CSCI 403: Databases
1 - Introductory Material

Fundamentals

What are some defining characteristics of a DBMS?

Self-describing databases

A DBMS allows us to define a database structure and store this definition together with the data. The data description is often called "meta-data". In DBMS terminology, the meta-data is stored in the database catalog and records details of all object types in the database (using object here loosely, not in the OOP sense). Key advantage here is that the database software is general-purpose and will work equally well for one application as for another.

Consider file-based system instead. Pre-xml, or even pre .ini file, no standard way to store data specific to an application; would often store data using the byte layout of the structs in the programming language. Each program would process its files and no others.

Program-data separation

Another benefit of self-description is that programs and the structure of data can be changed independently. In the file example, a change to the structure immediately requires both a change to program code and a rewrite of all data files from the old format to the new. With a DBMS, you can typically modify the database structure (e.g., by adding a field or a relationship) without recoding existing applications (obviously some changes will break software - deleting record types/relationships, some modifications). Beyond that, software can be written to recognize and work with modified structures by reading the catalog (meta-data), so for instance, adding a field can actually enhance result in a general-purpose query program.

Data abstraction

Structure of data storage on storage medium completely abstracted away - application programmer does not need to know!!!

Network multiuser access

A key feature from early days of DBMS is the ability to access database via multiple programs (or by multiple users) at the "same" time. This requires transaction control to ensure that updates from one user are not inconsistent with updates from another user (e.g., when one user changes data, the modification is to a current view of the data rather than an out-of-date view). This tends to go together with network access (since without it, you don't tend to have multiple users).

Client-server architecture

Client-server architecture arises organically from all of the above concerns. Since the database is self-describing and independent of application programs, it makes sense to build software devoted solely to providing database services. Add in network multi-user access, it is natural for applications to exist as separate client software, making a limited set of calls to execute DB functions on the server.

Early Models

Pre-1960, data was stored on sequential access media (tape, punch cards, punched tape). Data storage was typically application specific. No standard, general purpose system for storing, indexing, retreiving data. First general purpose

DBMSes date after introduction of direct access (e.g, disk) storage. Term "data-base" dates to 1962 (according to OED via Wikipedia)

Charles Bachman, GE 1964. IDS (Integrated Data Store) Bachman (b. 1924) is the 1973 Turing Award winner. IDS is the original network model database system, which was later standardized by CODASYL.

CODASYL stands for "Conference/Committee on Data Systems Languages", a consortium founded in 1959 to develop a standard programming language. Two notable achievements: COBOL standard, and network (aka CODASYL) database standard. In 1965 formed List Processing Task Force to develop extensions to COBOL for managing collections of records (with specific reference to IDS network model). In 1967 was renamed to "Data Base Task Group" (DTBG). 1969 - first network model standard. Defines DDL and DML (extensions to COBOL). Movement towards language independence in 1971. A number of vendors created DBMSes following the CODASYL standard, at least one of which is still sold today.

Bachman also developed (1965) an early transaction control system (allowing for multiuser network access).

Network model: flexible & powerful graph-based storage of records. Essentially composed of "records" and "sets". Records organized into named types defining the attributes for a particular record name. Set types define relationships between records. Specifically, a set type defines an "owner-member" relationship between two record types. An owner record is associated by a set with 0 or more member records, which are all of the same type. All records assigned a unique key value (tied to storage in system, allows for essentially direct access to record). Navigation between owner and members was via record id linkage (circular list anchored by owner). Power and flexibility in the system due to fact that records could participate in multiple sets (of multiple types), allowing for complex graph structures.

IBM 1966-68 IMS (Information Management System), for the Apollo space program (bill of materials tracking for Saturn V rocket). The first hierarchical model database system. IMS still sold today. In hierarchical model, records form a tree structure; each record can have at most one parent, but multiple child records. (Think of file systems.)

Both IDS and IMS were/are examples of navigational database systems. Access to a record is primarily predicated on knowing the record's unique key, and data retrieval follows linkages (like pointers) from parent to child (or to sibling in the case of network model).

The Relational Model

E. F. Codd (1923 - 2003). Employed by IBM, in 1970 publishes "A Relational Model of Data for Large Shared Data Banks". Internal pressure at IBM was to preserve revenue stream from IMS. IBM eventually developed a product based on Codd's ideas, called System R. System R project developed SQL (originally as SEQUEL). System R first sold in 1977. Codd wins 1981 Turing award for his work.

Oracle (initially developed 1977-79) is released by Larry Ellison's Relational Software in 1979. World domination soon follows.

INGRES - U. Berkeley project begun in 1973 by Michael Stonebraker & Eugene Wong after reading technical papers from the System R project. Source code available early on (first open-source DB?) Ingres initially competed with Oracle, but lost partly due to having own query language (QUEL). Project ended in 1985. Some of Ingres developers went on to develop commercial DBs, notably Sybase (which is also progenitor of MS SQL Server).

Postgres - Michael Stonebraker starts project at Berkeley in 1985 to address shortcomings of relational DBMSes of time, particularly inability to define new data types. (Still didn't support SQL until 1994.) Goes fully open source in 1994.

Michael Stonebraker is 2014 Turing Award winner.

Features of Relational Model

We will spend a lot of time on this in future lectures, but basically relational databases move away from pointer-based navigational approach to a more flexible approach based on set theory. Different views of the data can be dynamically created based on relational joins - these do not have to be designed into the database structure from the start. Initial versions were slow compared to navigational DBMSes, but rapidly improved with better indexing schemes, etc., to become dominant. The relational DBMSes dramatically improved data abstraction and program-data separation.

New (Old) Ideas

Flirtation in 1990s with OODBMSes (with rise of OOP). Note that OODB reverts to navigational concepts! OODB concepts were quickly subsumed into mainstream relational products such as Oracle to make the ORDBMS. Relational DBMSes have continued to expand into new data types (e.g., XML, BLOB, GIS) and gain new capabilities. However, the rise of the internet has led to a perceived need for new paradigms.

NoSQL (Not only SQL) databases are not well-defined, but exist in part due to the very large, very rapid, quickly changing data streams created by Internet usage and commerce. Some characteristics:

- Scalability - distributed over many nodes
- Flexible schema - e.g., document databases using JSON - good match to agile development processes - not tied to a fixed structure
- Fault tolerance - survival of single node downtimes

Some of the "new" databases are navigational in nature - graph databases share some similarities with network databases, for instance.

Note that all of these features are also being addressed by Oracle, Postgres in various ways, so whether NoSQL will remain a separate movement is yet to be determined.

CSCI 403: Databases
2 - The Relational Model

High Level

A *relation* resembles a table of values, or a flat file of records. A record is a collection of named data values representing some fact about the world. In the tabular view of things, each row is a record. Each column represents a specific field or *attribute* from every record in the table.

Tables and attributes in the relational model are named. For example, figure 1 shows a few rows and columns from the table **mines_courses**.

Definitions & Formalism

Tuples

In the relational model, rows are called *tuples*, and relations are simply *sets* of tuples. A tuple is an ordered and named collection of values. For instance, from figure 1, one tuple is
('Mehta, Dinesh', 'CSCI406', 'ALGORITHMS', 3).
The ordered collection
('instructor', 'course_id', 'title', 'max_credits')
contains the *attribute names* for the tuple.

The formal model is based on set theory, and formally the attributes of a tuple form a set. However, sets are intrinsically unordered, meaning we'd need to attach names to every value in every tuple to know what is going on unless we assume some convention of ordering. So by convention we order the attributes and then apply the same order to the values in the corresponding tuples.

Each attribute in a tuple has an associated *domain* of values which the values are constrained to belong to (e.g., strings of length 4, floating point numbers, dates).

Relation

We denote a *relation schema* R as $R(A_1, A_2, \ldots, A_n)$. The relation schema R has *degree* n and attributes $A_1, A_2, \ldots, A_n$. Each attribute has associated domain $D_i = \mathrm{dom}(A_i)$. (Notation here is not particularly important, e.g., something you will be tested on, but sometimes it helps in thinking about what is going on.)

A *relation state*, or simply *relation*, $r(R)$, is a set of tuples conforming to the relation schema R. That is, we can say for any tuple $t \in r$:

$$t = (v_1, v_2, \ldots, v_n) : v_i \in \mathrm{dom}(A_i).$$

We can additionally denote the value corresponding to the specific attribute A_i as $t.A_i$ or $t[A_i]$.

We can equivalently define a relation as any subset of the Cartesian product of the domains of the attributes:

$$r(R) \subseteq (\mathrm{dom}(A_1) \times \mathrm{dom}(A_2) \times \cdots \times \mathrm{dom}(A_n))$$

Some Notes

The statement that "a relation is a set" implies that there is no ordering of tuples in a relation. This is true in practice as well, in that a relational DBMS may choose to store tuples (rows) of a relation (table) in any order or fashion according to the design of the DBMS. Formally, two relations are equivalent if they have the same tuples, even if the tuples are in different orders.

The same statement also implies that there are no duplicates in a relation, i.e., that no two tuples contain the exact same values. In practice, this restriction is not enforced by relational DBMSs, so technically relations in a DBMS form a *multiset*. The implications of this will be discussed more when we talk about SELECT queries and the DISTINCT operator.

In the same vein, formally the attributes of a relation schema have no order, thus two relations

instructor	**course_id**	**title**	**max_credits**
Painter-Wakefield, Christopher	CSCI262	DATA STRUCTURES	3
Painter-Wakefield, Christopher	CSCI403	DATABASE MANAGEMENT	3
Rader, Cynthia	CSCI306	SOFTWARE ENGINEERING	3
Mehta, Dinesh	CSCI406	ALGORITHMS	3

Figure 1: Sample rows from the table **mines_courses** (only some columns shown)

that are equivalent except for the ordering of their attributes are equivalent.

NULL

A few words need be said about NULL at this point. NULL is a value that can exist in a relational database to represent a variety of concepts. Basically it is the absence of a value. NULL can mean that the value is unknown, or missing, or simply irrelevant to a particular tuple. In the mines_courses table there are many rows with NULL instructor; this likely represents the fact that when this data was obtained, not all course instructors had been assigned.

It is very important to note that NULL values cannot be compared. In particular, two NULL values are never equal to each other. In queries (see next lecture), there is a special operator (IS NULL) for detecting the presence of a NULL value.

Constraints

In addition to relations as described above, the relational model is also concerned with *constraints*. Constraints are simply restrictions on a relation. There are three flavors of constraints:

1. implicit (model-based)
2. explicit (schema-based)
3. application-based

Implicit constraints are those which are implied by the model of the world that a schema represents. For example, the instructor field of the mines_courses table is assumed to contain person's names. Application-based constraints are constraints which are not defined in the database, but which are enforced by the applications which use the database; these are often known as the "business rules" of an application, and can be things like "no employee can have a hire date that is not on the first of some month". Such constraints tend to be more complex to implement on the DBMS itself, and thus are relegated to application code.

In terms of the relational model, we are mostly interested in the explicit constraints. One trivial constraint enforced by the database are the domain constraints requiring that values in a column be part of the domain as defined for the column (attribute) in the relation schema. The next constraint of interest is called a *primary key constraint*, and to explain that we need to define some more terms.

Keys and Superkeys

A *superkey* of a relation schema R is some subset of R's attributes with the property that no relation of R may contain tuples with exactly the same values for the attributes in the superkey. For example, figure 2 shows some rows from the mines_courses table, this time with some different attributes shown. If we consider the set (course_id, section, instructor), no two rows shown (and in fact, in the actual table) have the same values for all three attributes. Thus (course_id, section, instructor) is a superkey of mines_courses.

Some facts about superkeys that are immediately obvious; first, if we consider the formal relational model in which duplicate tuples are not allowed, then every relation schema has at least one superkey, which is the set of all attributes of the schema. Second, clearly any superset of a superkey is also a superkey.

A *key* is simply a superkey with the property that no attribute can be removed from the key without destroying the superkey property. For example, the set (course_id, section, instructor) is not a key, as we can remove the instructor attribute and have the superkey (course_id, sec-

crn	course_id	section	instructor	title
82482	CSCI262	A	Painter-Wakefield, Christopher	DATA STRUCTURES
80386	CSCI262	B	Painter-Wakefield, Christopher	DATA STRUCTURES
84758	CSCI403	A	Painter-Wakefield, Christopher	DATABASE MANAGEMENT
81367	CSCI406	A	Mehta, Dinesh	ALGORITHMS

Figure 2: Some more rows from the mines_courses tables, this time selecting different attributes

tion). On the other hand, if we have (course_id, section), clearly we cannot remove either attribute, since both columns have duplicates by themselves. Therefore (course_id, section) is a key of mines_courses. Another way of defining key is that it is a *minimal* superkey.

Multiple keys are possible; in the mines_courses table, (crn) is also a key. All keys of a relation schema are called the *candidate keys*. Conventionally, we select one key to identify as the *primary key* for a relation. The primary key is a unique identifier for any tuple in the relation. Typically we prefer smaller sets over larger ones for the primary key, thus (crn) is a good choice for the primary key of the mines_courses table.

Note that no tuple in a relation can contain a NULL value for any attribute of a key; since NULLs cannot be compared, it is impossible to establish the uniqueness of tuples containing NULL.

In practice, we can set a primary key constraint as part of the relation schema in a DBMS. This constraint is then enforced by the DBMS, requiring uniqueness for the attributes in the primary key. (Also, the DBMS will not allow NULL values.) Typically other keys can be enforced as well by a uniqueness constraint, but the convention is to have only one primary key identified for a relation.

Relational Database

So far, we have been discussing constraints as applied to a single relation schema. In the larger picture, a relational database is made up of multiple tables and associated constraints, and conforms to what is called the *relational database schema*. The relational DB schema also allows for constraints that represent relationships between relation schemas. These are called *referential integrity constraints*, or *foreign key constraints*.

Basically, a foreign key constraint is used to require that entries in one table have corresponding entries in another. Formally, a set of attributes is a foreign key of a relation schema if its values are either NULL or exist in the specified attributes of a referenced relation schema (note that the domains must match). For example, suppose there exists a table named instructors containing name, office, email, and other information about every instructor at Mines. Then we can create a foreign key constraint relating the mines_courses table (instructor attribute) to the instructors table (name attribute). Each instructor value in mines_courses would then be required to be either in the instructors table, or NULL.

As another example, it is possible to have a referential integrity constraint from a table to itself; an example from the book is of a table containing employee information. The primary key of the employee table is the employee's SSN, and the table also contains the SSN of the supervisor of every employee. Thus there exists a foreign key constraint on employee (superssn) referencing employee (ssn), meaning that the supervisor of any employee must also be an employee. (This table is also in the csci403 database.)

CSCI 403: Databases
3 - Basic SQL Retrieval Queries

SELECT

The most basic retrieval query looks like
`SELECT * FROM <tablename>;`
or
`SELECT` a_1, a_2, `... FROM <tablename>;`

- a_i is an attribute of the relation (a column of the table)
- `*` stands for all attributes

This query selects all rows. By using the second form of the query, we choose only the attributes we are interested in, in order, from the relation (table). This is called a "projection".

The result of a query is a *relation*. It is anonymous (has no name), but otherwise is like any other relation; the attributes for the result relation are the attributes selected in the query (and have the same name and order). This has certain implications that we'll examine further later in the lecture.

WHERE

Typically want only *some* rows:
```
SELECT <attributes>
FROM <tablename>
WHERE <condition>;
```
The `<condition>` is a Boolean expression on the attributes or functions of the attributes of the table.

E.g.,
```
SELECT course_id FROM mines_courses
WHERE instructor = 'Painter-Wakefield,
Christopher';
```
gets the course_ids of all courses taught by me.

There are many operators and functions applicable to the WHERE condition. E.g., to get all courses *not* taught by me, use the "not equals" operator, which is `<>`, e.g., `SELECT ... WHERE instructor <> 'Painter-Wakefield, Christopher';`

You have access to all the other usual relational operators: `<`, `>`, `<=`, `>=`. There is also a shortcut operator, BETWEEN, which tests for (inclusive) containment in some interval. E.g.,
```
SELECT * FROM mines_courses
WHERE max_credits BETWEEN 3 AND 6;.
```
This is equivalent to
```
SELECT * FROM mines_courses
WHERE 3 <= max_credits
AND 6 >= max_credits;
```
Combine expressions into compound expressions using AND and OR; AND has precedence over OR (or can use parentheses to force order of evaluation). NOT can be used to invert any Boolean expression.

