Contents

1 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

2 Data Management

"Data management is the development, execution and supervision of plans, policies, programs and practices that control, protect, deliver and enhance the value of data and information assets." (The DAMA Guide to the Data Management Body of Knowledge)

2.1 Information and Data

2.1.1 Information

Knowledge about facts, events, and/or processes

- The presence (and absence) of information may influence decisions
- Common assumption: Information may be transferred

2.1.2 Communication

"Two-way process of reaching mutual understanding, in which participants not only exchange (encode and decode) information, new ideas and fealings but also create a **shared meaning**."

- How do we transmit information?
  Norbert Wiener, the "father of information theory", said something along the lines:
  "I don't know, what information is. What I know for sure however is, that it is neither matter nor energy"
- A channel transmits physical phenomena, intended to be interpreted properly
  so-called **signals**.

2.1.3 Data

Symbols or functions, which represent information for processing purposes, based on explicit or implicit agreements, so-called **interpretation rules**.

- Implicit interpretation rule e.g. as in natural languages
- Explicit interpretation rule e.g. as for trafic lights: signals red, yellow, green

2.2 Motivation

- Data management deals with design, usage, maintenance, and administration of data together with their interpretation rules
- Data management supports efficiency and security when dealing with the enterprise resource information
- What is the value of our data?
  - $>$ the accumulated effort/cost spent to collect, maintain, the data
  - there is no added value, otherwise!
- What is the lifecycle of our data?
  - $>$ 20 years (on average)
  - Much longer than the lifecycle of software used to produce it

3 Database Terminology

3.1 Database - set of persistent data

- A very generic term...
  - From flat text-files with simple records ...
  - ... up to distributed multi-TB data warehouses!
- Intended to be available for use by multiple application programs and/or users
- Together with agreements regarding data structures, their semantics and encodings (often called metadata)

3.2 Database Management System (DBMS)

Softwaresystem supporting the following functionality

- Definition and maintenance of data structures
- Creation, modification and deletion of data conforming to these data structures

- Administration of data structures, data, users, and applications

3.3 Schema

Formal definition of a database

- At least regarding data structures, semantics, encodings
- Preferrably also regarding integrity rules, access paths, authorization,

etc.

3.4 Database System (DBS)

Specific installation of a DBMS together with all databases controlled by this installation

3.5 Information System

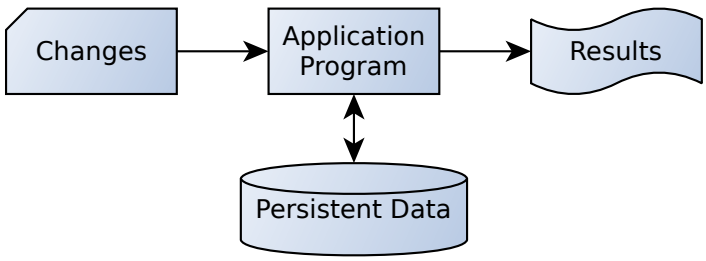
Database together with the application software used to process its content

4 Purpose of Database Systems

4.1 Before DBMSes Existed

database applications were built directly on top of file systems

- Ad-hoc or purpose-built data files
- Special-built programs implemented various operations against the database



4.1.1 Drawbacks

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
  - Data formats and organization optimized for a single application
- Integrity problems
  - Integrity constraints “buried” in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones

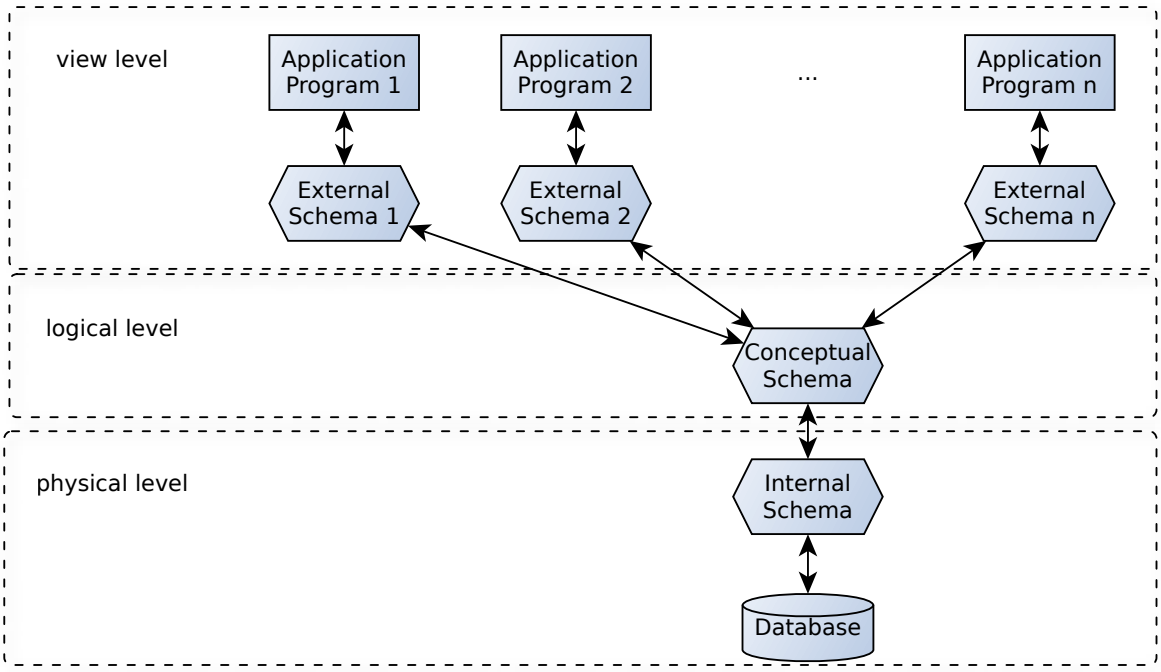
- Atomicity of updates
  - Failures may leave database in an inconsistent state
    - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Example: Flight reservation
- Security problems
  - Hard to provide user access to some, but not all, data

5 View of Data

- Major purpose of a database system is to provide an **abstract** view of the data.
  hide details of how data are stored and maintained

5.1 Data Abstraction / 3-Schema-Architecture (ANSI/SPARC-Architecture)

- Purpose:
  - Provide layers of abstraction to isolate users, developers from database implementation
  - Provide generic database capabilities that specific applications can utilize



- Physical level: **How** vales are stored/managed on disk
  - Internal Schema defines:
    - the data structures used to store the DB-content physically on all levels
    - the access paths
- Logical level: **What** data are stored, relationships exist
  - Conceptual Schema defines:

        * the logical information content of a database
          entities and associations
        * the global integrity constraints
- View level: queries and operations that users can perform
  - External Schema defines:
    * application and/or user and/or programming language specific views on the DB-content
    * application and/or user specific integrity constraints
    * authorizations of applications and/or users
- **Problem**
  Complexity of mappings between schemas
  - Conceptual and Internal schema
  - Conceptual and External schema

6 Database Technology Types

- Different types of databases, based on
  - usage
  - amount of data beeing managed
    * small, application specifict, (embeded databases), e.g. SQLite
    * large, (data warehousing), e.g. Oracle
  - Type/frequency of operations being performed
    * often, Online Transaction Processing (OLTP)
    * rare, Online Analytical Processing (OLAP)
  - data model

6.1 Data Model

The data model describes:

- What **data** can be stored.
- How **data relationships** are represented.
- Which **consistency constraints** can be enforced.
- How to **access and manipulate** the data.

On the physical, logical and view levels.

6.1.1 Relational Model

- **Most** database systems use the relational model!
- Conceptually all data in simple tables
- Logical access paths have to be built using data (key value) references
- Support for integrity constraints
- Support for views (external schema)
- Applications independent of physically implemented structures (internal schema)
- Examples: Oracle, db2, MySQL, PostgreSQL

6.1.2 Entitiy-Relationship Data Model

- Used for logical database design!
- Much higher level model than relational model
- Not supported by most databases, but used in many database design tools
- Easy to translate into the relational model

6.1.3 Object-based Data Models

6.1.3.1 Object-oriented Data Model

- Data model for object-oriented programming for popular OO languages such as C#, C++, Java,...
- **Quote:**
  "If we were applying relational technology to cars, we would upon return from a ride drive into the garage, disassemble the car and store the pieces properly on the shelves. We can re-assemble our car the next morning before driving to work."
- Object-Oriented Databases:
  - Persistent objects
  - Often with object lifecylcle
  - Problems
  - Lack of implemented standards
  - Lack of data re-usability
  - Examples: Objectivity, DB4O, ObjectStore

6.1.3.2 Object-Relational Databases

- Relational Databases with extensions
  - Data type definitions
    - Structures
    - Operations
  - Support of Pointers /Handles to define logical access paths
  - Methods
  - Examples: Oracle, db2, MySQL, PostgreSQL

6.1.4 Semistructured Data Model

- Individual data items of the same type may have different sets of attributes!
- Semantic mark-up improves data interpretation
  Querying, Optimization
- Hierarchical objects
  Potentially with links
- Examples: Tamino, Ipedo

6.1.5 Alternative Models

- NoSQL:
  key-value store, document store, column store or graph store

- Relax most of the constraints imposed by relational model
- Distribution over large number of nodes
  - Most don't offer consistent updates
  - Many if not most start offering SQL as query language
  - Optimized for fast retrieval of information
    - which doesn't change often
    - Or where the changes don't matter
- Examples: CouchDB, MongoDB, BigTable, Cassandra, Hbase, Redis

7 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

8 History

- Invented by Edgar F. Codd (early 1970s)
  Codd, Edgar F. "A relational model of data for large shared data banks." Communications of the ACM 13.6 (1970): 377-387.
- Focus was data independence from physical level design and implementation
- First implementers were IBM, Oracle
  - SQL

9 Structure of Relational Databases

9.1 Relations

- Relations are basically **tables** of data
  - Each row represents a record in the relation

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

- A relational database is a set of relations
  - Each relation has a unique name in the database
- Each row in the table specifies a relationship between the values in that row
  - The **instructor_id** "10101", **name** "Srinivasan", **dept_name** "Comp. Sci." and **salary** "65000" are all related to each other

9.2 Attributes

- Each relation has some number of **attributes**
  - Sometimes called “columns”

9.2.1 Domains

- Each attribute has a **domain**
  - Specifies the set of valid values for the attribute
- The relational model constrains attribute domains to be atomic
  - Values are indivisible units
- Mainly a simplification
  - Virtually all relational database systems provide non-atomic data types
- Attribute domains may also include the **null** value
  - **null** = the value is unknown or unspecified
  - **null** can often complicate things. Generally considered good practice to avoid wherever reasonable to do so.
- The **instructor** relation:
  - 4 attributes
  - Domain of **salary** is the set of nonnegative integers
- Domain of **dept_name** is the set of all valid department names in the university

9.3 Tuples

- Each **row** is called a **tuple**
  - A fixed-size, **ordered set** of name-value pairs
- A tuple variable can refer to any valid tuple in a relation
- Each attribute in the tuple has a unique name. Can also refer to attributes by index
  - Attribute 1 is the first attribute, etc.
- Example:
  - Let tuple variable t refer to first tuple in instructor relation
  - $t[salary] = 350$
  - $t[2] =$ "Srinivasan"
- In the instructor relation:
  - Domain of **instructor_id** is D_1
- Domain of **name** is D_2
- Domane of **dept_name** is D_3
- Domain of **salary** is D_4
- The instructir relation is a subset of the tuples in the Cartesian product $D_1 \times D_2 \times D_3 \times D_4$
- Each tuple included in instructor specifies a relationship between that set of values
  - Hence the name, “relational model”
  - Tuples in the instructor relation specify the details of valid instructurs of the university
- A relation is a set of tuples
  - Each tuple appears exactly once
  - If two tuples t_1 and t_2 have the same values for all attributes, then t_1 and t_2 are the same tuple
    (i.e. $t_1 = t_2$)
- The order of tuples in a relation is not relevant

9.4 Relation Schemas

- Every relation has a **schema**
  - Specifies the type information for relations
  - Multiple relations can have the same schema
- Relation schemas include:
  - an ordered sets of attributes
  - the domain of each attribute
- Naming conventions:
  - Relation names are written as all lowercase
  - Relation schema's name is capitalized
- Example:
  - The relation schema of **instructor** is:
    Instructor_schema = (instructor_id, name, dept_name, salary)
  - To indicate that **instructor** has schema **Instructor_schema**
    instructor(Instructor_schema)
- Can use set operations on them
  - Examples:
    - Relations $r(R)$ and $s(S)$
      - Relation r has schema R
      - Relation s has schema S
  - $R \cap S$
    - The set of attributes that R and S have in common
  - $R - S$
    - The set of attributes in R that are not also in S
    - (And, the attributes are in the same order as R)
  - $K \subseteq R$
    - K is some subset of the attributes in relation schema R

9.4.1 Relation Variables

- More formally:
- instructor is a **relation variable**
  - A name associated with a specific schema, and a set of tuples that satisfies that schema
  - (sometimes abbreviated "relvar")
- A specific set of tuples with the same schema is called a **relation value** (sometimes abbreviated "relval")
  - (Formally, this can also be called a relation)
  - Can be associated with a relation variable
  - Or, can be generated by applying relational operations to one or more relation variables
- Problem:
  - "Relation" normally means the collection of tuples
- The term "relation" is often used in slightly different ways
  - i.e. "relation" usually means "relation value"
- It is often used less formally to refer to a relation variable and its associated relation value
  - e.g. "the instructor relation" is really a relation variable that holds a specific relation value

10 Database Schema

11 Keys

- Relations are **sets** of tuples...
  - No two tuples can have the same values for all attributes...
  - But, some tuples might have the same values for some attributes
- Example:
  - Some instructors have the same salary
  - Some instructors are assigned to the same department

11.1 Superkeys and Candidate Keys

- A **superkey** is a set of attributes that uniquely identifies tuples in a relation
- Example:
  - {instructor_id} is a superkey
  - Is {dept_name} a superkey?
    - No. Each department can have multiple instructors
- Adding attributes to a superkey produces another superkey
  - If {instructor_id} is a superkey, so is {instructor_id, salary}
  - If a set of attributes $K \subseteq R$ is a superkey, so is any superset of K
  - Not all superkeys are equally useful...
- A **minimal** superkey is called a **candidate key**
  - A superkey for which no proper subset is a superkey
  - For instructor, only {instructor_id} is a candidate key

11.2 Primary Keys

- A relation might have several candidate keys
- In these cases, one candidate key is chosen as the primary means of uniquely identifying tuples
  - Called a **primary key**
- Example: **student** relation

student

student_id	name	dept_name	tot_cred	student_ssn
12856	Zhang	Comp. Sci.	102	3688 290383
12345	Shankar	Comp. Sci.	32	3134 181283
19991	Brandt	History	80	3500 180584

  - Candidate keys could be: {student_id}, {student_ssn}
  - Choose primary key: {student_id}
- Keys are a property of the relation schema, not individual tuples
  - Applies to all tuples in the relation

- Primary key attributes are listed first in relation schema, and are underlined
- Examples:
  - Instructor_schema = (instructor_id, name, dept_name, salary)
  - Student_schema = (student_id, name, dept_name, tot_cred, student_ssn)
- Only indicate primary keys in this notation
  - Other candidate keys are not specified
- Multiple records cannot have the same values for a primary key!
  - . . . or any candidate key, for that matter. . .
- Example: student(student_id, name, dept_name, tot_cred, student_ssn)
  - Two students cannot have the same ID.
- This is an example of an invalid relation

student

student_id	name	dept_name	tot_cred	student_ssn
12856	Zhang	Comp. Sci.	102	3688 290383
12345	Shankar	Comp. Sci.	32	3134 181283
19991	Brandt	History	80	3500 180584
...	...	...	...	...
12856	Miller	Comp. Sci.	180	3588 030383

  - Set of tuples doesn't satisfy the required constraints

11.3 Keys Constrain Relations

- Primary keys constrain the set of tuples that can appear in a relation
  - Same is true for all superkeys
- For a relation r with schema R
  - If $K \subseteq R$ is a superkey then
    $\langle \forall t_1, t_2 \in r(R) : t_1[K] = t_2[K] : t_1[R] = t_2[R] \rangle$
  - i.e. if two tuple-variables have the same values for the superkey attributes, then they refer to the same tuple
  - $t_1[R] = t_2[R]$ is equivalent to saying $t_1 = t_2$

11.4 Choosing Candidate Keys

- Since candidate keys constrain the tuples that can be stored in a relation. . .
  - Attributes that would make good (or bad) candidate keys depend on what is being modeled
- Example: student name as candidate key?
  - Very likely that multiple people will have same name
  - Thus, not a good idea to use it as a candidate key
- These constraints motivated by external requirements
  - Need to understand what we are modeling in the database

11.5 Foreign Keys

- One relation schema can include the attributes of another schema's primary key
- Example: advisor relation
  - Advisor_schema = (student_id, instructor_id)
  - Associates students with instructors
  - student_id and instructor_id are both foreign keys
    - student_id references the primary key of student
    - instructor_id references the primary key of instructor
  - advisor is the referencing relation
    - It refers to the student and instructor relations
  - student and instructor are the referenced relations

student

student_id	name	dept_name	tot_cred	student_ssn
12856	Zhang	Comp. Sci.	102	3688 290383
12345	Shankar	Comp. Sci.	32	3134 181283
19991	Brandt	History	80	3500 180584
...	...	...	...	...

advisor

student_id	instructor_id
12856	45565
12345	10101
23121	76543
...	...

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
...	...	...	...

11.5.1 Foreign Key Constraints

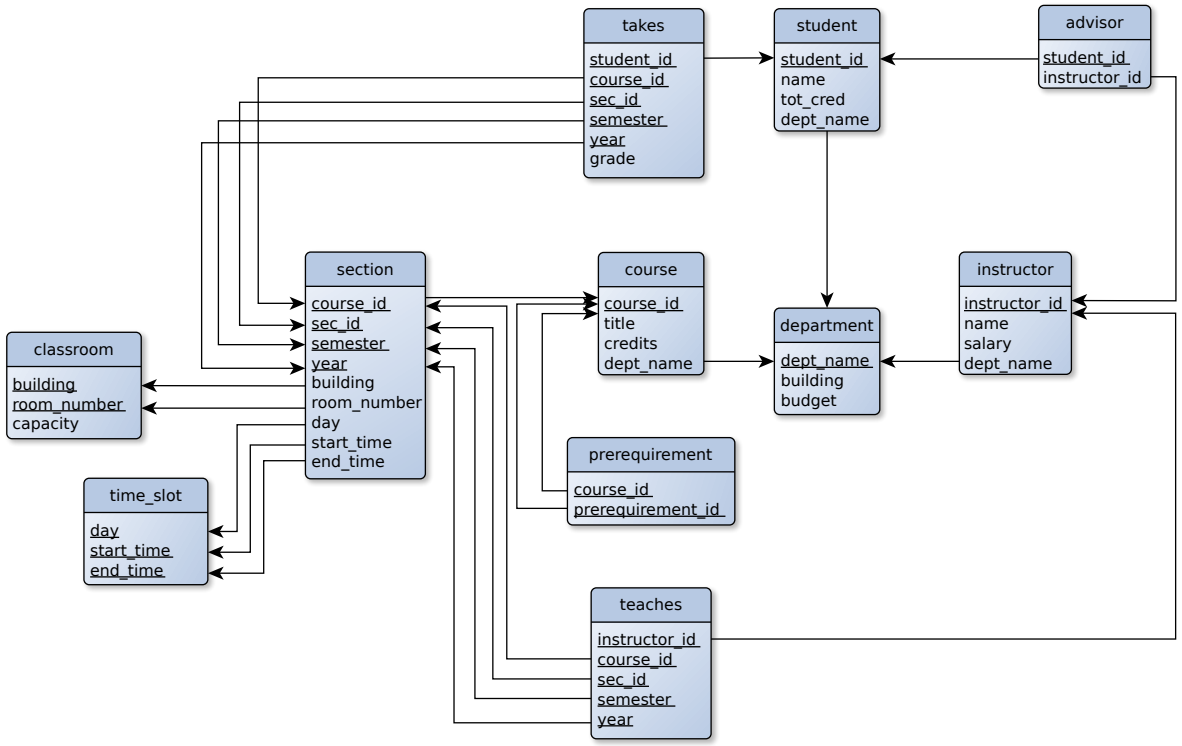
- Tuples in advisor relation specify values for student_id
  - student relation must contain a tuple corresponding to each student_id
- Same is true for instructor_id values and instructor relation
- Valid tuples in a relation are also constrained by foreign key references value in advisor
  - Called a **foreign-key constraint**
- Consistency between two dependent relations is called **referential integrity**
  - Every foreign key value must have a corresponding primary key value
- Given a relation $r(R)$
  - A set of attributes $K \subseteq R$ is the primary key for R
- Another relation $s(S)$ references r
  - $K \subseteq S$ too
  - $\langle \forall t_s \in s : \exists t_r \in r : t_s[K] = t_r[K] \rangle$
- Notes:
  - K is not required to be a candidate key for S, only R
  - K may also be part of a larger candidate key for S

11.5.2 Primary Key of advisor Relation?

- Advisor_schema = (student_id, instructor_id)
- If {student_id} is the primary key:
  - A student can only have one instructor as advisor
    - Each students's ID can appear only once in advisor
  - A instructor could advise multiple students

- If {instructor_id} is the primary key:
  - Each instructor can only advise one student
    - Each instructor ID can appear only once in advisor
  - Students could have multiple instructors as advisors
- If {student_id, instructor_id} is the primary key:
  - Instructors can advise multiple students
  - Students can have multiple advisors
- Last option is how most universities really work

12 Schema Diagrams



13 Query Languages

- A query language specifies how to access the data in the database
- Different kinds of query languages:
  - **Declarative** languages specify what data to retrieve, but not how to retrieve it
  - **Procedural** languages specify what to retrieve, as well as the process for retrieving it
- Query languages often include updating and deleting data as well
- Also called data manipulation language (DML)

13.1 Relational Algebra

- A procedural query language
- Comprised of relational algebra operations
- Relational operations:

  - Take one or two relations as input
  - Produce a relation as output
- Relational operations can be composed together
  - Each operation produces a relation
  - A query is simply a relational algebra expression
- Six "fundamental" relational operations
- Other useful operations can be composed from these fundamental operations

13.2 Why Relational Algebra?

- SQL is only loosely based on relational algebra
- SQL is much more on the "declarative" end of the spectrum
- Many relational database implementations use relational algebra operations as basis for representing execution plans
  - Simple, clean, effective abstraction for representing how results will be generated
  - Relatively easy to manipulate for query optimization

13.3 Fundamental Relational Algebra Operations

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{\text{salary} \geq 85000}(\text{instructor})$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{\text{instructor\_id,salary}}(\text{instructor})$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
$\times$ (Cartesian Product)	instructor $\times$ department Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
$\cup$ (Union)	$\Pi_{\text{name}}(\text{instructor}) \cup \Pi_{\text{name}}(\text{student})$ Output the union of tuples from the two input relations.
$-$ (Set Difference)	$\Pi_{\text{name}}(\text{instructor}) - \Pi_{\text{name}}(\text{student})$ Output the set difference of tuples from the two input relations.
ρ (Rename Operation)	$\rho_{\text{test}}(\text{instructor})$ Assigns a new name to a relation within the scope of an expression.

- Each operation takes one or two relations as input
- Produces another relation as output
- Important details:
  - What tuples are included in the result relation?
  - Any constraints on input schemas? What is schema of result?

13.3.1 Select Operation - selection of rows (tuples)

- Written as: $\sigma_P(r)$
  - P is the predicate for selection
    - P can refer to attributes in r (but no other relation!), as well as literal values
    - Can use comparison operators: $=, \neq, <, \leq, >, \geq$
    - Can combine multiple predicates using: $\wedge$ (and), $\vee$ (or), $\neg$ (not)
- r is the input relation
- Result relation contains all tuples in r for which P is true

- Result schema is identical to schema for r

13.3.1.1 Examples

Using the instructor relation:

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

- "Retrieve all tuples for instructors in the Physics department."
  - $\sigma_{\text{dept\_name="Physics"}}(\text{instructor})$

instructor

instructor_id	name	dept_name	salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

- "Retrieve all tuples for instructors in the Physics department with a salary under 90000."
  - $\sigma_{\text{dept\_name="Physics"} \wedge \text{salary}<90000}(\text{instructor})$

instructor

instructor_id	name	dept_name	salary
33456	Gold	Physics	87000

13.3.2 Project Operation - selection of columns (attributes)

- Written as: $\Pi_{a,b,\ldots}(r)$
- Result relation contains only specified attributes of r
  - Specified attributes must actually be in schema of r
  - Result's schema only contains the specified attributes
  - Domains are same as source attributes' domains
- Important note:
  - Result relation may have fewer rows than input relation!
  - Why?
    - Relations are sets of tuples, not multisets

13.3.2.1 Examples

Using the instructor relation:

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

- "Retrieve all department names that have at least one instructor."
  - $\Pi_{\text{dept\_name}}$(instructor)

$\Pi_{\text{dept\_name}}$(instructor)

dept_name
Comp. Sci.
Finance
Music
Physics
History

- Result only has five tuples, even though input has six
- Result schema is just (dept_name)

13.3.3 Composing Operations

- Input can also be an expression that evaluates to a relation, instead of just a relation

13.3.3.1 Examples

- "Retrieve all names of instructors with a salary less or equal to 65000"
  - $\Pi_{\text{name}}(\sigma_{salary \leq 65000}(\text{instructor}))$
  - Inputs relation's schema is:
    - Instructor_schema = (instructor_id, name, dept_name, salary)
  - Final result relation's schema:
    - (name)
  - Distinguish between **base** and **derived** relations
    - instructor is a base relation
    - $\sigma_{salary \leq 65000}$(instructor) is a derived relation

13.3.4 Set-Union Operation

- Written as: $r \cup s$
- Result contains all tuples from r and s
  - Each tuple is unique, even if it's in both r and s
- Constraints on schemas for r and s?
  - r and s must have compatible schemas:
    - r and s must have same arity (same number of attributes)
    - For each attribute i in r and s, $r[i]$ must have the same domain as $s[i]$

13.3.4.1 Examples

- Using the instructor and student relations:

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

student

student_id	name	dept_name	tot_cred	student_ssn
12856	Zhang	Comp. Sci.	102	3688 290383
12345	Shankar	Comp. Sci.	32	3134 181283
19991	Brandt	History	80	3500 180584

- "Retrieve names of all persons associated with the university either beeing an instructor or a student"
  - Easy to find the names of instructors:
    - $\Pi_{\text{name}}(\text{instructor})$

      $\Pi_{\text{name}}(\text{instructor})$

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
  - Also easy to find the names of students:
    - $\Pi_{\text{name}}(\text{student})$

      $\Pi_{\text{name}}(\text{student})$

name
Zhang
Shankar
Brandt
  - Result is set-union of these expressions:
    - $\Pi_{\text{name}}(\text{instructor}) \cup \Pi_{\text{name}}(\text{student})$

      $\Pi_{\text{name}}(\text{instructor}) \cup \Pi_{\text{name}}(\text{student})$

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Zhang
Shankar
Brandt

13.3.5 Set-Difference Operation

- Written as: $r - s$
- Result contains tuples that are only in r, but not in s
  - Tuples in both r and s are excluded
  - Tuples only in s do not affect the result
- Constraints on schemas of r and s?
  - Schemas must be compatible (Exactly like set-union.)

13.3.5.1 Examples

- "Find all departments which have associated instructors but no students"
  - All departments that have at least one associated instructor
    - $\Pi_{\text{dept\_name}}(\text{instructor})$

      $\Pi_{\text{dept\_name}}(\text{instructor})$

dept_name
Comp. Sci.
Finance
Music
Physics
History
  - All departments that have at least on associated student
    - $\Pi_{\text{dept\_name}}(\text{student})$

      $\Pi_{\text{dept\_name}}(\text{student})$

dept_name
Comp. Sci.
History
  - Result is the set-difference of these expressions
    - $\Pi_{\text{dept\_name}}(\text{instructor}) - \Pi_{\text{dept\_name}}(\text{student})$

      $\Pi_{\text{dept\_name}}(\text{instructor}) - \Pi_{\text{dept\_name}}(\text{student})$

dept_name
Finance
Music
Physics

13.3.6 Cartesian Product Operation (joining two relations)

- Written as: $r \times s$
  - Read as "r cross s"
- No constraints on schemas of r and s
- Schema of result is concatenation of schemas for r and s
- If r and s have overlapping attribute names:
  - All overlapping attributes are included; none are eliminated
  - Distinguish overlapping attribute names by prepending the source relation's name
    - Example
      - Input relations: $r(a, b)$ and $s(b, c)$
      - Schema of $r \times s$ is $(a, r.b, s.b, c)$
- Result of $r \times s$
  - Contains every tuple in r, combined with every tuple in s
  - If r contains N_r tuples, and s contains N_s tuples, result contains $N_r * N_s$ tuples

- Allows two relations to be compared and/or combined
  - If we want to correlate tuples in relation r with tuples in relation s...
  - Compute $r \times s$, then select out desired results with an appropriate predicate

13.3.6.1 Examples

- Using the instructor and department relations:

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

department

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000

- Compute result of instructor × department
- Result will contain $6 * 3 = 18$ tuples
- Schema for instructor is:
  - Instructor_schema = (instructor_id, name, dept_name, salary)
- Schema for department is:
  - Department_schema = (dept_name, building, budget)
- Schema for result of instructor × department is:
  - (instructor_id, name, instructor.dept_name, salary, department.dept_name, building, budget)
- Overlapping attribute names are distinguished by including name of source relation
- Result

instructor × department

instructor_id	name	instructor.dept_name	salary	department.dept_name	building	budget
10101	Srinivasan	Comp.Sci.	65000	Biology	Watson	90000
10101	Srinivasan	Comp.Sci.	65000	Comp. Sci.	Taylor	100000
10101	Srinivasan	Comp.Sci.	65000	Elec. Eng.	Taylor	85000
12121	Wu	Finance	90000	Biology	Watson	90000
12121	Wu	Finance	90000	Comp. Sci.	Taylor	100000
12121	Wu	Finance	90000	Elec. Eng.	Taylor	85000
15151	Mozart	Music	40000	Biology	Watson	90000
15151	Mozart	Music	40000	Comp. Sci.	Taylor	100000
15151	Mozart	Music	40000	Elec. Eng.	Taylor	85000
22222	Einstein	Physics	95000	Biology	Watson	90000
22222	Einstein	Physics	95000	Comp. Sci.	Taylor	100000
22222	Einstein	Physics	95000	Elec. Eng.	Taylor	85000
32343	ElSaid	History	60000	Biology	Watson	90000
32343	ElSaid	History	60000	Comp. Sci.	Taylor	100000
32343	ElSaid	History	60000	Elec. Eng.	Taylor	85000
33456	Gold	Physics	87000	Biology	Watson	90000
33456	Gold	Physics	87000	Comp. Sci.	Taylor	100000
33456	Gold	Physics	87000	Elec. Eng.	Taylor	85000

- Can use Cartesian product to associate related rows between two tables
  - ... but, a lot of extra rows are included!

instructor × department

instructor_id	name	instructor.dept_name	salary	department.dept_name	building	budget
...	...	...	...	...	...	...
10101	Srinivasan	Comp.Sci.	65000	Comp. Sci.	Taylor	100000
...	...	...	...	...	...	...

