Project

CSE 241
Database Systems
Spring 2016

Goal:

The goal of this project is to provide a realistic experience in the conceptual design, logical design, implementation, operation, and maintenance of a relational database and associated applications. First, I shall describe the application, then the categories of requirements, and then some suggestions on how deeply you need to go in each category. A real project of this sort would require a substantial development team working for several months (or more). You will do this alone over several weeks. I have chosen to go with individual rather than group projects because the goal of this project is for you to gain a personal appreciation of the depth and breadth of issues that go into the design of a database application, rather than to have you specialize in just one aspect (and rely on others for the rest).

The project can go well beyond the minimal requirements. I encourage such extensions, but be sure that your project will run in our intended testing environment.

The description given here of the enterprise you are modeling is necessarily somewhat vague and incomplete. This is by design — in real life, your "customers" are managers in the enterprise whose degree of computer literacy is, to put it kindly, variable. You will need to fill in the holes in this document to create a precise design and concrete implementation of the interfaces and applications using the database. The checkpoints specified for the project are designed to help you get some feedback along the way and keep you on schedule.

Please pay particular attention to the requirements for project submission and the testing protocol I shall use. Because of the large number of projects that need to be evaluated in a short time, the testing protocol will be strict and exceptions will not be granted.

Enterprise description:

The enterprise is a new wireless telephony provider, Jog Wireless ("We're like Sprint, only slower"[1]).

Jog provides mobile phone services to its customers. These services include voice calls, texts, and Internet access. Jog also sells phones both online and from physical retail stores. These products and services are priced in a variety of ways (described below). You will be modeling only sales and billing aspects of Jog, not employees and salaries, infrastructure maintenance, customer help-desks, nor actual call processing (just the billing for calls). We shall also omit security issues (i.e. customers won't need to log into their Jog accounts; we'll leave security for our security courses).

- **customers:** Customers have names and physical addresses. The name may be that of a person or a corporate entity. A customer may have more than one account.
- **accounts:** One does business with Jog by setting up one or more accounts. An account could be an individual account with one phone number, a family plan with multiple phone numbers up to some limit set by upper management, or a business plan that allows an unlimited number of phone numbers. Each account has a primary phone number associated with it that is used by Jog to reach the customer responsible for the account.

[1] Prof. Spear suggested that I add a note that Jog is better than Sprint for long-distance telephony.

- **phone numbers:** We'll assume all phone numbers are in the North American 10-digit format (Jog has not yet gone international). We'll use the notation 999-999-9999 for phone numbers. Each account has a primary phone number that is used by Jog to contact the account owner.

- **phones:** Normally there is a phone attached to each phone number, but there may be none (due to service suspension, a phone being reported as lost or stolen, etc.). There can be only one phone at most attached to a given phone number at a given time. Jog needs to know the type of phone attached as well as its identifying information: manufacturer, model, and MEID (a 56-bit unique ID, typically presented in hexadecimal). (Some real-world complexity is being left out here for simplicity). Jog keeps a history of phones that were on its network but have been removed. This history consists of the identifying information for the phone as well as each time period that it was on the Jog network and the phone number with which the phone was associated during that time period.

- **usage:** Usage can be a voice phone call, a text, or Internet access. Jog does not have its own messenger feature such as Apple's iMessage, as Jog is slow to adopt newer features. Depending on the billing plan usage will be charged in one of several ways. But regardless of how billing is done, data must be retained regarding every phone call and every text:

  - source phone number
  - destination phone number
  - time (for text), start and end time (for call)
  - size in bytes (for text), duration in seconds (for call)

  Jog does not retain the text of texts nor does it record calls. It also does not retain any data regarding user's Internet access except for total bytes used per month (upload and download are not recorded separately by Jog even though most of its competition does differentiate between upload and download). (Whether other agencies retain such information is beyond the scope of this course.)

