CS143: Database Systems

Professor Cho

Thilan Tran

Winter 2021

Contents

CS143: Database Systems

- a **database management system (DBMS)** is a way to manage and store data:
  - how does a database differ from a spreadsheet software like Excel?
    - expected to *efficiently* scale to a *massive* amount of data, without suffering
    - expected to persist the data
    - expected to provide secured and safe access to data
    - expected to be conveniently used by a large number of clients at a time
  - program data is not assumed to entirely reside in main memory RAM, but instead in disk
    - this leads to utilization of different data structures
- database architecture:
  1. disk for data (sometimes stored in main memory, if data can fit)
  2. OS
  3. DBMS engine
     - database system may access disk through OS, or directly through raw IO
  4. API
     - eg. standard APIs like JDBC (Java), ODBC (Microsoft)
  5. app or CLI
  - downloading a DBMS software like MySQL installs parts 3, 4, and CLI
- popular DBMS software:
  - relational:
    - open source: MySQL, PostgreSQL
    - closed source: Oracle, Microsoft SQL, IBM DB2
  - non-relational (NoSQL):
    - MongoDB, Spark
- five steps in database construction:
  1. domain analysis
     - captured in an entity-relationship (ER) model or using unified modeling language (UML)
  2. database design
     - normalization theory
  3. table creation
     - uses a data definition language (DDL)
  4. load
     - using SQL or bulk load
  5. query and update
     - using data manipulation language (DML)

Data Models

- what is a data model? why do we need it?
  - eg. the human brain acts as a data model
    - remembers information, easily accessible and understandable data
  - how do we translate the memory of a human brain in computers?
    - computers speak in binary, only stores data in a very specific, concrete format
  - a **data model** is a very specific way to model or represent data in computers:
- types of data models include graph, tree, relational models:
  - a **graph model** (or network model) has nodes, edges, and labels:
    - eg. the most natural way to represent airline flights
    - initially the most popular data model
    - more flexible and semistructured, often used for JSON data eg. MongoDB
  - a **tree model** (or hierarchical model) is a graph arranged as a tree, eg. company hierarchies
  - in a **relational model**, all data is instead represented as a set of tables:
    - introduced in 1970 by Codd, and completely revolutionized the database field
    - in graph and tree models, accessing and modifying data is not as efficient:
      - instead, represent all data through relationships and store them in tables, like a spreadsheet
      - a downside to the relational model is the rigidity of its schemas
    - in mathematics, tables are expressed as **relations** eg. tuples, triplets, etc.
    - now, the relational model is the most popular data model used in DBs

Relational Models

- relational model terminology:
  - **relations** ie. tables contain many **tuples** ie. rows, each with various **attributes** ie. columns
  - each attribute has a **domain** ie. type
  - a **schema** is the structure of relations in a database:
    - includes properties like the relation name, attribute names, domains

    - eg. `Student(sid: int, name: str, addr: Addr, age: int, GPA: int)`
  - the **instance** is the actual data populating a relation that conforms to some schema
    - ie. schema is the variable type, and the instance is the variable value
  - **keys** are a set of attributes that *uniquely* identify a tuple in a relation:
    - multiple keys are possible
    - eg. in `Course(dept, cnum, sec, unit, instructor, title)` , the keys may be the department, course number, and section number
      - or alternatively department, section, and title if course numbers are repeated
    - generally, assuming the set of attributes for keys is the minimum set
      - in the worst case, with no duplicates, a relation's keys are all its attributes
  - more specifically, different types of keys:
    1. a **super key** is any key
    2. a **candidate key** is a key with the minimum number of attributes
    3. a **primary key** is a candidate key chosen to be used as the main key for a relation
       - shouldn't have null values
  - **null values** are used to indicate not applicable, uninitialized values, etc.:
    - are compatible with every type, unless otherwise explicitly defined
    - leads to complications, eg. comparisons in conditional queries
      - thus DBMS may return unexpected answers
    - requires **3-valued logic** where every condition can be true, false, or unknown:
      - need concrete rules to deal with null and unknown values
      - adds increased implementation and execution complexity
- name scopes of relational models:
  - the name of a relation is unique across relations
  - the names of attributes are uniue in a table
    - thought there cane be the same attribute names in different tables
- set semantics in relational models:
  - in a set, there are no duplicate elements, and order is unimportant
  - thus in the relational model:
    1. no duplicate tuples are allowed
       - but SQL allows duplicate tuples for practical reasons
    2. attribute order does not matter

Relational Algebra

- relational query langauges include:
  - formal languages eg. relational algebra, relational calculus, datalog
  - practical languages eg. SQL, Quel, QBE
- in relational algebra:
  - inputs and outputs are both relations, so **piping** is possible
  - set semantics are followed, so duplicates are automatically eliminated

Table 1: Students

sid	name	addr	age	GPA
301	John	183 Westwood	19	2.1
303	Elaine	301 Wilshire	17	3.9
401	James	183 Westwood	17	3.5
208	Esther	421 Wilshire	20	3.1

Table 2: Classes

dept	cnum	sec	unit	title	instructor
CS	112	01	03	Modeling	Dick Muntz
CS	143	01	04	DB Systems	John Cho
EE	143	01	03	Signals	Dick Muntz
ME	183	02	05	Mechanics	Susan Tracey

Table 3: Enrollments

sid	dept	cnum	sec
301	CS	112	01
301	CS	143	01
303	EE	143	01
303	CS	112	01
401	CS	112	01

- queries can be done using the formal relational algebra language
  - consider the following example queries using Table 1, Table 2, and Table 3

1. Get all the students:

$$Student$$

- to get all the tuples from a relation, just write the name of the relation

2. Get all students with age > 18:

$$\sigma_{age<18}(Student)$$

- the select operator $\sigma_C(R)$ filters based on boolean expression C
  - R can be either a relation or a result from another operator

3. Get all students with GPA > 3.7 and age < 18:

$$\sigma_{GPA>3.7 \wedge age<18}(Student)$$

4. Get student ID and GPA of all students:

$$\Pi_{sid,GPA}(Student)$$

- need a different operator
- the project operator $\Pi_A(R)$ filters column-wide based on attributes A
  - returns a new set of *columns*

5. Get all departments offering a class:

$$\Pi_{dept}(Class)$$

- due to set semantics, does not return two elements for "CS"

6. Get student ID and GPA of students with age < 18:

$$\Pi_{sid,GPA}(\sigma_{age<18}(Student))$$

- composing operators next to each other
- however, for projection, it is not useful to compose projections

- the **cross product** ie. Cartesian product operator in relational algebra:
  - $R \times S = \{t | t = (r, s) \ \forall \ r \in R, s \in S\}$ concatenates every tuple of each relation together:
    - if $|R| = r$ and $|S| = s$, $|R \times S| = rs$
    - eg. $R \times S$ contains $a_1 b_2, a_1 b_2, \ldots, a_n b_m$
  - ie. creates one output per every pair of input tuples
  - useful when combining tuples from multiple relations

7. Get the names of students who take CS classes:

$$\Pi_{name}(\sigma_{dept='CS'}(\sigma_{Student.sid=Enroll.sid}(Student \times Enroll)))$$

$$\Pi_{name}(\sigma_{dept='CS' \wedge Student.sid=Enroll.sid}(Student \times Enroll))$$

