**Overview of Deliverables**

See below for details of what you need to submit for each.

1. Design documents for Phases I and II (2/4)
2. Phase I: Develop DBMS engine code (2/11)
3. Phase I: Develop DML Parser integrated with DBMS Engine (2/18)
4. Phase II: Develop application Integrated with Final Parser and Engine + Report (2/25)

**Project Phases**

1. Develop a simple *database management system* (DBMS): deliverables 2-3.
2. Develop an example application, which calls on the DBMS system developed in Phase 1 to input, store, manipulate, and present data.

**FAQ**

1. For the DBMS engine code submission (2/11), **You don't need to submit file IO functions since they require the parser.**
2. You **DO NOT** need to write a general parser that takes in an arbitrary grammar. You just need to hard-code the pseudo-SQL grammar to implement a recursive descent parser.
3. Table implementation: Vector of vector vs. matrix? Consider tradeoffs.
4. DB app code: will be a mixture of C++ code (menu output, user input, etc.) + DML/query language code (called within the C++ code).   
   Users of the DB app will not have direct access to the DML/query language.
5. [**Test inputs for DBMS.**](http://students.cse.tamu.edu/aalap/CSCE315_s12/sample_inputs/parser_milestone_good_inputs.txt)
6. You should **not** assume that blank space is inserted between all tokens. Example:

kind=="dog"&&name=="hobbes" (instead of kind == "dog" && name == "hobbes")

1. Coding style and commenting will be taken into account.
2. Read the "Deliverables" section carefully. For example, example output is required.
3. **Team configuration:** This project is a team project. Ideal team size is four. If the number of students is not divisible by four, we can have a few teams with size of three. Teams will be formed by the students within each section.

**PHASE I: DBMS**

Database management systems are very complex pieces of software. They support concurrent use of a database, transactions, permission handling, query optimizations, logging, … To be efficient, they utilize highly tuned algorithms developed during the course of decades. So for a four-week long project, we have to simplify. We thus base our DBMS on*relational algebra*.

Relational algebra is a formal system for manipulating relations. It consists of only six primitive operations. Each of the operations take *relations* as arguments, and produce a relation as a result. The operations thus compose freely.

The upside of using relational algebra is that the implementation effort of the DBMS stays manageable. The downside is that queries tend to be more verbose and maybe a bit harder to construct than, say, with SQL.

**Terminology**

Database

a collection of relations

Relation

a table with columns and rows

Attribute

a named column of a relation

Domain

the set of admissible values for one or more attributes

Tuple

a row of a relation (sequence of values, one for each attribute of a relation)

**Relational algebra**

The core operations of relational algebra are:

1. *Selection:* select the tuples in a relation that satisfy a particular condition.
2. *Projection:* select a subset of the attributes in a relation.
3. *Renaming:* rename the attributes in a relation.
4. *Set union:* compute the union of two relations; the relations must be *union-compatible.*
5. *Set difference:* compute the set difference of two relations; the relations must be *union-compatible.*
6. *Cross product:* compute the Cartesian product of two relations.

In addition to these operations, we include the *natural join* operation. The result of the natural join between relations *R*and *S* is the set of all combinations of tuples in *R* and *S* that are equal on their common attribute names. The common attributes only appear once in the result.

Natural join is expressible using the six fundamental operations, but a direct implementation for joins can reduce the need to use the (expensive) Cartesian product operation.

**Grammar**

The communication with the DBMS takes place using a domain-specific language. The grammar of *queries* in this language is as follows.

*query* ::= *relation-name* <- *expr* ;  
  
*relation-name* ::= *identifier*  
  
*identifier* ::= *alpha* { ( *alpha* | *digit* ) }  
  
*alpha* ::= a | … | z | A | … | Z | \_  
  
*digit* ::= 0 | … | 9  
  
*expr* ::= *atomic-expr*   
             | *selection*  
             | *projection*  
             | *renaming*  
             | *union*  
             | *difference*  
             | *product*  
             | *natural-join*  
  
*atomic-expr* ::= *relation-name* | ( *expr* )  
  
*selection* ::= select ( *condition* ) *atomic-expr*  
  
*condition* ::= *conjunction* { || *conjunction* }  
  
*conjunction* ::= *comparison* { && *comparison* }  
  
*comparison* ::= *operand* *op* *operand*  
                     | ( *condition* )  
  
*op* ::= == | != | < | > | <= | >=  
  
*operand* ::= *attribute-name* | *literal*  
  
*attribute-name* ::= *identifier*  
*literal* ::= intentionally left unspecified  
  
*projection* ::= project ( *attribute-list* ) *atomic-expr*  
  
*attribute-list* ::= *attribute-name* { , *attribute-name* }   
  
*renaming* ::= rename ( *attribute-list* ) *atomic-expr*  
  
*union* ::= *atomic-expr* + *atomic-expr*  
  
*difference* ::= *atomic-expr* - *atomic-expr*  
  
*product* ::= *atomic-expr* \* *atomic-expr*   
  
*natural-join* ::= *atomic-expr* JOIN *atomic-expr*

Queries generated from the above grammar compute new relations based on existing relations. Queries can also name those new relations. We need, however, some ways to create some initial relations (constituting a database), update the relations within the database, store the results of queries back to the database, and delete tuples from relations. We use the following commands for these purposes:

