Introduction to Databases & SQL

Computer Science II – Introduction to Computer Science II

Christopher M. Bourke
cbourke@cse.unl.edu

Introduction to Databases & SQL

- 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

Consider the following data, stored in a *flat file*:

Course	Course Name	Student	NUID	Email
		waits, tom	11223344	tomwaits@hotmail.com
CSCE 156	Intro to CSII	Lou Reed	11112222	reed@gmail.com
CSCE 156	Intro to CSII	Tom Waits	11223344	twaits@email.com
CSCE 230	Computer Hardware	Student, J.	12345678	jstudent@geocities.com
CSCE 156	Intro to CSII	Student, John	12345678	jstudent@geocities.com
CSCE 230	Computer Hardware	Student, J.	12345678	jstudent@geocities.com
CSCE 235	Discrete Math	Student, John	12345678	jstudent@geocities.com
CSCE 235	Discrete Math	Tom Waits	11223344	twaits@email.com
NONE	Null	Tom Waits	11223344	twaits@email.com

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

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

- 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 $\rightarrow$ Spanner
- Apache Cassandra (Facebook)
- Amazon's Dynamo

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 $\rightarrow$ 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`

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/faq`
- Option 1: Command Line Interface (CLI):
  ```
  >mysql -u cselogin -p
  ```
- Option 2: MySQL Workbench (`http://cse.unl.edu/account`)

Useful MySQL commands (not general SQL) to get you started:

- `USE dbdname;`
- `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
- MySQL: see `http://dev.mysql.com/doc/refman/5.0/en/date-and-time-functions.html`

Creating Tables

Example – Old School Style


```
1  CREATE TABLE book (
2    id      INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
3    title   VARCHAR(255) NOT NULL,
4    author  VARCHAR(255),
5    isbn    VARCHAR(255) NOT NULL DEFAULT '',
6    dewey   FLOAT,
7    num_copies INTEGER DEFAULT 0
8  );
```

Creating Tables

Example – Modern Style


```
1  create table Book (
2    bookId integer primary key auto_increment not null,
3    title  varchar(255) not null,
4    author varchar(255),
5    isbn   varchar(255) not null default '',
6    dewey  float,
7    numCopies integer default 0
8  );
```

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:
  `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:
  ```
  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:

```
FOREIGN KEY (column) REFERENCES table(column)
```

Quick Exercise I

(Re)write SQL to define two tables: one for authors and one for books. Simplify the book table, but include an ISBN as well as a constraint to keep it unique. Model the fact that an author may write more than one book.

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',
3            '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, ...
2    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:

```
1  SELECT author, title FROM book
2    WHERE isbn = '978-0312265052';
```

- Can select *all* columns by using the `*` wildcard:

```
SELECT * FROM book
```

- Names of the columns are part of the database
- SQL allows us to "rename" them in result of our query using *aliasing*
- Syntax: `column_name AS column_alias`
- Sometimes necessary if column has no name (aggregates)

```
1  SELECT title      AS bookTitle,
2         num_copies AS numberOfCopies
3  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`

```
1  SELECT * FROM book
2    WHERE num_copies > 10 AND
3    (title != 'The Naked and the Dead'
4      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 isbn LIKE '123%';
```

- Example:

```
SELECT * FROM book WHERE author LIKE '%Mailer%';
```

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',
2    '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);
```

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
```

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 content:

title	author	num_copies
Naked and the Dead	Norman Mailer	10
Dirk Gently's Holistic Detective Agency	Douglas Adams	4
Barbary Shore	Norman Mailer	3
The Hitchhiker's Guide to the Galaxy	Douglas Adams	2
The Long Dark Tea-Time of the Soul	Douglas Adams	1
Ender's Game	Orson Scott Card	7

GROUP BY clause II

Example

Grouping:

title	author	num_copies
Naked and the Dead	Norman Mailer	10
Barbary Shore	Norman Mailer	3
Dirk Gently's Holistic Detective Agency	Douglas Adams	4
The Hitchhiker's Guide to the Galaxy	Douglas Adams	2
The Long Dark Tea-Time of the Soul	Douglas Adams	1
Ender's Game	Orson Scott Card	7

Projection & aggregation:

author	num_copies
Norman Mailer	13
Douglas Adams	7
Orson Scott Card	7

JOINs

A *join* is a clause that combines records from two or more tables.

- Result is a set of columns/rows (a "table")
- Tables are joined by shared values in specified columns
- Common to join via Foreign Keys
- Table names can be aliased for convenience
- Types of joins we'll look at:
  - (INNER) JOIN
  - LEFT (OUTER) JOIN
- Other types of joins: Self-join, cross join (cartesian product), right outer joins, full outer joins

