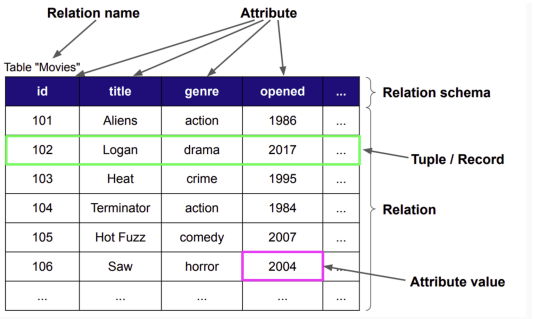
Basic Terminology

- Data Model - collection of concepts to describe data
- Schema - description of structure of db
- Schema Instance - contents of db at a particular time



Relational Data Model

Data modeled using **relations** (tables with row and col).

Rows & Cols

- x rows = **cardinality** of x rows
- y cols = **degree** of y.
- **number of rows don't include headers!**

Relation Schema

Contains attributes and constraints

- Attribute - possible values that satisfies the constraints
- Domain constraints - something like function definition - Students(name:**text**)
- Tuple/Record - a row in the table represented as a tuple - ('Bob'). Relation is a set of tuples.
- Component - a field in the col

Domain

- Atomic - can't be split further (usually for strings?), if a string for a name has the format 'first name, last name', it can't be split into first name and last name individually.
- Domain - set of atomic values (e.g. integer, numeric, text)
- null - can be an attribute, represents N.A or unknown.
- Can include **null** even if not shown.
- Value of attribute is either in domain or null.

Example of domains :

- Consider the relation schema **Lectures(course, day, hour)**
  - domain(course) = {'cs101', 'cs203', 'cs305'}
  - domain(day) = {1, 2, 3, 4, 5}
  - domain(hour) = {8, 10, 12, 14, 16}
- Each instance of `Lectures` is a subset of

{'cs101', 'cs203', 'cs305', null} × {1, 2, 3, 4, 5, null} × {8, 10, 12, 14, 16, null}

course	*day*	*hour*
cs101	1	8
cs101	1	10
cs101	1	12
cs101	1	14
cs101	1	16
cs101	2	8
⋮	⋮	⋮
null	null	16
null	null	null

Relational Database Schema

Relational Db Schema = Relational Schemes + Data Constraints.

- Relational Db - collection of tables
- Relational Db Schema - relation schema of the tables in the relational db.
- Relation schema - schema of 1 table
- Db schema - schema of all the tables in db
- Relation/Db instance - values of relation/db at a particular time

Integrity Constraints (IC)

Condition that limits data that can be stored.

- Can be enforced - check if constraints are violated
- Legal - satisfies all ICs.

Integrity Constraints (ICs) (cont.)

- Without any additional integrity constraints, each instance of $R(A_1, \cdots, A_n) \subseteq \{(a_1, a_2, \cdots, a_n) \mid a_i \in domain(A_i) \cup \{null\}\}$

Students

studentId	*name*	*birthDate*	*cap*
3118	Alice	1999-12-25	3.8
1423	Bob	2000-05-27	4.3
5609	Carol	1999-06-11	6.5
1423	Dave	2000-10-05	3.7

Courses

courseId	*name*	*credits*
101	Programming in C	5
112	Discrete Mathematics	4
204	Analysis of Algorithms	4
null	Compiler Design	4
311	Database Systems	5

Enrolls

sid	*cid*	*grade*
3118	101	5.0
3118	112	4.0
3118	202	3.0
1423	112	3.7
5609	101	4.5

In the figure above, IC is not satisfied. In domain(cid) which takes from domain(courseId), course id of 202 does not exist, hence IC is not satisfied here.

Key Constraints

Attribute can take multiple key identities. A primary key is a candidate key, a primary key can be a foreign key, etc.

SuperKey

Subset of attributes that uniquely identifies tuples. Sets that contain superkeys are superkeys as well.

- **Example**: Which of the following is a superkey for the relation Students (studentId, name, birthDate, cap)?
  - {studentId}
  - {name}
  - {birthDate}
  - {cap}
  - {studentId, name}
  - {studentId, birthDate}
  - {studentId, cap}
  - {name, birthDate}
  - {name, cap}
  - {birthDate, cap}
  - {studentId, name, birthDate}
  - {studentId, name, cap}
  - {studentId, birthDate, cap}
  - {name, birthDate, cap}
  - {studentId, name, birthDate, cap}

In the figure above, only studentId is unique. So the superkeys would be the tuples that contain studentId in them.

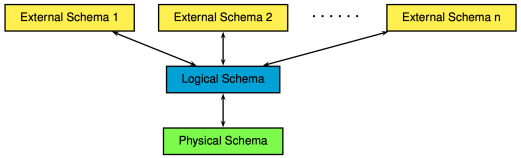
Key

- **Minimal** subset of attributes in a relation that uniquely identifies tuple
- Size of key can be more than 1.
- Attribute values cannot be null (must always have keys)
- Candidate key - multiple keys
- Primary key - one of the candidate key
- Foreign key - refer to **primary key** of another relation. Constraint : present or null. Within same relation, can reference other keys.