  Be aware that either the source or destination number (but not both) could be non-Jog numbers (as when a Jog customer calls a Verizon customer or vice versa). We'll ignore the number-translation and number-portability issues that come up in real-life (stuff I once had to worry about personally at Bell Labs, so ask me if you are interested).

- **billing plans:** There are a huge variety of billing plans in the wireless phone service market. We won't list them all here, but instead include a few examples.

  - fixed rate per call minute
  - fixed rate per text
  - fixed rate per byte
  - a fixed monthly rate for an upper limit of minutes, texts, bytes
  - free calls and texts to a specified list of "frenemies"
  - free calls and texts to other Jog customers

  There may be different charges for minutes, bytes, and/or texts that are incoming versus outgoing.

  You will need to work with Jog management to identify the plans in use and to implement them precisely in the billing interface. This means you have a (imaginary) discussion with Jog management and report the results in your README file so we can test whether you have properly implemented the intentions of the enterprise.

- **physical stores:** Jog's regular retail stores stock several types of phones and have a limited inventory of each one. Jog also has one special store that is not open to the public that serves online customers. For our purposes, we shall assume that this special store has infinite inventory and can accept restocking

requests from the regular retail stores. When a retail store's manager submits a restock request for a phone, its inventory goes back up to its preset limit.

- **Time:** Time is an important data item in phone records. You should represent time using the Oracle **timestamp** datatype and covert timestamps to character strings using the *to_char* and *to_timestamp* functions built into Oracle. If you are inputting time values, be sure to check them carefully for validity.

Interfaces:

There are many types of users of this information system: customers, retail store managers, the system that processes usage records, the system that generates monthly bills, the fraud detection and prevention system, etc. You should construct approximately 2 or 3 good user interfaces, with at least 2 of the 3 categories below (interactive, stream, and reporting represented. This is approximate because a single interface might accommodate multiple types of users and thus "count" as more than one. We shall grade focusing much more on quality than quantity (3 badly built interfaces will get a lower score than two elegant ones).

Your README file should tell us about each interface. Each interface should be designed to be usable by the customers or employees of the enterprise and not contain SQL jargon. The users vary for each interface.

- **interactive interfaces:** Interactive interfaces should test for proper input (data type, range and validity) and allow convenient re-entry after an error. It should be possible to query for information (e.g. don't make a customer guess what phones Jog sells; instead allow the customer to ask for a list). Retail store managers who process sales or request reorders should have an interface of quality similar to that of a customer. In no case should an exception be thrown (you can be sure that if your interface asks for a number, I'll type letters, for example).

  There are several interactive interfaces you might consider including a customer starting new service, a customer ordering a new phone, a customer changing a calling or data plan, a retail store whose clerk is selling a phone, a retail store manager checking inventory and reordering, as well as others.

- **stream-input:**

  Stream interfaces should also test input, but need to recover in an automatic and smooth manner, skipping erroneous data but re-synchronizing so that the next correct input record is processed properly.

  The usage record processor should accept a sequence of usage records from a file in the formats:
  source phone number, destination phone number, time, bytes
  source phone number, destination phone number, start time, end time
  source phone number, type of Internet access (upload or download), bytes
  If a record is erroneous, that record should be output to an error file and processing picked up again wth the next record. ("record" is synonymous with "line" in this context. beware of Java's idiosyncrasies with the newline character.)

  The interface to start record processing must ask the human user for input and output file names. As for any interaction with users, it needs to allow retries if a file is not found for input or if the output file cannot be created. Thrown exceptions lead to substantial point penalties.

  For our testing, you must provide us a sample file with time values after the project due date that we can run without creating duplicate entries in your database. We cannot create our own test file without knowing the phone numbers that you have in your database; thus we require that you generate the file for us. (Of course, we may then edit it for testing purposes, e.g., by intentionally introducing errors.) Note that you need to tell us the file name in your README file.

- **reporting systems (billing, fraud, etc.):**

  A reporting system begins by asking for certain parameters and then generates a "report" using data in the database.