Recall that NULL values are not comparable - you can not, for instance, test to see if an attribute is NULL by comparing using =. Instead, there is a special operator, IS NULL, which tests for NULL values:
```
SELECT * FROM mines_courses
WHERE instructor IS NULL;.
```
To test for non-NULL values, use IS NOT NULL.

A useful operator on string attributes is LIKE, which lets you do wildcard comparisons. To get all courses taught by me or Dr. Rader, for instance:
```
SELECT course_id FROM mines_courses
WHERE instructor LIKE 'Painter%' OR
instructor LIKE 'Rader%';
```
In wildcard expressions, % stands in for zero or more characters, so '%foo%' would find all text containing the string "foo", whereas 'bar%' only finds strings that start with "bar". Another wildcard is the underscore, _, which matches a single character. This forces there to be exactly one character in that position, but it can be any character. E.g., `SELECT ... WHERE course_id LIKE '____4__';` would get all rows for 400-level

courses in any subject (e.g., CSCI403, LAIS404, CEEN443, etc.)

Operators in Other Clauses

In addition to the relational operators discussed above, you have access to the normal mathematical operators, which can be used in expressions in most any clause. The relational operators above (evaluating to true/false) can also be used. For instance, you can SELECT the result of a mathematical operation on some attributes or constants; you can even use the database query engine as a calculator, if you want:

```
SELECT 16 * 22 - 17;
```

See the PostgreSQL documentation for all the operators available.

Functions

Many functions available, including mathematical, string, type conversion, etc. See function reference in PostgreSQL documents for examples (not all functions available in PostgreSQL are standard SQL.) An example of a useful function is substring():

```
SELECT substring(course_id from 1 for
4) AS subject FROM mines_courses WHERE
...;
```

This query gets just the 4-letter subjects (e.g., CSCI, LAIS, CEEN) from the mines_courses table.

Names and Aliasing

Note above the use of the keyword AS. This keyword lets us *rename* attributes in our SELECT query for the output relation. In the example above, the application of the substring() function results in a not very informative attribute name (substring) for the output relation, so I renamed it to be "subject". This affects e.g., output column names in your query tool, names of columns (entries in a dictionary, e.g.) when querying from a programming language, and so forth. It will also come in handy in some situations when we join two or more tables together (to avoid name collisions) and in ORDER BY clauses and GROUP BY clauses.

Name collisions when joining (see below) can also be resolved by specifying the table name together with the column:

```
SELECT mines_courses.course_id
FROM mines_courses;
```

We can also *alias* tables and use the aliases to resolve columns in a join query (very useful!) For example, the above query could be rewritten as

```
SELECT mc.course_id
FROM mines_courses AS mc;
```

The AS keyword is optional, so this is equivalent to

```
SELECT mc.course_id
FROM mines_courses mc;
```

Joins

A SELECT query can specify more than one table as the source of data:

```
SELECT table1.a₁, table1.a₂, ...,
table2.a₁, ...
FROM table1, table2, ...
WHERE ...;
```

With no WHERE conditions, this gives the *cross product* of the two tables; the resulting relation's tuples are concatenations of each tuple from table1 with each tuple from table2.

E.g.,

```
SELECT mc.instructor, mef.name FROM
mines_courses AS mc, mines_eecs_faculty
AS mef;
```

results in tuples like ('Painter-Wakefield, Christopher', 'Painter-Wakefield, Christopher') and ('Painter-Wakefield, Christopher', 'Hellman, Keith') and ('Mehta, Dinesh', 'Rader, Cynthia') etc.

So this is typically not what is desired. There are two main mechanisms for specifying *join conditions* to determine which rows from table 1 go with which rows from table 2 (and so forth). We'll talk about JOIN clauses later, for now we'll focus on what we can do in the WHERE clause.

Thinking conceptually of the result relation from the above query as a cross-product result, we can apply the WHERE clause to filter out the nonsense rows and retain the ones that actually make sense:

```
SELECT * FROM mines_courses AS mc,
mines_eecs_faculty AS mef
WHERE mc.instructor = mef.name;
```

We can join on multiple conditions, or add additional conditions as usual. For instance, if we

crn	course_id	section	instructor	title
82482	CSCI262	A	Painter-Wakefield, Christopher	DATA STRUCTURES
80386	CSCI262	B	Painter-Wakefield, Christopher	DATA STRUCTURES
84758	CSCI403	A	Painter-Wakefield, Christopher	DATABASE MANAGEMENT
81367	CSCI406	A	Mehta, Dinesh	ALGORITHMS

Figure 1: Some rows from the mines_courses table

name	office	email
Painter-Wakefield, Christopher	BB 280I	cpainter@mines.edu
Mehta, Dinesh	BB 280J	dmehta@mines.edu
Rader, Cynthia	BB 280D	crader@mines.edu
Hellman, Keith	BB 310F	khellman@mines.edu

Figure 2: Some rows from the mines_eecs_faculty table

provided instructor names as two fields (last name and first name) instead of one, we would need two join conditions ANDed together.

Here's a query to try:

```
SELECT course_id, instructor, office,
email
FROM mines_courses, mines_eecs_faculty
WHERE instructor = name;
```

Note that this query uses no aliases or fully specified column names; as long as this is unambiguous, the SQL command processor will accept it. However, if instead of "name", we had used the attribute name "instructor" in the mines_eecs_faculty table, then it would make no sense to have a join with the condition `WHERE instructor = instructor`. In that case we must fully specify the column names (either using the table names or aliases for the tables). In general, even where unambiguous, table aliases and fully specified column names are often preferable for making the query more readable and understandable.

We can join more than two tables. The mines_courses_meetings table contains info about the days/times/locations where course sections are held. This table can be joined to the mines_courses table through the common attribute "crn". Here's a query to try:

```
SELECT mc.instructor, mc.course_id,
mcm.days, mcm.building, mcm.room,
mcm.begin_time,
mef.office, mef.email
FROM mines_courses mc,
mines_eecs_faculty mef,
mines_courses_meetings mcm
WHERE mc.crn = mcm.crn
AND mef.name = mc.instructor;
```

Notes on joins

N.B., in a join, missing entries or NULLs in attributes in the join condition mean that rows are simply excluded, since NULL never equals anything (even NULL). For instance, in joining mines_courses with mines_eecs_faculty, you will only get back information for courses that are taught by EECS faculty (since that is all that is in the table), and only for courses whose instructor was known when the data was created (NULL instructors cannot match any name). Later we'll learn about outer joins, which will let us include rows that do not match in a join.

Also note that, even though we describe joins conceptually as forming a cross product which we then filter down with our join conditions, in actual practice this is not what happens - it would be way too expensive to make a cross product with a billion rows only to select seven of them! When we study relational algebra we'll see how queries can be reordered to make for much more efficient join queries. (Unless the user actually *wants* a cross-product result, of course.)

DISTINCT and ORDER BY

Since the result of a SELECT query is a relation, which is (in the relational model) a *set* of tuples, we might reasonably expect our queries to return no duplicate rows. However, relations in

crn	days	building	room	begin_time
82482	MWF	CT	B60	12:00
80386	MWF	CT	B60	1:00
84758	MWF	BB	W250	3:00

Figure 3: Some rows from the mines_course_meetings table. The data in this table is separate from the mines_courses table because some sections of some courses can have multiple entries due to different meeting times/locations on different days (e.g., a lab section in a computer classroom).

a relational DBMS actually do not adhere to this particular part of the model; duplicate rows are acceptable in tables, and duplicate rows in results are also normal. E.g., if we do
`SELECT instructor FROM mines_courses;`,
my name will pop up three times (one for each course section I teach).

To eliminate duplicate tuples, use the DISTINCT keyword:

```
SELECT DISTINCT instructor
FROM mines_courses;
```

On the other hand, relational DBMSes do follow the relational model in that rows have no intrinsic ordering. A query can provide results in any order (even in different orders for the same query done twice!). To set the order, use an ORDER BY clause, where ORDER BY is followed by the columns on which to sort:

```
SELECT firstname, lastname, otherstuff
FROM sometable
ORDER BY lastname, firstname;
```

This will sort alphabetically (assuming text attributes) by lastname and then firstname. If we want to reverse the order of sorting for any attribute, we can include the DESC keyword:

```
SELECT earnings, salesperson
FROM sales
ORDER BY earnings DESC;
```

CSCI 403: Databases
4 - Basic DDL

Schema

In the relational model of the database, we used schema as a term to describe the names and attributes of relations and the constraints on the relations in a database. In the SQL standard, schema is defined more broadly as a container for database objects, including tables, constraints, views, indexes, etc. A SQL database can contain many schemas.

- Like namespaces, a schema allows us to separate logical units in a database and avoid name collisions.
- Schemas also makes for easier application of security policies by letting us grant permissions on whole schemas at a time rather than table by table.
- Schemas are generally something created and managed by the database administrator/owner, not by the database user or programmer. (Acknowledging that of course, programmers may also be DBAs or DB owners!)
- You do not (AFAIK) have rights to create schemas in the course database. You have your own schema, named by your Mines id. Anything you create by default will be created in your personal schema. You also have read-only access to a schema named "public".

Catalog

In SQL, the *catalog* can be viewed as either a higher-level container containing schemas, or as the set of system tables within the database describing all of the other objects in the database. All relational DBMSes have some form of system tables which can be read to determine the structure of objects in the database. To see the system tables in psql, enter the command "\dS" at the prompt.

Table Creation

SQL uses a very "English-like" syntax. To create a table in SQL, you issue a `CREATE TABLE` command. A somewhat abbreviated syntax of the command looks like:
`CREATE TABLE [schemaname.]tablename (` { columnname datatype [NOT NULL] [UNIQUE] [PRIMARY KEY] | table_constraint } [,...])

Which probably looks like gibberish... you will need to learn how to read SQL documentation, most of which looks like this. Briefly, anything in [ ] is optional, while the construct { A | B } means you can have A or B. If you see [,...], that means you can have more copies of the last optional thing (with a comma separator). Anything else is required.

Here's an example of creating a throwaway table, showing some of the different data types you might use:

```
CREATE TABLE yourid.stuff (
id serial PRIMARY KEY,
name text NOT NULL,
age integer,
gender char(1),
salary numeric(9,2),
favorite_constant double precision,
date_hired date);
```

This will create a table named "stuff" in the schema "yourid" with several columns, each of a different type. (Note that text and serial types are specific to PostgreSQL, although similar types exist in other DBMSes.) This construction sets the column "id" as a primary key for the table (just the one column in the key), and the name field is constrained to not contain NULL values. We can also create foreign key references or other constraints within the table creation statement. Table constraints can also be created in a separate declaration in the comma-separated list within the (); here's the same table, but with a two-column primary key (on name and age):

```
CREATE TABLE yourid.stuff (
id serial,
name text NOT NULL,
age integer,
gender char(1),
salary numeric(9,2),
favorite_constant double precision,
date_hired date,
PRIMARY KEY (name, age));
```

Foreign key constraints can also be created in the CREATE TABLE command:

```
CREATE TABLE yourid.otherstuff (
other_id serial PRIMARY KEY,
stuff_id integer REFERENCES
yourid.stuff (id));
```

is equivalent to

```
CREATE TABLE yourid.otherstuff (
other_id serial PRIMARY KEY,
stuff_id integer,
FOREIGN KEY (stuff_id) REFERENCES
yourid.stuff (id));
```

Types

SQL defines a number of types for attributes in your relations, and most DBMSes define additional types. Some types that you will find useful are listed below.

Integers

- INTEGER - 32-bit integers
- SMALLINT - 16-bit integers
- BIGINT - 64-bit integers

Fixed-precision numeric (exact)

- NUMERIC(w,p) - Defines numbers with a maximum of w digits, and a precision of 2
- DECIMAL(w,p) - same as NUMERIC(w,p)

Floating point (inexact)

- REAL - 32-bit floating point
- DOUBLE PRECISION - 64-bit floating point

Strings

- CHAR(n) - strings of length exactly n, padded with spaces if necessary
- VARCHAR(n) - variable length strings of max length n
- TEXT - variable length strings, no limit (PostgreSQL type)

Date/Time

- DATE - holds dates. You can enter DATE values as strings in the format 'YYYY-MM-DD', other formats maybe possible depending on DBMS.
- TIME - holds times. Format 'HH:MM:SS', can also add decimal points after for sub-second times. Optionally, timezone can also be included.
- TIMESTAMP - date and time.

Other types

- BOOLEAN - holds true/false values - various formats are compatible
- SERIAL - an auto-incrementing integer type (PostgreSQL type)
- MONEY
- ... many more

Type Conversion

There are several ways to convert one type to another in PostgreSQL. The standard SQL way is to use the CAST function:

`CAST (expression AS type)`

e.g.,

`SELECT CAST('1/2/2016' AS DATE) AS foo;`.

Generally speaking, if the database can work out how to convert a type, it will do so without any special effort on your part. In particular, the database can usually figure out how to convert string representations of numbers, dates, times, etc. into the corresponding actual types. NULLs get left as NULLs.

However, you have to be careful when trying to convert, for instance, all entries in a column in a table - if not all the entries are recognizable as the type you are converting to, the query will fail. For instance, if a string column mostly contains integers, but a few rows have 'unknown' or 'N/A' or some other string, the database will refuse to try converting them to a numeric type.

Another (more compact) way to do casts in PostgreSQL (not portable to other systems) is the :: operator, which is used as follows:

```
expression::type
```

e.g., `SELECT '1/2/2016'::DATE AS foo;`

There are other, more involved ways to do type conversions using functions and/or the CASE statement, but generally you want to avoid the complication of these approaches.

Default values

Another option when creating a table is to set a default value for a column. The default value will be used whenever a new tuple (row) is INSERTed (see next lecture) into the table, but no value is provided for the column. For instance:

```
CREATE TABLE yourid.stuff (
id serial PRIMARY KEY,
name text NOT NULL,
salary numeric(9,2) DEFAULT 0.00,
date_hired date DEFAULT CURRENT_DATE);
```

creates a table where the salary field defaults to zero, and where the date_hired field defaults to the current date (the date on which the row was created), using the SQL standard function CURRENT_DATE.

The `SERIAL` type mentioned above is actually implemented as an integer type with the default value being obtained via the function `nextval()` applied to a *sequence* object associated with the column. A sequence is just a counter object which can be queried for its current value (currval()) or its next value (which also increments the counter). The value of the counter can also be set (using setval()).

CREATE... AS

An easy way to create a table from a SELECT query; do

```
CREATE TABLE schemaname.tablename AS
SELECT ...  ;
```

This will create a table with attributes and types determined by the SELECT query result. Column renaming together with functions, joins, etc., makes this a powerful way to create a new table. Note, however, that this does not create any keys or constraints - these will have to be added on using ALTER TABLE.

Notes on Workflow

Tables/schemas are generally created only occasionally and modified (relatively) seldom thereafter. [You may have a different experience in an agile environment; one reason NoSQL is popular right now is the "schema-less" nature of NoSQL databases.]

When making tables, it is easy to make small mistakes or wish you had done something differently. Changes after the fact are often harder than simply dropping everything and starting over. Thus: make scripts! Your scripts should (optionally) drop everything, then create all of your tables and constraints, load all of your data, etc. Re-run until you are happy with the result. (Having these scripts will help a lot downstream when your application goes into development for version 2...)

ALTER TABLE

When you *do* want to modify a table in place - which of course happens - then you want the ALTER TABLE command. You can do almost anything with this command (see the PostgreSQL documentation), including adding and removing columns, renaming columns, changing data types of columns, adding and removing constraints, setting column default values, etc.

CSCI 403: Databases
5 - Insert, Update, and Delete

Modifying the Database

So far, we have discussed SELECT queries, which merely retrieve data from the database. Now we turn to queries which can modify the database (these are called queries, also, even though they are more in the nature of commands).

Unlike SELECT queries, a correctly formed modification query has the possibility of *failing* due to a constraint violation. For instance, you might try to insert a NULL value into a NOT NULL constrained column, or insert into a foreign key column a value which doesn't exist in the referenced table. Update can likewise violate column or key constraints, and deletes can cause foreign key constraint violations. In all of these instances, the query will fail entirely rather than doing a partial job - e.g., an update will not update some rows unless it can update all the rows indicated by the query.