- **Example**: Which of the following is a superkey for the relation Students (studentId, name, birthDate, cap)?
  - {studentId}
  - {name}
  - {birthDate}
  - {cap}
  - {studentId, name}
  - {studentId, birthDate}
  - {studentId, cap}
  - {name, birthDate}
  - {name, cap}
  - {birthDate, cap}
  - {studentId, name, birthDate}
  - {studentId, name, cap}
  - {studentId, birthDate, cap}
  - {name, birthDate, cap}
  - {studentId, name, birthDate, cap}

In the figure above, only studentId is unique. So the key would be the tuple that contain studentId.

Data Abstraction Levels



- Physical - db implementation
- Logical - logical structure of db
- External - customized view of logical for users. May differ between users.

Data Independence

- Physical Data Independence - protect from changes in physical schema
- Logical Data Independence - protect from changes in logical schema

Transactions

ACID properties :

- Atomicity - either all effects are reflected or none are
- Consistency - transactions can't violate integrity
- Isolation - doesn't interfere with other transactions.
- Durability - recoverable on system crash

Sequential transactions are transactions that occur one after another. Concurrent transactions are transactions that occur at the same time (can lead to incorrect result because of interleaving).

Relational Algebra

- **Operands** - Table
- **Operator** - Used on table, outputs a table
- **Unary Operators** - 1 input
- **Binary Operators** - 2 inputs
- Satisfies **closure property**.
- Outputs are **always** table (or sets).
- Can be composed to form relational algebra expressions
- Input operands **not modified**, creates just a copy.
- **Close World Assumption** - data that don't appear in relation is **False**.
- Order of records don't matter - its a set of records

Selection Condition

attribute **op** (constant or another attribute). <> is not equals operator, rest are the same. **Precedence** : () , op, not, and, or

Null

Result of comparison is **unknown**. Result of arithmetic operation is **null**.

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE	
TRUE	FALSE	FALSE	TRUE	
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	

Selection

Symbol : σ_c. Filters records based on condition c. Output has same schema as input. $\sigma_{price<20}$.

Projection

Symbol : π_l. Filters attributes based on condition. Schema of output determined by l. Every attribute in l must exist in input. **No duplicates** in output.

Renaming

Symbol : ρ_l. Renames attributes on a list of renamings l. Format of l : old name : new name ... The original name has to exist in the input, and renamed **at most once**. Order doesn't matter.

Union Compatibility

Both conditions must be satisfied :

- Same number of attributes (columns)
- Attributes have same domain (types)

Domains must match for every attribute of both, pairwise. Attribute names need not be the same.

Set Operators

Output relation has attributes of the first operand.

- **Union** : $R \cup S$ - returns tuples that occur in both R or S
- **Intersection** : $R \cap S$ - returns tuples that occur in both R and S
- **Set-difference** : $R - S$ - returns tuples in R not S

For union, output table will follow the attributes of the first operand.

Cross-Product

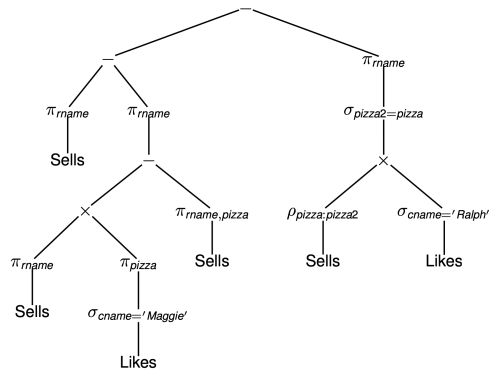
Symbol : $\times$. If $R \times S$, then every tuple in R is mapped to every entry in S. Total number of attributes = $R + S$. Total number of entries $\leq R^S$. Is associative, $(R \times S) \times T = R \times (S \times T)$. Can return no tuples if either table is empty.

Lexicographical Ordering

$C_1 < C_2$ ensures that only one occurrence appears in the relation. Otherwise, there will be repeat.

Writing Queries

Operator Trees



To get expression, perform **post-order traversal** of tree (Left → Right → Root)

Sequence of steps

$$R_1 = \pi_{pizza}(\sigma_{cname='Maggie'}(\text{Likes}))$$
$$R_2 = \pi_{rname}(\text{Sells}) \times R_1$$
$$R_3 = \pi_{rname}(R_2 - \pi_{rname,pizza}(\text{Sells}))$$
$$R_4 = \pi_{rname}(\text{Sells}) - R_3$$
$$R_5 = \rho_{pizza:pizza5}(\sigma_{cname='Ralph'}(\text{Likes}))$$
$$R_6 = \pi_{rname}(\sigma_{pizza5=pizza}(\text{Sells} \times R_5))$$
$$\text{Answer} = R_4 - R_6$$

Inner Join

Symbol : $R \bowtie_c S$. $R \bowtie_c S = \sigma_c(R \times S)$. Map entries in R to every entry in S. Then select tuples based on some condition c.

