) Web server logs record individual requests for resources, but we would also like to know how many "visits" we get to our website. Two web-server requests are part of the same "visit" if they are from the same client (i.e. same **`ip_addr`** value), and if the request times are within 20 minutes of each other. Correspondingly, if there are two requests from the same client, and they are more than 20 minutes apart, they are considered to be two separate visits to the website. (20 minutes is a bit of an arbitrary number. Probably the only meaningful values would be somewhere between 15 minutes and 1 hour.)

It turns out to be prohibitively expensive to have a simple relational database figure out what log records correspond to what "visits" (it requires a self-join with multiple inequality conditions, much like our "dense rank" query), so the log data has been annotated with a "**`visit_val`**" value. Multiple requests are part of the same visit if they have the same value for the **`visit_val`** column. In addition, all **`visit_val`** values in our log data are unique; if the values are different then the records correspond to different visits.

Write a query that computes these values for each visit:

- The total number of requests made during that visit.
- The starting time of the visit (this will be the minimum request-time).
- The ending time of the visit (this will be the maximum request-time).

Your result should include five columns:

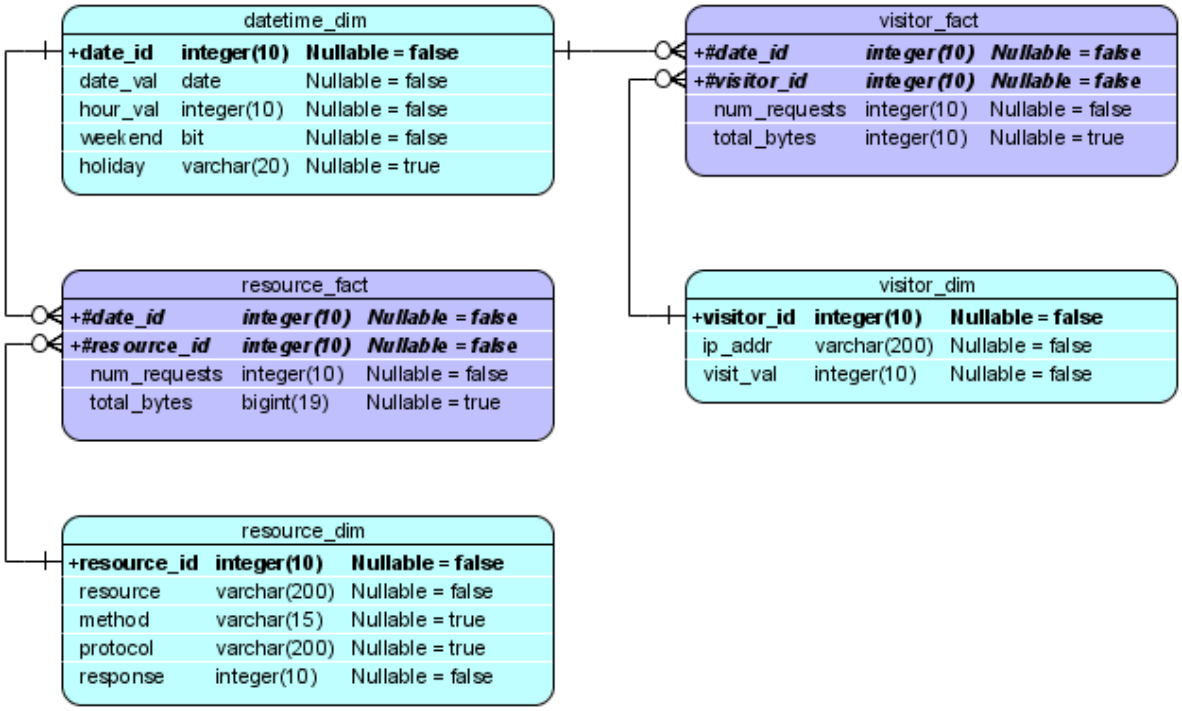
- The visit value
- The IP address of the requester
- The total number of requests in the visit
- The starting time of the visit
- The ending time of the visit

Order your results by decreasing "total requests," and include only the top 20 results. Give all columns meaningful names.

Step 3: Create a Star Schema *(6 points)*

Complete this step and the next step in a file ***`make-warehouse.sql`****.*

As mentioned before, OLAP database schemas typically follow a "star schema" pattern where each fact table references a number of dimension tables. There is no requirement that there be only one fact table; for our schema there will be two fact tables. (More on this in a bit.) Here is a diagram for the schema you must create. The dimension tables are light blue and the fact tables are violet.