The billing system generates a bill for a given month/year for either a specific customer or for all customers. The billing system asks for the month and customer and then scans usage data to generate the requested bill(s). Since the bill depends not only on the usage records but also on the current billing plan, generated bills need to be stored as an entity in the database for future reference. Otherwise, if a customer's billing plan changes, it would then be impossible to regenerate old bills.

The fraud system analyzes calling patterns of customers to detect patterns that suggest fraudulent calls as might occur if a phone is stolen. This is done best with advanced data analytic / data mining techniques not covered in this course. Apply such techniques if you've learned them elsewhere. Otherwise, design your own technique and describe it in your README file. One example: Have some customers who are "bad" people (e.g. D. Vader, C. de Vil, B. B. Wolf) and look for "good" customers who suddenly start making calls and sending texts to "bad" people. The generated report should alert Jog management to concerning customers and include a reason for such concerns.

- For whatever interfaces you write, you are encouraged to put as much code as possible into PL/SQL stored procedures, though Java-centric approaches are acceptable. Excellent PL/SQL work can earn an extra point or two.

What you need to do:
There are several steps to this project. Although it is inevitable that you will need to go back and change things as you move along, it is desirable to do a very good job at each step so as to reduce the amount of work that winds up being redone. Note that "very good" does not mean "perfect." If you take too long making the early stages of the project perfect, you'll find yourself pressed for time at the end. You'll need to stay on schedule and learn, as one has to in real life, when to "declare victory and move on." Here is a set of stages to follow:

1. **ER design** Construct a good, complete ER design for the enterprise. There will be a formal checkpoint in which a grader will review your design. It is worthwhile to refine this design to represent the enterprise with a considerable level of detail. That makes the next steps much easier.

   To start, it is best to sketch the diagram with pencil and paper. Not only are there a lot of changes initially, but often you discover that the the placement of entity sets on the page can influence how many lines cross. Relocating entity sets physically on the page can clean up a digram considerably.

   Note that a good ER design includes careful choice of what things are entity sets and which are relationship sets, proper placement of attributes, use of generalization/specialization where appropriate, etc. A common error is to think relationally and then reverse-engineer the ER design. That approach often leads to hidden relationship sets encoded in common attributes between entity sets. Foreign keys are a relational concept; they are not a feature of ER designs.

   As you make decisions, include notes explaining the assumptions you made about the enterprise leading to those decisions. That will help you when you go back and reconsider your design. You may want to turn in some of those notes with your diagram to help me understand how you view this enterprise.

   Once your design is well along the way, you will need to create an electronic version. If you use software tools for this, be sure the tool can generate a pdf file. We shall be accepting only pdf for ER diagrams. There are many ER notations in use. For this project, you are required to use the notation used in the text and not any other notation.

2. **Relational schema** The text gives a set of rules for generating a relational schema, including primary-key and foreign-key constraints, directly from the ER design. If your ER design is good, you will be nearly done at this point. There may be some data dependencies that were not captured in the ER design that may lead to some further normalization. Check for this, but for a good ER design there won't be many, if any. You may decide to add some additional indices for performance. You may decide to add some triggers or stored procedures later on. Those don't all have to be done at the start of your work, you can always add them.

Once you have a conceptual version of your relational database schema, you need to generate a SQL DDL version of it. That means deciding on reasonable datatypes and getting the syntactic details of SQL right. Enter this in Oracle under your account. By default only you (and DBAs) have access to these tables.

Note that if at any point you find flaws in your database design, you need not only to fix the design in Oracle but also make any changes that may be required in your ER design. When you submit the final project, your ER diagram must be consistent with the database you have created in Oracle.