Natural Join

Symbol : $R \bowtie S$

- renames S
- perform **inner join**, with $c = (a_1 = b_1) and ... (a_n = b_n)$
- filter out repeated attributes (ie, those that were renamed)

Use to collate data into one big table. For example, R has attributes A, B and S has attributes A, C, D. Doing a natural join results in a table with attributes A, B, C, D.

Dangling Tuple

Tuples that don't participate in inner join operations.

Left Outer Join

Symbol : $R \rightarrow_c S$. Adds dangling tuples from R to the resulting relation.

Right Outer Join

Symbol : $R \leftarrow_c S$. Adds dangling tuples from S to resulting relation.

Full Outer Join

Symbol : $R \leftrightarrow_c S$. Adds dangling tuples from R and S to resulting.

R

D	B	A
0	x	100
2	y	100
4	w	400
5	z	200

S

E	A2	D2	C
a	100	0	i
b	300	1	j
c	200	5	k

Inner join: $R \bowtie_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k

Left outer join: $R \rightarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
2	y	100	null	null	null	null
4	w	400	null	null	null	null

Right outer join: $R \leftarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
null	null	null	b	300	1	j

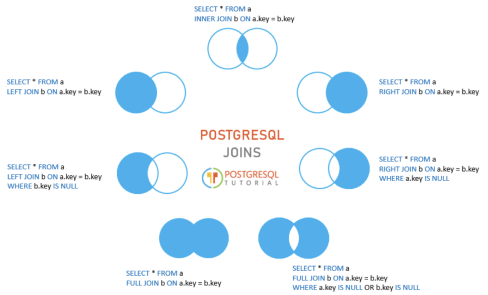
Full outer join: $R \leftrightarrow_{D=D2} S$

D	B	A	E	A2	D2	C
0	x	100	a	100	0	i
5	z	200	c	200	5	k
2	y	100	null	null	null	null
4	w	400	null	null	null	null
null	null	null	b	300	1	j

Null values will be placed at the attributes of which the dangling tuple did not come from.

Natural Left/Right/Full Outer Join

- **Natural left outer join**
  $R \rightarrow S = \pi_l(R \rightarrow_c \rho_{a_1:b_1 \ldots a_n:b_n}(S))$. Natural join with dangling tuples from R.
- **Natural right outer join**
  $R \leftarrow S = \pi_l(R \leftarrow_c \rho_{a_1:b_1 \ldots a_n:b_n}(S))$. Natural join with dangling tuples from S.
- **Natural full outer join**
  $R \leftrightarrow S = \pi_l(R \leftrightarrow_c \rho_{a_1:b_1 \ldots a_n:b_n}(S))$. Natural join with dangling tuples from R and S.

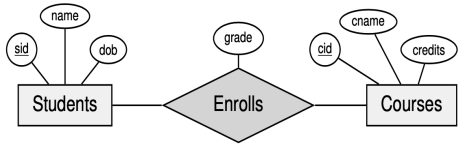


Database Design Process

- Talk to end users
- Construct high level idea
- Implement high level idea to actual schemas
- Improve schemas - cleaning up
- Improve performance - low level
- Security

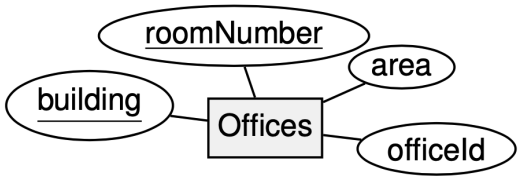
Entities & Attributes

- **Entities** - Objects of interest, represented as rectangles
- **Relationship** - Relationship between entities, represented as diamonds
- **Attribute** - information about entity or relationship, represented as ovals



Entities

- Underlined attributes **form a primary key**.

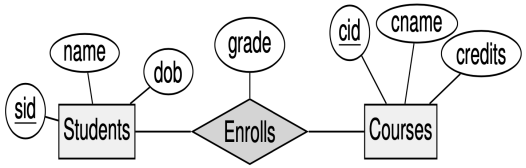


Primary key in the above example is **(building, roomNumber)**

Relationships

Association between 2 or more entities.

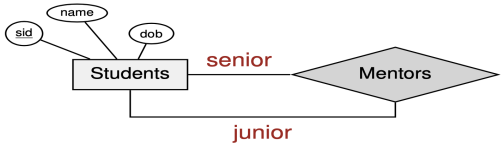
Many-to-Many



Each <**entity**> <**relationship**> 0 or more <**entity**>. Works both ways

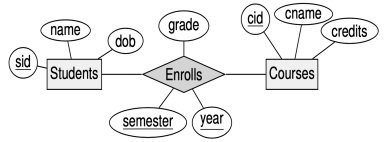
Roles

- Roles in relationship
- Don't have to explicitly mention
- Label the edges involved
- Usually when 1 entity appears 2 or more times in a relationship, or have a relationship with itself
- Degree - number of edges



Relationship Keys

Relationship keys is the union of the keys from all the entities and itself. Attributes of relationships are the keys of the entities in the relationship and its own relationship attributes.



