Introduction to Databases & SQL

CSCE 156 - Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

Lifetime of a program is short-lived
Applications perform small ephemeral operations
Can crash and die
Programs may be limited to sessions or even single requests
Need a way to persist data or program state across program lives
Databases provide such a means

Motivating Example I
Flat Files
Consider the following data, stored in a flat file:

<table>
<thead>
<tr>
<th>Course</th>
<th>Course Name</th>
<th>Student</th>
<th>NUID</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCE 156</td>
<td>Intro to CSII</td>
<td>Lou Reed</td>
<td>11112222</td>
<td><a href="mailto:reed@gmail.com">reed@gmail.com</a></td>
</tr>
<tr>
<td>CSCE 230</td>
<td>Computer Hardware</td>
<td>Student, J.</td>
<td>12345678</td>
<td><a href="mailto:jstudent@geocities.com">jstudent@geocities.com</a></td>
</tr>
<tr>
<td>CSCE 235</td>
<td>Discrete Math</td>
<td>Tom Waits</td>
<td>11223344</td>
<td><a href="mailto:twaits@email.com">twaits@email.com</a></td>
</tr>
<tr>
<td>NONE</td>
<td>Null</td>
<td>Tom Waits</td>
<td>11223344</td>
<td><a href="mailto:twaits@email.com">twaits@email.com</a></td>
</tr>
</tbody>
</table>

Table: Course enrollment data

Motivating Example II
Flat Files
Problems?
- Repetition of data
- Incomplete data
- Integrity of data
- Organizational problems: any aggregation requires processing all records
- Updating information is difficult (must enumerate all possible changes, side effects so that information is not lost)
- Formatting Issues
- Concurrency Issues

Relational Databases I
Key Aspects
Solution: Relational Database Systems (RDBS) or Relational Database Management System (RDMS)
- Stores data in tables
- Tables have a unique name and type description of its fields (integer, string)
- Each column stores a single piece of data (field)
- Each row represents a record (or object!)

Relational Databases II
Key Aspects
- Each row may have a unique primary key which may be
  - Automatically incremented
  - An external unique identifier: SSN, ISBN, NUID
  - Based on a combination of fields (Geographical data)
- Rows in different tables are related to each other through foreign keys
- Order of rows/columns is meaningless
Relational Databases III
Key Aspects

- Supports **Transactions**: an interaction or batch of interactions treated as one unit
- Constraints
  - Allowing or disallowing NULL
  - Disallowing “bad” values (ranges)
  - Enforcing formatting (capitalization, precision)
  - Limiting combinations of data fields

Relational Databases IV
Key Aspects

- **ACID principles**
  - **Atomicity**: Data transactions must be an all-or-nothing process
  - Atomic operation: not divisible or decomposable
  - **Consistency**: Transactions will retain a state of consistency
  - All constraints, triggers, cascades preserve a valid state after the transaction has completed
  - **Isolation**: No transaction interferes or is even aware of another; state transitions are equivalent to serial transactions
  - **Durability**: Once committed, a transaction remains so
    - Data is to be protected from catastrophic error (power loss/crash)

Commercial RDBMs

- MS Access :)
- MySQL (owned by Oracle, released under GNU GPL)
- PostgreSQL (true FOSS!)
- Informix (IBM)
- DB2 (IBM)
- SQLServer (Microsoft)
- Oracle Database
- SQLite

Others:
- Google’s BigTable → Spanner
- Apache Cassandra (Facebook)
- Amazon’s Dynamo

Advantages I

- Data is **structured** instead of “just there”
- Better organization
- Duplication is minimized (with proper **normalization**)
- Updating information is easier
- Organization of data allows easy access
- Organization allows aggregation and more complex information

Advantages II

- Data integrity can be enforced (data types and user defined constraints)
- Faster
- Scalable
- Security
- Portability
- Concurrency

Structured Query Language

We interact with RDBMs using **Structured Query Language** (SQL)

- Common language/interface to most databases
- Developed by Chamberlin & Boyce, IBM 1974
- Implementations may violate standards: portability issues
- Comments: -- or # (MySQL on cse)
- Create & manage tables: **CREATE, ALTER, DROP**
- Transactions: **START TRANSACTION, ROLLBACK, COMMIT**
Structured Query Language
CRUD