- but this cross product is quite expensive, so we can change the ordering to improve efficiency

$$\Pi_{name}(\sigma_{Student.sid=Enroll.sid}(Student \times \sigma_{dept='CS'}(Enroll)))$$

- equivalently, using the natural join operator:

$$\Pi_{name}(\sigma_{dept=\text{'CS'}}(Student \bowtie Enroll))$$

- the **natural join** operator $\bowtie$ is used to join two tables *naturally*:
  - filters the Cartesian product by the tuples based on the equality conditions of *all* shared attributes
  - equivalent to:

$$R \bowtie S = \sigma_{\langle equal \;\; shared \;\; attributes \rangle}(R \times S)$$

Table 4: $Class \bowtie Enroll$

dept	cnum	sec	unit	title	ins	sid	dept	cnum	sec
CS	112	01	03	Modeling	Dick Muntz	301	CS	112	01
CS	112	01	03	Modeling	Dick Muntz	303	CS	112	01
CS	112	01	03	Modeling	Dick Muntz	401	CS	112	01
CS	143	01	03	DB Systems	John Cho	301	CS	143	01
EE	143	01	03	Signals	Dick Muntz	303	EE	143	01

8. Get the names of students who take classes offered by `'Dick Muntz'` :

$$\Pi_{name}(Student \bowtie (Enroll \bowtie \sigma_{inst=\text{'Dick Muntz'}}(Class)))$$

9. Get the names of student pairs who live at the same address:

$$\Pi_{Student.name,S.name}(\sigma_{C_1 \wedge C_2}(Student \times \rho_S(Student)))$$

   - where $C_1 = (Student.addr = S.addr)$ and $C_2 = Student.sid > S.sid$
   - crossing a relation with itself is also known as **self-join**:
     - need to use the rename operator $\rho_{S(A)}$
     - renames table to S and optionally, specific attribute to A
   - the comparison check on `sid` prevents duplicates such as
     - `('John', 'John'), ('John', 'James'), ('James', 'John')`

10. Get all students and instructor names:

$$\Pi_{name}(Student) \cup \rho_{Person(name)}(\Pi_{inst}(Class))$$

    - when using union, the schema must be equal between the relations
    - ie. the attribute names have to match up
    - no duplicate tuples in the result

11. Get all the courses (department, number, section) that no one takes:

$$\Pi_{dept,cnum,sec}(Class) - \Pi_{dept,cnum,sec(Enroll)}$$

- easier to express this query using its complement
- using the set difference operator $R - S$

12. Get instructor names who teach both CS and EE courses:

$$\Pi_{inst}(\sigma_{dept=\text{'CS'}}(Class)) \cap \Pi_{isnt}(\sigma_{dept=\text{'EE'}}(Class))$$

- when using intersection, the schema should again be the same
- $R \cap S = R - (R - S)$

13. Get IDs of students who did not take any CS class:

$$\Pi_{sid}(Student) - \Pi_{sid}\sigma_{dept=\text{'CS'}}(Enroll)$$

- the core relational operators are:
  - $\sigma, \Pi, \times, \cup, \rho, -$
    - set difference is the only non-monotonic operator, so it is core
  - while the other operators $\bowtie, \cap$ can be expressed with other operators

SQL

- **structured query language (SQL)** is the standard langauge for interacting with relational DBMS (RDBMS):
  - many versions of the SQL standard exists, SQL92 or SQL2 is the main standard
  - has multiple components:
    * the **data definition language (DDL)** includes schema definitions, constraints, etc.
      · eg. `CREATE, ALTER, DROP`
    * the **data manipulation language (DML)** includes queries, modifications, etc.
      · eg. `SELECT, INSERT, UPDATE`
    * other components for transactions and authorization

Data Definition Language (DDL)

- the DDL component of SQL allows for expressing schema definitions
- basic common SQL data types:
  - string:
    * `Char(n)` with padded fixed length n
      · will pad shorter values
    * `Varchar(n)` with variable length and max length n
  - number:
    * `Integer` 32-bit
    * `Decimal(d, f)` with d total digits and f precision
      · eg. the max value of `Decimal(5, 2)` is `999.99`, useful for financial values
    * `Real` 32-bit, `Double` 64-bit
  - datetime:
    * `Date` eg. `2010-01-15`
      · no timezone in SQL standard!
    * `Time` eg. `13:50:00`
    * `Timestamp` eg. `2010-01-15 13:50:00`
      · on MySQL, `Datetime` is preferred instead
- SQL table creation:
  - `CREATE TABLE` statement

  - one `PRIMARY KEY` per table (but can be composed of multiple attributes):
    * `UNIQUE` is used for other keys (attributes may be null by SQL92)
    * by SQL standard, primary keys cannot be null
      · system will automatically mark those keys as not null, though the standard requires it to be explicitly written
  - `DEFAULT` sets the default value for an attribute
  - `DROP TABLE` statement for deleting a table

SQL table creation example:

```
CREATE TABLE Course(
    dept CHAR(2) NOT NULL DEFAULT 'CS',
    cnum INTEGER NOT NULL,
    sec  INTEGER NOT NULL,
    unit INT,
    instructor VARCHAR(50),
    title      VARCHAR(100),
    PRIMARY KEY(dept, cnum, sec),
    UNIQUE(dept, sec, title)
);
```

Loading Data

- there is no SQL standard for bulk data loading:
  - in Oracle and MySQL, `LOAD DATA INFILE <file> INTO TABLE <table>`
- options:
  - comma vs. tab separation for columns
    * `FIELDS TERMINATED BY ','`
  - columns enclosed with quotes
    * `OPTIONALLY ENCLOSED BY '"'`

Database Manipulation Language (DML)

- SQL gives a high-level description of what a user wants:
  - given a SQL query, DBMS figures out how best to execute it *automatically*
  - the core query operators are selection, projection, and join (SPJ)

The following is the full general format for SQL `SELECT` :

```
SELECT attributes, aggregates
FROM relations
WHERE conditions
GROUP BY attributes
HAVING aggregate condition
ORDER BY attributes
FETCH FIRST n ROWS ONLY
```

- note that `SELECT` appears first, but is the last clause to be interpreted
  - all other clauses can be semantically interpreted as executed in order

Relational Algebra Analogs

- consider the following example queries using Table 1, Table 2, and Table 3

1. Get the titles and instructors of all CS classes:

```
SELECT title, instructor -- project in SELECT clause
FROM Class               -- specify relations in FROM clause
WHERE dept='CS';         -- filter through condition in WHERE clause
```

- thus, the SQL statement `SELECT A1...An FROM R1...Rm WHERE C` is roughly equivalent to:
  - $\Pi_{A_1 \dots A_n}(\sigma_C(R_1 \times \cdots \times R_m))$
  - differences:
    - `SELECT` is projection rather than selection
    - SQL *does not* remove duplicates, uses *multiset* ie. bag semantics instead
      - sets can have duplicate elements and $\{a, b, a\} \neq \{a, b\}$
- multiset semantics and equivalence relation differences:
  - $R \cup S = S \cup R$ and $R \cap S = S \cap R$
  - but $R \cap (S \cap T) \neq (R \cap S) \cup (R \cap T)$

