

# The Relational Model



# A Bit o' Review

- A database stores data and metadata
- A DBMS controls access to the data
  - Eliminated the limitations of a file system
- Hierarchical and Network models require knowledge of the physical structure to access data (in other words the DBMS isn't very good!)
- Relational Model allowed one to ignore the physical aspect and only look at the logical data structure

# Tables



They hold stuff!

...

A table contains a group of related entity occurrences (re: entity set from ERM)

# What, you want more than that?

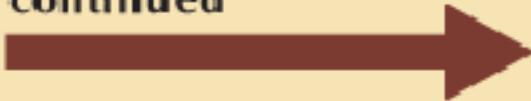
- Characteristics of a (relational) table
  - 2-D structure of rows and columns, a.k.a. Matrix
  - A row represents a single entity within the entity set
  - A column represents an attribute and each has a distinct name
  - Each row-column intersection represents a data value
  - Each column has a data type which constrains its data
  - Each column has a specific range of values called the **attribute domain** (think of constraints here)
  - The order of rows and columns is unimportant
  - Each table must have an attribute or combination of attributes that uniquely identifies each row

# Case Study

Table name: STUDENT

STU_NUM	STU_LNAME	STU_FNAME	STU_INIT	STU_DOB	STU_HRS	STU_CLASS
321452	Bowser	William	C	12-Feb-1975	42	So
324257	Smithson	Anne	K	15-Nov-1981	81	Jr
324258	Brewer	Juliette		23-Aug-1969	36	So
324269	Oblonski	Walter	H	16-Sep-1976	66	Jr
324273	Smith	John	D	30-Dec-1958	102	Sr
324274	Katinga	Raphael	P	21-Oct-1979	114	Sr
324291	Robertson	Gerald	T	08-Apr-1973	120	Sr
324299	Smith	John	B	30-Nov-1986	15	Fr

STUDENT table,  
continued



STU_GPA	STU_TRANSFER	DEPT_CODE	STU_PHONE	PROF_NUM
2.84	No	BIOL	2134	205
3.27	Yes	CIS	2256	222
2.26	Yes	ACCT	2256	228
3.09	No	CIS	2114	222
2.11	Yes	ENGL	2231	199
3.15	No	ACCT	2267	228
3.87	No	EDU	2267	311
2.92	No	ACCT	2315	230

**STU\_HRS** = Credit hours earned  
**STU\_CLASS** = Student classification  
**STU\_DOB** = Student date of birth

**STU\_GPA** = Grade point average  
**STU\_PHONE** = 4-digit campus phone extension  
**PROF\_NUM** = Number of the professor  
who is the student's advisor

# Typing

- Each column has a data type which constrains its data
  - Very similar to how variables are typed in programming
  - Numeric → int, smallint, bigint, float, double, etc.
  - Character(s) → char, varchar, text
  - Dates
  - Boolean → either an actualy boolean T/F or could be a bit type of 0 or 1

# Keys

- Each table must have an attribute or combination of attributes that uniquely identifies each row
- This is known as a ***primary key***
  - Note: a PK must be unique and must not be null
- Remember one use of these?
- There are many types of keys though, and they can be used for other things such as establishing relationships or ensuring integrity

# Keys

- **Key** → consists of one or more attributes that determine other attributes
  - **Determination** → If A determines B, then by knowing A you can find out B (for a particular entity instance)
  - **Functionally Dependent** → B is functionally dependent on A if A determines B -OR- if each value in column A determines *one and only one* value in column B
  - There is a second definition for this that allows for multiple occurrences of A as long as they all agree on B



# Misc. Info

- Be careful, dependence is not bi-directional
  - In other words "A depends on B" does not necessarily mean "B depends on A"
- Sometimes more than one attribute is needed to uniquely define an instance
  - This is a **composite key**
- The attribute or attributes that define a key are called **key attributes**

# Specialized Keys

- If attribute B is *functionally dependent* on a composite key A but not on any subset of that key, the attribute B is **fully functionally dependent** on A
- **Super key** uniquely defines a row
- **Candidate key** is a super key without unnecessary attributes
- Typically we aren't concerned with these, we just use the term **primary key**, which could be either

