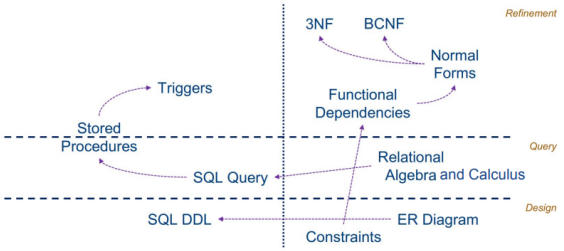
CS2102 Database Sys Summary

AY23/24 Sem 1, github.com/gerteck

Topics & Objectives

- **Design**: Entity-Relationship (ER) Model, Functional Dependencies, Normal Forms
- **Implementation**: SQL (Data definition language, Queries, Stored procedures, Triggers)
- **Theory**: Relational Calculus and algebra
- Module covers fundamental concpets and techniques for:
  - Understanding and practice of design & implementation of database applications and management of data with relational db management systems.
  - Design of ER data models to capture data requirements, translate to relational database schema, refine using schema decompositions to avoid anomalies.
  - Use SQL to define relational schemas, write queries.
  - Reason about correctness using concepts of formal query lang (relational calculus & algebra) and apply knowledge to develop database applications.



1. Database Management Sys DBMS

Challenges for Data-Intensive applications

- **Efficiency**: Fast access to information in volumes of data
- **Transactions**: "All or nothing" changes to data
- **Data Integrity**: Parallel access and changes to data
- **Recovery**: Fast and reliable handling of failures (e.g. HDD/Sys crash, power outage, network disruption)
- **Security**: Fine-grained data access rights

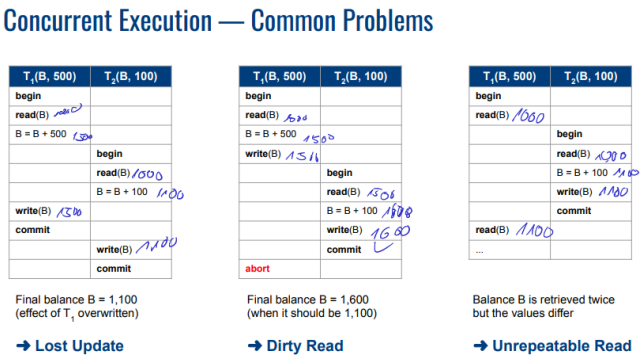
File-based data management to DBMS

- Complex, low level code, Often similar requirements across different programs
- **Problems**: High development effort, Long development times, Higher risk of (critical) errors
- **DBMS**: Set off universal and powerful functionalities for data management, with faster application development, higher stability, less errors.

Core concepts of DBMS

- **ACID Transaction**: Finite sequence of database operations (reads and/or writes), smallest logical unit of work
- **Atomicity**: either all effects of T are reflected in the database or none ("all or nothing")
- **Consistency**: the execution of T guarantees to yield a correct state of the database
- **Isolation**: execution of T is isolated from the effects of concurrent transactions
- **Durability**: after commit of T, its effects are permanent even in case of failures

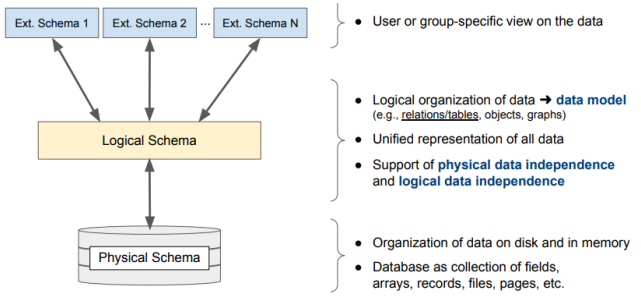
Concurrent Execution



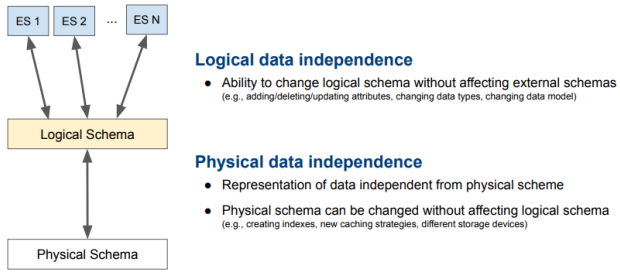
Require Serializable transaction execution:

- A concurrent execution of a set of transactions is serializable if this execution is equivalent to some serial execution of the same set of transactions
- Two executions are equivalent if they have the same effect on the data
- **DBMS**: Support concurrent executions of transactions to optimize performance, Enforce serializability of concurrent executions to ensure integrity of data

Data Abstraction



Data Independence



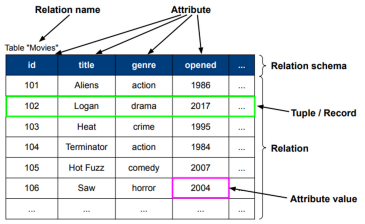
Terminology / Definitions

- **Data Model**: Collection of concepts for describing data
- **Schema**: Description of structure of DB using data model
- **Schema Instance**: Content of a DB at a particular time

Relational Data Model

Data is modelled by relations, and each relation has a definition called a relation schema. This schema specifies attributes (columns) and data constrains (e.g. domain constraints)

- **Relation**: Can be seen as Tables with rows and columns:
  - No. of cols = Degree/Arity, No. of rows = Cardinality
  - Each row is called a tuple/record. It has a component for each attribute of the relation.
  - A relation is thus a set of tuples and an instance of the relation schema, i.e. of a single table.
- **Domain**: Set of atomic values, e.g. integers. All values for an attribute is either in this domain or null.
- **Relational database schema**: Set of relation schemas and their data constraints, i.e. of multiple tables
- **Relational database**: Instance of the schema and is a collection of tables.



Term	Description (informal)
attribute	Column of a table
domain	Set of possible values for an attribute
attribute value	Element of a domain
relation schema	Set of attributes (with their data types + relation name)
relation	Set of tuples
tuple	Row of a table
database schema	Set of relation schemas
database	Set of relations / tables

Integrity Constraints

Condition that restricts the data that can be stored in a database instance. A legal relation instance is a relation that satisfies all specified ICs.

- **Domain Constraints**: Restrict the attribute values of relations, e.g. only integers allowed
- **Key Constraints**:
  - **Superkey**: A superkey is a subset of attributes in a relation that unique identifies its tuples.
  - **Key**: A key is a superkey which is minimal, i.e. no proper subset of itself is a superkey.
  - **Candidate keys**: Set of all possible keys for a relation. One of these keys is selected as the primary key.
  - **Primary key**: Chosen candidate key for a relation, Cannot be null (entity integrity constraint), Underlined in relation schema. Prime attribute: Attribute of a primary key (cannot be null)
- **Foreign Key Constraints**:
  - **Foreign key**: A foreign key refers to the primary key of a second relation (which can be itself)
  - Each foreign key value must be the primary key value in the referenced relation or be null (foreign key constraint)
  - Also known as referential integrity constraints.

Term	Description (informal)
(candidate) key	Minimal set of attributes that uniquely identify a tuple in a relation
primary key	Selected key (in case of multiple candidate keys)
foreign key	Set of attributes that is a key in referenced relation
prime attribute	Attribute of a (candidate) key

- Terminology: DB. vs DBS vs. DBMS

DBS = DBMS + n*DB (n>0)

2. SQL: Structured Query Language

- Declarative language: focus on what to compute, not on how to compute
- Contains two parts: Data Definition Language and Data Manipulation Language

Datatypes

type	description
boolean	logical Boolean (true/false)
integer	signed 4-byte integer
float8	double precision floating-point number (8 bytes)
numeric(p, s)	number with p significant digits and s decimal places
char(n)	fixed-length character string
varchar(n)	variable-length character string
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

Integrity Constraints 2

- A **consistent state** of the database is a state which complies with the with the business rules as defined by the structural constraints and the integrity constraints in the schema.
- If an integrity constraint is violated by an operation or a transaction, the operation or the transaction is aborted and rolled back and its changes are undone, otherwise, it is committed and its changes are effective for all users.
- Five main kinds of integrity constraints in SQL: **NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK.**

Primary Key

A primary key is a set of columns that uniquely identifies a record in the table. Each table has at most one primary key. The primary key can be one column or a combination of columns.

```
-- Declare primary key as column constraint
CREATE TABLE customers (
 firstname VARCHAR(64 ,
 lastname VARCHAR(64) ,
 email VARCHAR(64) ,
id VARCHAR(16) PRIMARY KEY);
```