2. Get the names and GPAs of all students who take CS classes:

```
SELECT name, GPA AS grade -- renaming attributes, AS optional
FROM Student S, Enroll E  -- renaming operator for tuples, ie. tuple variables
WHERE dept='CS' AND S.sid=E.sid;

-- this returns duplicates if a student takes multiple classes!
SELECT DISTINCT name, GPA -- DISTINCT removes duplicates on final projected output
...
```

3. Get all student names and GPAs who live on Wilshire:

```sql
SELECT name, GPA
FROM Student
WHERE address LIKE '%Wilshire%';
```

- string variables:
  - `%` for any length string, `_` for a single character
  - eg. `LIKE %Wilshire%` matches any string containing `Wilshire`
  - eg. `LIKE ___%` matches any string with length ≥ 3
- other string functions include `UPPER(), LOWER(), CONCAT()`

4. Get all student and instructor names:

```sql
(SELECT name
FROM Student)
UNION -- set operator automatically takes care of duplicate instructors
(SELECT instructor -- can optionally rename to match input schemas
FROM Class);
```

- set operators:
  - eg. `UNION, INTERSECT, EXCEPT`
  - these operators *do* follow set semantics and remove duplicates
  - schemas of input relations should be the same
    - in practice, compatible types are fine
- to keep duplicates use `UNION ALL` etc. to enable bag semantics:
  - $\{a, a, b\} \cup \{a, b, c\} = \{a, a, a, b, b, c\}$
  - $\{a, a, a, b, c\} \cap \{a, a, c\} = \{a, a, b\}$
  - $\{a, a, b, b\} - \{a, b, b, c\} = \{a\}$

5. Get all SIDs of students who do not take any cs class

```sql
(SELECT sid FROM Student)
EXCEPT
(SELECT sid FROM Enroll WHERE dept='CS');
```

Subqueries

- SQL has extensions that allow certain queries that can not be expressed using purely relational algebra eg. subqueries and aggregate queries
- a SQL **subquery** is a nested `SELECT` statement within another:
  - the result from the inner `SELECT` is treated like a regular relation
  - if the result is a single-attribute, single-tuple relation, the result can also be used as a constant value ie. a **scalar-valued subquery**

- consider the following example queries using Table 1, Table 2, and Table 3

1. Get SIDs of students who live sith student 301:

```sql
SELECT sid
FROM Student
WHERE addr=(SELECT addr FROM Student WHERE sid=301)
      AND sid<>301;

-- without using subqueries:
SELECT S2.sid
FROM Student S1, Student S2
WHERE S1.addr=S2.addr AND S1.sid<>S2.sid AND S1.sid=301;
```

- using a scalar-valuled subquery
  - can not always guarantee that the subquery is a scalar, but filtering on the primary key does guarantees the result is a single tuple
- if we can always rewrite subqueries in a non-subquery systems, the two would be expressively equivalent:
  - generally, we can rewrite subqueries to non-subqueries as long as there is no negation
    - with negation, need to use `EXCEPT`
  - thus does not lend extra expressive power to SQL
    - ie. is a syntactic sugar for SQL

2. Get student names who take CS classes:

```sql
SELECT name
FROM Student
WHERE sid IN (SELECT sid FROM Enroll WHERE dept='CS');

-- without using subqueries:
SELECT DISTINCT name -- without DISTINCT, would return duplicates
FROM Student S, Enroll E
WHERE S.sid=E.sid AND dept='CS';
```

- the `IN` keyword is the set membership operator ie. $a \in S$:
  - note that in the example, the set may be a multiset operator, but `IN` would still does not return duplicates

3. Get student names who take no CS classes:

```sql
SELECT name
FROM Student
WHERE sid NOT IN (SELECT sid FROM Enroll WHERE dept='CS');
```

```sql
-- without using subqueries:
(SELECT name FROM Student)
EXCEPT
(SELECT name
FROM Student S, Enroll E
WHERE S.sid=E.sid AND dept='CS');
```

4. Get student IDs who have higher GPA than any student of age 18 or less:

```sql
SELECT sid
FROM Student
WHERE GPA>ALL( SELECT GPA FROM Student WHERE age≤18);
```

- `ALL, SOME` are set comparison operators that compares a value against an entire set of values:
  - eg. `a>ALL R` , `a≤SOME R`
  - note that `= SOME` is equivalent to `IN` and `<> ALL` is equivalent to `NOT IN`

5. Get student names who take any class:

```sql
SELECT name
FROM Student
WHERE sid IN (SELECT sid FROM Enroll);

-- using a correlated subquery to reference an outer relation
SELECT name
FROM Student S
WHERE EXISTS(SELECT * FROM Enroll E WHERE E.sid=S.sid);
```

- conceptually, for **correlated subqueries** to reference outer relations:
  - outer query looks at one tuple at a time and binds the tuple to `S`
  - for each `S` we execute the inner query and check the condition
  - real DBMS executes it more efficiently
- `EXISTS Q` is true if query `Q` returns one or more tuples
- subqueries can also appear inside a `FROM` statement, but they *must* be renamed
  - eg. `... FROM (SELECT name, age FROM Student) S ...`
- a **common table expression** can be used to alias a subquery
  - convenient for using the result of the same subquery multiple times

Using aliases:

```sql
WITH S AS (SELECT name, age FROM Student)
SELECT name FROM S WHERE AGE > 17;
```

Aggregates

- an **aggregate function** is a *new* mechanism to combine information from multiple input tuples into a *single* output tuple:
  - rather than getting information from one input tuple per output
  - eg. `AVG, SUM, COUNT, MIN, MAX`
  - thus aggregates *do* increase the expressiveness of SQL compared to relational algebra
  - because the output is a single tuple, all selected attribute names in an aggregate must be unique over the *entire* group
    - the groups selected to be aggregated can be adjusted using the `GROUP BY` operator
- consider the following example queries using Table 1, Table 2, and Table 3

1. Get average GPA of all students:

```sql
SELECT AVG(GPA)
FROM Student;
```

2. Get number of students taking CS classes:

```sql
SELECT COUNT(DISTINCT sid) -- need to avoid duplicates
-- SELECT DISTINCT COUNT(sid) is incorrect!
FROM Enroll
WHERE dept='CS';
```

3. Get average GPA of students who take CS classes:

```sql
-- incorrect, averages duplicate GPAs
SELECT AVG(GPA)
FROM Enroll E, Student S
WHERE E.sid=S.sid AND dept='cs';

-- subqueries are useful for handling duplicates
SELECT AVG(GPA)
FROM Student
WHERE sid IN (SELECT sid FROM Enroll WHERE dept='cs');
```

4. Get average GPA for each age group:

```sql
SELECT age, AVG(GPA)
-- SELECT sid, age, AVG(GPA) fails because an age group does not have the same sid
FROM Student
GROUP BY age;
```

- the `GROUP BY` operator partitions the groups that the aggregate function performs on
  - with `GROUP BY`, `SELECT` can have only aggregate functions or attributes that have a *single* value in each group

5. Get number of classes each student takes:

```sql
SELECT sid, COUNT(*)
FROM Enroll
GROUP BY sid;
-- this skips students who take no classes!
-- need to perform a union with those students, or use another technique
```