Insert

The INSERT query is the primary means by which data is added to a table in a SQL database. At its simplest, INSERT queries let you add a single tuple to a single table, providing values in the order of the attributes in the table:

```
INSERT INTO tablename
VALUES (v1, v2, ...);
```

More generally, you can insert multiple tuples (into a single table), and you can specify certain fields for which you have values:

```
INSERT INTO tablename (A1, A2, ...)
VALUES (v1, v2, ...), (w1, w2, ...);
```

The above will insert two rows into the table specified; the first row will get attribute A1 = value v1, A2 = v2, etc., and the second row will get A1 = w1, A2 = w2, etc. Any attributes which are not in the attribute list for the INSERT will get NULL unless a default value has been provided in the table definition.

You can also insert multiple tuples resulting from a SELECT query into a table with compatible attribute types:

```
INSERT INTO tablename (A1, A2, ...)
SELECT B1, B2, ...  ;
```

Example:

```
INSERT INTO mines_eecs_faculty (name,
office, email)
VALUES
('Painter-Wakefield, Christopher',
'BB 280I','cpainter@mines.edu');
```

Delete

A DELETE statement simply eliminates tuples matching the WHERE clause of the DELETE:

```
DELETE FROM tablename
WHERE condition;
```

If no condition is provided, then the statement deletes all rows from the table.

Example:

```
DELETE FROM mines_courses
WHERE instructor = 'Painter-Wakefield,
Christopher';
```

Update

Update is used to modify existing data by changing the value of one or more attributes for one or more tuples in a table. The UPDATE query looks like

```
UPDATE tablename
SET A1 = e1, A2 = e2, ...
WHERE condition;
```

The UPDATE query can be the trickiest to understand in that it allows the updating of multiple rows (depending on the WHERE condition), and each row can be updated in an independent fashion (depending on the expressions used in the SET clause).

When the UPDATE query is executed, the database will iterate over all tuples in the specified table that match the WHERE condition; the SET clause is then applied to each matching tuple individually. For each assignment in the SET clause of the form `A = e`, A is the name of the attribute receiving the update, and e is some expression that evaluates to a value that is type-compatible with A. Because the expression can include constants, functions, other attribute values, or even sub-queries, UPDATE is quite powerful.

Here's a simple example, affecting only one row of a table:

```
UPDATE mines_eecs_faculty
SET office = 'BB 280I',
email = 'cpainter@mines.edu'
WHERE name = 'Painter-Wakefield,
Christopher';
```

Here's a more complex example; suppose we have a person table composed of first and last names. When we want the full name as a single string, we can compute it on the fly using the concatenation (||) operator, but that might be tedious; instead, we might choose to maintain an additional column for full name. The update query might, perhaps, run on a regular schedule to update any full names of persons entered in the database since the last update. The full name column may be overwritten (some people might prefer a different form for their full name), so our update query should ignore rows that already have a full name:

```
UPDATE person
SET fullname =
firstname || ' ' || lastname
WHERE fullname IS NULL;
```

Note that the RHS of the assignment in the above query is an expression composed of concatenated attributes in the table; the attribute values come from the row that is being updated.

Bulk Loading of Data

While it is possible that your database may start empty and be filled one tuple at a time as data comes in, the more common case has you starting with some body of data from a previous system in a different format. In this case, while you could write code to generate INSERT statements for all of the data, running such code (and executing the resulting queries) is tedious and inefficient. Most DBMSes therefore provide some method for bulk-loading of data.

In the PostgreSQL database, the provided mechanism is the COPY command. The COPY command looks like

```
COPY tablename (A1, A2, ...)
FROM filename
WITH (option1, ...);
```

This specifies a filename from which data will be read, and a tablename into which the rows of data will be inserted (optionally specifying the attribute names corresponding to the values in the file). The options specify information about the formatting and encoding of the file.

The COPY command is limited in that it expects the filename to be the full path of some file on the database server's filesystem, which isn't always accessible to developers. Fortunately, the psql utility provides a command for utilizing the COPY command from a remote machine. In psql, you can issue the \COPY command; other than the beginning backslash, this command has the same syntax as the PostgreSQL COPY command. From psql, the filename needs to include the full path or the path relative to the working directory (the directory from which psql was launched).

The command below might be used to fill a table named 'foo' with values from a comma-separated values file named 'bar.csv':

```
\COPY foo FROM 'bar.csv'
WITH (FORMAT csv, ENCODING 'utf-8');
```

CSCI 403: Databases
6 - Subqueries

Subqueries

Also known as *nested queries*, subqueries allow you to use the results from nested SELECT queries in the SELECT, FROM, and WHERE clauses of another SELECT query, in the WHERE clause of a DELETE query, and in the WHERE and SET clauses of an UPDATE query.

Evaluating a Subquery

Recall that a SELECT query returns a relation, i.e., a set of tuples. In the case of a subquery we need to separate the possible results into those with no tuples, those with exactly one tuple, and those with multiple tuples. Some uses of subqueries require zero or one result, whereas others can handle multiple results.

Another way of looking at this is that a subquery may result in:

- A relation
- A tuple
- A (scalar) value
- NULL

Subqueries in WHERE

Most applications of subqueries tend to be in the WHERE clause of some outer query. Note that WHERE clauses occur in SELECT, DELETE, and UPDATE queries, and thus we can use subqueries in all of those cases.

IN and NOT IN

If we view the result of our subquery as being a relation (e.g., a set of tuples of any size), then the primary use of the subquery is with the operator IN. IN tests the tuples of a subquery result for a value match with some attribute or expression in the main query, resulting in a Boolean TRUE if a match is found, and FALSE otherwise. The expression being tested can be a scalar or a tuple, and the rows in the subquery result must match the expression in degree and type.

Example:

```
SELECT course_id FROM mines_courses
WHERE instructor IN
(SELECT name FROM mines_eecs_faculty);
```

The subquery is given in parentheses, and in this instance, returns rows with a single attribute. If we wish to compare tuple values instead, the query might look like (assuming we had separate last and first names in our tables):

```
SELECT course_id FROM mines_courses
WHERE (instr_first, instr_last) IN
(SELECT first, last FROM
mines_eecs_faculty);
```

Here we must enclose the attributes being compared in parentheses to mark them off as a tuple.

Note that the above query can equivalently be expressed using a join:

```
SELECT course_id
FROM mines_courses AS mc,
mines_eecs_faculty AS mef
WHERE mef.name = mc.instructor;,
```

although it is equivalent in this case only because each choice of instructor can match at most one row in mines_eecs_faculty (otherwise the two queries would produce the same values, but in different multiplicities). Adding DISTINCT to both queries would ensure their equivalence, and in general, subqueries in the WHERE clause can often be rewritten using joins.

If we change the query to use NOT IN instead of IN, we select the set of rows where the expression does *not* match any results from the subquery. (This can be equivalently expressed using an OUTER JOIN and an IS NULL test (see next lecture for outer joins), but the NOT IN query may be easier to write and understand.)

Comparison with Single Tuple or Scalar Result

If a subquery can be guaranteed to return zero or one rows, then a number of additional options become available. In particular, the result of such a query can be used with comparison operators, particularly = and <>. For example, to obtain faculty information about EECS faculty who do not teach CSCI 403, we can do:

```
SELECT * FROM mines_eecs_faculty
WHERE name <>
(SELECT instructor FROM mines_courses
WHERE course_id = 'CSCI403');
```

When the returned value is scalar and comparable, we can also use comparison operators such as <, <=, >, and >=.

Note that our query will fail if the subquery returns more than one row; the SQL engine will report an error. In the above query, we can produce the error by replacing 'CSCI403' with 'CSCI262', which has two rows in the mines_courses table.

When the subquery returns 0 rows, then the return values are interpreted as NULL, and thus will cause any comparison to be false.

Correlated Subqueries

In a correlated subquery, rows in the nested query are selected using some value from the outer query in the nested query's WHERE clause. This results in a correlation between the inner query and the tuples selected in the outer query. For example:

```
SELECT DISTINCT instructor, course_id
FROM mines_courses AS mc1
WHERE course_id IN
 (SELECT course_id
 FROM mines_courses AS mc2
 WHERE mc2.course_id = mc1.course_id
 AND mc2.instructor <>
mc1.instructor);
```

To understand what this query is doing, we must first understand how the SQL engine processes a correlated subquery (at least conceptually). Conceptually, you can think of the correlated subquery being executed once for each row of the outer query. So we iterate over every row in mines_courses and see if the course_id is in the result set from the nested query. Inside the nested query, we have access to the values of attributes in the outer query for the current row. In this case, we look for rows in mines_courses where the course_id matches the course_id in the currently iterated row, but where the instructor does not match the instructor in the currently iterated row. That is, we only take rows in the outer query when there exists a row in mines_courses which matches in course_id but not in instructor. In effect, this query is finding out the instructors and courses where different instructors teach different sections.

Once again, this query could be expressed using a join of mines_courses to itself. Which you use may depend on a variety of factors; which is most clear to understand, which one is easier to express in application software, etc.

EXISTS and UNIQUE

In the above query, we didn't really need to check to see if course_id was in the result set for the correlated subquery; it would suffice for there to be any result whatsoever (1 or more rows). The EXISTS operator tests a subquery for a non-zero number of rows in the result (NOT EXISTS tests for zero rows). The above query could be rewritten using EXISTS as:

```
SELECT DISTINCT instructor, course_id
FROM mines_courses AS mc1
WHERE EXISTS
 (SELECT course_id
 FROM mines_courses AS mc2
 WHERE mc2.course_id = mc1.course_id
 AND mc2.instructor <>
mc1.instructor);
```

Yet another operator that can be applied to a subquery is UNIQUE. This operator simply tests to see if the result tuples from a query are all unique (e.g., that the result is a set rather than a multiset).

Subqueries in FROM

Since the result of a SELECT query is simply a relation, we can treat a subquery as if it were an unnamed table, and select from it. For a trivial

example,
SELECT course_id FROM
 (SELECT course_id, instructor
 FROM mines_courses) AS mc
WHERE mc.instructor LIKE 'Painter%';

Subqueries in SELECT and SET

You can use a subquery returning a single (scalar) value in other situations where a scalar expression is appropriate. For instance, you can use a subquery to obtain a value inside the SELECT clause of a query, for instance:

```
SELECT instructor, course_id,
 (SELECT office
 FROM mines_eecs_faculty AS mef
 WHERE mef.name = instructor) AS
office
FROM mines_courses;
```

It is also occasionally useful to use a subquery to retrieve a value that you want to set some column to in an update query (typically using a correlated subquery):

```
UPDATE table1 AS T1
SET attribute =
 (SELECT val FROM table2 AS T2
 WHERE condition on attributes of T1,
T2)
WHERE condition on T1;
```

Note that the above is uniquely useful in that it acts kind of like a join inside an UPDATE query.

CSCI 403: Databases
7 - Aggregate Functions and GROUP BY

Aggregate Functions

SQL has a number of *aggregate* functions - functions which summarize a whole relation or part of a relation. The aggregate functions are:

- COUNT - counts the number of non-NULL entries in a column, or non-NULL tuples in a relation
- SUM - adds the non-NULL entries in a column
- MAX - finds the maximum non-NULL entry in a column
- MIN - finds the minimum non-NULL entry in a column
- AVG - takes the average of the non-NULL entries in a column

Note that each of these results in a scalar result. Examples:

```
SELECT COUNT(*) FROM mines_courses;
SELECT COUNT(instructor) FROM
mines_courses;
SELECT AVG(max_credits) FROM
mines_courses;
```

Grouping

Grouping lets us compute aggregates on subgroups of rows, as organized by distinct values of some subset of attributes of the relation. The general form of a grouping query is:

```
SELECT A1, A2, ..., FN1(A3), FN2(A4),
...
FROM table1, table2, ...
WHERE conditions
GROUP BY A1, A2, ...;
```

In the above, FN represents some aggregate function. In a GROUP BY query, the most important thing to understand is that the set of attributes SELECTed in addition to the aggregate function expressions must be a subset of the attributes in the GROUP BY clause. That is, if we SELECT an attribute, we must also GROUP BY it, or the query is invalid.

The result of the GROUP BY query is a set of tuples with the distinct values of the selected attributes together with aggregate functions computed on the subset of tuples from the relation which share the distinct values. For instance,

```
SELECT instructor, COUNT(*)
FROM mines_courses
GROUP BY instructor;
```

will return all distinct instructors (including NULL as a distinct "value") and the count of tuples in the mines_courses table for each instructor. (In the case of this query, it is a count of the distinct course CRNs associated with an instructor - not exactly a measure of how many courses an instructor teaches.)

Grouping is applied *after* any WHERE conditions are applied.

GROUP BY can be combined with ORDER BY, but only if ORDER BY is applied to either attributes in the GROUP BY clause or to any (not necessarily SELECTed) aggregate functions on the relation. (Also, renaming of aggregate function result columns is often a good idea for readability in the result, and the column alias can be used in the ORDER BY clause.) For example, either of the following is valid:

```
SELECT instructor, COUNT(*)
FROM mines_courses
GROUP BY instructor
ORDER BY instructor;
```

and

```
SELECT instructor, COUNT(*) AS
count
FROM mines_courses
```

```
GROUP BY instructor
ORDER BY count;
```

Another example, showing that you can ORDER BY an aggregate function not SELECTED on:

```
SELECT instructor, COUNT(*)
FROM mines_courses
GROUP BY instructor
ORDER BY SUM(max_credits), instructor;
```

A final example, answering the question: "What MWF timeslots have the most courses scheduled at Mines?"

```
SELECT begin_time, COUNT(*)
FROM mines_courses_meetings
WHERE days = 'MWF'
GROUP BY begin_time
ORDER BY COUNT(*) DESC;
```

HAVING

Sometimes it is desirable to filter grouped data based on some condition which can be applied to the aggregate functions computed on the groups. The HAVING clause acts like a WHERE clause which is applied *after* grouping (as opposed to WHERE, which is applied before grouping). For instance, suppose we want to list only instructors responsible for more than 5 CRNs:

```
SELECT instructor, count(*)
FROM mines_courses
GROUP BY instructor
HAVING count(*) > 5;
```

You can have both WHERE and HAVING conditions in your query.

CSCI 403: Databases
8 - Miscellaneous SELECT queries, More DDL

Join Clauses

Inner Joins

Joins are conveniently constructed using the WHERE clause of a SELECT query; however, another approach is available, which is to use a JOIN clause.

Example:

```
SELECT course_id, instructor, office
FROM mines_courses JOIN
mines_eecs_faculty ON instructor =
name;
```

is equivalent to

```
SELECT course_id, instructor, office
FROM mines_courses, mines_eecs_faculty
WHERE instructor = name;
```

This most common join is formally known as an "inner join". Thus, the above query can also be expressed as

```
SELECT course_id, instructor, office
FROM mines_courses INNER JOIN
mines_eecs_faculty ON instructor =
name;
```

Outer Joins

As we discussed when we looked at joins the first time, an inner join will only return rows that match in the two joined tables; in the example above, we only get rows from mines_courses with matching instructor names in mines_eecs_faculty; similarly, we only get rows from mines_eecs_faculty with matching names in mines_courses (in the latter case, it happens to be all rows).

Sometimes it is desirable to see all rows from one table, joining in data from another table *where available*. The technique is called an outer join, and comes in three flavors: left, right, and full.

A left outer join uses the same syntax as above, but with the keywords LEFT OUTER replacing INNER[1]. In the case of a left outer join, all rows in the table on the left hand side of the JOIN clause will be returned; where data is available in the table on the right hand side, it too will be returned, with NULLs filling in when data is not available.

E.g.,

```
SELECT course_id, instructor, office
FROM mines_courses LEFT OUTER JOIN
mines_eecs_faculty ON instructor =
name;
```

This shows all rows from mines_courses, with office info for EECS faculty only (and NULLs for everyone else). If we instead do

```
SELECT course_id, name, office
FROM mines_courses RIGHT OUTER JOIN
mines_eecs_faculty ON instructor =
name;
```

we get a right outer join, and thus all rows from mines_eecs_faculty together with any matching info from mines_courses. Note that this does *not* select all rows from mines_courses!

If we want to select all rows from both tables, we can use a full outer join. As you might expect, the keyword FULL replaces RIGHT in the above query. Now we get all rows from mines_courses and all rows from mines_eecs_faculty, with the matching tuples showing data from both tables as usual.

