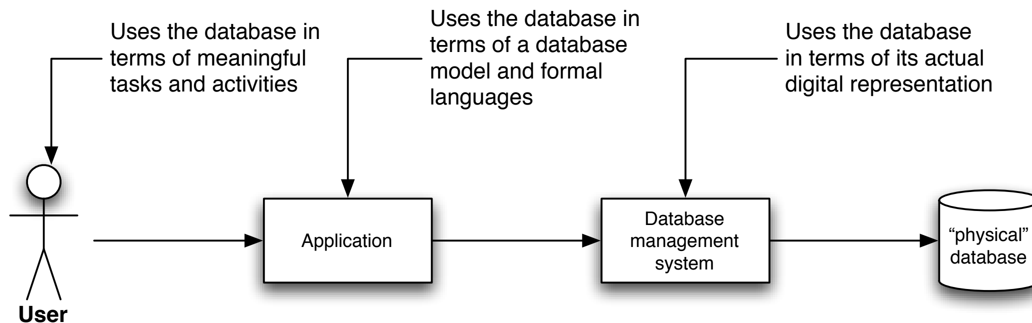
Database Systems

- A database, in the broadest sense, is any kind of information repository
  - The address book is a database
  - Your organizer is a database
  - LMU keeps a database of its students
  - Hospitals keep databases of their patients, procedures, etc.
  - Your browser bookmark list is a database
  - Your brain contains a really sophisticated database
- Note that databases need not be in electronic or digital form; typically, however, the word "database" refers to information stored in this manner

- Before a database can be used, it must first be defined
  - A database definition describes a database's structure and content
  - A formal database definition is called a *schema*
- Once defined, there are four basic database operations, known as "CRUD" for short:
  - *Create* (insert): Add new information to the database
  - *Retrieve*: Look up information from the database
  - *Update*: Modify existing information in the database
  - *Delete*: Remove existing information from the database
- Not surprisingly, getting a machine to store and manipulate a database requires a language — while not strictly a *programming* language, it's still handled in a similar manner (formal rules, translation, etc.):
  - A *data definition language* (DDL) defines schemas
  - A *data manipulation language* (DML) performs database operations

Database Application Layers

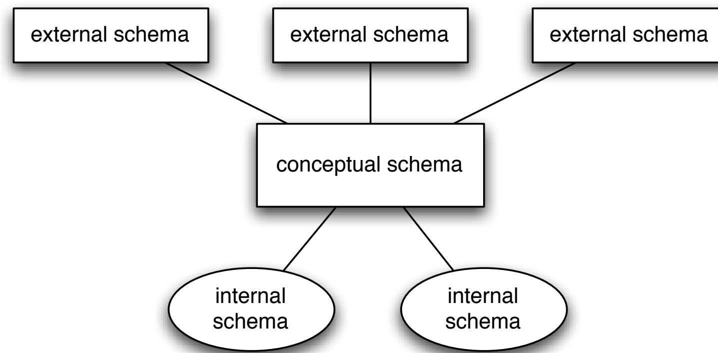
- One typically does not use a database system directly or in isolation; instead, database users see an *application* that performs meaningful operations for them, with the database working in the background
  - eBay uses a database for its products, sellers, buyers, and auctions, but the Web site itself is an electronic auction application
  - MySpace and FaceBook are social networking applications, with a database underneath for tracking its users, their Web pages, and their social networks
  - World of WarCraft maintains a database of its players and their characters, but presents (and modifies) this information through the mechanics of a role-playing game
- This separation into *layers* allows developers to use the same database system in different ways:



  - Users never have to see the database directly — they just do the things that are meaningful to them
  - An application doesn't care about how a database is truly represented; as long as the formal model and languages are followed, then changes can be made without strictly having to affect the application program — this is *data independence*
  - The database management system can change the actual digital representation of the database without affecting the application or user — this allows for "transparent" changes such as a *distributed database*, where information is stored in multiple places

ANSI/SPARC Specification

- This layered approach has been standardized by the American National Standards Institute's Standards Planning and Requirements Committee (ANSI/SPARC):