Basic SQL functionality: CRUD:
▶ Create – insert new records into existing tables
▶ Retrieve – get a (subset) of data from specific rows/columns
▶ Update – modify data in fields in specified rows
▶ Destroy – delete specific rows from table(s)

 Misc RDMS Issues I

Important aspects that will be omitted (good advanced topics):
Views – RDBSs allow you to create view of data; predefined select statements that aggregate (or limit) data while appearing to be a separate table to the end user
Triggers – SQL routines that are executed upon predefined events (inserts/updates) in order to create side-effects on the database

 Misc RDMS Issues II

Stored Procedures – SQL routines (scripts) that are available to the end user
Temp Tables – Temporary tables can be created to store intermediate values from a complex query
Nested Queries – SQL supports using subqueries to be used in other queries

 MySQL
Getting Started
You have access to a MySQL database on cse
▶ Database name: your cse login
▶ Password: see the system FAQ, http://cse.unl.edu/systems-faqs
▶ Option 1: Command Line Interface (CLI):
  > mysql -u cselogin -p
▶ Option 2: MySQL Workbench
  (http://www.mysql.com/products/workbench/)

Useful MySQL commands (not general SQL) to get you started:
▶ USE dbname;
▶ SHOW TABLES;
▶ DESCRIBE tablename;

Creating Tables

Syntax:
```
1 CREATE TABLE table_name ( 
2   field_name fieldType [options], 
3   ... 
4   PRIMARY KEY (keys) 
5 );
```

Options:
▶ AUTO_INCREMENT (for primary keys)
▶ NOT NULL
▶ DEFAULT (value)

Column Types

▶ VARCHAR(n) – variable character field (or CHAR, NCHAR, NVCHAR – fixed size character fields)
▶ INTEGER or INT
▶ FLOAT (or DOUBLE, REAL, DOUBLE PRECISION)
▶ DECIMAL(n,m), NUMERIC(n,m) (max total digits, max decimal digits)
▶ Date/Time functions: rarely portable
Creating Tables

Example

```sql
CREATE TABLE book (  
id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
title VARCHAR(255) NOT NULL,
author VARCHAR(255),
isbn VARCHAR(255) NOT NULL DEFAULT '',
dewey FLOAT,
num_copies INTEGER DEFAULT 0
);
```

Primary Keys

- Records (rows) need to be distinguishable
- A primary key allows us to give each record a unique identity
- At most one primary key per table
- Must be able to uniquely identify all records (not just those that exist)
- No two rows can have the same primary key value
- PKs can be one or more columns (composite key)
- Should not use/allow NULL values
- Can/should\(^1\) be automatically generated
- External identifiers should not be used
- Should use integers, not strings or floats

\(^1\)How to handle the foreign key problem?

Keys

- Tables can have multiple keys
- May be a combination of columns (composite key)
- NULL values are allowed
- Uniqueness is enforced (updates, inserts may fail)
- May be declared non-unique in which case it serves as an index
  (allows database lookup optimization)
- MySQL syntax:  
  ```sql
  KEY(column1, column2,...)
  ```

Indexes and Unique Constraints

- The keyword `INDEX` is often synonymous with `KEY`
- Keys don’t need to be unique
- We can make them unique by using `UNIQUE` (as a constraint or index)
- Syntax: `CONSTRAINT constraint_name UNIQUE INDEX(column_name)`
- Multicolumn uniqueness:  
  ```sql
  CONSTRAINT constraint_name UNIQUE INDEX(c1, c2)
  ```

Foreign Keys I

- Relations between records in different tables can be made with foreign keys
- A FK is a column that references a key (PK or regular key) in another table
- Inserts cannot occur if the referenced record does not exist
- Foreign Keys establish relationships between tables:
  - One-to-one (avoid)
  - One-to-many relations
  - Many-to-one
  - Many-to-Many relations: requires a Join Table

Foreign Keys II

- Table with FK (referencing table) references table with PK (referenced table)
- Deleting rows in the referenced table can be made to cascade to the referencing records (which are deleted)
- Cascades can be evil
- MySQL Syntax:  
  ```sql
  FOREIGN KEY (column) REFERENCES table(column)
  ```
Inserting Data

- Need a way to load data into a database
- Numerical literals
- String literals: use single quote characters
- Ordering of columns irrelevant

MySQL Syntax:

```
1 INSERT INTO table_name (c1, c2, ...) 
2 VALUES (value1, value2); 
```