6. Get students who take two or more classes:

```sql
SELECT sid
FROM Enroll
-- WHERE COUNT(*)≥2 fails, aggregate cannot appear as part of WHERE
GROUP BY sid
HAVING COUNT(*)≥2;
```

- the `HAVING` clause is used to check on the condition of an aggregate
  - appears after `GROUP BY`

7. Another example with `HAVING` to get names of employees whose total salary is greater than all employees in Los Angeles:

```sql
SELECT name
FROM Work
GROUP BY name
HAVING SUM(salary) > ALL(
    SELECT SUM(salary)
    FROM Work W, Employee E
    WHERE W.name=E.name AND city='Los Angeles'
    GROUP BY W.name);

-- alternatively, using NOT EXISTS
SELECT name
FROM (SELECT name, SUM(salary) total-salary)
      FROM Work
```

```
      GROUP BY name) TotalSalary
WHERE NOT EXISTS(
    SELECT W.name
    FROM Work W, Employee E
    WHERE W.name=E.name AND city='Los Angeles'
    GROUP BY W.name
    HAVING SUM(salary) ≥ TotalSalary.total-salary);
```

More SQL Extensions

- consider the following example queries using Table 1, Table 2, and Table 3

1. For each student, return their name, GPA, and the *overall* GPA average:

```
SELECT name, GPA, AVG(GPA) FROM Student;
-- is an error, using attribute names in an aggregate
-- that are not unique over the *entire* group

SELECT name, GPA, AVG(GPA) FROM Student GROUP BY sid;
-- does not give an error, but AVG(GPA) is the incorrect value

SELECT name, GPA, AVG(GPA) OVER() FROM Student;
```

- the `OVER()` function is the SQL **window function**, used after some aggregate `FUNCTION(attr)`
  - generates one output tuple per input tuple, but the function is computed over *all* input tuples

2. For each student, return their name, GPA, and the average GPA in their age group:

```
SELECT name, GPA, AVG(GPA) OVER(PARTITION BY AGE) FROM Student;
```

- the `PARTITION` clause applies the window function over a specific grouping:
  - analogous to `GROUP BY` but for window functions
  - ie. changing the window over which the parent aggregate function is computed

3. Order students by GPA:

```
ORDER BY GPA DESC, sid ASC; -- primary ordering GPA, secondary ordering SID
```

- since SQL is based on multiset semantics, tuple order is ignored:

  - but for presentation purposes, the `ORDER BY` clause orders the result tuples by certain attributes
  - the default ordering direction is `ASC` if omitted
  - note that this not change SQL semantics and is purely for presentation

4. Get top 3 students order by GPA:

```
SELECT * FROM Students
ORDER BY GPA DESC
FETCH FIRST 3 ROWS ONLY;
```

- can limit the number of returned tuples with the `FETCH` clause:
  - `[OFFSET <num> ROWS] FETCH FIRST <count> ROWS ONLY`
  - skip the first `num` tuples and return the subsequent `count` rows
  - this was standardized too late, MySQL variation is `LIMIT <count> OFFSET <num>`
- note that with all of its expressiveness, SQL is *not* a Turing-complete language:
  - relatively simple extension to make it TC, just have to allow for user-defined aggregate functions
  - many requests to add more expressiveness to SQL

Table 5: Example Database with Ancestry Relationship

Child	Parent
Susan	John
John	James
James	Elaine
...	...

5. Find all ancestors of Susan from Table 5.

```
SELECT parent FROM Parent WHERE child='Susan'; -- get parents

SELECT P2.parent grandeparent
FROM Parent P1, Parent P2
WHERE P1.parent=P2.child AND P1.child='Susan'; -- get grandparents

...

WITH RECURSIVE Ancestor(child, ancestor) AS (
    (SELECT * FROM Parent) -- base case, initially populates Ancestor relation
                          -- with all parents
UNION -- UNION prevents duplicates
```

```
    (SELECT P.child, A.ancestor -- can now recursively use the Ancestor relation
     FROM Parent P, Ancestor A
     WHERE P.parent=A.child))
SELECT ancestor FROM Ancestor WHERE child='Susan';
```

- in order to find *all* ancestors, we need an additional **recursion** mechanism in SQL:
  - want to join multiple times until there are no more tuples to return
    - ie. running until a *fixed* point has been reached
  - can be structured as a graph problem eg. find all reachable cities given connections
  - support for recursion was added with SQL99, allows for computing closures of a set
    - when writing recursive queries, typically a `UNION` is used to connect a base ie. seed case with the recursive case
  - note that this recursive extension is not enough to make SQL TC

Handling `NULL`

What will be returned from the following query if GPA is `NULL` ?

```
SELECT name FROM Student WHERE GPA * 100/4 > 90
```

- SQL is based on **three-valued logic**:
  - all conditions are evaluated to be `True, False, Unknown`
  - if input to an arithmetic operator is `NULL` , its output is `NULL`
  - arithmetic comparison with `NULL` then returns `Unknown`
    - eg. `NULL > 90`
  - SQL returns a tuple only if the result from condition is `True`
  - note that `NOT Unknown` is still `Unknown`
- assume GPA is `NULL` and age is 17:
  - `GPA > 3.7 AND age > 18` gives `Unknown AND False` which evaluates to `False`
    - `AND` short-circuits
  - `GPA > 3.7 OR age > 18` gives `Unknown OR False` which evaluates to `Unknown`

Table 6: Truth Table for Three-Valued Logic

`AND, OR`	True	False	Unknown
True	True, True	False, True	Unknown, True

`AND, OR`	True	False	Unknown
False	False, True	False, False	False, Unknown

Table 7: Example Database for Aggregates

sid	GPA
1	3.0
2	3.6
3	2.4
4	`NULL`

- handling null values in aggregates from Table 7:
  - SQL aggregates ignore null values and apply the aggregate on the remaining tuples
  - `SELECT SUM(GPA) FROM Student` gives 9.0
    - in theory `Unknown` would be more valid
  - `SELECT AVG(GPA) FROM Student` gives 3.0
  - `SELECT COUNT(GPA) FROM Student` gives 3
  - `SELECT COUNT(*) FROM Student` gives 4
    - `COUNT(*)` is the exception that also counts tuples that might have null values everywhere
  - when an input to an aggregate function is *empty*:
    - `COUNT` returns 0
    - all other aggregates return `NULL`
- handling null values in set operators:
  - eg. $\{2.4, 3.0, null\} \cup \{3.6, null\} = \{2.4, 3.0, 3.6, null\}$
  - `NULL` is treated like regular values for set operators
  - to checking `NULL`, can use `IS NULL` or `IS NOT NULL`:
    - note that `= NULL` and `<> NULL` do *not* work to check `NULL`
    - `= NULL` evaluates to `Unknown`

Ex. Get the number of classes each student takes, including 0-class students:

```sql
SELECT sid, COUNT(*)
FROM Enroll
GROUP BY sid;
-- this does not return 0-class students!

SELECT sid, COUNT(*)
FROM Student S, Enroll E
WHERE S.sid=E.sid
```

```
GROUP BY sid;
-- still does not return 0-class students!

SELECT sid, COUNT(cnum) -- CANNOT use COUNT(*) since it counts the null cnum as 1
FROM Student S LEFT OUTER JOIN Enroll E ON S.sid=E.sid
GROUP BY sid;
```