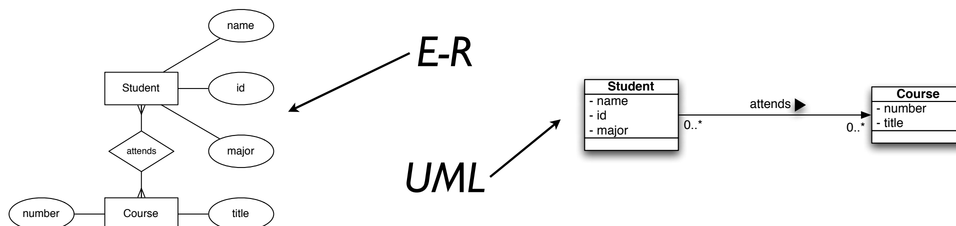


- The ANSI/SPARC specification separates a database application into three distinct schemas or *views*:
    - An *external schema* or *view* represents what a specific type of user can see in a database application — note how the same database may have multiple external schemas, depending on what application is using it
    - The *conceptual schema* or *view* represents an overall "big picture" view of the database — it shows *everything* in the database in a manner that is *independent* of a specific database management approach, technology, or product
    - An *internal schema* or *view* is a "rendering" of the conceptual schema in terms of a specific database approach, technology, or product

ANSI/SPARC Examples

- The notion of multiple external views for the same underlying information is actually more common than it might initially seem to be:
  - Online stores such as Amazon or iTunes provide different views for customers (products for sale, items on order, recent purchases), support staff (product inventory levels, shipping information), and business managers (sales trends, upcoming product lines)
  - For airline systems, the same database may "present" itself as a reservation system for passengers or a personnel scheduling system for managers — both applications need the same flight schedule data, but for different purposes
  - Multiplayer role-playing games provide the game itself for its member-players, but display more information (and apply different rules) for the "game god" staffers who make sure that the game world is running smoothly
- Each of these systems has one conceptual schema

- Conceptual schemas are usually expressed in diagram form (what some folks like to call a *visual language*):
  - *Entity-relationship* (E-R) diagrams have been the traditional notation for conceptual schemas
  - *Unified Modeling Language* (UML) is a newer notation, and is growing in use today
- E-R was specifically designed for representing conceptual schemas, while UML covers conceptual schemas as well as other types of technical diagrams



The Relational Database Model

- An *internal schema* or *view* of a database itself consists of two parts: a *logical model* that describes the structure of the database, and a *physical model* that determines how this structure is actually stored in digital form (whether on one or more computers)
- All database management system products (e.g., Oracle, Sybase, Access, MySQL, PostgreSQL) have their own physical models — we won't get into that here
- Most systems in use today, though, do share a common logical model: the *relational database model*

- The relational database model is, by far, the most popular internal logical model that is in use today
- The relational model's popularity can be attributed to a "1-2-3 punch" of key characteristics:
  - It is conceptually simple — its fundamentals are intuitive and easy to pick up
  - It is backed (powered!) by mathematics, resulting in extremely effective algorithms and techniques
  - Its most popular database manipulation language, *SQL*, is fairly easy to learn and use
- On a historical note, the model was first published by Ted Codd in 1970, based on work by mathematicians Georg Cantor and D. L. Childs

Relational Model Fundamentals

- A *relational database* is a collection of *tables* or *relations*
- Each table is defined by a *relation schema* consisting of its *name* and its *attributes*; each attribute has a *domain*, which is the set of acceptable values for that attribute
  - For example, the *Student* entity from the previous conceptual schema would translate into the relational model as a *Student* table with three attributes: *name*, *id*, and *major*
  - The *name* attribute's domain would be all possible text strings, *id*'s domain would be the set of strings consisting of nine numeric digits (e.g., "103 110 120"), and *major*'s domain would be the set of majors offered by the university
- A specific "population" of tables with particular data is viewed as an *instance* of the relational database

- A relation or table instance is a set of *tuples* which in turn are sets of *values* for each attribute of the table
  - A particular tuple in an instance of the *Student* table, for example, may be { *name* = "Johnny Mnemonic", *id* = "325 312 728", *major* = "computer science" }
- Some attributes in the table are *keys*: sets of attributes that *uniquely determine* a specific tuple in the table
  - For example, { *id* } is a key of the *Student* table, since only "Johnny Mnemonic" can have an *id* attribute of "325 312 728"; put another way, if you know the value of the *id* attribute, you know exactly which tuple in the table you're talking about
- One last thing — there's more than one term for each of these concepts (you've already seen *relation* and *table*); here's a quick crib sheet:
  - *Tuples* = *records*, *rows*
  - *Attributes* = *fields*, *columns*, *properties*
  - *Domains* = *types*
  - *Keys* = *identifiers*

(The specific "version" used is typically a matter of context, ranging from mathematical to business, IT, or technical perspectives)