Example:

```
1 INSERT INTO book (title, author, isbn) 
2 VALUES ('The Naked and the Dead', 'Normal Mailer', '978-0312265052'); 
```

Updating Data

- Existing data can be changed using the **UPDATE** statement
- Should be used in conjunction with *clauses*

Syntax:

```
1 UPDATE table SET c1 = v1, c2 = v2, ... WHERE (condition); 
```

Example:

```
1 UPDATE book SET author = 'Norman Mailer' 
2 WHERE isbn = '978-0312265052'; 
```

Deleting Data

- Data can be deleted using the **DELETE** statement
- Should be used in conjunction with *clauses*
- Unless you *really* want to delete everything

Syntax:

```
DELETE FROM table WHERE (condition) 
```

Example:

```
DELETE FROM book WHERE isbn = '978-0312265052'; 
```

Querying Data

- Data can be retrieved using the **SELECT** statement
- Syntax:

```
SELECT column1, column2... FROM table WHERE (condition); 
```

Example:

```
SELECT title AS bookTitle, 
num_copies AS numberOfCopies 
FROM book; 
```

WHERE Clause

- Queries can be quantified using the **WHERE** clause
- Only records matching the condition will be affected (updated, deleted, selected)
- Compound conditions can be composed using parentheses and:
  - **AND**
  - **OR**

Example:

```
SELECT * FROM book WHERE num_copies > 10 AND 
(title != 'The Naked and the Dead' OR author = 'Dr. Seuss'); 
```

To check nullity: **WHERE** *dewey IS NULL, WHERE dewey IS NOT NULL**
LIKE Clause

- VARCHAR values can be searched/partially matched using the LIKE clause
- Used in conjunction with the string wildcard, %
- Example: `SELECT * FROM book WHERE author LIKE ‘%Mailer%’;`
- Example: `SELECT * FROM book WHERE isbn LIKE ‘123%’;`

ORDER BY Clause

- In general, the order of the results of a SELECT clause is irrelevant
- Nondeterministic, not necessarily in any order
- To impose an order, you can use ORDER BY
- Can order along multiple columns
- Can order descending or ascending (DESC, ASC)
- Example: `SELECT * FROM book ORDER BY title;`
- Example: `SELECT * FROM book ORDER BY author DESC, title ASC`

IN Clause

- The IN clause allows you to do conditionals on a set of values
- Example:
  ```
  1. SELECT * FROM book WHERE isbn IN ('978-0312265052', '789-65486548', '681-0654895052');
  ```
- May be used in conjunction with a nested query:
  ```
  1. SELECT * FROM book WHERE isbn IN
  2. (SELECT isbn FROM book WHERE num_copies > 10);
  ```

Aggregate Functions

- Aggregate functions allow us to compute data on the database without processing all the data in code
- COUNT provides a mechanism to count the number of records
- Example:
  ```
  SELECT COUNT(*) AS numberOfTitles FROM book;
  ```
- Aggregate functions: MAX, MIN, AVG, SUM
- Example:
  ```
  SELECT MAX(num_copies) FROM book;
  ```
- Using nested queries:
  ```
  1. SELECT * FROM book WHERE num_copies =
  2. (SELECT MAX(num_copies) FROM book);
  ```
- NULL values are ignored/treated as zero

GROUP BY clause

- The GROUP BY clause allows you to project data with common values into a smaller set of rows
- Used in conjunction with aggregate functions to do more complicated aggregates
- Example: find total copies of all books by author:
  ```
  1. SELECT author, SUM(num_copies) AS totalCopies
  2. FROM book GROUP BY author;
  ```
- The projected data can be further filtered using the HAVING clause:
  ```
  1. SELECT author, SUM(num_copies) AS totalCopies
  2. FROM book GROUP BY author HAVING totalCopies > 5;
  ```
- HAVING clause evaluated after GROUP BY which is evaluated after any WHERE clause

GROUP BY clause I

Example

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Num_copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naked and the Dead</td>
<td>Norman Mailer</td>
<td>10</td>
</tr>
<tr>
<td>Dirk Gently’s Holistic Detective Agency</td>
<td>Douglas Adams</td>
<td>4</td>
</tr>
<tr>
<td>Barbary Shore</td>
<td>Norman Mailer</td>
<td>3</td>
</tr>
<tr>
<td>The Hitchhiker’s Guide to the Galaxy</td>
<td>Douglas Adams</td>
<td>2</td>
</tr>
<tr>
<td>The Long Dark Tea-Time of the Soul</td>
<td>Douglas Adams</td>
<td>1</td>
</tr>
<tr>
<td>Ender’s Game</td>
<td>Orson Scott Card</td>
<td>7</td>
</tr>
</tbody>
</table>
### GROUP BY clause II