# I Don't Know!

- What if you don't have data for a certain attribute? This is a **null** value
- When a primary key must be unique to do their jobs and enforce **entity integrity**
  - Thus primary keys are not permitted to contain nulls. Why?
- Nulls in general should be avoided if at all possible

# But...!

- Sometimes a null is unavoidable!
  - Well yes, but they should be avoided at all costs
- "null" can often mean more than one thing so they can become confusing
  - Some functions have trouble with them
- **Flags** can be used as null stand-ins
  - In other words have a value that will never be in the particular data set represent a null
  - ex. a SS# will never be negative

# Foreign and Secondary Keys

- A **foreign key** is an column of a table whose data is the *primary* key of another table
- **Referential integrity** → all foreign keys must point to valid entity instances
- A **secondary key** is a key that identifies a row in a meaningful way and only used to speed data retrieval
  - Effectiveness is related to how well it narrows down the number of matches

# Key Summary

**TABLE 3.3** Relational Database Keys

KEY TYPE	DEFINITION
<b>Superkey</b>	An attribute (or combination of attributes) that uniquely identifies each row in a table.
<b>Candidate key</b>	A minimal (irreducible) superkey. A superkey that does not contain a subset of attributes that is itself a superkey.
<b>Primary key</b>	A candidate key selected to uniquely identify all other attribute values in any given row. Cannot contain null entries.
<b>Secondary key</b>	An attribute (or combination of attributes) used strictly for data retrieval purposes.
<b>Foreign key</b>	An attribute (or combination of attributes) in one table whose values must either match the primary key in another table or be null.

# Integrity Summary

**TABLE 3.4** Integrity Rules

ENTITY INTEGRITY	DESCRIPTION
Requirement	All primary key entries are unique, and no part of a primary key may be null.
Purpose	Each row will have a unique identity, and foreign key values can properly reference primary key values.
Example	No invoice can have a duplicate number, nor can it be null. In short, all invoices are uniquely identified by their invoice number.
REFERENTIAL INTEGRITY	DESCRIPTION
Requirement	A foreign key may have either a null entry, as long as it is not a part of its table's primary key, or an entry that matches the primary key value in a table to which it is related. (Every non-null foreign key value <i>must</i> reference an <i>existing</i> primary key value.)
Purpose	It is possible for an attribute NOT to have a corresponding value, but it will be impossible to have an invalid entry. The enforcement of the referential integrity rule makes it impossible to delete a row in one table whose primary key has mandatory matching foreign key values in another table.
Example	A customer might not yet have an assigned sales representative (number), but it will be impossible to have an invalid sales representative (number).

# Controlled Redundancy

- We have established that redundancy is bad (for many reasons)
  - Storage cost
  - Management, such as value changes
- (primary) keys helped to eliminate redundancy
  - However, when a table's PK is used as an attribute in another table (FK) it could be listed more than once
- This is *controlled* redundancy and is acceptable
- Example: A Parts table might have a Vendor attribute that holds a Vendor's ID, and obviously a vendor can make more than one part

# Relational Schema

- A textual representation of a table
  - ex. Student( stud\_id, fname, lname, etc. )
- These are useful on some level but cannot relay some important information like foreign key relationships

# Relational Set Operations

- Ways of relating or combining data in different tables
  - They are the basis for SQL's JOINS
- UNION → the combination of all rows from two tables excluding duplicates
  - Must be **union-compatible**, or the same # of columns, same column names, and same domains
- INTERSECT → returns only rows that appear in both tables

# Relational Set Operations

- DIFFERENCE → Yields the rows from the *first* table that are not in the *second*
  - Note the order is important
  - Must be union-compatible
- SELECT → Return values that satisfy some set of given conditions (or lack thereof)

# Lesser Used Operations

- PRODUCT → Yields all possible combinations of rows (see Fig. 3.8)
- PROJECT → Yields all values for selected attributes
  - This is more commonly done with a SELECT
- DIVIDE → Uhhh?