Composite Primary Key & NOT NULL

A not null constraint guarantees that no value of the column can be set to null. A not null constraint is always declared as a row constraint. When it is explicit, it is declared with the keyword NOT NULL.

```
CREATE TABLE games (
name VARCHAR(32) ,
version CHAR(3) ,
price NUMERIC NOT NULL,
PRIMARY KEY (name, version) );
```

Data Insertion Populating Tables

```
INSERT INTO customers VALUES(
'Carole', 'Yoga', 'cyoga@email.org',
'Carole89');
```

Deleting Tables

```
DELETE FROM customers
-- DROP deletes content \& table definition
DROP TABLE customers
DROP TABLE IF EXISTS downloads
```

Unique

A unique constraint on a column or a combination of columns guarantees the table cannot contain two records with the same value in the corresponding column or combination of columns.

```
CREATE TABLE customers (
firstname VARCHAR(64) NOT NULL ,
lastname VARCHAR(64) NOT NULL,
email VARCHAR(64) UNIQUE NOT NULL,
id VARCHAR(16) PRIMARY KEY,
UNIQUE (firstname, lastname));
```

Foreign Key

- A foreign key constraint enforces **referential integrity**. The values in the columns for which the constraint is declared must exists in the corresponding columns of the referenced table.
- Referenced columns are usually required to be the primary key of the referenced table. Some systems relax this.

- A foreign key is declared using the keyword REFERENCES as a row constraint and the keywords FOREIGN KEY and REFERENCES as a table constraint.

```
CREATE TABLE downloads (
customerid VARCHAR (16) REFERENCES customers
    (id),
name VARCHAR (32)
version CHAR (3),
FOREIGN KEY ( name, version) REFERENCES games
    (name, version) );
```

Check

- Check constraint enforces any other condition that can be expressed in SQL. Declared as row or table constraint.

```
CREATE TABLE games (
name VARCHAR(32),
version CHAR(3),
PRIMARY KEY (name, version)
price NUMERIC NOT NULL CHECK (price > 0)
-- or as table constraint:
CHECK (price > 0) );
```

Update and Delete Propagation

- The annotations ON UPDATE/DELETE with the option CASCADE propagate the update or deletion, when there are chains of foreign key dependencies.

```
CREATE TABLE downloads(
id VARCHAR (16) REFERENCES customers (id)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
name VARCHAR(32),
version CHAR(3),
PRIMARY KEY (id, name, version),
FOREIGN KEY (name, version) REFERENCES
    games(name, version)
  ON UPDATE CASCADE
  ON DELETE CASCADE);
```

- Generally a good idea to constraint all columns not to be null unless there is a good design or tuning reason for not doing so.
- Think carefully about which foreign keys should be subject to cascade.
- Good idea to defer all the constraints that can be deferred. These are checked at the end of a transaction and not immediately after each. operation.

Querying Tables

Print Table

- Wildcard '*' to include all attributes

```
SELECT *
FROM customers;
```

View

- We can give a name to a query, called a view. Once created, a view can be queried like a table.
- Creating a view is generally a better option than creating and populating a table, temporary or not.

```
CREATE VIEW sg\_customers AS
SELECT c.firstname, c.lastname, c.email, c.id
FROM customers c
WHERE country = 'Singapore';
```

```
SELECT * FROM sg\_customers;
```

3. SQL Queries

Printing one Table

```
SELECT firstname, lastname
FROM customers
WHERE country = 'Singapore';
```

DISTINCT and ORDER BY

- Selecting a subset of columns may result in duplicate row even if original table has a primary key.
- DISTINCT keyword eliminates eventual duplicates, requests results contain distinct rows.
- Both DISTINCT and ORDER BY involve sorting and conceptually ORDER BY is applied before SELECT DISTINCT.

```
SELECT DISTINCT name, version
FROM downloads
ORDER BY name ASC, version DESC;
```

WHERE

- Returns rows that evaluate to true, filter rows on a Boolean condition
- Uses Boolean operators such as AND, OR and NOT, and various comparison operators such as `>, <, >=, <=, <>, IN, LIKE` and `BETWEEN AND`
- Does not return rows that evaluate to unknown/null!
- '-' matches single char
- '%' matches any sequence of zero or more chars

```
SELECT firstname, lastname
FROM customers
WHERE country IN ('Singapore', 'Indonesia')
AND (dob BETWEEN '2001-01-01' AND
    '2000-12-01' OR since >= '2016-12-01 )
AND lastname LIKE 'B%'
```

- PostgreSQL use "——" for concatenation

```
-- Not to collect GST below 30 cents:
SELECT name || ' ' || version AS game, price
    * 1.07
FROM games
WHERE price * 0.07 < 0.3
```

De Morgan's Laws

```
SELECT name
FROM games
-- all 3 are the same:
WHERE (version = '1.0' or version = '1.1')
WHERE version IN ('1.0', '1.1')
WHERE NOT (version <> '1.0' AND version <>
    '1.1');
```

NULL value

- Every domain has additional value, null. Ambiguous, could be "unknown", "does not exists", or both. In SQL it is generally (but not always) "unknown".
- With null values, the logic of SQL is a three valued logic with unknown.
- Use `IS (NOT)NULL` for comparison with null
- `COALESCE()` returns the first non-null of its argument.
- `COUNT(*)` counts NULL values.
- `COUNT(att)AVG(att)MAX(att)MIN(att)` eliminate null values

Cross Join

• Cross join & Cartesian Product & cross product, represented by comma `CROSS JOIN ,`

• Cross join with `WHERE` clause: add condition that FK columns equal to the corresponding PK columns.

• Systematically define table variables (e.g. `games AS g`)

```
SELECT *
FROM customers c, downloads d, games g
WHERE d.id = c.id
AND d.name = g.name
AND d.version = g.version
```

4. Algebraic SQL Queries

Inner Join

• `JOIN` interpreted as `INNER JOIN` • Inner joins combine records from two tables whenever there are matching values in a field common to both tables.

```
SELECT *
FROM customers c JOIN downloads d ON d.id =
    c.id
JOIN games g ON d.name = g.name AND d.version
    = g.version;
```

Natural Join

• If we give the same name to columns that are the same, can use natural join. Joins the rows that have the same values for their columns that have the same names. It also prints one of the two equated columns

```
SELECT *
FROM customers c NATURAL JOIN downloads d
    NATURAL JOIN games g;
```

Outer Join

• outer join keeps the columns of the rows in the left (left outer join), right (right outer join) or in both (full outer join) tables that do not match anything in the other table according to the join condition and pad the remaining columns with null values.

• Better to avoid outer joins whenever possible as they introduce null values.

• `RIGHT (OUTER)JOIN`, `LEFT (OUTER)JOIN`

• `FULL (OUTER)JOIN`

```
-- finds customers, never downloaded a game
SELECT c.id FROM customers c
LEFT JOIN downloads d ON c.id = d.id
WHERE d.id IS NULL;
```

Set Operations

• Union, intersect and non-symmetric difference.

• Eliminate duplicates: `UNION`, `INTERSECT`, `EXCEPT`

• Keep duplicates: `UNION ALL`, `INTERSECT ALL` `EXCEPT ALL`

4. Aggregate SQL Queries

Aggregate Functions

• The values of a column can be aggregated aggregation functions such as `COUNT()`, `SUM()`, `MAX()`, `MIN()` `AVG()`, `STDDEV()` etc.

```
SELECT COUNT(*)
FROM customers c;
```

```
-- ALL is default and omitted
-- DISTINCT needed
SELECT COUNT(ALL DISTINCT c.country)
FROM customers c;
```

```
-- Finds min, max, avg and stddev, TRUNC()
    displays 2 d p
SELECT MAX(g.price),
MIN(g.price),
TRUNC(AVG(g.price), 2) AS ave,
TRUNC(STDDEV(g.price), 2) AS std
FROM games g;
```

Group By

• The GROUP BY clause creates groups of records that have the same values for the specified fields before computing the aggregate functions.

• Groups are formed after the rows have been filtered by the WHERE clause

• Recommended (and required by SQL standard) to include attributes projected in the SELECT clause in the GROUP BY clause.

• The order of columns in the GROUP BY clause does not change the meaning of the query.

```
SELECT c.country, COUNT(*)
FROM customers c
WHERE c.dob >= '2000-01-01'
GROUP BY c.country
```

```
SELECT c.country, EXTRACT( YEAR FROM c.since)
    AS regyear, COUNT(*) AS total
FROM customers c, downloads d
WHERE c.id = d.id
GROUP BY c.country, regyear,
ORDERBY regyear, c.country;
```

Having

• Aggregate functions can be used in conditions. However, agg. functions not allowed in WHERE.