Natural Joins (Equijoins)

Another join type lets us shorten the join clause a bit in the case when the join conditions simply equate all attributes sharing the same name in

[1]In Oracle, there is a special operator, (+), which can be used in a WHERE clause join to effect outer joins. Watch out for this if you work with Oracle.

both tables. For example, mines_courses and mines_courses_meetings are only usefully joined by equating the crn attribute in both tables; since this is the only attribute which appears in both tables, we can do a natural join:

```
SELECT * FROM mines_courses NATURAL
JOIN mines_courses_meetings
WHERE instructor = 'Painter-Wakefield,
Christopher';
```

Set operations

Since we can view a relation as a set of tuples with the same schema, it is natural to expect that we can apply set operations to relations, and so it is. Specifically, SQL provides the operators UNION, INTERSECTION, and EXCEPT. These provide set union, intersection, and difference, and can be applied to the results of two or more SELECT queries.

For a trivial example, a UNION can replace an OR in a WHERE clause (this is not a recommended use, I provide it only for example):
SELECT * FROM mines_courses WHERE course_id LIKE 'CSCI%'
UNION
SELECT * from mines_courses WHERE course_id LIKE 'LAIS%';

Note that the two relations *must* have compatible schemas, that is, all the types must match. The *names* of the resulting relation are taken from the first query in the union; names in the second (and third, fourth, ...) queries do not have to match (just the types of the columns must match).

Note that all of the set operators will return a true set, not a multiset, unless you add the keyword ALL after the set operator. That is, query1 UNION query2 will result in a relation with all duplicate tuples removed, whereas query1 UNION ALL query2 retains any duplicates.

Other SELECT voodoo

There are many more techniques we cannot cover in this course, such as WITH queries, recursive queries, etc. Some of these will be covered in the textbook, so you can at least get some picture of their use.

More DDL

ALTER TABLE

ALTER TABLE lets us modify tables that have already been created. Here are a few of the more common uses:

Adding a primary key:

```
ALTER TABLE tablename ADD PRIMARY KEY
(attr1, attr2, ...);
```

Adding a foreign key:

```
ALTER TABLE table1
ADD FOREIGN KEY (attr1, attr2, ...)
REFERENCES table2 (attr1, attr2, ...);
```

Adding a column:

```
ALTER TABLE tablename
ADD COLUMN columnname type;
```

DROP

`DROP objecttype objectname;` lets us drop tables, views, constraints, etc.

Views

A view is like a saved query given a name; you can select from it just like any other relation, but underlying it is a SELECT query, not a simple table. So views can be as complex as desired (with joins, unions, whatever).

Syntax is identical to CREATE TABLE AS, just VIEW instead of TABLE:

```
CREATE VIEW viewname AS SELECT ...;
```

CSCI 403: Databases
9 - Relational Database Design and Entity-Relationship Diagrams

Database Design

The notion that a database should be designed (rather than just thrown together in an ad-hoc fashion) leads to the development of Entity-Relationship Diagrams (ERDs) in around 1976 by Chen.

We can consider three different levels at which design activities occur:

- Conceptual - understanding the data *entities* and *relationships* among them. This is a high-level design which may be helpful for communicating with non-technical stakeholders about the data model. ERDs live at this level. (While ERDs may help with communication, it does require learning a new visual language.)
- Logical - mapping from an ERD to a SQL (or other) DBMS. This is concerned mainly with the realization of a data model into an actual database schema.
- Physical - the bare-metal stuff: where files, indexes, etc. should live on disk, network architectures, etc.

We will only be concerned in this lecture with the conceptual level of design. The following lecture will take us through the logical level with an algorithm that maps an ERD to a relational schema.

ERD Components

The three main components of an ERD are the entities, attributes, and relationships.

Entities - things or objects with independent existence, such as persons, products, companies, courses. Entities are the *nouns* of the ERD.

Attributes - the properties describing an entity.

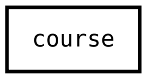
Relationships - the way entities interact or refer to each other. Relationships are the *verbs* of the ERD. For example, a person **supervises** a department, an instructor **teaches** a course, a person **works on** a project.

ERD: Visual Language

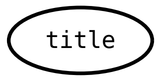
In this section we will develop an ERD for courses at Mines, modeling something very close to the actual tables we have in the database. (The ERD we develop is actually a better data model. The database tables need a slight refactoring.) Along the way we will examine all the elements of the visual language for an ERD.

Entities and Attributes

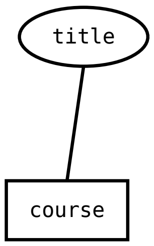
Let's start with an entity. A course is an entity in our database, as something which has a real existence. Entities are visualized using a rectangle containing the name of the entity:



An entity has attributes which describe it; for example, a course has a course title. Attributes are ovals with the name of the attribute:



Attributes are connected to their entity via straight lines:



Attributes may be designated as keys, which has basically the same meaning as a key in the relational model of the database: a key is a value which is unique for all instances of an entity (therefore uniquely identifying an instance). Key attributes have their names underlined.

Attributes may also be composite. A composite attribute represents some property of an entity which is not itself atomic, but which is composed of smaller sub-attributes. For example, a name attribute for a person entity might further break down into first and last names. In the diagram we are developing, the course entity has a composite key, designation, which is composed of course id and section id. Figure 1 shows the full course entity with all of its attributes. The reason in this diagram for the composite attribute is that keys cannot be broken up over multiple attributes in an ERD. Each underlined attribute is assumed to be a key on its own; therefore, a key with multiple attributes must be combined into a composite key attribute.

Attributes may also be multivalued. Multivalued attributes are shown as ovals with a double border; the attribute meetings in 1 is a multivalued attribute. A multivalued attribute is one which may have multiple values for a single instance of an entity. For example, a car entity might have a multivalued attribute for color, since some cars have more than one color; a person might have multiple titles (or even names). In the case of a course, a course at mines may meet in different rooms at different times on different days. I've chosen to model this as a composite multivalued attribute for our course entity.

Other modeling choices are available; for instance, a meeting could be modeled as a weak entity owned by course. A weak entity is one which is not wholly defined except in relationship with another entity. In particular, a weak entity will not have any keys, but may have a partial key. Weak entities are designated using a

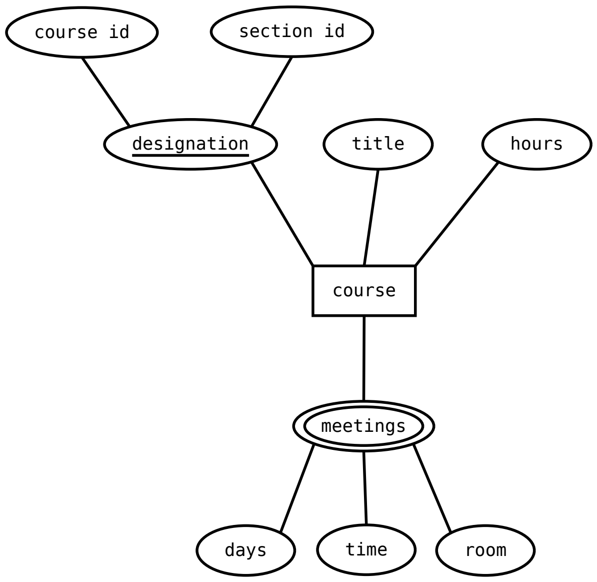


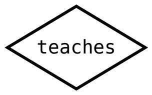
Figure 1: The full course entity

rectangle with a double border (see the section entity in figure 4). We will return to the weak entity a bit later when we refine our model.

Finally, a model may have derived attributes. A derived attribute is one which can be computed from other attributes or components of the model. For instance, age might be derivable from birthdate. A derived attribute is represented as a dashed oval.

Relationships

A relationship is drawn using a diamond shape containing a descriptive relationship name:



Relationships must be attached to two or more entities with straight lines (we focus on the case of two entities for this lecture). The connected entities are the ones participating in the relationship. The teaches relationship connects the instructor entity and the course entity in figure 2.

Relationships may also denote

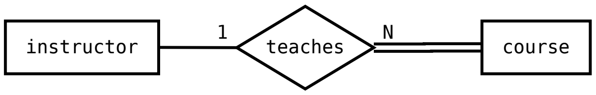


Figure 2: The teaches relationship and its participating entities

cardinality ratios by marking each connection to the relationship with 1 or N (or M). The cardinality ratio gives the maximum number of instances of the entity participating on each side of the relationship. For instance, 1 on the instructor side of the teaches relationship tells us that a course has at most 1 instructor. An N (or M) represents "many"; an instructor may teach 0 or more courses. The possible cardinality ratios are 1:1, 1:N, and N:M. Cardinality may be specified more exactly using ranges (see book for details).

A closely related concept is the participation of an entity in a relationship. An entity is said to have total participation in a relationship if the existence of an instance of the entity depends on there being a related instance on the other side. Total participation is represented using a doubled line connection on the side with total participation, whereas partial participation uses a single line. For example, in figure 2, we see that a course must have an instructor (while the opposite is not true). In some sense, whereas cardinality ratios give us a maximum, participation constraints give us a minimum.

A relationship connected to a weak entity, in which the other side of the relationship is the owning entity, is called an identifying relationship, and is drawn using a double border (see the has sections relationship in figure 4).

Finally, a relationship may also have associated attributes. These are represented using the usual attribute ovals attached by straight lines to a relationship diamond. Relationship attributes represent some properties unique to the relationship which don't properly belong to any of the connected entities. There are no examples of this in our example diagram, but the book gives an example.

Complete Example

A full example based on what we've discussed so far is given in figure 3. The course entity has already been thoroughly described. The instructor entity is fleshed out with attributes; the instructor's name is a key attribute, and other attributes are office and email. An additional entity, department, is shown in relationship with instructor. This relationship is N:M, signifying that an instructor can belong to more than one department, while a department may have more than one instructor within it. (This is a common situation at other institutions; not sure if it happens at Mines.)

The ERD in figure 3 is a fairly accurate reflection of the relations in the course database. However, an initial design is rarely "perfect" - modeling is an iterative process. In particular, this design feels as if it is combining two independent concepts into the course entity; courses and sections. As we will see when we discuss normalization, there are clues to bad design in the database, such as redundancy in some of the fields; for instance, a course should have a course title that doesn't vary by section, but that isn't well reflected in the design, and in the database we see that course title is repeated information - it is redundant over all sections of the course. Ditto for course hours (and description, if we had such, etc.)

This suggests that sections should be separated from courses. Since sections are not truly independent, however, it makes sense to model them as a weak entity owned by a course (no section exists independent of a course). See figure 4 for the final (for now) model.

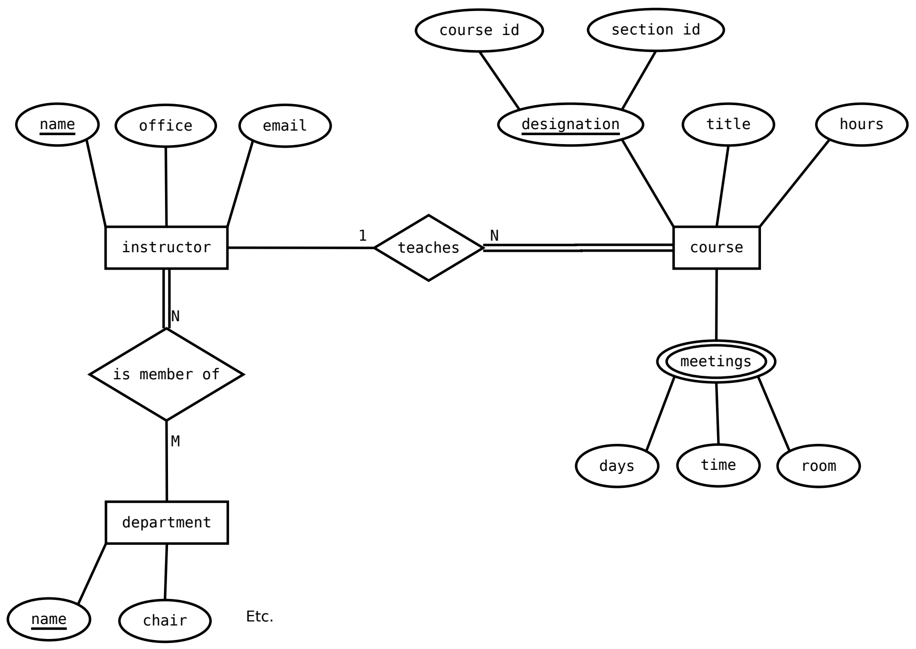


Figure 3: A complete example for a Mines courses database

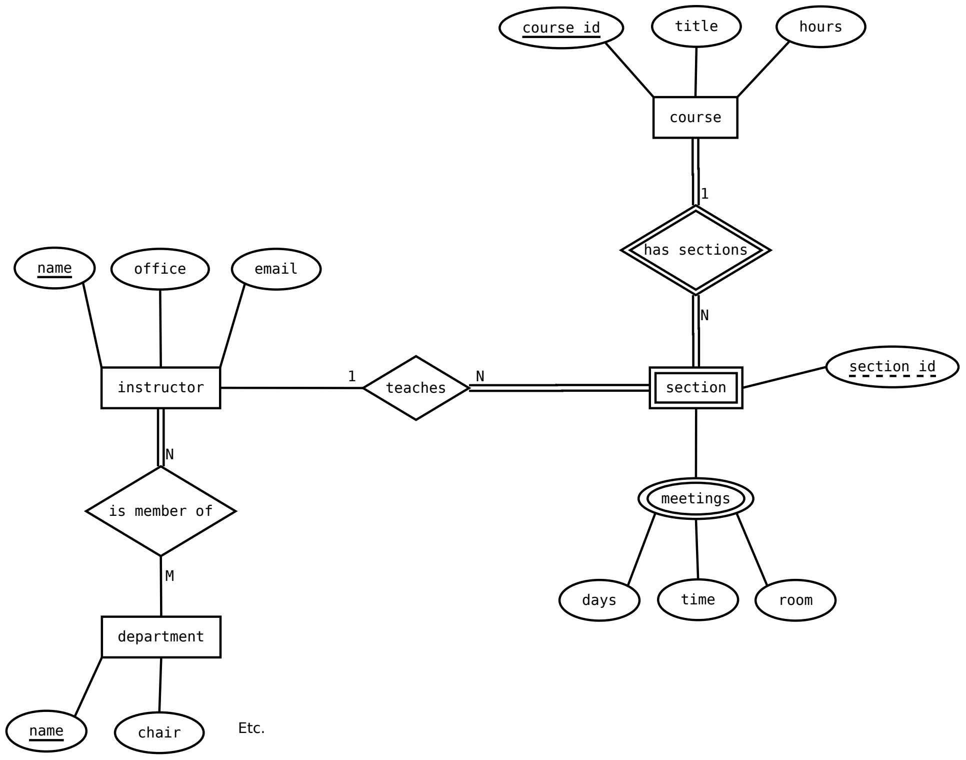


Figure 4: A refined model for the Mines courses database

Final Notes

One thing we didn't address above was the possibility that a relationship can be recursive; that is, an entity can be related to itself. An example from the book is that of an employee entity which also tracks the supervisor information about each employee. Since supervisors are themselves employees, we model this situation with 1:N relationship from the employee table to itself. For further clarification, the lines on each side of the relationship can be annotated with a description of the nature of the relationship, e.g., "Supervisor" and "Supervisee".

Also not covered (except to mention its existence) is the possibility of higher-order relationships, also known as n-ary relationships. These are somewhat rare, although certainly not impossible. An example of this is a relationship between three entities: vendor, part, and project. The relationship represents the situation that a particular part may be supplied by a particular vendor for a particular project at a company. See the book for more discussion of this topic.

CSCI 403: Databases
10 - ERD to Relational Schema Mapping

The Algorithm

In order for our ERD to be useful, we need to turn it into a relational schema for our database. This can be accomplished in a fairly straightforward fashion by following a few simple steps. The book provides the following seven-step algorithm, here presented in brief with examples drawn from the ERD created in the last lecture (see figure 1).

1 - Regular entities

Regular entities in the ERD become relations in the relational schema with all simple attributes of the entity becoming attributes in the relation. "Simple attributes" means that we take only the components of composite attributes, and we do not include multivalued or derived attributes. Multivalued attributes will be addressed in step 6 below. Note that additional attributes may be added on to this relation in later steps.