Hint: Don't type your DDL directly into SQL Developer. Instead create a plain text file with your DDL and copy/paste it into SQL Developer. If there are errors, edit the file and then copy/paste again. This allows you to retain a full version of your DDL for future editing without having to extract it from Oracle. This is important for several reasons: (1) should we have a catastrophic failure of edgar1, you can restore your schema quickly on a replacement machine (2) If you take a look at the SQL DDL Oracle generates for your schemas, you'll see that it is unnecessarily complex since it includes specification of all sorts of parameters for which we are using the default.

Note that you have to drop tables before you can re-create them. I just put the drop statements at the start of my DDL text file. Also note that foreign keys can't reference a relation that does not exist. So be sure to list the create table commands in an order that ensures that referenced tables are created before referencing tables. For dropping tables, you need to drop the referencing tables before the referenced tables.

3. **Data generation and population of relations:** You need to put data into your tables. Include enough data to make answers to your queries interesting and nontrivial for test purposes, but there is no need to create huge databases.

   To avoid a typing marathon for data generation, write a program to generate test data, or use data that you can find on the web (but use only data that may be copied legally). You can get some data fairly easily without much typing. For example, you can get a bunch of names for people by doing a **select** *name* **from** *student* in the university database we are using for SQL homework assignments. Then you can write a program to pick names from that list randomly. Similarly, you can generate random numeric data. In any case, you will want to automate data loading so that if you need to redesign part of your database or your interface code trashes the data due to a bug, you can reload your data without too much effort. A trick in the automated process is working around referential integrity constraints. Inserts have to be done in the right order to avoid a foreign-key violation.

   Be sure to cite your data sources in your README file.

   There are a couple situations that merit special attention:

   - **Representing time:**
     Recall the JDBC example *Time.java* where we input data of type **timestamp**, stored them in the database, and took the difference between two such values, then printed that out. The format of **timestamp** data is detailed and it is easy to make mistakes. Thus, you need to be really carefully testing user input of time data. For good quality output, you likely will need the *to_char* function to get the format of output you'd like.
   - **Generation of unique id values**
     Clever Java programming can deal with this. Alternatively, you might consider *sequence counters* created in SQL using the **create sequence** statement (see page 1043 of the text). Here is an example:

     ```
     create table someTable (id numeric(5,0), name varchar(20));
     create sequence getID;
     create trigger nextID before insert on someTable
     for each row
     ```

```
begin
select getID.nextval into :new.id from dual;
end;
insert into someTable values (null,'aardvark');
insert into someTable values (null,'bobcat');
select * from someTable;

ID                   NAME
-------------------- --------------------
1                    aardvark
2                    bobcat
```

Oracle has a proprietary bulk loader whose use we do not cover in this course, but simply generating insert statements and running them is probably easiest.

You are allowed to share raw data files but not code. Note in your README file your data sources and also to whom you've provided raw data.

4. **Interface coding:**

   Don't forget the basics of good programming.

   - Check user input for being valid. If you are inputting an **int** using *nextInt* first check that the user did not enter a nonnumeric character (recall *hasNextInt*). Produce good-looking output. And so on... Then when the user does something wrong, don't just quit. Provide a chance to try again without requiring unnecessary re-entry of input data.

     You may note that my sample code in lecture does not do this. That's only because I'm trying there to focus on the new material and avoid code that distracts from my main point. But for the project, I expect better quality code. (yes, a double standard!)
   - Make good use of classes and methods in your Java code. Use PL/SQL triggers, functions, and procedures where appropriate. I would prefer to see as much code as possible in stored PL/SQL procedures and as little as possible in Java. Although the grading rubric is about database functionality and not Java coding, I will deduct points for seriously poor coding and perhaps give a point or two extra for particularly good use of database procedures that help streamline the Java code and ensure database integrity.
   - Integrity checking: A well-designed database will protect against many types of bad updates. But others may not be easy to express using SQL constraints. Think about bad things I might enter using your interfaces and try to protect against them in your interface code if your database integrity constrains and triggers are not enough to do the job. In all cases, avoid having your code crash on an exception. Catch them and do something reasonable. That includes catching exceptions thrown by JBDC methods due to errors generating by Oracle.
   - Concurrency: In real life, lots of updates and queries would be run on this database concurrently. We should be able to run several instances of your code from separate terminal windows and not run into anomalies. In most cases, Oracle's default concurrency will do well enough, but see the note below about self-inflicted concurrency disasters.
   - Test with care. Get the basics running first.