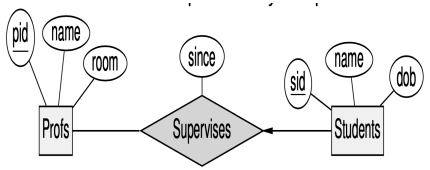
- Each instance of Enrolls has attributes {*sid*, *cid*, *year*, *semester*, *grade*}
- $A' = \{\text{year, semester}\}$, $E' = \{\text{Students, Courses}\}$
- Key(Enrolls) = {*sid*, *cid*, *year*, *semester*}
- Each (sid,cid,year,semester) appears at most once in `Enrolls` relationship set

Relationship Constraint

- Key constraint on E wrt R - E can participate in ≤ 1 instance of R
- Total participation on E wrt R - E can participate in ≥ 1 instance of R

Many-to-One

Each <entity1> can <relationship> **at most 1** <entity2> - key constraint on entity1, ie arrow from entity1 to relationship. **One-to-Many** - just the opposite. Relationship key - set of keys from the limiting entity.



- Key(Supervises) = {sid}

One-to-One

Instead of a one sided **Many-to-One** rs, its two-sided now. Relationship - set of keys from either entity.

Participation Constraint

Whether the entities need to participate or not.

- Partial - each <entity1> <relationship> in ≥ 0 <entity2>, default undirected edge
- Total - each <entity1> <relationship> in ≥ 1 <entity2>

Combining key and total lines will give the overlap which is **exactly 1**.

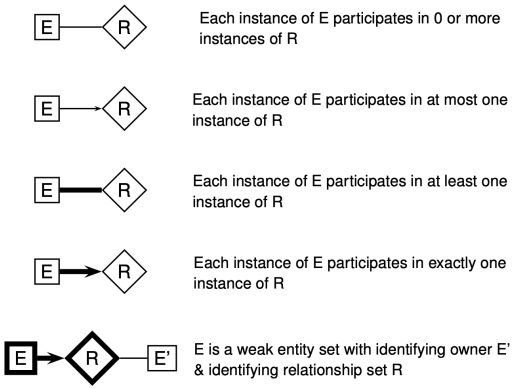
Weak Entity

- Entities that don't have their own keys
- Uniquely identified by considering primary key of another entity - **owner entity**
- Depends on existence on owner entity; if owner deleted, all weak entities that depend on owner deleted.

Conditions :

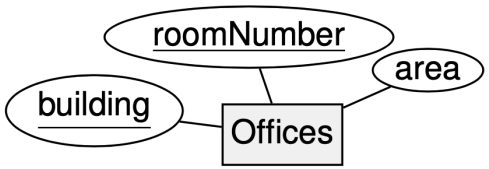
- Must be a many-to-one rs from weak to owner
- Weak entity must have total partiicaption in identifying relationship

Summary of Relationship Constraints



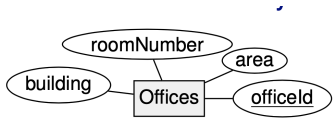
Implementation

Entity Sets




```
create table Offices (
        building        char(10),
        roomNumber      char(7),
        area            integer,
        primary key (building, roomNumber)
);
```

Entity Sets w Candidate Keys Since sql only allows 1 primary key, declare other keys as **unique not null**



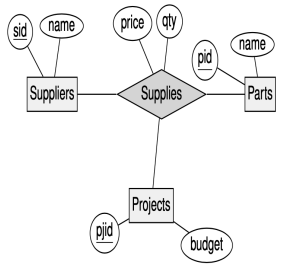
Assume that (building, roomNumber) is a candidate key of Offices

```
create table Offices (
        officeId        char(10) primary key,
        building        char(10) not null,
        roomNumber      char(7) not null,
        area            integer,
        unique (building, roomNumber)
);
```

Relationship Sets

w/o Constraints

```
create table Supplies (
    sid     char(10),
    pid     char(10),
    pjid    char(10),
    price   real,
    qty     integer,
    primary key (sid, pid, pjid),
    foreign key  (sid)
        references Suppliers,
    foreign key  (pid)
        references Parts,
    foreign key  (pjid)
        references Projects
);
```



w Key Constraints - can either be implemented as a seperate table like before, or merged with the entity that has the constraint. As a seperate table, more "calls" needed to get data. Compared to combining, more convinient if requesting relationship and entity together, can have records with null values as attributes.

- First approach: Represent `Supervises` using a separate table
  - **Profs** (pid, name, room)
  - **Students** (sid, name, dob)
  - **Supervises** (sid, pid, since)
- Second approach: Combine `Supervises` & `Students` into one table
  - **Profs** (pid, name, room)
  - **Students** (sid, name, dob, pid, since)

```
create table Students (
        sid     char(20),
        name    char(30),
        dob     date,
        pid     char(7),
        since   date,
        primary key (sid),
        foreign key  (pid)  references Profs
);
```

Key & Total Participation Not enforced in database schema when seperate tables are used for relationships and entities.

```
create table Students (
        sid     char(20),
        name    char(30),
        dob     date,
        pid     char(7) not null,
        since   date,
        primary key (sid),
        foreign key  (pid)  references Profs
);
```

Roles

```
create table Mentors (
        seniorSid       char(20),
        juniorSid       char(20),
        primary key (seniorSid, juniorSid),
        foreign key  (seniorSid)
            references Students(sid),
        foreign key  (juniorSid)
            references Students(sid),
        check           (juniorSid <> seniorSid)
);
```