- Combine Cartesian product with a select operation
  - $\sigma_{\text{instructor.dept\_name=department.dept\_name}}(\text{instructor} \times \text{department})$
- "Retrieve the names of all instructors associated to departments in the Taylor building"
  - We need both the instructor and department relations
  - Correlate tuples in the relations using dept_name
  - Then, computing result is easy.
  - Associate instructor names with department details, using Cartesian product and a select:
    - $\sigma_{\text{instructor.dept\_name=department.dept\_name}}(\text{instructor} \times \text{department})$
  - Select out departments in the Taylor building
    - $\sigma_{\text{building=Taylor}}(\sigma_{\text{instructor.dept\_name=department.dept\_name}}(\text{instructor} \times \text{department}))$
  - Project result down to instructor name
    - $\Pi_{\text{name}}(\sigma_{\text{building=Taylor}}(\sigma_{\text{instructor.dept\_name=department.dept\_name}}(\text{instructor} \times \text{department})))$
  - Final result:

name
Srinivasan

13.3.7 Rename Operation

- Results of relational operations are unnamed
  - Result has a schema, but the relation itself is unnamed
- Can give result a name using the rename operator
- Written as: $\rho_x(E)$
  - E is an expression that produces a relation
  - E can also be a named relation or a relation-variable
  - x is new name of relation
- More general form is: $\rho_{x(A_1,A_2,\ldots,A_n)}(E)$
  - Allows renaming of relation's attributes
  - Requirement: E has arity n
- Rename operation ρ only applies within a specific relational algebra expression
  - This **does not** create a new relation-variable!
  - The new name is only visible to enclosing relational-algebra expressions
- Rename operator is used for two main purposes:
  - Allow a derived relation and its attributes to be referred to by enclosing relational-algebra operations
  - Allow a base relation to be used multiple ways in one query
    - $r \times \rho_s(r)$
- Rename operation ρ is used to resolve ambiguities within a specific relational algebra expression

13.3.7.1 Examples

- "Find the ID of the instructor with the largest salary"

instructor

instructor_id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

- Hard to find the instructor with largest salary!
- Much easier to find all instructors that have a salary smaller than some other instructor
- Then, use set-difference to find the instructor with largest salary
- Find all instructors with a salary smaller than some other instructor
  - Use Cartesian Product of instructor with itself:
    - instructor $\times$ instructor
  - Compare each instructors's salary to all other instructors
- Problem: Can't distinguish between attributes of left and right instructor relations!
- Solution: Use rename operation
  - instructor $\times$ ρ_{test}(instructor)
  - Now, right relation is named test
- Find IDs of all instructors with a salary smaller than some other instructor:
  - $\Pi_{\text{instructor.instructor\_id}}(\sigma_{\text{instructor.salary<test.salary}}(\text{instructor} \times \rho_{\text{test}}(\text{instructor})))$
- Obtain result
  - $\Pi_{\text{instructor\_id}}(\text{instructor}) - \Pi_{\text{instructor.instructor\_id}}(\sigma_{\text{instructor.salary<test.salary}}(\text{instructor} \times \rho_{\text{test}}(\text{instructor})))$

13.4 Additional Relational Operations

- The fundamental operations are sufficient to query a relational database...
- Can produce some large expressions for common operations!
- Several additional operations, defined in terms of fundamental operations:

Symbol (Name)	Example of Use
$\cap$ (Intersection)	$\Pi_{\text{name}}(\text{instructor}) \cap \Pi_{\text{name}}(\text{student})$ Output the intersection of tuples from the two input relations.
$\bowtie$ (Natural Join)	instructor $\bowtie$ advisor Joins two relations on their common attributes.
$\leftarrow$ (Assignment Operation)	instructor_names $\leftarrow \Pi_{name}(\text{instructor})$ Assigns a relation-value to a relation-variable.

13.4.1 Set-Intersection Operation

- Written as: $r \cap s$
- $r \cap s = r - (r - s)$
  - $r - s$ = the rows in r, but not in s
  - $r - (r - s)$ = the rows in both r and s
- Relations must have compatible schemas

13.4.1.1 Examples

- "Find all departments with both associated instructors and associated students

- $\Pi_{\text{dept\_name}}(\text{instructor}) \cap \Pi_{\text{dept\_name}}(\text{student})$

13.4.2 Natural Join Operation

- Most common use of Cartesian product is to correlate tuples with same key-values
  - Called a join operation
- The **natural join** is a shorthand for this operation
- Written as: $r \bowtie s$
  - r and s must have common attributes
  - The common attributes are usually a key for r and/or s, but certainly don't have to be

13.4.2.1 Definition

- For two relations $r(R)$ and $s(S)$
- Attributes used to perform natural join:
  - $R \cap S = \{A_1, A_2, \ldots, A_n\}$
- Formal definition:
  - $r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s))$
  - r and s are joined on their common attributes
  - Result is projected so that common attributes only appear once

13.4.2.2 Examples

- "Find the names of all instructors that advise students."
- Result:
  - $\Pi_{\text{name}}(\sigma_{\text{instructor.instructor\_id=advisor.instructor\_id}} (\text{instructor} \times \text{advisor}))$
- Rewritten with natural join:
  - $\Pi_{\text{name}}(\text{instructor} \bowtie \text{advisor})$
- Very common to compute joins across multiple tables
- Example: instructor $\bowtie$ advisor $\bowtie$ student
- Natural join operation is associative:
  - (instructor $\bowtie$ advisor) $\bowtie$ student is equivalent to instructor $\bowtie$ (advisor $\bowtie$ student)
- Note:
  - Even though these expressions are equivalent, order of join operations can dramatically affect query cost!

13.4.3 Assignment Operation

- **Recall**: relation variables refer to a specific relation
  - A specific set of tuples, with a particular schema
- Example: student relation!
  - student is actually technically a relation-variable, as are all our named relations so far

student

student_id	name	dept_name	tot_cred	student_ssn
12856	Zhang	Comp. Sci.	102	3688 290383
12345	Shankar	Comp. Sci.	32	3134 181283
19991	Brandt	History	80	3500 180584

- Can assign a relation-value to a relation-variable

- Written as: relvar $\leftarrow E$
  - E is an expression that evaluates to a relation
- Unlike ρ, the name relvar persists in the database
- Often used for temporary relation-variables:
  - temp1 $\leftarrow \Pi_{R-S}(r)$
  - temp2 $\leftarrow \Pi_{R-S}((\text{temp1} \times s) - \Pi_{R\tilde{}S,S}(r))$
  - result $\leftarrow$ temp1 $-$ temp2
  - Query evaluation becomes a sequence of steps
  - (This is an implementation of the $\div$ operator)
- Can also use to represent data updates

14 Codds Rules for Relational DBMSs

- Information Rule
  One representation for all information
  Values in tables
- Guaranteed Access Rule
  Access to every value using table name, column name, and value of primary key
- Null Value Rule
  Missing information is represented by NULL
- Online Catalogue Rule
  Dynamic online data dictionary
- Comprehensive Data Sublanguage Rule
  Language support for all concepts of the RDM
- View Updating Rule
  Where logically possible, support for update of base tables through views
- High Level Insert, Update, and Delete Rule
  Unlimited number of tuples in single operation
- Physical Data Independence Rule
  Applications isolated of storage structure and location
- Logical Data Independence Rule
  Applications isolated against information preserving changes of the conceptual schema
- Integrity Independence Rule
  Support of definition of integrity constraints and their administration in the data dictionary
- Distribution Independence Rule
  Logical and physical data independence in distributed databases
- Non-Subversion Rule
  No by-passing of integrity constraints

15 Acknowledgement

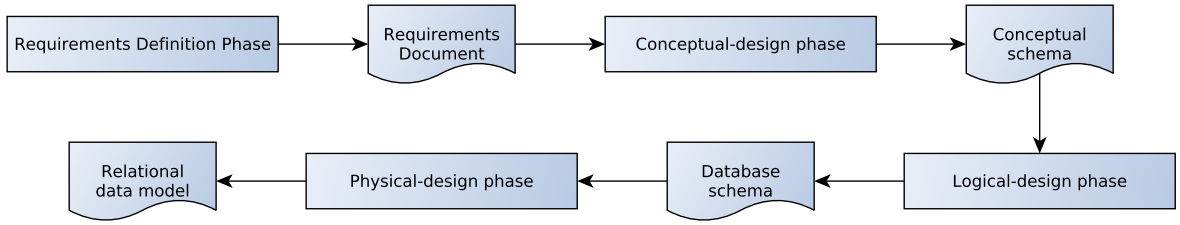
This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

16 Designing Database Applications

- Database application are large and complex
- Some of the design areas:
  - Database schema (physical/logical/view)
  - Programs that access and update data
  - Security constraints for data access
- Requires familiarity with the problem domain
  - Domain experts **must** help drive requirements

16.1 Overview of the Design Process



- Requirements definition phase
  - Collect user requirements
    - Information that needs to be represented
    - Operations to perform on that information
    - Several techniques for representing this info (**requirements document**), i.e. textual description, UML,...
- Conceptual-design phase
  - The **conceptual schema** describes the information structure of an application, which is not subject to change as the application evolves.
    - A high-level representation of the database's structure and constraints
    - Physical and logical design issues are ignored at this stage
    - Often represented graphically
  - Includes specification of **functional requirements**
    - What operations will be performed against the data?
    - Can be used to verify the conceptual schema
      - Are all operations possible?
      - How complicated is it?
- Logical-design phase
  - Convert conceptual schema into an implementation data model, e.g. E-R schema
- Physical-design phase
  - Implement database model including possible additional design an tuning decisions (e.g. indexes, disk-level partitioning of data)

16.2 Design Decisions

- Different ways to represent data
- Two major problems:
  - Unnecessary redundancy
    - Wastes space
    - Greater potential for inconsistency!

        * Ideally: each fact appears in exactly one place
    - Incomplete representation
        * Schema must be able to fully represent all details and relationships required by the application
- Other concers:
    - How easy/hard is it to access useful information? (e.g. reporting or summary info)
    - How hard is it to update the system?
    - Performance considerations?
    - Scalability considerations?
- Balance between aesthetic and practical concerns

17 The Entity-Relationship Model

- Developed to facilitate database design by allowing specification of an enterprise schema that represents the overall **logical structure** of a database.
- Useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema.
- Employs three basic concepts:
    - entity sets,
    - relationship sets,
    - attributes.
- The E-R diagram expesses the overall logical structure of a database graphically.

17.1 Entity-Sets

- An **entity** is an object that exists and is distinguishable from other objects.
- An **entity-type** is a common abstraction of a group of entities which have similar attributes.
- An **entity-set** is a set of entities of the same type that share the same properties.
- An entity is represented by a set of **attributes**; i.e., descriptive properties possessed by all members of an entity set.
- Each entity has a **value** for each of its attributes.
- A subset of the attributes form a **primary key** of the entity-set; i.e., uniquely identifiying each member of the set.
- Entity-sets do not need to be disjoint.

17.2 Relationship-Sets

- A **relationship** is an association among several entities.
- A **relationship-type** is a common abstraction of a group of similar relationships.
- A **relationship-set** is a set of relationships of the same type. Formally, it is a mathematical relation among $n \geq 2$ (possibly nondistinct) entity sets

$$\{(e_1, e_2, \ldots, e_n) | e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

  where $(e_1, e_2, \ldots, e_n)$ is a relationship.
- The association between the entity-sets is reffered to as **participation**.
- A **relationship instance** represents an association between named entities int the real-world.

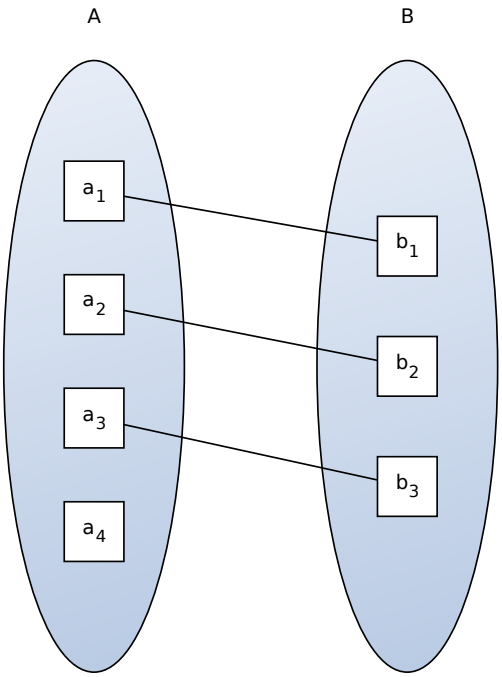
- The function an entity plays in a relationship is called that entity's **role**.
  - In general the participating entity-sets in a relationship-set are **distinct**, thus roles are implicit and are not usually specified.
  - When a entity-set participates in a relationship-set more than once, in different roles, the roles are explicitly specified for clarification. E.g. in a **recursive** relationship-set, explicit role names are necessary to specify how an entity participates in a relationship instance.
- A relationship may also have attributes called **descriptive attributes**.
  - Each relationship instance must be uniquely identifiable from its participating entities, without the descriptive attributes.

17.3 Degree of a Relationship-Set

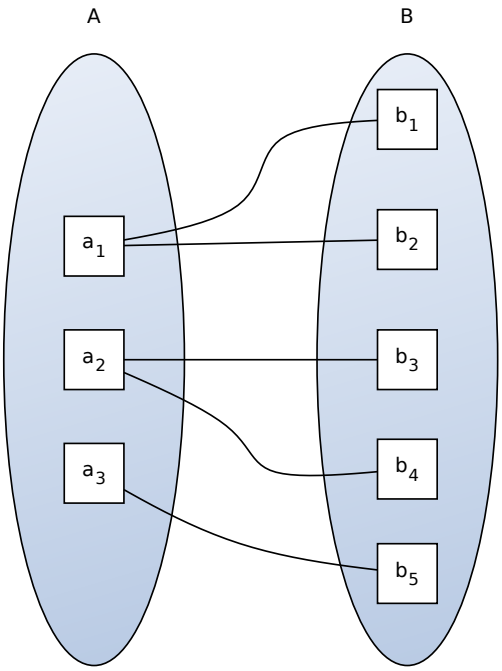
- Binary relationship-sets ($n = 2$)
  - involve two entity-sets
  - are the most common relationship-sets in a database system.
- n-ary relationship-sets ($n \geq 3$)
  - are rare in a database system, but occasionally present.

17.4 Mapping Cardinality (Cardinality Ratios)

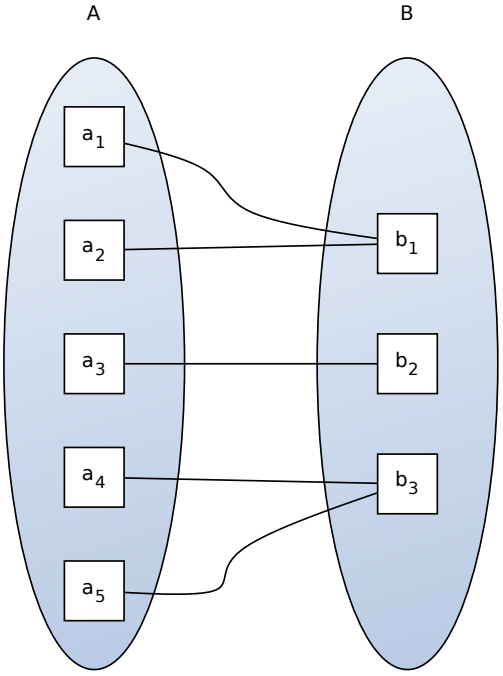
- express the number of entities to which another entity can be associated via a relationship-set.
- most useful in describing binary relationship-sets.
- For a binary relationship-set the mapping cardinality must be one of the following types:
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many



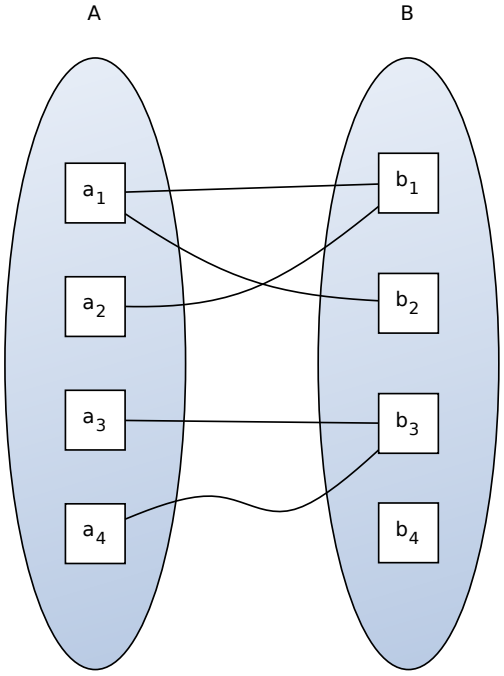
One-to-one



One-to-many



Many-to-one



Many-to-many

17.5 Participation Constraints

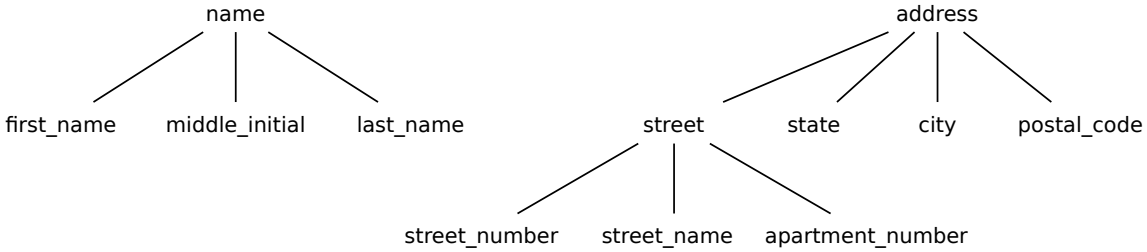
- The participation of an entity-set E in a relationship R is said to be **total** if every entity participates **at least once** in R.
- If **only some** entities in E participate in relationships in R the participation of E in R is said to be **partial**.

17.6 Attributes

- For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute.
- Formally, an attribute is a function that maps from the entity into a domain.
- Attribute values describing an entity constitute a significant part of the data stored in a database.

Attributes can be characterized by the folling attribute types:

- **Simple** and **composite** attributes:
  - **Simple** attributes can not be divided into subparts
  - **Composite** attributes, on the other hand, can be divied into subparts (other attributes); i.e., the attribute *name* could be structures as attributes *first_name*, *middle_initial*, and *last_name*.



- **Single-valued** and **multivalued** attributes:
  - **Single-valued** each entity can only have on single value for the attribute.
  - **Multivalued** each entity may be have a *set* of values for the attribute; i.e., *phone_number* a person may have zero, one, or several phone numbers.
- **Derived** attributes:
  - The value for this type of attributes can be derived from the values of other related attributes; i.e., *age* can be derived from the attribute *date_of_birth* and the current date.
  - Are not stored in the database
- **Null value**
  An attribute takes **null** when an entity doesn't have a value for it. May indicate:
  - "not applicable" or
  - "unknown" ("missing", "not known")

17.7 Keys

- provide a way to how entities are distinguished within a given entity-set.
- Conceptually, individual entities are distinct; from a database perspective their differences must be expressed in terms of their attributes.

- The values of the attribute values of an entity must be such that they can *uniquely identify* the entity.
- The **key** for an entity is a set of attributes that suffice to uniquely identify the entity.
- Keys are a property of the entity-set
  - They apply to all entities in the entity-set

17.7.1 Types of Keys

- **Superkey**: a set of one or more attributes that can **uniquely identify** an entity
- **Minimal key**: a superkey is minimal, if every subset of its attributes isn't a superkey anymore
- **Candidate key**: a minimal superkey
- **Primary key**: a candidate key chosen by DB designer as the primary means of accessing entities
- **Foreign key**: a set of columns in a relational schema for which the values reference uniquely a row in a different schema

17.7.2 Choosing Candidate Keys

- Candidate keys constrain the values of the key attributes
  - No two entities can have the same values for those attributes
  - Need to ensure that database can actually represent all expected circumstances
- Bad design example:
  - Using customer name as a candidate key, different customers can have the same name

17.7.3 Choosing Primary Keys

- An entity-set may have multiple candidate keys
- The primary key is the candidate key most often used to reference entities in the set
  - In logical/physical design, primary key values will be used to represent relationships
  - External systems may also use primary key values to reference entities in the database
- **The primary key attributes should never change!**
  - If ever, it should be extremely rare.

17.7.4 Choosing Keys: Performance

- Large, complicated, or multiple-attribute keys are generally slower
  - Use smaller, single-attribute keys
    - (You can always generate them...)
  - Use faster, fixed-size types
    - e.g. INT or BIGINT
- Especially true for primary keys!
  - Values used in both database and in access code
  - Use something small and simple, if possible

17.8 Weak Entity-Sets

- An entity-set that doesn't have sufficient attributes to form a primary key is termed a **weak entity-set**.
- Every weak entity-set must be associated with an **identifying** or **owner entity-set**. The weak entity-set is **existence dependent** on the identifying set.
- The **identifying relationship** is *many-to-one* from the weak entity set to the owner, and the participation from the weak entity set is *total*.
- The **identifying relationship** shouldn't have any descriptive attributes.
- The **discriminator** or **partial key** of a weak entity set is used to uniquely identify the weak entities associated with a single owner entity.
- The **primary key** of a weak entity-set is formed by the primary key of the owner set, plus the weak enity-set's discriminator.

17.8.1 Modeling Considerations

A weak entity-set

- can participate in relationships other than the identifying relationship.
- may participate as owner in an identifying relationship.
- may have more than one identifying entity-set.
- can be expressed as a multivalued, composite attribute, if it only participates in the identifying relationship.

18 Entity-Relationship Diagrams

- A diagrammatic representation of the data model is a very important part of designing a database schema.
  - It eases communication between modeling experts, domain experts and developers.
- **There is no universal standard for E-R diagram notation!!!**
- There is a plethora of alternative notations for E-R diagrams
  - E.g. Chen, IDEF1X, Bachman, Marint/ IE/ Crow's Foot, Min-Max/ ISO, UML, ...
  - Different styles for entities, relationships and attributes
  - Also, notations are often freely mixed
  - ER diagrams can look completely different depending on the used tool / book

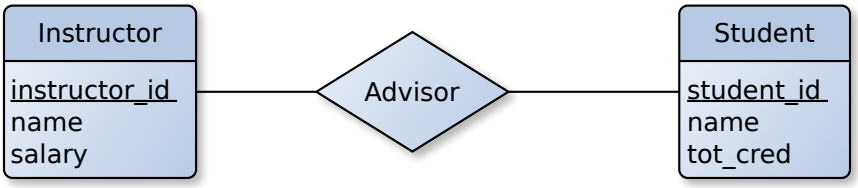
18.1 Basic Structure

18.1.1 Entity Sets

- Rectangles divided into two parts:
  - Represent entity sets.
  - The first part contains the name of the entity set.
  - The second part contains the names of all the attributes.
  - Attributes part of the primary key are underlined.

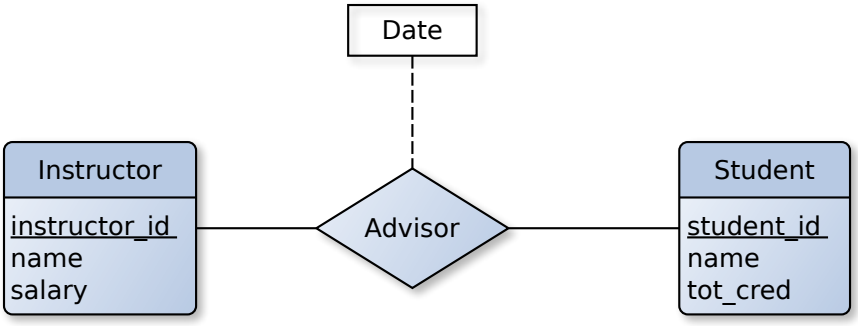
18.1.2 Relationship Sets

- Diamonds
  - Represent relationship sets.
- Lines
  - Link entity sets to relationship sets.



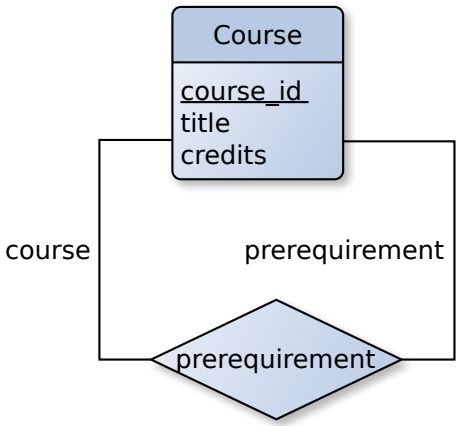
18.1.3 Relationship Sets with Attributes

- Undivided rectangles
  - Represent the attributes of a relationship set.
- Dashed lines
  - Link attributes of a relationship set to a relationship set.



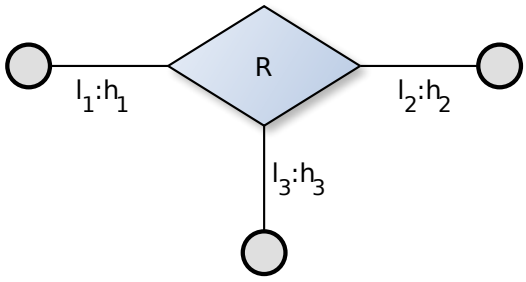
18.1.4 Roles

- Entity sets of a relationship need not be distinct
  - Each occurrence of an entity set plays a “role” in the relationship
- The labels “course” and “prerequirement” are called roles.



18.1.5 Cardinality Constraints

- Constraint for the number of associations of the current relationship set, which may be connected at any point in time with an entity of the entity set.
- Represented by cardinality intervals shown in the form $l : h$, where l is the minimum and h the maximum cardinality.
  - A minimum value of ≥ 1 indicates **total** participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship.
  - A maximim value of $*$ indicates no limit.
  - $l = 0, 1, 2, \ldots, *$
  - $h = 1, 2, 3, \ldots, *$
  - $l \leq h$

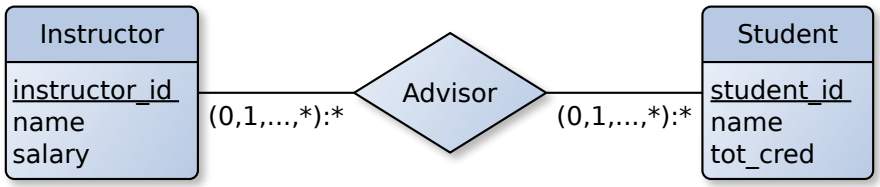


- One-to-one relationship between *instructor* and a *student*
  - An instructor is associated with at most one student via the relationship advisor,
  - and a student is associated with at most one instructor via the relationship advisor.

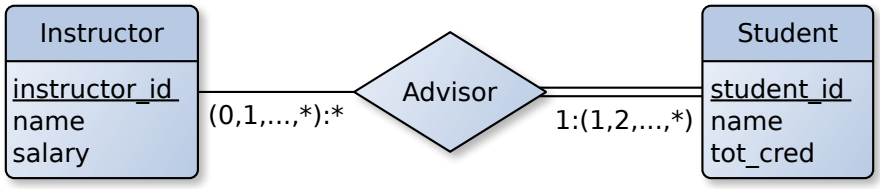
- One-to-many between *instructor* and a *student*
  - An instructor is associated with several (including 0) students via advisor,
  - and a student is associated with at most one instructor via advisor.

- Many-to-one
  - An instructor is associated with at most one student via advisor,
  - and a student is associated with several (including 0) instructors via advisor.

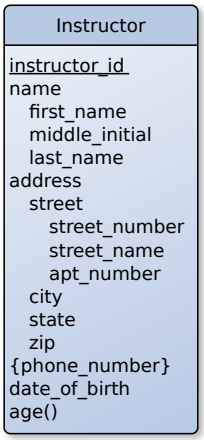
- Many-to-many between *instructor* and a *student*
  - An instructor is associated with several (including 0) students via advisor,
  - and a student is associated with several (including 0) instructors via advisor.



- Total participation *instructor* and a *student*
  - A student is associated with at least one instructor via advisor.



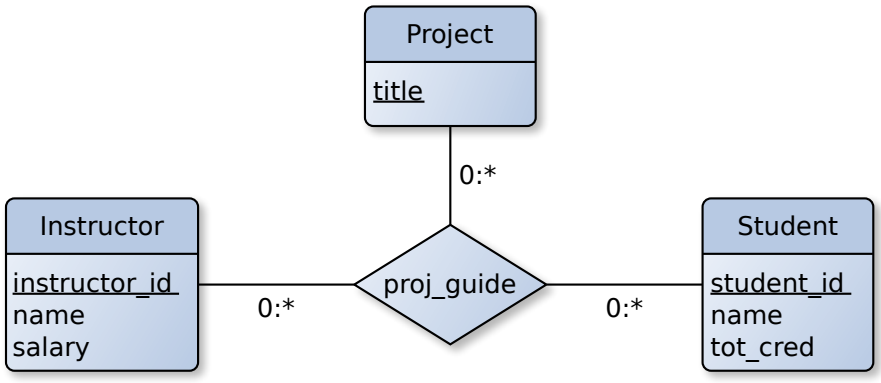
18.1.6 Complex Attributes



- The example shows how composite attributes can be represented in the E-R notation.
  - *name* is a composite attribute, with components *first_name, middle_initial*, and *last_name*
  - *address* is a composite attribute, with components *street, city, state*, and *zip_code*
    - *street* itself is a composite attribute, with components *street_number, street_name*, and *apt_number*
  - *phone_number* is a multivalued attribute, each instructor may have more than one phone number
  - *age* is a derived attribute from the attribute *date_of_birth*

18.1.7 n-ary Relationship Sets

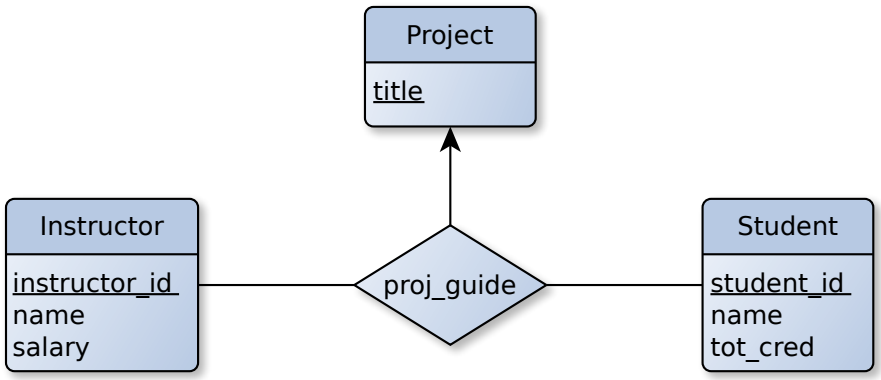
- Can specify relationships of degree > 2 in E-R model



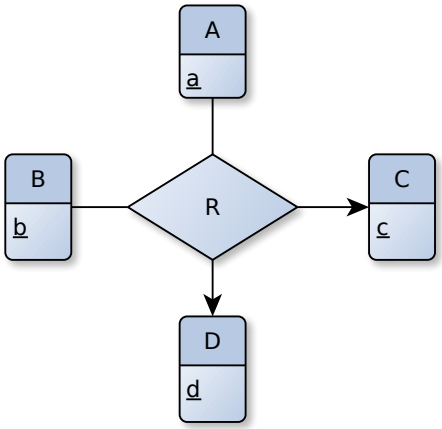
- Example
- Instructors are assigned to students in various projects
- Many-to-many mapping: any combination of instructor, student, and project is allowed
- A student may have several instructors in one project

18.1.7.1 n-ary Mapping Cardinalities

- Can specify some mapping cardinalities on relationships with degree > 2



- Example
  - Each combination of instructor and student can only be associated with **one** project
  - Each instructor can guide only one project for each student
- For degree > 2 relationships, we only allow at most one edge with an arrow
  - Reason: multiple arrows on N-ary relationship-set is ambiguous
    - (several meanings have been defined for this in the past)
  - Relationship-set R associating entity-sets $A_1, A_2, \ldots, A_n$
    - No arrows on edges $A_1, \ldots, A_i$
    - Arrows are on edges to $A_{i+1}, \ldots, A_n$
  - Meaning 1 (the simpler one):
    - A particular combination of entities in $A_1, \ldots, A_i$ can be associated with at most **one** set of entities in $A_{i+1}, \ldots, A_n$
    - Primary key of R is union of primary keys from set $\{A_1, A_2, \ldots, A_i\}$
  - Meaning 2:
    - **For each** entity-set A_k $(i < k \leq n)$, a particular combination of entities from all other entity-sets can be associated with at most one entity in A_k
    - R has a candidate key for each arrow in n-ary relationship-set
    - **For each** k $(i < k \leq n)$, another candidate key of R is the union of primary keys from entity-sets $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$
  - Both interpretations of multiple arrows have been used in books and papers...
  - If we only allow one edge to have an arrow, both definitions are equivalent
    - The ambiguity disappears



Meaning 1:
Primary key: (a,b)
Examples:
(a_1, b_2, c_3, d_5)
(a_3, b_4, c_3, d_2)
(a_3, b_4, c_3, d_2)
...