5. **Self-Inflicted Concurrency Disasters and Zombie Attacks:**

   Although real-life concurrency is not likely to cause much trouble in this project, you can get yourself into trouble as you test your code.

   If your Java code with JBDC calls to the database terminates abnormally (i.e., it throws an exception) or you simply forget to close your connection, your code may leave behind zombie transactions that

Oracle thinks are still active. If those transactions hold locks, your subsequent test runs may wind up waiting for those zombie transactions to complete. You'll see this as the system "hanging" without explanation. Eventually, in a matter of minutes, the connection (and its transactions) will time out and all will be well again.

You can avoid this by careful testing. Test your plain Java code before you add JDBC code. Be sure to catch every exception and then close everything before your code terminates. Exceptions can occur anywhere: during connection, during SQL execution, during result fetching, and during execution of plain Java code (array bounds, for example).

In principle, as DBA, I can kill transactions manually. However, this is an error-prone process since I need to find your session ID, guess your transaction ID (since they may be several,) and then kill that transaction. I could misfire and kill a useful test run you are doing. Typos can be catastrophic. More likely, by the time I'd get to this, the connection would have timed out anyway. For this reason, plus the large number of projects, I don't offer transaction-killing service and ask instead that you be patient with the timeout period if something slips past your attempts to code carefully.

6. **Password Security:**

   Since your Oracle password is needed to run JDBC, and since the default file protection in on the CSE machines is publicly readable[2], you are **forbidden** to include your password in plain text in your code. Instead, prompt for the user to input the password at the start of each of the assigned programs. We don't need to know your password. I'll use my DBA authority to change your password and we can enter that new password to text your code.

Checkpoints: There are several checkpoints scheduled. These are set in order to keep you on target for completing the project on time.

- **checkpoint 1: Mar 1.** It is very important that you get direct, interactive feedback on your ER design. During the weeks of Feb 22 and 29, we shall allocate a substantial block of time for 15 minute meetings. There are currently over 130 students total in 241 and 341, so it will be a challenge to get everyone scheduled. We will have sign-ups in advance for specific time slots and we shall try to stay on schedule to avoid queuing. Of course, we can schedule additional meetings as needed. Look for a posting from on Piazza regarding signup logistics.

  By the start of class on Mar 1 or the time of your scheduled 15 minute meeting, whichever comes first, you must have a pdf file with your ER diagram submitted on CourseSite. Also, bring a hard copy of your most recent ER design with you to the meeting.

  It is to be expected that we will suggest changes. The 1 point out of 33 for the project allocated to this checkpoint will be awarded for a reasonable ER diagram. It need not be perfect to get the point (and there is no partial credit). The ER grading will be much more stringent in the final version submitted at the end of the project. Thus, a perfect score on this checkpoint is not a guarantee of a perfect score for the ER component on the final version of the project, just an indication that you are on pace at this point.

- **checkpoint 2: Mar 24.** At this point you should have your relation schemas created in Oracle. At some point after this date one of the graders will look online to see that they are there and include reasonable key declarations (including foreign keys). I don't expect to see "fancy" features like triggers or stored procedures, but it is certainly fine if they are there. The whole point of this checkpoint is simply to keep you on pace. This review will be cursory rather than evaluative (that is, if there is reasonable stuff there you get the point regardless of quality and there won't be any comments), so if you have questions, you should be sure to ask (on Piazza or in office hours).

[2]Note that I've also required that you change the file protections, but this is one extra layer of security on my part.