Weak Entity Sets

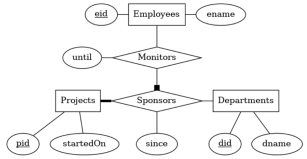
Since we want the deletion for all weak entities to happen when the owner entity is deleted, we use the **on delete cascade** command.

```
create table Chapters (
        number          integer,
        title           char(50),
        isbn            char(30),
        primary key (isbn, number),
        foreign key  (isbn) references Books
                    on delete cascade
);
```

Aggregation

A relationship between entities and relationships - let the relationships be another entity and model the 2 "entities" with a relationship. To get the keys, same logic as with entities and relationship model. Aggregation can be modeled as a ternary relationship. Implementation will be more general and not as accurate as a result.

- Each sponsorship might be monitored by 0 or more employees
  - Each monitoring has a "until" attribute

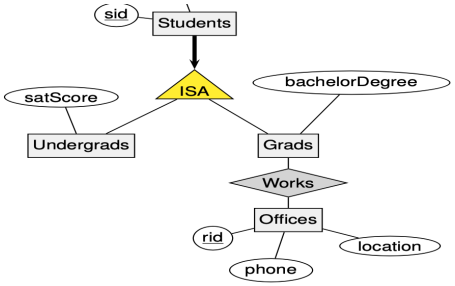


The ■ connected to Sponsors relationship denotes that Sponsors is participating in the Monitors relationship as an aggregation

```
create table Monitors (
        eid     char(20) references Employees,
        pid     char(30),
        did     char(30),
        until   date,
        primary key (eid,pid,did),
        foreign key (pid,did)
                references Sponsors (pid,did)
);
```

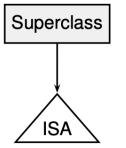
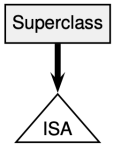
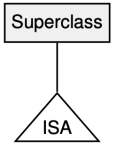
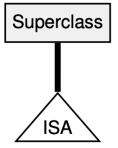
ISA Hierachies

Superclass-subclass hierachy. Key of subclass inherits from superclass.



Constraints :

- Overlap - does superclass belong to multiple subclasses? (satisfied by undirected edge from super to triangle, unsatisfied by directed edge from super to triangle)
- Covering - does every entity in superclass belong to some subclass? (satisfied by thick edge from super to triangle, unsatisfied by thin edge from super to triangle)

Overlap Constraint	Covering Constraint: false	Covering Constraint: true
false		
true		

Implementation - one relation per subclass or superclass. Since the keys of the subclass references the superclass, have to **on delete cascade**.

SQL commands

Comments in SQL are either preceded by 2 hyphens or C-style comments.

Create

Creates a table with the name and specified attributes.

```
create table Students (
        studentId       integer,
        name            varchar(100),
        birthDate       date
);
```

Drop

Completely removes a table from the schema

```
drop table Students;
drop table if exists Students;
drop table if exists Students cascade;
```

The first line removes the table. If it doesn't exist, an error is thrown. Second line removes the table **only if** it exists. Third line does what the second line does **and** removes all the other tables that depend on it.

IS NULL

Returns true if the value is **null**.

```
x IS NULL
```

IS DISTINCT FROM

```
x IS DISTINCT FROM Y
```

- If x and y are non-null, then it evaluates to $x <> y$
- If both are null, then it evaluates to **False**
- If one of the values is null, then it evaluates to **True**

Not-Null Constraints

```
create table Students (
  name varchar(100) not null,
);
```

Constraint is violated if there is a null value in the name field of students.

Unique Constraints

```
create table Students (
  studentId integer unique,
);
```

Listing 1: Column constraint

Constraint is violated if there is are 2 records in Students such that $x.studentId <> y.studentId$ is false. **Null** values are ok because it evaluates to true. Similar to **IS DISTINCT FROM** command.

```
create table Census (
        city varchar(50)
        state char(2),
        unique(city, state)
);
```

Listing 2: Table constraint

Table constraints checks each attribute specified with the **IS DISTINCT FROM** command. Order doesn't matter in this case.

Primary Key Constraints

```
create table Students (
    studentId       integer primary key,
    name            varchar(100),
    birthDate       date
);
```

```
create table Students (
    studentId       integer unique not null,
    name            varchar(100),
    birthDate       date
);
```

Only 1 allowed in sql. Since primary keys are unique and cannot be null, can be used interchangably with **unique not null**. Can specify others as **unique not null**, means the same thing.

Foreign Key Constraints

Column and Table constraints

```
create table Enrolls (
    sid             integer references Students (studentId),
    cid             integer,
    grade           char(2),
    primary key     (sid, cid),
    foreign key     (cid) references Courses (courseId));
```

Multiple attributes of table constraint, order matters.

```
create table Events (
      eid           integer,
      ename         varchar(100),
      date          date,
      level         integer,
      number        integer,
      primary key (eid),
      foreign key   (level, number)
          references Rooms (level, number)
);
```

Ensures that the records with attributes are all non-null or all null

```
create table Events (
      eid           integer,
      ename         varchar(100),
      date          date,
      level         integer,
      number        integer,
      primary key (eid),
      foreign key   (level, number)
          references Rooms (level, number)
          match full
);
```

Referring table **must** be created before getting reference. If column is not specified in the referencing table, primary key will be used. For 3, there could be possible null values since there may be null values in referencing table.

