Project

CSE 241 (both sections)
Database Systems
Spring 2018

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, we 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. We 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. We 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. (We note some of these below by asking you to consult with the managers of the enterprise, which means you need to decide yourself on enterprise policy perhaps after some web searches.)

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 we 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 *Big River Crossing* (BRC), an online retailer selling almost everything.[1] BRC recently acquired a supermarket chain, *Regork*,[2] giving BRC physical stores to complement its online sales and to provide an entrance into the fresh-food and grocery markets.

You will be modeling only product-sales and inventory aspects of BRC, not employees and salaries, maintenance of physical infrastructure, etc. We shall also omit security issues (i.e. customers won't need to log into their BRC accounts; we'll leave security for our security courses).

- **customers:** Customers have names and physical addresses. They may have a credit card (and must have one for online sales). BRC management is considering accepting Bitcoin or other cryptocurrencies, but currently BRC accepts only credit and debit cards online. Regork accepts cash and checks as well as credit and debit cards (though no customer has used a check since the last millennium).

  Customers who join a frequent-shopper program provide some personal information based on what the enterprise requests. They may refuse to provide some information. Customers come into a Regork

[1] Any resemblance between BRC and another online retailer named after a big river is purely coincidental.
[2] Any resemblance between Regork and a supermarket chain serving, among other cities, the home city of BRC, is purely coincidental.

Figure 1: On the Tennessee/Arkansas line above the Mississippi River, Memphis in background.

store or go online with BRC to buy a market basket of goods. They may choose to order from BRC online and pick up at a Regork location or have the order shipped.[3] Not only must this data be stored, but also the system must be able to handle multiple customers buying goods at the same time.

- **physical locations:** BRC's acquisition of Regork means that BRC maintains a variety of store locations. You need to keep track of the inventory at each such location.
- **products:** A particular type of product may be sold in a variety of sizes and a variety of brands. For example, cola is sold under such brands as Pepsi and Coke. Product types form an specialization/generalization hierarchy. For example cola is a type of soda, which is a type of beverage, which is a type of food. Some products fit into multiple categories. For example, baking soda is a cleaner, a food (since it is used for baking), and a drug (since it may be used an an antacid), but it is not a type of soda. Some products are sold only in Regork stores, others only online, and some in both.
- **vendors and supply chain:** Products are sold to stores by vendors. A vendor may sell many brands (e.g. Pepsico sells Pepsi, Tropicana, Aquafina, Gatorade, Lay's, Doritos, Quaker, and others; Samsung sells phones, computers, televisions, and many other items under its own brand name). Products may be shipped from vendors periodically and/or as a result of specific reorders for specific quantities. For simplicity, we shall assume that when a reorder request is make, the products arrive instantaneously. (This avoids you have to model the supply chain and the delays involved therein.)

[3]We shall not be modeling the actual shipping of goods so as to keep the project within reasonable bounds.

- **prices:** There are lots of prices. At any point in time a product has a price. But if the price changes, the price paid in prior sales is unaffected. That's true both for customer purchases of products and the enterprise's purchase of new inventory from vendors. Furthermore, physical locations do not all have to have the same prices, nor do they need to have the same price as the online store. So clearly, a product does not have one single specific price. There will need to be some way to represent all this in a reasonable way in the database. Figuring out how to represent prices in a way that leads to a clean database design will take considerable thought.

Interfaces:

Before your real interface runs, prompt for the user to enter the Oracle password for your Oracle account on Edgar0. This keeps your password out of your Java source code. We shall be running a script at the project deadline to change your password to something we know, so having your password in your Java code is not only insecure but also ensures that your final project will not execute for us when we evaluate it.

There are many types of users of this information system: customers, store managers, product managers, enterprise-wide marketers, etc. You should construct approximately 3 good user interfaces (and just note in your README document a few others that would be included in a real system). This is approximate because a single interface might accommodate multiple types of users. We shall grade focusing much more on quality than quantity (4 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.

- **Code quality:**

  Just as you would not expect software that you buy or use to crash, throw exceptions, or fail to handle input errors gracefully, we expect well written, error-tolerant code. Interactive interfaces should test for proper input (data type, range, and validity) and allow convenient re-entry after an error.

  You should not expect your uses to have the database memorized. It should be possible for users to query for information.

  Your code should *never* throw an exception. Catch exceptions, print out a reasonable message, and have the code carry on in a reasonable way, or, if no reasonable continuation exists, terminate gracefully with a message to the user.

