­­CS 348 – Intro to Database Management

# Intro

## Data

* representation of facts, concepts or instructions organized in a formal manner for communication or interpretation
* can be volatile (disappears when computer turned off) or persistent (saved)
* data model: collection of high-level data description constructs that hide low level storage details

## Database

* large persistent collection of relatively similar info organized to facilitate efficient retrieval and modification
* Used when there is lots of data, you need persistence, reliability, concurrent access, and efficiency. When you need data to be managed independently across programs

## Database Management System (dbms)

* program that manages details relating to the storage and access to a database
* Abstracts common functions and creates an interface for interacting with the database
* Has a data model, access control, concurrency control, database recovery, and allows for database maintenance
* Data Independence – applications don't directly access data, they are immune to changes in data storage structure and organization
* Uses Data Definition Languages (DDL) for specifying schema, and Data Manipulation Languages (DML) for specifying queries and updates
* Advantages:
  + Data Independence (details of data representation and storage hidden)
  + Efficient Data Access
  + Data Integrity and Security (Integrity constraints and access controls can be implemented)
  + Central Data Administration, by experienced professionals
  + Concurrent Access, Crash Recovery, simplification of application development
    - Writes “checkpoints” to disk to recover after a crash, + logs of all transactions
* Disadvantages
  + Real time code, when there are only a few specialized queries that will be run with custom code
  + May need to manipulate data in ways not possible with a DBMS

## Transactions

* an atomic unit of work on a database, has the following properties
  + Atomic – transaction occurs entirely or not at all (if something stops a transaction half way, everything it has done so far is undone – via log files written before the action is performed)
  + Consistency – preserves database consistency
  + Isolated – other transactions can't interfere
  + Durable – changes made are permanent once completed
* Allows concurrent access and ensures transactions don’t interact (using locks)

## Schema

* 1. Defines the structure of the data and how it is organized

## Instance

* 1. An implementation of the schema, data conforms to the schema (data)

## 3 level schema

* External Schemas – what the programs/user sees, may differ for each user
  + “fake” schemas for abstraction to users, tables don’t actually exist
* Conceptual Schema – description of logical structure of all data in database
  + Tables and their columns
* Physical Schema – description of physical aspects, files, devices, algorithms, etc. How the data will actually be stored

# history

* + 2000 BC – First recorded records
  + 296 BC – Library of Alexandria – attempt to gather all human knowledge
  + 1884 – US Census uses punch cards for counting
  + 50s-60s – Batch processing, sequential files/tape, input on punch-cards
  + Late 60s – Disks with random access files, start sharing files
    - start to have problems with sharing files
    - Hierarchical Data Model – IBM, tree structure, pointers
    - Network Data Model – IDS (Bachman), data points to each other based on relations, navigate one record at a time
  + 1970s – Relational Data Model proposed (Codd), E-R Model proposed (Chen)
    - Relational databases implemented by IBM and UC Berkeley
  + 1980s – Commercial Relational Databases (IBM, Sybase, Oracle, etc)
    - SQL is standardized
    - Object Oriented DBMSs, never caught on
  + 1990s – Continued SQL expansion, new application areas
    - Relational DBMSs incorporate objects

# entity-relationship model

* Used for conceptual database schema design
* 3 Steps
  + 1. Requirements Analysis – what users want from the database
  + 2. Conceptual Database Design – ER Model, design high level description of data
  + 3. Logical Database Design – Choose DBMS, translate model into actual database
* Describes the world in terms of entities, attributes, and relationships

## <See E-R Model Paper Notes>

## Entity SET

* Distinguishable object (Represented in a box)
* Weak Entities (Existence Dependancies) (Represented in double bordered entity/relationship)
  + Entities that have no primary key that can only exist in conjunction with another entity (the strong/dominant entity) and are deleted if their dominant entity is deleted
  + Have a key (dashed underline) that is primary in conjunction with the dominant entity’s key
  + Must have an N:1 total participation relationship with strong entity (i.e. each weak related to one and only one strong)
* Specialization/Generalization (Represented as Triangle)
  + Used to have subclasses of an entity set that share some characteristics
  + Subclasses inherit all superclass attributes
  + Constraints (represented as text next to triangle)
    - Overlap – allow multiple subclasses to contain the same entity (default no)
    - Covering – every superclass entity must belong to a subclass (default no)
  + Use to separate entity sets for different relationships or attributes applicable to some but not others

## Attribute

* Property of an entity set or relationship with a domain of allowed values, (Represented as an oval)
* Each entity set has a primary key set of attributes that uniquely identifies the entity (represented with underline)
* May have multivalued (set) attributes (Double line) and composite attributes (attributes with attributes)

## Relationship

* Represents a relation between entities (represented as a diamond)
* May have attributes, relations between multiple entities of a single entity set (must name roles)
* Key Constraints – How many relationships each entity may have
  + Placed on the connecting line to indicate how many relationships that entity can be in
  + Arrow (pointing to relationship) indicates at most 1
  + Bold Line indicates at least 1 (total participation, vs partial participation)
  + (L, U) to represent some other constraint

## Aggregation

* Consider a relationship as a higher order entity to allow it to engage in other relationships
* Represent with box around entire relationship