Check Constraint

```
create table Lectures (
    cname       char(5),
    pname       varchar(50) not null,
    day         smallint
                check (day in (1,2,3,4,5)),
    hour        smallint
                check ((hour >= 8) and (hour <= 17)),
    primary key (cname,day,hour),
    unique (pname,day,hour)
);
```

```
create table Lectures (
    cname       char(5),
    pname       varchar(50) not null,
    day         smallint,
    hour        smallint,
    primary key (cname,day,hour),
    unique (pname,day,hour),
    check (
      (day in (1,2,3,4,5) and hour >= 8 and hour <= 17)
      or
      (day = 6 and hour in (9,10,11))
    )
);
```

A boolean expression that restricts the values of an attribute.

Constraint Names

```
create table Lectures (
    cname       char(5),
    pname       varchar(50)
                constraint lectures_pname check (pname is not null),
    day         smallint
                constraint lectures_day check (day in (1,2,3,4,5)),
    hour        smallint
                constraint lectures_hour check (hour >= 8 and hour <= 17),
    constraint lectures_pri_key primary key (cname,day,hour),
    constraint lectures_cand_key unique (pname,day,hour)
);
```

On a violation, the error message would be more meaningful, shows the constraint name that was violated.

Assertions

General constraints, for complex constraints. SQL uses triggers instead.

Insert

```
insert into Students
values      (12345, 'Alice', '1999-12-25', 'Maths');

insert into Students (name, studentId)
values      ('Bob', 67890), ('Carol', 11122);
```

studentId	name	birthDate	dept
12345	Alice	1999-12-25	Maths
67890	Bob	null	null
11122	Carol	null	null

Inserts a record into the table. Inserting into specific columns will result in other columns to be null.

```
        dept            varchar(20) default 'CS'
    );

    insert into Students
    values      (12345, 'Alice', '1999-12-25', 'Maths');

    insert into Students (name, studentId)
    values      ('Bob', 67890), ('Carol', 11122);
```

studentId	name	birthDate	dept
12345	Alice	1999-12-25	Maths
67890	Bob	null	CS
11122	Carol	null	CS

Something like a placeholder value if the attribute is not inserted into.

Delete

```
-- Remove all students
delete from Students;

-- Remove all students from Maths department
delete from Students
where       dept = 'Maths';
```

If we remove all the records in the table, there will be an empty table left behind. The where command is a boolean expression, and records that fulfill that boolean expression will be deleted.

Update

```
-- Add 2% interest to all accounts
update Accounts
set    balance = balance * 1.02;

-- Add $500 to account 12345 & change name to 'Alice'
update   Accounts
set      balance = balance + 500,
         name = 'Alice'
where    accountId = 12345;
```

updates a specific table and sets all values in an attribute to a specific value. Where command specifies the specific record we want to update.

Select

select * from, basically a projection of all the columns in the table.

Foreign Key Violation

Deletion or update could violate the key constraints - update may change to a value that it is outside of the set or deleting an entry from the referenced relation. Specify action to deal with violation **ON DELETE/UPDATE**.

- NO ACTION: rejects delete/update if it violates constraint (default option)
- RESTRICT: similar to *NO ACTION* except that constraint checking can't be deferred
- CASCADE: propagates delete/update to referencing tuples
- SET DEFAULT: updates foreign keys of referencing tuples to some default value
- SET NULL: updates foreign keys of referencing tuples to *null* value

default value may violate key constraint - hence the default value has to exist in foreign key. Cascading deletes the records that violates the constraint.

Transactions

```
begin;
update Accounts
set balance = balance + 1000
where accountId = 2;

update Accounts
set balance = balance - 1000
where accountId = 1;
commit;
```

Either **commit** or **rollback** at the end.

- Commit - makes immediate changes to the database
- Rollback - ignores the actions

Difference between transactions and statements - Transactions satisfies **ACID** properties while a statement does not.

Deferrable Constraints

Usually constraints are checked immediately at the end of execution. Now, the checking is deferred. Deferrable constrainst include - unique attributes, primary and foreign keys.

```
create table  Employees (
      eid           integer primary key,
      ename         varchar(100),
      managerId     integer,
      foreign key   (managerId) references Employees
                    deferrable initially deferred
);

insert into Employees values (1, 'Alice', null), (2, 'Bob', 1), (3, 'Carol', 2);

begin;
delete from Employees where eid = 2;
update Employees set managerId = 1 where eid = 3;
commit;
```

Changes the previous action (deferred or immediately) to deferrable. Additionally, checking of **deferrable** constraints can be changed using **set constraints**

```
      managerId     integer,
      constraint employees_fkey  foreign key (managerId)  references Employees
                    deferrable initially immediate
);

insert into Employees values (1, 'Alice', null), (2, 'Bob', 1), (3, 'Carol', 2);

begin;
set constraints employees_fkey deferred;
delete from Employees where eid = 2;
update Employees set managerId = 1 where eid = 3;
commit;
```

deferrable initially deferred - all rows checked at end of transaction. **deferrable initially immediate** - all rows check at the end of **insert/update**. Different from end of transaction since inserts and updates can occur anywhere in transaction.