The World [of Relations] is Flat

- An important constraint in the relational model is that a table's attributes cannot themselves be sets — they must be *flat* or *normalized*
- For example, a *Course* table, which we may visualize as "containing" multiple students (representing the students who are taking a course), can*not* have a *students* attribute consisting of a set of student tuples
- Instead, we keep *Student* and *Course* as separate tables, and use another table to "link" which students are in which courses: *Attends(student_id, course_number)*

- Note the convenient name *Attends* for this table, as it effectively captures the relationship expressed in the conceptual schema; a "joint" name for these tables is also typical, such as *Student_Course*
- The secret to this table is its use of the *key* attributes in *Student* and *Course*, respectively — by using the keys, we can be assured that a single tuple in *Attends* refers to a pairing of a single student to a single course
  - When one table uses another table's key in this way, that key is called a *foreign key*
- Another situation to avoid is duplicate attribute values — for example, if *Course* had *instructor_name* and *instructor_status* (full-time/part-time) attributes, those would be duplicated in multiple *Course* tuples; a separate *Instructor* table should be created instead

Relational Algebra and Calculus

- At this point, you may be asking, "Show me the math!"
  - Be careful what you ask for **:)**
- The relational model actually has two mathematical perspectives: *relational algebra* and *relational calculus*
- In the relational algebra, tables/relations serve as *operands* to a set of relational *operators*; as with any algebra, it is focused on *what to do* with relations
- In contrast, the relational calculus focuses on *what is desired* from a relational database

- The relational algebra has six fundamental operations — all others can be derived from these six: *project*, *select*, *union*, *difference*, *Cartesian product*, and *rename*
  - These operations are *closed* over the set of all relations: they take relations as operands, and produce new relations as results
  - Properties and concepts that are important to any algebra also apply to the relational algebra: *associativity*, *commutativity*, *distributivity*, *identity*, and *inverse operations* — of course, these all apply to *tables* and not numbers, as you might have seen elsewhere
- There actually two relational "calculi:" *tuple* and *domain*
  - In both cases, instead of manipulating relations, the calculus *describes* new relations by stating what they contain and what rules (constraints) they follow or satisfy
- The mathematical foundation (whether algebra or calculus) enables formal reasoning (proofs) on relations, which in turn leads to excellent algorithms
- Anything you can build with the algebra can be "declared" with the calculus, and vice versa

Back to Those Multiple Tables…

- We'll conclude this "bird's eye tour" of databases with a look at a non-fundamental, yet extremely important, relational algebra operation: the relational *join*
- Join is "non-fundamental" because it can actually be derived through a combination of projection, union, selection, and Cartesian product — but it is so important that most database systems provide direct support for this operation
- The idea behind join is actually quite intuitive: let's revisit our *Student/Attends/Course* database example

- Suppose we would like to retrieve the names of the students who are attending a particular course:
  - We know who's in which course via the *Attends* table, which can give us the student IDs that are in a course
  - But we want the *names* of these students — so we need to take those IDs, look up the *Student* table for the tuples with those IDs (only one apiece, since *id* is a key of *Student*), and then we'll have their names
- This sequence of "look for the tuples in one field with values that match the tuples in another field" is the essence of a relational *join* operation
- As mentioned, relational join is a derived operation; to join relations *r* and *s* over attributes $A_1 \ldots A_n$, we do:
  - Take the *Cartesian product* of *r* and *s*, then perform a *select* with the condition that the corresponding attribute values of A_1 to A_n from *r* and *s* are equal
  - Perform a *project* on the result, extracting only the union of the attributes of *r* and *s* — note how union eliminates duplicates, leaving only one "copy" of attributes A_1 to A_n

Go Out and Play

- Some industrial-strength database systems are available as free downloads, so with enough knowledge, you can store and manage volumes of data without spending volumes of cash
- The database shown in class is called PostgreSQL, available at *http://www.postgresql.org*
- MySQL is also very popular, particularly with Web applications; you can get it from *http://www.mysql.com*
- Both systems are installed as "servers" and are built for multi-user, network access from the ground up

- Do recall that a database management system forms only one layer of a full-fledged application, so don't expect to have the next *eBay* or *FaceBook* written up over the weekend!
- If anything, it is hoped that this overview demystifies database management systems somewhat, so that they don't seem to arcane or mysterious
- One major take-home point: there *have* been other candidates for internal logical data models (the network model; object-oriented database models) — but so far, only the relational model has the power of formal, proof-driven reasoning behind it, and this is a major factor in its effectiveness and the quality of the software that is based on this model