Example:

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>num_books</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naked and the Dead</td>
<td>Norman Mailer</td>
<td>10</td>
</tr>
<tr>
<td>Barbary Shore</td>
<td>Douglas Adams</td>
<td>4</td>
</tr>
<tr>
<td>Dirk Gently’s Holistic Agency</td>
<td>Douglas Adams</td>
<td>2</td>
</tr>
<tr>
<td>The Hitchhiker’s Guide to the Galaxy</td>
<td>Douglas Adams</td>
<td>1</td>
</tr>
<tr>
<td>The Long Dark Tea-Time of the Soul</td>
<td>Orson Scott Card</td>
<td>7</td>
</tr>
</tbody>
</table>

Projection & aggregation:

<table>
<thead>
<tr>
<th>author</th>
<th>num_books</th>
</tr>
</thead>
<tbody>
<tr>
<td>Norman Mailer</td>
<td>13</td>
</tr>
<tr>
<td>Douglas Adams</td>
<td>7</td>
</tr>
<tr>
<td>Orson Scott Card</td>
<td>7</td>
</tr>
</tbody>
</table>

### INNER JOIN I

- Most common type of join
- Combines rows of table A with rows of table B for all records that satisfy some predicate
- Predicate provided by the `ON` clause
- May omit `INNER`
- May provide join predicate in a `WHERE` clause

### INNER JOIN II

```sql
1 SELECT * FROM book b
2 INNER JOIN person p ON b.author = p.name
3
4 SELECT * FROM book b
5 JOIN ... p.name
10
11 SELECT s.student_id,s.last_name,e.address FROM student s
12 JOIN email e ON s.student_id = e.student_id;
```

### INNER JOIN

Example

<table>
<thead>
<tr>
<th>student_id</th>
<th>last_name</th>
<th>first_name</th>
<th>email_id</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>Castro</td>
<td>Starlin</td>
<td>1111</td>
<td><a href="mailto:rsnandberg@cubbies.net">rsnandberg@cubbies.net</a></td>
</tr>
<tr>
<td>5678</td>
<td>Rizzo</td>
<td>Anthony</td>
<td>1112</td>
<td><a href="mailto:rsnandberg@cubbies.net">rsnandberg@cubbies.net</a></td>
</tr>
<tr>
<td>1122</td>
<td>Sveum</td>
<td>Dale</td>
<td>1113</td>
<td><a href="mailto:rsnandberg@cubbies.com">rsnandberg@cubbies.com</a></td>
</tr>
<tr>
<td>9988</td>
<td>Sandberg</td>
<td>Ryme</td>
<td>1114</td>
<td><a href="mailto:rsnandberg@cubbies.com">rsnandberg@cubbies.com</a></td>
</tr>
</tbody>
</table>

(a) Student Table

<table>
<thead>
<tr>
<th>student_id</th>
<th>last_name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>Castro</td>
<td><a href="mailto:number13@cubs.com">number13@cubs.com</a></td>
</tr>
<tr>
<td>5678</td>
<td>Rizzo</td>
<td><a href="mailto:rizzo@cubs.com">rizzo@cubs.com</a></td>
</tr>
<tr>
<td>9988</td>
<td>Sandberg</td>
<td><a href="mailto:rsandberg@cubbies.net">rsandberg@cubbies.net</a></td>
</tr>
<tr>
<td>9988</td>
<td>Sandberg</td>
<td><a href="mailto:rsandberg@unl.edu">rsandberg@unl.edu</a></td>
</tr>
</tbody>
</table>

(b) Email Table

Table : Joined tables

### LEFT OUTER JOIN I

- Left Outer JOIN joins table A to table B, preserving all records in table A
- For records in A with no matching records in B: `NULL` values used for columns in B
- `OUTER` may be omitted
### LEFT OUTER JOIN II

```sql
1. SELECT * FROM book b
2. LEFT OUTER JOIN person p ON b.author = p.name
3. SELECT * FROM book b
4. LEFT JOIN person p ON b.author = p.name
5. SELECT s.student_id, s.last_name, e.address FROM student s
6. LEFT JOIN email e ON s.student_id = e.student_id;
```