## Choices

* Attribute or Entity Set
  + Attribute unless it’s a separate object with more info about it, multiple can belong to an entity, it can be deleted, missing from some entities, or shared by different entities
* Tertiary relationships can always be broken down into multiple binary ones

# The Relational Model

* All data is organized in flat relations (tables)
* Simple and clean data model with powerful declarative query/update language
* Incorporates semantic integrity constraints and data independence

## Formal Definition

* Universe: A set of atomic values **D** (with equality)
* Domain: A name *D* with a set of values that is a subset of **D**
* Relation: (table)
  + Schema – R(A1 : *D*1 … Ak : *D*k) with
    - Name R, Set of distinct attributes A1 … Ak. and set of domains *D*1 … *D*k
    - E.g. Students(sid: string, name: string, age: integer, gpa: real)
  + Instance – A finite relation **R** ⊆ *D*1 x … x *D*k
* Database
  + Schema – Finite set of uniquely names relation schemas
  + Instance – A relation **Ri** for each relation instance (Ri)

## Properties

* Based on finite set theory
  + Ordering not important
  + Finite number of tuples in an instance
  + Just the values are visible and important
  + Set semantics: no duplicates, no ordering
* Attribute values are atomic
* Degree = number of attributes in the schema (# columns)
* Cardinality = number of tuples in the instance (# rows)

## Example

Database Schema: author(aid:int, name:string)

wrote(author:int, publication:int)

publication(pubid:int, title:string)

Database Instance: author = {(1, John), (2, Sue)}

Wrote = {(1, 1), (1, 4), (2, 3)}

Publication = {(1, Mathmatical Logic), (3, Book), (2, Pie)}

|  |  |
| --- | --- |
| aid | Name |
| 1  2 | John  Sue |

Table Form: <- author

## Relations vs SQL Tables

* SQL tables use multisets of tuples (you can have duplicates)
* SQL allows NULL (unknown) values which have special properties and require 3-value logic

## Integrity Constraints

* Relational schema only capture the structure of relations, we want to extend this to allow only certain rules
* An instance is only valid if it satisfies all schema integrity constraints
* Ensures data entry and modification respects the database design and protects the data from bugs in applications
* Types
  + Tuple-Level - Domain restrictions and attribute comparisons
  + Relation-Level
    - Key Constraints
      * Superkey: set of attributes for which no pair of distinct tuples will ever agree on
      * Candidate Key: a minimal superkey (set of attributes)
      * Primary Key: the designated candidate key
    - Functional Dependencies, where the value of one attribute depends on another
  + Database-Level
    - Referential Integrity
      * Foreign Key: primary key of one relation appearing as an attribute(s) of another relation
      * A tuple with a non-null value for a foreign key that does not match a primary key in the referenced relation is not allowed
    - Inclusion Dependencies
* Shown using arrows on a Database Schema diagram

# Relational Algebra

## Basics

* Consists of a set of operators taking zero or more relations as input and returning a single relation. Can be composed
* R is a relation name, E is a relational algebra expression
* Procedural – tells you how to get the result

## Primary Operators

* Selection: σcondition(E) [Remove Tuples not satisfying condition]
  + Result schema is the same, instance includes only those tuples that satisfy the condition
* Projection: πattributes(E) [Remove columns not listed, also removes duplicate tuples]
  + Result schema includes only the specified attributes, instance includes all tuples of E with all duplicates removed
* Rename: ρ(R(F), E) [Rename columns]
  + F is a list of terms of the form (oldname -> newname)
  + Returns the result of E with columns renamed according to F
* Product E1 x E2
  + Result schema has all attributes of E1 and E2. Instance includes one tuple for every pair of tuples
  + (# tuples)|E1 x E2| = |E1|\*|E2|, (# attributes) arity(E1 x E2) = arity(E1) + arity(E2)
  + Must rename any common attributes
* Joins
  + Conditional Join: E1 ⋈condition E2 [Join and remove tuples not satisfying condition]
    - Equivalent to σcondition(E1 x E2)
  + Equijoin: If we do a conditional join with only equalities, it is redundant to keep two same columns, so one is removed with a projection
  + Natural Join: E1 ⋈ E2 [Join tuples with matching common attributes (no duplicate columns)]
    - Essentially an Equijoin on all columns with matching names
    - Do cross product of E1 and E2, eliminate any tuples that don’t have all pairs of common attributes equal, remove duplicate attributes
    - If no common attributes, this is just a product
  + Outer Joins [Join tuples with matching common attributes and all tuples from specified side]
    - Left outer Join – includes all tuples from the left hand side (same for Right Outer Join)
    - Full outer Join – includes all tuples
    - Empty spaces become NULL

## Set Based Operators

* Union Compatible: Required for all set operators. Schemas must have the same number of fields with corresponding fields having the same type
* Union R ∪ S
  + Result includes all tuples that appear in either R or S or both
* Difference R – S
  + Result includes all tuples in R that do not appear in S
* Intersection R ∩ S
  + Result includes all tuples that appear in both R and S
* Division R / S [Inverse of Product]
  + Result is largest Q such that Q x B ≤ A, the inverse of product (RxS)/S = R
  + Number of columns of R/S = number of columns of R – number of columns of S