Queries

```
select    [ distinct ] select-list
from      from-list
[ where   qualification ]
```

- Select - select cols, same as projection operator
- From - specify which tables to take from
- Where - boolean condition

With no **distinct**, there could be duplicates. Can also use * to refer to all. **select** * **from** ... refers to selecting all the columns in some relation.

Select Clause

- **as** - outputs a table with the aliases used here
- || - string concatenation operator
- **round()** - rounds to nearest integer

Set Operations

Q_1 **union** $Q_2 = Q_1 \cup Q_2$

Q_1 **intersect** $Q_2 = Q_1 \cap Q_2$

Q_1 **except** $Q_2 = Q_1 - Q_2$

intersect has highest precedence over the other 2. Other 2 have same precedence. All set operators removes duplicates. Operands are the queries. To preserve duplicates, use **all** after the set operator.

Multi Relation Queries

- **cross join** - cross product
- **as** - used in **from** clause, alias is used for the where clause. Query local.
- **inner join ... on** - joins the 2 operands and then obtains the records that satisfy the condition after **on**.
- **natural join** - same as the query above but don't have to specify the condition.
- **... outer join... on** - specify the type of **outer join** to be performed on 2 operands and then select records that satisfy the condition after **on**
- **... natural left join ...** - as name suggests.

2 instances of the same table has to be distinguished with different aliases.

```
select   cname, rname
from     Customers C inner join Restaurants R
         on C.area = R.area;
```

Subquery Expressions

Subquery format is similar to the normal query. Subqueries can use the alias from the main query. Putting a constant in the **select** clause indicates no interest in contents of table.

- **EXISTS(subquery)** - true if the output is of subquery is non-empty. Can be negated by adding **NOT** infront.
- **IN(subquery)** - subquery must return 1 column, and expression returns false if query is empty. Else, checks if the out of the expression is in the subquery.
- **ANY(subquery)** - subquery return 1 column, returns false if empty. Else applies an operator on the output with every value of subquery. More general form of **IN**
- **ALL(subquery)** - similar to **ANY** but uses the **and** operator between pairs

```
price > any (
select   price
from     Sells
where    rname = 'Corleone Corner'
);
```

If one of the values for any of the expression is **null**, resulting output would be **unknown**. Put a check constraint to remove possibility of **null**.

```
select   pname, day, hour
from     Lectures L
where    row(day,hour) <= all (
         select day, hour
         from   Lectures L2
         where L2.pname = L.pname
         );
```

Possible to return **IN/ANY/ALL** subqueries with more than 1 column using **rowConstructor**

Scalar Subquery

- Returns a single value if present else null.
- Can be used in **WHERE, FROM, HAVING** clauses
- In **FROM** clause, must have parentheses and table alias

```
select   distinct pizza
from     Sells natural join
         (
            select   rname
            from     Restaurants
            where    area = 'East'
         ) as East_restaurants (rname);
```

ORDER BY

By default is ascending, **asc, dsc**. Evaluates left to right. Usually written after **where** clause.

LIMIT

Limits maximum number of records to be retrieved, counting from top. On a tie, result is non-deterministic. Usually used with **ORDER BY**, otherwise its just random records.

OFFSET

Skip top x number of records in table where x is specified by the user.

Aggregate Functions

Query	Meaning
select min(A) **from** R	Minimum non-null value in A
select max(A) **from** R	Maximum non-null value in A
select avg(A) **from** R	Average of non-null values in A
select sum(A) **from** R	Sum of non-null values in A
select count(A) **from** R	Count number of non-null values in A
select count(*) **from** R	Count number of rows in R
select avg(distinct A) **from** R	Average of distinct non-null values in A
select sum(distinct A) **from** R	Sum of distinct non-null values in A
select count(distinct A) **from** R	Count number of distinct non-null values in A

Usage : Cannot be used in where, even though its tempting Can be used as a scalar subquery since

aggregate functions return a 1x1 result. If want to use in where clause, need to wrap in scalar query.

Group By

Syntax - group by <attribute> **Steps** :

- Partition relation based on attribute
- Compute aggegate function for each group
- Output 1 tuple for each group (can be used with ascending or descending)

Properties

- Attributes specified must contain same values to be in the same group.
- Invalid query if selecting attributes that are not unique, ie the attributes that are not in the group by clause.

For each column A in relation R that appears in **SELECT**, one of the following must hold

- A appears in **GROUP BY**
- A appears in aggregate expression in **SELECT**
- **Primary Key** of R appears in **GROUP BY**

If no **GROUP BY** clause but aggregate function present, table taken as 1 group. Equivalent to saying: **SELECT * FROM** table **GROUP BY** primary key of table.