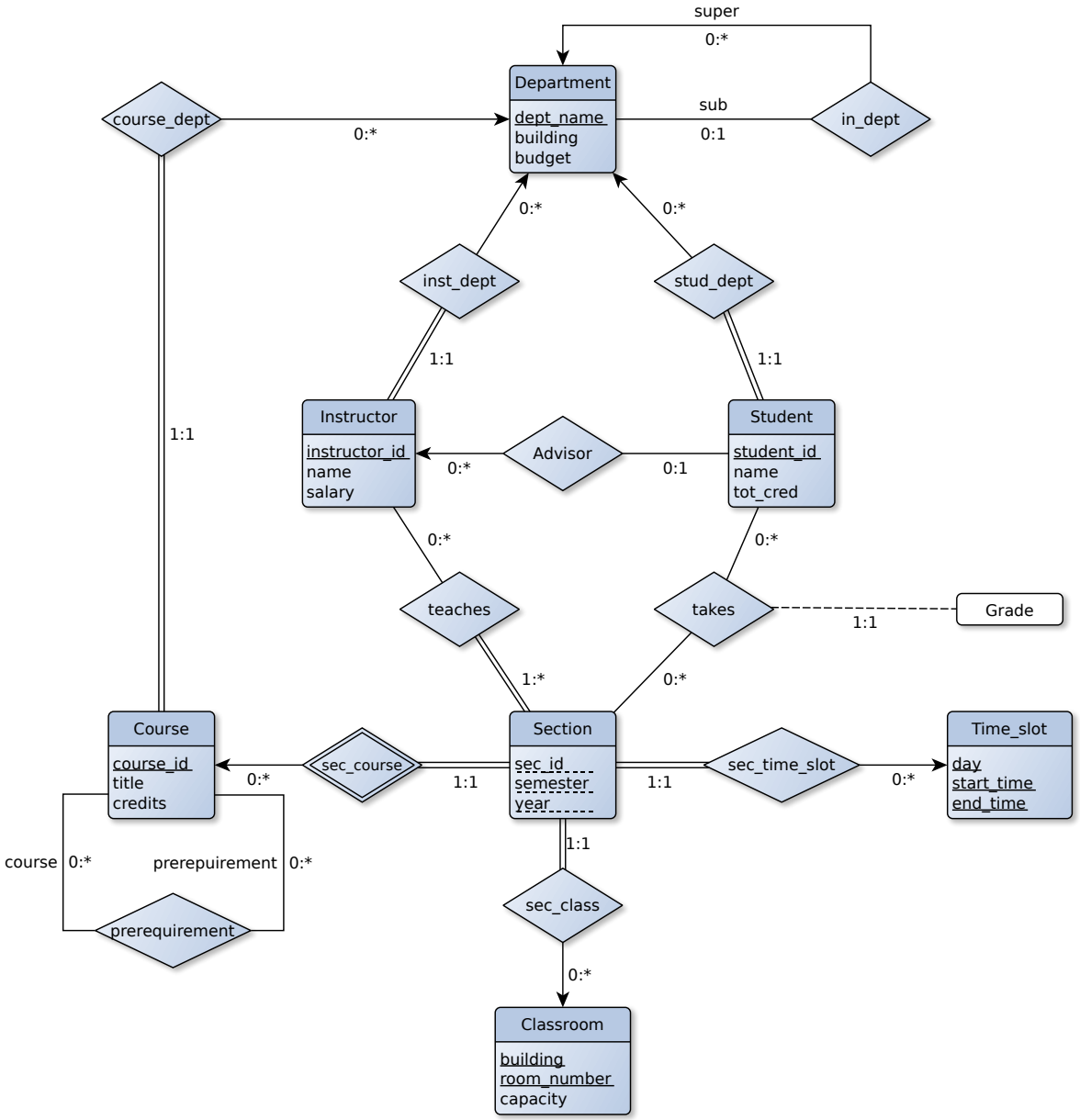
Meaning 2:
Candidate keys: (a,b,c), (a,b,d)
Examples:
(a_1, b_2, c_3, d_5)
(a_3, b_4, c_3, d_2)
$(a_1, b_2, c_1, d_4)^*$
$(a_3, b_4, c_5, d_7)^*$
$(a_1, b_2, c_3, d_6)^*$
$(a_3, b_4, c_8, d_2)^*$
...
all * dissallowed by meaning 1!

18.1.8 Weak Entity-Sets

- Double diamonds
  - Represent identifying relationships linked to weak entity sets.
  - The discriminator of a weak entity set is underlined with a dashed line.
  - The relationship between the weak entity and it's owner entity sets is always a total many-to-one relationship.

- In the example the weak entity set *section* depends on the strong entity set *course* via the identifying relationship *sec_course*.

18.2 Example E-R Diagram

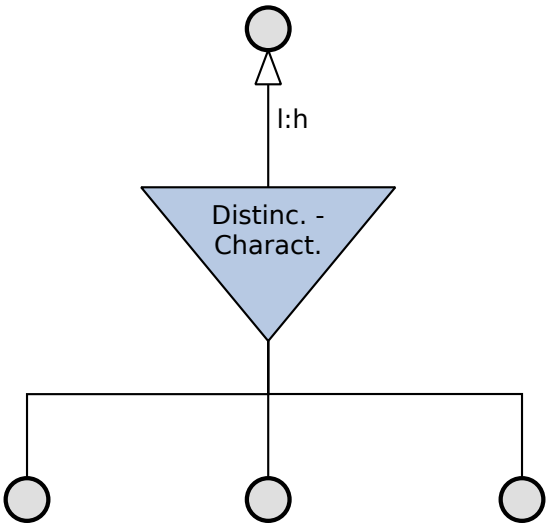


19 Extended E-R Features

19.1 Specialization

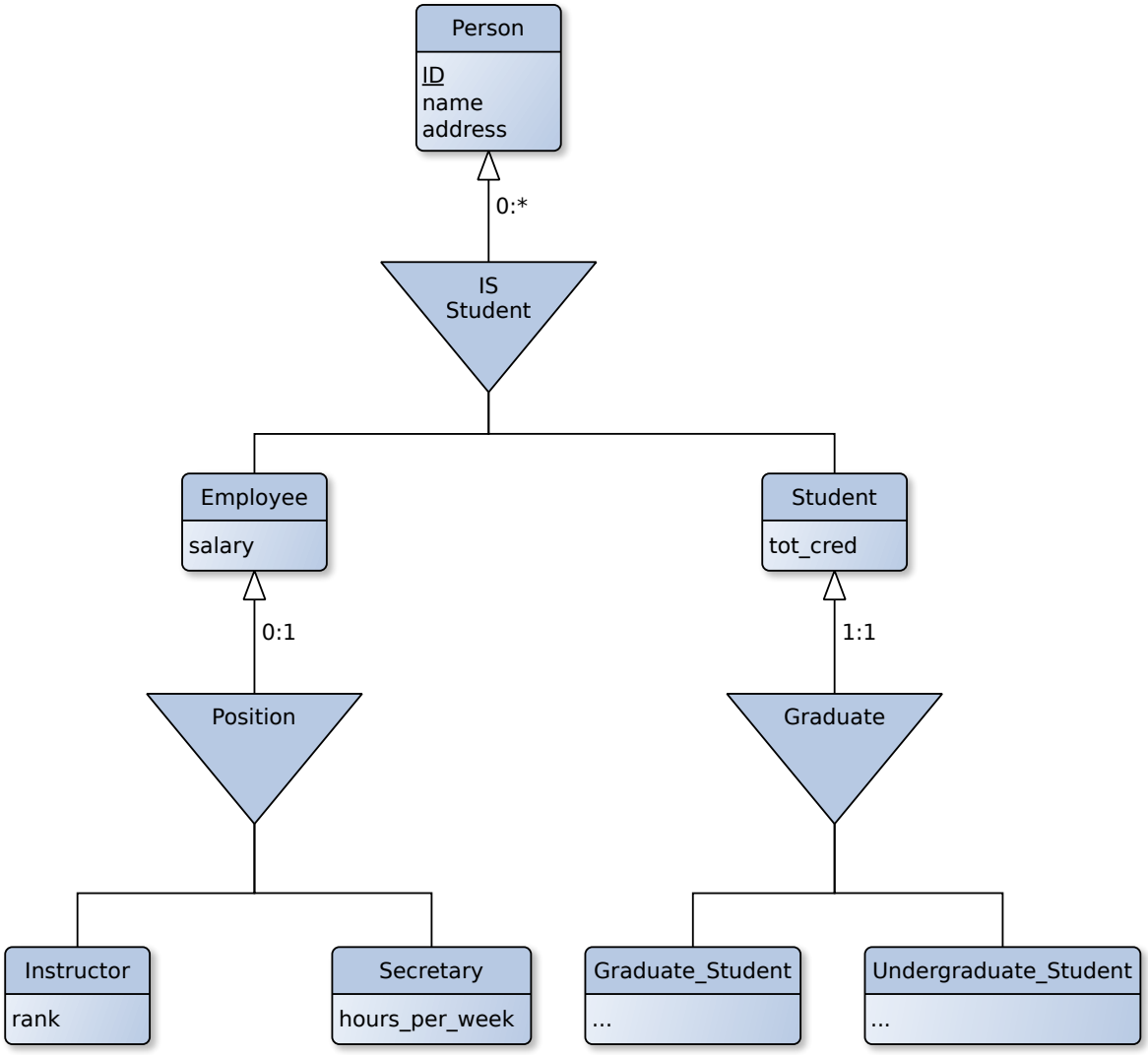
- **Top-down** design process: refine from an initial entity set into successive lower-level entity sets.
- An entity set may include subgroupings of entities that are **distinct** from other entities in the set.
- These subgroupings (**subtypes**) become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set (**supertype**).
- **Attribute inheritance**: A subtype inherits all the attributes and relationship participation of the supertype to which it is linked.

- Represented by a *triangle* component labeled with the distinguishing characteristic.
  - The cardinality interval $l : h$: represents different constraints on generalizations
    - Lower limit l: 0 ... partial; 1 ... total
    - Upper limit h: 1 ... disjoint; * ... overlapping

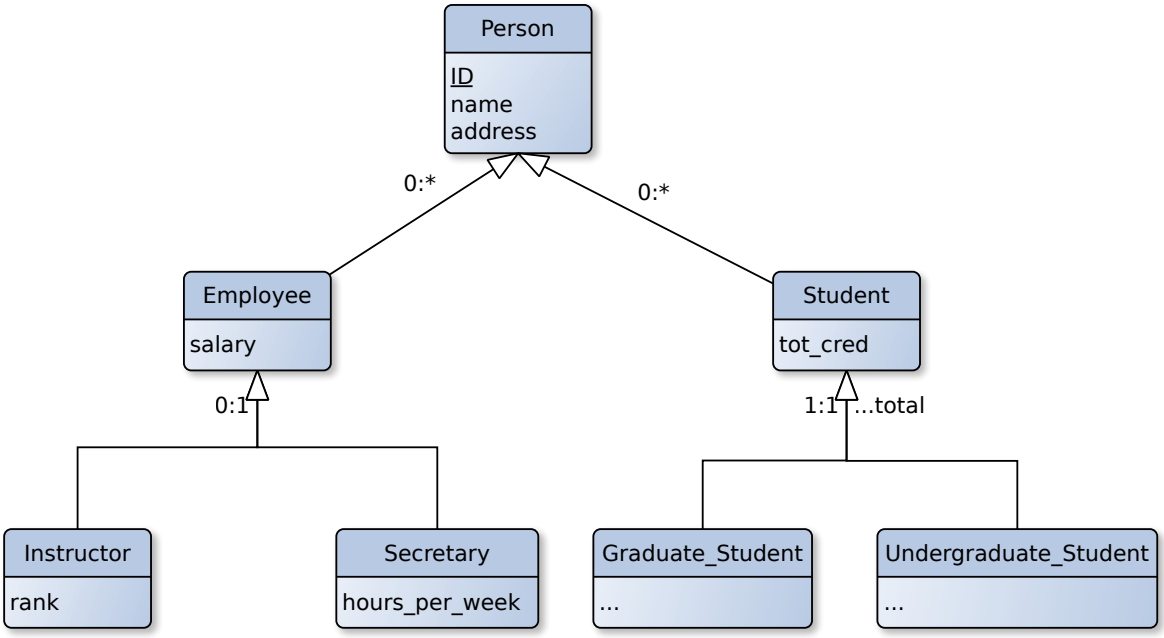


19.1.1 Example

- The entity set *person* can be classified as either *employee* or *student*. In general, a person could be an employee, a student, both, or neither.
- The entity set *employee* could be further classified as *instructor* or *secretary*. In the example an employee could be an instuctur, a secretary, or neither.
- The entity set *student* could be further classified as *graduate* or *undergraduate*. In the example a student must be a graduate student or an undergraduate student.



19.1.1.1 Alternative Representation



19.2 Generalization

- **Bottom-up** design process: synthesize multiple entity sets into a higher-level entity set on the basis of common features.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- Generalization is used to emphasize the similarities among lower-level entity sets
- Permits an economy of representation, doesn't repeat shared attributes

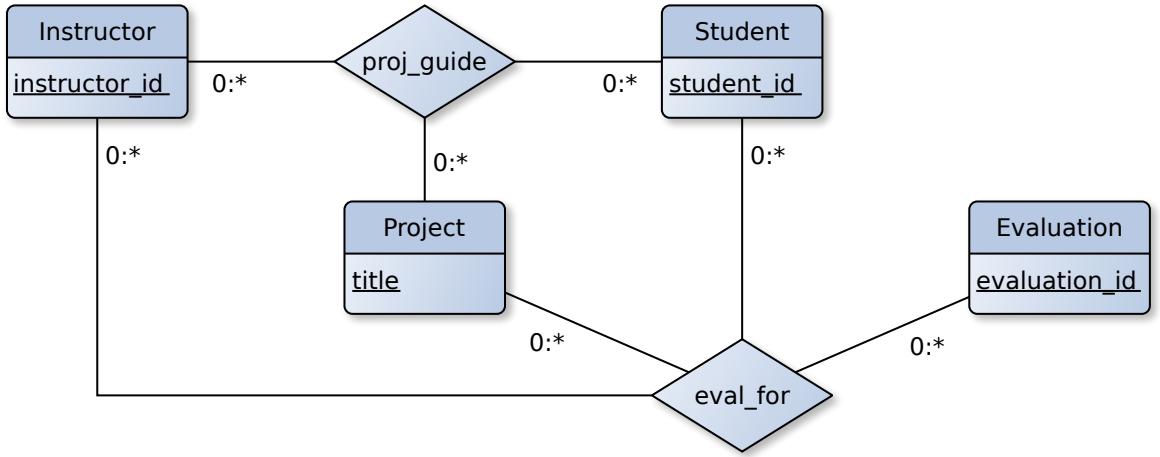
19.3 Constraints on Generalizations

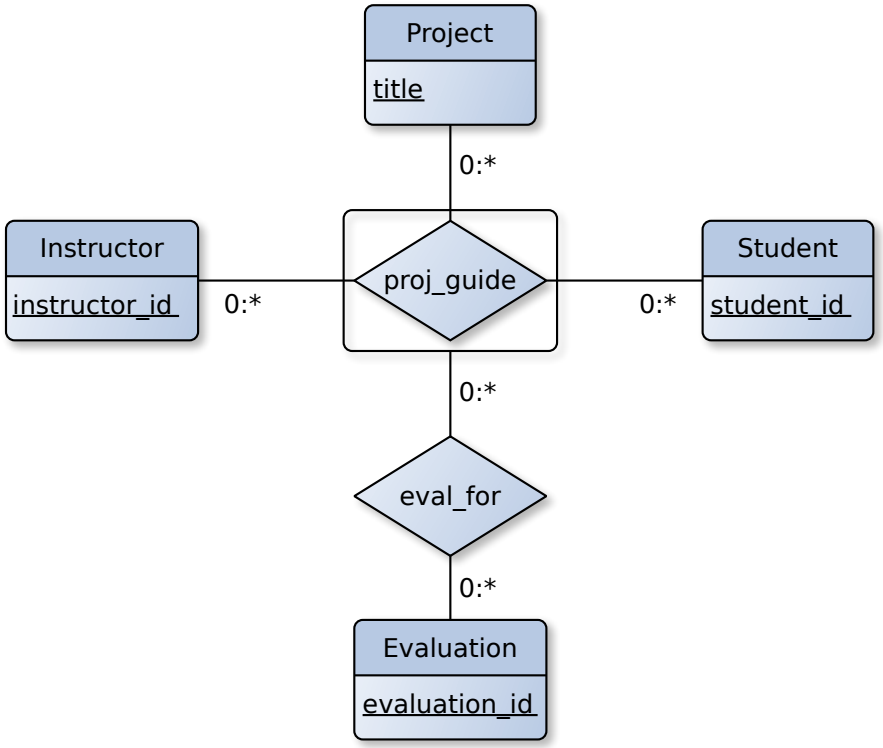
- Constraint on which entities can be members of a given lower-level entity set.
  - **Attribute-defined (condition-defined)**: Membership is evaluated on the basis whether or not an entity satisfies an explicit condidtion.
    I.e. The supertype *student* has the attribute *student_type*. And only those students where the condition *student_type = "graduate"* belong to the subtype *graduate_student*.
  - **User-defined**: The **database user** assigns entities to a given entity set.
    I.e.
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**: An entity can belong to only one subtype.
    I.e. An employee entity can be either an instructor, a secretary or neither but not both.
  - **Overlapping**: An entity can belong to more than one subtype.
    I.e. A person entity can be either an employee, a student, both or neither.
- Completeness constraint: specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **Total**: An entity must belong to one of the subtypes.
    I.e. All student entities must be either graduate or undergraduate students.

– **Partial**: An entity need not belong to one of the subtypes.
I.e. Employee entities need not belong to the subtypes instructor or secretary. They also may have unspecified roles.

19.4 Aggregation

- E-R model limitation: it cannot express **relationships among relationships**.
- Solution: **Aggregation**, an abstraction which treats relationships as higher-level entities (**hybrid**).





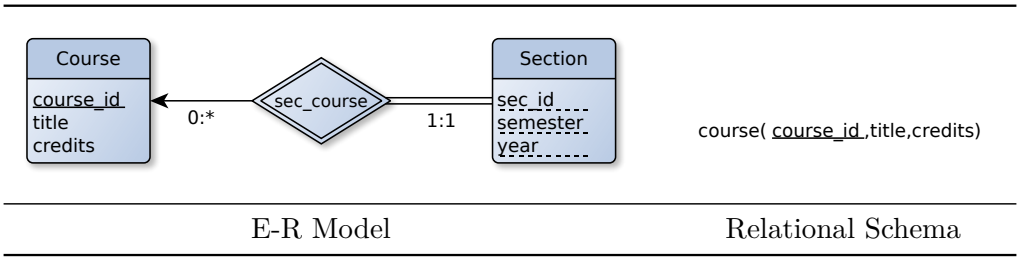
20 Reduction to Relational Schemas

- For E-R model to be useful, need to be able to convert diagrams into an implementation schema
- Turns out to be very easy to do this!
  - Big overlaps between E-R model and relational model
  - Biggest difference is E-R composite/multivalued attributes, vs. relational model atomic attributes
- Three components of conversion process:
  - Specify schema of the relation itself
  - Specify primary key on the relation
  - Specify any foreign key references to other relations

20.1 Representation of Strong Entity Sets with Simple Attributes

- Strong entity-set E with attributes $a_1, a_2, \ldots, a_n$
- Create a relation schema with same name E, and same attributes $a_1, a_2, \ldots, a_n$
- Primary key of relation schema is same as primary key of entity-set
  - Strong entity-sets require no foreign keys to other things
- Every entity in E is represented by a tuple in the corresponding relation

20.1.1 Example:



- A strong entity set reduces to a schema with the same attributes

20.2 Representation of Strong Entity Sets with Complex Attributes

- Relational model simply doesn't handle **composite** and **multivalued** attributes
  - All attribute domains are atomic in the relational model
- When mapping E-R **composite** attributes to relation schema: simply flatten the composite
  - Each component attribute maps to a separate attribute in relation schema
  - In relation schema, simply can't refer to the composite as a whole
  - (Can adjust this mapping for databases that support composite types)
- When mapping E-R **multivalued** attributes a separate relation is needed
  - E-R constraint on multivalued attributes: in a specific entity's multivalued attribute, each value may only appear **once**
  - For a multivalued attribute M in entity-set E
    - Create a relation schema R to store M, with attribute(s) A corresponding to the single-valued version of M
    - Attributes of R are: $primary\_key(E) \cup A$
    - Primary key of R includes all attributes of R

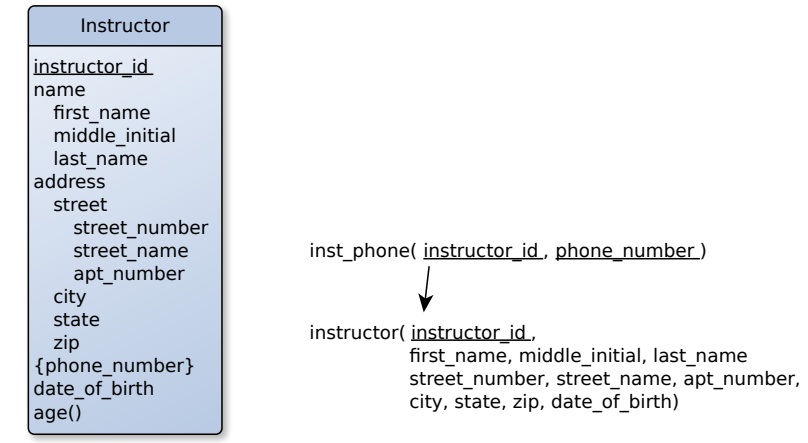
· Each value in M for an entity e must be unique
∗ Foreign key from R to E, on *primary_key*(E) attributes

20.2.1 Example: Composite attributes (ignoring multivalued attributes):

- Given entity set Instructor with composite attribute *name* with components *first_name*, *middle_initial*, and *last_name*, the corresponding schema has three attributes *name_first_name*, *name_middle_initial*, and *name_last_name*
  - Prefix omitted if there is no ambiguity

20.2.2 Example: Multivalued Attribute

- The multivalued attribute *phone_number* of instructor is represented by a separate schema *inst_phone*
- Each value of the multivalued attribute maps to a separate tuple on schema *inst_phone*

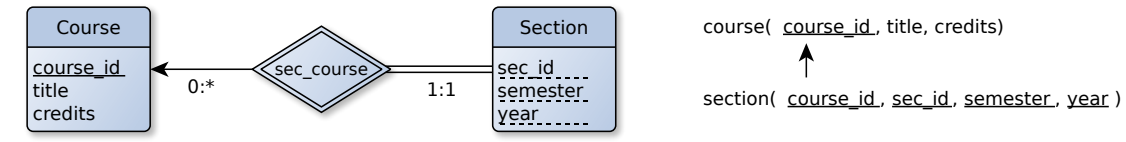


E-R Model Relational Schema

20.3 Representation of Weak Entity Sets

- Weak entity-sets depend on at least one strong entity-set
  - The identifying entity-set, or owner entity-set
  - Relationship between the two is called the identifying relationship
- Weak entity-set A owned by strong entity-set B
  - Attributes of A are $\{a_1, a_2, \ldots, a_m\}$
    - Some subset of these attributes comprises the discriminator of A
  - *primary_key*(B) = $\{b_1, b_2, \ldots, b_n\}$
  - Relation schema for A: $\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$
  - Primary key of A is *discriminator*(A) ∪ *primary_key*(B)
  - A has a foreign key constraint on *primary_key*(B), to B
- The identifying relationship is **many-to-one**, with no descriptive attributes
- Relation schema for weak entity-set already includes primary key for strong entity-set
  - Foreign key constraint is imposed, too
- No need to create relational model schema for the identifying relationship
- Would be redundant to the weak entity-set's relational model schema!

20.3.1 Example:



E-R Model | Relational Schema

- Strong entity set: Course
- Weak entity set: Section
  - Disriminator is sec_id, semester, year
  - Primary key: course_id, sec_id, semester, year
  - Foreign key constraint on course_id

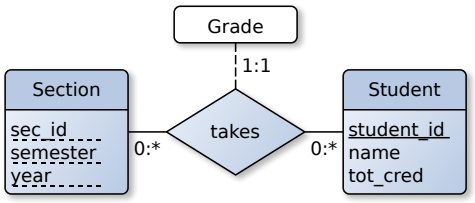
20.4 Representation of Relationship Sets

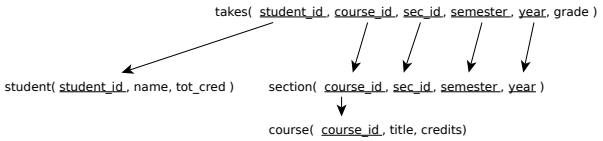
- Relationship-set R
  - For now, assume that all participating entity-sets are strong entity-sets
  - $a_1, a_2, \ldots, a_m$ is the union of all participating entity-sets' primary key attributes
  - $b_1, b_2, \ldots, b_n$ are descriptive attributes on R (if any)
- Relational model schema for R is:
  - $\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$
- $\{a_1, a_2, \ldots, a_m\}$ is a superkey, but not necessarily a candidate key
  - Primary key of R depends on R's mapping cardinality

20.4.1 Binary Relationship Sets Primary Keys

- e.g. between strong entity-sets A and B
- If **many-to-many** mapping:
  - Primary key of relationship-set is union of all entity-set primary keys
  - $primary\_key(A) \cup primary\_key(B)$
- If **one-to-one** mapping:
  - Either entity-set's primary key is acceptable
  - $primary\_key(A)$ or $primary\_key(B)$
  - Enforce **both** candidate keys in DB schema!
- If **many-to-one** or **one-to-many** mappings:
  - Primary key of entity-set on "many" side is primary key of relationship
  - Example: relationship R between A and B
    - One-to-many mapping, with B on "many" side
    - Schema contains $primary\_key(A) \cup primary\_key(B)$, plus any descriptive attributes on R
  - $primary\_key(B)$ is primary key of R
    - Each $a \in A$ can map to many $b \in B$
    - Each value for $primary\_key(B)$ can appear only once in R

20.4.2 Example: many-to-many mapping with descriptive attribute

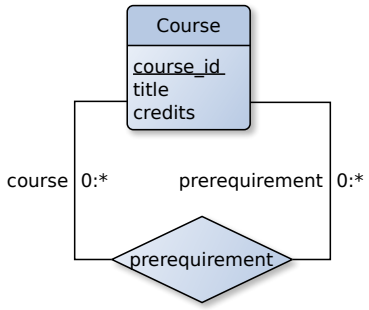


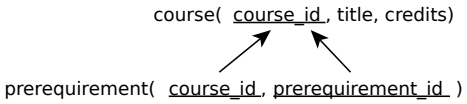


E-R Model | Relational Schema

- Relation schema for takes:
  - Primary key of Student is student_id
  - Primary key of Section is course_id, sec_id, semester, year
  - Descriptive attribute grade
  - Mapping cardinality is many-to-many

20.4.3 Example: Relationship sets with required roles



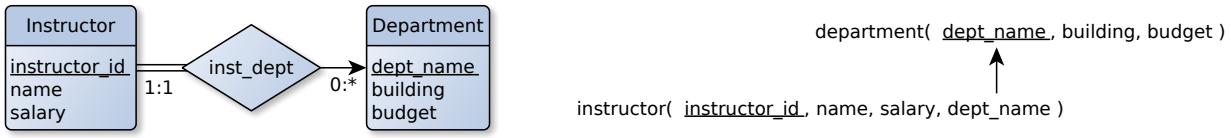


E-R Model | Relational Schema

- In cases like this, roles must be used to distinguish between the entities involved in the relationship-set
  - Course participates in prerequirement relationship-set twice
  - Can't create a schema which uses course_id twice!
- Change names of key attributes to distinguish roles!
- Relation schema for prerequirement
  - Many-to-many mapping from course to prerequirement

20.4.4 Example: Redundancy of Schemas (many-to-one, one-to-many and one-to-one relationship-sets)

- **Many-to-one**, **one-to-many** and **one-to-one** relationship-sets can be represented by adding an extra attribute to the "many" side, containing the primary key of the "one" side
- For one-to-one relationship-sets, either side can be chosen to act as the "many" side
  - Extra attribute can be added to either of the tables corresponding to the two entity-sets
- If participation is partial on the "many" side, replacing a schema by an extra attribute in the schema corresponding to the "many" side could result in **null** values



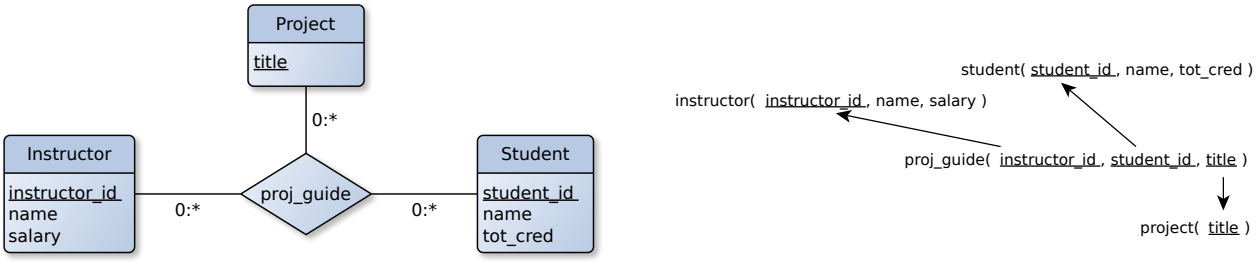
E-R Model | Relational Schema

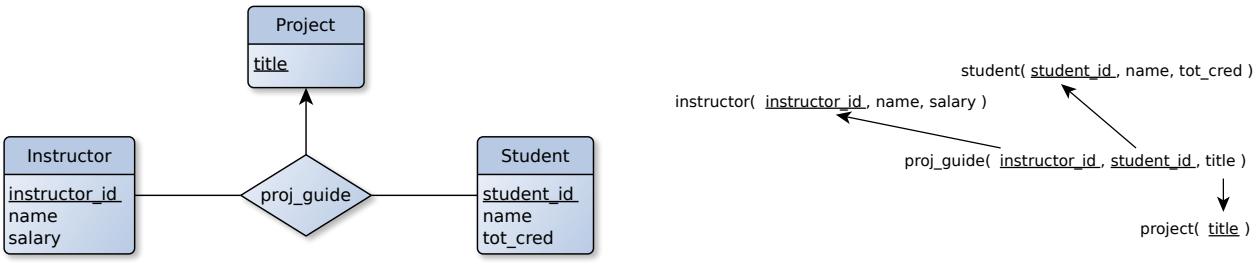
- Relation schema for inst_dept
  - Instead of creating a schema for the relationship add an attribute dept_name to the schema for the entity-set instructor

20.4.5 n-ary Relationship Sets Primary Keys

- If no arrows (**many-to-many** mapping), relationship-set primary key is union of all participating entity-sets' primary keys
- If one arrow (**one-to-many** mapping), relationship-set primary key is union of primary keys of entity-sets without an arrow
- Don't allow more than one arrow for relationship-sets with degree > 2

20.4.6 Example: n-ary Relationship Sets





E-R Model | Relational Schema

20.4.7 Relationship Sets Foreign Keys

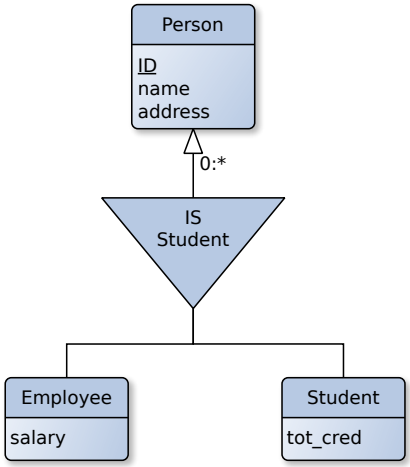
- Relationship-sets associate entities in entity-sets
  - We need foreign-key constraints on relation schema for R !
- For each entity-set E_i participating in R:
  - Relation schema for R has a foreign-key constraint on E_i relation, for $primary\_key(E_i)$ attributes
- Relation schema notation doesn't provide mechanism for indicating foreign key constraints

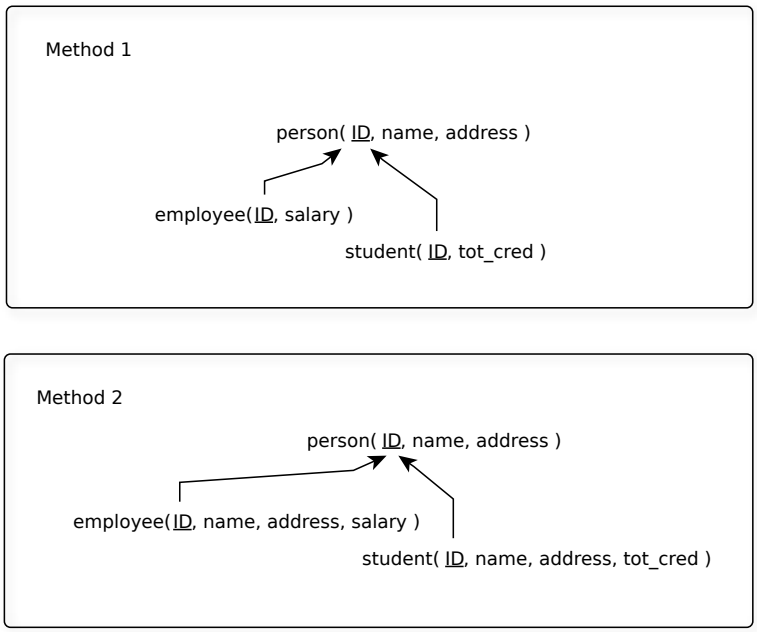
  - Don't forget about foreign keys and candidate keys!
    - Making notes on your relational model schema is a very good idea
  - Can specify both foreign key constraints and candidate keys in the SQL DDL

20.5 Representation of Generalization/Specialization

- Mapping generalization/specialization to relational model is straightforward
- Method 1:
  - Create relation schema for higher-level entity-set
    - Including primary keys, etc.
  - Create schemas for lower-level entity-sets
    - Subclass schemas include superclass' primary key attributes!
    - Primary key is same as superclass' primary key
      - Subclasses can also contain their own candidate keys!
      - Enforce these candidate keys in implementation schema
  - Foreign key reference from subclass schemas to superclass schema, on primary-key attributes
  - Drawback: getting information about, an employee requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema
- Method 2:
  - Create relations schemas for each entity set with all local and inherited attributes
  - If specialization is total, the schema for the generalized entity set is not required to store information
    - Each lower-level entity-set has its own relation schema
      - All attributes of the supertype are included on each subtype
      - The generalized entity set can be defined as a "view" relation containing union of specialization relations
      - But explicit schema may still be needed for foreign key constraints
  - Drawback: If the specialization is overlapping attributes of the supertype may be stored redundantly for subtypes

20.5.1 Example: Generalization/Specialization



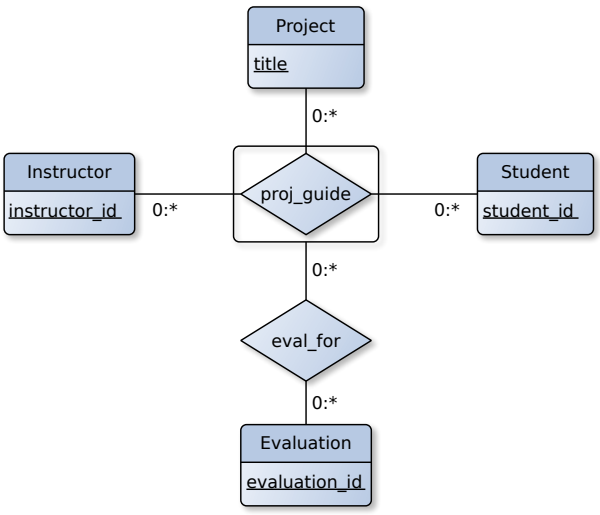


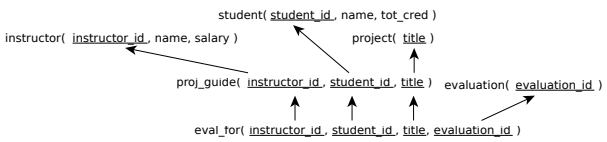
E-R Model | Relational Schema

20.6 Representation of Aggregation

- Mapping aggregation to relational model is straightforward
- Create relation schema for the relation containing
  - primary key of the aggregated relationship,
  - the primary key of the associated entity set
  - any descriptive attributes.