Choose one key of the entity and make its corresponding attribute(s) the primary key for the relation. Other keys can simply be created with unique constraints.

Looking at figure 1, we have three regular entities: course, instructor, and department. These three have only simple attributes, and single keys, thus the production of the schema is straightforward, and we obtain three initial relations:

course:
(<u>course_id</u>, title, hours)
instructor:
(<u>name</u>, office, email)
department:
(<u>name</u>, chair, . . .)

Here we have underlined the primary keys. Note that one thing not captured in the ERD is the data type, so that is one design decision to make in the transition from ERD to schema. Also note that we have complete discretion to change entity and attribute names as necessary to make them valid for use in the database or to meet our own style requirements.

2 - Weak entities

Weak entities in the ERD become relations in the relational schema, again with all simple attributes becoming attributes in the relation. In addition to the simple attributes, the primary key attribute(s) of the relation arising from the owning entity are added to the new relation as a foreign key referencing the owning entity relation. The primary key of the new relation is the combination of the primary key borrowed from the owning entity and the partial key of the weak entity.

Looking at our example, we have one weak entity, section. It is owned by course and has a partial key attribute section id. It also has a multivalued attribute meetings, but since it is multivalued, we ignore it for now. We create a table named section:

section
(<u>course_id, section_id</u>).

The course_id and section_id attributes together form the primary key of the new relation. The course_id attribute is a foreign key referencing course_id in the course relation.

3 - 1:1 relationships

Let R be a 1:1 relationship between entities S and T. There are three basic choices (depending on the participation constraints). (Note: from now on, I will speak of the entities and the relations generated from them interchangeably. So S stands both for the entity S and for the relation created from it in steps 1 and 2.)

(a) Suppose S has total participation in R (the case when T has total participation is symmetrical). Then the first option we have is

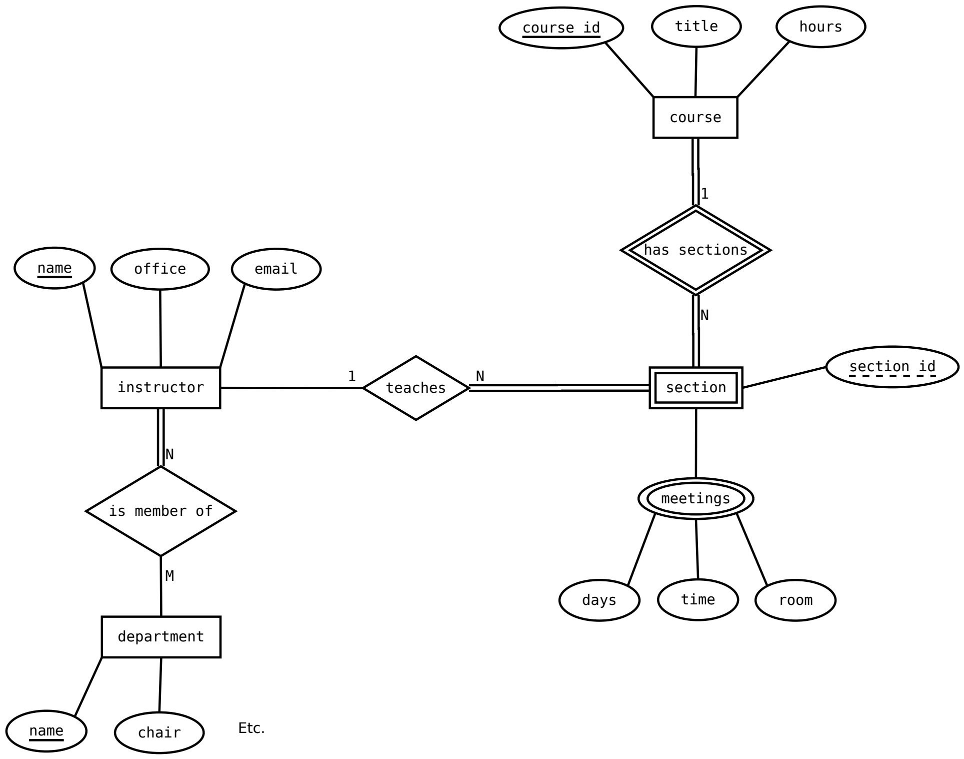


Figure 1: A model for a Mines courses database

to take the primary key of T and add it as an attribute in S. The borrowed attribute in S is made a foreign key referencing the primary key of T.

(b) If both S and T have total participation in R, then every tuple in S will correspond with a tuple in T and vice-versa. Thus it is safe, if desired, to merge S and T into one relation with all the attributes from both S and T. However, option (a) is preferred, particularly if the two entities model different concepts (simply choose one entity to take the primary key of the other).

(c) A third option, not recommended unless there is no total participation on either side, is to create a third relation as a cross-reference (or lookup) table. This option will be discussed further below when we discuss N:M relationships.

In our example, there are no 1:1 relationships. However, option (a) is essentially the same as for option (a) in the 1:N relationship mapping, so we will move on.

4 - 1:N relationships

Let R be a 1:N relationship between entities S and T, with T being on the 1 side and S on the N side.

(a) If S has total participation in R (which is more often the case than not), then once again we can take the primary key from T and add it to the relation S as a foreign key referencing back to T. Note that this is not a symmetrical option: since S can have many tuples for each tuple of T, there can be more than one primary key value in S which would have to be associated with a single tuple of T.

(b) The second option is the same as option (c) above: create a cross-reference table.

Our example has two 1:N relationships. The "has sections" relationship is already taken care of; the primary key from course is already included in the section relation, so nothing more needs to be done. The teaches relationship is also 1:N, with instructor being on the 1 side. Thus we take the primary key from instructor (name) and add it to the section relation. Noting that name is not particularly descriptive, we'll take the liberty of renaming the attribute in the section relation to instructor_name. The modified section relation now looks like:

section
(<u>course_id, section_id</u>, instructor_name).

We make instructor_name a foreign key referencing name in instructor. The previously created primary and foreign keys are unchanged.

5 - N:M relationships

In the case of N:M (many-to-many) relationships, there is no way to add a key from one relation to the other to represent the relationship. The only possibility in this case is to create a cross-reference, or lookup table. The table contains primary key attributes from both relations, and each entry in the table is a pair representing the existence of a relationship between a tuple on one side with a tuple on the other side.

More formally, suppose S and T are the two relations participating in the relationship, with S having primary key P_S and T having primary key P_T. Then we create a third relation, Q, with attributes P_S and P_T. The primary key for the new relation is composed of both attributes. The attribute P_S is set as a foreign key referencing P_S in S, and likewise P_T is a foreign key referencing P_T in T.

In our example, "is member of" is an N:M relationship between instructor and department. We must create a new relation, the name of which is up to us; in general follow the style that your organization uses. I will take the approach of calling the new relation S_T_xref (or in our example, instructor_department_xref). Note that the primary keys of both instructor and department are named "name", so at least one must be renamed in the new table; since both are non-descriptive, I will rename them both.

Here's the new table:

instructor_department_xref:
(<u>instructor_name, department_name</u>),
with instructor_name being a foreign key referencing name in instructor, and department_name being a foreign key referencing name in department.

Now we can easily retrieve all departments of an instructor or all instructors of a department by joining the three tables and applying the necessary where condition, e.g.

```
SELECT i.name
FROM instructor AS i,
department AS d,
instructor_department_xref AS x
WHERE d.name = x.department_name
AND i.name = x.instructor_name
AND d.name = 'Computer Science';
```

6 - Multivalued Attributes

Mapping a multivalued attribute onto a relational schema is essentially the same as mapping a weak entity, treating the multivalued attribute itself as the partial key and only attribute of the weak entity. That is, we create a new relation which contains as attributes the (simple components of) the multivalued attribute plus the primary key attribute from the owning relation.

The primary key of the resulting relation is composed of all attributes. The primary key attribute from the owning relation, as usual, is made into a foreign key referencing the owning table primary key.

In our example, meetings is a multivalued attribute of section. The meetings attribute is also composite, so in our new table we will have attributes for each component of meetings. We also borrow the primary key from the owning relation; in this case, the owning relation is section. Looking back through our work so far, we see that the primary key of the owning relation is also composite, made from course_id and section_id. Accordingly, we create a table (again name is up to us):

section_meetings
(<u>course_id, section_id, days, time, room</u>),
with (course_id, section_id) as a foreign key back to (course_id, section_id) in section.

7 - n-ary relationships

In general n-ary relationships (relationships between 3 or more entities) must be modeled using a cross-reference table. There are some subtleties that can arise (see the book for details), but generally you can get away with creating a cross-reference table from the n primary keys of the participating relations.

Notes

In the above, I didn't address what happens when a relationship itself has attributes. What you do depends on the nature of the relationship and the choices you made in mapping the relationship onto the schema. If you chose a cross-reference table option, then adding the attributes into the cross-reference table is the simplest and most effective option. Otherwise, you will need to add the attributes into one of the participating relations, or create an additional table just for the relationship attributes to live in (with appropriate foreign key references).

CSCI 403: Databases
11 - The Relational Algebra

Relational Model Redux

We return to the theory of relational databases as developed by E.F.Codd in his 1970 paper. Recall that a relation is defined as a set of tuples, where a tuple is a set of values associated with named attributes. The relational algebra considers the useful operations that can be applied to relations and the resulting algebra of relations. These operations all have realizations in the SQL language (which came about substantially later than the algebra), although as we've already seen, SQL does not adhere tightly to the relational model (e.g., multiset relations).

There is also a relational calculus, which we will not cover in this course, but which is discussed in your textbook. The relational calculus forms part of the foundation of the SQL language.

Unary Operations

The relational algebra has two unary operations, projection and selection, which give rise to the SELECT and WHERE clauses (respectively) as well as a renaming operation which gives rise to the renaming and aliasing facilities in SQL (i.e., the AS keyword). There is notation associated with each of these operations, which lets us compactly write expressions in the algebra.

Selection

The Select operation chooses a subset of tuples from a relation according to a condition on the attributes of the tuples (think WHERE clause). In effect, the Select operation partitions the tuples into two sets according to whether they satisfy the condition, discarding the set of tuples which do not satisfy the condition.

Letting R represent some relation, the Select operation is notated as

$$\sigma_{condition}(R).$$

R can be a variable, as in

$$R = mines\_courses,$$

or it can be the bare name of a relation, or it can be an expression in the algebra which results in a relation.

For example,

$$\sigma_{course\_id='CSCI403'}(mines\_courses)$$

applies the selection operation to the mines_courses relation to obtain only those tuples where the attribute course_id is equal to the constant 'CSCI403'. Note that the result here is another relation, that is, a set of tuples which retain the attribute names from the original relation.

As with the WHERE clause of a SQL query, we can make compound conditions using the usual AND, OR, and NOT operations. A common extension is to also allow type-appropriate functions on attributes as well, such as a substring operation:

$$\sigma_{substring(course\_id\ from\ 1\ for\ 4)>=300}(mines\_courses)$$

selects tuples from mines_courses for courses at the 300 level or higher.

The selection operation is unary (applies to a single relation) and the condition applies to single tuples; no aggregate functions here!

Properties of the Select operator

- The degree of the relation (# of attributes) resulting is the same as the degree of the of the original relation.

- The number of tuples in the resulting relation is always less than or equal to the number in the original. The fraction of tuples selected is referred to as the selectivity of the selection condition.

- Selection is commutative:

$$\sigma_{cond1}(\sigma_{cond2}(R)) = \sigma_{cond2}(\sigma_{cond1}(R))$$

- A sequence of Select operations can always be converted into a single Select operation with the conjunction (AND) of all the Select conditions:

$$\sigma_{cond1}(\sigma_{cond2}(R)) = \sigma_{cond1 AND cond2}(R).$$

Projection

Whereas the Select operation partitions relations horizontally (by rows), the Project operation partitions relations vertically (by attributes). The Project operation is notated as

$$\pi_{attr1,attr2,...}(R),$$

where attr1, attr2, ... specifies the attributes of R to be retained in the resulting relation. Note that we can repeat attributes, so the resulting relation can have higher degree than the original.

For example:

$$\pi_{instructor,course\_id}(mines\_courses)$$

would result in a relation with only unique tuples of instructor and course_id from the mines_courses relation.

Note that the Project operator functions much like the SELECT part of a SQL query (while the Select operator functions like the WHERE clause).

Properties of the Project operator

- The number of tuples in the resulting relation is always <= the number of tuples in the original. It can be less than due to the elimination of duplicates (if the selected attributes do not form a superkey of the relation, for instance).

- The projection operation is not commutative, rather

$$\pi_{<list1>}(\pi_{<list2>}(R)) = \pi_{<list1>},$$

assuming list1 only contains attributes also found in list2 (otherwise the expression is malformed).

Renaming

Formally, the renaming operator lets us rename relations, attributes, or both as a unary operation - similar to using the AS keyword in SQL. The general form is

$$\rho_{S(B1,B2,...)}(R)$$

where S is the new name of the relation, and B1, B2, ... are the new names of R's attributes. S or the attribute list are optional. If the attribute list is included, it must match in degree the number of attributes in R, and the attributes will be renamed in the same order as the usual ordering of the attributes.

Sequences of Operations

All of the above unary operations can be nested, e.g.,

$$\rho_{(name,course\_id)}(\pi_{instructor,course\_id}(\sigma_{department='EECS'}(mines\_courses))),$$

which corresponds to (is not necessarily equivalent to) the SQL query

```
SELECT instructor AS name, course_id
FROM mines_courses
WHERE department = 'EECS';
```

Alternately, we can do a sequence of operations, giving a name to each relation (think variables) as we go:

$$\begin{aligned} R1 &= \sigma_{department='EECS'}(mines\_courses) \\ R2 &= \pi_{instructor,course\_id}(R1) \\ R3 &= \rho_{(name,course\_id)}(R2) \end{aligned}$$

Set Operations

Union, Intersection, and Set Difference (or Minus) can be applied as binary operations when both operands are relations with the same number and types of attributes. These work the same as UNION, INTERSECTION, and EXCEPT in SQL.

Notation:
Union: $A \cup B$
Intersection: $A \cap B$
Difference: $A - B$

Properties of Set Operations

- Union and Intersection are both commutative:

$$\begin{aligned} A \cup B &= B \cup A \\ A \cap B &= B \cap A \end{aligned}$$

- Union and Intersection are both associative:

$$\begin{aligned} A \cup (B \cup C) &= (A \cup B) \cup C \\ A \cap (B \cap C) &= (A \cap B) \cap C \end{aligned}$$

- Set difference is **not** commutative or associative!

- Intersection can be expressed in terms of Difference (and thus is not technically necessary for completeness):

$$A \cap B = A - (A - B)$$

Cartesian Product and Joins

The Cartesian (cross) product:

$$A \times B$$

has the same meaning in relational algebra as a cross-product join in SQL: the resulting relation has each tuple from A paired with every tuple from B. The attributes of the new relation are the concatenation of attributes from A and B. Thus if A has m attribute and B has n attributes, then $A \times B$ has $m + n$ attributes. The size of $A \times B$ is $|A| \times |B|$.

As in SQL, this operation has little utility on its own. Typically it is paired in relational algebra with a subsequent Select operation to eliminate irrelevant pairings of tuples. Since this is such a common pattern, another operation, Join, was created. The notation for the binary operation Join is:

$$A \bowtie_{condition} B$$

which equates to

$$\sigma_{condition}(A \times B)$$

Note: there are some variations in notation between the book and other sources I've looked at concerning joins and related operations. I will try to follow the book's notation where there are differences.

For example, we could perform the join

$$mines\_courses \bowtie_{instructor=name} mines\_eecs\_faculty.$$

When we have $A \bowtie_{cond} B$ and the join condition is of the general form
cond1 AND cond2 AND ...,
and cond$n$ is of the form $A_i \theta B_j$ where $A_i \in A$ and $B_j \in B$ and θ is one of the usual comparison operators (equals, less than, etc.), then the join is called a "theta-join".

When θ is the equality operation, the join is called an "equijoin". The nature of this join is such that the resulting relation will agree completely on every value between the pairs of columns participating in the join expression. Therefore, it is usually desirable to project away one of the duplicates.