- in this example, students taking no classes become **dangling tuple**
  - these tuples do not get preserved in the join condition
- need to use the **outer join** function to preserve dangling tuples:
  - `<relation1> <dir> OUTER JOIN <relation2> ON <condition>`
    - can perform a `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`
  - remaining attributes that are supposed to come from the other side are set as `NULL`

Data Modification

- the `INSERT` statement inserts a new tuple:
  - `INSERT INTO <relation> <tuples>`
  - `VALUES` keyword is used to literally specify tuples
- the `DELETE` statement removes tuples:
  - `DELETE FROM <relation> WHERE <condition>`
- the `UPDATE` statement updates tuples attributes:
  - `UPDATE <relation> SET <a1> = <v1>, ... WHERE <condition>`

1. Insert new tuples into the `Enroll` table:

```
INSERT INTO Enroll VALUES (301,'CS',201,1), (420,'EE',401,2);
```

2. Populate `Honors` table with students of GPA > 3.7:

```
INSERT INTO Honors (SELECT * FROM Student WHERE GPA>3.7);
```

3. Delete all students who are not taking classes:

```
DELETE FROM Student
WHERE sid NOT IN (SELECT sid FROM ENROLL);
```

4. Increase all CS course numbers by 100:

```
UPDATE Class
SET cnum=cnum+100
WHERE dept='CS';
```

Database Design

- how should we design tables in our database?
  - tables are not given
  - *"good"* tables may not be easy to come up with
- different database models:
  - **entity-relationship (ER)** model
    - developed onlongside relational databases
  - **universal modeling language (UML)** is more generalized
    - has less specialized tools compared to ER

Entity-Relationship Model

- the ER model is a graphical and *infoprmal* representation of infromation on the database:
  - used to capture what we learn from domain experts as well as database users
  - not directly implemented by DBMS
    - instead, tools exist to automatically convert E/R model into tables
  - has two main components, entity sets and relationship sets
- the **entity set** is a set of entities:
  - an **entity** is any thing or object in the real world
  - an **attribute** is a property ie. field of entities:
    - denoted by ellipses in ER and connected to the entity
    - entities with attributes are almost like tuples
  - a **key** is a set of attributes that uniquely identifies an entity in an entity set:
    - all entity sets in ER need a key
    - denoted by an underline in ER
  - entity set is analagous a class in OOP
    - denoted by a rectangle in the ER model
- the **relation set** is a set of relationships:
  - a **relationship** is a connection between entities
    - ie. edges between entities
  - relationships can also *have* attributes
  - relationship set is a set of relations of the same kind
    - denoted by diamonds in ER
- the **cardinality** of a relationship is how many times entities participate in a relationship:

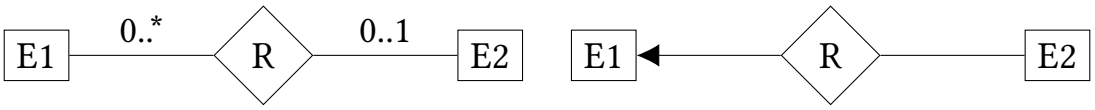

Figure 1: General Cardinality Notation

- in a **one-to-one** relationship:
  - entities on either side can participate in at *most* 1 relationship
  - eg. dorm room bed assignments
  - denoted by arrows on either end of the relationship diamond
- in a **one-to-many** relationship:
  - entites on one side can participate in more than one relationship
  - eg. faculty and classes, person and pets
  - denoted by an arrow on the "one" side
- in a **many-to-many** relationship:
  - entites on both sides can participate in more than one relationship
  - eg. students and classes
  - denoted by no arrows on either side
- in **total participation**, every entity participates in the relationship *at least* once
  - denoted by a double-line on the side of the relationship diamond
- for general cardinality notation, label an edge with $a..b$:
  - $*$ indicates unlimited cardinality
  - indicates the entity participates in the relationship between a through b times, inclusive
  - eg. in Figure 1, the two ER models are equivalent

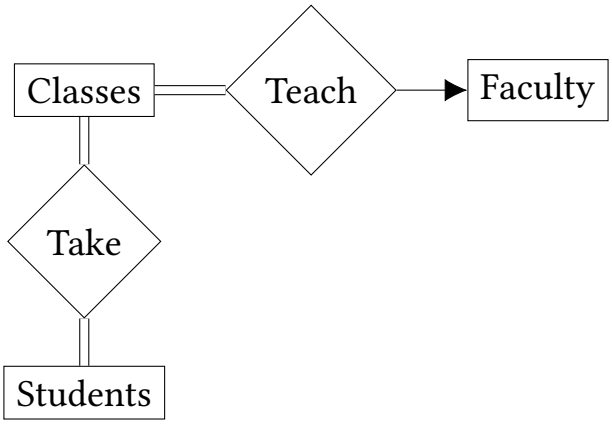

Figure 2: Relationship Notation Example

- ex. What do the relationships in Figure 2 mean?
  - a faculty member can teach multiple classes
  - every class must be taught by a faculty member
  - every student must take at least one class and every class must be taken by at least one student

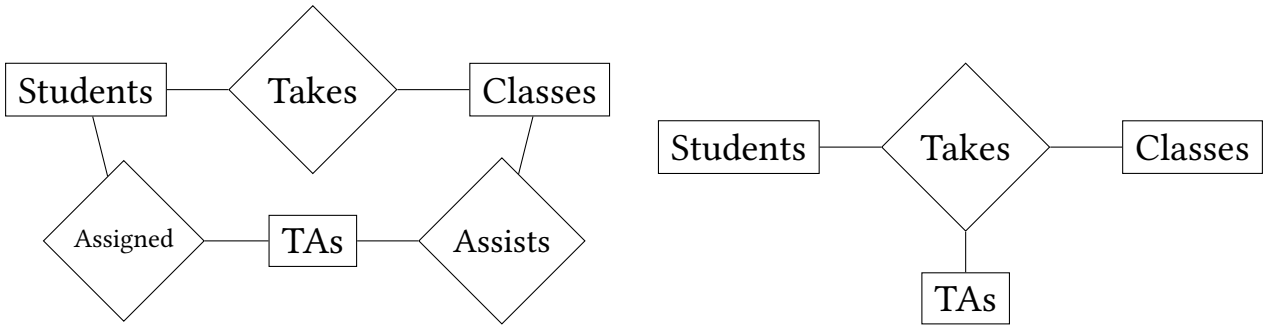

Figure 3: Tertiary Relationship Example

- sometimes, it is necessary to have more than just a binary relationship:
  - need an N-ary relationship where relationships connect N entites instead of just 2
  - ex. As seen in Figure 3:
    - if every TA is only assigned to assist a class, we can use purely binary relationships
    - however, if each student is *also* assigned to a particular TA, we may first try to add another binary relationship between student and TAs
      - but this leads to *redundancy* since the TAs are related to students through *both* their assignment to the class and student
    - instead, use a triple ie. tertiary relationship that encapsulates all of the information
      - eg. the Takes relationship would hold `(sid, TA, class)`