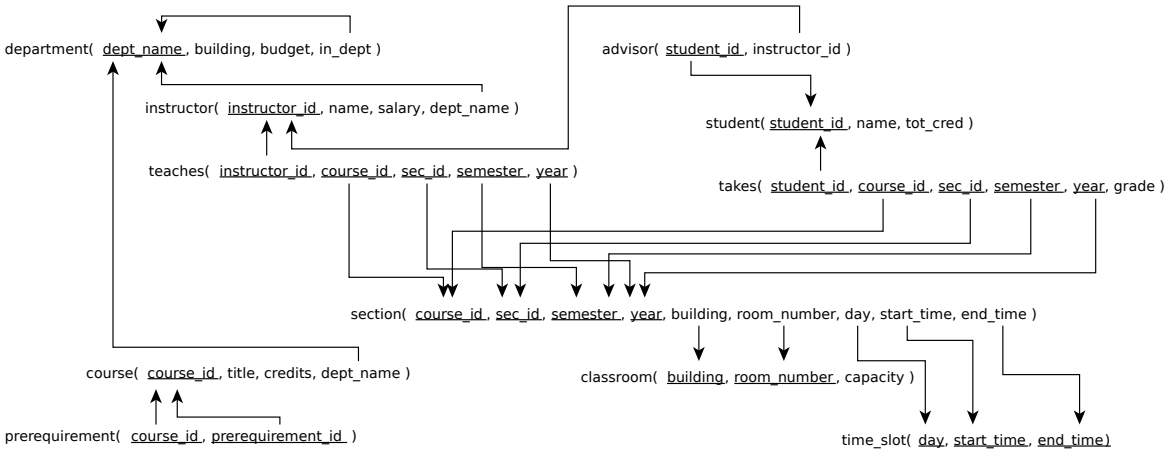
20.6.1 Example: Aggregation





E-R Model | Relational Schema

20.7 Example Schema



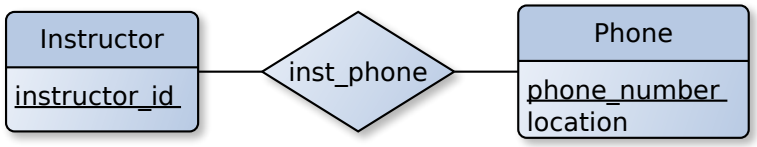
21 Entity-Relationship Design Issues

21.1 Naming of Entities, Attributes and Relationships

- **Unique-role assumption**, each entity, attribute and relationship name has a unique meaning
  - Prevents us from using the same attribute to mean different things
    - E.g. Using **number** as an attribute in entity-set **instructor** for **phone_number** and in **classroom** for **room_number** is meaningless.
- Use singular or plural form for naming entity-sets, but be consistent.

21.2 Use of Entity Sets versus Attributes

- Use of phone as an entity type allows extra information about phone numbers (plus multiple phone numbers)

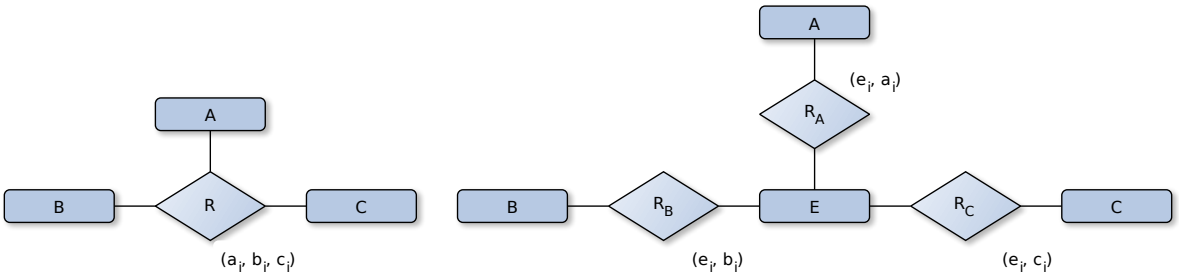


21.3 Use of Entity Sets versus Relationship Sets

- Possible guideline is to designate a relationship-set to describe an action that occurs between entities

21.4 Binary versus n-ary Relationship Sets

- Although it is possible to replace any nonbinary (n-ary, for n $>$ 2) relationship-set by a number of distinct binary relationship-sets, a n-ary relationship-set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
  - E.g., A ternary relationship parents, relating a child to his/her father and mother, is best replaced by two binary relationships, father and mother
    - Using two binary relationships allows partial information (e.g., only mother being know)
    - If it makes sense to model as separate binary relationships, do it that way!
  - But there are some relationships that are naturally non-binary
    - Example: proj_guide
- For degree $>$ 2 relationships, could replace with binary relationships
  - Replace n-ary relationship-set with a new entity-set E
    - Create an identifying attribute for E (e.g. an auto-generated ID value)
  - Create a relationship-set between E and each other entity-set
  - Relationships in R must be represented in R_A, R_B , and R_C

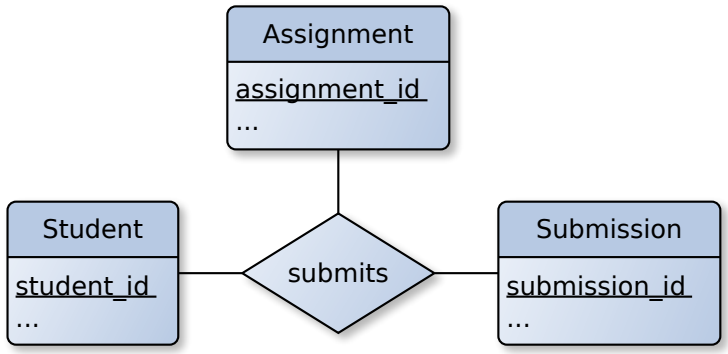


- **Are these representations identical?**
- Example: Want to represent a relationship between entities a_5 ,b_1 and c_2
  - How many relationships can we actually have between these three entities?
- Ternary relationship set:
  - Can only store one relationship between a_5, b_1 and c_2, due to primary key of R
- Alternate approach:
  - Can create many relationships between these entities, due to the entity-set E!
    - (a_5, e_1), (b_1, e_1), (c_2, e_1)
    - $(a5, e_2)$, (b_1, e_2), (c_2, e_2)
    - $(\ldots)$
  - Can't constrain in exactly the same ways

- Also need to translate constraints
  - Translating all constraints may not be possible

21.4.1 Example: Use Case Assignments

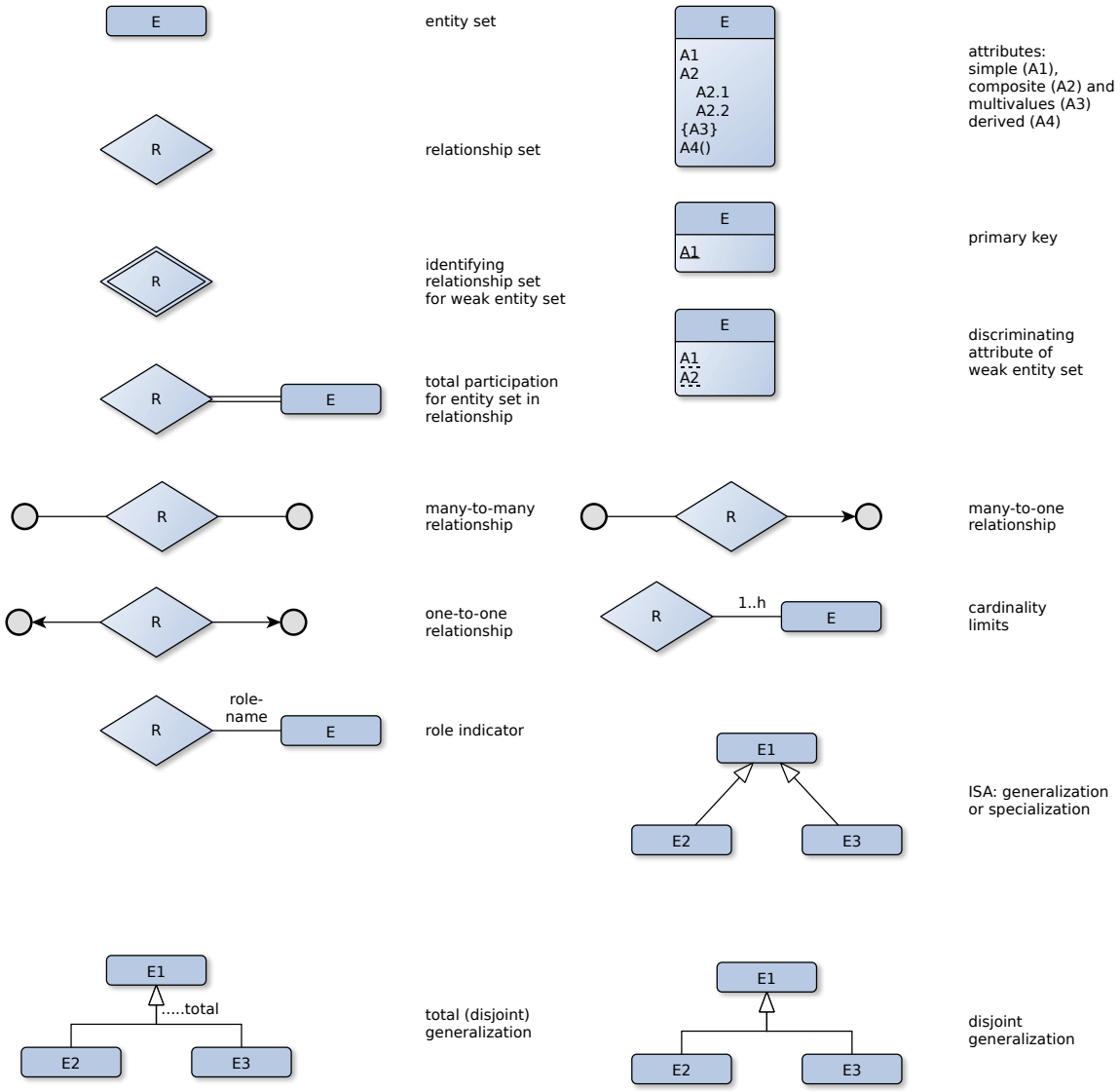
- Ternary relationship between student, assignment, and submission
  - Need to allow multiple submissions for a particular assignment, from a particular student
- In this case, it could make sense to represent as a ternary relationship
  - Doesn't make sense to have only two of these three entities in a relationship



- Alternative equivalent representations:
- Two binary total relationships
  - Required to ensure that every submission has an associated student, and an associated assignment

- Could even make submission a weak entity-set
  - Student and assignment are identifying entities!
  - Discriminator for submission is version number.

22 Summary of Symbols Used in E-R Notation



23 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

24 Motivation

- Different schemas to represent a set of data

  - Which is best? How is “best” defined.

An optimal logical database structure is key to good database performance.

- Redundancies in the logical database structure are by far the most common reason for poor database performance.

24.1 Main goals:

- Complete representation of information
- Data should
  - not be unnecessarily redundant
  - be easy to manipulate
- Constraints should be easy to enforce

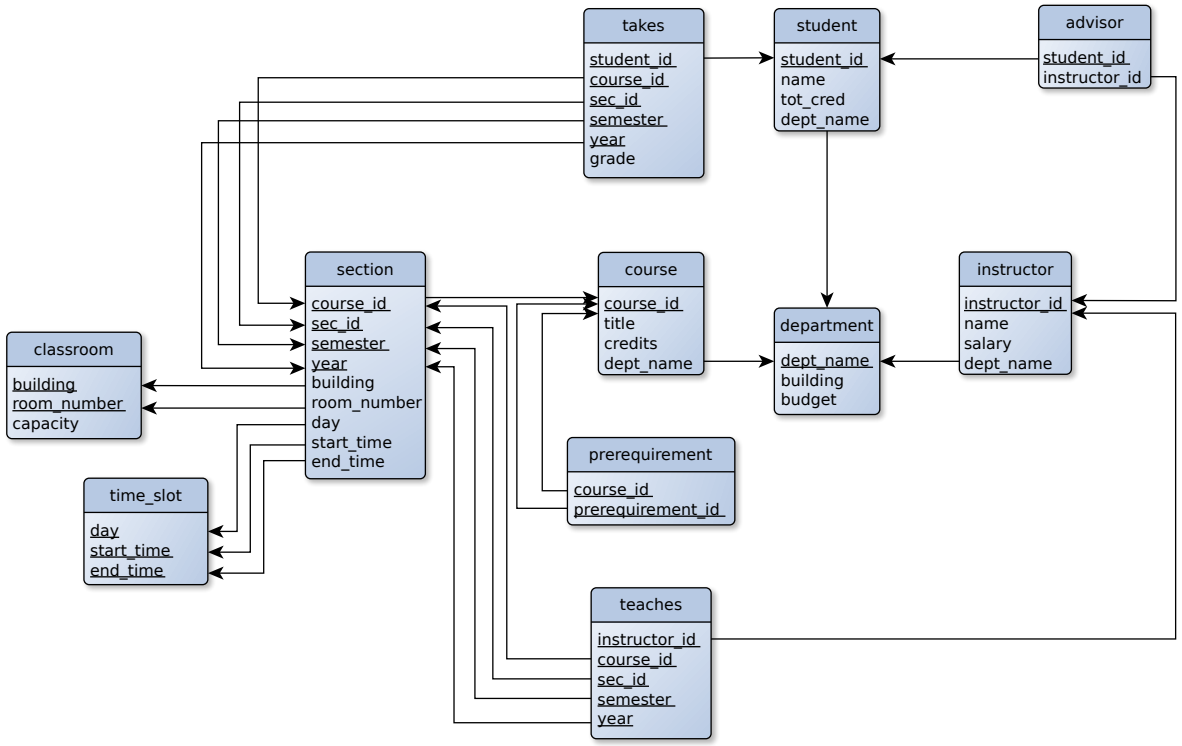
24.2 Database Normalization

- a process helping to understand the requirements of efficient, redundance-free database structures
- not a practical method to develop efficient, redundance-free database structures

24.3 Normal Forms

- Are “good” patterns for database schemas
- Several different normal forms, with different constraints
- Are formally specified
  - test + transform is possible

24.4 Example Schema Design



- Many-to-one mapping
  - A instructor is part of one department
  - A department can have multiple instructors

24.5 Larger Schemas

- Suppose we combine instructor and department into inst_dept
  - Without all other mappings

inst_dept

instructor_id	name	dept_name	salary	building	budget
10101	Srinivasan	Comp.Sci.	65000	Taylor	100000
12121	Wu	Finance	90000	Painter	120000
15151	Mozart	Music	40000	Packard	80000
22222	Einstein	Physics	95000	Watson	70000
32343	ElSaid	History	60000	Painter	50000
33456	Gold	Physics	87000	Watson	70000
45565	Katz	Comp.Sci.	75000	Taylor	100000
58583	Califieri	History	62000	Painter	50000
76543	Singh	Finance	80000	Painter	120000
76766	Crick	Biology	72000	Watson	90000
83821	Brandt	Comp.Sci.	92000	Taylor	100000
98345	Kim	Elec.Eng.	80000	Taylor	85000

- Rationale:
  - Eliminates a join when retrieving all instructors associated with a certain department
- Problem: Mapping between instructors and departments is one-to-many
  - Multiple redundant copies of department to keep in sync!

- Identifying the problem
  - "Because we see values that appear multiple times"
    - Not good enough
    - E.g. different instructors with the same salary
  - Repeated values don't automatically indicate a problem !!!!

24.6 Functional Dependency

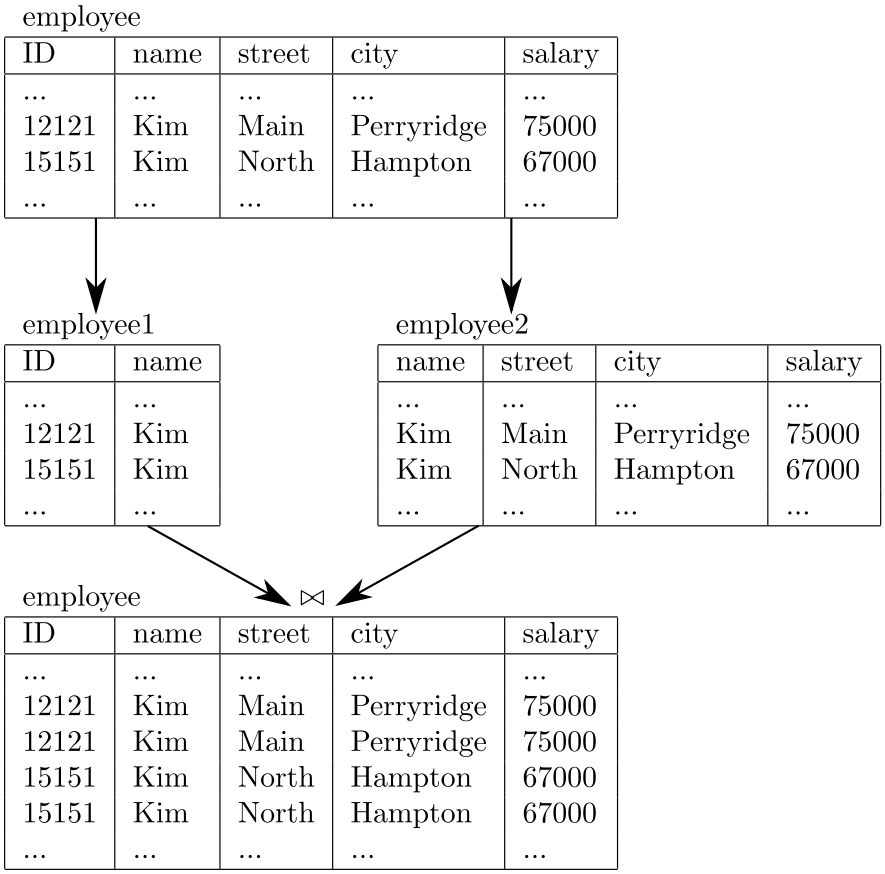
- One rule which we are modelling is:
  - "Every department (identified by its name) is located in exactly one building and can only have budget."
- In other words:
  - Every dept_name corresponds to exactly on building and budget
  - "If there were a schema (dept_name, building, budget), dept_name would be a candidate key."
- Such rules are denoted as **functional dependency**:
  - dept_name $\rightarrow$ building, budget
  - dept_name functionally determines building and budget
  - are very important in schema analysis
    - Have a lot to do with keys!
    - "Good" schema designs are guided by functional dependencies
    - Frequently helpful to identify them during schema design

24.7 Smaller Schemas

- Looking at the inst_dept relation schema, as dept_name is not a candidate key, the building and budget information have to be repeated.
  - This indicates the need to decompose inst_dept

24.7.1 Another Example Schema

- Looking at a "large" schema for employee information.
  - employee(ID, name, street, city, salary)
  - Employee ID is unique, but other attributes could have duplicate values
- Suppose we decompose this schema into
  - employee1(ID, name) and
  - employee2(name, street, city, salary)



- Problems:
  - We cannot generate the original data with a join: employee1 ⋈ employee2
  - The join on non-unique values can generate invalid tuples!
  - This is a **lossy decomposition**!!!
- **"Good" schema designs avoid lossy decompositions!!!**

25 Atomic Domains and 1st Normal Form

25.1 1st Normal Form

25.1.1 Definition

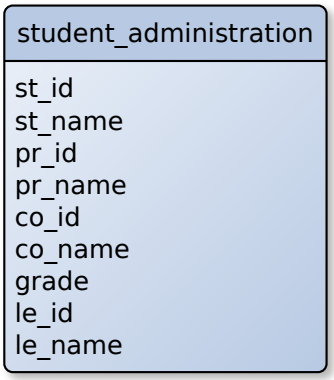
A relation schema R is in **1st normal form (1NF)**, if all attribute domains of R are atomic.

25.1.2 Atomic Domains

- Domain is atomic if its elements are considered to be indivisible (simple, unstructured) units
- Examples of non-atomic domains:
  - Set of names, composite attributes
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
- E-R model supports non-atomic attributes
  - Multivalued attributes, Composite attributes
- Relational model specifies atomic domains for attributes
  - Schemas are automatically in 1NF

  - Mapping from E-R model to relational model changes composite/multivalued attributes into an atomic form

25.1.3 Example



- **student-administration** is an example for a **universal relation**
- **Universal relation**
  - Single relation containing all relevant information in form of simple, unstructured, atomic values

25.1.3.1 Information Content

- Student: ID, Name, Program, Courses, Grade per course
- Program: ID, Name, Courses, Students
- Course: ID, Name, Responsible lecturer, Students, Grades ps students
- Lecturer: ID, Name, Courses

25.1.3.2 Problems

- INSERT Anomaly
  - Introduction of a new program
    - no students in the beginning
    - no courses in the beginning
- UPDATE Anomaly
  - Correction of the lecturer name “Karl Müller” should be “Carl Muller”
  - How many rows do we need to update?
- DELETE Anomaly
  - Student with ID 4711 decides to leave
  - How many rows do we have to remove?
- **Origin**: dependencies between attributes

25.1.4 Further Normal Forms

- Relate to functional dependencies
  - Decide whether a particular relation R is in “good” form.
  - If R is not in “good” form, decompose R in a set of relations
    - each relation should be in good form

        * the decomposition should be lossless

26 Decomposition Using Functional Dependencies

26.1 Functional Dependencies

- Constraints on the set of **legal** relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a key.

26.1.1 Formal definition

- Given a relation schema R with attribute sets $\alpha, \beta \subseteq R$ with $\alpha \neq \beta$
- β is functional dependent on α ($\alpha \rightarrow \beta$), iff for every set of values for α there is exactly one set of values for β
  in this case α is often called a **determinant** of β
  - In other words:
    * For all pairs of tupples t_1 and t_2 in $r(R)$, if $t_1[\alpha] = t_2[\alpha]$ then $t_1[\beta] = t_2[\beta]$
- β is **fully functional** depedent on α, iff
  $\alpha \rightarrow \beta \wedge (\gamma \subseteq \alpha) \wedge (\gamma \rightarrow \beta) \Rightarrow (\gamma = \alpha)$
  - In other words:
    * No subset γ of attribute set α exists, that functionally determines β.

26.1.2 Example, Functional Dependencies

- st_id $\rightarrow$ st_name
- pr_id $\rightarrow$ pr_name
- co_id $\leftrightarrow$ co_name, co_id $\rightarrow$ le_id, co_id $\rightarrow$ le_name
- le_id $\rightarrow$ le_name
- (st_id, co_id) $\rightarrow$ grade

26.1.3 Example, Candidate Keys

- {st_id, pr_id, co_id}
- {st_id, pr_id, co_name}

Root cause of our problems
Some attributes are functional dependent on parts of the keys

26.2 2nd Normal Form

26.2.1 Definition

A relation R is in **2nd normal form (2NF)**, if R is in 1NF and all non-key attributes of R are fully functional dependent on every key of R.

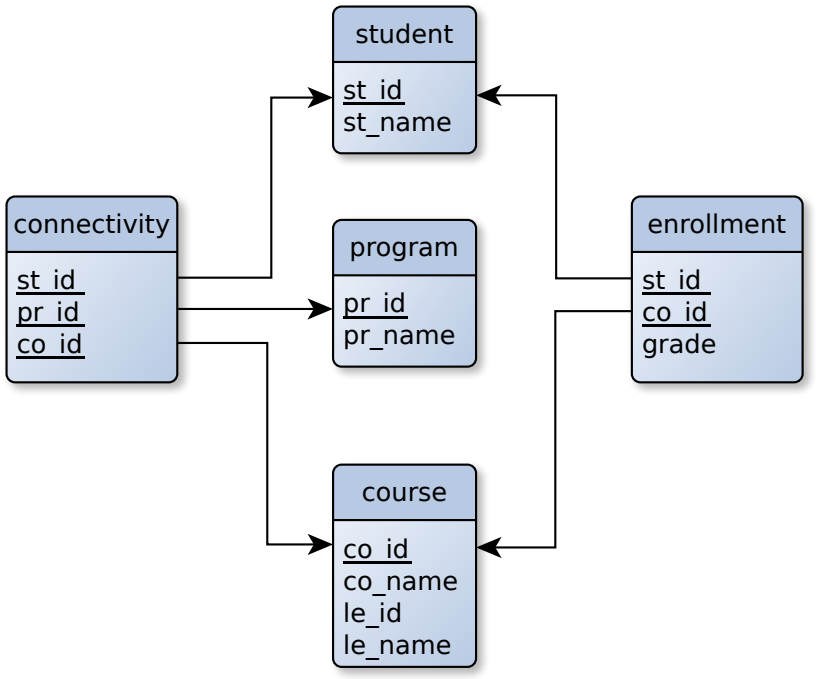
26.2.2 Check

If some attributes of R are fully determined by a subset of the attributes of the primary key, then 2NF doesn't hold.

26.2.3 How dow we get there?

1. Define primary key
   - If key only contains one attribute, stop, 2NF holds
2. Perform check
   - If ok, stop, 2NF holds
3. Perform decomposition
   - Create a new relation with the partial key as primary key
   - Move all functional dependent attributes to the new relation
4. Create a relation which joins the old and new relation
5. Repeat as long as 2NF doesn't hold for any relation in the schema

26.2.4 Example



26.2.4.1 Problems

- INSERT Anomaly
  - Hiring a new lecturer no course so far
- UPDATE Anomaly
  - Name change of lecturer (because of marriage) "Thomas Miller" becomes "Thomas Schuster"
- DELETE Anomaly
  - Replacing course 304 by course 344
    What happens with the lecturer of course 304?

- **Origin**: dependencies between attributes as before

26.3 Third Normal Form

26.3.1 Transitive Dependency

- Given a relation schema R with attribute sets $\alpha, \beta, \gamma \subseteq R$ with $\alpha \neq \beta$, $\alpha \neq \gamma$ and $\beta \neq \gamma$
- γ is transtive dependent on α ($\alpha \rangle \gamma$), iff
  (fully $\alpha \rightarrow \gamma$) $\wedge$ ($\exists \beta$: fully $\alpha \rightarrow \beta \wedge$ fully $\beta \rightarrow \gamma$)

26.3.1.1 Example, Transitive Dependencies in Course

- co_id⟩le_name
  - co_id → le_id → le_name
- co_name⟩le_name
  - co_name → le_id → le_name

26.3.2 Definition

A relation R is in **3rd normal form (3NF)**, iff R is in 2NF and no non-key attribute of R is transitive dependent on a key of R.

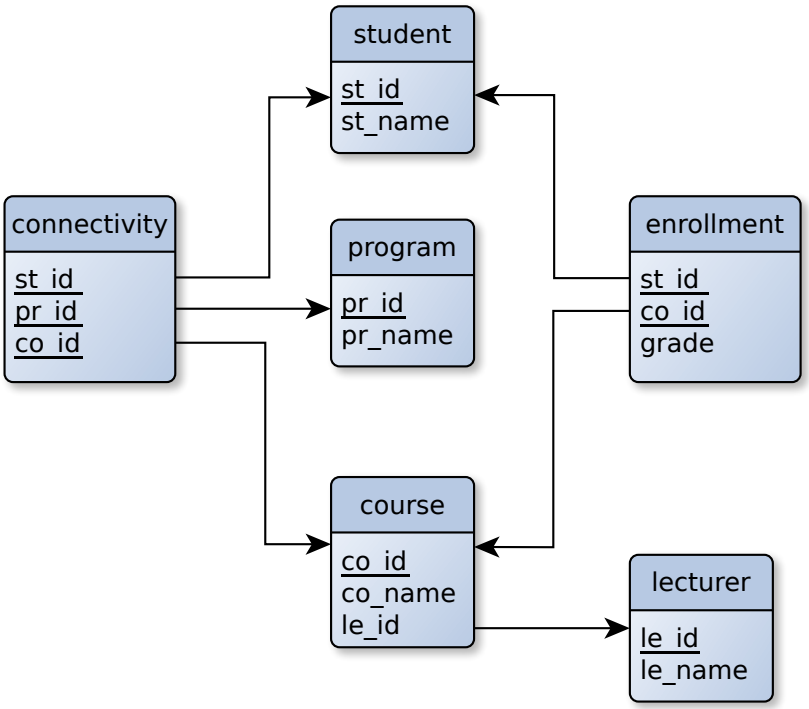
26.3.3 Check

If some attributes are fully determined by a non-key attribute, then 3NF doesn't hold.

26.3.4 How dow we get there?

1. Perform check
   - If ok, stop, 3NF holds
2. Perform decomposition
   - Create a new relation with the non-key attribute as primary key
   - Move all functional dependent attributes to the new relation
3. Repeat as long as 3NF doesn't hold for any relation in the schema

26.3.5 Example



26.3.5.1 Problems

- INSERT Anomaly
  - Adding a course to a programme without any students yet
- UPDATE
  - Anomaly Student IDs have to be replaced per student and course
- DELETE Anomaly
  - Once the last student leaves a programme we'll loose the information about its courses
- **Origin**: dependencies between attributes as before

26.4 Boyce-Codd Normal Form

26.4.1 Definition

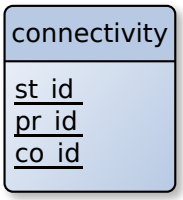
A relation R is in **Boyce-Codd Normal Form (BCNF)**, iff R is in 2NF and every determinant in R is a key of R

- BCNF is a stronger requirement than 3NF
- BCNF $\subset$ 3NF

26.4.2 Example

- In our example we have treated all determinants as keys
  - This means, we are in BCNF already

26.4.2.1 Problems



- INSERT Anomaly
  - Adding a course to a programme without any students yet
- UPDATE Anomaly
  - Student IDs have to be replaced per student and course
- DELETE Anomaly
  - Once the last student leaves a programme we'll loose the information about its courses
- **Origin**: dependencies between attributes as before

26.4.3 Dependency Preservation

26.4.3.1 Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - E.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The closure F^+ of F, is the set of all functional dependencies logically implied by F

26.4.3.2 Definition

Let F_i be the set of dependencies F^+ that include only attributes in R_i.

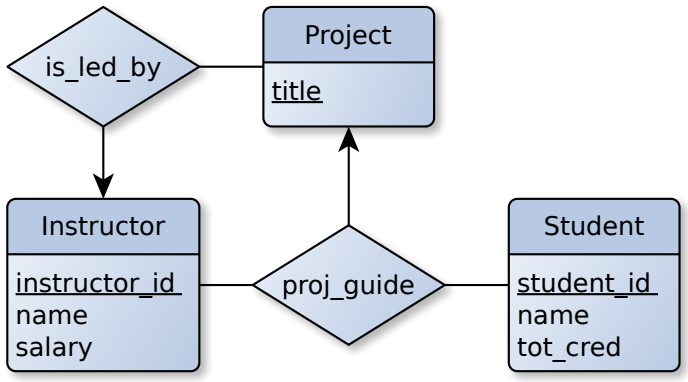
- A decomposition is dependency preserving, if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
- If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

26.4.3.3 BCNF and Dependency Preservation

- It is not always possible to get a BCNF decomposition that is dependency preserving.
  - Some functional dependencies are not enforceable within a single table
  - Can't enforce them with a simple key constraint, so they are more expensive

26.4.4 Another example

- University sets a requirement on projects
  - Each project can only have one instructor as leader
  - title $\rightarrow$ instructor_id
- University wants to give each student a personal instructor for each project
  - In each project, a student has exactly one personal instructor
  - (Different instructors in different projects are posssible)
  - instructor_id, student_id $\rightarrow$ title



- Relationship-set schemas:
  - led_by(title, instructor_id)
  - proj_guide(instructor_id, student_id, title)
- **Problem**:
  - This schema isn't in BCNF
  - proj_guide isn't in BCNF, as title isn't a candidate key
  - proj_guide repeats instructor_id unnecessarily
- **Decompose**:
  - led_by(title, instructor_id) already has title → instructor_id
  - create student_proj(student_id, title)
    - A student can have one instructor for each project so both student_id and title must be in the primary key
- **Problem**
  - Not easy to constraint that each student hase only one instructor for each project.