• `HAVING` clause to add conditions to be checked after the evaluation of the GROUP BY.

• `HAVING` can only involve aggregate functions, columns listed in the GROUP BY clause and subqueries.

```
SELECT c.country,
FROM customers c
GROUP BY c.country
HAVING COUNT(*) >= 100;
```

5. Nested SQL Queries

Subqueries

• In FROM clause: Must be enclosed in parenthesis, Table alias mandatory, Column aliases optional
• Not recommended, can be written as simple query

```
SELECT cs.lastname, d.name
FROM (SELECT *
   FROM customers c
  WHERE c.country = 'Singapore') AS cs,
      downloads d
WHERE cs.id = d.id;
```

• In WHERE clause, also can be written as simple query.
• Never use a comparison to a subquery without specifiying the quantifier ALL or ANY

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price >= ALL (
  SELECT g2.price
  FROM games g2);
-- or do:
WHERE g1.price = ALL (
  SELECT MAX(g2.price)
  FROM games g2);
-- Note HAVING g.price=MAX(g.price) will not
    work
```

Exists

• `EXISTS` evaluates to true if the subquery has some results.
• Generally correlated. If uncorrelated, likely unnecessary.

```
SELECT c.id FROM customers c
WHERE NOT EXISTS (
  SELECT d.id
  FROM downloads d
  WHERE c.id = d.id);
-- same as
WHERE c.id NOT IN (
  SELECT d.id
  FROM downloads d);
-- same as
WHERE c.id <> ALL (
  SELECT d.id
  FROM downloads d);
```

```
-- Find countries with most customers
SELECT c1.country
FROM customers c1
GROUP BY c1.country
HAVING COUNT(*) >= ALL (
  SELECT COUNT(*)
  FROM customers c2
  GROUP BY c2.country);
```

SQL Conditionals

CASE expression • Generic conditional, like if/else statement, Used in SELECT, ORDER BY, etc

```
-- Regular if else
CASE
  WHEN cond1 then res1
  WHEN cond2 then res2
  ...
  WHEN condN then resN
  ELSE res0
END
```

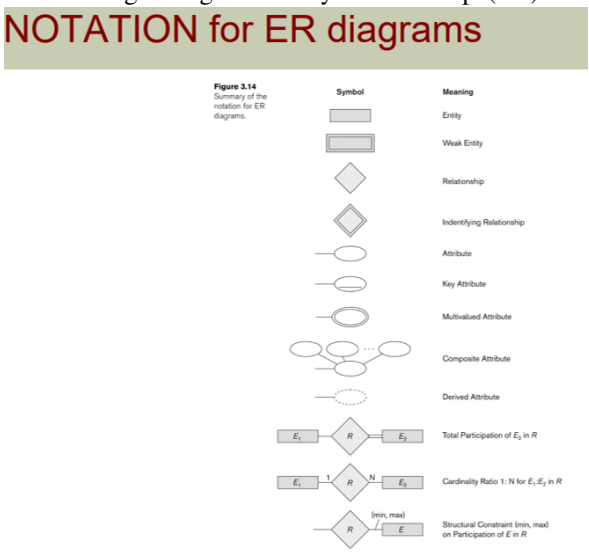
```
-- Switch like statement
CASE expression
  WHEN val1 then res1
  WHEN val2 then res2
  ...
  WHEN valN then resN
  ELSE res0
END
```

Conceptual evaluation of queries

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT/OFFSET

6. ER Model

Data Modeling Using the Entity-Relationship (ER) Model

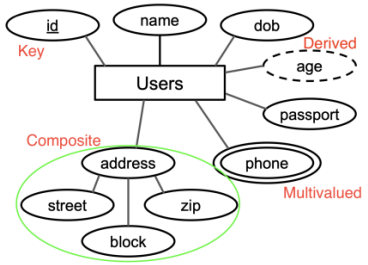


Entity

• Objects that are distinguishable from other objects
• **Entity set:** Collection of entities of the same type
• In ER diagrams, an entity type is displayed in a rectangular box

Attributes

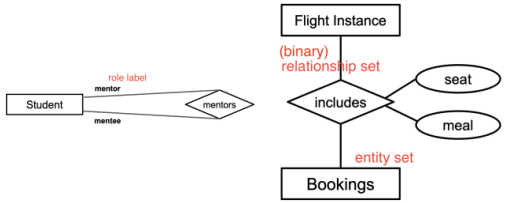
• **Attributes** are properties used to describe an entity
• Each attribute has a value set (or data type) associated with it – e.g. integer
• **Key attribute(s)** uniquely identifies each entity
• **Composite attribute** composed of multiple other attributes
• **Multivalued attribute** may consist of more than one value for a given entity
• **Derived attribute** derived from other attributes



• Attributes are displayed in ovals, multivalued attributes double ovals

Relationship

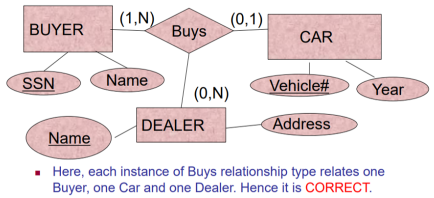
• **A relationship** relates two or more distinct entities with a specific meaning.
• **Degree** of a relationship type is the number of participating entity types. A n-ary relationship set involves n entity roles. Typically binary or ternary.



• **Relationship Set**: Collection of relationships of the same type, can have their own attributes that further decribe the relationship
• Most **relationship attributes** are used with M:N relationships: (In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship)
• We represent the relationship type as **Diamond-shaped box**, connected to the participating entity types.
• Relationship type typically readable from left to right and top to bottom.

N-ary relationships

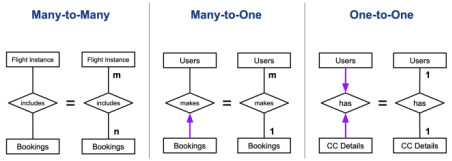
• **Implicit Constraint** In ternary relationship, every instance of relationship must have one instance of each entity.
• Rule extends to n-ary relationships



• **Avoid Ternaries for Easy Modeling**: e.g. "objectifying" the relationship type "Interview" into an entity type 'Interview".

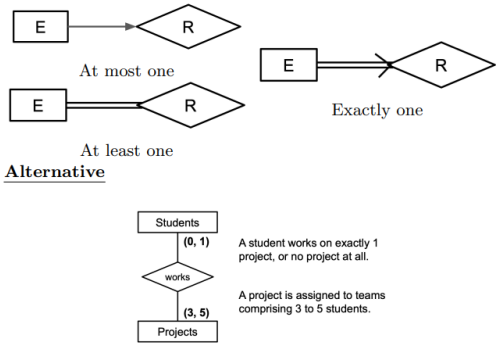
Cardinality (Ratio) constraints

• **Upper bound** for entity's participation



Participation / Existence Dependency constraints

• **Lower bound** for entity's participation
• Partial (default): participation not mandatory
• Total: mandatory (at least 1)

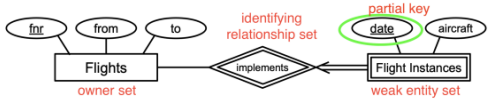


Recursive Relationship Type

• A relationship type between the same participating entity type in **distinct roles**.
• In ER diagram, need to display role names to distinguish participations.

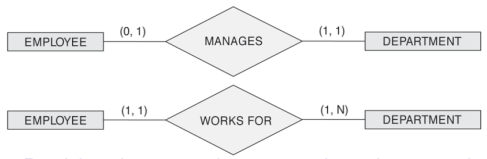
Dependency constraints

Weak Entity Types: No key attribute and is identification dependent on another entity.
• Participate in an identifying relationship type with an owner or identifying entity type
• Two kinds of dependencies:
Existence (no weak entity) dependency
Identification (weak entity) dependency
Partial Key
• Set of attributes of weak entity set that uniquely identifies a weak entity, for a given owner entity.



(min,max) notation for relationship constraints

• Read the min,max numbers next to the entity type and looking away from the entity type



7. EER Model

Enhanced ER or Extended ER

IS-A Hierarchies

• **"Is a" relationship** - used to model generalization/specialization of entity sets.
• **Hierarchy**: constraint that every subclass has only one superclass (single inheritance); basically a tree structure.
• **Lattice**: a subclass can be subclass of more than one superclass (multiple inheritance).

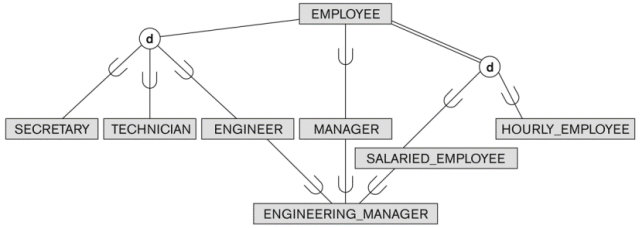


Figure 4.6
A specialization lattice with shared subclass ENGINEERING_MANAGER.

• **Subclass:** subclass member is entity in a **distinct specific role**. Entity inherits all attributes and relationships of superclass.
• **Specialization:** process of defining a set of subclasses of a superclass, based on **distinguishing characteristics**
• **Type of Specialization:** Can be Predicated defined, Attribute defined (Written beside joining line) or User defined.
• **Generalization:** reverse of specialization, based upon some **distinguishing characteristics**.

Constraints on Specialization / Gen

• **Disjointness Constraint:** Subclasses must be disjoint, entity can be member of at most one. (specified by **d** in EER)
• If not disjoint, specialization is **overlapping**, (specified by **o** in EER)
• **Completeness Constraint:** Total specifies every entity must be a member of some subclass. (specified by **double line** in EER)
• **Partial** allows entity not to belong to any subclass. (specified by **single line** in EER)
• Hence, we have four types: Disjoint/Overlapping x Total/-Partial. Note generalization usually is total.

8. Relational Mapping

• Preserve info, maintain constraints, minimize null.

1. Map Regular Entity Types

• Create new table (relation R), include all simple attributes.

2. Map Weak Entity Types

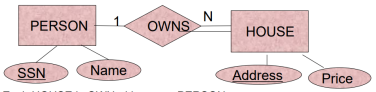
• Weak entity type, create table with corresponding FK.

3. Map Binary 1:1 Relationship Type

• Option 1: 2 Foreign Key (2 relations tables) option.
• 2: Merged relation option: Combine relationship set and either entity set into **one** table
• 3: Cross-reference (3 relations, one lookup table).

4. Map Binary 1:N Relationship Type

• For 1:N create table of participating entity (N-side).
• Include FK in table & attributes of the 1:N relation type.



Each HOUSE is OWNed by some PERSON
Hence, SSN of owner is like a mandatory attribute of each HOUSE.
Hence, it becomes a foreign key attribute in the relation HOUSE.
Mapping Rule: Transfer the primary key from the "one side" of the relationship type to the "many-side" of the relationship type and consider it as a Foreign Key.
PERSON (SSN, Name)
HOUSE (Address, Price, ……., Owner_SSN)
Foreign Key

5. Map Binary N:M Relationship Type

• Represent relationship set with a new table. Include as FK the PK of relations, combination will form PK of table.

6. Map Multi-valued Attributes

• Create new table / relation for each multivalued attribute.

7. Map N-ary relationship type

• For each n-ary relationship type R, where n >2, create new table. Include FK of relations, and attributes.

Summary:

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key

8. Map Specialization or Generalization

• Option 1: Multiple tables (relations) - Superclass and subclasses
• 2: Table each for each subclass relations, inherit superclass attributes
• 3: Single table with one type attribute
• 4: Single table with multiple type attributes
• For specialisation hierarchies with one superclass and n subclasses, possibility of mapping from 1 relation to (n+1) relations. Design highly subjective, try to maintain appropriate attributes to determine subclass identity.

9. Relational Calculus & Algebra

- Both are formal query languages. A query is composed of a collection of operators called relational operators.
- **Relational Calculus** (declarative language: what is to be done rather than how to do it). Order not specified, concerned with result we have to obtain.
- **Relational Algebra**: (procedural language) Order specified in which operations have to be performed.
- Relational algebra basis of implementation of relational DBMS.
- SQL queries translated into execution plans, orchestrate physical implementation of relational algebra operators.

Predicate Logic

- **Predicate / first order logic**: formulae built from:
- **predicates** (lowercase) , **operators** (=, etc.) **constants** (lower case), **variables** (upper case, quantified or free), **connectives** ($\rightarrow$), and **quantifiers** ($\forall$ and $\exists$).
- **Existential quantifier**, $\exists$: disjunction:

$$\exists X, F[X] \equiv F[a] \vee F[b] \vee ...$$

- **The universal quantifier**, $forall$: conjunction.

$$\forall X, F[X] \equiv F[a] \wedge F[b] \wedge ...$$

- **De Morgan laws** applies to quantifiers.

$$\neg(\exists X, F[X]) \equiv \forall X, \neg F[X]$$
$$\neg(\forall X, F[X]) \equiv \exists X, \neg F[X]$$

Relational Calculus (Tuple)

- Concerned with **result we have to obtain**.
- In **Tuple Relation Calculus (TRC)** variables values of rows of tables (t-uples.), is the theoretical basis of SQL.
- **Material Implication:** $P \rightarrow Q \equiv \neg P \vee Q$
- **Relational Calculus denoted as:**

$$\{t|P(t)\}$$

- **t**: set of tuples, **P**: condition which is true for the given set of tuples.
- E.g. $\{T|\exists T1(T1 \in$ department $\wedge$ T1.faculty = 'School of Computing' $\wedge T.$department = T1.department)$\}$

Relational Algebra

- Order is specified in which operations have to be performed.

Unary Operators

Selection, σ_c

- For each tuple $T \in R, T \in \sigma_c(R)$, means selection condition **c** evaluates to true for tuple t.
- E.g. Find the customers from Singapore.

$$\sigma_{c.country='Singapore'}(\rho(customers, c))$$

- Equivalent to:

```
SELECT * FROM customers c
WHERE c.country = 'Singapore'
```

- **Condition** is boolean expression of form:

expression	example
attribute **op** constant	$\sigma_{\text{start=2020}}$(Projects)
$attr_1$ **op** $attr_2$	$\sigma_{\text{start=end}}$(Projects)
$expr_1 \wedge expr_2$	$\sigma_{\text{start=2020} \wedge \text{end=2021}}$(Projects)
$expr_1 \vee expr_2$	$\sigma_{\text{start=2020} \vee \text{end=2021}}$(Projects)
$\neg\, expr$	$\sigma_{\neg(\text{start=2020})}$(Projects)
$(expr)$	-

- **op** $\in \{=, <>, <, \leq, \geq, >\}$
- Precedence: $()$, **op**, $\neg, \wedge, \vee$
- **null** comparison is **unknown**, arithmetic with **null** is **null**

Projection π_l

- Projects columns of a table specified in **list** l.
- `(SELECT xx, yy FROM games)`
- **Order of attribute in** l matters.
- Duplicates removed.
- **Examples:**

$$\pi_{g.name,g.version,g.price}(\rho(games, g))$$

Teams

en	pn	hours
Sarah	BigAI	10
Sam	BigAI	5
Sam	BigAI	3

$\pi_{\text{pn,en}}$(Teams)

pn	en
BigAI	Sarah
BigAI	Sam

Renaming, ρ_l

- Can change name of relation, of attributes, or both.
- Change name of relation: $\rho(R_1, R_2)$
- Change attribute names: $\rho(R_1, R_1(a_1 \rightarrow b_1, a_2 \rightarrow b_2))$

Set Operations

- Set operations include $\cup, \cap, \times$, set difference $(\backslash)$
- Intersection able to express with union and set difference:

$$R \cap S = (R \cup S) - ((R - S) \cup (S - R))$$

- **Union Compatability**: two relations must be union compatible. Have same number of attributes, corresponding attributes have same or compatible domains.
- (i.e. relations must have same columns).
- **Cross Product**: (Cartesian Product) Forms all possible pairs of tuples from two relations.

$$R_1 \times R_2$$

Join Operations

- Combines $\times, \sigma_c, \pi_l$ into a single op.
- Simple relational algebra expressions

Inner Joins

- Eliminates tuples that do not satisfy matching criteria (i.e. selection)
- Is a selection from cross product

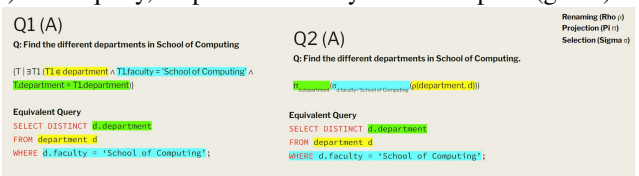
$$R \bowtie_C S = \sigma_C (R \times S)$$

- Example:

$$\rho(customers, c) \bowtie_{d.id=c.id} \rho(downloads, d)$$

Relational Calculus & Algebra

- **4 Steps to construct calculus and algebra queries:**
  1) Construct SQL query you are familiar with (difficult)
  2) From query, map the tables that you need (yellow)
  3) From query, map the conditional statements (blue)
  4) From query, map the columns you need to print (green)



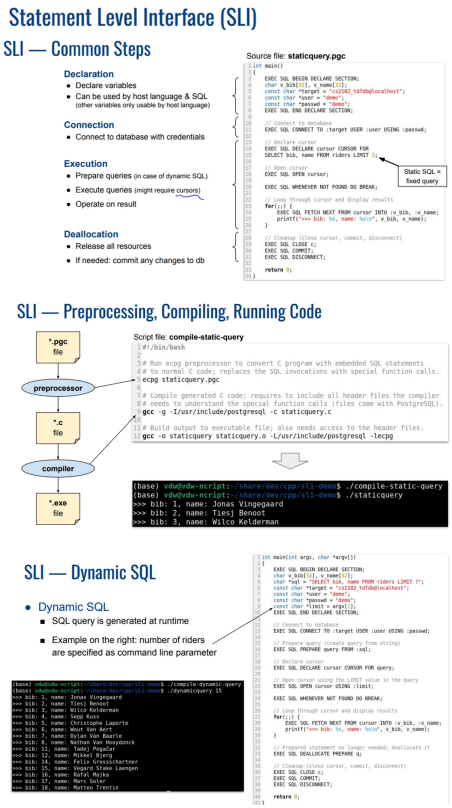
10. Programming with SQL

Writing Database Applications

- **Interactive SQL**: Directly writing SQL statements to an interface. (e.g. PostgreSQL's psql cli, pgAdmin).
- **Non-interactive SQL:** SQL statements included in application written in host language.
- 2 Main alternatives: **Statement Level Interface (SLI), and Call Level Interface (CLI).**
- Crudely, SLI = CLI in disguise, as SLI preprocessor generates CLI code.

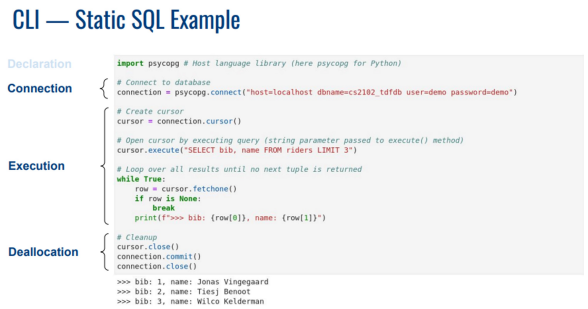
Statement Level Interface (SLI)

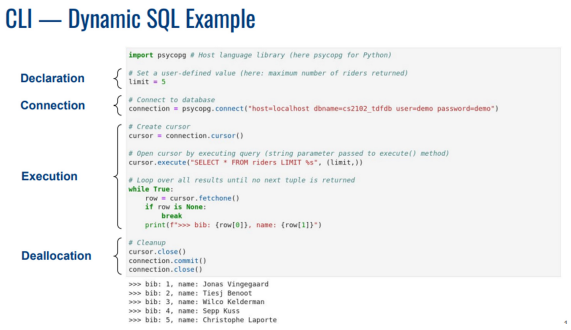
- Code is mix of host language statements and SQL statements (e.g. embedded SQL, dynmaic SQL).
- **Basic process for SLI**: Write code that mixes host langugae with SQL, preprocess code using a preprocessor, compile code into exe program.



Call Level Interface (CLI)

- Application completely written in host language, while **SQL statements are strings** passed as **arguments** to host language procedures or libraries
- E.g. ODBC (Open DataBase Connectivity), JDBC (Java DB Connectivity), psycopg library for Python - PostgreSQL.





SQL Injection Attack

- Class of cyber attacks on dynamic SQL, goal is to execute unintended (malicious) SQL statements.
- **Typical cause**: dynamic queries are generated by merging / concatenating strings.
- **Common attack point**: Omnipresent form fields in web interfaces. Entered values define some SQL statement.
- **Key Points**: Don't manually merge values to a query, don't use % or + operator to merge values, use provided methods.

11. SQL Functions and Procedures

- Tasks **requiring multiple DB operations** common, involve any combination of reads and writes.
- E.g. update user password: check user exists → check new password differs from old → if ok, update password (3 separate requests/accesses to DB)
- **Problems**: Application and DB may run on different machines, poor performance or DB becomes bottleneck.
- Different DB operations only loosely connected, difficult to ensure "all or nothing" behavior.
- **Approach:** Move (some) application logic into DB, group DB operations that form task together, treat task as single DB operation.

11. Stored Functions and Procedures

- **Collection of SQL statements and procedural logic**, precompiled and reusable code, allows execute multiple database operations as a single unit.
- **Procedural Logic:** Relevant for application logic that requires assignments, conditionals or loops, and queries that cannot be expressed using basic SQL.
- **ISO standard:** SQL/PSM (Persistent Stored Modules). Different DBMS have their own flavor.
- **Advantages:** better performance, code reuse, ease of maintenance, added security.
- **Disadvantages:** testing & debugging more challenging, limited portability / vendor lock in, no simple versioning of code, not the most intuitive language.

Stored Functions, Procedures

Syntax: Stored Functions