Write SQL DDL to create these tables in a file named `make-warehouse.sql`. For the dimension tables, *make sure* to use auto-incrementing values for the ID columns. For the fact tables, *don't* use auto-incrementing values for the fact tables, since those ID values are populated from the dimension tables. Also, make sure to follow all of the "nullable" constraints specified above.

Even though the diagram indicates otherwise, specify column types in upper case, as usual. For the "integer" and "bigint" columns, just say "**`INTEGER`**" or "**`BIGINT`**"; ignore the size-specifications since the database already knows the width for these types. Use "**`BOOLEAN`**" for the "bit" column.

There are some additional constraints that you should specify, to make sure your results are correct. Sometimes you don't want these constraints since they would slow down your database, but our database is pretty small, and MySQL will use these constraints to build helpful indexes.

- All dimension tables have additional unique constraints:
  - (*date_val*, *hour_val*) → *datetime_dim*
  - (*resource*, *method*, *protocol*, *response*) → *resource_dim*

  - (*visit_val*) → *visitor_dim*
- The fact tables have foreign-key references to the dimension tables. The dimension tables should have no foreign keys.
- For the fact tables, you have these primary keys:
  - (*date_id*, *resource_id*) → *resource_fact*
  - (*date_id*, *visitor_id*) → *visitor_fact*

Once you have written your DDL, load it into your database so that you can begin to process the raw logs.

Step 4: Populate the Dimension Tables *(parts a-b are 6pts each, part c is 6pts; 12pts total)*

Complete this step and the next step in a file ***`populate.sql`****.*

Remember that you gave each dimension table a primary-key column that is an auto-incrementing integer; here is where you will put this to use. Complete these two steps to populate your dimension tables. Include in your answer all of the SQL necessary to perform the steps, along with brief documentation to describe what is going on.

The **`resource_dim`** and **`visitor_dim`** tables will be populated from the raw log data itself. For each of these dimension tables, write an **`INSERT...SELECT`** statement that populates the non-primary-key columns with only distinct combinations of values that appear in the raw web logs. For example, with the **`visitor_dim`** table, you will insert all distinct combinations of **`ip_addr`** and **`visit_val`** that appear in the raw web logs. Your **`INSERT...SELECT`** should not specify the primary-key column in the target table; let the database auto-generate that value for you.

4a) Populate **`resource_dim`**.

4b) Populate **`visitor_dim`**.

(Populating ***`resource_dim`*** *takes around 35 seconds, and* ***`visitor_dim`*** *takes around 3 minutes.)*

4c) The **`datetime_dim`** table is a bit more complicated than the previous two tables. Date and time dimension-values are usually not taken from raw data, because you may want to generate a report covering a time interval for which there is no data. Instead, we will populate the date-time dimension table with a SQL stored procedure.

(Note: Dates and times are normally maintained in separate dimension tables, but since our data warehouse is so small, we keep them together in one table.)

Here is a simple skeleton for you to use:

```
DELIMITER !

-- TODO: Write a comment describing the function's purpose.
CREATE PROCEDURE populate_dates(d_start DATE, d_end DATE)
BEGIN
    -- TODO: You fill in the body...
END !

DELIMITER ;
```

The procedure should implement the following logic:

let *d* = a DATE variable
let *h* = an INTEGER variable

Delete all rows whose *date_val* is between *d_start* and *d_end*.

-- There is no for-loop construct in MySQL.
-- In general, the FOR construct is for cursors anyway.

d = *d_start*
while *d* <= *d_end* **do**:
    h = 0
    while *h* <= 23 **do**:
        -- Again, don't assign to the primary key column;
        -- let the database generate that ID automatically.
        Add a row to *datetime_dim* that contains:
            date_val = *d*, *hour_val* = *h*,
            weekend = is_weekend(*d*),
            holiday = is_holiday(*d*)

        h = *h* + 1
    done

    d = *d* + INTERVAL 1 DAY
done

Make sure to write a comment describing the function.

Once you have this stored procedure working, populate your date/time dimension table with the function. We only have two months of data, so you should invoke it like this:

```
CALL populate_dates('1995-07-01', '1995-08-31');
```

You may notice that this stored procedure is a bit slow, on the order of 2-3 minutes. (This is why we only generate two months of dimension records.) This kind of operation is usually done *very* infrequently; a company might generate 5 years of date-dimension records at a time.

Make sure to check your **`datetime_dim`** table's values for correctness!