In the frequent case in which the paired attributes have the same name in both relations, we can apply yet another special join: the "natural join". In the book, the notation for the natural join is

$$A * B$$

but other sources use $\bowtie$ with no conditions to mean a natural join.

E.g.,

$$mines\_courses \ * \ mines\_courses\_meetings$$

would join on the equality of the CRN attribute.

If the paired attributes do not have the same name, then a renaming must be done before applying the natural join, e.g.:

$$mines\_courses * \rho_{name=instructor}(mines\_eecs\_faculty).$$

Note that the natural join will equate all attributes of the same name in the two relations. Terminology: the join selectivity of a condition is the expected size of the join divided by the max size of the cross product.

Completeness

It can be demonstrated that the set $\{\sigma, \pi, \cup, \rho, -, \times\}$ forms a complete set of operators for the relational algebra, at least as originally formulated. That is, all other relational algebra operations can be expressed as some combination of operations using the above set.

Other Operations and Extensions

Division ($\div$) - this is a truly odd operator, which is difficult to explain the utility of, and which has no equivalent in SQL. Please read the book for more info. (Division is equivalent to a sequence of basic relational algebra operations.)

Aggregates/grouping: this is not expressible using the base relational algebra, but obviously has importance for SQL databases. The notation for doing aggregates and grouping is

$$< grouping\ attributes > \mathfrak{I}_{<function\ list>}(R)$$

Generalized projection: projection is generally extended to provide for functions on attributes, e.g.

$$\pi_{F_1(A_1),F_2(A_2),\ldots}(R)$$

Recursive or transitive closure: no real notation. This is some hardcore relational juju that actually does have a SQL equivalent, and may be useful for certain kinds of self-referential querying (e.g., getting a hierarchical company view from supervisor-employee relationships).

Outer joins: no LaTeXsymbols found for these! Look in book - they look like the join symbol with extensions.

CSCI 403: Databases
12 - Functional Dependencies and Normalization

Introduction

The point of this lecture material is to discuss some objective measures of the "goodness" of a database schema. The method relies on strong theory based in relational algebra and the notion of *functional dependency*. While it is possible to apply the theory (in theory) to generate a complete schema from some unnormalized starting point, for us the goal is to acquire some practical measures of "goodness" combined with an algorithm that can be easily applied to achieve a top-down design of our own schemas.

Informal guidelines

Some informal guidelines on what makes a schema "good":

- Clear semantics. Basically, the relations you create should make sense as independent units. You generally want clear entity distinctions and separation of concerns. This should mostly fall out naturally if you've done ER modeling first.

- Reducing redundancy. Data should be stored once and only once in the database (excluding foreign keys). This is not just a storage space consideration; redundancy in the database also leads to modification anomalies (more on this soon). For an example of redundancy in the data, see figure 1; for each instructor, office and email is repeated several times; for each course_id, title is repeated several times.

- Reducing NULLs. NULL values are undesirable for several reasons - they can be ambiguous in meaning, they don't play well with aggregate functions and joins, etc.

- Disallowing spurious tuple generation in joins. This has to do with not creating meaningless tuples when doing natural joins on tables.

Modification Anomalies

The redundancy in figure 1 occurs mainly because two (or more) concepts have been bundled together into one table. Specifically, instructor information is in the table together with course information, and these are independent and somewhat orthogonal pieces of data. Redundancy often goes hand in hand with a problem known as "modification anomalies", in which modifications to the database result in constraint violations or inconsistent data. Figure 1 demonstrates many opportunities for anomalies.

The various anomalies are easier illustrated than defined. An insertion anomaly in the table in figure 1 would occur, for instance, when we try to add a new faculty member who has not yet been assigned any courses. What values do we put in for course_id, section, and title? If we put in NULL values, we are possibly violating a key constraint (the only real key for this table is {course_id, section, instructor}). Also, when we later have course information for the faculty member, we would have to remember that there is NULL information in the table that needs to be overwritten.

Deletion anomalies are the inverse of insertion anomalies. What happens, for example, if we remove the last course taught by an instructor? The faculty member then ceases to exist! Similarly, if a faculty member leaves Mines, his or her courses vanish. This occurs because we are essentially bundling two separate entities into one table.

Finally, update anomalies are directly related to redundancy; if we try to update a faculty mem-

instructor	course_id	section	title	office	email
Painter-Wakefield, Christopher	CSCI403	A	Database Management	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	A	Data Structures	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	B	Data Structures	BB 280I	cpainter@mines.edu
Mehta, Dinesh	CSCI406	A	Algorithms	BB 280J	dmehta@mines.edu
Mehta, Dinesh	CSCI561	A	Theory of Computation	BB 280J	dmehta@mines.edu
Hellman, Keith	CSCI101	A	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI101	B	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI101	C	Intro to Computer Science	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI274	A	Intro Linux OS	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI274	B	Intro Linux OS	BB 310F	khellman@mines.edu

Figure 1: One possible relation with data about Mines courses and Mines faculty

ber's office information in the table in figure 1, we must make sure to update their information *everywhere* it occurs, not just in one or some rows, otherwise we introduce inconsistency into the data.

Spurious Tuples

Spurious tuples will occur primarily when a table has been factored incorrectly, or when a relation has been insufficiently or incompletely specified. For example, suppose we had two tables:

mines_courses
(instructor, course_id, section)

and

mines_faculty
(instructor, course_id, office, email)

In this example, if we join on the shared columns (instructor and course_id), we end up with tuples matching instructors to sections they do not actually teach.

Functional Dependencies

Our primary tool in the effort to eliminate redundancies and related anomalies from our database is something called a *functional dependency* (FD). A functional dependency is a constraint between two sets of attributes in a relation schema.

Definition:
Given a relation schema R and sets of attributes X and Y, then we say a functional dependency $X \rightarrow Y$ (read: X functionally determines Y, or Y is functionally dependent on X) exists if, whenever tuples t_1 and t_2 are two tuples from any relation $r(R)$ such that $t_1[X] = t_2[X]$, it is also true that $t_1[Y] = t_2[Y]$.

In words, if it is always true that whenever two tuples agree on attributes X they must also agree on attributes Y, then $X \rightarrow Y$.

For example, one functional dependency in the table in figure 1 is
instructor $\rightarrow$ {office, email}
if we assert that an instructor is always associated with only one office and email in our database.

This brings up an important point about functional dependencies; they are not properties of the *data*, but rather properties of the *world* which we impose on the data. That is, determining functional dependencies is a design activity (possibly informed by available data), which imposes a constraint on our database. A specific relation instance $r(R)$ may coincidentally have the property that all tuples agreeing on some attribute set X also agree on attribute set Y, but unless this *should hold true for any* $r(R)$ *that can exist*, we cannot say $X \rightarrow Y$.

Types of FDs

Just following the definition of an FD, we can make the observation that $X \rightarrow X$, trivially. More generally, if $Y \subseteq X$, then $X \rightarrow Y$ is called a *trivial* functional dependency. If $Y \nsubseteq X$ and $X \rightarrow Y$, this is called a *nontrivial* FD. Finally, if there is no overlap in attributes between X and Y, i.e., $X \cap Y = \varnothing$, then $X \rightarrow Y$ is called a *completely nontrivial* FD. We are mostly interested in completely nontrivial FDs.

Some nontrivial FDs in our example:
instructor $\rightarrow$ office
instructor $\rightarrow$ email
{course_id, section} $\rightarrow$ instructor
course_id $\rightarrow$ title

Properties

FDs can be viewed as a generalization of the notion of a superkey. Recall that a superkey is a set of attributes of a relation schema for which all tuples will be unique. A superkey is thus a subset of attributes of a relation that functionally determines the remaining attributes. Or the other way around, if $X \rightarrow Y$ and $X \cup Y = R$, then X is a superkey of R.

There are some inference rules (easily proved) that let us infer other FDs from an existing set of FDs:

- Splitting rule: If $A \rightarrow \{B_1, B_2\}$, then $A \rightarrow B_1$ and $A \rightarrow B_2$.
- Combining rule: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow \{B, C\}$.
- Transitive rule: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Additional rules can be found in the textbook. These inference rules, particularly the combining and transitive rules, are mostly of value to use in computing the closures of sets of attributes.

Closures

Given some set of functional dependencies F on a relation schema R, and some subset of attributes A, then the set $\{B_i : A \rightarrow B_i\}$ is called the *closure* of A and is denoted A^+. The closure plays a role in normalization (as we shall see), and can also be used in an algorithm to find all superkeys of a relation schema.

The closure A^+ is easily computed using the following algorithm:
Start with $S = A$. Clearly $A \rightarrow S$.
Repeat until no change:
if there exists an FD $X \rightarrow B$ in F such that $X \subset S$, then let $S = S \cup B$.

When no more expansions can be made to S, then $A^+ = S$. This algorithm simply applies the combining rule and transitive rules to expand the set S.

We can use the closure algorithm to find all superkeys of a relation schema as follows: simply construct every subset of the attributes of the relation schema and compute their closure. If the closure of a subset is equal to the entire set of attributes of the relation schema, then the subset is a superkey. While this may seem impractical, we can compute all superkeys relatively efficiently by starting with the smallest attribute subsets first; any superset of a superkey is a superkey, so we don't need to test supersets.

For our purposes, this algorithm has small utility, as we can typically identify the keys in our relation schemas by intuition. However, it is feasible to construct a normalized database from a single unnormalized relation schema automatically, if all functional dependencies are provided as part of the input. The algorithm to do this must compute keys, so the above algorithm plays a part in the larger algorithm.

Boyce-Codd Normal Form

The concept of normalization was developed to address the question of what a good relational design looks like. There are a series of *normal forms* which describe certain properties of a schema: first normal form (1NF), for instance eliminates any multivalued or compound (non-atomic) atrributes from the relation schema. Second normal form (2NF) and third (3NF) have their own rules, but these are mostly stepping stones on the way to the stronger normal forms which are our main focus: Boyce-Codd Normal Form (BCNF) and fourth normal form (4NF). Schemas in BCNF may be considered "good" for most purposes; 4NF is even stronger than BCNF, and thus are also "good".

Definition: A relation R is in Boyce-Codd Normal Form (BCNF) if for every nontrivial functional dependency $X \rightarrow A$ on R, X is a superkey of R.

Consider the example table in figure 1. If we start to examine some of the FDs we listed for this relation schema, we can quickly see that it is not in BCNF. For instance, consider the FD
instructor $\rightarrow$ office.

It is not difficult to see just from the sample data that instructor by itself cannot be a superkey of the relation. Therefore, we say that the FD
instructor $\rightarrow$ office
violates BCNF.

Decomposition algorithm

Fortunately, there are easy steps we can take to move closer to BCNF in our relation. There is a basic algorithm which starts with a relation schema not in BCNF and brings it to BCNF, which we detail now.

While not in BCNF:
- choose some R not in BCNF
- compute (super)keys of R
- choose some FD ($X \rightarrow Y$) in R violating BCNF
- (optional) Expand Y to be X^+
- Let $Z = R - (X \cup Y)$ be remaining attributes
- Replace R with two new relations R1 and R2:
— $R1 = \{X, Y\}$
— $R2 = \{X, Z\}$

The decomposition into R1 and R2 are accomplished by simply projecting R onto the attributes specified. Note that as usual, projection may result in fewer tuples in the resulting tables as duplicate tuples are removed.

As a final step, the FDs of R1 and R2 need to be recomputed. This step and the computing of superkeys of R is assumed to be (mostly) intuitive for a human, but if being performed by a machine, the key computation algorithm mentioned earlier and the FD inference rules can be used for these steps.

Note the optional step expanding Y; this tends to create larger/better relations, so it is a recommended step. For instance, suppose we used the FD we already noted violated BCNF in our example:
instructor $\rightarrow$ office.

If we follow the decomposition step above without the optional expansion step, we will create a table with just instructor and office, and another table with instructor and everything else. However, this still leaves the violating FD
instructor $\rightarrow$ email.

It makes more sense to keep all instructor info together, and that is in fact what will happen in this cases if we first computer instructor$^+$ = {instructor, office, email}. We will decompose one table with these three fields, and another table that now no longer has BCNF violating FDs with instructor on the left hand side. (We still have a violating FD, just not one involving instructor on the LHS.)

Example

Let's work through our example in more detail. We have the example table in figure 1, and a set of FDs:
instructor $\rightarrow$ office
instructor $\rightarrow$ email
{course_id, section} $\rightarrow$ instructor
course_id $\rightarrow$ title.

The only key for our table is {course_id, section}; any superset of attributes containing this set will be a superkey.

Applying the algorithm from the previous section, we choose a violating FD:
instructor $\rightarrow$ office.

Expanding the RHS as much as possible, we get
instructor $\rightarrow$ {office, email}.
(Technically the algorithm as I described would put instructor on the RHS as well; but since that is a trivial dependency, and a redundant column, I've left it out. Same end result.)

Now we decompose into two tables, shown in figures 2 and 3. Note how all the redundant information about instructors has been neatly eliminated by this step.

Now it is straightforward to show that the table in figure 2 is in BCNF. However, we still have a violating FD for the table in figure 3:
course_id $\rightarrow$ title.

Note that this FD corresponds pretty directly to the only real remaining redundancy in the table (aside from key values): the course titles. Again we continue through with the algorithm; the closure of course_id doesn't net us any additional columns, so we decompose into the two tables show in figures 4 and 5, which are now both in BCNF.

At this point, we are done, and our resulting schema has three tables in it: the ones in figures 2, 4, and 5. All inessential redundancy has been removed; we have a table of instructor information; a table with course information (course_id and title - but this could easily include things like number of hours, course description, etc.); and a table with actual schedule/section information for the current semester.

Note that the algorithm is not completely deterministic - there are points at which we get to make choices. The result is that there may be multiple correct normalizations of a complex

instructor	office	email
Painter-Wakefield, Christopher	BB 280I	cpainter@mines.edu
Mehta, Dinesh	BB 280J	dmehta@mines.edu
Hellman, Keith	BB 310F	khellman@mines.edu

Figure 2: One table resulting from the first decomposition of the example table.

instructor	course_id	section	title
Painter-Wakefield, Christopher	CSCI403	A	Database Management
Painter-Wakefield, Christopher	CSCI262	A	Data Structures
Painter-Wakefield, Christopher	CSCI262	B	Data Structures
Mehta, Dinesh	CSCI406	A	Algorithms
Mehta, Dinesh	CSCI561	A	Theory of Computation
Hellman, Keith	CSCI101	A	Intro to Computer Science
Hellman, Keith	CSCI101	B	Intro to Computer Science
Hellman, Keith	CSCI101	C	Intro to Computer Science
Hellman, Keith	CSCI274	A	Intro Linux OS
Hellman, Keith	CSCI274	B	Intro Linux OS

Figure 3: The other table resulting from the first decomposition of the example table.

course_id	section
CSCI403	Database Management
CSCI262	Data Structures
CSCI406	Algorithms
CSCI561	Theory of Computation
CSCI101	Intro to Computer Science
CSCI274	Intro Linux OS

Figure 4: First result of second decomposition.

instructor	course_id	section
Painter-Wakefield, Christopher	CSCI403	A
Painter-Wakefield, Christopher	CSCI262	A
Painter-Wakefield, Christopher	CSCI262	B
Mehta, Dinesh	CSCI406	A
Mehta, Dinesh	CSCI561	A
Hellman, Keith	CSCI101	A
Hellman, Keith	CSCI101	B
Hellman, Keith	CSCI101	C
Hellman, Keith	CSCI274	A
Hellman, Keith	CSCI274	B

Figure 5: Second result of the second decomposition.

database.

Correctness of Decomposition

There are two requirements which must be met in any decomposition of relations, which primarily add up to the statement that we must be able to recover the original relations exactly by natural joins.

The first requirement is simply that a natural join of the decomposed relations must include all of the attributes in the original relation. This is trivially satisfied in our decomposition algorithm by construction; when we decomposed, we partitioned the attributes so that a natural join gives us back exactly the attributes we started with.

The second requirement is called the *lossless join property*, and basically it says that when we natural join our decomposed tables together, we should regain exactly the tuples we started with, that is, all of the original tuples and no spurious tuples. I showed a proof of that this property is satisfied by our decomposition algorithm in class, which I omit here - it is not a difficult proof to reconstruct for yourself.