```
CREATE [OR REPLACE] FUNCTION <name> (<arg1>, <arg2>, …)
RETURNS <type> AS
$$
DECLARE
    -- Declaration of variables
BEGIN
    -- Sequence of SQL statement
    -- and/or procedural logic
    [RETURN …]
END;
$$
LANGUAGE <language>;
```

Name of function

Function arguments (mode, name, type)

Return type

Body of function (enclosed within dollar quotes; treated as a string)

Language used in body (mainly: `plpgsql` or `sql`)

- `CREATE OR REPLACE` helps to re-declare function/procedure if already previously defined
- Code is enclosed within `$$ <> $$`
- Calling a function: (USE SELECT, e.g.)
  `SELECT * FROM swap(2, 3);`
- Call a procedure: (USE CALL, e.g.)
  `CALL transfer( 'Alice', 'Bob', 100);`

Syntax: Stored Procedures

```
CREATE PROCEDURE add_bonus_proc(sid INT, amount INT)
AS
$$
    UPDATE students
    SET points = points + amount
    WHERE id = sid;
$$
LANGUAGE sql;
```

```
CALL add_bonus(3, 5);
```

No output / result, but table gets updated

- Syntax essentially same for procedures and functions, but procedures invoked using CALL command.
- **Obvious Difference**: Procedures no RETURNS clauses.
- **Functions** must return something (but can be VOID).
- **Procedures** do not have to return anything, but can (using INOUT and OUT params).

Function Arguments for Functions

- **Each argument described by 3 values**
- **Mode**: of argument (mainly IN, OUT, INOUT)
- **Name**: of argument (optional)
- **Type**: datatype of argument. (e.g. INT, VARCHAR)

IN	OUT	INOUT
Default	Explicitly specified	Explicitly specified
Value is passed to a function	Value is returned by a function	Value is passed to the function which returns another updated value
Behaves like constants	Behaves like an uninitialized variable	Behaves like an initialized variable
Value cannot be assigned	Value must be assigned	Value can/should be assigned

sql VS plpsql

- **sql:** Use where body consists of only SQL statements, often a wrapper of single / few SQL statments. Simpler syntax, no {BEGIN ... END}
- **PL/pgSQL:** Procedural Lang/ PostgreSQL, allows writing of procedural code providing control flows, variables, error handling. Statements generated at runtime, used for trigger functions.

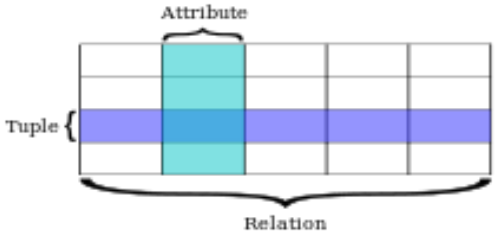
Function

```
CREATE OR REPLACE FUNCTION swap
    (INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
  temp INT;
BEGIN
  temp := val1; val1 := val2; val2 := temp;
END;
$$ LANGUAGE plpgsql;
```

Procedure