## Relational Completeness

* A query language that is at least as expressive as relational algebra is relationally complete
* SQL and safe relational calculus are both relationally complete

## Outer Joins

* Joins normally only join where all common attributes match
* Left outer Join – includes all tuples from the left hand side (same for Right Outer Join)
* Full outer Join – includes all tuples
  + Empty spaces become NULL

# Relational Calculus

## Basics

* Uses first order logic to specify properties of the query answer, doesn’t tell you how to get there
* Vocabulary includes constants, variables, predicates, functions, logical connective, quantifiers, and parenthesis
* Declarative – Tells you what constraints hold on the result tuples

## Tuple Relational Calculus (TRC)

* Variables take on tuples as values
* Query has the form {T | p(T)} – set of all tuples T such that p(T) holds (only one free variable T)
* p(T) is written using first order logic formulae, defined as: (variables T, S)
  + T ∈ Table
  + T.attribute *op* S.attribute
  + T.attribute *op* constant
  + Combination of formulas using first order operators (not, and, or, implies, there exists, for all)
* Variables implicitly have types

e.g. {W | ∃S(S ∈ Employee ^ W.lastname = E.lastname

^ W.hiredate = E.hiredate

^ E.salary > 100000)}

S ranges over the relation

W is the returned tuple(s)

## Domain Relational Calculus (DRC)

* Domain variables range over domain values of some attribute
* Queries have the form {<X1, X2, …, Xn> | P(<X1, X2, …, Xn>)} – each Xi is either a domain variable or a constant
  + Result is the set of tuples <X1, X2, …, Xn> for which the formula P evaluates to true
  + \_ = don’t care
* Returns a set of tuples

DOMAIN

e.g. {(N, D) | ∃S(Employee(\_,\_,N,\_,D,S) ∧ S > 100000)}

OR {(N, D) | ∃S[(\_,\_,N,\_,D,S) ∈ Employee ∧ S > 100000]}

S is a free variable (can have many) related to an attribute

N and D are bound variables, what is returned

## Safety of Relational Calculus

* May not have queries where the answer is not finite (unsafe queries)
* Query is safe if for all databases conforming to the schema, the query can be computed using only constants appearing in the database instance or the query itself

## Query By Example (QBE)

* Trademark of IBM, graphical language that is convenient for simple queries, but awkward for more complex queries
* Write in a table with attributes as columns (skeletons of each relation)
* Mark output attributes with a P., conditions are placed below attribute or in CONDITIONS box using variables
* Variable below attribute implicitly mean ‘equals’
* Multiple rows mean OR

# Translating E-R to Relational Tables

## Entitiy Sets

* Each entity set maps to a table, each entity is a row, primary key of entity set becomes primary key of table
* Weak entity sets also map to a table, but also include attributes of the identifying relationship and the primary key(s) of the dominant entity

## Relationship Sets

* Weak identifying relationship – do nothing
* If the general cardinality constraint is (1,1) for a component entity set E, add the attributes of the relationship set and the primary key attributes of the remaining component entity sets to table E
* Otherwise map the relationship set to a table with columns including attributes of the relationship set and primary keys of each component entity set (use foreign key in SQL)
  + Primary key is primary key of component E if cardinality constraint is (0,1)
  + Otherwise use primary key attributes of each component

## Aggregation

* Treat as the tabular representation of the relationship set

## Specialization

* Create table for higher level entity set, treat specialized entities as weak entity sets

## Generalization

* Either treat the same as specialization
* Or create a table for each lower-level entity set including its attributes and the attributes of the superset

## CLASSES

* Either map each entity set to a table, with subclasses having foreign keys to reference the superclass
* Or just map the subclasses with each having all attributes of the superclass (easier, but not always applicable)

# SQL Basics – DML

* SQL is a declarative language to specify modifications to a database
* Many more features than relational algebra
* Deals with multisets (duplicates allowed, no order)

## SELECT

* Specify which attributes you want to return (use variables if from multiple relations), must have in every query
* \* is shorthand for all attributes
* SELECT DISTINCT – eliminates duplicates
* Can do expressions (eg SELECT wage-4000), and aggregate funtions
* AS – rename attribute (SELECT wage-4000 AS newwage)
* Can have CASE statements (SELECT CASE WHEN wage<4000 THEN 0 ELSE wage-4000 END)

## FROM

* Specify which table/relation to use, must have in every query
* Name variables if using multiple relations (FROM Employee E, Project P, Employee EMgr) (if not named, table names are implicitly used)
* May create own table with a subquery FROM (…) AS A

## Where

* Specify conditions to be enforced on the returned data (+, -, /, \*, =, <, >, <=, >=, <>, and, or, not)
* Use AND to specify multiple conditions
* Filters out any tuples evaluating to FALSE or NULL
* LIKE – used for string matching
  + % - 0 or more arbitrary characters
  + \_ - exactly 1 character

## Insertion and Deletion

* INSERT INTO table VALUES (\_,\_,\_,\_); Insert a single tuple
* DELETE FROM table; Delete all tuples
* DELETE FROM table WHERE condition; Delete tuples meeting condition