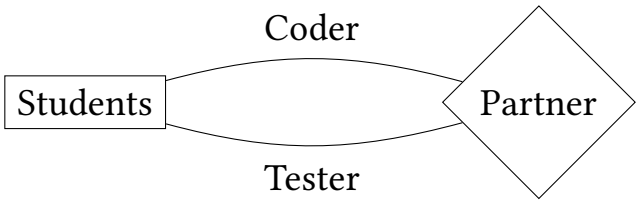

Figure 4: Role Example

- we can designate a **role** to each entity set that participates in a relationship set:
  - useful if an entity set participates more than once in the same relationship
  - denoted by labels on edges of a relationship in ER eg. in Figure 4
- may have superclasses and subclasses in ER:
  - ie. specializations and generalizations
  - subclass inherits all attributes of its superclass
  - subclass participates in the relationships of its superclass
  - subclass may participate in its own relationship

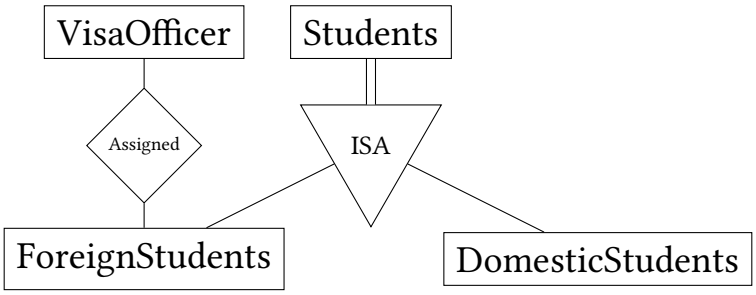

Figure 5: Subclass Example

- use an "IS-A" relationship in ER that connects superclass and subclass as seen in Figure 5:
  - denoted by a triangle that spreads into subclasses
  - by convention, top class is the superclass
- in **total specialization**, entity is *always* one of its subclasses:
  - denoted with double lines in ER
  - eg. *every* student is either a foreign or domestic student
- note that a superclass may belong to multiple subclasses
- to make subclasses distinct, we can demote add a "disjoint" keyword notation under the triangle

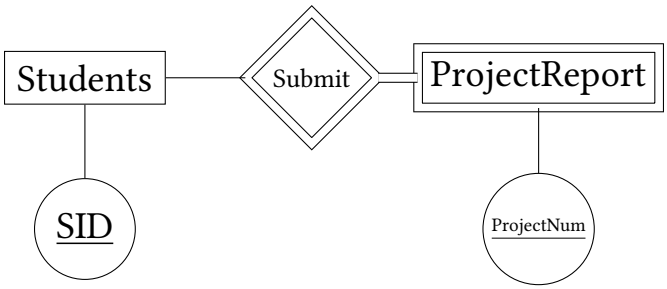

Figure 6: Weak Entity Set Example

- a **weak entity set** is one without a unique key:
  - denoted by a double rectangle and double diamond
  - *part* of its key will come from one or more entity sets it is linked to:
    - the **owner entity set** is the entity set providing part of the key
    - the **identifying relationship** is the relationship between a weak and owner entity set
    - needs a double edge between weak entity set and identify relationship because total participation is required to get the key
    - the **discriminators** are the attributes in a weak entity set that will also become part of the key
      - are denoted by dotted underlines
  - compared to the typical **strong entity set**
  - ex. As seen in Figure 6, a project report is submitted by a student but only has a project number attribute

    * owner entity set is the student, identifying relationship is the project submission, and the discriminator is the project number

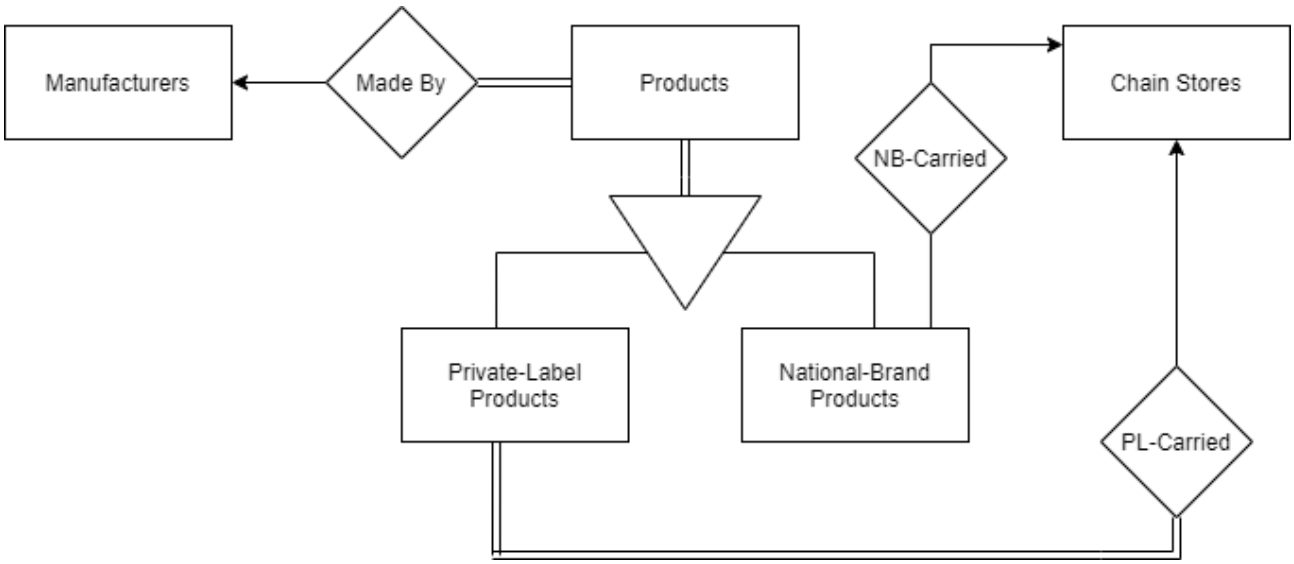

Figure 7: Barebones ER Example without Attributes

- ex. Encapsulating the following database design for stores and products in the ER in Figure 7:
  - All products are either private-label or national-brand.
  - Every product is manufactured by exactly one manufacturer.
  - Every private-label product is carried by exactly one chain store.
  - Some national-brand products may not be carried by any chain store.
- ex. Creating a new entity vs. simply encapsulating it as attributes:
  - if `Faculty(name, addr)` are instructors of `Class(dept, cnum, title)` , do we need a separate entity for faculty?
  - if it is a one-to-one relationship, it is usually better to encapsulate the entity as an attribute
  - for many-to-many or many-to-one relationships, it is better to separate out the entity to prevent redundancy in attributes

ER Model Conversion

- converting an ER model to tables is mostly straightforward:
  - automatic conversion tools exist
  1. for every strong entity set, convert it into a table with all the attributes
  2. for each relationship set, create one table with keys from the linked entity sets and its own attributes:
     - name conflicts should be prefixed with the entity name
     - if there are roles, those roles are preferred over the conneted entity's key