The 1 point out of 33 for the project allocated to this checkpoint will be awarded for a reasonable set of relation schemas. As for checkpoint 1, it need not be perfect. We will take a more careful look at your schemas when we evaluate the final project. So, as is the case for the first checkpoint, a perfect score on the checkpoint is not a guarantee of a perfect score on the relational design component of the final version of the project.

At this point you should also have in place a plan for user-interface development. We shall not be reviewing that plan at this time.

- **checkpoint 3: Mar 31.** We shall check online to see that your relations have been populated with data. We shall award the one point allocated for this checkpoint if all your relations have a reasonable amount of data. Note that if you have a very large database schema (because you are doing more than the minimum), it is okay that only some relations are populated.

  You should have some degree of user-interface functionality at this point, but we will not check that.

- **checkpoint 4: Apr 7.** At this point you need to be well along in completing the project. There will be no graded submission at this stage, but you should take a moment to write down a specific schedule for the remaining tasks and then refer to it to stay on a path to completion.

- **checkpoint 5: Apr 21.** At this point, you need to test your ability to generate a submittable project. If you wait until minutes before the deadline to generate the required jar file, you'll likely make an error and wind up with no credit for the executable part of the project. If you have not made sure you know how to extract your java code from your IDE in such a way that we can recompile your code, now is the time to figure that out.

  I expect at this point that you have at least one interface working well and other nearing completion. Pretend this is your completed project and generate the required jar file, java source code file or directory, and then the required zip file. Move this to a new subdirectory of your cse241 directory (not elsewhere, lest you accidentally make it publicly readable), then unzip it and try to run it the way we will "java -jar xyz123.jar". Also, try to compile your extracted source code using "javac *.java" or by providing us a makefile allowing us to type "make xyz123.jar".

- **project due dates**. The project will be due in 3 parts: the ER design, the relational data, and the executable project code, each with its own due date.

  - **ER design:** *Monday April 25 at 11:55pm hard deadline* Submit on CourseSite a single pdf file with your ER diagram (just one file, not several zipped ones, no formats other than pdf). CourseSite will enforce the deadline.
  - **Relational data:** *Monday May 2 at 11:55pm soft deadline* At any point after the deadline, we may look at your relational design and data for the purpose of assigning a final grade. We realize that as you test your executable, some data may be added or deleted, but at this point, there should be no further changes to the relational design and the main loading of data should already have happened.
  - **Executable code:** *Friday May 6 at 11:55pm hard deadline*

  Read the details below on what to turn in very carefully.

What to turn in with your executable code:

1. The top-level directory should be named using your loginID and last name (for example, lbj265johnson). This top-level directory should then be compressed using the zip format under the same name (for example if the folder is lbj265johnson, the zip file should be lbj265johnson.zip). No other compression formats will be accepted for this project. To zip a project on the sunlabs, use the command "zip -r ZipFileName FolderName".

   Submissions that use any format other than zip will not be read and will receive a score of zero. (So no *rar* format files and no tarballs)

2. A README file at the top-level in the directory hierarchy that explains what is where, etc. Include usage instructions for the interfaces (perhaps suggestions of good customers to test, for example). Also include sources of all data and code obtained from others (note the collaboration rules below). Be sure to include your name at the start of the README file. If you have done anything extra, be sure to point it out here!

   The goal here is ensure that we see all the goodness in your project and to help us have a good first experience with your code. A project that opens with "enter your ID" is not very friendly to a grader who has no idea what ID to enter. A good README file can provide some first suggestions. We'll read your database to find others and create our own new users too.

3. One executable jar file that provides access to all of your interfaces. The file is to be named xyz123.jar where xyz123 is your Lehigh loginID. We'll run "java -jar xyz123.jar" in a terminal window on a Sunlab machine of our choice, using a "virgin" student account. So you cannot rely on any special settings for environment variables, etc. You cannot rely on us having the Oracle ojdbc jar file or oracle subdirectory in any specific place, so put it in the right spot yourself. If your jar file fails to run, you get a zero for the executable part of the project. Since you have the ability to test this yourself, there is no reason for us to attempt to do debugging.