## Update

* UPDATE table SET attribute = expression; - Update every row
* UPDATE table SET attribute = expression WHERE condition; - Update certain rows

## Set Operations

* Used between two select queries
* Q1 UNION Q2 – includes any tuple found at least once in Q1 or in Q2 (Eliminates duplicates between Q1 and Q2)
* Q1 INTERSECT Q2 – includes any tuple found at least once in both Q1 and Q2
* Q1 EXCEPT Q2 – includes any tuple that is found at least once in Q1 but is not in Q2
* Add ALL to any set operation to keep duplicates

## NULL Values

* Assigned to an attribute to represent unknown or missing data, can be prohibited
* Any operator on NULL returns NULL
* Operator IS NULL returns true if the value is NULL

## SubQueries

* Can be placed in the WHERE clause with a predicate (eg WHERE IN (Q))
* Predicates: IN, NOT IN, op SOME, op ALL, EXISTS, NOT EXISTS (op is one of =,<,>,<=,>=,<>)
  + First four, Q must return a single comparable attribute
  + EXISTS – tests if a subquery is non-empty
  + Be careful if the subquery can return NULL values
* Can refer to variables from the outer query
* Scalar Subqueries return an atomic value (one row/column) and can be placed almost anywhere
  + WHERE clause
  + SELECT clause
  + FROM clause – use ‘AS variable’
  + WITH clause – to create an additional table before the SELECT clause
    - Eg WITH Table(attributes) AS (Q) SELECT …
* Can have multiple nesting, no limit
* Can put together multiple attributes in brackets (a1, …, an) IN (…)

## Outer Joins

* Joins normally only join where all common attributes match
* Left outer Join – includes all tuples from the left hand side (same for Right Outer Join)
* Full outer Join – includes all tuples
  + Empty spaces become NULL
* Specify in FROM clause to use an outer join instead of a regular join

## Ordering Results

* No ordering can be initially assumed in any query
* ORDER BY - order the final result by some attribute (either ASC or DESC - default)

## Grouping/Aggregation

* GROUP BY clause – after where clause, allows you to group multiple tuples into one
* Aggregate Functions: COUNT, SUM, AVG, MAX, MIN. Can include DISTINCT keyword
* All attributes appearing in SELECT clause must be in either the GROUP BY clause or an aggregate function
* HAVING clause – filters groups (like the WHERE clause filters tuples) , can use aggregate functions
* Result Semantics:
  + Form cross product of relations in FROM clause
  + Eliminate tuples not satisfying WHERE clause
  + Group all tuples on that match on all attributes
  + Eliminate groups not satisfying the HAVING clause
  + Generate one tuple per group via the SELECT clause

# SQL Basics – DDL

* CREATE TABLE Name (Attribute CHAR(6), …) Create a table with specified columns
* ALTER TABLE Name ADD COLUMN Attribute VARCHAR(15) Add a column (modify schema)
* DROP TABLE Name Delete a table

## Data Types

* INTEGER, CHAR(length), VARCHAR(maxlength), DECIMAL(p, q) – p digits, q right of the decimal
* FLOAT(numbits), DATE (, TIME, TIMESTAMP, YEAR/MONTH INTERVAL, DAY/TIME INTERVAL, …

## INtegrit y COnstraints

* Go with table creation or alteration
* NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK
  + E.g. Column CHAR(6) NOT NULL PRIMARY KEY,
  + E.g. Column CHAR(3) NOT NULL REFERENCES Department ON DELETE CASCADE, (foreign key)
  + E.g. Column DECIMAL(9,2) CHECK (Column > 5),
  + E.g. CONSTRAINT constraint\_name UNIQUE(col1, col2, col3) (or use a CHECK with CONSTRAINT)
  + E.g. PRIMARY KEY (col1, col2)
  + E.g. FOREIGN KEY (col1, col2) REFERENCES table2
  + E.g. CHECK (constraint) – can be complicated, involving subexpressions

## Domain Constraints

* Can create own domains
  + CREATE DOMAIN name source DEFAULT x CHECK (…)
    - New domain called ‘name’, with underlying type ‘source’
    - Default value x (optional), must conform to check constraint (use VALUE in check constraint to reference a value in the domain)
  + CREATE TYPE instead creates a distinct type that can’t be compared with other different typed values, even source typed values, also can’t use existing operations

## ASSERTIONs

* CREATE ASSERTION name CHECK (…)
* Only used if referenced tables are non-empty

## Triggers

* Procedure executed by the database in response to a change in the database instance
* Components:
  + Event: type of change that will cause the trigger to fire
  + Condition: Test performed by the trigger to determine if action should be taken (either a true/false statement, or a query – considered true if returns non-empty set)
  + Action: What to do if condition is met
* Can create chains of triggers or recursive triggers, be careful
* Creates an active database, that does things for you behind the scenes, be aware of these things

# Advanced SQL

## Basics

* SQL has bindings for various programming languages that allow it to be used within, create a link between set-oriented SQL and the programming language to pass data back and forth (impedence mismatch)
* SQL is usually handled by a preprocessor