- **Interface details**

  At least one interface (and ideally more) should be highly interactive and include updates to the database. Examples include supporting a customer entering an order online, a customer bringing a market basket to the checkout, or a new customer creating an account. We'll expect to be able to run multiple instances of this interface in separate 2 terminal windows and have everything work. (Yes, we'll have two customers both try to buy the last widget in the store.)

  Of course, beyond the customer-facing interfaces, there are other database activities that take place, such as re-order for low inventory, tracking total sales by each physical store, etc. These data then enable a manager's interface that may report sales statistics, possibly making good use of SQL's OLAP and ranking features. BRC management may be interested in top locations by sales, top products by sales, trends in sales over recent months, sales breakdown by customer demographic data, etc. It is possible to design some clever interactive interfaces for business analytics if you choose to do so.

  As we note again later, projects will be tested only *from the command line on the sunlab machines* and possibly via remote login. This means that not only do you not get credit for having a GUI, but rather that you will perhaps lose all credit for the executable, since GUIs do not run over a remote terminal window. We realize that a command-line interface is a last-millennium concept, but it simplifies the overall project and provides us with major efficiency gains when we have to evaluate all projects in a tight time window.

- For whatever interfaces you write, you may use as much PL/SQL as you wish. We generally prefer that you have the database system do the work rather than your non-JDBC Java code as much as possible. Making effective use of what Oracle has to offer can compensate for a lost point of two elsewhere.

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." If you manage your coursework on an "earliest deadline first" basis, you'll find yourself in deep trouble on this project. But if you stay ahead of our suggested pace, you'll likely have a quality project.

Here is a set of stages to follow:

1. **ER design** There are many ER notations in use. For this project, you are required to use the primary notation used in the text and not any of the other notations that appear at the end of Chapter 7, nor any other notations that you may find online. Note also that ER diagrams are *not* schema diagrams, which appear in Section 2.4 of the text and which we did not cover in the course.

   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 diagram 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. (and, such errors are one of the prime ways students lose points on the ER section of the grading rubric)

   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 us 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**. If you hand-write your diagram, be careful that your writing is legible and be careful that your scan/photo is also readable.

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 the instructional team[4]) have access to these tables.

[4] So, don't keep any personal data in your 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 the system, 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. Just put the drop statements at the start of the 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.

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

   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**

     Creating unique identifiers automatically in Oracle took a bit of work until the introduction of the autoincrement feature in Oracle 12.

     Text below is from the upcoming 7th edition of the book.

     Suppose that instead of declaring instructor IDs in the *instructor* relation as "*ID* **varchar**(5)", we instead choose to let the system select a unique instructor ID value. We could write instead:

     ID **number**(5) **generated always as identity**

     Then, when we insert into the *instructor* relation, we must omit the *ID* attribute, leaving it to the system to generate the value. The *ID* value can be found via a normal **select** query. If we replaced **always** with **by default**, we have the option of specifying our own choice of *ID* or relying the system to generate one. Additional options can be specified including setting minimum and maximum values, choosing the starting value, choosing the increment from one value to the next, and deciding whether exceeding the maximum causes the values to wrap around back to the minimum. Below, we show an example with many of these options specified:

ID **number**(5) **generated always as identity**
minvalue 1
maxvalue 99999
increment by 1 **start with** 1
cycle
cache 10

This example sets the range of IDs to be 1 through 99999, with the first ID issued being 1, with the values going up by 1 from that point. The **cycle** option allows wrap-around. The **cache** option specifics the number of IDs the system precomputes so that it can provide a new *ID* rapidly on demand (otherwise, there could be a delay as the system, after wrap-around, searches for an unclaimed value).

When the **always** option is used, any **insert** statement must avoid specifying a value for the automatically generated key. To do this, use the syntax for **insert** in which the attribute order is specified (see Section 3.9.2). For our example of *instructor*, we need specify only the values for *name, dept_name*, and *salary*, as shown in the following example:

insert into *instructor* (*name, dept_name, salary*)
values ('Newprof', 'Comp. Sci.', 100000);

- **Bulk loading**

  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.

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 our sample code in lecture does not do this. That's only because we're trying there to focus on the new material and avoid code that distracts from our main point. But for the project, we 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. Although the grading rubric is about database functionality and not Java coding, we 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 we might enter using your interfaces and try to protect against them in your interface code if your database integrity constraints 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 generated by Oracle. Note that we may attempt SQL-injection techniques to try to cause errors or exceptions.