3. for every weak entity set, create one table with its own attributes and keys from the owner entity set
   - there is *no* table for identifying relationship set, but its attributes are also added to the table
4. for every subclass relationship, there are two approaches:
   - create one table for each subclass with its own attributes and the key of its superclass
     * requires a join to connect the information in the subclass with its superclass
   - alternatively, create one gigantic table for the super class that includes all atrributes
     * requires using null values for missing attributes
   - note that this initial conversion loses the cardinality information from the ER model

- ex. Converting Figure 8 to relations:

```
-- entity sets
Student(PRIMARY KEY name, addr)
TA(PRIMARY KEY name, addr)
Class(PRIMARY KEY cnum, title)
Faculty(PRIMARY KEY name, addr)

-- relationship sets
Teach(PRIMARY KEY cnum, name)
Take(PRIMARY KEY quarter, PRIMARY KEY Student_name,
     PRIMARY KEY TA_name, PRIMARY KEY cnum)
Partner(PRIMARY KEY coder, tester)
ProjectReport(PRIMARY KEY num, grade, PRIMARY KEY name)

-- subclasses
ForeignStudent(PRIMARY KEY name, country)
HonorStudent(PRIMARY KEY name, fellowship)
-- alternatively, Student(PRIMARY KEY name, addr, country, fellowship)
```

- to find the keys in a relationship, consider the edges in the relationship graph:
  - each edge represents a relationship
  - eg. in many-to-one graph between classes and faculty, the class key will uniquely identify the relationship
  - eg. in one-to-one graph between students, either students key can uniquely identify the relationship

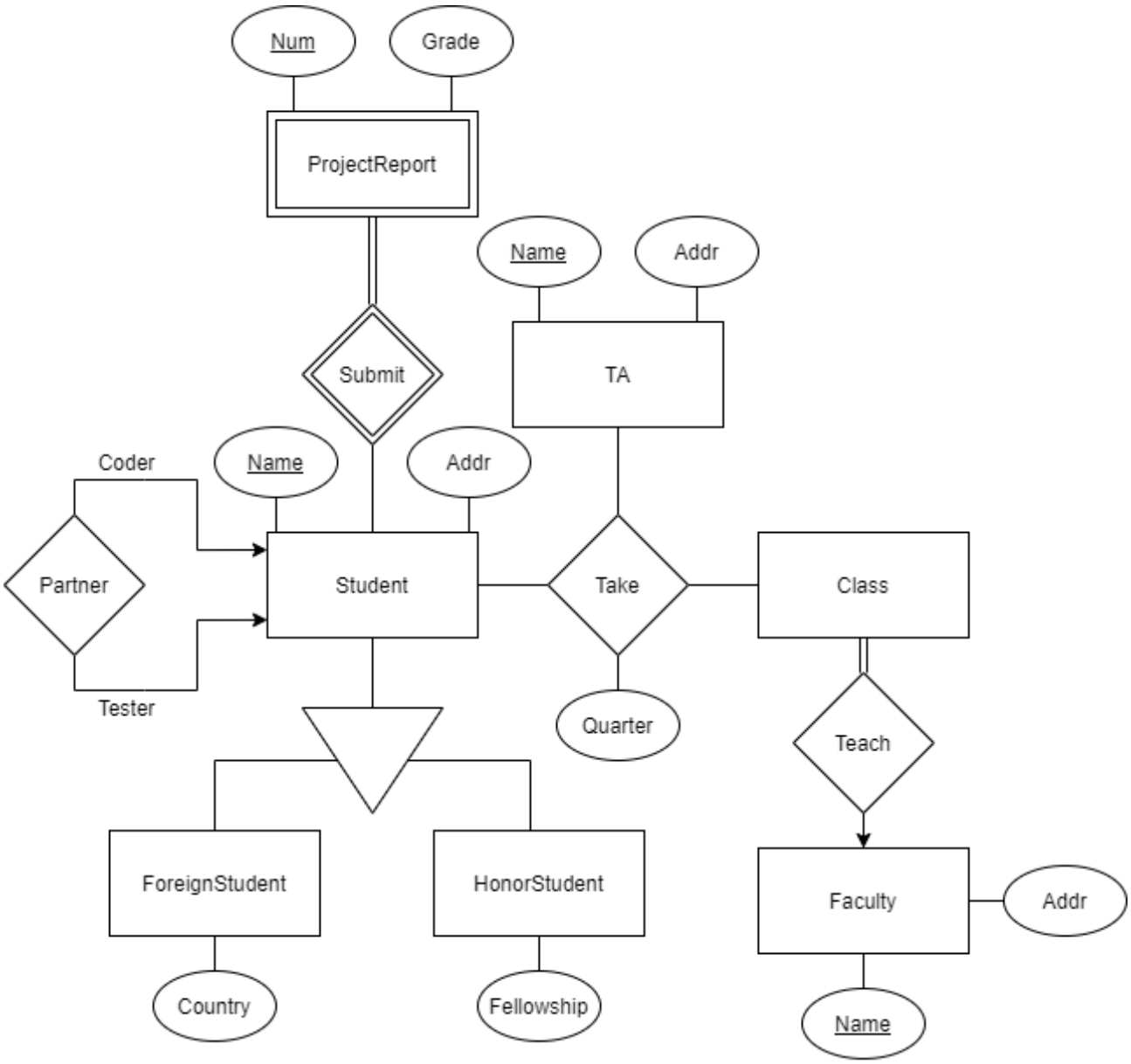

Figure 8: ER Conversion Example

Normalization Theory

- how do we design *"good"* tables for a relational database?
  - typically, we start with ER and convert it into tables
  - however, there are many choices to make in ER that lead to different tables
  - **relational design theory** aka **normalization theory** is a theory on what are "good" table designs:
    - hinges on trying to minimize *redundancy* in table design
    - provides an algorithm that converts "bad" designs into "good" designs automatically
  - note that reasonable performance may be another metric that conflicts with minimizing redundancy
- redundancy leads to potential anomalies:
  1. when updating, information may be updated partially and inconsistently
     - eg. a student changes their address
  2. when inserting, we may not include some information at all
     - eg. a student does not take any class
  3. when deleting, we may delete other information
     - eg. the *only* class a student takes gets cancelled
- ex. Consider the following table design:
  - `StudentClass(sid, name, addr, dept, cnum, title, unit)`
  - should this be split into multiple entities?
    - yes, we can see that information corresponding to students and classes will both be redundantly represented in the design
    - but could lead to faster information lookup if we want to associate classes to students
  - tradeoff between redundant storage vs. access efficiency
  - split into the following tables
    - `Student(sid, name, addr)`, `Class(dept, cnum, title, unit)`, and `Enroll(sid, dept, cnum)`
- is there an algorithm to arrive at the better design more systematically?
  - need to determine where the redundancy stems from ie. which attributes in which tuples can be "hidden" without losing information:
    - some attributes are uniquely *determined* by other attributes
    - eg. $sid \to (name, addr)$ and $(dept, cnum) \to (title, unit)$
      - note that the way we split the tables corresponds to these determinations, while adding a new table that connects the two
    - uses the concept of functional dependencies in normalization theory

  - when there is a functional dependency due to this determination, we may have redundancy
    * can decompose these tables into smaller ones to reduce redundancy, no need to store multiple instances of the relationship