## C/C++

* All SQL statements are prefixed by “EXEC SQL” and suffixed with a ;
* General Form:
  + EXEC SQL INCLUDE SQLCA; (with includes)
  + EXEC SQL WHENEVER SQLERROR GOTO error; (error: is the code executed when an error occurs, not necessary)
  + EXEC SQL CONNECT TO database;
  + Run queries
  + EXEC SQL COMMIT
  + EXEC SQL CONNECT RESET

## STATIC EMBEDDED SQL

* Host Variables: allow data to be transferred between C and SQL
  + Contain declarations in EXEC SQL BEGIN DECLARE SECTION; and EXEC SQL END DECLARE SECTION;
  + When variables are used within SQL, must be prefixed with a :
  + Use sqlint\_32 for integers and char[x] for char/varchar
* Select Statements:
  + Use SELECT col1 … coln INTO :var1 … :varn …
  + After running the SELECT command, check SQLCODE to determine the result
    - <0 means error, 100 means no result
* NULL values
  + Require special indicator variables if a NULL value is a possibility
  + Indicator variables are of type “short int” and are listed directly after the relevant variable (in the INTO clause)
    - SELECT x INTO :num :num-indicator
* Cursors – used when a query returns more than one row
  + EXEC SQL DECLARE C1 CURSOR FOR <query>
  + EXEC SQL OPEN C1;
  + For(;;) EXEC SQL FETCH C1 INTO <variables>, if SQLCODE is 100 done, if SQLCODE <0 error, work with data
  + EXEC SQL CLOSE C1;
  + SQL defines different ways to FETCH (NEXT, PRIOR, FIRST, LAST, ABSOLUTE n, RELATIVE n), DB2 doesn’t support

## DYNAMIC EMBEDDED SQL

* Used when tables, columns, etc aren’t known when the application is written (and compiled), statements must be prepared during runtime
* In declare section, declare a char[x] (s) with the query as the string
* EXEC SQL PREPARE S1 FROM :s; (S1 is the name of the query)
* EXEC SQL EXECUTE S1; (can be done multiple times)
* Placeholders
  + If values are unknown, use question marks in the string as placeholders, use strcpy to get values into host variables, and use EXEC SQL EXECUTE S1 USING :var1 …; (still have to prepare the statement ahead of time)
  + Placeholders can only take the place of literals, not column names, etc.
* The INTO clause goes with the EXECUTE clause for dynamic
  + EXEC SQL EXECUTE S1 INTO … USING … (NOTE DB2 DOESN’T ALLOW THIS, use dynamic cursor)
* Dynamic Cursors
  + EXEC SQL DECLARE C1 CURSOR FOR S1;
  + EXEC SQL OPEN C1 USING …;
  + While(1) EXEC SQL FETCH FROM C1 INTO …;
* Descriptors
  + SQLDA – SQL Descriptor Area, used if the numbers and types of input/output values aren’t known
  + Uses DESCRIBE command

## MISC

* SQLJ
  + Embedded SQL in Java, supported by most DBMS’s, but not part of the SQL standard
* CLI – Call Level Interface
  + Vendor neutral ISO standard interface for SQL
  + Queries are represented as strings, they are then prepared and executed, use descriptors
* ODBC – Open Database Connectivity
  + Programming interface for SQL (like CLI)
* JDBC – java classes that provide interface like ODBC/CLI
* All these enable standardized embedded database access, so programs don’t need to be recompiled for each DBMS

## Stored Procedures

* Executes application logic directly inside the DBMS process
* Minimize data transfer, centralize application code, logical independence
* Atomic Valued Function (returns single value)
  + CREATE FUNCTION function(<argname> <type> …) RETURNS <type> LANGUAGE SQL RETURN <query>
  + Functions can then be used in the SELECT clause
* Table Valued Function
  + CREATE FUNCTION function(<argname> <type> …) RETURNS TABLE(<argname> <type> …) LANGUAGE SQL RETURN <query>
  + Use in FROM clause (FROM TABLE function(…) AS x)
* Multiple Results
  + CREATE PROCEDURE name(vars) RESULT SETS x LANGUAGE SQL BEGIN
  + DECLARE name1 CURSOR WITH RETURN FOR <query> … (x DECLARE’s)
  + OPEN name1 (one open per declare)
  + END
  + Use in DB2 with: CALL function(…)
  + Can use IF statements, etc. See slides example (last page of advanced SQL)

# Transactions

## Why

* Need to be able to manage concurrent access and issues due to failures during access
* Concurrency Issues (all involve at least one write)
  + Inconsistent reads (one application reading, one updating)
  + Lost Updates (Two applications updating the same thing at the same time)
  + Non-repeatable Reads (An application reads twice, but in the middle values get changed by another application, resulting in inconsistent values between the 2 reads)
* Failure Issues
  + System crashing in the middle of updating every row (some get updated, some don’t)
  + System crashing after an update is processed, but before changes are made permanent (eg because of a buffer)
  + System failing between two updates that are supposed to go together (eg a withdraw, then deposit bank account)