Multivalued Dependencies and Fourth Normal Form

Functional dependencies let us ferret out violations of BCNF and, as in the preceding example, this is often enough to attain a good database design. However, there are stronger normal forms than BCNF: 4NF, 5NF, and more besides. However, normal forms higher than 4NF involve rather exotic conditions, and we won't explore them in this class. We will take a brief look at 4NF, however. The tool that lets us understand 4NF is the multivalued dependency (MVD).

A multivalued dependency can be defined as follows:

A MVD $X \twoheadrightarrow Y$ exists on a relation R if whenever there are two tuples t_1 and t_2 which agree on the attributes in X, then there also exists tuple t_3 (possibly the same as t_1 or t_2) such that the following is true:

- $t_3[X] = t_1[X] = t_2[X]$
- $t_3[Y] = t_2[Y]$
- $t_3[Z] = t_1[Z]$,

where Z is everything that is not X or Y. By symmetry, there must also exist a tuple t_4 which also agrees on X with the other three tuples, but which reversed agreement on Y and Z with t_2 and t_1.

The notation $X \twoheadrightarrow Y$ can be read as "X multidetermines Y".

The situation can be summarized in the table below, where X, Y, and Z are our sets of attributes, and the lower-case x, y, and z's represent settings of these attributes.

	X	Y	Z
t_1	x	y_1	z_1
t_2	x	y_2	z_2
t_3	x	y_2	z_1
$(t_4$	x	y_1	$z_2)$

Note that not only does X multidetermine Y, X also necessarily multidetermines Z as well; it is common to write $X \twoheadrightarrow Y|Z$ to represent this.

This kind of situation arises primarily when there are two independent concepts bundled together in a table together. For example, consider a table which tracks an instructor's courses as well as their hobbies (what a strange relation!):

instructor	course_id	hobby
CPW	CSCI 403	reading sci-fi
CPW	CSCI 403	gardening
CPW	CSCI 262	reading sci-fi
CPW	CSCI 262	gardening

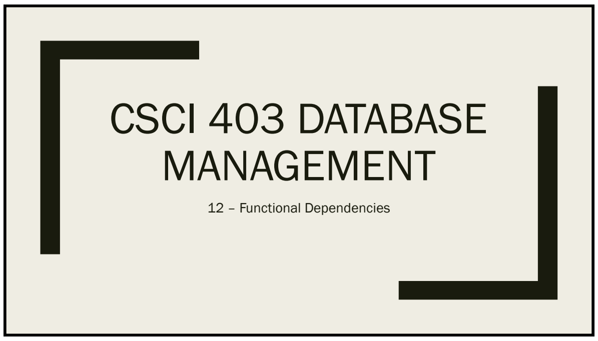
The hobbies and the course ids are completely independent, so you get a cross product situation happening where the table has to hold every pairing of these independent ideas. Note that the above table is in BCNF; indeed, there are no nontrivial FDs in the table at all!

Not surprisingly, the definition of fourth normal form uses MVDs in pretty much the same way as BCNF uses FDs:

Definition: A relation R is in 4NF with respect to some set of functional and multivalue dependencies if, for every nontrivial MVD $X \twoheadrightarrow Y$, X is a superkey in R.

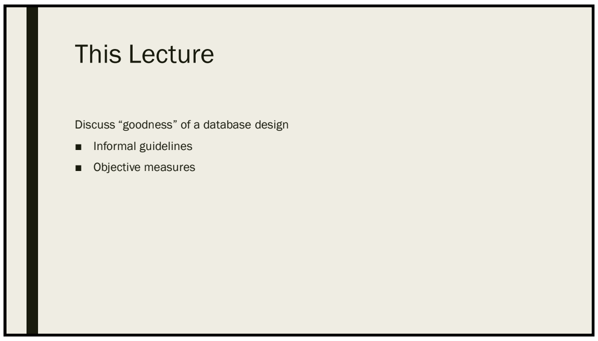
Also not surprisingly, the resolution of the problem tracks identically as well - find a violating MVD $X \twoheadrightarrow Y|Z$ and decompose into tables $R1 = \{X, Y\}$ and $R2 = \{X, Z\}$.

As a final note, any FD also qualifies as an MVD; therefore, if all violating MVDs are removed (putting us in 4NF), then all violating FDs are necessarily removed, and we are also in BCNF.



CSCI 403 DATABASE MANAGEMENT

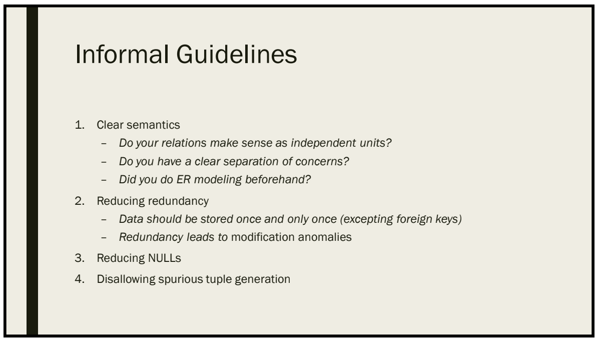
12 – Functional Dependencies



This Lecture

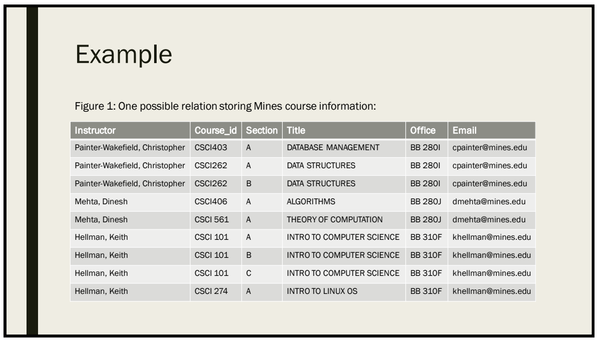
Discuss “goodness” of a database design

- Informal guidelines
- Objective measures



Informal Guidelines

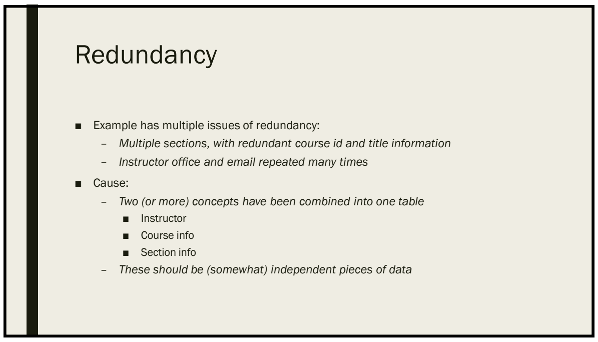
1. Clear semantics
   - *Do your relations make sense as independent units?*
   - *Do you have a clear separation of concerns?*
   - *Did you do ER modeling beforehand?*
2. Reducing redundancy
   - *Data should be stored once and only once (excepting foreign keys)*
   - *Redundancy leads to* modification anomalies
3. Reducing NULLs
4. Disallowing spurious tuple generation



Example

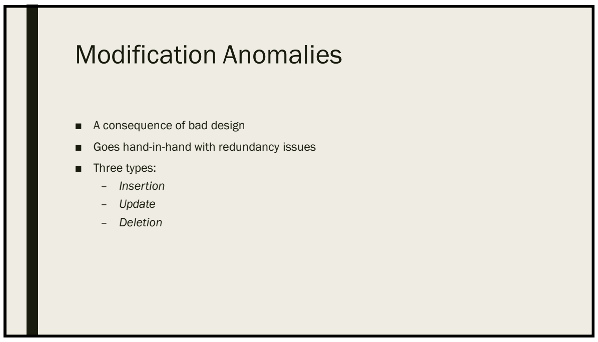
Figure 1: One possible relation storing Mines course information:

Instructor	Course_id	Section	Title	Office	Email
Painter-Wakefield, Christopher	CSCI403	A	DATABASE MANAGEMENT	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	A	DATA STRUCTURES	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	B	DATA STRUCTURES	BB 280I	cpainter@mines.edu
Mehta, Dinesh	CSCI406	A	ALGORITHMS	BB 280J	dmehta@mines.edu
Mehta, Dinesh	CSCI 561	A	THEORY OF COMPUTATION	BB 280J	dmehta@mines.edu
Hellman, Keith	CSCI 101	A	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 101	B	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 101	C	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 274	A	INTRO TO LINUX OS	BB 310F	khellman@mines.edu



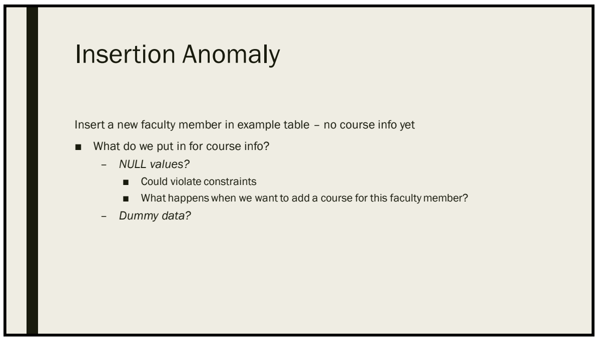
Redundancy

- Example has multiple issues of redundancy:
  - *Multiple sections, with redundant course id and title information*
  - *Instructor office and email repeated many times*
- Cause:
  - *Two (or more) concepts have been combined into one table*
    - Instructor
    - Course info
    - Section info
  - *These should be (somewhat) independent pieces of data*



Modification Anomalies

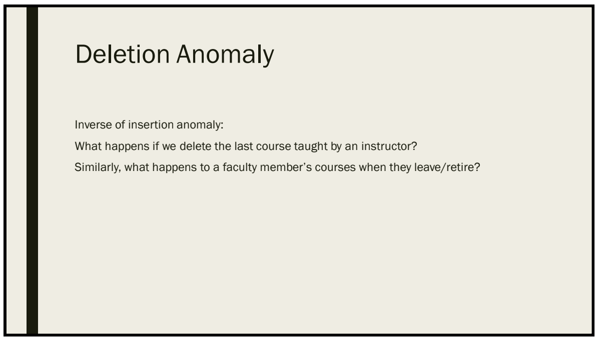
- A consequence of bad design
- Goes hand-in-hand with redundancy issues
- Three types:
  - *Insertion*
  - *Update*
  - *Deletion*



Insertion Anomaly

Insert a new faculty member in example table – no course info yet

- What do we put in for course info?
  - *NULL values?*
    - Could violate constraints
    - What happens when we want to add a course for this faculty member?
  - *Dummy data?*

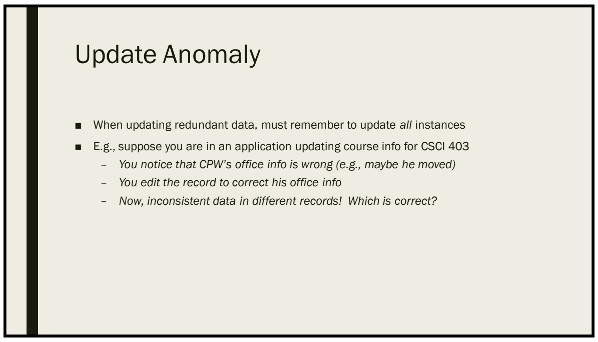


Deletion Anomaly

Inverse of insertion anomaly:

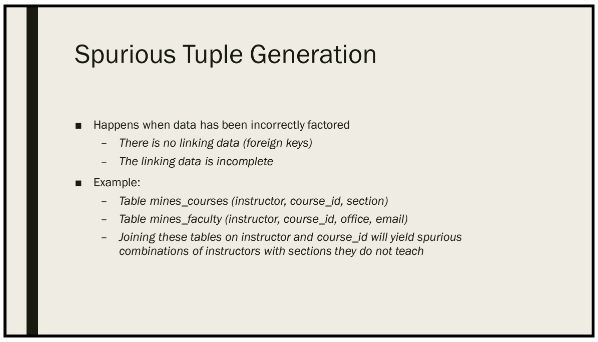
What happens if we delete the last course taught by an instructor?

Similarly, what happens to a faculty member's courses when they leave/retire?



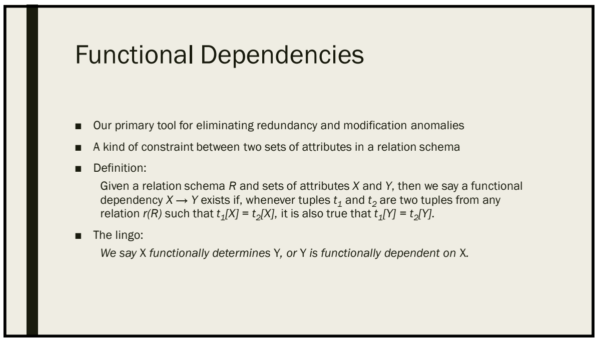
Update Anomaly

- When updating redundant data, must remember to update *all* instances
- E.g., suppose you are in an application updating course info for CSCI 403
  - *You notice that CPW's office info is wrong (e.g., maybe he moved)*
  - *You edit the record to correct his office info*
  - *Now, inconsistent data in different records! Which is correct?*



Spurious Tuple Generation

- Happens when data has been incorrectly factored
  - *There is no linking data (foreign keys)*
  - *The linking data is incomplete*
- Example:
  - *Table mines_courses (instructor, course_id, section)*
  - *Table mines_faculty (instructor, course_id, office, email)*
  - *Joining these tables on instructor and course_id will yield spurious combinations of instructors with sections they do not teach*

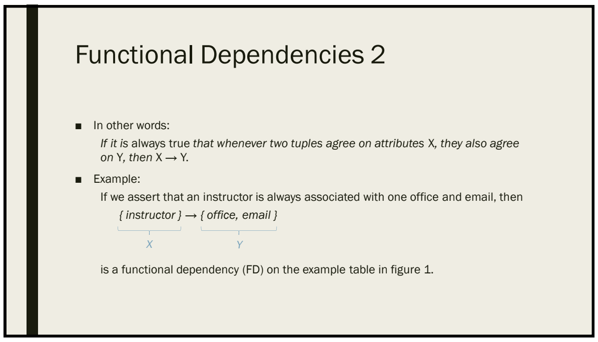


Functional Dependencies

- Our primary tool for eliminating redundancy and modification anomalies
- A kind of constraint between two sets of attributes in a relation schema
- Definition:

  Given a relation schema *R* and sets of attributes *X* and *Y*, then we say a functional dependency $X \longrightarrow Y$ exists if, whenever tuples t_1 and t_2 are two tuples from any relation *r(R)* such that $t_1[X] = t_2[X]$, it is also true that $t_1[Y] = t_2[Y]$.
- The lingo:

  We say X *functionally determines* Y, *or* Y *is functionally dependent on* X.



Functional Dependencies 2

- In other words:

  If it is always true *that whenever two tuples agree on attributes* X, *they also agree on* Y, *then* X → Y.
- Example:

  If we assert that an instructor is always associated with one office and email, then

  $$\underbrace{\{\ instructor\ \}}_{X} \longrightarrow \underbrace{\{\ office,\ email\ \}}_{Y}$$

  is a functional dependency (FD) on the example table in figure 1.

Functional Dependencies 3

Note:

FD's are *properties of the world that we impose on the data*, **not** properties of the data.

That is, finding FD's is a *design activity*.

The result is a constraint on the data that is allowed in our database.

Example:

It may be that we have a particular set of courses data in which each course_id is associated with one instructor. Then, *for that data*, it is true that whenever a tuple agrees on course_id, it also agrees on instructor. However, unless this is required to be true *for any set of data* we can put in our database, we cannot say { course_id } ⟶ { instructor}.

Types of Functional Dependency

- Trivial FD's
  - *Trivially,* $X \to X$
  - *More generally, if* $Y \subseteq X$*, then* $X \to Y$
- Non-trivial FD's
  - $X \to Y$
  - $Y \nsubseteq X$
- Completely non-trivial FDs
  - $X \to Y$
  - $X \cap Y = \emptyset$ (No overlap between X and Y)

Non-Trivial FDs

- We are primarily interested in non-trivial and completely non-trivial FD's.
- In our figure 1 example, we might identify the following completely non-trivial FD's:
  - *instructor ⟶ office*
  - *instructor ⟶ email*
  - *{ course_id, section } ⟶ instructor*
  - *course_id ⟶ title*
- Can you identify others?

Note the abuse of set notation here. I just find it more readable.