```
CREATE OR REPLACE PROCEDURE transfer
    (src TEXT, dst TEXT, amt NUMERIC)
AS $$
  UPDATE Accounts
  SET bal = bal - amt WHERE name = src;
  UPDATE Accounts
  SET bal = bal + amt WHERE name = dst;
$$ LANGUAGE sql;
```

Return and Type



Return	Type
One existing tuple from table	`<table_name>`
Set of tuples from table	`SETOF <table_name>`
Single new tuple	`RECORD`
Set of new tuples	`SETOF RECORD` or `TABLE(attributes...)`
No return value	`VOID`, or use `PROCEDURE` instead of `FUNCTION`
Trigger	`TRIGGER`

Important: If we use RECORD, we must have at least two OUT parameters. But if we use TABLE construct, we can just have one attribute.

Stored Functions vs. Procedures

- Procedures can **commit or roll back transactions** during execution, cannot be involved in DML commands (select, insert, update, delete).
- Procedures invoked in isolation using `CALL`, functions invoked in `SELECT` statements.
- **Best practice**: return value(s): create function, no return value: create procedure.

Assignments (of values of variables)

- **Basic Assignment** with `:=`, e.g. `age := 29;`
- **Assignment of query result** to declared variable(s): `SELECT ... INTO ...`

```
SELECT points INTO mark
FROM students WHERE id = sid;
```

Control Structures:

- **Conditionals:**
  - 4 types of `IF` expressions
    - `IF … THEN … END IF`
    - `IF … THEN … ELSE … END IF`
    - `IF … THEN … ELSIF … THEN … ELSE … END IF`
  - 2 types of `CASE` expressions
    - `CASE … WHEN … THEN … ELSE … END CASE`
    - `CASE WHEN … THEN … ELSE … END CASE`
- **Simple Loops**
  - `LOOP … END LOOP` (typically requires `EXIT…WHEN…` to jump out of loop)
  - `WHILE … LOOP … END LOOP`
  - `FOR … IN … LOOP … END LOOP`

- No curly braces or colons, hence additionaly keywords to indicate where loop begins and ends.
- `END IF` for conditionals, `END LOOP` for loops.
- **Simple example**: Compute sum of first n integers, if n is negative, return 0.

```
CREATE FUNCTION sum_n(IN n INT)
RETURNS INT AS $$
DECLARE sum INT;
BEGIN
    sum := 0;
    IF n <= 0 THEN
        RETURN sum;
    END IF;
    FOR val IN 1..n
    LOOP
        sum := sum + val;
    END LOOP;
    RETURN sum;
END; $$
LANGUAGE plpgsql;
SELECT * FROM sum_n(5);
```

We can also raise an exception if n is negative:

```
    IF n <= 0 THEN
        RAISE EXCEPTION 'n<0 error' ;
    END IF;
```

Errors & Messages

- `RAISE` keyword. 6 raise levels in PostgreSQL.

`RAISE DEBUG`	• Generate messages of different priority levels • Whether messages of a particular priority are reported to the client, depends on the PostgreSQL configuration
`RAISE LOG`	
`RAISE INFO`	
`RAISE NOTICE`	
`RAISE WARNING`	
`RAISE EXCEPTION`	• Raises an error • Typically aborts current transaction

Loop Use Case: Loop through Query Results

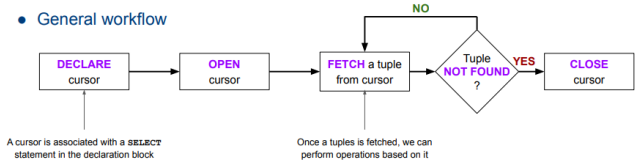
- Special FOR loop to iterate through results and manipulate data. (Common use case).
- $< target >$: Record or row variable that is successively assigned each row from query.
- $< statements >$: List of stmts using current target row.

```
FOR target IN query LOOP
    statements
    statements
END LOOP;
```

```
CREATE FUNCTION compute_points_gaps()
RETURNS TABLE(
    name TEXT, points INT, gap INT) AS $$
DECLARE
    s RECORD; prev INT:= -1;
BEGIN
    FOR s IN SELECT *
        FROM students ORDER BY points DESC
    LOOP
        name := s.name;
        points := s.points;
        IF prev >= 0 THEN
            gap := prev - s.points;
        ELSE
            gap := 0;
        END IF;
        RETURN NEXT; -- (OUTPUT) TABLE TUPLE
        prev := s.points;
    END LOOP;
END; $$
LANGUAGE plpgsql;
```

Cursors

- **Purpose**: Declare on a query, access each indiv row.
- Helps avoids memory overrun when the query result is large (don't access whole query at once).




```
CREATE FUNCTION compute_points_gaps()
RETURNS TABLE(
    name TEXT, points INT, gap INT) AS $$
DECLARE
    c CURSOR FOR (SELECT * FROM students ORDER
        BY points DESC);
    s RECORD; prev INT;
BEGIN
    prev := -1;
    OPEN c;
    LOOP
        FETCH c INTO s;
        EXIT WHEN NOT FOUND;
        name := s.name;
        points := s.points;
        IF prev >= 0 THEN
            gap := prev - s.points;
        ELSE
            gap := 0;
        END IF;
        RETURN NEXT; -- (OUTPUT) TABLE TUPLE
        prev := s.points;
    END LOOP;
    CLOSE c;
END; $$
LANGUAGE plpgsql;
-- ** To check NULL: IF high IS NULL THEN,
    not IF high = NULL THEN
```

Advantage of Cursor (over for loops etc.):

- Flexible 'navigation' through query results in **different directions**.
- `FETCH` to move row and read data.
- `MOVE` only to move to row (no read).

Cursor Directions

NEXT	Fetch the next row (default)
PRIOR	Fetch the prior row
FIRST	Fetch the first row of the query (same as `ABSOLUTE 1`)
LAST	Fetch the last row of the query (same as **ABSOLUTE -1**)
ABSOLUTE *n*	• Fetch the *n*-th row of the query, if *n* >=0 • Fetch abs(*n*)-th row from the end, if *n* < 0. • **ABSOLUTE** 0 positions before the first row
RELATIVE *n*	• Fetch the *n*-th succeeding row, if *n* >= 0 • Fetch the abs(*n*)-th prior row, if n < 0 • Position before first row or after last row if *n* is out of range • **RELATIVE** 0 re-fetches the current row, if any
FORWARD	Fetch the next row (same as **NEXT**)
BACKWARD	Fetch the prior row (same as **PRIOR**).

Examples

- Using FETCH ABSOLUTE, FETCH NEXT, FETCH RELATIVE to calculate median points().
- **Dynamic cursors**: Cursors can also have inputs, which are taken from function inputs, that affect the query results.
- SELECT median_points(TRUE);
  vs SELECT median_points(FALSE);

Cursors — Example (beyond NEXT)

```
CREATE OR REPLACE FUNCTION median_points()
RETURNS NUMERIC AS
$$
DECLARE
    c CURSOR FOR (SELECT * FROM students ORDER BY points DESC);
    s1 RECORD; s2 RECORD; num_students INT;
BEGIN
    OPEN c;
    SELECT COUNT(*) INTO num_students FROM students;
    IF num_students%2 = 1 THEN
        FETCH ABSOLUTE (num_students+1)/2 FROM c INTO s1;
        RETURN s1.points;
    ELSE
        FETCH ABSOLUTE num_students/2 FROM c INTO s1;
        FETCH NEXT FROM c INTO s2;
        RETURN (s1.points+s2.points)/2;
    END IF;
    CLOSE c;
END;
$$
LANGUAGE plpgsql;
```

Dynamic Cursors — Example

```
CREATE OR REPLACE FUNCTION median_points(IN has_graduated BOOLEAN)
RETURNS NUMERIC AS
$$
DECLARE
    c CURSOR (grad BOOLEAN) FOR (SELECT * FROM students
                                 WHERE graduated = grad
                                 ORDER BY points DESC);
    s1 RECORD; s2 RECORD; num_students INT;
BEGIN
    OPEN c(has_graduated);
    SELECT COUNT(*) INTO num_students
    FROM students WHERE graduated = has_graduated;
    IF num_students%2 = 1 THEN
        FETCH ABSOLUTE (num_students+1)/2 FROM c INTO s1;
        RETURN s1.points;
    ELSE
        FETCH ABSOLUTE num_students/2 FROM c INTO s1;
        FETCH NEXT FROM c INTO s2;
        RETURN (s1.points+s2.points)/2;
    END IF;
    CLOSE c;
END;
$$
LANGUAGE plpgsql;
```

12. Triggers

Motivation

- **Model constraints**, where user is not forced to call stored procedure or funtion.
- Example constraints: restricted change in data, derived values, set cardinality constraints.
- **Application Requirement**: E.g. automatic logging of changes.
- We could create a stored procedure that combines insertion and logging, but users can circumvent this by directly running `INSERT INTO` instead of calling procedure.

Trigger Concept (Trigger fires Trigger Function)

- Triggers has **Event-Condition-Action** rule.
- When event occurs, test condition, and if satisifed, perform action.
- **ECA rule** is split into 2 parts:
  Event and Condition under **Trigger**, and
  Action under **Trigger Function**.

Trigger