## Transactions

* Transaction: An atomic and durable unit of work, moves database between consistent states
* Properties (ACID)
  + Atomic: transaction occurs entirely or not at all
    - Uses logs to track changes
  + Consistency: transaction preserves the consistency of the database
  + Isolated: transactions do not interfere with each other
    - Uses mutual exclusion, locks, etc.
  + Durable: transactions changes are permanent once it is completed
* Formal Definition
  + oi(x) is an operation of a transaction T on data item x. o is either a read or a write and is atomic
  + OS = union of all oi
  + N = abort(rollback in SQL) or commit
    - Commit: Any updates the transaction did become completely permanant
    - Abort: any updates the transaction may have done and reversed, as if the transaction had never run
    - A transaction is considered active until it has either aborted or committed
  + A transaction T is a partial ordering of OS consisting of a set of operations Σ and an ordering ≺
    - If oi = r(x) and oj = w(x) for any data item x, then oi and oj have an ordering
    - N is always last in the ordering

## SQL

* New transaction is explicitly begun when application starts executing or after a commit/rollback
* Use COMMIT or ROLLBACK commands after each transaction
* Transaction Properties
  + SET TRANSACTION <mode>, <mode>, <mode>
    - Mode is either diagnostic size (# error conditions recorded), access mode (whether transaction is read only or read-write), or isolation level (determines how transaction interactions are managed)
  + Isolation Levels
    - 0 – Read Uncommitted – transaction may see uncommitted updates (as if no concurrency) – “dirty read”
      * May be incorrect if the update isn’t committed
    - 1 – Read Committed – transaction sees only committed updates, but non-repeatable reads are possible
      * E.g. T1 may read a value, then T2 updates that value and commits, then T1 reads the value again, but it is different
    - 2 – Repeatable Read – reads are repeatable, but phantoms are still possible
      * Phantom: New records inserted in between reads in a transaction
        + E.g. T1 reads some rows from a table using a WHERE clause, T2 inserts some tuples into the table and commits, T1 issues the same read again and may get more tuples than the first time, these additional tuples are phantoms
    - 3 – Serializability – Full concurrency protection, transactions appear to execute sequentially in some order
    - Lower levels give better performance, higher levels better protection

## Histories

* Conflicting Operations – two operations conflict if:
  + They belong to different transactions
  + They operate on the same object
  + At least one operation is a write
* Execution History: over a set of transactions 1 … n, is an interleaving of the operations for the transactions such that each transaction’s ordering is preserved (each transaction is a fixed length, made up of reads and writes)
* Conflict Equivalence: Two histories are conflict equivalent if they are over the same set of transactions and the ordering within each pair of conflicting operations is the same in both histories
* Serializability: A history H is serializable if there exists a serial history H’ that is conflict equivalent to H

## Serialization Graphs

* + Directed graph such that each transaction is a vertex
  + Edge for each pair of conflicting transactions on a particular input, points toward second operation in the history
  + A history is serializable iff its serialization graph is acyclic
  + To find Serial history: start with the transaction with no incoming edges, repeat

## Snapshot Isolation

* A transaction sees a consistent snapshot of the database taken when it started executing
* If it makes no updates, or updates that don’t conflict with updates made by any other transactions, it can commit
* Otherwise it has to rollback
* Read-write conflicts are avoided (get old values no matter what), and write-write conflicts are managed

# Views

* View: A relation in the external schema whose instance is determined by the instances of the relations in the conceptual schema
  + Shares many of the properties of a base relation in the conceptual schema
* Types
  + Virtual: Used only for querying, not stored in the database
  + Materialized: View is constructed and stored in the database
* Syntax (DDL)
  + CREATE [MATERIALIZED] VIEW name AS (query)
  + Views can then be accessed by name as if they were a base relation
* Updating – a modification to a view must be propagated to the conceptual schema
  + Sometimes its not possible to update unambiguously depending on how the view is defined
  + In order for a view to be updatable, it must satisfy a variety of conditions
    - Must reference only one table, output simple attributes, no grouping/aggregation/distinct, no nested queries, no set operations
* Materialized Views
  + When base tables are changes, the views may change as well, they need to be updated accordingly

# Schema Refinement & Normalization

Improving the schema design

## Relational Design Principles

* Avoid info repetition, spurious joins (joins that give extra tuples) and NULL values as much as possible
* Reduce anomalies with updates, inserts, deletes (eg with one giant table)
* Maintain all relationships (extreme – one table for each attribute)

## Functional Dependencies

* The idea that values of one set of attributes uniquely determine the values of another set of attributes
* Formal Definition: If R is a relation schema and X, Y ⊆ R are sets of attributes
  + The functional dependency X ⟶ Y holds on R if whenever an instance of R contains two tuples such that *t[X] = u[X]* then it’s also true that *t[Y] = u[Y]*
  + *X functionally determines Y (in R)*
* Any superkey K for a relation schema R functionally determines R (K ⟶ R) and vice versa (if duplicates not allowed)
* Closure: The closure of the set of functional dependencies F is the set of all functional dependencies that are satisfied by every relational instance that satisfies F (denoted F+)
  + F+ includes all the dependencies in F and any others that they imply
* Armstrong’s Axioms (are sound and complete)
  + Reflexivity: if Y ⊆ X then X ⟶ Y (e.g. A, B ⟶ B)
  + Augmentation: if X ⟶ Y then X,Z ⟶ Y,Z
  + Transitivity: if X ⟶ Y and Y ⟶ Z then X ⟶ Z
  + Union: if X ⟶ Y and X ⟶ Z then X ⟶ YZ (can be derived)
  + Decomposition: if X ⟶ Y,Z then X ⟶ Y (can be derived)
* Attribute Closure of X: The maximal set of attributes functionally determined by some X
  + ComputeX+(X, F): set X+ = X
    - While there exists some (Y ⟶ Z) in F such that Y ⊆ X+ and Z ⊈ X+ then add Z to X+
  + Theorem: X is a superkey of R iff ComputeX+(X, F) = R
  + Theorem: X ⟶ Y ∈ F+ iff Y ⊆ ComputeX+(X, F)