- 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. Unit testing is your friend.

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. The try-with-resources construct in Java is a big help here. 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, the DBA can kill transactions manually. However, this is an error-prone process since it involves finding your session ID, guessing your transaction ID (since they may be several,) and then killing that transaction. The DBA could misfire and kill a useful test run you are doing. Typos can be catastrophic. More likely, by the time we'd get to this, the connection would have timed out anyway. For this reason, plus the large number of projects, we 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.

   An even worse condition occurs if you do not close your connections properly, Oracle sessions may survive for quite some time even though you kill the operating system process that ran your code. These zombie sessions, if sufficiently numerous, impede the function of the system. We can tell who is leaving such sessions behind, and while we realize the occasional zombie is inevitable, students who generate large numbers of zombies may get points deducted even before grading begins.

6. **Password Security:**

   Since your Oracle password is needed to run JDBC, and since the default file protection in on the CSE machines was publicly readable until recently[5], 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. We shall use DBA authority to change your password and we can enter that new password to text your code. *If you ignore this and you do hard-code your password, you code will no longer work once we change your password resulting in your code failing to execute.*

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

- **checkpoint 1: Feb 20.** It is very important that you get direct, interactive feedback on your ER design. During the weeks of Feb 19 and 26, we shall allocate a substantial block of time for 15 minute meetings. We will have sign-ups in advance for specific time slots and we shall try to stay on schedule

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

to avoid queuing. Look for a posting from on Piazza regarding signup logistics. Bring a **hard** copy of your most recent ER design with you to the meeting (we don't have time in a 15 minute meeting to deal with WiFi issues).

It is to be expected that we will suggest changes to your diagram. 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).

For this project, you are required to use the primary notation used in the text and not any of the other notations that appear at the end of Chapter 7, nor any other notations that you may find online. Note also that ER diagrams are *not* schema diagrams, which appear in Section 2.4 of the text and which we did not cover in the course.

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 22.** 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). We 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. 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 29.** 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 most of your relations have a reasonable amount of data.

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

- **checkpoint 4: Apr 5.** 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 19.** At this point, you need to test your ability to generate a submittable project.[6] 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.

  At this point that you should 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". This will allow you to replicate exactly what we shall do to evaluate your project. If your Unix/Linux skills are limited, seek help from a grader or friend now – don't wait (graders have projects due on the last day of class too!) Getting help on Unix/Linux usage is permitted; just acknowledge in your README file any help you get from anyone not part of the course instructional team.

[6] Recall that Homework 4 was a trial run of this process.

- **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 23 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. *Recall that, for this project, you are required to use the primary notation used in the text and not any of the other notations that appear at the end of Chapter 7, nor any other notations that you may find online. Note also that ER diagrams are* **not** *schema diagrams, which appear in Section 2.4 of the text and which we did not cover in the course.* ER designs not in the required format may receive reduced or even zero credit.
  - **Relational data:** *Monday Apr 30 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 4 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.

   Your README file must be in PLAIN TEXT – not doc, pdf, rtf, or anything else. It must be named either README or README.txt, but note that having a .txt extension is not enough to guarantee plain text format. It must be possible for us to run "cat README" in a terminal window and read your file without having to transfer it to our own machine so as to be able to open it with some app. Test your file to be sure this is the case.

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 Oracle for any personal data or for any other course, since you'll lose access to such data, but we will have access!).

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

Grading: We 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 BRC.

6. We reserve the right to give extra points for exceptional solutions to parts of the project. We 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, we 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 our Oracle system, it is your responsibility to restore your database if it were to fail disastrously and all backups on our side were lost. We suggest backups to a personal external hard drive or USB drive.

   Past experience suggests that the Oracle system will be slow leading up to the final project deadline. Slowness of the system will not be a reason for granting extension of the deadline. If you are working right up to the deadline, always have a latest version ready to submit just in case you don't get that one last test run done in time.

Collaboration:

- Your project database design and interface implementation is to be your own work with no outside help except from the instructional team.
- 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.
- You may receive help on Unix/Linux logistics, but if that help is not from the instructional team, include a note in the README file.
- We intend to run project code through the Moss system to detect cases of possible plagiarism. In addition we may select some projects for "interview meetings" where you will discuss your projects with us 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 we call you for an interview does not imply that Moss flagged you – instead you could just have been selected by the "lottery".