*command* ::= ( *open-cmd* | *close-cmd* | *write-cmd* | *exit-cmd* | *show-cmd* | *create-cmd* | *update-cmd* | *insert-cmd* |*delete-cmd* ) ;  
  
*open-cmd* ::== OPEN *relation-name*   
*close-cmd* ::== CLOSE *relation-name*   
*write-cmd* ::== WRITE *relation-name*   
*exit-cmd* ::== EXIT   
*show-cmd* ::== SHOW *atomic-expr*   
*create-cmd* ::= CREATE TABLE *relation-name* ( *typed-attribute-list* ) PRIMARY KEY ( *attribute-list* )  
  
*update-cmd* ::= UPDATE *relation-name* SET *attribute-name* = *literal* { , *attribute-name* = *literal* } WHERE *condition*   
  
*insert-cmd* ::= INSERT INTO *relation-name* VALUES FROM ( *literal* { , *literal* } )  
                       | INSERT INTO *relation-name* VALUES FROM RELATION *expr*  
  
*delete-cmd* ::= DELETE FROM *relation-name* WHERE *condition*  
  
*typed-attribute-list* ::= *attribute-name* *type* { , *attribute-name* *type* }  
*type* ::= VARCHAR ( *integer* ) | INTEGER  
*integer* ::= *digit* { *digit* }

A program in our data manipulation language (DML) is then defined as:

*program* ::= { ( *query* | *command* ) }

**Example**

CREATE TABLE animals (name VARCHAR(20), kind VARCHAR(8), years INTEGER) PRIMARY KEY (name, kind);

INSERT INTO animals VALUES FROM ("Joe", "cat", 4);

INSERT INTO animals VALUES FROM ("Spot", "dog", 10);

INSERT INTO animals VALUES FROM ("Snoopy", "dog", 3);

INSERT INTO animals VALUES FROM ("Tweety", "bird", 1);

INSERT INTO animals VALUES FROM ("Joe", "bird", 2);

SHOW animals;

dogs <- select (kind == "dog") animals;

old\_dogs <- select (age > 10) dogs;

cats\_or\_dogs <- dogs + (select (kind == "cat") animals);

CREATE TABLE species (kind VARCHAR(10)) PRIMARY KEY (kind);

INSERT INTO species VALUES FROM RELATION project (kind) animals;

a <- rename (aname, akind) (project (name, kind) animals);

common\_names <- project (name) (select (aname == name && akind != kind) (a \* animals));

answer <- common\_names;

SHOW answer;

WRITE animals;

CLOSE animals;

EXIT;

Note that we made a distinction between queries and commands in the grammar of the DML. The result of a query is a*view*. A view is not stored in the database. Rather, it is a temporary relation whose lifetime ends when a DML program finishes. So only the updates caused by the commands persist from one DML program execution to another.

The relations themselves should be saved in a file in plain ASCII text (**DO NOT use binary**), using the same DML described above (e.g., CREATE ... INSERT ... INSERT .... ). To make it simple, let us assume that each database file can only store one relation and the filename is the same as the relation name with the suffix ".db". To load a relation from a database file, use the OPEN command. Opening a nonexisting file will result in nothing. To create a new relation file, use the WRITE command (the filename will be by default "relationname.db"). To save all changes to the relation in a database file and close, use the CLOSE command.

To exit from the DML interpreter, use the EXIT command.

To print a certain relation or a view, use the SHOW command.

**PHASE II: DB Application**

The second phase of this project is to write a simple DB application written in the DML described above. The DB app will run on your DB engine. Since you will need to take user I/O and implement a custom control flow (conditional statements, loops, etc.), the DML described above alone is not enough. Instead of extending the DML to include such non-DB commands, you will use a host language (e.g., C++) to interact with your DBMS.

The host program will provide most of the user interface: displaying menus, taking user input, and showing results. Based on these user inputs, a custom query or command string can be generated and passed on to the DBMS to be parsed and executed..

You will need to develop some mechanism to transfer the results of queries to variables of the host program. An example protocol would be, for example, that if a DML program has a relation named answer, that relation is accessible in the host program in some suitable way. For example, its contents could be serialized into a container of typevector<string>, or the function that executes a DML program could accept an *STL output iterator* as a parameter, and write the result into the sequence indicated by that iterator.