```
CREATE TRIGGER on_student_entered
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_basic();
```

Trigger Function

```
CREATE FUNCTION log_student_basic()
RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO basic_logs (student) VALUES (NEW.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Triggers

- Triggers can listen on multiple event types.

```
CREATE TRIGGER
    on_student_modified_advanced
AFTER INSERT OR DELETE OR UPDATE ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_advanced();
```

- Triggers options: **Event, Timing, Granularity.**
- **Event**: Situation that fires trigger.
- **Timing**: When trigger is fired (BEFORE or AFTER) for tables, (INSTEAD OF) for views.
- **Granularity**: Specifies if triggered for each affect row or only once. (FOR EACH ROW / F.E. STATEMENT)

Trigger Event types:

```
INSERT ON table
DELETE ON table          → TG_OP = 'INSERT' | 'DELETE' | 'UPDATE'
UPDATE [OF column] ON table
```

Access to transition variables:

NEW: modified row *after* the triggering event
(can only exist for INSERT and UPDATE operations)

OLD: modified row *before* the triggering even
(can only exist for DELETE and UPDATE operations)

	NEW	OLD
INSERT	✔	✘
UPDATE	✔	✔
DELETE	✘	✔

Trigger Timing:

AFTER	Triggers fires after the operation has completed (this includes that any relevant constraints have been checked)
BEFORE	Trigger fires before the operation is attempted
INSTEAD OF	Trigger fires if an operation on a view is attempted

Importance of timing
- BEFORE and INSTEAD OF triggers can skip or modify the operation
- For BEFORE and INSTEAD OF triggers the return value of the trigger function matters

	RETURN value	
	NULL tuple	non-NULL tuple t
BEFORE INSERT	No tuple inserted	Tuple t inserted
BEFORE UPDATE	No tuple updated	Tuple t updated
BEFORE DELETE	No tuple deleted	Tuple t deleted
AFTER INSERT	No effects!	
AFTER UPDATE		
AFTER DELETE		

Views (Recap) and Triggers

- **Views**: virtual table, a permanently named query.
- Query results not permanently stored, executed each time query used, hides complexity, heavily used.

```
CREATE VIEW <name> AS
    SELECT ... FROM ... ...;
```

- **Updateable Views**: Must have only one entry in FROM clause, no GROUP BY / LIMIT etc, no UNION, INTERSECT, no aggregate functions. Otherwise direct modification not possible.
- Triggers useful for (non-updateable) views.
  (Trigger Timing: (INSTEAD OF) for views.)

Trigger Granularity:

- **Row-level triggers**
  - Trigger function is executed for each affected row
  - Keyword: `FOR EACH ROW`
- **Statement-level triggers**
  - Trigger function is executed once for each transaction (no matter how many rows are affected)
  - Keyword: `FOR EACH STATEMENT`
  - Ignored return value of trigger function (Enforcing a rollback requires `RAISE EXCEPTION`!)

Example for a statement-level trigger

- Prohibit the deletion of rows from the logs
- Show warning to user only once no matter how many rows the user attempted to delete

```
CREATE TRIGGER on_delete_from_log
BEFORE DELETE ON advanced_log
FOR EACH STATEMENT
EXECUTE FUNCTION show_warning();
```

```
CREATE FUNCTION show_warning()
RETURNS TRIGGER AS
$$
BEGIN
    RAISE EXCEPTION 'Do not DELETE the logs!!!';
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Trigger Conditions

- Execute trigger function only if condition is true.
- Instead of having condition in trigger function, we move to trigger, and only execute if condition is true.
- **Condition**: generally can formulate any boolean expression, but no SELECT, OLD for INSERT, NEW for DELETE, in the WHEN() clause.

```
CREATE TRIGGER on_student_updated_advanced
AFTER UPDATE ON students
FOR EACH ROW WHEN (NEW.points <> OLD.points)
EXECUTE FUNCTION log_student_advanced();
```

Deferrable Triggers

- Triggers run immediately for every statement that fire them.
- Operations of multiple statements yielding intermediate inconsistent states
- **Deferred triggers**: Run trigger only at end of transactions.

```
CREATE CONSTRAINT TRIGGER on_account_modified
AFTER INSERT OR DELETE OR UPDATE ON accounts
DEFERRABLE INITIALLY IMMEDIATE
FOR EACH ROW
EXECUTE FUNCTION check_balance();
```

- Only work for AFTER and FOR EACH ROW triggers.
- Both CONSTRAINT and DEFERRABLE must be specified.
- INITIALLY DEFERRED: trigger deferred by default.
- INITIALLY IMMEDIATE: trigger not deferred by default (but can be deferred on demand).

```
BEGIN TRANSACTION;
SET CONSTRAINTS on_account_modified DEFERRED;
UPDATE accounts SET balance = balance - 50
    WHERE id = 10;
UPDATE accounts SET balance = balance + 50
    WHERE id = 11;
END TRANSACTION;
```

Other Trigger Notes

- **Trigger Order**: Triggers for same event on same table:
- Order of activation: BEFORE statement-level, BEFORE row-level, AFTER row-level, AFTER statement-level.
- Within each, fired in alphabetic order. If BEFORE row-level trigger returns NULL, subsequent triggers on same row omitted.

Trigger Functions

- Trigger functions do not take in (ordinary) arguments.
- Must have return type TRIGGER
- Must be defined before trigger itself.
- **"Input"**: Special internal data structure from trigger.

Useful Data available in trigger function

- When function is called as a trigger, several special variables created automatically in top level block.

`TG_NAME`	Name of the trigger that fired
`TG_OP`	Operation that fired the trigger (`INSERT`, `UPDATE`, `DELETE`)
`TG_WHEN`	Time when the trigger was fired (`BEFORE`, `AFTER`, or `INSTEAD OF`)
`NEW`	Record holding the new row for `INSERT`/`UPDATE` operations
`OLD`	Record holding the old row for `UPDATE`/`DELETE` operations
...	...
`TG_ARGV[]`	Array of arguments from the `CREATE TRIGGER` statement.

Trigger Function Example

- Schema: advanced_logs(student: INT, operation: TEXT, p_old: INT, p_new: INT, created_at: TIMESTAMP)
- Log all events that might modify students' points (INSERT, UPDATE, DELETE)