## Schema Decomposition

* If R is a relation schema (set of attributes), a decomposition of R is a set of relation schemas R1 … Rn such that R = R1 ∪ … ∪ Rn
* A good decomposition doesn’t lose information, complicate constraint checking, or contain additional anomalies
* Should be able to construct the original table instance from the decomposed tables instances (Lossless Join Decomposition if this is always true)
* A decomposition is lossless iff the common attributes of the decomposed relations form a superkey for either schema
* Dependency Preservation: Given a schema R and functional dependencies F, a decomposition is dependency preserving if there is an equivalent ser of functional dependences F’ such that none of them are interrelational (require joins to test) in D

## Normal Forms

* “Good” relational schema: One where each relation consists of a primary key and a set of mutually independent attributes. Independent relationships are stored in separate tables
* Superkey: set of attributes for which no pair of distinct tuples will ever agree on
* Boyce-Codd Normal Form (BCNF)
  + A database schema is in BCNF if each of its relation schemas is in BCNF
  + A relation schema is in BCNF iff any group of its attributes that functionally determines any other attributes functionally determines all other attributes (i.e. is a superkey)
  + Formal Definition:
    - Relation schema R (with functional dependencies F) is in BCNF iff whenever X ⟶ Y ∈ F+ and XY ⊆ R then either (X ⟶ Y) is trivial (Y ⊆ X) or X is a superkey of R
  + Redundancy is avoided by BCNF
  + DecomposeBCNF(R, F): set Result = R (R=relation schema [set of attributes])
    - While some Ri ∈ Result and (X ⟶ Y) ∈ F+ violate the BCNF condition
      * Replace Ri with Ri – (Y – X) in Result
      * Add {X, Y} to Result
    - There is no efficient way to do this, result depends on the sequence of FDs used to decompose the relations, and it is possible that no lossless join dependency preserving BCNF decomposition exists
  + Lossless decomposition always possible
* Third Normal Form (3NF)
  + A schema R is in 3NF iff whenever X ⟶ Y ∈ F+ and XY ⊆ R then either (X ⟶ Y) is trivial (Y ⊆ X) or X is a superkey of R or each attribute in Y – X is contained in a candidate key of R
  + Looser than BCNF because it allows more redundancy
  + Lossless and dependency preserving decomposition always possible
  + Decompose3NF(R, F): set Result = R (R=relation schema [set of attributes])
    - While some Ri ∈ Result and (X ⟶ Y) ∈ F+ violate the 3NF condition
      * Replace Ri with Ri – (Y – X) in Result
      * Add {X, Y} to Result
    - N = minimal cover for F – (union of all Fi’s)+
    - For each (X ⟶ Y) in N, add {X, Y} to Result (add relations to preserve missing dependencies)
  + Can be efficiently computed (see slides)

## Minimal Cover

* Two sets of dependencies F and G are equivalent iff F+ = G+
* A set of dependencies G is minimal if
  + Every right hand side of a dependency in G is a single attribute
  + For no X ⟶ A is the set G – {X ⟶ A} equivalent to G
  + For no X ⟶ A and Z a proper subset of X is the set G – {X ⟶ A} ∪ {Z ⟶ A} equivalent to G
* Theorem: For every set of dependencies G, there is an equivalent minimal set of dependencies (minimal cover)
* Finding a minimal cover:
  + Replace X ⟶ YZ with the pair X ⟶ Y and X ⟶ Z
  + Remove A from the left hand side of X ⟶ B in F if B is in ComputeX+(X – {A}, F)
  + Remove X ⟶ A from F if A ∈ ComputeX+(X, F – {X ⟶ A})

## Summary

* Want BCNF, but if that’s not possible, 3NF
* Need lossless-join, dependency preserving

# PHYSICAL DESIGN AND TUNING

* Physical Design: Process of selecting a physical schema (data structures/layout) for the database conceptual schema
* Tuning: Periodically adjusting schemas of a database to adapt to changing requirements and characteristics
* Workload Description: most important queries and updates and their frequencies and desired performance goal
  + Query – which relations are accessed, attributes retrieved, how selective, which attributes in join/select
  + Update – type of update and relations/attributes affected, how selective, which attributes in join/select
* Choose a storage strategy for each relation, then add indexes (speed up queries, slow updates)