Here is an example interaction:

string name;

cin << name;

string query = string(*""*) +

*"project (age) ( select (kind == \"dog\" && name == "* + name + *") animals )"*;

std::vector<string> result;

dbms.execute(animal\_database, query, std::back\_inserter(result));

The DB application itself will be of your own design, with the following requirements:

1. The application domain is open-ended: bank, web forum, warehouse app, point-of-sale, auto maintenance shop, etc., but it CANNOT BE one of the following: DVD or any kind of rental app, Library app, or PDA app.
2. The domain needs to be sufficiently complex. It should include at least three entities and two relations (a total of five relational tables, minimum). Thus, having a single table (e.g., a simple TODO list, etc.) is not acceptable.
3. The following command/query needs to be used at least once.
   * open, close, write, exit, show
   * create table, insert into, update, delete
   * select, project, +, - , \* (note: you have to think hard how to utilize all these in Phase II)
4. Provide a rough sketch of your application and its functionality in your design document. We will provide you with feedback in case it is too simple or too complex so that you can revise your plan early on.

As an example, here's a spec from a previous semester. This was for a DVD rental shop. Think about what should be an entity and what should be a relation.

1. Customer data: USER-ID, Firstname, Lastname, Phone Number
2. DVD data: Inventory number, DVD-ID, Title
3. Rental log: USER-ID, Inventory number, Check out date, Due date
4. Operations: Add new customer, Add new DVD, Remove customer, Remove DVD, Update customer, Update DVD, List customer, Search customer (by name, phone), List DVD, Search DVD (by ID, title), Search inventory, Check out DVD, Check in DVD, Show rental list by customer, Show customer list by DVD-ID.
5. The user interface can be a simple scrolling text with CLI (command-line interface) that prints the menu and takes user input from the keyboard.

Note that most of the code in the host language will be user interface, as you will be able to use the DML for most of the required operations.

**Requirements and Deliverables**

* Each team should maintain a development log (Google doc or similar) updated by the team members. Give access to the TA. This log will be graded. There is no designated format. We will check your daily progress. Update your log daily.
* Perform unit testing for major routines using a test driven development framework, such as Visual Studio test system, NUnit, Boost Test, and Googletest. Since you need to do this, engagement in test driven development is strongly encouraged!
* Expect to be called on to demo in the lab for deliverables 3 and 4.

1. Design documents: Follow the guidelines in [**Scott' Hackett's "How to Write an Effective Design Document" (Writing for a Peer Developer)**](http://blog.slickedit.com/2007/05/how-to-write-an-effective-design-document/). Include all four sections described in the guide.
   * The design document should cover both Phase I (DB engine) and Phase II (DB app).
   * Phase II documents should include ER diagram and corresponding relation schema, besides other things.
2. DBMS engine code: This will be the core DB function library, without the parser.
   * Upload your DBMS engine library code.
   * You should include a test application that tests all functionality (function calls relating to all queries, all commands).
   * Also submit the input and output (text file) from your test application.
   * **You may omit the file I/O code since it requires the parser.**
   * Theoretically, you should be able to build a DB app just using this library. (However, we will not do this -- you will use the DML command-line for the most part and not directly call the DBMS library functions from your final DB app.)
3. DML Parser integrated with DBMS Engine: Upload completed DBMS system code, including parser. It should be able to take input from the keyboard and accept or reject an arbitrary command or query. Accept syntactically correct commands or queries. Reject anything that violates the syntax.
   * Upload your parser code.
   * Submit the input and output (text file) of your parser.
   * Keep in mind how you will later integrate your parser code with the DB engine as you write the parser code.
   * Provide a CLI (command-line interface) where you can type in the commands and queries and it should be fully functional as a DB.
   * Include example inputs and actual outputs from your DBMS.
4. Final project code + DB application code + report. Provide all source code and a stand-alone executable. The report should include:
   * Updated version of your design document
   * Post-production notes (changes you had to make to your design and why, difficulties, solutions, lessons learned)
   * Individual work load distribution (percentage) -- how much of the total work did you and each of your teammates perform?
   * Development log.
   * A log of an example session with your DB application: show menu, take user input, application's output and messages, etc.
   * The following must be submitted individually:
     + Self Evaluation: honestly rate your performance on the scale of 1 to 5, 5 being the best. Briefly discuss how you contributed to the project.
     + Peer Evaluation: honestly rate all other team members on the scale of 1 to 5, 5 being best. For each member, briefly comment on their contribution. If you disagree with the work load distribution reported by the team, make this explicit. Provide your version, and written explanation of what went down.
     + Your team performance on this project will impact our decisions about team formation for future assignments. Please let us know if there are any signifiant issues about your current team. If you want to leave your team, or would not want to work with any particular members again, indicate so and provide justification in writing.

**Extra Credit (optional)**

In addition to building the full project as a team, partner with another team. Port your DB app, with the same functionality, to use the other team's Phase I code. Total value: 10 points.  
Include a short report about what issues arose and what you learned while porting your application to their DBMS.

**Submission**

All project deliverable submissions must be through [**https://csnet.cs.tamu.edu**](https://csnet.cs.tamu.edu/).

**Except for the individual peer evaluation deliverable, only one person from your team may submit, as a designated submitter. All other submissions will be ignored.**

See [**policies web page**](http://ecologylab.cse.tamu.edu/courses/programmingStudio/policies.html) for information on late penalties.