Functional Dependencies and Superkeys

- FD's can be viewed as a generalization of the notion of a *superkey*
- Recall a superkey is a set of attributes which will contain a unique subset of values for any tuple in a relation.
- Thus, if X is a superkey of R, $X \to R$.
- Alternately, if $X \to Y$ and $X \cap Y = R$, then X is a superkey of R.

Inference Rules

Allow us to infer additional FD's from an existing set of FD's

- Splitting rule:
  If $A \to \{B_1, B_2\}$ *then* $A \to B_1$ *and* $A \to B_2$
- Combining rule:
  If $A \to B$ *and* $A \to C$ *then* $A \to \{B, C\}$
- Transitive rule:
  If $A \to B$ *and* $B \to C$ *then* $A \to C$

More set notation abuse here. A, B, C, etc. are all sets. {B, C} is the union of sets B and C.

Additional rules can be derived and can be found in your textbook.

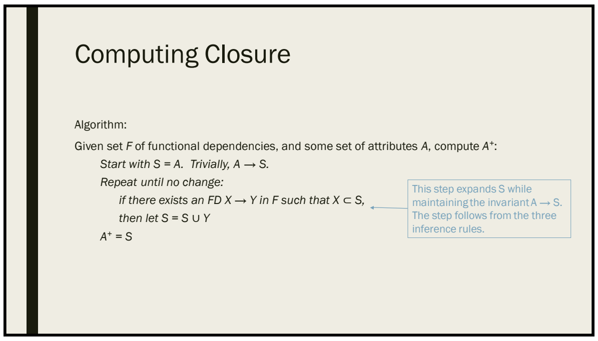
Closures

Definition:

Given some set of functional dependencies *F* on a relation schema *R*, and some subset of attributes *A*, then the set $\{B_i : A \to B_i\}$ is called the *closure* of *A* and is denoted A^+.

Closures are useful in:

- Normalization
- Finding all superkeys of a relation schema



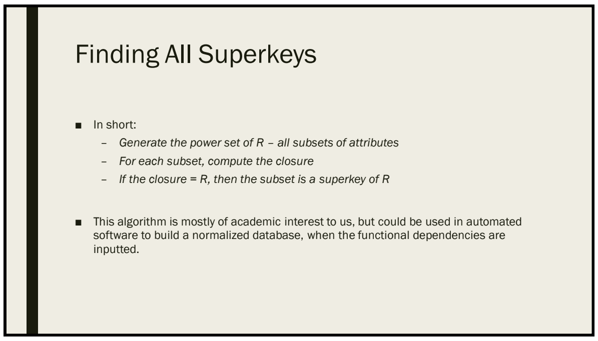
Computing Closure

Algorithm:

Given set F of functional dependencies, and some set of attributes A, compute A^+:

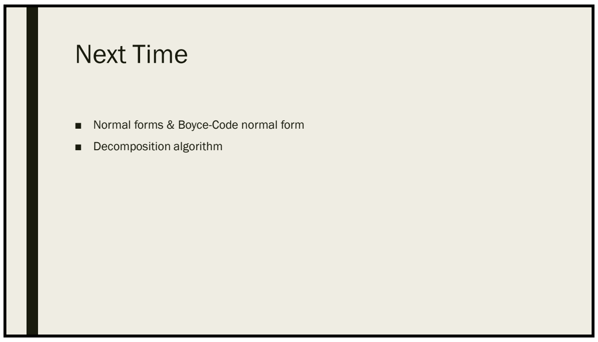
```
    Start with $S = A$.  Trivially, $A \rightarrow S$.
    Repeat until no change:
        if there exists an FD $X \rightarrow Y$ in $F$ such that $X \subset S$,
        then let $S = S \cup Y$
    $A^+ = S$
```

This step expands S while maintaining the invariant $A \rightarrow S$. The step follows from the three inference rules.



Finding All Superkeys

- In short:
  - *Generate the power set of R – all subsets of attributes*
  - *For each subset, compute the closure*
  - *If the closure = R, then the subset is a superkey of R*
- This algorithm is mostly of academic interest to us, but could be used in automated software to build a normalized database, when the functional dependencies are inputted.



Next Time

- Normal forms & Boyce-Code normal form
- Decomposition algorithm

CSCI 403 DATABASE MANAGEMENT

12.1 –Normalization

This Lecture

- Normalization
- Boyce-Codd Normal Form

Example Relation

Figure 1: One possible relation storing Mines course information:

Instructor	Course_id	Section	Title	Office	Email
Painter-Wakefield, Christopher	CSCI403	A	DATABASE MANAGEMENT	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	A	DATA STRUCTURES	BB 280I	cpainter@mines.edu
Painter-Wakefield, Christopher	CSCI262	B	DATA STRUCTURES	BB 280I	cpainter@mines.edu
Mehta, Dinesh	CSCI406	A	ALGORITHMS	BB 280J	dmehta@mines.edu
Mehta, Dinesh	CSCI 561	A	THEORY OF COMPUTATION	BB 280J	dmehta@mines.edu
Hellman, Keith	CSCI 101	A	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 101	B	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 101	C	INTRO TO COMPUTER SCIENCE	BB 310F	khellman@mines.edu
Hellman, Keith	CSCI 274	A	INTRO TO LINUX OS	BB 310F	khellman@mines.edu

Functional Dependencies Review

- Our primary tool for eliminating redundancy and modification anomalies
- A kind of constraint between two sets of attributes in a relation schema
- Definition:

  Given a relation schema *R* and sets of attributes *X* and *Y*, then we say a functional dependency $X \rightarrow Y$ exists if, whenever tuples t_1 and t_2 are two tuples from any relation *r(R)* such that $t_1[X] = t_2[X]$, it is also true that $t_1[Y] = t_2[Y]$.
- The lingo:

  We say X *functionally determines* Y, *or* Y *is functionally dependent on* X.

Functional Dependencies Review 2

- In other words:

  If it is always true *that whenever two tuples agree on attributes* X, *they also agree on* Y, *then* X → Y.
- Example:

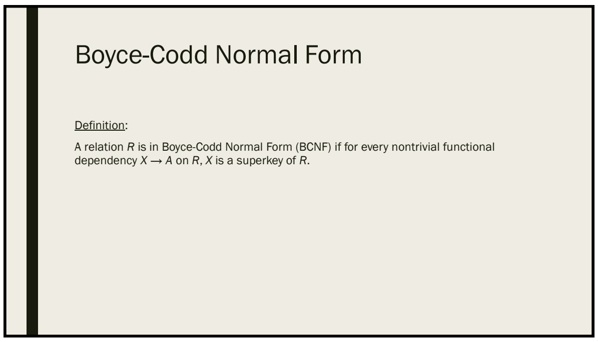
  If we assert that an instructor is always associated with one office and email, then

  $\underbrace{\{\ instructor\ \}}_{X} \rightarrow \underbrace{\{\ office,\ email\ \}}_{Y}$

  is a functional dependency (FD) on the example table in figure 1.

Normal Forms

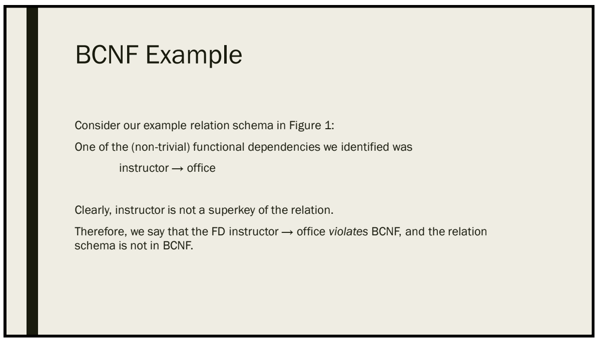
- Developed to define "good" design for a database
- Several forms: First normal form (1NF), Second (2NF), etc.
- Each normal form describe certain properties of a database
  - *E.g., 1NF eliminates multivalued and compound attributes*
  - *Mostly later normal forms subsume earlier normal forms*
- 1NF – 3NF are not terribly interesting stepping stones to the forms we care about:
  - *Boyce-Codd Normal Form (BCNF)*
  - *Fourth Normal Form (4NF)*
- There exist even stronger normal forms (5NF etc.), but BCNF and 4NF suffice for most purposes.



Boyce-Codd Normal Form

Definition:

A relation *R* is in Boyce-Codd Normal Form (BCNF) if for every nontrivial functional dependency $X \longrightarrow A$ on *R*, *X* is a superkey of *R*.



BCNF Example

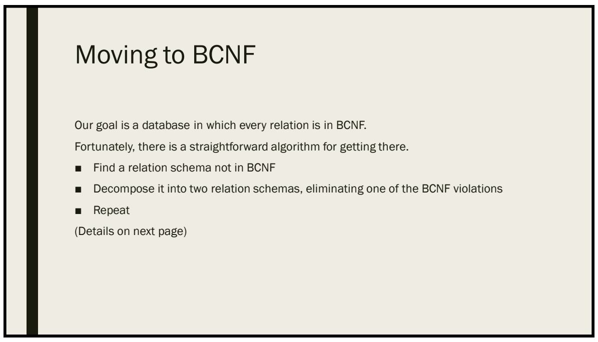
Consider our example relation schema in Figure 1:

One of the (non-trivial) functional dependencies we identified was

instructor ⟶ office

Clearly, instructor is not a superkey of the relation.

Therefore, we say that the FD instructor ⟶ office *violates* BCNF, and the relation schema is not in BCNF.



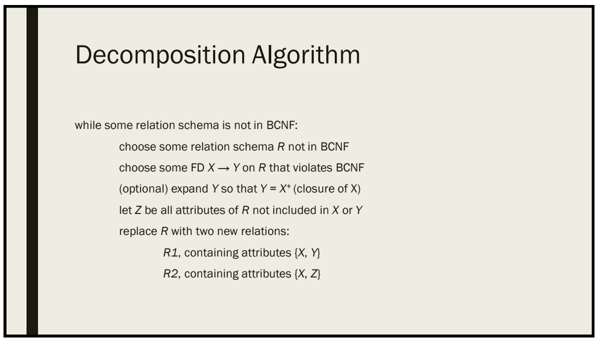
Moving to BCNF

Our goal is a database in which every relation is in BCNF.

Fortunately, there is a straightforward algorithm for getting there.

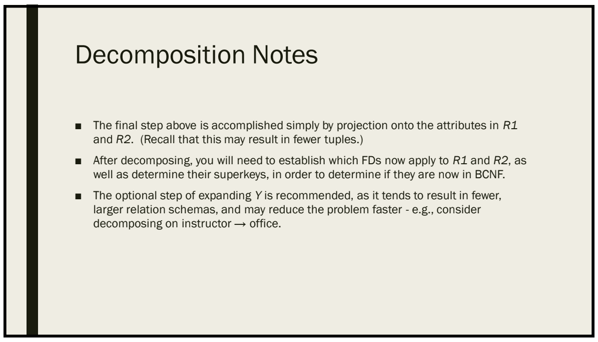
- Find a relation schema not in BCNF
- Decompose it into two relation schemas, eliminating one of the BCNF violations
- Repeat

(Details on next page)



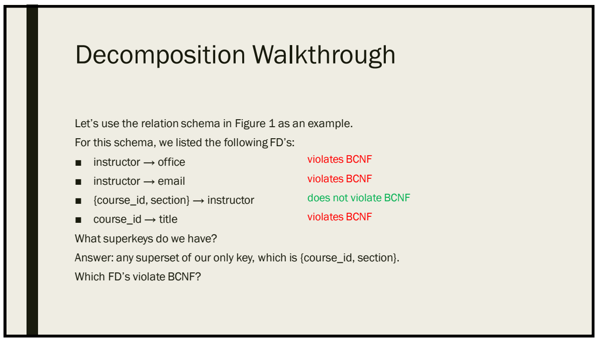
Decomposition Algorithm

```
while some relation schema is not in BCNF:
        choose some relation schema $R$ not in BCNF
        choose some FD $X \longrightarrow Y$ on $R$ that violates BCNF
        (optional) expand $Y$ so that $Y = X^+$ (closure of X)
        let $Z$ be all attributes of $R$ not included in $X$ or $Y$
        replace $R$ with two new relations:
                $R1$, containing attributes {$X$, $Y$}
                $R2$, containing attributes {$X$, $Z$}
```



Decomposition Notes

- The final step above is accomplished simply by projection onto the attributes in *R1* and *R2*. (Recall that this may result in fewer tuples.)
- After decomposing, you will need to establish which FDs now apply to *R1* and *R2*, as well as determine their superkeys, in order to determine if they are now in BCNF.
- The optional step of expanding *Y* is recommended, as it tends to result in fewer, larger relation schemas, and may reduce the problem faster - e.g., consider decomposing on instructor ⟶ office.



Decomposition Walkthrough

Let's use the relation schema in Figure 1 as an example.

For this schema, we listed the following FD's:

■ instructor ⟶ office	violates BCNF
■ instructor ⟶ email	violates BCNF
■ {course_id, section} ⟶ instructor	does not violate BCNF
■ course_id ⟶ title	violates BCNF

What superkeys do we have?

Answer: any superset of our only key, which is {course_id, section}.

Which FD's violate BCNF?

Walkthrough 2

- Let's pick our first violating FD to work with first: instructor ⟶ office
- Next, expand the RHS as much as possible (we want the closure of instructor):
  instructor ⟶ {instructor, office, email}
- Now we decompose into two new tables, shown on the next slide:
  - $R1 = \pi_{instructor,office,email}(R)$
  - $R2 = \pi_{instructor,course\_id,section,title}(R)$
- We can now discard the table from figure 1

Tables After One Step

R1:

Instructor	Office	Email
Painter-Wakefield, Christopher	BB 280I	cpainter@mines.edu
Mehta, Dinesh	BB 280J	dmehta@mines.edu
Hellman, Keith	BB 310F	khellman@mines.edu

R2:

Instructor	Course_id	Section	Title
Painter-Wakefield, Christopher	CSCI403	A	DATABASE MANAGEMENT
Painter-Wakefield, Christopher	CSCI262	A	DATA STRUCTURES
Painter-Wakefield, Christopher	CSCI262	B	DATA STRUCTURES
Mehta, Dinesh	CSCI406	A	ALGORITHMS
Mehta, Dinesh	CSCI 561	A	THEORY OF COMPUTATION
Hellman, Keith	CSCI 101	A	INTRO TO COMPUTER SCIENCE
Hellman, Keith	CSCI 101	B	INTRO TO COMPUTER SCIENCE
Hellman, Keith	CSCI 101	C	INTRO TO COMPUTER SCIENCE
Hellman, Keith	CSCI 274	A	INTRO TO LINUX OS

Walkthrough 3

- Table *R1* is now in BCNF (yay!)
  - *Note this was not guaranteed by the algorithm – we could have had other violating FDs*
- Table *R2* has a violating FD, though: course_id ⟶ title

Walkthrough 4

Decomposition of *R2* via course_id ⟶ title:

$course\_id^{+}$ = {course_id, title}

Decompose into *R3* and *R4*:

- $R3 = \pi_{course\_id,title}(R2)$
- $R4 = \pi_{instructor,course\_id,section}(R2)$

R3:

Course_id	Title
CSCI403	DATABASE MANAGEMENT
CSCI262	DATA STRUCTURES
CSCI406	ALGORITHMS
CSCI 561	THEORY OF COMPUTATION
CSCI 101	INTRO TO COMPUTER SCIENCE
CSCI 274	INTRO TO LINUX OS

R4:

Instructor	Course_id	Section
Painter-Wakefield, Christopher	CSCI403	A
Painter-Wakefield, Christopher	CSCI262	A
Painter-Wakefield, Christopher	CSCI262	B
Mehta, Dinesh	CSCI406	A
Mehta, Dinesh	CSCI 561	A
Hellman, Keith	CSCI 101	A
Hellman, Keith	CSCI 101	B
Hellman, Keith	CSCI 101	C
Hellman, Keith	CSCI 274	A

Walkthrough Wrap-up

- Done!
  - *Three tables remain: R1, R3, R4*
  - *All non-essential redundancy has been removed*
  - *Each table now represents a fundamental entity:*
    - R1 = instructor info
    - R3 = course info
    - R4 = section info
- As a final note: this algorithm is not deterministic – you can different decompositions following different choices of FD to work with.

Next Time

- Correctness of decomposition algorithm
- Multi-valued dependencies and 4NF