## Indexes

* SQL: CREATE INDEX name ON table(attribute) [CLUSTER]
  + DROP INDEX name
* Reduce execution time for selections with conditions involving attribute
* Increase execution time for insertions
* May help or hurt updates/deletions
* Increases amount of space needed
* Clustering: An index on attribute A is a clustering index if tuples in the relation with similar values for A are stored together in the same block. Otherwise A is non-clustering (relation may have at most one clustering index)
* Co-Clustering: Two relations are co-clustered if their tuples are interleaved in the same file
  + Speeds up joins, slows sequential scans on either relation
* Range Queries (use B-trees)
* Multi-Attribute Indices: can have an index on multiple attributes, order is important
  + Tuples are arranged by the first attribute, then the second for those that have a common first, etc.
  + Good when conditions are on the attributes (must be on all earlier ones)
  + Great when selections are on the attributes (must be on all earlier ones)
  + Not useful when only the second attribute is used

## Guidelines

* Don’t index unless the performance increase outweighs the update overhead
* Attributes in WHERE clause are candidates for indexes
* Choose indices that benefit as many queries as possible
* Only one clustering index per schema (range queries benefit most from clustering, join queries benefit most from co-clustering)
* DB2 has an index advisor that will tell you its best guess for what the indices should be based on a query

## Tuning the Conceptual Schema

* Often involved changes to the user end
* Normalization: process of decomposing schemas to reduce redundancy
* Denormalization: process of merging schemas to increase redundancy
  + Increased redundancy also increases update overhead, but decreases query overhead
  + May want different degrees of normalization depending on what the database will be used for
* Partitioning
  + Horizontal
    - Keep all the columns, but separate the rows into subsets
    - Tuples are assigned to a partition based on some natural criteria
    - Often used to separate archival from operational data
  + Vertical
    - Keep all the rows, but partition the columns
    - Usually used to separate frequently used columns from rarely used ones
* Tuning Queries - Guidelines
  + Avoid unnecessary uses of ORDER BY, GROUP BY, DISTINCT (since sorting is expensive)
  + Replace subqueries with joins, replace correlated subqueries with uncorrelated
  + Watch the statistics of the database and make modifications if necessary
* Tuning Applications – Guidelines
  + Minimize communication costs – return as few rows/columns as possible, use WHERE to update multiple rows instead of a cursor
  + Minimize lock contention and hot spots – delay updates and hot-spot operations, keep transactions short, perform them in batches, consider lower isolation levels

# Security and Authorization

## Intro

* Objectives
  + Secrecy: Info should be shown only to people who are allowed to see it
  + Integrity: Info should only be modified by those who are allowed to
  + Availability: Everyone who is allowed to should be able to see and/or modify data
* Security Policy: defines who is allowed to see/modify specific data
  + Implemented by access control mechanisms (two types)

## Discretionary Access Control

* Specify which schema objects (tables, views) a user may access
* Users are given privileges to access an object and are then able to grant privileges to others (if grant option is given)
* GRANT and REVOKE commands
  + GRANT privileges ON object TO users [WITH GRANT OPTION]
  + REVOKE [GRANT OPTION FOR] privileges ON object FROM users {RESTRICT or CASCADE}
  + (possible privileges are SELECT, INSERT(column), UPDATE(column), DELETE, REFERENCES(column))
  + When a privilege is revoked from a user X, it is also revoked from all users that were granted the privilege solely from X
* Views can be used to allow access to only certain tuples of a table

## Mandatory Access Control

* Specify which data (instance) objects a user may access
* Susceptible to Trojan Horse attacks (trick copying)
* Bell-LaPadula Model
  + Objects get security classes (tables, views, rows, columns)
  + Subjects get security clearances (users, programs, roles)
  + Subjects can only access object of lesser or equal security levels
  + Individual tuples or columns can be assigned different security classes
    - Users with different clearances see different tables
    - Security class is essentially part of the key for rows

# Data Warehousing

## Transaction Processing

* Most relational databases are used for operational data (eg students enrollments, customers purchases, etc)
* On-Line Transactional Processing (OLTP): Simple queries with many short transactions making small changes
* Tuned to maximize throughput of concurrent transactions

## Data Warehousing

* Separate copy of operational data used for decision support (summarizing data for high level decision making – complex aggregation queries) and data mining (searching for trends in data to exploit – intensive simple queries)
* On-Line Analytical Processing (OLAP): data modeled as a multidimensional array, ad hoc queries (iterative) that select and aggregate cells of the array, for use in decision support (2 categories)
  + Special Purpose – special query language, stored as multidimensional arrays (MOLAP)
  + Relational – stored in relations, uses SQL (ROLAP)
* Star Schemas – Main table of IDs, multiple tables of attributes (see slides)
* Often want to look at a subset of the whole table (via aggregations)
* Data Cube: Extends a multidimensional array to include all possible aggregated totals (see slides) – sums everything, usually stored with NULL as the ID in relations
  + SQL CUBE operator

## Warehouse

* Steps
  + Extract – run queries to retrieve the necessary data from the operational database
  + Clean – Delete or repair tuples with missing/invalid info
  + Transform – reorganize data to fit warehouse conceptual schema
  + Load – populate warehouse tables, build indexes and views
  + Data needs to be refreshed periodically (nightly or weekly usually)
* Materializing Views
  + Pre calculate some aggregates
  + Need to decide which ones, same issues apply that apply to the entire warehouse
    - When to refresh, how to refresh
    - Which indexes and views are useful