### LEFT OUTER JOIN III

<table>
<thead>
<tr>
<th>student_id</th>
<th>last_name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>Castro</td>
<td><a href="mailto:number13@cubs.com">number13@cubs.com</a></td>
</tr>
<tr>
<td>5678</td>
<td>Rizzo</td>
<td><a href="mailto:rizzo@cubs.com">rizzo@cubs.com</a></td>
</tr>
<tr>
<td>9988</td>
<td>Sandberg</td>
<td><a href="mailto:rsandberg@cubbies.net">rsandberg@cubbies.net</a></td>
</tr>
<tr>
<td>9988</td>
<td>Sandberg</td>
<td><a href="mailto:rsandberg@unl.edu">rsandberg@unl.edu</a></td>
</tr>
<tr>
<td>1122</td>
<td>Sveum</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table: Left-Joined Tables

### Other Joins


- Many records may have the same column value
- May want to query only the unique values
- May only want to count up the number of unique values
- SQL keyword: `DISTINCT`

```sql
1. SELECT DISTINCT author FROM book;
2. SELECT COUNT(DISTINCT author) FROM book;
```

### DISTINCT Clause

- Many records may have the same column value
- May want to query only the unique values
- May only want to count up the number of unique values
- SQL keyword: `DISTINCT`

### Database Design

- **Table Design**
  - Identify all “entities” in different tables
  - Ask “what defines an entity” to determine the columns and their types
  - Properly define whether or not a column should be allowed to be null and/or defaults
  - Each table should have a primary key
  - Relations between tables should be identified and defined with foreign keys
  - Security concerns: don’t store sensitive information (passwords) in plaintext

### Normalization I

Normalizing a database is the process of separating data into different tables to reduce or eliminate data redundancy and reduce anomalies (preserve integrity) when data is changed. It also reduces the amount of book keeping necessary to make such changes.

- **1 Normal Form**: the domain of each attribute in each table has only atomic values: each column value represents a single value
  - Example: Allowing multiple email records for one Person
  - Storing these as (say) a CSV in one column is a violation of 1NF
  - Hardcoding a fixed number of columns (Email1, Email2, Email3) for multiple values is a violation of 1NF
  - Separating records out into another table and associating them via a FK conforms to 1NF
Normalization II

- 2 Normal Form: 1NF and no non-prime attribute is dependent on any proper subset of prime attributes
  - Mostly relevant for tables with composite PKs (multiple columns)
  - If another, non-prime column is dependent only on a subset of these, its not 2NF
  - Must split it out groups of data into multiple tables and relate them through FKs so that non-prime column is dependent only on the subset of prime columns
  - Example: Employee-Title-Address (must split into Employee-Title and Employee-Address)

Normalization III

- 3 Normal Form: 2NF and no non-prime column is transitively dependent on the key
  - No non-prime column may depend on another non-prime column
  - Example: Storing a price-per-unit, quantity, and total (total can be derived from the other two)
  - Example: CourseOfferingId-CourseId-InstructorId-Instructor Name (name should be derivable from the ID via another table)
  - Example: OrderId-CustomerId-CustomerName (OrderId is the PK, CustomerId is FK, CustomerName should be derivable from CustomerId)

Primary Keys

- Should be integers (not varchars nor floats)
- Best to allow the database to do key management (auto increment)
- Should not be based on external identifiers that are not controlled by the database (NUIDs, SSNs, etc.)
- Be consistent in naming conventions (tableNameId or table_name_id)

Good Practice Tip 1

Use consistent naming conventions

- Short, simple, descriptive names
- Avoid abbreviations, acronyms
- Use consistent styling
  - Table/field names: Lower case, underscores, singular or Camel case, pluralized
- Primary key field: tableNameId or table_name_id
- Use all upper-case for SQL commands
- Foreign key fields should match the fields they refer to
- End goal: unambiguous, consistent, self-documenting

Good Practice Tip 2

Ensure Good Data Integrity

Data can break code, code should not break data.

- Data/databases are a service to code
- Different code, different modules can access the same data
- The database does not use the code!
- Should do everything you can to prevent bad code from harming data (constraints, foreign & primary keys, etc.)
- Database is your last line of defense against bad code
Good Practice Tip 3
Keep Business Logic Out!