# Joins

- This is where you learn to use the Force
- A way of combining information from two or more tables
  - Used in conjunction with `SELECT`
- These are very powerful because they relate tables based on their common attributes
- The idea is to somehow determine rows that equal values for some number of attributes and then return the combined attributes from each table



# Natural Join

- Let's look at a particular type of join, a *natural* join as an example
  - It should come naturally right?
- A natural join is very "automagic" → doing a NJ of two tables causes the SQL engine in the DBMS to compare all similarly named columns from each table and only return rows where all values are equal between the two
- See Fig 3.12-3.14 on pgs 76-77
- Natural joins are not often used because of the "automagic" nature of the comparison

# Inner Join

- The Natural Join is a specific type of an *inner join*
  - A combination of table attributes to form result rows based on some sort of join predicate (re: comparison)
  - The most common type of join and also the default
- Equi join → operates the same way as a natural join except you specify which columns you want to perform the join on
  - Both always use equality
- Theta join → Similar, but uses a non-equality comparison like <

# I told you!

- Remember "avoid NULL at all costs?"
- NULL values can never match anything, even itself, so columns that allow a NULL are hard to join on using the above join types
- The only exception is you can, instead of equality, say "attribute\_name IS NOT NULL" or IS NULL

# Data Dictionary (again)

- Fig 3.6 on pg 79 shows a data dictionary
- A detailed description of all tables found within the user/designer-created database
- Fancy term for a design document
- Meant to be an human read document to describe the logical arrangement of the database
- **System catalog** → a DD with extra info like permissions, data types, and timestamps
- Typically the terms are used interchangeably because software usually works with catalogs

# Relationships in Tables

- We know the types of relationships, but how do we represent them in tables?
  - 1:M → DB bread-and-butter
  - 1:1 → rare but not so bad
  - M:N → can only be done through a hack

# 1:M

- These are the variety where one table's rows get associated with many rows in another table
  - The Salespeople in our car example form a 1:M relationship with the Sales table entries
  - At a university there is a 1:M link between courses and classes (Fig. 3.21). Also, the paragraph after the figure is a decent review of some vocab.

# 1:1

- These relationships are rare but do occur
  - Be careful, 1:1 relationships can indicate the entities are not well defined or that both pieces of data should actually be in a single table
- Fig 3.23
- Architected the same as 1:M with the (possibly unenforceable) constraint that the M side is restricted to one entry
- What do you think of using EMP\_NUM as the name in the Department table?

# M:N

- Impossible!

# M:N

- Okay fine, it can be done but it's a (not necessarily bad) hack called a ***composite entry***
- First look at Fig 3.25
  - What are some problems with it?
  - Redundancy!
  - Are there alternatives? You could have multiple CLASS\_CODE columns but that's a bad idea.

# Composite Entry

- Basically this process means making a third table that relates *at least* the primary keys of each table
  - Sometimes this table is called a *linking table*
- Redundancies are still created but these are considered *controlled redundancy* and are thus acceptable
- Figs. 3.26 and 3.28

# Data Redundancy

- When thinking of redundancy it may be helpful to worry about whether removing something takes information out of a database rather than looking at whether there is duplicate info
- Let's look at Fig. 3.30

# Indexes

- Think of a book index
- A way of ordering things for quick lookup
- Painter example from book is correct but may not be very illustrative of the point
- Think of a Student table instead
- **Index key** → the attributes that an index is based on
- **Unique Index** → an index that has only one row associated with it
  - Primary keys are common index keys for these

# Indexes

- Typically used for fast lookup
- Books says it can be used for ordering but this isn't as great or accurate as they claim
- **Covering Index** → a special index that contains all the attributes needed for a query that will take advantage of it of the index
  - Can be very costly!
- When trying to find good indexes, think about what the attributes you're typically going to search on/for

# Indexes

- Speed up read queries *a lot*
- Slow down writes
- Take up extra space, especially covering indexes
- Cannot help with some types of wildcard queries
- A table can have many indexes, make use of this fact!

# The 12 Rules

Table 3.8 pg 92

# Homework!

- Review Questions: 1, 2, 8, 12, 13, 16
- Problems: 1, 6, 7, 8, 9, 10, 11, 14