```
CREATE OR REPLACE FUNCTION
    log_student_advanced()
RETURNS TRIGGER AS $$ BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO points_log_advanced VALUES
            (NEW.id, TG_OP, NULL, NEW.points,
            DEFAULT);
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        INSERT INTO points_log_advanced VALUES
            (OLD.id, TG_OP, OLD.points, NULL,
            DEFAULT);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO points_log_advanced VALUES
            (OLD.id, TG_OP, OLD.points,
            NEW.points, DEFAULT);
        RETURN NEW;
    END IF;
END; $$ LANGUAGE plpgsql;
```

13. Basics of Functional Dependencies

Chapter Outline

1. Informal Design Guidelines for Relational Databases
2. Functional Dependencies (FDs)
3. 3 Normal Forms based on Primary Keys
4. 4 General Normal Form Definitions for 2NF, 3NF (Multiple Candidate Keys)
5. BCNF (Boyce-Codd Normal Form)

Goals

- Relational Database Design as a practical activity followed in large organizations worldwide. Relational model dominates the commerical market.
- Have some informal guidelines that can point out problems with relational design.
- Understand theoretical basis for analyzing designs called **functional dependencies**.
- Understand and utilize process of **normalization** to "improve" / "purify" poor designs.
- Understand formal basis for synthesizing good relations strictly based on knowledge of dependencies among attributes.

Informal Design Guidelines for Relational Databases

- **Relational Database Design**: Grouping of attributes to form "good" relation schemas.
- Two levels of relation schemas': Logical "user view" level & storage "base relation" level.

Criteria for "Good" Design

1. **minimality**: should express information with minimum number of distinct relations.
2. **lack of redundancy**: should minimize amount of redundancy among relations.
3. **Information preservation**: should preserve all information captured by the conceptual design (in terms of entity types, relationship types, attributes, and constraints).
4. **consistency**: among the relations
5. **efficiency**: (beyond course scope). Typically addressed in the physical design.

Guidelines Summary

1. **Guideline 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes)**
   - Bottom line: Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

2. **Guideline 2: Design a schema that does not suffer from update anomalies: (Insertion, Deletion, Modification anomalies).**
   - If there are any anomalies present, then note them so that applications can be made to take them into account. (e.g. introduced for reasons as attributes needed for reporting or accounting purposes).
   - Introduction of anomalies referred to as De-Normalization.

3. **Guideline 3: Relations should be designed such that their tuples will have as few NULL values as possible.**
   - Attributes that are NULL frequently could be placed in separate relations (with the primary key).
   - Reasons for nulls: Attribute not applicable or invalid, Attribute value unknown (may exist), Value known to exist, but unavailable.

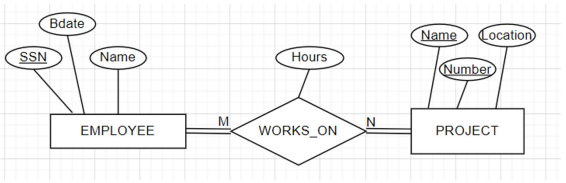
4. **Guideline 4: Avoid generation of "spurious data" when tables are joined – an absolute "MUST".**
   - Bad designs for a relational database may result in erroneous results for certain JOIN operations. Generating bad data cannot be accepted at any cost.
   - The "lossless join" (non-additive) property is used to guarantee that join operation will not create bad data.
   - The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.

5. Tool for analysis: **functional dependency**. Methodology for "fixing" (bad) designs is specified by process of "**Normalization**."

Guideline 1: Don't mix Relations

- **Guideline 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes)**
  - Do not mix attributes of different entities in same relation.
  - Only use foreign keys to refer to other entities.
  - Entity and relationship attributes be kept apart as much as possible.
  - E.g. (EMPLOYEEs, DEPARTMENTs, PROJECTs) attributes should not be mixed in the same relation (table).

ER Diagram: (Portion)



Bad Relational Design:

EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)

Update / Modification Anomaly:

- Changing name of project number P1 from "X" to "Y" may cause update to be made for all 100 employees working on project P1.

Insert Anomaly:

- Cannot insert a project unless an employee is assigned to it. + Converse on inserting employee.

Delete Anomaly:

- When project deleted, result in deleting all employees who work on that project.
- Conversely, if employee sole employee on project, deleting employee -> delete correspond. project

Correct Relational Design:

EMPLOYEE(Ssn, Fname, .., Lname,.., $Dno_{(FK)}$)

PROJECT (Pnumber, Pname, Plocation, $Dnum_{(FK)}$)

WORKS_ON ($Ssn_{(FK)}$, $Pnumber_{(FK)}$, No-hours)

Optimalization:

- Top 2 relations strictly stand for an entity type
- Third relation strictly stands for a relationship type
- No mixing of entity and relationship information as done in previous example
- **NET RESULT:** Design DOES NOT suffer from any of the anomalies.

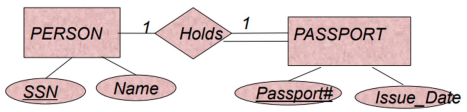
Guideline 2: Avoid designs with anomalies

- Design a schema that does not suffer from the insertion, deletion and update anomalies.
- If there are any anomalies present, then note them so that applications can be made to take them into account.
- **Corollary of guideline 1**: Basically states that when you are forced to mix descriptor attributes of an entity type into the table for another entity type or a relationship type (for performance reasons or reporting requirements etc.), you should document them and take care of the consistency preservation via the application.

Guideline 3: Minimize Null values in Tuples

- Relations should be designed such that their tuples will have as few NULL values as possible
- Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- **Reasons for nulls**:
  - Attribute not applicable or invalid
  - Attribute value unknown (may exist)
  - Value known to exist, but unavailable.

ER Diagram: (Portion)



Suppose only 60% of persons in the database of 2 million persons hold a Passport

Bad Relational Design:

PERSON (SSN, Name, Passport#, Issue_date)

Null Values:

- Will contain 40% of tuples with NULL values for Passport# and Issue_date.

Correct Relational Design:

PERSON (SSN, Name) – 2 million tuples

PASSPORT (Passport#, Issue_date, $SSN_{(FK)}$)
– 1.2 million tuples

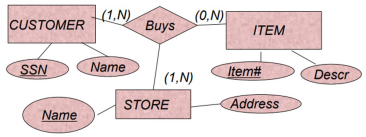
Optimalization:

- No null values

Guideline 4: Relations satisfy lossless (Non-Additive) join condition to avoid generating spurious tuples.

- Bad designs for relational database: **erroneous results for certain JOIN operations**. The relations should be designed to satisfy the lossless join condition, guarantee meaningful results for join operations.
- No spurious tuples should be generated by doing a natural-join of any relations. Applies to a natural join among any pairs or collections of relations.
- How to know whether a decomposition will be lossless: Use functional dependencies and algorithm (algo 15.3, chapter 15 test losslessness property of an n-way decomposition).

ER Diagram: (Portion)



Each instance of BUYS relationship type relates one customer, one item, one store. Customer buys at least one to N items, an item may not be sold, but can be sold in any number of BUYS transaction (0, N).

Correct Relational Design:

BUYS($SSN_{(FK)}$, $Item\#_{(FK)}$, $Store\_name_{(FK)}$, ...)

Right way to map this ternary relationship is to create one relation for "BUYS".

Bad Relational Design:

Cust_Item (Ssn, Item#)

Cust_Store(Ssn, Store_name)

The two resulting tables:

Cust_Item

SSN	ITEM#
s1	x1
s1	x2
s2	x1
s2	x2

Cust_Store

SSN	STORE_name
s1	t1
s1	t2
s2	t2
s2	t1

*Select **
From Cust_Item x Inner Join Cust_Store t on x.ssn = t.ssn

Ssn	Item#	Store_name	
s1	x1	t1	
s1	x1	t2	BAD
s1	x2	t1	BAD
s1	x2	t2	
s2	x1	t2	
s2	x1	t1	BAD
s2	x2	t2	BAD
s2	x2	t1	

- The **BAD tuples are spurious** (incorrect/invalid) data that resulted from the BAD design
- The ternary relation BUYS (Cust_ssn, Item#, Store#) was wrongly decomposed into the 2 relations.
- **There are two important properties of decompositions:**
  a) Non-additive or losslessness of the corresponding join
  b) Preservation of the functional dependencies.

Functional Dependencies (FDs)

- Used to specify **formal measures** of the "goodness" of relational designs.
- And keys (derived from FDs) are used to define **normal forms** for relations.
- FDs are **constraints** that are derived from the meaning of and interrelationships among the data attributes.
- A set of attributes X **functionally determines** a set of attributes Y if **value of X determines a unique value for Y**.

Defining Functional Dependencies

- $X \rightarrow Y$ holds if whenever two tuples have the same value for X, they must have the same value for Y.
- For any two tuples $t1$ and $t2$ in any relation instance: r(R): If t1[X] = t2[X], then t1[Y] = t2[Y]
- $X \rightarrow Y$ in R specifies constraint on all $relat^n$ instances r(R).
- Written as $X \rightarrow Y$; can be displayed graphically on a relation schema as in Figures. (denoted by the arrow).
- FDs are derived from the real-world constraints on the attribute.

Bad Relational Design:

Figure 14.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.

(a)

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn

(b)

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation

FD1
FD2
FD3

Examples of FD Constraints:

- SSN → Ename
- Pnumber → {Pname, Plocation}
- {SSN, Pnumber} → Hours

- An FD is a (semantic, logical) property of the attributes in the schema R.
- The constraint must hold on every relation instance r(R).
- If K is a key of R, then K functionally determines all attributes in R.
- We never have two distinct tuples with t1[K] = t2[K], i.e., the projection of each tuple in a relation on the K column(s) must yield distinct values.

Defining FDs from Instances

- To **define** FDs, need to understand meaning of attributes involved and relationship between them.
- Given the instance (population) of a relation, all we can conclude is that an FD **may exist** between certain attributes.
- What we can definitely conclude is – that certain FDs **do not exist** because there are tuples that show a violation of those dependencies.

Defining Functional Dependencies (from Instance):

- A relation *R*(A, B, C, D) with its extension.
- Which FDs *may exist* in this relation?

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

FD Constraints that may exist:

- {A, B} → C, {A, C} →D, {A, B} → {C, D}

FD that cannot exist (ruled out): ✕

- A → B, B → A, C → D, D → C, A → {B, C}, C → {B, D}, {B, C} → A.

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

FD Constraints that may exist:

- Text → Course

FD that cannot exist (ruled out): ✕

- Teacher → Course, Teacher → Text, Course → Text

Normalization of Relations

- **Normalization**: The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations by the process of decomposition.
- **Normal form**: Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form.
- **2NF, 3NF, BCNF** based on keys and FDs of a relation schema
- **4NF**: based on keys, multi-valued dependencies: MVDs;
- **5NF**: based on keys, join dependencies : JDs;
- Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation; see Chapter 15)