- Databases offer “programming functionality”
- Triggers, cascades, stored procedures, etc.
- Use them sparingly!!!
- RDMSs are for the management and storage of data, not processing
- DBAs should not have to be Application Programmers, and vice versa

Good Practice Tip 4
Dealing with Inheritance

- Relational data model ≠ Object model
- Relational Databases do not have an inheritance mechanism
- Several strategies exist for dealing with class hierarchy with different state
- Single table mapping: all state is on one table (recommended) with a discriminator column (irrelevant columns made nullable)
- Joined tables: each subclass has a “sub” table joined via a foreign key from the “super” table
- Table-per-class: each table represents a distinct class (repeated data)

Current Trends I

- Additional Data-layer abstraction tools (JPA, ADO for .NET)
- XML-based databases (using XQuery)
- RDBMs are usually tuned to either small, frequent read/writes or large batch read-transactions
- Nature and scale of newer applications does not fit well with traditional RDBMs
- Newer applications are data intensive:
  - Indexing a large number of documents
  - High-traffic websites
  - Large-scale delivery of multimedia (streaming video, etc.)
  - Search applications, data mining
  - New tools generating HUGE amounts of data (biological, chemical, sensor networks, etc)

Current Trends II

- Newer (revived) trend: NoSQL
  - Non-relational data
  - Sacrifices rigid ACID principles for performance
  - Eventual consistency
  - Limited transactions
  - Emphasis on read-performance
  - Simplified Key-Value data model
  - Simple interfaces (associative arrays)
- Example: Google’s BigTable (key: two arbitrary string (keys) to row/column with a datetime, value: byte array)

Current Trends III

- Even newer trend: NewSQL
  - NoSQL too extreme: need scalability, but not at the cost of no consistency or other ACID principles
  - Large amount of data, large number of transactions, but
  - Short-lived, small subset of data access, repeated
  - Sacrifice durability and concurrency
  - Examples: Google’s Spanner, new MySQL engines (MemSQL)
Designing A Database

**Exercise** Design a database to support a course roster system. The database design should be able to model Students, Courses, and their relation (ability of students to enroll in courses). The system will also need to email students about updates in enrollment, so be sure your model is able to incorporate this functionality.

![A normalized database design, ER diagram generated in MySQL Workbench](image)

**End Result**

- Pieces of data are now organized and have a specific type
- No duplication of data
- Entities are represented by IDs, ensuring identity (Tom Waits is now the same as Tom Waits)
- Data integrity is enforced (only one NUID per Student)
- Relations are well-defined
  - A student has email(s)
  - A course has student(s) and a student has course(s)

**Data from flat file**

<table>
<thead>
<tr>
<th>Student ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>NUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom</td>
<td>Waits</td>
<td>11223344</td>
</tr>
<tr>
<td>2</td>
<td>Lou</td>
<td>Reed</td>
<td>11112222</td>
</tr>
<tr>
<td>5</td>
<td>John</td>
<td>Student</td>
<td>12345678</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Email ID</th>
<th>Student ID</th>
<th>Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td><a href="mailto:reed@gmail.com">reed@gmail.com</a></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td><a href="mailto:tomwaits@hotmail.com">tomwaits@hotmail.com</a></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td><a href="mailto:twaits@email.com">twaits@email.com</a></td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td><a href="mailto:jstudent@geocities.com">jstudent@geocities.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42123</td>
<td>CSCE 156</td>
<td>Computer Science II</td>
</tr>
<tr>
<td>12333</td>
<td>CSCE 230</td>
<td>Computer Hardware</td>
</tr>
<tr>
<td>11132</td>
<td>CSCE 235</td>
<td>Discrete Mathematics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enrollment ID</th>
<th>Student ID</th>
<th>Course ID</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>5</td>
<td>11132</td>
<td>Fall 2011</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>42123</td>
<td>Fall 2011</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>42123</td>
<td>Fall 2011</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11132</td>
<td>Fall 2011</td>
</tr>
<tr>
<td>29</td>
<td>5</td>
<td>42123</td>
<td>Spring 2011</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>12333</td>
<td>Fall 2011</td>
</tr>
</tbody>
</table>

**SQL Demo and Exercises**

Make queries as specified for the video game database described in the figure.