Functional Dependencies

- for a **functional dependency (FD)** $X \to Y$:
  - $\forall\ u_1, u_2 \in R$, if $u_1[X] = u_2[X]$ then $u_1[Y] = u_2[Y]$
    * ie. no two tuples in R can have the same X values but different Y values
  - the notation $u[X]$ gives the values for the attributes X of tuple u
    * eg. $u[sid, name] = (100, James)$
  - ex. For `StudentClass` :
    * $sid \to name$ is a functional dependency
    * $dept, cnum \to title, unit$ is a functional dependency
    * $dept, cnum \to sid$ is *not* a functional dependency
  - ex. For Table 8 and Table 9, $AB \to C$ does not violate the functional dependency
    * however, Table 10 does
- types of FD:
  - an FD $X \to Y$ is a **trivial FD** when $Y \subseteq X$
    * *always* true regardless of real world semantics
  - an FD $X \to Y$ is a **nontrivial FD** when $Y \not\subseteq X$
  - an FD $X \to Y$ is a **completely nontrivial FD** when $X \cap Y = \emptyset$

Table 8: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_3

Table 9: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_1

Table 10: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_3

- in a **logical implication**, a *set* of FDs may imply a new FD:
  - ex. With table $R(A, B, C, G, H, I)$ and a set of FDs $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
    - $A \rightarrow H$ is true since $A \rightarrow B \rightarrow H$ ie. we say A logically implies H
- the **canonical database** is a method to check logical implication:
  - to check logical implication $A \rightarrow H$:
    - assume a random tuple u_1 with random values for each attribute
    - assume a second random tuple u_2 with random values except $u_2[A] = u_1[A]$
    - want to show $u_2[H] = u_2[H]$
    - $u_1[B] = u_2[B]$ from $A \rightarrow B$
    - $u_1[C] = u_2[C]$ from $A \rightarrow C$
    - $u_1[H] = u_2[H]$ from $B \rightarrow H$
    - given assumption of matching A, these are the only four values that are logically implied
  - to check logical implication $AG \rightarrow I$:
    - assume a random tuple u_1 with random values for each attribute
    - assume a second random tuple u_2 with random values except $u_2[A] = u_1[A]$ and $u_2[G] = u_2[G]$
    - want to show $u_2[I] = u_2[I]$
    - $u_1[B] = u_2[B]$ from $A \rightarrow B$
    - $u_1[C] = u_2[C]$ from $A \rightarrow C$
    - $u_1[H] = u_2[H]$ from $B \rightarrow H$
    - $u_1[I] = u_2[I]$ from $CG \rightarrow I$
- thus the **closure** of a functional dependency set F is $F+$
  - represents the set of all FDs that are logically implied by F
  - while the closure of the attribute set X is $X+$
    - represents the set of all attributes that are functionally determined by X
  - closure $X+$ computation algorithm:
    1. start with $X+ = X$
    2. repeat until no change in $X+$:
       - if there is $Y \rightarrow Z$ and $Y \subseteq X+$, then set X to $X+ \cup Z$
  - ex. What is the attribute set closure $\{sid, dept, cnum\}+$ for

$StudentClass$ given the FDs $\{sid \rightarrow name.(dept, cnum) \rightarrow (title, unit)\}$?
    - $\{sid, dept, cnum, name, title, unit\}$
  - ex. With table $R(A, B, C, G, H, I)$ and a set of FDs $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$:
    - $\{A\}+ = \{A, B, C, H\}$
    - $\{A, G\}+ = \{A, G, B, C, H, I\}$
      - $\{A, G\}$ is a *key* of R, while $\{A\}$ or $\{A, B\}$ are not
  - thus can create a new formal definition for a key X of R:
    1. $X+ = R$
    2. no subset of X satisfies (1) ie. X is minimal
- projecting functional dependencies:
  - ex. For $R(A, B, C, F)$ and a set of FDs $F = \{A \rightarrow B, B \rightarrow A, A \rightarrow C\}$, what FDs hold for $R'(B, C, D)$, where R' is a projection of R?
    - $\{B \rightarrow C\}$, need to consider $F+$ and pick the FDs that hold on the projected table

Decomposition

- now, we can remove redundancy by perform a series of decompositions on the ER model using FDs
  - in general, want to split $R(A_1, \dots, A_n) \rightarrow R_1(A_1, \dots, A_i), R_2(A_j, \dots, A_n)$ such that $\{A_1, \dots, A_n\} = \{A_1, \dots, A_i\} \cup \{A_j, \dots A_n\}$
  - need to be careful not to *lose* information when performing the decompositions:
    - ie. perform *lossless* decomposition
    - a decomposition of R into R_1, R_2 is a **lossless-join decomposition** if and only if $R = R_1 \bowtie R_2$
      - ie. can always get back the original table R if needed
    - but we need a more concrete general condition to guarantee the lossless-join decomposition in implementation

Table 11: Student Table

cnum	sid	name
143	1	James
143	2	Elaine
324	3	Susan

Table 12: S_1

cnum	sid
143	1
143	2
325	3

Table 13: S_2

cnum	name
143	James
143	Elaine
325	Susan

Table 14: $S_1 \bowtie S_2$

cnum	sid	name
143	1	James
143	1	Elaine
143	2	James
143	2	Elaine
325	3	Susan

- ex. In Table 11, is the decomposition into $S_1(cnum, sid), S_2(cnum, name)$ lossless?
  - no, $S_1 \bowtie S_2$ does not equal the original table, as seen in Table 14
    - number of tuples actually increased
- ex. In Table 11, is the decomposition into $R_1(cnum, sid), R_2(sid, name)$ lossless?
  - yes, $R_1 \bowtie R_2$ would equal the original table
  - the lossless quality is met because each tuple only joins with *exactly one* tuple of the other table:
    - ie. each decomposing table to needs have a unique identifying attribute
    - unlike in Table 14
- thus we can formalize that decomposition $R(X, Y, Z) \to R_1(X, Y), R_2(Y, Z)$ is lossless-join if $Y \to X$ or $Y \to Z$:
  - ie. a shared attribute uniquely determines every remaining attribute in *one* of the remaining tables
  - ie. the shared attribute is the *key* of one of the decomposed tables

        - only needs to be the key of one for the join condition to be met
    - note that a single table may be decomposed into many tables, but the decomposition is typically performed iteratively, into two tables at a time
- ex. Decomposing Student class into
    - r1 sid,name,addr and r2 sid,dept,cnum,title,unit using sid->name,addr
    - yes, the shared attribute is the key of one
- functional dependencies have potential to cause redundancy ie. repeated information in tuples:
    - eg. `StudentClass(sid, name, addr, dept, cnum, title, unit)` has redundancy caused by the FD $sid \rightarrow (name, addr)$
    - however, sometimes FDs does not lead to redundancy
        - eg. in `Student(sid, name, addr)`, the same FD $sid \rightarrow (name, addr)$ does *not* lead to redundancy
    - whenever the LHS of a FD contains the key, the FD does not cause redundancy
        - this is because the RHS will only be repeated once since the tuples referred to by the LHS are unique
- a relation R is in **Boyce-Codd normal form (BCNF)** with regard to the set of FDs F if and only if for every nontrivial functional dependency $(X \rightarrow Y) \in F$, X contains a key:
    - informally, this normal form indicates a "good" table design
    - ensures there is no redundancy in the table due to FD