26.5 Comparison of BCNF and 3NF

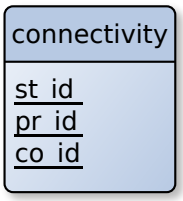
- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.

27 Decomposition Using Multivalued Dependencies

27.1 Multivalued Dependencies

- Given a relation schema R with attribute sets $\alpha, \beta \subseteq R$ with $\alpha \neq \beta$
- β is multi-valued dependent on α ($a\rangle\rangle b$), iff the set of values for β, which is associated with a single value of α is dependent on α only
- A multi-valued dependency is trivial, iff $\alpha \cup \beta = R$

27.1.1 Example



- Multi-valued Dependencies in connectivity
  - st_id⟩⟩pr_id, Student is enrolled in a program
  - pr_id⟩⟩co_id, Course belongs to program

27.2 Fourth Normal Form

27.2.1 Definition

A relation R is in **4th Normal Form (4NF)**, iff R is in BCNF and all multi-valued dependencies $\alpha\rangle\rangle\beta$ in R are trivial or α is a superkey for R.

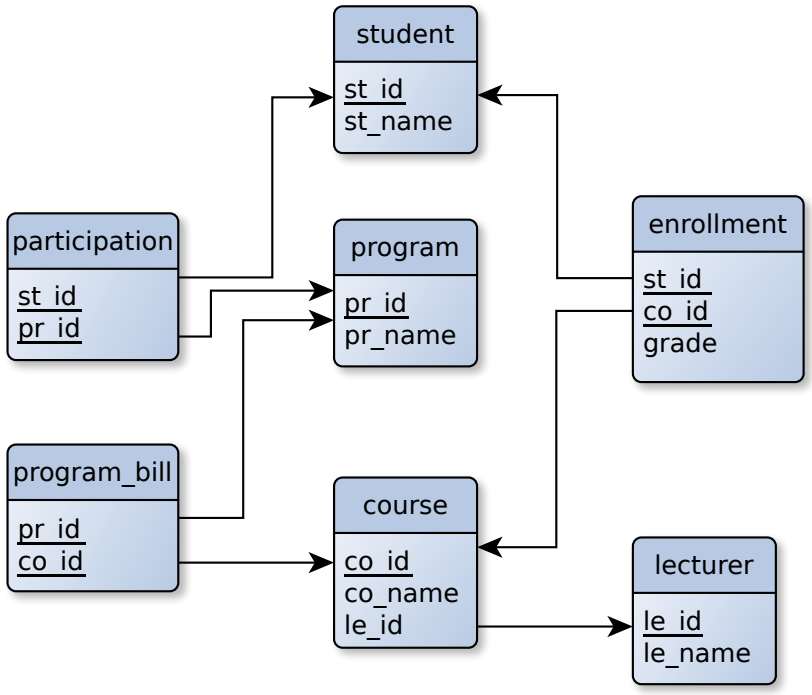
In other words:

No attribute (which by itself is not a candidate key or a superkey) is a determinant for other attributes.

27.2.2 How dow we get there?

- Lossless decompositionof the relation into a set of its projections.

27.2.3 Example



28 Further Normal Forms

28.1 Fifth Normal Form (Project-Join Normal Form (PJNF))

28.1.1 Definition

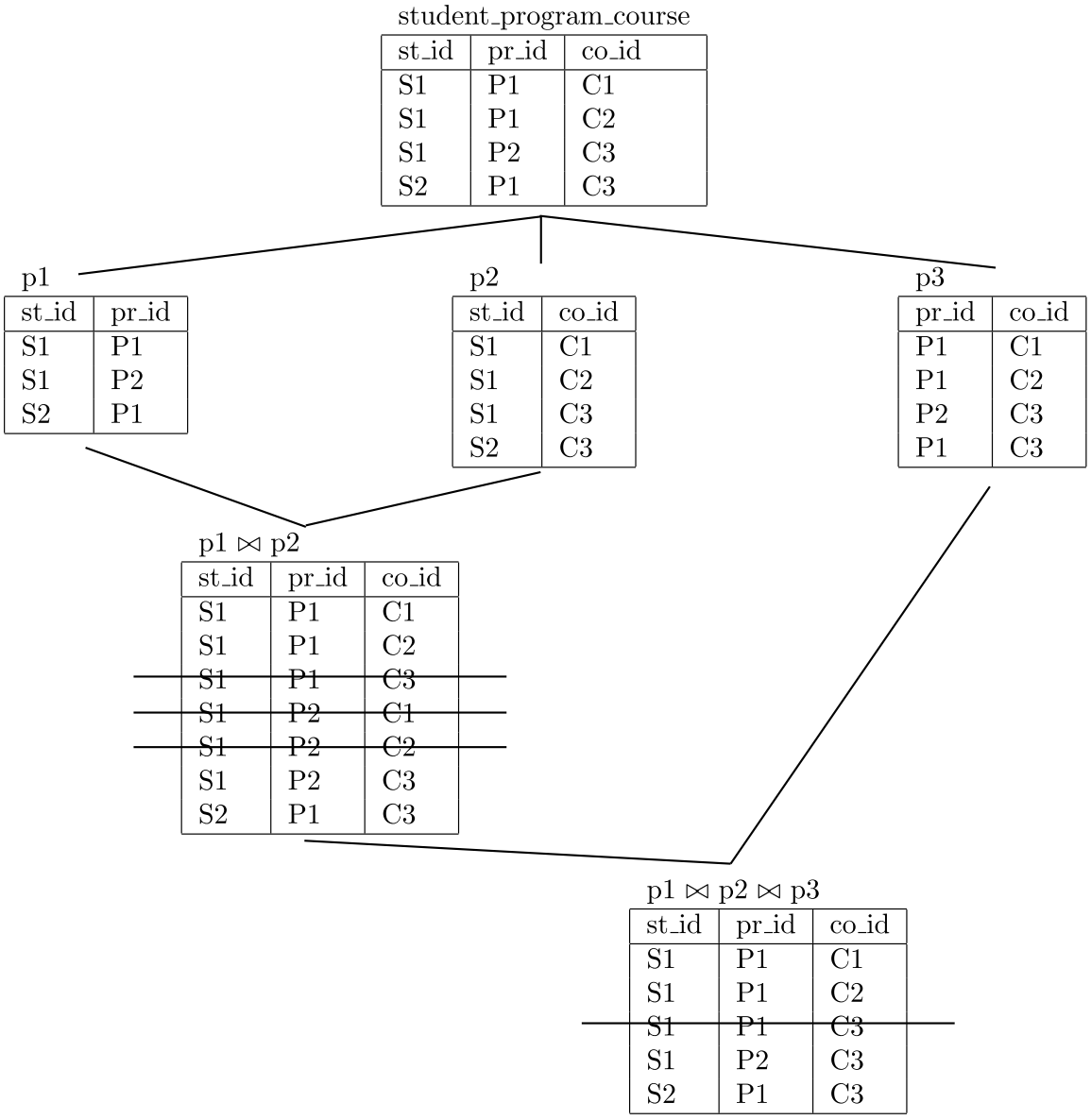
- A relation R is in **5th Normal Form (5NF)**, iff R is in 4NF and every **join dependency** in R is implied by the candidate keys
  - R cannot be decomposed into projections without loss of information
  - This decomposition results in 3 or more relations usually

In other words:

- A relation R is in **5th Normal Form (5NF)**, iff R can't be losslessly decomposed into a set of other relations.
- A relation R is subject to a join dependency, iff R can always be recreated by joining multiple tables each having a subset of the attributes of R. If one of the relations in the join has all attributes of R the join dependency is trivial.

28.1.2 Example

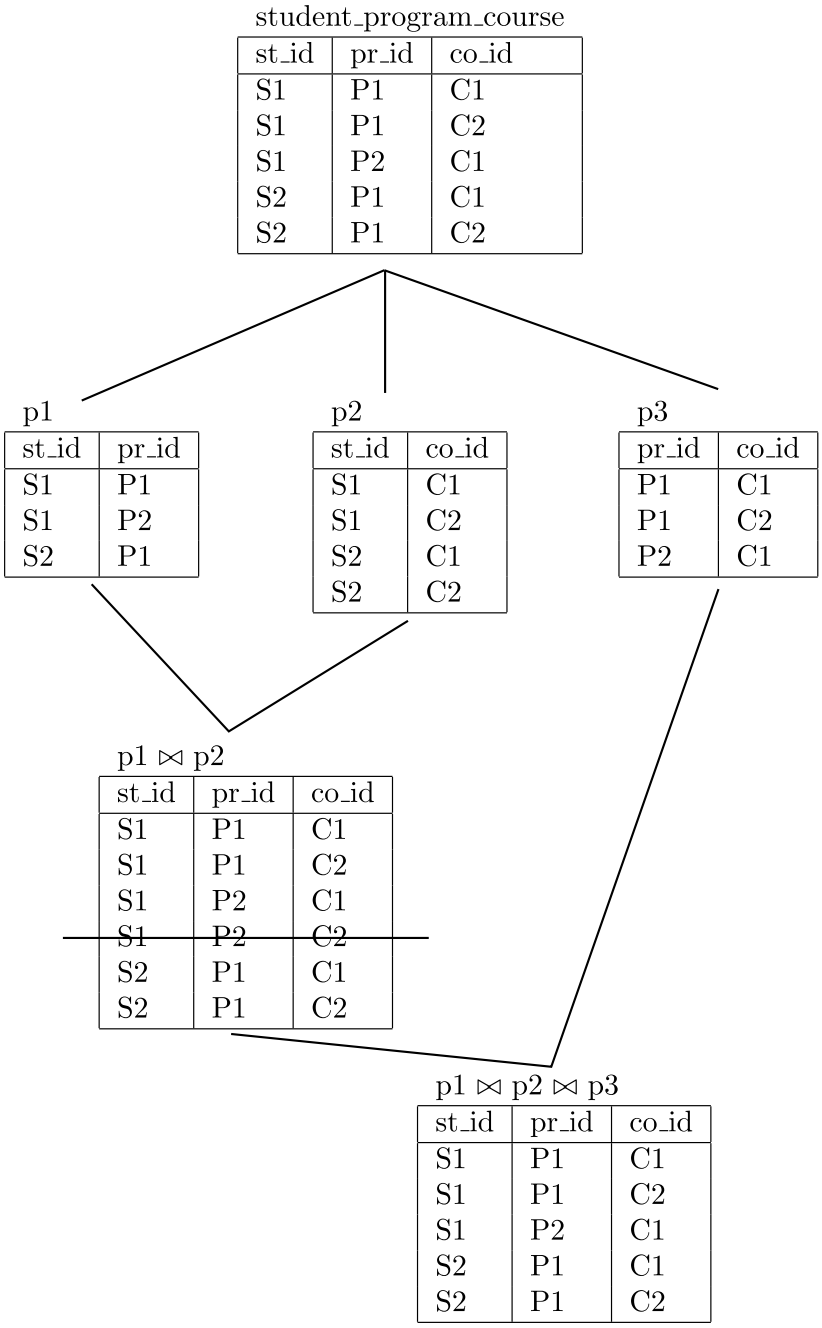
- A relation student_program_course(st_id, pr_id, co_id) records students, the programs they participate and courses of these programs they take.
- Students do not necessarily take all courses of programs in which they participate.



- The initial relation is necessary in order to show all the information required.
  - it is in 4NF
  - but redundant elements
- Suppose a decomposition into the two projections
  - p1(st_id, pr_id)
  - p2(st_id, co_id)
  - The redundancy is now eliminated.
  - But the natural join p1 ⋈ p2, contains spurious entries
- Even when joining with the 3rd possible projection p3(pr_id, co_id)
  - The result contains spurios entries

28.1.3 Altered Example

- Now suppose a student has to take **all** courses of a propgram which he participates.
- We assume that P1 contains courses C1 and C2, and P2 contains course C2



- The join p1 ⋈ p2 still contains one spurious entries.
- If this result is joined with p3 a correct recomposition of the original table is obtained.
- Therefore the original table violated 5NF.

28.2 Proposed 6NFs

- Domain-key Normal Form (DKNF) based on:
  - Domain constraints: what values may be assigned to attribute A
    - Usually inexpensive to test, even with `CHECK` constraints
  - Key constraints: all attribute-sets K that are a superkey for a schema R (i.e. $K \to R$)
    - Almost always inexpensive to test
  - General constraints: other predicates on valid relations in a schema
    - Could be very expensive to test!

  - A schema R is in DKNF if the domain constraints and key constraints logically imply the general constraints

29 E-R Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- Functional dependencies from non-key attributes of a relationship set possible
- Most relationships are binary

30 Normal Forms Overview

- 1NF
  - Tables of simple, atomic, unstructured values
  - As required by the relational model
- 2NF
  - Attributes are functional dependent on complete key
- 3NF and BCNF
  - Attributes are functional dependent on key only
- 4NF
  - Only trivial multi-valued dependencies in komplex keys
- 5NF or PJNF
  - Decomposition of complex keys as long as they can be re-created by joins
- 6NF
  - Multiple proposals

31 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

32 Overview of the SQL Query Language

- SQL = Structured Query Language
- Original language: "SEQUEL"
  - developed as part of System R project at the IBM San Jose Research Laboratory (early 1970's)
  - "Structured English Query Language"
- Simple, declarative language for writing queries
- Standardized by ANSI/ISO
  - SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011
- Most implementations loosely follow the standards (plenty of portability issues)
  - most systems offer most, if not all, SQL-92 features

32.1 SQL Features

- Data Definition Language (DDL)
  - Specify relation schemas (attributes, domains)
  - Specify a variety of integrity constraints
  - Access constraints on data
  - Indexes and other storage "hints" for performance
- Data Manipulation Language (DML)
  - Generally based on relational algebra
  - Supports querying, inserting, updating, deleting data
  - Very sophisticated features for multi-table queries

32.2 SQL Basics

- SQL language is case-insensitive
- SQL statements end with a semicolon
- SQL comments have two forms:
  - Single-line comments start with two dashes

    ```
    -- This is a SQL comment.
    ```
  - Block comments follow C style

    ```
    \*
    * This is a block comment in SQL.
    */
    ```

32.2.1 SQL Names

- Tables, columns, etc. require names
- Rules on valid names can vary dramatically across implementations
- Good, portable rules:
  - First character should be alphabetical
  - Remaining characters should be alphanumeric or underscore "_"
  - Use same the case in DML that you use in DDL

```
%reload_ext sql
%sql postgresql://universitydb:universitydb@127.0.0.1:65432/university
%config SqlMagic.feedback = True
%config SqlMagic.style = 'DEFAULT'
%config SqlMagic.short_errors=True
```

33 SQL Data Manipulation Language (DML)

33.1 Basic Query Structure

- SQL queries use the `SELECT` statement
- Central part of SQL language
- General form:

```
SELECT select_list
FROM table_expression
WHERE search_condition;
```

33.2 The Select Clause

- `SELECT` $A_1, A_2, \ldots$
  - Corresponds to a relational algebra **project** operation
    - $\Pi_{A_1,A_2,\ldots}(\ldots)$
- `FROM` $r_1, r_2, \ldots$
  - Corresponds to **Cartesian product** of relations $r_1, r_2, \ldots$
    - $r_1 \times r_2 \times \ldots$
- `WHERE` P
  - Corresponds to a **selection** operation
    - $\sigma_P(\ldots)$
- Assembling it all:
  `SELECT` $A_1, A_2, \ldots$ `FROM` $r_1, r_2, \ldots$ `WHERE` P;
  - Corresponds to:
    $\Pi_{A_1,A_2,\ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$
- Example: "We'd like to see the name and budget of all departments with a budget less than 90000.–"

```
%%sql
SELECT dept_name, budget
FROM department
WHERE budget < 90000;
```

33.2.1 SQL and Duplicates

- Biggest difference between relational algebra and SQL is use of multisets
  - In SQL, relations are **multisets** of tuples, not sets
- Reason:
  - Removing duplicate tuples is time consuming!
- SQL provides ways to exclude duplicates for all operations
  - To force the elimination of duplicates, insert the keyword `DISTINCT` after select.
- Example: "Find all department names with at least one associated instructor."

```
%%sql
SELECT dept_name FROM instructor;
```

- Equivalent to:

```
%%sql
SELECT ALL dept_name FROM instructor;
```

- To eliminate duplicates:

```
%%sql
SELECT DISTINCT dept_name FROM instructor;
```

33.2.2 Selecting Specific Attributes

- Can specify one or more attributes to appear in result

- An asterisk in the select clause denotes “all attributes”
- An attribute can be a literal with no from clause

- An attribute can be a literal with no from clause
  `SELECT '437'`
  - Results is a table with one column and a single row with value “437”
- Examples:

```
%sql SELECT * FROM instructor;
```

```
%sql SELECT '437';
```

33.2.3 Computing Results

- The `SELECT` clause is a generalized projection operation
  - Can compute results based on attributes
  - Computed values don’t have a (standard) name!
- Can also name (or rename) values
- Example: “Create a list of the names and monthly budgets of all departments.”

```
%%sql
SELECT dept_name, ROUND(budget/12,2) AS monthly_budget
FROM department;
```

33.3 The Where Clause

- The `WHERE` clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- Can use comparison operators:
  - $=, <>$ equals, not-equals ($! =$ also usually supported)
  - $<, \leq$ less than, less or equal
  - $>, \geq$ greater than, greater or equal
- Can refer to any attribute in FROM clause
- Can include arithmetic expressions in comparisons
- Comparison results can be combined using the logical connectives `AND`, `OR`, and `NOT`
- Example: “Find all instructors in Comp. Sci. department with salary > 80000”

```
%%sql
SELECT name, salary
FROM instructor
WHERE dept_name = 'Comp. Sci.' AND salary > 80000;
```

- SQL also has `BETWEEN` and `NOT BETWEEN` syntax
- `BETWEEN` **includes** interval endpoints
- Example: “Find all information for instructors with a salary between 40000 and 80000.”

```
%%sql
SELECT *
FROM instructor
WHERE salary BETWEEN 40000 AND 80000;
```

- Tuple comparison
  - Example: “Find all ID’s of courses tought by instructors of the”Biology" department. Togehter with the names of the instructors."

```
%%sql
SELECT name, course_id
FROM instructor, teaches
WHERE (instructor.instructor_id, dept_name) = (teaches.instructor_id, 'Biology');
```

33.3.1 String Operations

- String values can be compared
  - Lexicographic comparisons
  - Default is often to ignore case!
    - `SELECT 'HELLO' = 'hello';` Evaluates to true
- Pattern matching with `LIKE` expression
  `string_attr LIKE pattern`
  - pattern is a string literal enclosed in single-quotes
    - % (percent) matches a substring
    - _ (underscore) matches a single character
    - Can escape % or _ with a backslash

- Example: "Create a list of the names and monthly budgets of all departments the names of which start with the character"H""

```
%%sql
SELECT dept_name, ROUND(budget / 12, 2) AS monthly_budget
FROM department
WHERE dept_name LIKE 'H%';
```

- Example: "Find the names of all instructors whose name includes the substring"ar"".

```
%%sql
SELECT name
FROM instructor
WHERE name like '%ar%';
```

- Patterns are case sensitive.
- Pattern matching examples:
  - `Intro%` matches any string beginning with "Intro".
  - `%Comp%` matches any string containing "Comp" as a substring.
  - `_ _ _` matches any string of exactly three characters.
  - `_ _ _ %` matches any string of at least three characters.
- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

33.4 The From Clause

- The `FROM` clause lists the relations involved in the query
  - `FROM` $r_1, r_2, \ldots$
  - Corresponds to **Cartesian product** of relations $r_1, r_2, \ldots$
    - $r_1 \times r_2 \times \ldots$
- If multiple tables:
  - Select/project against Cartesian product of relations

  - Produces a row for every combination of input tuples.
- Cartesian product not very useful directly, but useful combined with `WHERE`-clause condition (selection operation in relational algebra)

- Example: "Find the Cartesian product of instructors and departments."

```
%%sql
SELECT *
FROM instructor, department;
```

- If tables have overlapping attributes, use `tbl_name.attr_name` to distinguish

```
%%sql
SELECT *
FROM instructor, department
WHERE instructor.dept_name = department.dept_name;
```

- The example query is the most basic form of a **join** (equijoin)
- Databases optimize equijoin queries very effectively.
- Will cover other forms of SQL join syntax later.

33.5 The Order By Clause

- The `ORDER BY` $A_1, A_2, \ldots$ is used to order SQL query results by particualr attributes.
- Two main categories of query results:
  - "Not ordered by anything"
    - Tuples can appear in any order
  - "Ordered by attributes $A_1, A_2, \ldots$"
    - Tuples are sorted by specified attributes
    - Results are sorted by A_1 first
    - Within each value of A_1, results are sorted by A_2 etc.
- We may specify `DESC` for descending order or `ASC` for ascending order, for each attribute; ascending order is the default.
- Example: "List in alphabetic order the names of all instructors."

```
%%sql
SELECT name
FROM instructor
ORDER BY name;
```

33.6 Aggregate Functions and Grouping

- SQL provides grouping and aggregate operations, just like relational algebra
- Aggregate functions:
  - `SUM` sums the values in the collection
  - `AVG` computes average of values in the collection
  - `COUNT` counts number of elements in the collection
  - `MIN` returns minimum value in the collection
  - `MAX` returns maximum value in the collection
- `SUM` and `AVG` require numeric inputs (obvious)

33.6.1 Examples

- "Find the average salary of instructors in the Computer Science department"

```
%%sql
SELECT AVG(salary)
FROM instructor
WHERE dept_name = 'Comp. Sci.';
```

- "Find the total number of instructors who teach a course in the Spring 2010 semester"

```
%%sql
SELECT COUNT(DISTINCT instructor_id)
FROM teaches
WHERE semester = 'Spring' AND year = 2010;
```

- "Find the number of tuples in the course relation"

```
%%sql
SELECT COUNT(*)
FROM course;
```

- "Find the average salary of instructors in each department"

```
%%sql
SELECT dept_name, AVG(salary)
FROM instructor
GROUP BY dept_name;
```

- Attributes in **SELECT** clause outside of aggregate functions must appear in **GROUP BY** list

```
%%sql
SELECT dept_name, instructor_id, AVG(salary)
FROM instructor
GROUP BY dept_name;
```

33.6.2 Having Clause

- The `HAVING` clause can use aggregate functions in its predicate
  - It's applied after grouping/aggregation, so those values are available
  - The `WHERE` clause cannot do this, of course

33.6.2.1 Examples

- "Find the names and average salaries of all departments whose average salary is greater than 42000."
- **Note**: predicates in the `HAVING` clause are applied after the formation of groups whereas predicates in the `WHERE` clause are applied before forming groups

```
%%sql
SELECT dept_name, AVG(salary)
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

33.7 Modification of the Database

33.7.1 Entering Data

- Tables are initially empty
- Use **INSERT** statement to add rows
  - String vales are single-quoted
  - (Double-quoted strings refer to columns names)
  - Values appear int the same order as table's attributes

33.7.1.1 Examples

- Add a new tuple to course

```
%%sql
INSERT INTO course
VALUES ('CS-437', 'Database Systems','Comp. Sci.',4);
```

- equivalent
  - Can specify which attributes in **INSERT**
    - Can list attributes in a different order
    - Can exclude attributes that have a default value

```
%%sql
INSERT INTO course(course_id, title, dept_name, credits)
VALUES ('CS-437', 'Database Systems','Comp. Sci.',4);
```

- Add a new tuple to student with tot_creds set to null

```
%%sql
INSERT INTO student
VALUES ('3003','Green','Finance',null);
```

- Add all instructors to the student relation with tot_creds set to 0

```
%%sql
INSERT INTO student
SELECT instructor_id, name, dept_name, 0
FROM instructor;
```

- The **SELECT FROM WHERE** statement is fully evaluated before any of it's results are inserted.

33.7.2 Replacing Data

33.7.2.1 Example

- "Increase salaries of instructors whose salary is over 100,000 by 3%, and all others receive a 5% raise"
  - Write two update statements
  - The order is important
  - Can be done better using **CASE** statement

```
%%sql
UPDATE instructor
SET salary = salary * 1.03
```

```
WHERE salary > 100000;
UPDATE instructor
SET salary = salary * 1.05
WHERE salary <= 100000;
```

- Alternative using the **CASE** statement

```
%%sql
UPDATE instructor
SET salary = CASE
    WHEN salary <= 100000 THEN
        salary * 1.05
    ELSE
        salary * 1.03
    END;
```

33.7.3 Removing Data

33.7.3.1 Examples

- Delete all instructors

```
%%sql
DELETE FROM instructor;
```

- Delete all instructors from the Finance department

```
%%sql
DELETE FROM instructor
WHERE dept_name = 'Finance';
```

- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.

```
%%sql
DELETE FROM instructor
WHERE dept_name IN (
    SELECT dept_name
    FROM department
    WHERE building = 'Watson');
```

- Delete all instructors whose salary is less than the average salary of instructors.

```
%%sql
DELETE FROM instructor
WHERE salary < (
    SELECT AVG(salary)
    FROM instructor);
```

33.8 Nested Queries

- Motivation: Often we don't know a value needed in a query, but we know a way to query it from the database.
- SQL provides a mechanism for the nesting of subqueries.

- A **subquery** is a **SELECT-FROM-WHERE** expression that is nested within another query.
- Common use: tests for set membership, set comparisons, and set cardinality.

33.8.1 Syntax

- The result of a query may be used as operand in a **WHERE** condition or in the **FROM** clause
- Nesting depth unlimited
- Nested queries need to be enclosed in parantheses

33.8.2 Operators for Nested Queries

- Binary operators
  - $(= \mid <> \mid > \mid \geq \mid < \mid \leq)$ `(<unordered_select>)`
    - Value comparison
    - Nested query results in one value exactly (One column of one row)
  - `[NOT] IN (<unordered_select>)`
    - Set mebership test
  - $(= \mid <> \mid > \mid \geq \mid < \mid \leq)$ `ANY/SOME(<unordered_select>)`
    - TRUE, if comparison results is TRUE for at least one element
    - Empty result set causes FALSE
  - $(= \mid <> \mid > \mid \geq \mid < \mid \leq)$ `ALL(<unordered_select>)`
    - TRUE, if comparison results is TRUE for all elements
    - Empty result set causes TRUE
- Unary operators
  - `[NOT] EXISTS (<unordered_select>)`
    - Is the result set non-empty?
  - `UNIQUE (<unordered_select>)`
    - Has the result set duplicate tuples?

33.8.3 Examples

- Find courses offered in Fall 2009 and in Spring 2010

```
%%sql
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND YEAR = 2009 AND course_id IN (
    SELECT course_id
    FROM section
    WHERE semester = 'Spring' AND year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
%%sql
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND YEAR = 2009 AND course_id NOT IN (
    SELECT course_id
```

```
    FROM section
    WHERE semester = 'Spring' AND year = 2010);
```

- Find the total number of (distinct) students who have taken course sections taught by the instructor with instructor_id 10101

```
%%sql
SELECT COUNT(DISTINCT student_id)
FROM takes
WHERE (course_id, sec_id, semester, year) IN (
    SELECT course_id, sec_id, semester, year
    FROM teaches
    WHERE teaches.instructor_id = '10101'
);
```

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
%%sql
SELECT DISTINCT T.name
FROM instructor AS T, instructor AS S
WHERE T.salary > S.salary AND S.dept_name = 'Biology';
```

- Same query using $>$ **SOME** clause

```
%%sql
SELECT name
FROM instructor
WHERE salary > SOME (
    SELECT salary
    FROM instructor
    WHERE dept_name = 'Biology'
);
```

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
%%sql
SELECT name
FROM instructor
WHERE salary > ALL (
    SELECT salary
    FROM instructor
    WHERE dept_name = 'Biology'
);
```

33.8.4 Synchronized Nested Queries (Correlation Variables)

- The **WHERE** condition of the nested query references fields from the nesting query
- The result set of the nested query is dependent on values from the nest query

33.8.4.1 Example

- Another way for:
  Find courses offered in Fall 2009 and in Spring 2010

```
%%sql
SELECT course_id
FROM section AS S
WHERE semester = 'Fall' AND year = 2009 AND EXISTS (
    SELECT * FROM section AS T
    WHERE semester = 'Spring' AND year = 2010 AND S.course_id = T.course_id
);
```

- **NOT EXISTS**
  Find all students who have taken all courses offered in the Biology department.

```
%%sql
SELECT DISTINCT S.student_id, S.name
FROM student AS S
WHERE NOT EXISTS (
        (SELECT course_id
         FROM course
         WHERE dept_name = 'Biology')
    EXCEPT
        (SELECT T.course_id
         FROM takes AS T
         WHERE S.student_id = T.student_id)

);
```

- **UNIQUE**
  Find all courses that were offered at most once in 2009

```
%%sql
SELECT T.course_id, R.
FROM course AS T, section AS R;
```

33.8.5 Subqueries in the From Clause

- SQL allows a subquery expression to be used in the from clause

33.8.5.1 Example

- Find the average instructors' salaries of those departments where the average salary is greater than 42000.

```
%%sql
SELECT dept_name, avg_salary
FROM (
    SELECT dept_name, AVG(salary) AS avg_salary
    FROM instructor
    GROUP BY dept_name
) alias
WHERE avg_salary > 42000;
```

33.9 Cartesian Product and Join

33.9.1 Motivation

- Very often we need data from more than one table in a single row of the result set
  - Example: "The name of an instructor together with the name of the building where his department is located."
- Wouldn't it be nice, if we could reference more than one table in the `FROM` clause of a `SELECT` statement?
  Yes We Can!!!

33.9.2 General

- Join operations take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the `FROM` clause

33.9.3 Syntax (SQL-92)

```
SELECT [DISTINCT] * | <expression>[, <expression>]...
FROM <t_exp>[, <t_exp>]...
[WHERE <condition>]
```

- `<t_exp>`
  - `<table> | <t_exp><join_specification>`
  - Evaluation from left to right
- `<join_specification>`
  - `<cross_join> | <nat_join> | <cond_join>`

33.9.4 Semantics (SQL-92)

- `<cross_join>`
  - Produces the Cartesian product
- `<nat_join>`
  - Natural Join, Join based on identical values for all columns having the same name
- `<cond_join>`
  - Conditional Join/ Theta Join
  - Join based on condition (logical expression)
  - Join based on identical values for a defined subset of the columns having the same name

33.9.5 Syntax Details (SQL-92)

- `<cross_join>`
  - `CROSS JOIN <table>`
- `<nat_join>`
  - `NATURAL (INNER | (LEFT | RIGHT | FULL) [OUTER]) JOIN <table>`

- `<cond_join>`
  - `(INNER | (LEFT | RIGHT | FULL) [OUTER]) JOIN <table> ON <condition> | USING (<column>[, <column>]...)`

33.9.6 Conditional Join/ Theta Join

- We skipped one relational algebra operation
- A generalized join operation the ''conditional join''or''theta join"
  - Written as $r \bowtie_\theta s$ short for $\sigma_\theta(r \times s)$
- SQL provides a syntax for theta joins

33.9.7 Examples

```
course = %sql SELECT * FROM course_s;
prerequirement = %sql SELECT * FROM prerequirement_s;
print("\n Relation course")
print(course)
print("Relation prerequirement")
print(prerequirement)
```

- Observe that
  - prerequirement information is missing for CS-315 and
  - course information is missing for CS-437

33.9.7.1 Example Theta Join 1

- "Associate all courses with their prerequirements"

```
%%sql
SELECT c.course_id, c.title, p.*
FROM course_s c INNER JOIN prerequirement_s p
ON c.course_id = p.course_id;
```

33.9.7.2 Example Theta Join 2

- Can join across multiple tables with this syntax
- "Associate all courses with their prerequirements and also print the title of the prerequired course."

```
%%sql
SELECT c.course_id, c.title, p.prerequirement_id, c2.title
FROM course_s c
    INNER JOIN prerequirement_s p
        ON c.course_id = p.course_id
    INNER JOIN course c2
        ON p.prerequirement_id = c2.course_id;
```

- Generally evaluated left to right
- Can use parentheses to specify join order
- Order usually doesn't affect results or performance (if outer joins are involved, results can definitely change)
- Can specify any condition (including nested subqueries) in `ON` clause

- Hints:
  - Use `ON` clause for join conditions
  - Use `WHERE` clause for selecting rows

33.9.7.3 Example Theta Join 3

- "Associate all courses with their prerequirements and also print the title of the prerequired course. Only show those courses where the credits are larger than the credits of the prerequired course"

```
%%sql
SELECT c.course_id, c.title, c.credits, p.prerequirement_id, c2.title, c2.credits
FROM course_s c
    INNER JOIN prerequirement_s p
        ON c.course_id = p.course_id
    INNER JOIN course c2
        ON p.prerequirement_id = c2.course_id
WHERE c.credits > c2.credits;
```

33.9.7.4 Example Cartesian Product

- "Prepare a list of all possible combinations of courses (course_id and title) with all prerequirements."