(If you don't trust your User-Defined Functions from Homework 4, you can use the versions from the HW4 solution-set, provided in the **`date-udfs.sql`** file with this assignment. Just drop the UDF definitions into your script file, and *make sure you note their source*. Always attribute your sources…)

Step 5: Populate the Fact Tables *(10 points)*

Complete the previous step and this step in a file ***`populate.sql`***.

Now that your dimension tables are properly populated, you can write the queries to fill in the fact tables. There are only two fact tables to fill in, **`resource_fact`** and **`visitor_fact`**.

- In the **`populate.sql`** file, annotate the SQL for populating **`resource_fact`** as Problem 5a, and the SQL for populating **`visitor_fact`** as Problem 5b.

You will notice that for each of these tables, some of the columns are ID columns that reference dimension tables, and the rest of the columns are "measures" (a.k.a. "facts") computed over each unique combination of dimension values.

To populate the fact tables, your queries will have a structure something like this:

```
INSERT INTO fact_table (
    dim1_id, dim2_id, ...,
    fact1, fact2, ...
)
SELECT dim1_id, dim2_id, ...,
       fact1_expr, fact2_expr, ...
FROM raw_data JOIN dim1 ON ... JOIN dim2 ON ...
GROUP BY dim1_id, dim2_id, ... ;
```

In other words, your query will:

1. Join the dimension tables against the fact table to bring in the dimension ID values
2. Group the raw data on all of the dimension ID values
3. Compute your facts over each group of raw rows
4. Insert the dimension IDs and the computed facts into the fact table.

You can use the **`DATE()`** function on **`raw_web_log.logtime`** to get the date portion out of the timestamp, and you can use the **`HOUR()`** function on **`logtime`** to get the hour value out of the timestamp.

Important Note 1: Include all appropriate dimension columns in your joins!

> If you accidentally leave out some of the dimension columns in your joins, the result will be to **multiply** the number of rows generated into the fact tables. For example, if you happen to join on date values but forget to also join on hour values, you will **multiply** the number of rows in your results by 24. And you will multiply the amount of time you have to wait by at least 24.
>
> This is not good. Take it from one who has made this mistake. ☺
>
> To prevent runaway queries, see **Important Note 3** below.

Important Note 2: Take `NULL` values into account!

> The dimension tables contain **`NULL`** values in several columns; for example, **`request_dim.protocol`** contains several **`NULL`** values. If you use a **`NATURAL JOIN`** in the above queries, all rows with **`NULL`** dimension-values will be excluded from the fact tables, which will give you incorrect answers.
>
> So, use a **`JOIN ... ON`** clause, and use the **`NULL`**-safe equality operator **`<=>`**. This is a MySQL extension, but a very helpful one for data warehousing. Unlike the normal equals operator, **`NULL <=> NULL`** evaluates to **`TRUE`**. This allows us to treat **`NULL`**s as "just another value" when performing our aggregation into the fact tables.

Important Note 3: During testing, prevent runaway queries!

As you can see from the below figures, none of these queries should take more than 3 or 4 minutes to run. If you are testing your script from the MySQL command-line, it is pretty good about stopping right away if you hit Ctrl-C.

However, if you are using the MySQL Query Browser, it may be a little less responsive. Thus, while you are testing your queries, you can tell the MySQL server to kill the operation if nothing happens for a certain amount of time. During testing, use the **`interactive_timeout`** and **`wait_timeout`** system-variables:

```
SET interactive_timeout = 300;
SET wait_timeout = 300;
```

Then, try out your queries. If the query is broken, the database will come back with an error after 300 seconds (5 minutes) has elapsed. (You can adjust this timeout based on your preferences, but don't set it to be *too* short.)

Finally, here are some sanity-check figures for you to look at:

Populating **`resource_fact`** with the full dataset:

- Takes around 2 minute 30 seconds to populate.
- Inserts 788,524 records.
- If you run this query:

```
SELECT date_id, COUNT(*) AS c FROM resource_fact
GROUP BY date_id ORDER BY c DESC LIMIT 3;
```

Assuming that you generate the same ID values, you should get { (300, 1218), (111, 1179), (304, 1167) }. If you have different ID values, the second number in each pair should at least be the same.

Populating **`visitor_fact`** with the full dataset:

- Takes around 2 minutes to populate.
- Inserts 353,020 records.
- If you run this query:

```
SELECT date_id, COUNT(*) AS c FROM visitor_fact
GROUP BY date_id ORDER BY c DESC LIMIT 3;
```

Again, assuming that you generate the same ID values, you should get { (298, 1173), (299, 1021), (300, 893) }.