4. One directory containing your Java project code, named xyz123 where xyz123 is your Lehigh loginID. If we wish, we may recompile your code. We'll do this either by "javac *.java" or use a makefile you provide. If there is a makefile to use, it must be noted in the README file. Note that we will compile using the default Java version in the Sunlab and will NOT do it in the framework of any specific IDE. If we decide to compile your code and compilation fails you may get a zero for the executable part of the project.

5. A second directory for any data-generation code you wrote. We most likely won't test this, but your code needs to be there for our review.

6. We will use DBA rights on Oracle to change your password and thus we shall be able to look at your tables. This means that your data must be on the Oracle system we are using for our course and cannot be on a personal installation of a database system. Our tests may modify your database and we will not restore your database to its status prior to our tests. (Also this means you should not be using your course account on edgar1 for any personal data or for any other course, since you'll lose access to such data).

7. Do NOT turn in a listing of all your data. We can see them online.

Grading: I shall use the following approximate template for grading:

1. Checkpoints:

   3 points, 1 each for checkpoints 1, 2, and 3.

2. ER design:

   6 points

3. Relational design, including constraints, triggers, and indices:

   6 points

4. Data creation: sufficient quantity, reasonable realism, sufficiently "interesting":

   3 points

5. User interfaces, including proper features, proper updating of the database, etc.:

   15 points

   Note that bad database design can lead to interface problems, so design issues actually can have a larger impact that just the points specifically assigned to them.

   Getting details right is important. Something as minor as throwing an exception on user input can cost a couple of points. Just like when you buy software yourself, the evaluation is not based on the percent of code that is correct! We'll be evaluating your code as if we were users/managers at Jog Wireless.

6. I reserve the right to give extra points for exceptional solutions to parts of the project. I also reserve the right to deduct points in the unlikely event that we identify problems not covered by the items above.

7. Draft submissions: CourseSite will be set up to allow you to upload a draft without clicking submit. We won't grade drafts until after the deadline. But if you click submit before the deadline, that constitute a submission and we will then be free to grade your submission before the deadline at our convenience.

   Submitting a draft is a good way to protect yourself by submitting something that mostly works while you continue to make improvements. That way, you have something there just in case your fixes turn out to fail badly.

8. Lateness: With the deadline set for the last day of classes, there is no grace period for late submission. If you are behind schedule, focus on having some things work and explain those things that don't. If everything is "99 percent okay" but nothing actually runs, that's a zero on the executable.

9. **Draconian Policies:** It is very important that you take the time to ensure that the logistics of project submission are done right. If you make a simple goof (upload the wrong file or zip a wrong directory, etc.) it may seem unreasonable to give a grade of zero on the executable part considering all the work you did. However, lacking this policy, we would be making it trivial to gain an extension of time through a careless "goof". While our course sizes are this large, I have to be strict in this regard. Check, recheck and be careful!

   Note that computers seem to crash around project deadlines. You are responsible for backing up all your project code and data. While we take due precautions with edgar1, it is your responsibility to restore your database if edgar1 were to fail disastrously and all backups on our side were lost. Edgar1 backups are not as frequent as those for general sunlab or LTS data. I suggest backups to a personal external hard drive or USB drive.

Collaboration:

- Your project database design and interface implementation is to be your own work with no outside help except from the course graders or from me.

- You may share raw data to load into your database or obtain data from public domain online data sources. Include a note in the README file as to the source of your data; also include a note if you have given data to someone else. Data sharing that is not documented in the README file constitutes an honor code violation.
- We intend to run project code through the Moss system to detect cases of possible plagiarism. In addition I may select some projects for "interview meetings" with me where you will discuss your projects with me sometime between the end of class and the end of finals. Selection for these interviews will be based both on random selection and flagging by Moss. So the fact that I call you for an interview does not imply that Moss flagged you – instead you could just have been selected by the lottery.