```
%%sql
SELECT c.course_id, c.title, p.*
FROM course_s c CROSS JOIN prerequirement_s p;
```

- This is the same as a theta join with no condition:

```
SELECT c.course_id, c.title, p.*
FROM course_s c INNER JOIN prerequirement_s p
ON TRUE;
```

- Or, simply

```
SELECT c.course_id, c.title, p.* FROM course_s c, prerequirement_s p;
```

```
%%sql
SELECT c.course_id, c.title, p.* FROM course_s c, prerequirement_s p;
```

33.9.8 Outer Joins

- An extension of the join operation that avoids loss of information.
- Result combines matching rows in both tables as for inner join
- If there is no match in the other table a "matching" row of empty fields in the other table is assumed
- `LEFT OUTER JOIN`
  - All qualified rows from the first (left) table remain in the result
  - a "matching" row of null-fields is used for the second (right) table, where required
- `RIGHT OUTER JOIN`
  - All qualified rows from the second (right) table remain in the result
  - a "matching" row of null-fields is used for the first (left) table, where required
- `FULL OUTER JOIN`

- All qualified rows from both tables remain in the result
- a "matching" row of null-fields is used for either table, where required

33.9.8.1 Examples

```
%%sql
SELECT *
FROM course_s LEFT OUTER JOIN prerequirement_s
ON course_s.course_id = prerequirement_s.course_id;
```

```
%%sql
SELECT *
FROM course_s RIGHT OUTER JOIN prerequirement_s
ON course_s.course_id = prerequirement_s.course_id;
```

```
%%sql
SELECT *
FROM course_s FULL OUTER JOIN prerequirement_s
ON course_s.course_id = prerequirement_s.course_id;
```

33.9.8.2 Outer Joins and Aggregates

- Outer joins can generate `NULL` values
- Aggregate functions ignore `NULL` values
  - Especially usefull for `COUNT`

33.9.8.2.1 Example

- "Find out how many directly prerequired courses each course has. Include courses with no prerequirements, show 0 for those courses. And show the courses in descending order of the number of prerequired courses."

```
%%sql
SELECT course_id, COUNT(prerequirement_id)
FROM course_s LEFT OUTER JOIN prerequirement_s
USING (course_id)
GROUP BY course_id
ORDER BY COUNT(prerequirement_id) DESC;
```

33.9.9 Common Attributes and Natural Joins

- `ON` syntax is clumsy for simple joins
  - Also, it's tempting to include conditions that should be in the `WHERE` clause
- Often, schemas are designed such that join columns have the same names
  - e.g. course.course_id and prerequirement.course_id
- `USING` clause is a simplified form of `ON` `sql SELECT * FROM t1 LEFT OUTER JOIN t2 USING (a1, a2, ...);`
- Equivalent to:

```
SELECT *
FROM t1 LEFT OUTER JOIN t2
ON (t1.a1 = t2.a1 AND t1.a2 = t2.a2 AND ...);
```

- `USING` also eliminates duplicate join attributes
- SQL natural join operation:

```
SELECT *
FROM t1 NATURAL INNER JOIN t2;
```

  - No `ON` or `USING` clause is specified
  - **All** common attributes are used in natural join operation

33.9.9.1 Examples

```
%%sql
SELECT *
FROM course_s FULL OUTER JOIN prerequirement_s
USING (course_id);
```

```
%%sql
SELECT *
FROM course_s NATURAL FULL OUTER JOIN prerequirement_s;
```

33.10 Set Operations

- SQL also provides set operations, like relational algebra
- Operations take two relations and produce an output relation

33.10.1 Syntax

```
<query expression>
{UNION | EXCEPT | INTERSECT}
[ ALL | DISTINCT ]
<query expression>
```

- Rules and restrictions
  - Identical number of columns
  - Pairwise type compatibility
  - First `SELECT` defines columns headings
- Paranthesis to control execution sequence

33.10.2 Semantics

- `UNION` Union of both partial result sets
- `INTERSECT` Intersection of both partial result sets
- `EXCEPT` Set-difference of both partial result sets

default for all three set operations is duplicate elimination

The optional keywords `ALL` and `DISTINCT` control the duplicate handling * `ALL` “keep all duplicates” * `DISTINCT` “eliminate duplicates”

Suppose a tuple occurs m times in r and n times in s, then, it occurs: * $m + n$ times in `r UNION ALL s` * $\min(m, n)$ times in `r INTERSECT ALL s` * $\max(0, m \urcorner n)$ times in `r EXCEPT ALL s`

33.10.3 Examples

- "Find courses that ran in Fall 2009 or in Spring 2010"

```
%%sql
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Fall' AND year = 2009
)
UNION
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Spring' AND year = 2010
)
```

- "Find courses that ran in Fall 2009 and in Spring 2010"

```
%%sql
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Fall' AND year = 2009
)
INTERSECT
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Spring' AND year = 2010
)
```

- "Find courses that ran in Fall 2009 but not in Spring 2010"

```
%%sql
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Fall' AND year = 2009
)
EXCEPT
(
    SELECT course_id, semester, year
    FROM section
    WHERE semester = 'Spring' AND year = 2010
)
```

33.11 With Clause (Common Table Expression)

The `WITH` clause provides a way of defining a temporary view whose definition is available only to the query in which the `WITH` clause occurs.

- Very useful for writing complex queries
- Supported by most database systems, with minor syntax variations

33.11.1 Examples

- "Find all departments with the maximum budget"

```
%%sql
WITH max_budget(value) AS (
    SELECT max(budget) FROM department
)
SELECT dept_name, budget
FROM department, max_budget
WHERE department.budget = max_budget.value;
```

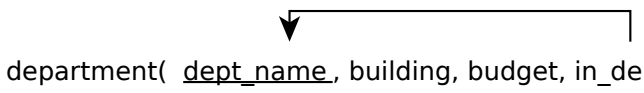
- "Find all departments where the total salary is greater than the average of the total salary of all departments."

```
%%sql
WITH
dept_total(dept_name, value) AS (
    SELECT dept_name, sum(salary)
    FROM instructor
    GROUP BY dept_name
),
dept_total_avg(value) AS (
    SELECT avg(value)
    FROM dept_total
)
SELECT dept_name, dept_total.value AS total, dept_total_avg.value AS total_avg
FROM dept_total, dept_total_avg
WHERE dept_total.value >= dept_total_avg.value;
```

33.11.2 Recursive Queries

- Tree structures are common in almost every application domain
- The most simple relational representation of a tree structure is an optional reference (foreign key) to the next higher node in every non-root node

33.11.2.1 Example, Tree



- With this representation a complete tree traversal requires an n-fold self-join
  - n = number of levels in tree
  - n usually unknown upfront

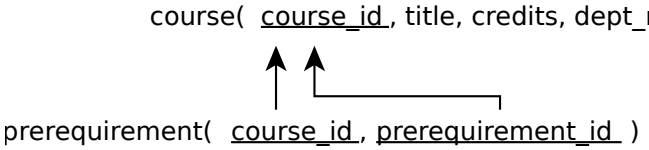
- Alternative:
  - Recursive Query

"Show the complete organization tree structure, displaying tree level, department name, starting with the whole university (no value for column in_dept)"

```
%%sql
WITH RECURSIVE v_dept(dept_name,depth, path, cycle) AS (
    SELECT d.dept_name, 1, ARRAY[ROW(d.dept_name)], FALSE
    FROM department d
    WHERE in_dept IS NULL
    UNION ALL
    SELECT d.dept_name, depth + 1, path || ROW(d.dept_name), ROW(d.dept_name) = ANY(path)
    FROM department d INNER JOIN v_dept
    ON d.in_dept = v_dept.dept_name AND NOT cycle
)
SELECT depth, dept_name, path
FROM v_dept
ORDER BY path;
```

33.11.2.2 Example, Graph

- The concept of tree traversals can be easily extended to general graphs



"Traverse the complete structure of prerequired courses."

```
%%sql
WITH RECURSIVE search_graph(course_id,depth,path,cycle) AS (
    SELECT p.course_id, 1, ARRAY[ROW(p.course_id, p.prerequirement_id)], false
    FROM prerequirement p
    UNION ALL
    SELECT p.prerequirement_id, sg.depth + 1, path || ROW(p.course_id, p.prerequirement_id),
    FROM prerequirement p, search_graph sg
    WHERE p.course_id = sg.course_id AND NOT cycle
)
SELECT sg.*, c.title
FROM search_graph sg, course c
WHERE sg.course_id = c.course_id
ORDER BY path;
```

34 Data Definition Language (DDL)

- Specify relation schemas (attributes, domains)
- Specify a variety of integrity constraints
- Access constraints on data
- Indexes and other storage "hints" for performance

35 SQL Attribute Domains

35.1 Some Standard SQL domain types

- `CHAR(N)`
  - A character field, fixed at `N` characters wide
  - Short for `CHARACTER(N)`
- `VARCHAR(N)`
  - A variable-width character field, with maximum length `N`
  - Short for `CHARACTER VARYING(N)`
- `INT`
  - A signed integer field (typically 32 bits)
  - Short for `INTEGER`
  - Also `TINYINT`, `SMALLINT`, `BIGINT`, etc.
  - Also unsigned variants
    - Non-standard, only supported by some vendors
- `NUMERIC(P,D)`
  - A fixed-point number with user-specified precision
  - P total digits; D digits to right of decimal place
  - Can exactly store numbers
- `DOUBLE PRECISION`
  - A double-precision floating-point value
  - An approximation! Don't use for money!
  - `REAL` is sometimes a synonym
- `FLOAT(N)`
  - A floating-point value with at least N bits of precision
- `DATE`, `TIME`, `TIMESTAMP`
  - For storing temporal data
- `BLOB`, `CLOB`, `TEXT`
  - Large binary/text data fields

35.1.1 `CHAR` vs. `VARCHAR`

- Both `CHAR` and `VARCHAR` have a size limit
- `CHAR` is a fixed-length character field
  - Can store shorter strings, but storage layer pads out the value to the full size
- `VARCHAR` is a variable-length character field
  - Storage layer doesn't pad out shorter strings
  - String's length must also be stored for each value
- Use `CHAR` when all values are approximately (or exactly) the same length
- Use `VARCHAR` when values can be very different lengths

35.2 User-defined Data Types (Domains)

In SQL a **domain** is a user-defined data type

35.2.1 Defining a new Domain

35.2.1.1 Syntax

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

where constraint is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

35.2.1.2 Semantics

- `name`: The name of a domain to be created.
- `data_type`: The underlying data type of the domain.
- `collation`: An optional collation for the domain.
- `DEFAULT expression`: The value is any variable-free expression.
  - date and time
    - `CURRENT_DATE`
    - `CURRENT_TIME [<sec_precision>]`
    - `CURRENT_TIMESTAMP [<sec_precision>]`
  - user name
    - `USER`: author of domain definition
    - `CURRENT_USER`: author of executed code
    - `SESSION_USER`: user who has logged in

- `CONSTRAINT constraint_name`: An optional name for a constraint.
  - `NOT NULL`, `NULL`: Values of this domain are prevented from being null.
  - `CHECK (expression)`: `CHECK` clauses specify integrity constraints or tests which values of the domain must satisfy.

35.2.1.3 Example

- Our database contains quite a lot of 10-digit ID numbers. It would make sense to define a domain for this purpose

```
%reload_ext sql
%sql postgresql://universitydb:universitydb@127.0.0.1:65432/university
%config SqlMagic.feedback = True
%config SqlMagic.style = 'DEFAULT'
%config SqlMagic.short_errors=True
```

```
%%sql
CREATE DOMAIN idno AS DECIMAL(10)
CONSTRAINT positive_idno CHECK (VALUE > 0);
```

35.2.2 Changing a Domain

- No modification of the base type!
- Changing/removing the default value

  ```
  ALTER DOMAIN name
  { SET DEFAULT expression | DROP DEFAULT }
  ```

- Modifying constraints

```
ALTER DOMAIN name
    { SET | DROP } NOT NULL
```

  - Removing an integrity constraint

```
ALTER DOMAIN name
DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
```

  - Add an integrity constraint

```
ALTER DOMAIN name
ADD domain_constraint [ NOT VALID ]
```

  - This form validates a constraint previously added as `NOT VALID`

```
ALTER DOMAIN name
VALIDATE CONSTRAINT constraint_name
```

  - Rename an integrity constraint

```
ALTER DOMAIN name
 RENAME CONSTRAINT constraint_name TO new_constraint_name
```

35.2.3 Removing a Domain

35.2.3.1 Syntax

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

35.2.3.2 Semantics

- `CASCADE`: removes all references to the domain as well
  - if you love the risk
- `RESTRICT`: raises an exception, if references to the domain exist

35.2.3.3 Example

```
%%sql
DROP DOMAIN idno;
```

35.3 Choosing the Right Type

- Need to think carefully about what type makes most sense for your data values
- Example 1: storing ZIP codes
  - US postal codes for mail routing
  - 5 digits
- Does `INTEGER` make sense?
  - Problems:
    - Some ZIP codes have leading zeroes!
    - US mail also uses ZIP+4 expanded ZIP codes
    - Many foreign countries use non-numeric values
- Example 2: monetary amounts
- Does `FLOAT` make sense?

  - Problems:
    * Floating-point representations cannot exactly represent all values
    * e.g. 0.1 is an infinitely-repeating binary decimal value
  - Use `NUMERIC`

36 Tables

36.1 Defining a new Table

- Table and column names must follow specific rules
  - Table must have a unique name within schema
  - All columns must have unique names within the table

36.1.1 Syntax

```
CREATE [TEMPORARY] TABLE table_name ([
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
```

where column_constraint is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and table_constraint is:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

36.1.2 Semantics

- `TEMPORARY`: Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT` below).
- `table_name`: The name of the table to be created.
- `column_name`: The name of a column to be created in the new table.

- `data_type`: The data type of the column.
- `COLLATE collation`: An optional collation for the column.
- `CONSTRAINT constraint_name:` An optional name for a column or table constraint.
  - `NOT NULL`: The column is not allowed to contain null values.
  - `NULL`: The column is allowed to contain null values.
  - `CHECK`: The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed.
  - `DEFAULT default_expr`: Assigns a default data value for the column whose column definition it appears within.
  - `UNIQUE (column constraint)` or `UNIQUE ( column_name [, ... ] )` (table constraint): Specifies that a group of one or more columns of a table can contain only unique values.
  - `PRIMARY KEY (column constraint)` or `PRIMARY KEY ( column_name [, ... ] )` (table constraint): Specifies that a column or columns of a table can contain only unique (non-duplicate), nonnull values.
  - `REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ] [ ON UPDATE action ]` (column constraint)
    or
    `FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ ON DELETE action ] [ ON UPDATE action ]` (table constraint): Specifies a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table.
- `action`:
  - specifies the actions when referenced rows are deleted or updated
  - `NO ACTION`: Produce an error indicating that the deletion or update would create a foreign key constraint violation. **default**
  - `RESTRICT`: Produce an error indicating that the deletion or update would create a foreign key constraint violation.
  - `CASCADE`: Delete or update any rows referencing the deleted, updated row.
  - `SET NULL`: Set the referencing column(s) to null.
  - `SET DEFAULT`: Set the referencing column(s) to their default values.
- `[ DEFERRABLE | NOT DEFERRABLE ]`: Controls whether the constraint can be deferred (Postponed until the end of the transaction).

- `[ INITIALLY IMMEDIATE | INITIALLY DEFERRED ]`: If a constraint is deferrable, this clause specifies the default time to check the constraint.

36.1.3 Example

```
%%sql
DROP TABLE instructor1;
CREATE TABLE instructor1 (
    instructor_id VARCHAR(5),
    name VARCHAR(20) NOT NULL,
    dept_name VARCHAR(20),
    salary NUMERIC(8,2) CHECK (salary > 0),
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department ON DELETE SET NULL
);
```

36.2 Table Constraints

- By default, SQL tables have no constraints
  - Can insert multiple copies of a given row
  - Can insert rows with NULL values in any column

36.2.1 Primary Key Constraint

- Specify columns that comprise primary key
  - e.g. `PRIMARY KEY (instructor_id)`
  - No two rows can have same values for primary key
  - A table can have only one primary key
  - The primary key can never contain `NULL` as values

36.2.2 Null-Value Constraints

- Every attribute domain contains `NULL` by default
- Often, `NULL` is not an acceptable value!
  - e.g. instructors must **always** have a name
- Can specify `NOT NULL` to exclude `NULL` values for particular columns
  - e.g. `name VARCHAR(20) NOT NULL`

36.2.3 Other Candidate Keys

- Some relations have multiple candidate keys
- Can specify candidate keys with `UNIQUE` constraints
  - Unlike primary keys, `UNIQUE` constraints do not exclude `NULL` values!

36.2.3.1 Example

- `instructor_ssn CHAR(9) NOT NULL UNIQUE`

```
%%sql
DROP TABLE instructor2;
CREATE TABLE instructor2 (
    instructor_id VARCHAR(5),
    name VARCHAR(20) NOT NULL,
    dept_name VARCHAR(20),
    salary NUMERIC(8,2) CHECK (salary > 0),
    instructor_ssn CHAR(9) NOT NULL UNIQUE,
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department ON DELETE SET NULL
);
```

36.2.3.2 UNIQUE and NULL

- Example

```
%%sql
DROP TABLE customer;
CREATE TABLE customer (
    cust_name VARCHAR(30) NOT NULL,
    address VARCHAR(60), UNIQUE (cust_name, address)
);
```

- Try inserting values:

```
%%sql
INSERT INTO customer VALUES ('John Doe', '123 Spring Lane');
INSERT INTO customer VALUES ('John Doe', '123 Spring Lane');
```

- Try inserting more values:

```
%%sql
INSERT INTO customer VALUES ('Jane Doe', NULL);
INSERT INTO customer VALUES ('Jane Doe', NULL);
```

- **Be careful using nullable columns in `UNIQUE` constraints!**

36.2.4 CHECK Constraints

- Constraints on values
- Can require values in a table to satisfy some predicate, using a `CHECK` constraint
- Must appear after the column specifications

36.2.4.1 Examples

- `salary NUMERIC(8,2) CHECK (salary > 0)`
  - Ensures that all instructors have a positive salary.
- `semester varchar(6) check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))`
  - Ensures that semester must be one of the specified values

```
CREATE TABLE instructor (
    instructor_id VARCHAR(5),
    name VARCHAR(20) NOT NULL,
    dept_name VARCHAR(20),
    salary NUMERIC(8,2) CHECK (salary > 0),
    instructor_ssn CHAR(9) NOT NULL UNIQUE,
    PRIMARY KEY (instructor_id),
    CHECK (dept_name IN (SELECT dept_name FROM department))
);
```

- Rows in the instructor table should only contain valid department names.
  - This is a **referential** integrity constraint.
- Usage:
  - Easy to write very expensive `CHECK` constraints
  - Lack of widespread support; using them limits portability
  - When used, they are usually very simple
  - Guided by the laws of logics and physics
  - Don't implement business processes this way

36.3 Referential Integrity Constraints, Foreign Key Constraints

- Referential integrity constraints are very important!
  - These constraints span multiple tables
  - Allow us to associate data across multiple tables
  - One table's values are constrained by another table's values
- A relation can also include attributes of another relation's primary key
  - Referencing relation's values for the foreign key must also appear in the referenced relation

36.3.1 Example

```
%%sql
DROP TABLE IF EXISTS instructor2;
CREATE TABLE IF NOT EXISTS instructor2 (
    instructor_id VARCHAR(5),
    name VARCHAR(20) NOT NULL,
    dept_name VARCHAR(20) REFERENCES department,
    salary NUMERIC(8,2) CHECK (salary > 0),
    instructor_ssn CHAR(9) NOT NULL UNIQUE,
    PRIMARY KEY (instructor_id)
);
```

- Foreign-key constraint does `NOT` imply `NOT NULL`!
- Can also specify the column in the referenced relation
  - Especially useful when referenced column is a candidate key, but not the primary key
  - e.g. `FOREIGN KEY (dept_name) REFERENCES department(dept_name)`

36.3.2 Multi-Column Foreign Keys

- Multi-column foreign keys can also be affected by `NULL` values
  - Individual columns may allow `NULL` values
- If all values in foreign key are non-`NULL` then the foreign key constraint is enforced
  - e.g. In the table `section`: `FOREIGN KEY (building, room_number) REFERENCES classroom`

```
%%sql
DELETE FROM section WHERE course_id = 'BIO-101' AND sec_id = '1' AND semester = 'Summer' AND
DELETE FROM section WHERE course_id = 'BIO-101' AND sec_id = '2' AND semester = 'Summer' AND
DELETE FROM section WHERE course_id = 'BIO-101' AND sec_id = '3' AND semester = 'Summer' AND

INSERT INTO section VALUES ('BIO-101','1', 'Summer', 1900, 'Packard', '101', 'Mo', NULL, NUL
INSERT INTO section VALUES ('BIO-101','2', 'Summer', 1900, NULL, '101', 'Mo', NULL, NULL);
INSERT INTO section VALUES ('BIO-101','3', 'Summer', 1900, 'Packard', NULL, 'Mo', NULL, NULL
INSERT INTO section VALUES ('BIO-101','4', 'Summer', 1900, 'Packard', '102', 'Mo', NULL, NUL
```

36.3.3 Foreign Key Violations

- If referencing relation gets a bad foreign-key value, the operation is simply forbidden
  - e.g.trying to insert a row into `instructor` relation, where the row contains an invalid department name

  - e.g.tyring to update a row in `instructor`, setting the department name to an invalid value.
- More complicated when the **reference** relation is changed.
  - What happens with `instructor` when a row is deleted from `department`?
- Option 1: Disallow the delete from `department`
  - Default for SQL
- Option 2: **Cascade** the delete operation
  - If user deletes a department, all referencing rows in `instructor` should also be deleted.
- Option 3: Set foreign key value to `NULL`
  - Doesn't make sense in every situation (think of relationships with lower cardinality limit ≥ 0)
- Option 4: Set foreing key to some default value

36.3.3.1 Example, Option 1

```
%%sql
DROP TABLE IF EXISTS instructor2;
DROP TABLE IF EXISTS department1;

CREATE TABLE IF NOT EXISTS department1 (
    dept_name VARCHAR(20),
    PRIMARY KEY (dept_name)
);

INSERT INTO department1 VALUES ('History');


CREATE TABLE IF NOT EXISTS instructor2 (
    instructor_id VARCHAR(5),
    dept_name VARCHAR(20),
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department1 (dept_name)
);

INSERT INTO instructor2 VALUES ('a','History');

DELETE FROM department1 WHERE dept_name = 'History';
```

36.3.3.2 Example, Option 2

```
%%sql
DROP TABLE IF EXISTS instructor2;
DROP TABLE IF EXISTS department1;

CREATE TABLE IF NOT EXISTS department1 (
    dept_name VARCHAR(20),
    PRIMARY KEY (dept_name)
);

INSERT INTO department1 VALUES ('History');
```

```
CREATE TABLE IF NOT EXISTS instructor2 (
    instructor_id VARCHAR(5),
    dept_name VARCHAR(20),
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department1 (dept_name) ON DELETE CASCADE
);

INSERT INTO instructor2 VALUES ('a','History');

DELETE FROM department1 WHERE dept_name = 'History';

SELECT * FROM instructor2;
```

36.3.3.3 Example, Option 3

```
%%sql
DROP TABLE IF EXISTS instructor2;
DROP TABLE IF EXISTS department1;

CREATE TABLE IF NOT EXISTS department1 (
    dept_name VARCHAR(20),
    PRIMARY KEY (dept_name)
);

INSERT INTO department1 VALUES ('History');


CREATE TABLE IF NOT EXISTS instructor2 (
    instructor_id VARCHAR(5),
    dept_name VARCHAR(20),
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department1 (dept_name) ON DELETE SET NULL
);

INSERT INTO instructor2 VALUES ('a','History');

DELETE FROM department1 WHERE dept_name = 'History';

SELECT * FROM instructor2;
```

36.3.3.4 Example, Option 4

```
%%sql
DROP TABLE IF EXISTS instructor2;
DROP TABLE IF EXISTS department1;

CREATE TABLE IF NOT EXISTS department1 (
    dept_name VARCHAR(20),
    PRIMARY KEY (dept_name)
);
```

```
INSERT INTO department1 VALUES ('History');
INSERT INTO department1 VALUES ('Def');

CREATE TABLE IF NOT EXISTS instructor2 (
    instructor_id VARCHAR(5),
    dept_name VARCHAR(20) DEFAULT 'Def',
    PRIMARY KEY (instructor_id),
    FOREIGN KEY (dept_name) REFERENCES department1 (dept_name) ON DELETE SET DEFAULT
);

INSERT INTO instructor2 VALUES ('a', 'History');
DELETE FROM department1 WHERE dept_name = 'History';
SELECT * FROM instructor2;
```

36.4 Changing a Table

- Adding a new column

  ```
  ALTER TABLE [ IF EXISTS ] name
  ADD COLUMN [ IF NOT EXISTS ] column_def
  ```

- Changing the default value of a column

  ```
  ALTER TABLE [ IF EXISTS ] name
  ALTER [COLUMN] column_name (SET DEFAULT expression | DROP DEFAULT)
  ```

- Removing a column

  ```
  ALTER TABLE [ IF EXISTS ] name
  DROP [COLUMN] column_name [CASCADE | RESTRICT]
  ```

  - `CASCADE` removes all references to the removed column
  - `RESTRICT` raises an exception, if the removed column is referenced

- Adding an integrity constraint

  ```
  ALTER TABLE [ IF EXISTS ] name
  ADD table_constraint
  ```

- Removing an integrity constraint

  ```
  ALTER TABLE [ IF EXISTS ] name
  DROP CONSTRAINT constraint_name
  ```

  - We cannot remove anonymous integrity constraints
    - integrity constraints with no names are unprofessional!

36.5 Removing a Table

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

- `CASCADE` removes all references to the removed table automatically
- `RESTRICT` raises an exception, if references to the removed table exist
- Specifying neither of them avoids all checking
  - You might replace the dropped table by another one having the same name

  - Would you consider this safe?

37 Views

Views: * Virtual table * kind of table, the data content of which is derived from other tables and views * therefore dependent on these other tables and views * A **view** provides a mechanism to hide certain data from the view of certain users.

- Two main reasons for using views:
  - Performance and convenience
    - Define a view for a widely used derived relation
    - DBMS automatically computes view's contents when it is used in a query
    - **Materialized view**, result is pre-computed and stored on disk
  - Security!
    - In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
    - e.g. hide private information on employees

37.1 Defining a new View

```
CREATE VIEW name [ ( column_name [, ...] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- **name**: The name of a view to be created.
- **column_name**: An optional list of names to be used for columns of the view.
- **query**: A command which will provide the columns and rows of the view.
- `WITH [ CASCADED | LOCAL ] CHECK OPTION` controls the behavior of automatically updatable views.
  - If specified checks will be performed to ensure that new rows satisfy the view-defining condition.
  - `LOCAL` ... new rows are only checked against the conditions directly in the view itself.
  - `CASCADE` ... new rows are also checked against the conditions in all underlying base views.

37.1.1 Examples

- A view of instructors without their salary

```
%%sql
DROP VIEW IF EXISTS faculty;

CREATE VIEW faculty AS
SELECT instructor_id, name, dept_name
FROM instructor;

SELECT * FROM faculty;
```

- Find all instructors in the Biology department

```
%%sql
DROP VIEW IF EXISTS biology_dept_instructors;

CREATE VIEW biology_dept_instructors AS
SELECT name
FROM faculty
WHERE dept_name = 'Biology';

SELECT * FROM biology_dept_instructors;
```

- Create a view of department salary totals

```
%%sql
DROP VIEW IF EXISTS departments_total_salary;

CREATE VIEW departments_total_salary(dept_name, total_salary) AS
SELECT dept_name, SUM(salary)
FROM instructor
GROUP BY dept_name;

SELECT * FROM departments_total_salary;
```

37.2 Nesting of Views

- The `FROM`-clause of a `SELECT`-statement may contain tables and views optionally connected using join-expressions

37.2.1 Example

Our controllers would like a list containing the following information: * department name * budget * fix planned expenses consisting of * sum of budgets of all its subordinate departments * sum of annual salaries of all its instructors

```
%%sql
DROP VIEW IF EXISTS dept_finance_status;
DROP VIEW IF EXISTS dept_subbudget;
DROP VIEW IF EXISTS dept_salary;

CREATE VIEW dept_subbudget (dept_name, budget_sum) AS
SELECT top.dept_name, SUM(COALESCE(bottom.budget, 0))
FROM department top
LEFT OUTER JOIN department bottom ON top.dept_name = bottom.in_dept
GROUP BY top.dept_name;

CREATE VIEW dept_salary (dept_name, salary_sum) AS
SELECT dept.dept_name, SUM(COALESCE(inst.salary, 0))
FROM department dept
LEFT OUTER JOIN instructor inst ON dept.dept_name = inst.dept_name
GROUP BY dept.dept_name;

CREATE VIEW dept_finance_status (dept_name, budget, planned) AS
SELECT dept_name, budget, subbud.budget_sum + subsal.salary_sum
```

```
FROM department NATURAL JOIN dept_subbudget subbud NATURAL JOIN dept_salary subsal;

SELECT * FROM dept_finance_status;
```

37.3 Changing a View

A view cannot be changed

Instead: * drop the out-dated view * create the desired new view definition

This works best if you don't use `CASCADE` or `RESTRICT`

37.4 Removing a View

```
DROP VIEW name [, ...] [ CASCADE | RESTRICT ]
```

- **name**: The name of the view to remove.
- `CASCADE`: Automatically drop objects that depend on the view (such as other views)
- `RESTRICT`: Refuse to drop the view if any objects depend on it. This is the default.

37.4.1 Example

Remove the created views.

```
%%sql
DROP VIEW dept_finance_status;
DROP VIEW dept_subbudget;
DROP VIEW dept_salary;
```

38 Global Integrity Constraints

Assertion * global integrity constraint * concept not yet supported by Oracle and postgres * Alternative: Trigger (SQL-99)

38.1 Defining a new Global Integrity Constraint

```
CREATE ASSERTION name CHECK (expression) [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRE
```

- **name**: The name of the assertion to create.
- **expression**: no restrictions
  - very often nested queries
  - often based on views defined for this pupose only

38.1.1 Example

No department may allocate more than its budget for salaries budgets of subordinate departments.

```
CREATE ASSERTION budget_constraint
CHECK (0 = (SELECT COUNT(*) FROM dept_finance_status WHERE budget < planned))
DEFFERABLE INITIALLY IMMEDIATE;
```

38.2 Changing a Global Integrity Constraint

An assertion cannot be changed * Instead: * drop the out-dated assertion * create the desired new assertion * no dependences between assertions * the checking may take considerable time

38.3 Removing a Global Integrity Constraint

```
DROP ASSERTION name
```

- **name**: The name of the assertion to remove.

```
%%sql
SELECT COUNT(*) FROM dept_finance_status WHERE budget < planned;
```

```
%%sql
DROP FUNCTION IF EXISTS checkBudget() CASCADE;

CREATE FUNCTION checkBudget() RETURNS TRIGGER AS '
BEGIN
IF (0 < (SELECT COUNT(*) FROM dept_finance_status WHERE budget < planned))
THEN RAISE EXCEPTION ''budget violated'';
END IF;
RETURN NULL;
END
'
LANGUAGE plpgsql;

CREATE TRIGGER checkBudgetTrigger
AFTER UPDATE ON department
FOR EACH ROW
EXECUTE PROCEDURE checkBudget();

SELECT * FROM department;

UPDATE department
SET budget = 10
WHERE dept_name = 'Technics';
```

39 Authorization

39.1 Allowing Access

```
GRANT {ALL [ PRIVILEGES ] | action, [,action ...]}
ON db_object
TO role_specification [WITH GRANT OPTION]
```

- **action**
  - SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | CONNECT (column_name [, ...])
    - column_names must exists
- **db_object**

  - `TABLE | DATABASE | VIEW | FUNCTION | SCHEMA |...`
- **role_specification**
  - `[ GROUP ] role_name | PUBLIC | CURRENT_USER | SESSION_USER`
- Semantics
  - `WITH GRANT OPTION` allows to forward the received authorization to others
  - Not all possible combinations of `action` and `db_object` make sense

39.2 Disallowing Access

```
REVOKE [ GRANT OPTION FOR ] {ALL [ PRIVILEGES ] | action, [,action ...]}
ON db_object
FROM role_specification [ CASCADE | RESTRICT ]
```

- Semantics
  - Not all possible combinations of `action` and `db_object` make sense
  - `CASCADE`
    to revoke forwarded authorizations as well
    GRANT OPTION
  - `RESTRICT`
    to raise an exception if authorization to be revoked has been forwarded further

40 Indices

40.1 Motivation

- optimal support of applications
  - fast data access
  - no dependences between applications unless logically required
  - protection against data loss
- together with efficient use of resources
  - disk space
  - main memory
  - network bandwidth
  - processor time
  - elapsed time (user time)

40.2 Access Paths

- E-R-Methodology tells us about
  - Candidate keys
  - Foreign keys
- Doesn't tell us anything about
  - Additional access paths (application requirements)
    - not model dependent
    - such as "names"

40.2.1 Observations

- Access paths may improve performance

- Access paths may deteriorate performance not only when writing

40.2.2 Characteristics of Keys

- Length
- Datatype (incl. struct)
- Selectivity

40.2.3 Performance

1. Object Identifier
   - Oracle: ROWID
   - db2: TID
   - postgres: 'CTID'
2. Unique single-column key
3. Unique multi-column key
4. Non-unique single-column key
5. Non-unique multi-column key

The shorter the better!