If this is what you see, congratulations!

Step 6: Write Some OLAP Queries *(32 points total)*

Write all queries in a file ***`reporting.sql`****.*

Now that your OLAP database is all ready to go, it's time to try some queries. **Make sure to give all aggregate values a descriptive name in your query results.** You will lose points if you fail to do this.

Note: For your sanity, approximate run-times are included for the long-running queries. Other queries should complete in under 5 seconds.

Some easier queries (4 points per query):

6a) Write a query that reports each distinct HTTP protocol value, along with the total number of requests using that protocol value. Order your results by descending "total requests" count, and only include the top 10 results. *(Solution's query takes about 15 seconds to execute.)*

(HTTP/1.1 doesn't appear because it was introduced in 1999. Also, you will notice some really wonky HTTP protocol values – web browsers were pretty buggy back in 1995, since this whole "World Wide Web" thing was pretty new!)

6b) Report the top 20 (resource, error response) combinations. HTTP response values of 400 or higher indicate an error of some kind. The result should contain the resource, the response code, and the count of errors for that combination. Order the result by decreasing error count.

6c) Find the top 20 clients based on total bytes sent to each client. Your result should include:
- the IP address of the client
- the number of visits the client made (i.e. number of distinct visit_val values for the client)
- the total requests made by the client
- the total bytes sent to that client

Order the results by decreasing "total bytes sent", and include only the top 20 results. *(Solution's query takes about 33 seconds to execute.)*

(Prodigy[2] was the most popular Internet Service Provider at the time.)

6d) Write a query that reports the daily request-total and the total bytes served, for each day starting on July 23, 1995 (a Sunday), and ending on August 12, 1995 (a Saturday). The query result should include the date of each day (`datetime_dim.date_val` would be fine), the total number of requests on that day, and the total number of bytes served on that day. **Make sure that every day in this date-range actually appears in the result!**

You should notice couple of gaps in that time period. One of the gaps has a good reason; what caused this gap? (Hint: See the web page mentioned at the start of the assignment, where the NASA logs were retrieved from.) **Include your answer to this question, as a comment before the query.** (The other gap has no explanation. Make sure you identify which gap is which.)

Some harder queries (8 points per query):

These queries require multiple aggregation steps, but they are much more similar to typical OLAP queries. (OLAP database systems actually provide functionality to perform multiple aggregation steps at once, but frequently you don't have this luxury. The `WITH` clause would make these easier, but MySQL doesn't support `WITH`.)

6e) For each day that appears in the data-set, report the resource that generated the largest "total bytes served" for that day. The result should include the date, the resource, the total number of requests for that resource, and the total bytes served for the resource during that day. Order the results by increasing date value. Don't worry about including the dates for which there are no log records.

[2] http://en.wikipedia.org/wiki/Prodigy_(online_service)

Important Note: a given resource can have multiple **`resource_id`** values associated with it, if the method, protocol or response was different when the resource was accessed. For example, the resource **`/shuttle/countdown/count.gif`** has four **`resource_id`** values. Therefore, your query must group on resources, not resource IDs.

To double-check your answers, here are some partial results:

- For 1995-07-03, the number of requests is 204, and total bytes is 143,209,714.
- For 1995-07-10, the number of requests is 123, and total bytes is 91,171,362.
- For 1995-08-07, the number of requests is 858, and total bytes is 37,037,014.
- For 1995-08-29, the number of requests is 96, and total bytes is 69,974,274.

6f) **We would like to see if the pattern of "average visits per hour" is different between weekdays and weekends.** Write a query that computes the average number of visits per hour, over all weekend days, and over all weekday days. The result should have rows and columns like this:

hour_val	avg_weekday_visits	avg_weekend_visits
0	194.2195	201.3750
1	162.4878	166.6250
2	147.4878	130.5000
…	…	…

In other words, one column contains the average visits per hour, computed only over the weekdays, and the other column contains the average visits per hour, computed only over the weekends. (It is essential to compute the *average* visits per hour, since there are many more weekdays than weekends in the log data.)

The number of visits in a given time period is simply the number of distinct **`visit_val`** values that fall within that time period.

In a comment before your query, include a brief explanation of any significant variations you see between weekday visit patterns and weekend visit patterns, as well as the probable cause of these variations. *(Hint: In 1995, did most people have always-on and/or broadband Internet access at home? Things were a bit different back then.)* Keep in mind that hour 0 corresponds to 12:00AM, hour 12 corresponds to 12:00PM, etc.