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


```
1  SELECT * FROM book b
2      INNER JOIN person p ON b.author = p.name
3
4  SELECT * FROM book b
5      JOIN person p ON b.author = p.name
6
7  SELECT *
8  FROM book b, person p
9  WHERE b.author = 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

student_id	last_name	first_name
1234	Castro	Starlin
5678	Rizzo	Anthony
1122	Sveum	Dale
9988	Sandberg	Ryne

(a) Student Table

email_id	student_id	address
1111	9988	rsandberg@cubbies.net
1112	9988	rsberg@unl.edu
1113	5678	rizzo@cubs.com
1114	1234	number13@cubs.com

(b) Email Table

student_id	last_name	adddress
1234	Castro	number13@cubs.com
5678	Rizzo	rizzo@cubs.com
9988	Sandberg	rsandberg@cubbies.net
9988	Sandberg	rsberg@unl.edu

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


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

LEFT OUTER JOIN III

student_id	last_name	adddress
1234	Castro	number13@cubs.com
5678	Rizzo	rizzo@cubs.com
9988	Sandberg	rsandberg@cubbies.net
9988	Sandberg	rsberg@unl.edu
1122	Sveum	NULL

Table: Left-Joined Tables

Nice tutorial: `http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins`

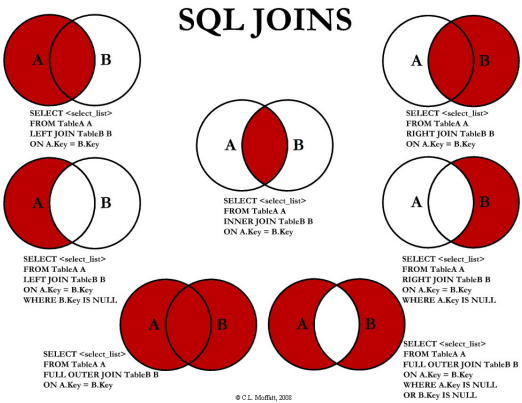


Figure: Types of Joins

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`

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

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 anomolies (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)

Normalization IV

- Bottom line: 3NF is common sense; design your database to be normalized to begin with
- Other advanced forms (BCNF, 4NF, 5NF, 6NF)
- Every non-key attribute must provide a fact about the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd[2]
- Good tutorials: `http://support.microsoft.com/kb/283878` `http://phlonx.com/resources/nf3/` (available on BB)

[2] Edgar Codd, inventor of the relational model, 1970–1971

Primary Keys

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
- Demarcation of responsibility
- DBAs should not have to be Application Programmers, and vice versa

Good Practice Tip 4

Dealing with Inheritance

- Relational data model $\neq$ 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)

Other considerations

- Hard vs. Soft deletes: many times it is preferable not to permanently delete data; instead an active/inactive flag is defined (care must be taken up “new” inserts)

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

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.

Designing A Database

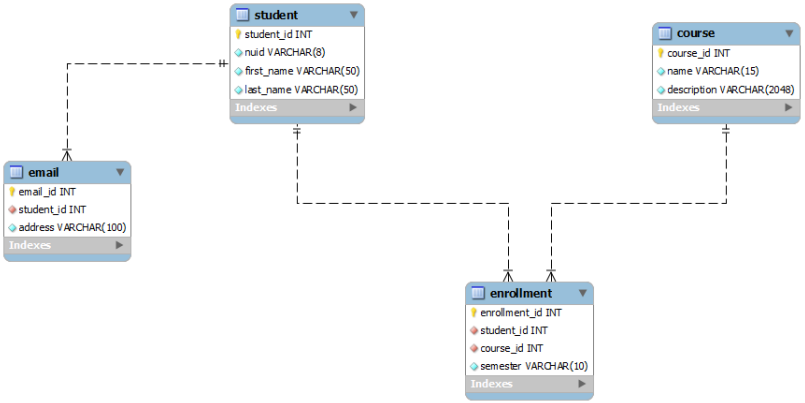
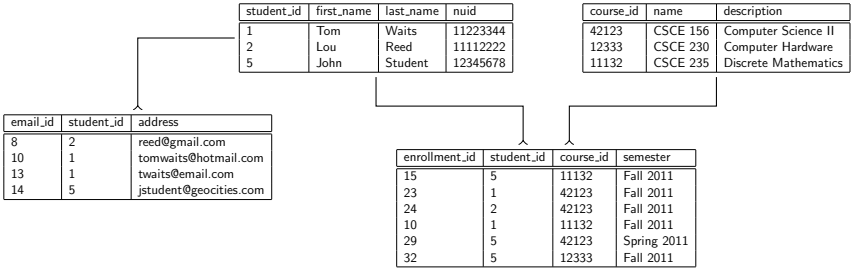


Figure: A normalized database design, ER diagram generated in MySQL Workbench

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 t. 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



student_id	first_name	last_name	nuid
1	Tom	Waits	11223344
2	Lou	Reed	11112222
5	John	Student	12345678

course_id	name	description
42123	CSCE 156	Computer Science II
12333	CSCE 230	Computer Hardware
11132	CSCE 235	Discrete Mathematics

email_id	student_id	address
8	2	reed@gmail.com
10	1	tomwaits@hotmail.com
13	1	twaits@email.com
14	5	jstudent@geocities.com

enrollment_id	student_id	course_id	semester
15	5	11132	Fall 2011
23	1	42123	Fall 2011
24	2	42123	Fall 2011
10	1	11132	Fall 2011
29	5	42123	Spring 2011
32	5	12333	Fall 2011

SQL Demo and Exercises

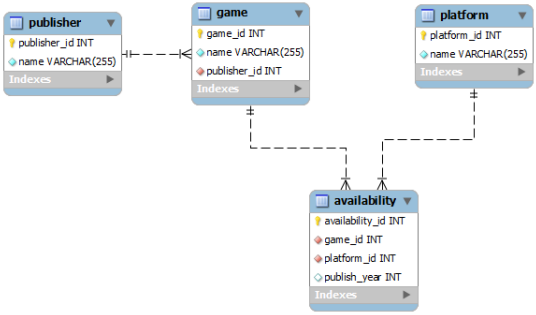


Figure: Video Game Database

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

CSE Gotchas

- Default character set/collation may be case *insensitive*
- `SELECT @@character_set_database, @@collation_database;`
- Case sensitive: use `latin1_general_cs` or `utf8_bin`
- Default engine? `show engines;`
- Be sure to use InnoDB

Easiest workarounds:

```
1  create table Foo (
2    ...
3  )Engine=InnoDB,COLLATE=latin1_general_cs;
```