40.2.4 Realization

- All primary keys
  - usually a good choice (shortest, complete and unique key)
  - some DBMSs will automatically add a primary key to every table, which doesn't have one
- Usually all secondary keys
  - that is all other candidate keys
  - absolute requirement for those protected by UNIQUE-constraints
- Usually all foreign keys
  - that is references to other keys (not only primary keys)
- Typically on entry points (such as "names")
- Exceptions
  - Very small tables (< 10 blocks)
  - Columns with high update frequency
  - Columns with low selectivity
  - Overlapping access path exists

40.3 SQL Index

- Realization of an access path
- Set of tuples
  - (key, pointer to row in table)
  - just like the index in a book
- What is it good for?
  Improving the search performance
- Is it for free?
  - No

    - disk space
    - writing effort
  - Bear in mind
    every data field is read 10 times more often than written on average

40.4 Defining an Index

```
CREATE [ UNIQUE ] INDEX  name
ON table_name ( { column_name | ( expression ) }
```

Recommendation: You should have a uniqueness constraint on the indexed columns of a unique index.

- Many product specific extensions to define
  - data organisation form
  - function index
  - index order
    - ascending/descending
    - reversed key (bytewise, bitwise)
  - partial index
  - etc.

40.4.1 Example

Let's reduce the search time for departments when using the department name

```
%%sql
DROP INDEX IF EXISTS department_name_index;

CREATE INDEX department_name_index ON department (dept_name);
```

40.5 Changing an Index

Depends on the DBMS used, indices are not part of the SQL standard.

40.6 Removing an Index

```
DROP INDEX name
```

41 Motivation

41.1 The value of a database?

Lower limit for reasons of economy:

- Cost for setting up, running, and maintaining it
- Significant investment

41.2 Integrity threats

- Incorrect / incomplete data modification caused by
  - User mistakes
  - Application bugs
  - Interleaved modifications
- Deadlocks
- System crashes
- Media failures
- etc.

41.3 Consequences of integrity problems

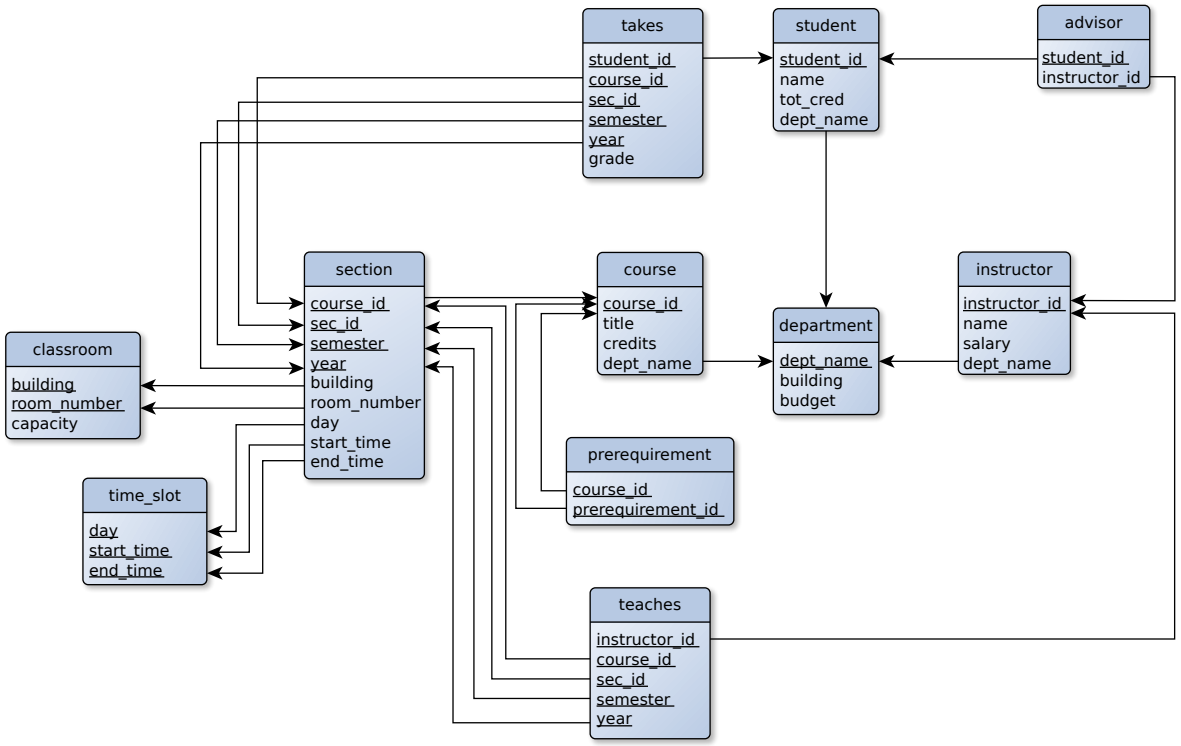
- Wrong decisions
  - Regarding fortune or life
  - Loss of reputation
- Loss of opportunities
- Even the right answer is wrong, if it is late
- Loss of trust
- Loss of financial transaction data
- etc.

42 Logical Integrity

42.1 Referential Integrity

42.1.1 In general

A field value may be a unique or non-unique reference to rows containing the same value in some field ### In particular
The target, to which a foreign key refers, shall always exist ### Violation reactions * Exception: To be handled by the application * Update propagation: Potentially dangerous * Cascading delete * Very powerful * Extremely dangerous



42.2 General Data Integrity

42.2.1 Integrity constraints definitions

- on various levels
  - Column
    `budget > 0`
  - Every row in table
    `saldo + creditlimit >= 0`
  - Whole table
    `sum(budget) < 1000000.-`
  - Whole database
- Independent of operation
  See previous examples
- Dependent of operation
  - `[new]salary > [old]salary`
  - `set name IF NULL`
  - `overwrite salary IF NOT NULL`
  - `insert with name IS NULL`
- For temporary violations
  See transactions

42.3 Application Considerations

- Integrity constraints on the conceptual / database schema level are dangerous although sometimes necessary

  - Their lack can turn an expensive database into a useless data dump
  - Should only reflect the laws of logics and nature, not the rules of one's business
    Otherwise required adaptations in the way business is done can become hard if not impossible
  - Often highly over-used
- Most integrity constraints belong in the external schema / are application specific

43 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

44 Motivation

44.1 Problem

- Not every consistency-preserving modification of a database can be expressed with a single SQL statement
  Temporary inconsistent state
- Many situations where a sequence of database operations must be treated as a **single unit**
  - A combination of reads and writes that must be performed "as a unit"
  - If any operation doesn't succeed, all operations in the unit of work should be rolled back

44.2 Example

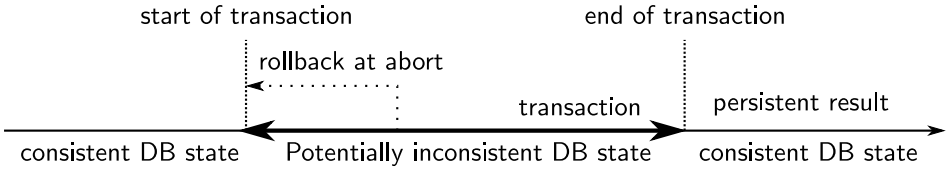
- Transfer of 5000.- from account 4711 to account 1174 "'

1) read(4711)
2) 4711 := 4711 - 5000
3) write(4711)
4) read(1174)
5) 1174 := 1174 + 5000
6) write(1174) "'

- We need to violate temporarily the rule, that the sum of all accounts never changes. Independent of our sequence of operations

44.3 Transaction Definition

- A **transaction** is a unit of program execution that accesses and possibly updates various data items, leading from a consistent state to a consistent state.
  - During a transaction the database may be in an inconsistent state.
- An **essential** database feature for the correct implementation of tasks that could fail.
- Also essential for databases with concurrent access!



45 ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j , finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

46 Example

- Transfer of 5000.- from account 4711 to account 1174 “‘

1) read(4711)
2) 4711 := 4711 - 5000
3) write(4711)
4) read(1174)
5) 1174 := 1174 + 5000
6) write(1174) “‘

- **Atomicity requirement**
  - fail after 3 and before 6
    - money “lost”
    - cause: software or hardware
  - system should ensure that update of partially executed transactions are not persisted
- **Durability requirement**
  - after transaction completion, DB updates must persist even if there are software or hardware failures
- **Consistency requirement**
  - total amount on both accounts is unchanged by the execution of the transaction
- **Isolation requirement**
  - If between steps 3 and 6, another transaction is allowed to access the partially updated DB, it will se an inconsistent database “‘

  1) T1: read(4711)
  2) T1: 4711 := 4711 - 5000
  3) T1: write(4711) T2: read(4711), read(1174), print(4711 + 1174)
  4) T1: read(1174)

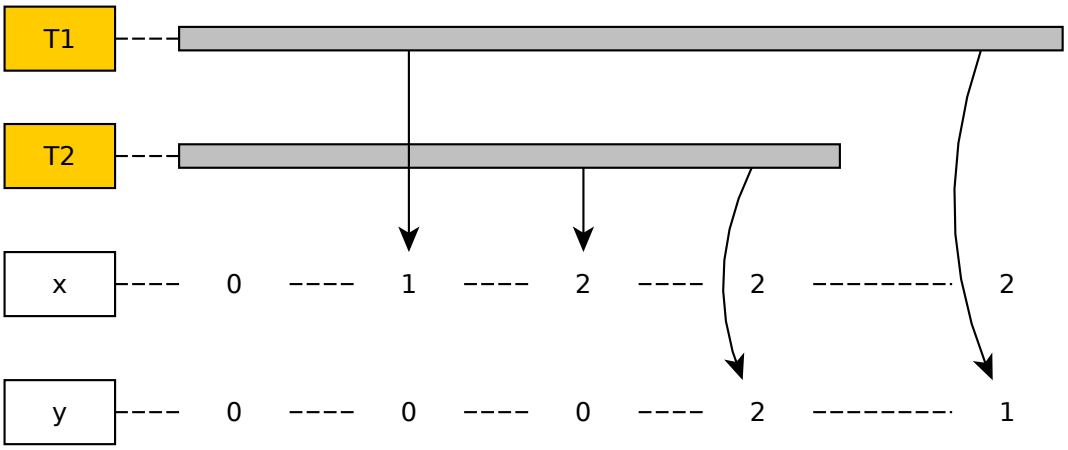
5) T1: 1174 := 1174 + 5000
6) T1: write(1174) “‘
– Can be achieved by **serial** execution

47 Concurrent Transaction Issues

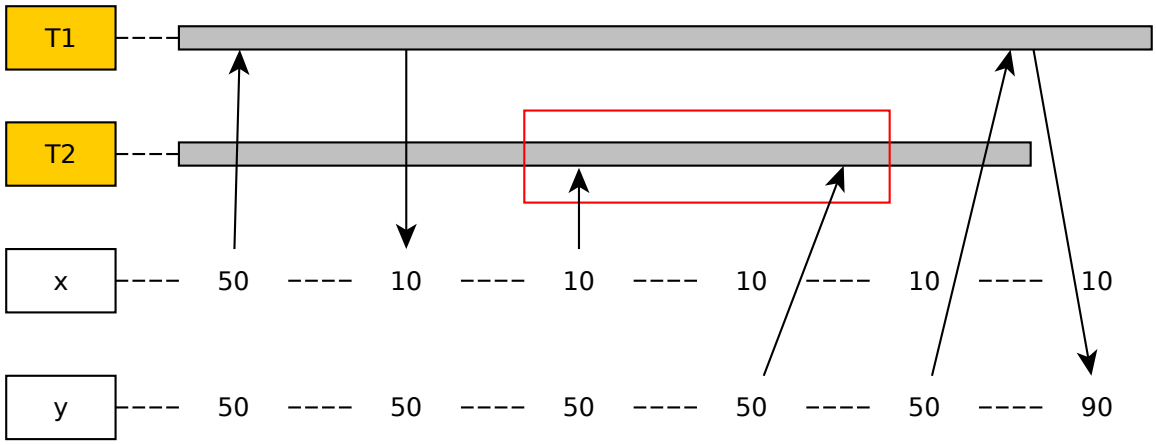
- If database has multiple client connections at a time **isolation** becomes important.
  - Without proper transaction isolation, the database would quickly become corrupt
- Advantages of concurrent transactions are:
  - **increased processor and disk utilization**, leading to better transaction throughput
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.

47.1 Problems with Simultaneous Access

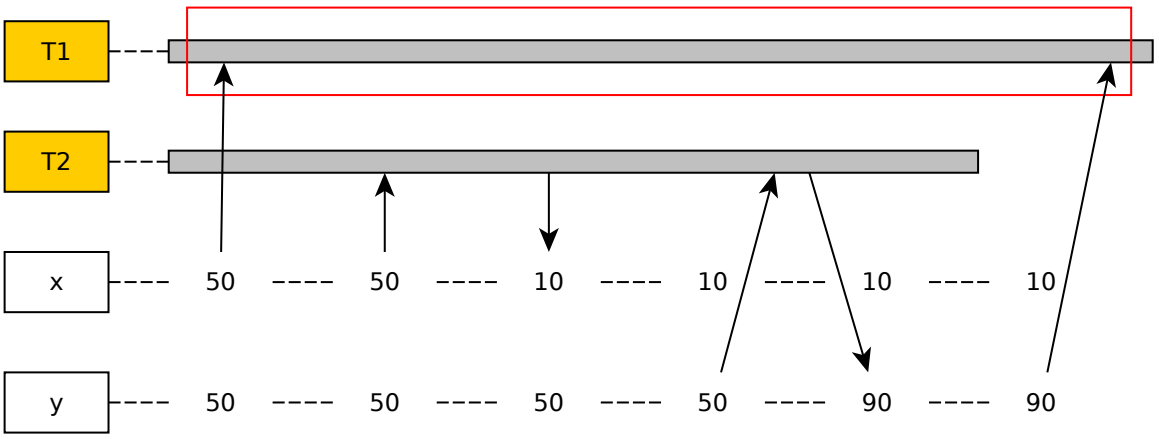
- Dirty writes
  - “A **dirty write** occurs when one transaction overwrites a value that has previously been written by another still in-flight transaction.”
  - Example: constraint $x = y$



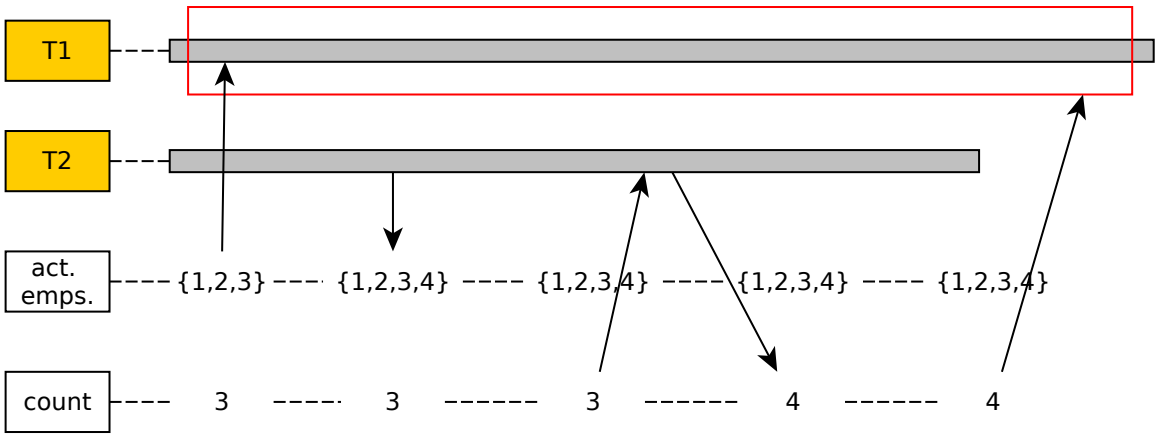
- Dirty reads (write–read conflict)
  - “A **dirty read** occurs when one transaction reads a value that has been written by another still in-flight transaction.”
  - Example: transfer 40, constraint $x + y = 100$



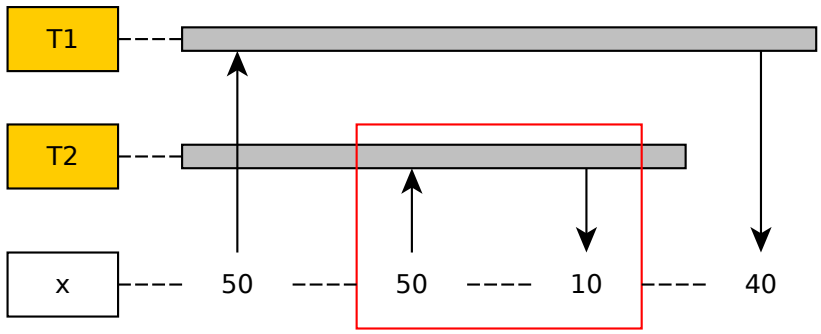
- Non-reapeatable reads (read–write conflict, fuzzy read)
  - "A **fuzzy** or **non-repeatable read** occurs when a value that has been read by a still in-flight transaction is overwritten by another transaction."
  - Example: transfer 40, constraint $x + y = 100$



- Phantom rows
  - "A **phantom** occurs when a transaction does a predicate-based read (e.g. `SELECT... WHERE P`) and another transaction writes a data item matched by that predicate while the first transaction is still in flight."
  - Example: list of active employees and count of active employees



- Lost updates (write–write conflict)
  - A **lost update** occures when a transaction reads and modifies a values and other transaction writes a new value while the first transaction is still in progress.



47.2 Solution

- Simple solution:
  - **Serialize all transactions**
    - No two transaction can overlap ever.
    - Very slow, but works ...
      e.g. different transactions may use completely separate resources, thus now problem at all
- Advanced solution:
  - **Serializable transaction schedules**

47.3 Serializable Transaction Schedules

47.3.1 Schedule

- **Definition**: a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed.
  - Must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction
- Last statement of a transaction:

- `COMMIT` if sucessfull
- `ABORT` if failed

47.3.2 Examples

- Transaction 1: T_1 transfer of 5000.- from account 4711 to account 1174
- Transaction 2: T_2 transfer 10% of the balance from 4711 to 1174
- **Serial schedule**, T_2 after T_1

T_1	T_2
read(4711)	
4711 := 4711 - 5000	
write(4711)	
read(1174)	
1174 := 1174 + 5000	
write(1174)	
commit	
	read(4711)
	temp := 4711 * 0.1
	4711 := 4711 - temp
	write(4711)
	read(1174)
	1174 := 1174 + temp
	write(1174)
	commit

- **Non-serial schedule**, but equivalent

T_1	T_2
read(4711)	
4711 := 4711 - 5000	
write(4711)	
	read(4711)
	temp := 4711 * 0.1
	4711 := 4711 - temp
	write(4711)
read(1174)	
1174 := 1174 + 5000	
write(1174)	
commit	
	read(1174)
	1174 := 1174 + temp
	write(1174)
	commit

- In both schedules the sum of 4711 and 1174 is preserved.S
- The following schedule does not perserve the value of accounts 4711 and 1174

T_1	T_2
read(4711)	
4711 := 4711 - 5000	
	read(4711)
	temp := 4711 * 0.1
	4711 := 4711 - temp
	write(4711)
	read(1174)
write(4711)	
read(1174)	
1174 := 1174 + 5000	
write(1174)	
commit	
	1174 := 1174 + temp
	write(1174)
	commit

47.3.3 Serializability

- **Basic Assumption**: Each transaction preserves database consistency.
  - => serial execution of transactions preserves database consistency.
- **Definition**: A schedule is serializable if it is equivalent to a serial schedule.
- Different forms:
  1. Conflict serializability
  2. View serializability

47.3.4 Simplified View

- Only consider **read** and **write** instructions.
- All other computations are performed on data in local buffers.

47.3.5 Conflicting Instructions

- **Definition**: Instructions l_i and l_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

l_i	l_j	conflict
read(Q)	read(Q)	false
read(Q)	write(Q)	true
write(Q)	read(Q)	true
write(Q)	write(Q)	true

- A conflict between l_i and l_j forces a (logical) temporal order between them.
  - If there is no conflict between two consecutive operations l_i and l_j there order can be swaped without altering the result.

47.3.6 Conflict Serializability

- **Definition**: Two schedules S and S' are **conflict equivalent** if S can be transformed into S' by a series of swaps of non-conflicting instructions.
- **Definition**: A schedule is **conflict serializable** of it is **conflict equivalent** to a **serial** schedule.
- Example **conflict serializable** schedule:

T_1	T_2	T_1'	T_2'
read(A)		read(A)	
write(A)		write(A)	
	read(A)	read(B)	
	write(A)	write(B)	
read(B)			read(A)
write(B)			write(A)
	read(B)		read(B)
	write(B)		write(B)

- Example **not conflict serializable** schedule:

T_1	T_2
read(Q)	
	write(Q)
write(Q)	

47.3.7 View Serializability

- **Definition**: Two schedules S and S' with the same set of transactions are **view equivalent** if the following conditions are met for each data item Q.
  1. If T_i in S read the **initial** value of Q then, T_i in S' also must read the **initial** value of Q.
  2. If T_i in S executes **read(Q)** and that value was produced by T_j, then T_i in S' also must read the value of Q that was produced by the same **write(Q)** operation of T_j.
  3. The transaction that performs the final **write(Q)** in S must also perform the **write(Q)** in S'.
- Conditions 1 and 2: each transaction reads the same values in schedules S and S'
- Condition 3 + (1+2): schedules S and S' result in the same final system state
- **Definition**: A schedule S is view serializable if it is view equivalent to a serial schedule.
- Properties:
  - Every **conflict serializable** schedule is also **view serializable**.
  - Every **view serializable** schedule that is **not conflict serializable** has **blind writes**.
    - **Blind write**: a **write** operation without a **read** operation
- Example **view serializable schedule**:

T_1	T_2	T_3	T'_1	T'_2	T'_3
read(Q)			read(Q)		
	write(Q)		write(Q)		
write(Q)				write(Q)	
		write(Q)			write(Q)

- S' is the equivalent serial schedule
  - **read(Q)** reads the initial value of Q in both schedules
  - T_3 performs the final write in both schedules

47.3.8 Testing for Serializability

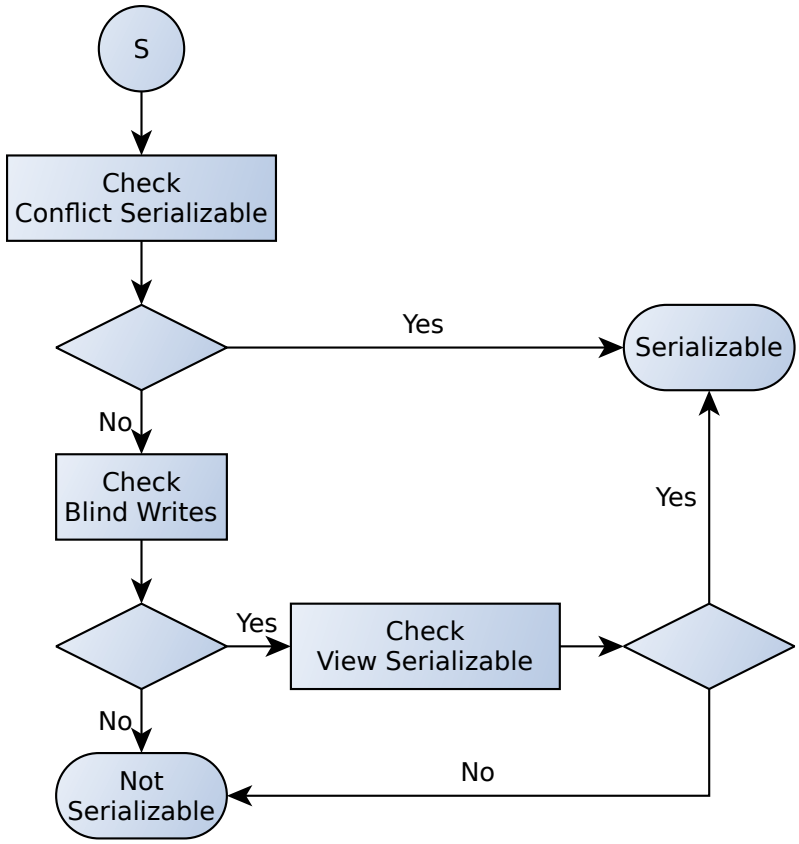
47.3.8.1 Testing for Conflict Serializability

- Create a **Precedence graph** (a directed graph)
  - **Vertices**: names of transactions
  - **Arcs**: for each conflicting pair of transactions (T_i, T_j) where T_i accessed the data item "earlier"
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
  - Takes $O(n + e)$ time where n denotes the number of vertices, e denotes the number of arcs

- Serial schedule can be obtained by **topological sorting** the graph

47.3.8.2 Testing for View Serializability

- Falls in the class of NP-complete problems
  - efficient algorithm is extremly unlikely
- Test for sufficient conditions



47.4 Recoverable Schedules

What happens when a transaction fails in a concurrent setting?

- **Definition**: A schedule is **recoverable** if and only if for each pair of transactions T_i and T_j the following condition holds.
  - If T_j reads a data item previously written by T_i, then the commit operation of T_i appears before the commit operation of T_j.
- Example **non-recoverable schedule**:

T_1	T_2
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- If T_1 aborts, T_2 would have read an inconsistent database state.

47.5 Cascading Rollbacks

A single transaction failure may lead to a series of transaction rollbacks.

- Example:

T_1	T_2	T_3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

- If T_1 aborts, also T_2 and T_3 must be undone.

47.5.1 Cascadeless Schedules

- **Definition**: A schedule is **cascadeless** if and only if for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable.

48 Acknowledgement

This document is based on

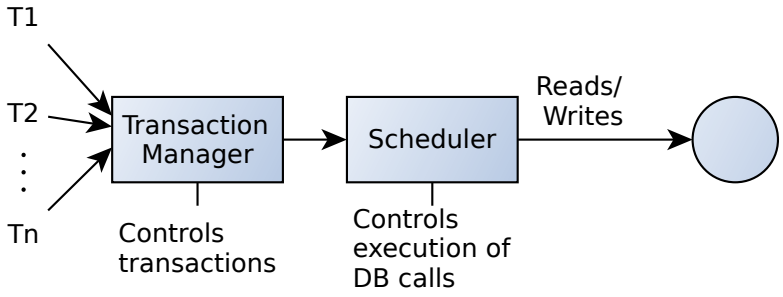
- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which "Database System Concepts" is the prescribed text. (see Copyright)

49 Motivation

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability after it has executed is a little too late!

49.1 Goal

Effective real-time scheduling of operations with guaranteed serializability of the resulting execution sequence.



49.2 Concurrency Control in DBMS

- **Definition**: Methods for scheduling the operations of database transactions in a way which guarantees serializability of all transactions.

49.3 Primary Concurrency Control Methods

- Lock-Based Protocols
- Optimistic Concurrency-Control (Validation-Based Protocols)
- Timestamp-Based Protocols
- Multiversion Schemas

49.4 Optimistic vs. pessimistic Concurrency Control

- **Locking** is **pessimistic**:
  - Assumption: during operation l_i of transaction T_i on a data item Q a (potentially) conflicting operation l_j of transaction T_j will access the same data item Q.
  - Has to be avoided by locking Q before accessing Q
- An **optimistic** strategy would be:
  - Perform all operations on a copy of the data. Check at the end (before commit) os there were any conflicts.
  - If no: commit, else abort the transaction (rollback, or resolve conflicts)

49.5 Lock-Based Protocols

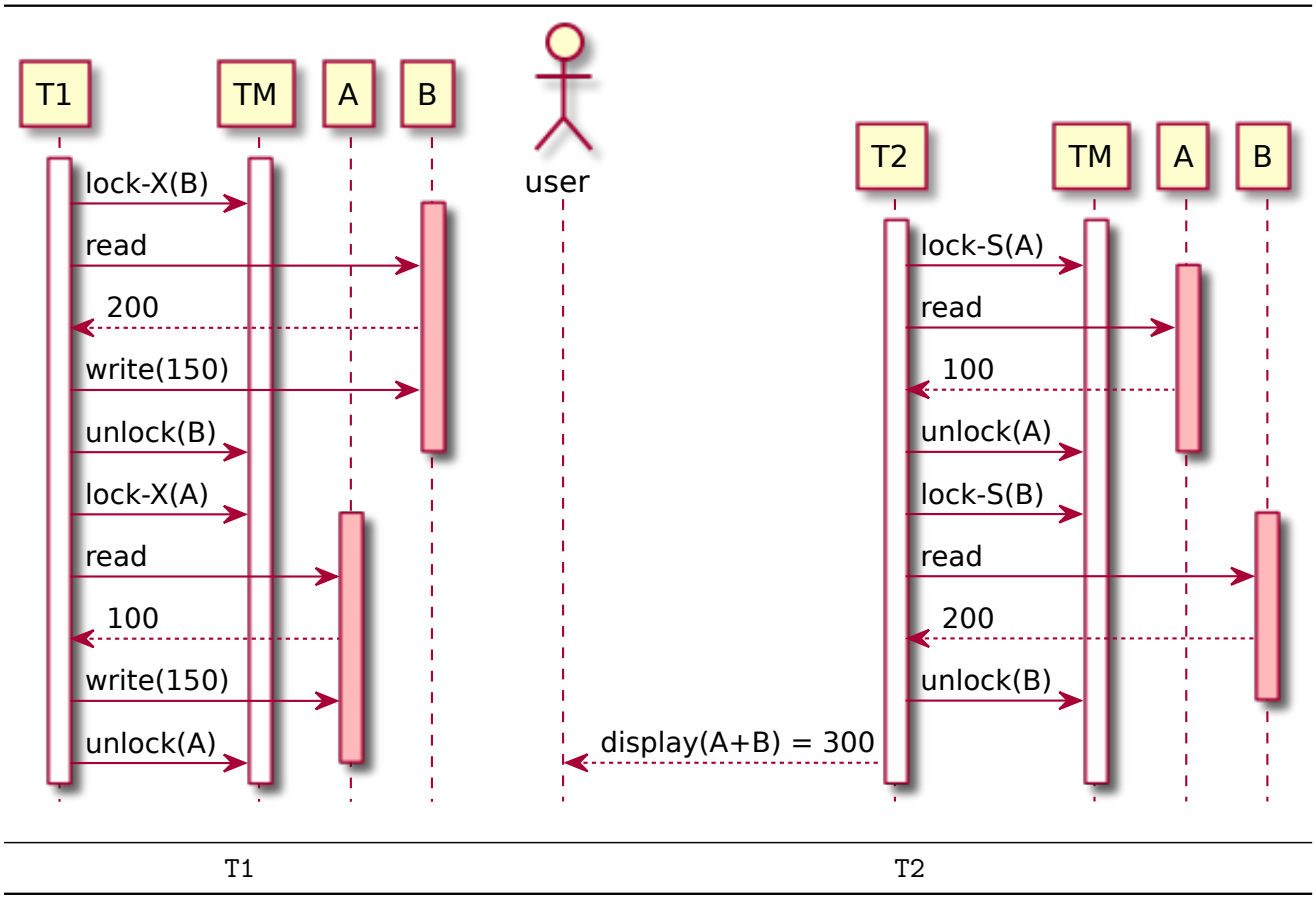
- **Definition**: A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.
- **Definition**: A **lock** is a mechanism to control concurrent access to a data item.
- Lock modes:
  - **exclusive**: Data item can be **read** and **written**.
  - **shared**: Data item can only be **read**.
- Transaction manager: controls the lock requests.
- Lock-compatability matrix:

	Shared	Exclusive
Shared	true	false
Exclusive	false	false

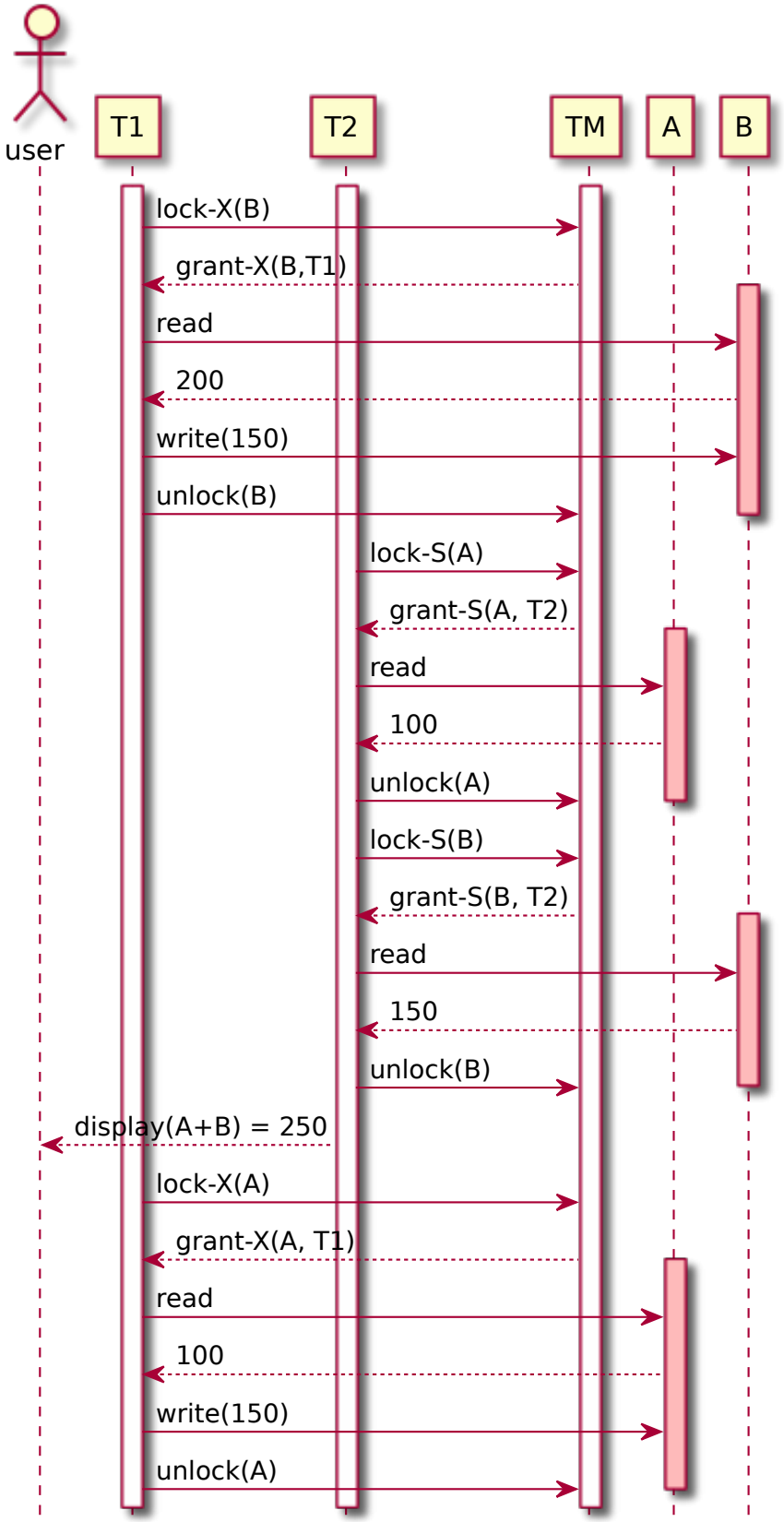
- Lock is granted of the requested lock is compatible already held locks.
- If a lock can not be granted the transaction has to wait until the incompatible locks habe been released.

49.5.1 Pitfalls of Lock-Based Protocols

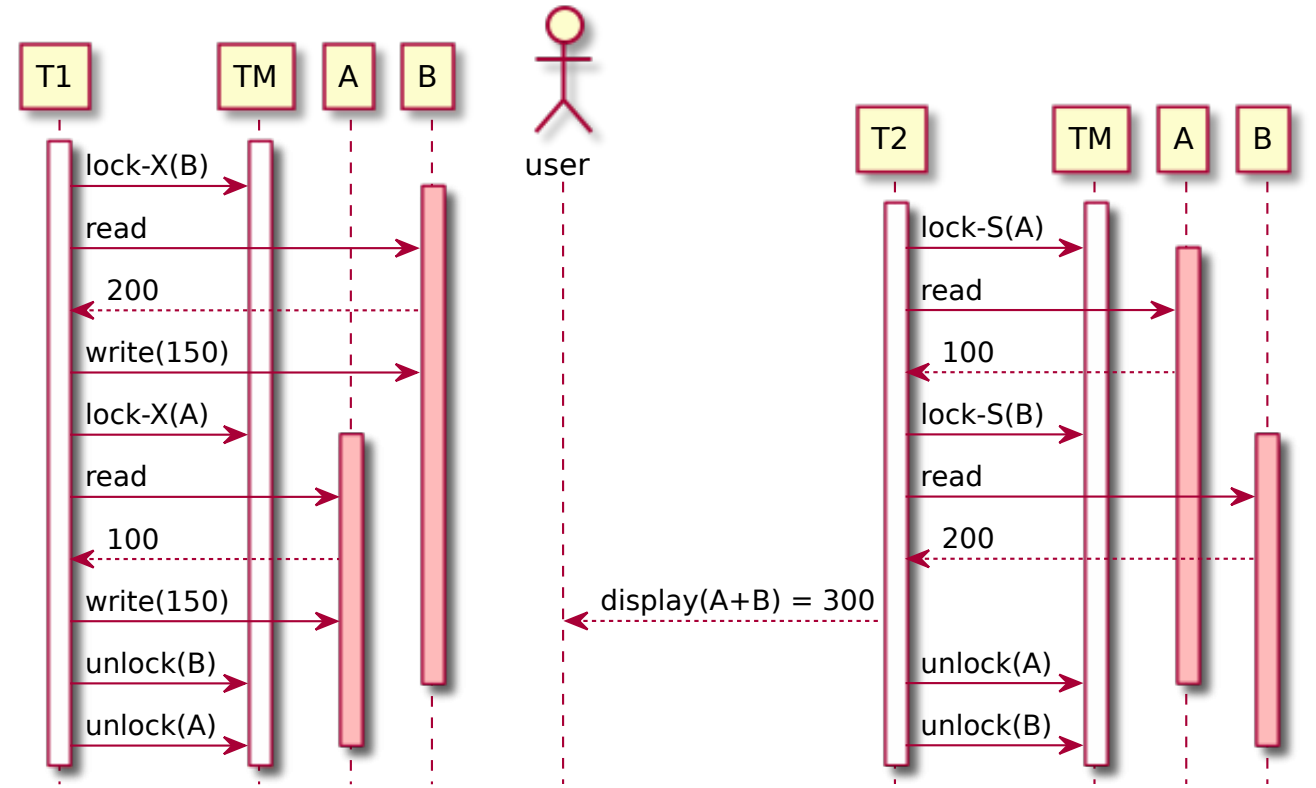
- Non-serializable schedules
  - Example:
    - `T1`: transfer 50 from account A to account B
    - `T2`: display the total amount in both accounts



- Non-serializable schedules

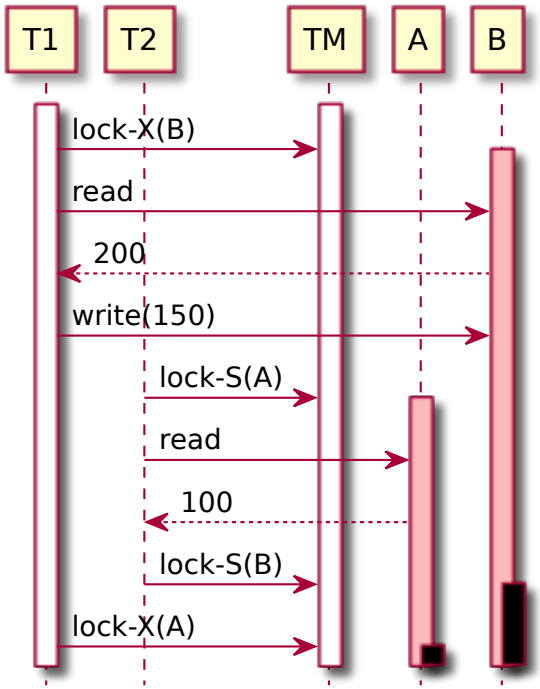


  - Possible result:
  - Problem: `T2` will show a wrong result (250)
  - Reason: `T1` unlocked data item B to early
- Deadlock
  - Delay the unlocking till the end of the transactions

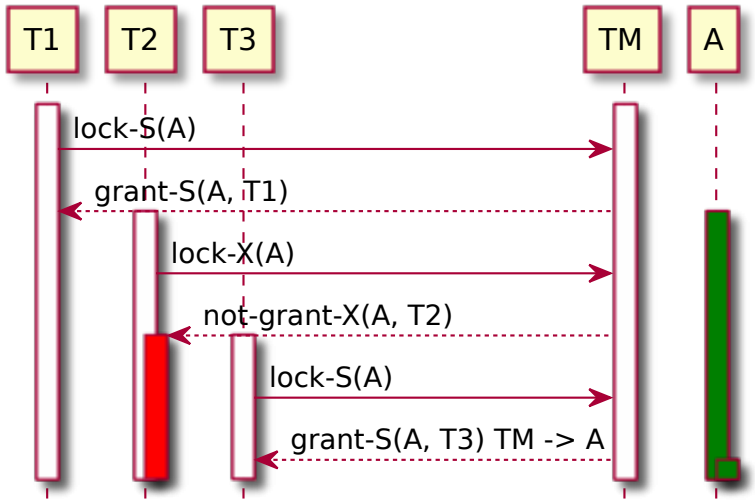


T3 T4

- Deadlock



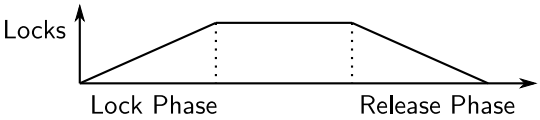
  - Possible result:
    - T4 will wait for T3 to release the lock on B
    - T3 will wait for T4 to release the lock on A
- Starvation



- Example
  - `T1` holds a shared-mode lock on A
  - `T2` requests an exclusive-mode lock on A, thus `T2` has to wait
  - `T3` requests a shared-mode lock on A, and gets it
  - `T2` has to wait
  - ...
- Solution:
  - The transaction manager grants the lock provided that:
    1. There is no other transaction holding a lock on data item Q in a mode that conflicts with mode M.
    2. There is no other transaction that is waiting for a lock on Q ant that made its lock request before the requesting transaction.

49.5.2 Two-Phase Locking Protocol

- To ensure serializablity, all transactions
  - shall not request locks they already own.
  - shall release all locks at commit time the latest.
  - shall enter a waiting queue, if a requested lock cannot be granted.
  - shall follow 2PL (2 Phase Lock Protocol).
    No lock shall be requested after the first lock has been released.

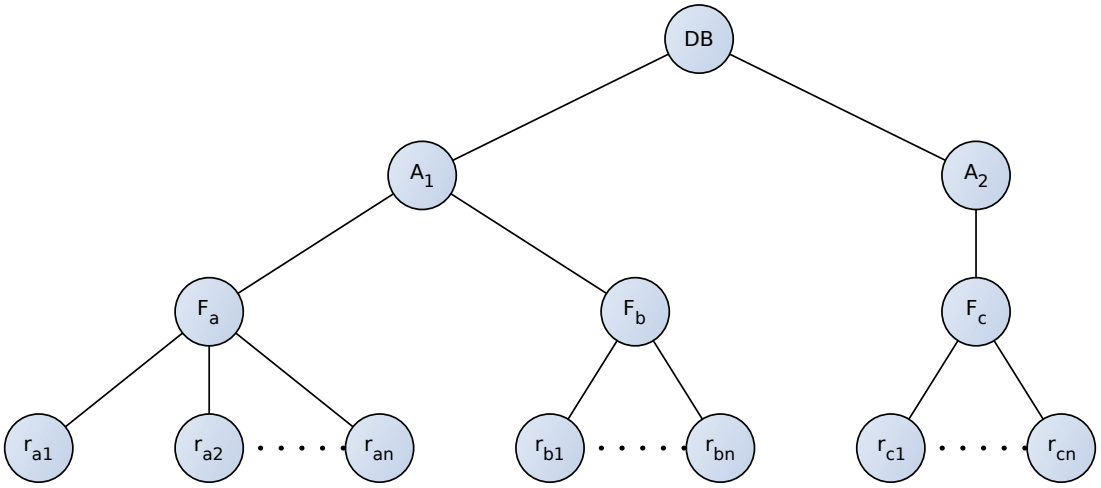


- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- Notes regarding 2PL+
  - 2PL+ guarantees serializability
  - 2PL+ does not prevent deadlocks
  - 2PL+ avoids cascading rollbacks during deadlock resolution
  - The SQL commit protocol is a special case of 2PL+
    Releasing all locks at commit time (so-called SQL commit protocol) avoids all cascading

rollbacks

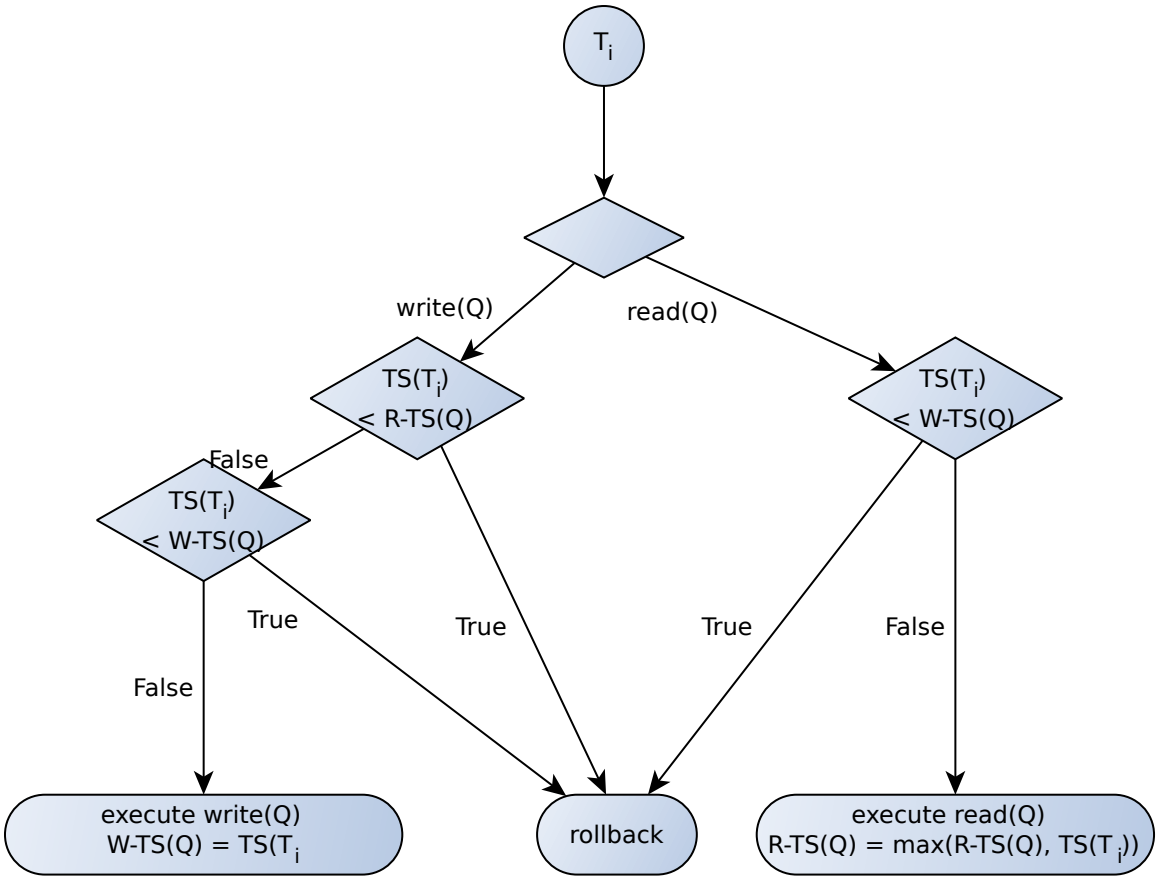
49.6 Multiple-Granularity Locking Protocol

- Motivation:
  - Degree of parallelism should be as high as possible, even when locking is used.
  - One single lock granularity (e.g. data items) insufficient, large overhead when many data items have to be locked.
  - E.g. A transaction T_i needs to access the entire database, and a locking protocol is used, then T_i must lock each item in the database.



49.7 Timestamp-Based Protocols

- Basic idea:
  - Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has timestamp $TS(T_i)$, a new transaction T_j is assigned timestamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
  - Timestamps determine the serializability order.
  - Ordering rule:
    If l_i of T_i and l_j of T_j are conflicting operations then l_i is executed before l_j if and only if $TS(T_i) < TS(T_j)$.
- Realization:
  - For each data item Q maintain two timestamp values:
    - *W-timestamp*(Q) is the largest timestamp of any transaction that executed write(Q) successfully.
    - *R-timestamp*(Q) is the largest timestamp of any transaction that executed read(Q) successfully.



1. T_i issues read(Q)
   1. If $TS(T_i) < W\text{-}TS(Q)$
      value of Q was already overwritten, rollback
   2. else
      execute read
2. T_i issues write(Q)
   1. If $TS(T_i) < R\text{-}TS(Q)$
      value produced by T_i was already needed, rollback
   2. If $TS(T_i) < W\text{-}TS(Q)$
      T_i tries to write an obsolete value of Q, rollback
   3. else
      execute write

49.8 Validation-Based Protocols (Optimistic Concurrency-Control)

- Motivation:
  - In the case where the majority of transaction are **read-only** transactions
    - rate of conflicts may be low
    - executed without concurrency-control would leave the system state consistent
    - reduction of overhead possible
- Basic idea:
  - all transactions work on copies, check for conflicts before write into DB
  - if conflict: abort else commit

- Realization:
  - Execution of T_i done in three phases:
    1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
    2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
    3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- Each transaction T_i has 3 timestamps
  - *Start*(T_i): the time when T_i started its execution
  - *Validation*(T_i): the time when T_i entered its validation phase
  - *Finish*(T_i): the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus $TS(T_i)$ is given the value of *Validation*(T_i).
- Validation Test for T_j
  - If for all T_i with $TS(T_i) < TS(T_j)$ either one of the following condition holds:
    - *Finish*(T_i) < *Start*(T_j)
    - *Start*(T_j) < *Finish*(T_i) < *Validation*(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j.

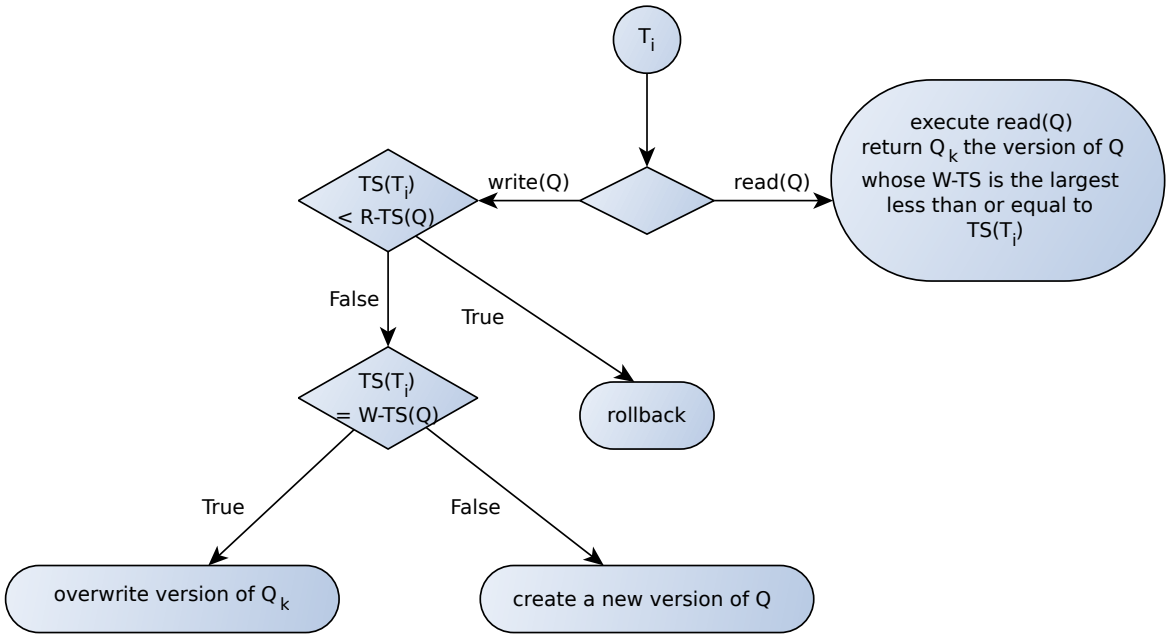
  then validation succeeds and T_j can be **committed**. Otherwise, validation fails and T_j is aborted.

49.9 Multiversion Schemas

- Main idea: **Reads** should see a consistent (and committed) state, which might be older than the current object state.
- Multiversion schemas keep old versions of data items to increase concurrency.
- Each successful **write** results in the creation of a **new version** of the data item written.
- Timestamps are used to lable versions
- Realizations:
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking

49.9.1 Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $< Q_1, Q_2, \ldots, Q_m >$. Each version Q_k contains three data fields:
  - Content: the value of version Q_k.
  - *W-timestamp*(Q_k): timestamp of the transaction that created (wrote) version Q_k
  - *R-timestamp*(Q_k): largest timestamp of a transaction that successfully read version Q_k
- when a transaction T_i creates a new version Q_k of Q, Q_k's *W-timestamp* and *R-timestamp* are initialized to $TS(T_i)$.
- *R-timestamp* of Q_k is updated whenever a transaction T_j reads Q_k , and $TS(T_j) >$ *R-timestamp*(Q_k).



49.9.2 Multiversion Two-Phase Locking

- **Read-only transactions** are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow **rigorous two-phase locking** (all locks are held to the end of the transaction).
  - Each successful write results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter ts-counter that is incremented during commit processing.
- Update transaction process on data item Q:
  - Read data item:
    - obtain a shared lock on Q
    - read Q_k
  - Write data item:
    - obtain an exclusive lock on Q
    - create a new version of Q, set timestamp to ∞
  - Commit processing:
    - Set timestamp counter on the created versions to ts-counter + 1
    - increment ts-counter by 1

50 Acknowledgement

This document is based on

- The slides and figures of Silberschatz, Korth, Sudarshan
  - Original material
  - which are are authorized for personal use, and for use in conjunction with a course for which “Database System Concepts” is the prescribed text. (see Copyright)

51 Failure Classification

- Transaction failure
  - Logical errors: transaction cannot complete due to some internal error condition
  - System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash
  - e.g. a power failure or other hardware or software failure causes the system to crash.
  - **Assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
- Disk failure
  - e.g. a head crash or similar disk failure destroys all or part of disk storage

52 Storage Structure

- Volatil storage:
  - does not survive system crashes
  - examples: main memory, cache memory
- Nonvolatil storage:
  - survives system crashes
  - e.g. disk, tape, flash memory, non-volatile (battery backed up) RAM
  - may still fail
- Stable storage:
  - information is **never** lost
  - approximated by maintaining multiple copies on distinct nonvolatile media (**backups**)

52.1 Backup Types

52.1.1 Physical Backup

- are copies of physical database files
- e.g. `DISKCOPY`

Can be hot or cold: * Hot backup (Hot mirror): Users can modify the database * Changes are logged and then applied to synchronize the database and backup copy. * No system downtime * Cold backup: Users can not modify the database * System is “down”

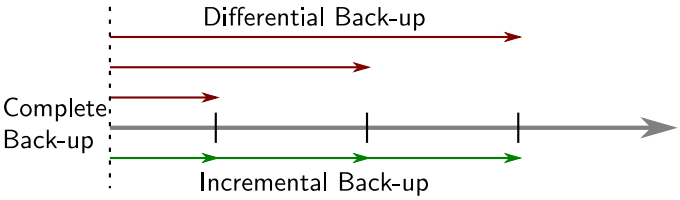
52.1.2 Logical Backup

- Copies data, but not physical files
  - Generates necessary Structured Query Language (SQL) statements to obtain all table data written to a binary file.

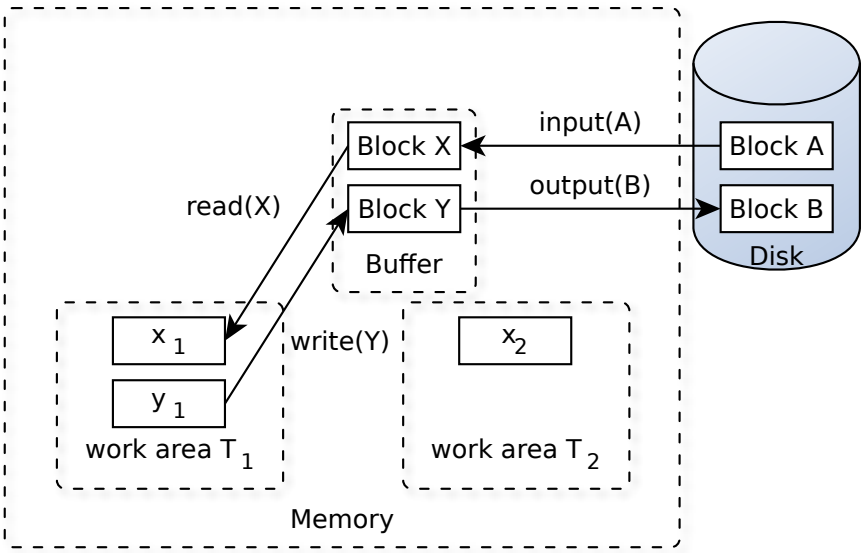
52.2 Backup Quantity

- Complete (Full)
  - Create a **complete** copy of the whole database (data files, transaction logs,...)
- Differential

  - Create a copy of those files that have been changed since the last **full** backup took place.
- Incremental
  - Copy all of the files that have changed since the last backup (either a full or incremental backup) was made.
  - Can be done **hot**.



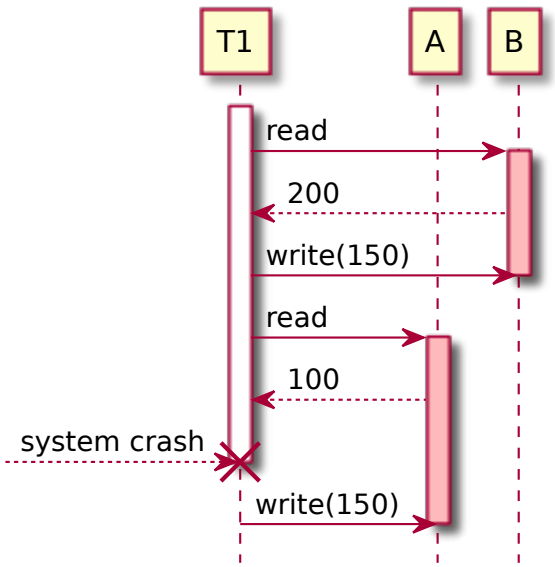
53 Data Access



- **Block**: storage units of data
  - units of data transfered to and from disks
  - **Physical blocks** are those blocks residing on the disk.
  - **Buffer blocks** are the blocks residing temporarily in main memory.
- **Block movements**:
  - `input(B)` transfers the physical block `B` to main memory.
  - `output(B)` transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- **Work-area**:
  - Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
- **Transfer within Memory**:
  - `read(X)` assigns the value of data item `X` to the local variable x_i.
  - `write(X)` assigns the value of local variable x_i to data item `{X}` in the buffer block.

54 Recovery and Atomicity

54.1 Example



- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- **Log-based recovery mechanisms**

54.2 Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a $< T_i$ start $>$ log record
- Before T_i executes `write(X)`, a log record $< T_i, X, V_1, V_2 >$ is written, where V_1 is the value of X before the write (the old value), and V_2 is the value to be written to X (the new value).
- When T_i finishes it last statement, the log record $< T_i$ commit $>$ is written.
- Two approaches using logs
  - **Deferred** database modification
  - **Immediate** database modification

54.2.1 Immediate Database Modification

- Update of an **uncommitted** transaction to be made to the buffer, or the disk itself, before the transaction commits
- Log must be written **before** DB item is written
- Output to stable storage
  - at any time, and order

54.2.2 Deferred Database Modification

- Updates to buffer/disk only at the time of transaction commit
  - Simplifies recovery
  - Overhead of storing local copy

54.2.3 Undo and Redo Operations

- **Undo** of a log record $< T_i, X, V_1, V_2 >$ writes the old value V_1 to X
- **Redo** of a log record $< T_i, X, V_1, V_2 >$ writes the new value V_2 to X
- Undo and Redo of Transactions
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - each time a data item X is restored to its old value V a special log record $< T_i, X, V >$ is written out
    - when undo of a transaction is complete, a log record $< T_i$ abort $>$ is written out.
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

54.2.4 Actions in Case of Failure

- Transaction T_i needs to be **undone** if the log
  - contains the record $< T_i$ start $>$,
  - but does not contain either the record $< T_i$ commit $>$ or $< T_i$ abort $>$.
- Transaction T_i needs to be **redone** if the log
  - contains the records $< T_i$ start $>$
  - and contains the record $< T_i$ commit $>$ or $< T_i$ abort $>$

54.2.4.1 Examples

1. $< T_0$ start $>$; $< T_0, A, 1000, 950 >$; $< T_0, B, 2000, 2050 >$
2. $< T_0$ start $>$; $< T_0, A, 1000, 950 >$; $< T_0, B, 2000, 2050 >$; $< T_0$ commit $>$; $< T_1$ start $>$; $< T_1, C, 700, 600 >$
3. $< T_0$ start $>$; $< T_0, A, 1000, 950 >$; $< T_0, B, 2000, 2050 >$; $< T_0$ commit $>$; $< T_1$ start $>$; $< T_1, C, 700, 600 >$; $< T_1$ commit $>$

- Recovery actions:
  1. **undo**(T_0)
     1. set B to 2000, A to 1000
     2. log records $< T_0, B, 2000 >$; $< T_0, A, 1000 >$; $< T_0$ abort $>$
  2. **redo**(T_0), **undo**(T_1)
     1. set A to 950, B to 2050
     2. set C to 700
     3. log records $< T_1, C, 700 >$; $< T_1$ abort $>$
  3. **redo**(T_0), **redo**(T_1)
     1. set A to 950, B to 2050 and C to 600

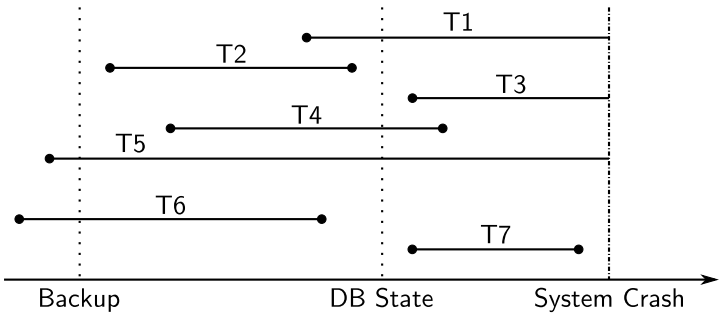
54.2.5 Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record $<$ checkpoint $L>$ onto stable storage where L is a list of all transactions active at the time of checkpoint.

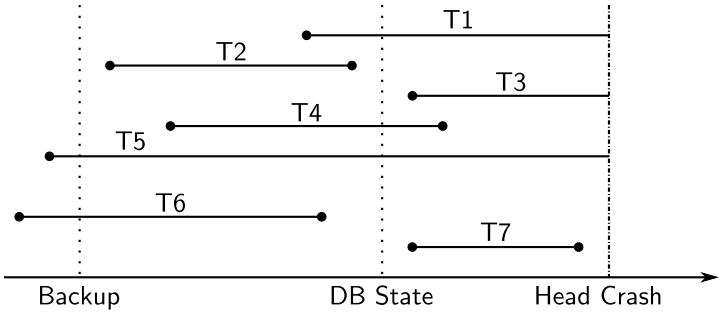
- All updates are stopped while doing checkpointing
- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent $<$ checkpoint $L>$ record
     - Only transactions that are in L or started after the checkpoint need to be redone or undone
     - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  1. Continue scanning backwards till a record $<T_i$ start $>$ is found for every transaction T_i in L.
     - Parts of log prior to earliest $<T_i$ start $>$ record above are not needed for recovery, and can be erased whenever desired.

54.2.5.1 Examples

- Recovery from system crashes
  - Situation: Buffers lost, media intact
  - Action: undo all non-commited transactions, redo all transactions after last backup



- Actions:
  - Undo: T1, T3, T5
  - Redo: T2, T4, T6, T7
- Recovery from media failures
  - Situation: Buffers lost, some media lost
  - Action: Re-build from last backup, undo all transactions not commited before last backup, redo all transactions commited after last backup



- Actions:
  - Undo: T1, T2, . . . , T7
  - Redo: T2, T4, T6, T7