Dealing with Duplicates

Distinct, Order by - ordering needs to be shared by all attributes of the same name specified by distinct. Ordering by expression needs to appear in the select if the values are different.

Having

Eliminates a groups created by **GROUP BY** that does not fulfill a criteria. Criteria can be scalar or non-scalar queries. For each column A in relation R that appears in **HAVING**, one of the following must hold

- A appears in **GROUP BY**
- A appears in aggregate expression in **HAVING**
- **Primary Key** of R appears in **GROUP BY**

Conceptual Evaluation of Queries

- **from-list** compute cross product of tables
- **where-condition** select tuples that evaluates to true from the list
- **groupby-list** partition selected tuples into groups
- **having-condition** select groups that satisfy the condition
- **select-list** for each group select particular columns
- **distinct** removes Duplicates
- **orderby-list** sorts the results in a particular order
- **offset-specification, limit-specification** specify the tuples wanted based on some size restriction.

Common Table Expressions

Breaking up complex query by creating temporary tables for the query which can then be referenced in the actual query. Something like a helper table.

```
with
    R1 as (Q1),
    R2 as (Q2),
    ...,
    Rn as (Qn)
select/insert/update/delete statement S;
```

Views

Creating a customised view of the logical schema for audience. Creates logical data independence. Apparently users can just reference it and the view wil be evaluated at run time or sth.

Conditional Expressions

Case

```
case
  when condition1 then result1
  ...
  when conditionn then resultn
  else result0
end
```

```
case expression
  when value1 then result1
  ...
  when valuen then resultn
  else result0
end
```

If the condition is true, then it will do result. Similar to switch statements

Coalesce

1 statement if-else. Goes left to right until a non-null values is reached. Otherwise, returns null. Use COALESCE(VALUE, 0) to always return 0 if VALUE is null

NULLIF

nullif ($value_1$, $value_2$)

Returns *null* if $value_1$ is equal to $value_2$; otherwise returns $value_1$

Pattern Matching for Strings

```
like '_ _ _%e';
```

- Underscore matches any single character
- Percent matches any sequence of 0 or more characters

Universal Quantification

All word in query, usually easier to negate the query, write it out, then negate the entire thing.

Statement Level Interface

- Write program that mixes SQL and host language (C, python, etc) - .pgc file
- Preprocess file - .c
- Compile file to get executable - .exe

Static SQL

EXEC indicates SQL query as a section

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;                          Declaration
      char name[30]; int mark;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT @localhost USER john;                   Connection
    // some code that assigns values to                      Host language
    // name and mark.
    EXEC SQL INSERT INTO                                     Query execution
      Scores (Name, Mark) VALUES (:name, :mark);
    EXEC SQL DISCONNECT;                                     Disconnect
}
```

Dynamic SQL

Generate queries during runtime. Store the query in a string instead. Query needs to be assigned to SQL statements. **:** - access to query.

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;                          Declaration
      char *query; char name[30]; int mark;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT @localhost USER john;                   Connection
    // some code that assigns values to                      Host language
    // name and mark

    // assign any SQL statement to the query,
    // the query may include name and/or mark.
    EXEC SQL EXECUTE IMMEDIATE :query;                       Query execution
    EXEC SQL DISCONNECT;                                     Disconnect
}
```

Call Level Interface

- Write in host language while making reference to APIs to access DB
- Compile into executable

```
void main() {
    char *query; char name[30]; int mark;                    Declaration
    connection C("dbname = testdb user = postgres \          Connection
      password = test hostaddr = 127.0.0.1 \
      port = 5432");
    // assign any SQL statement to the query,                Query execution
    // the query may include name and/or mark.

    work W(C);
    W.exec(query);
    W.commit();
    C.disconnect();                                          Disconnect
}
```

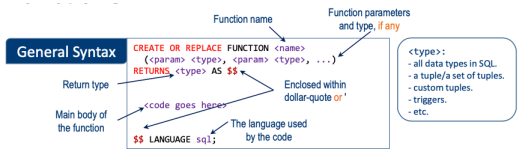
Above figure is dynamic SQL because of assignment.

Statement-level	Call-level
Code written in mix of host and SQL	Code written in host
Code preprocessed before compiling	Code directly compiled to exe

PL/pgSQL

- Server-side programming
- standardise syntax and semantics of SQL
- Provides : Code reuse, ease of maintenance, performance, security

Functions



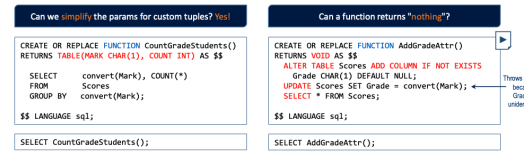
REPLACE - if function already exists and want to overwrite. To call function, similar to calling aggregate function. **This is not an aggregate function!!**

```
CREATE OR REPLACE FUNCTION GradeStudent
  (Grade CHAR(1))
RETURNS Scores AS $$

  SELECT    *
  FROM      Scores
  WHERE     convert(Mark) = Grade;

$$ LANGUAGE sql;
```

To return a **tuple** - change return type to the resulting table of the query in the code. To return **multiple tuples**, use **SETOF** keyword. Custom tuples can be made by specifying the output types.

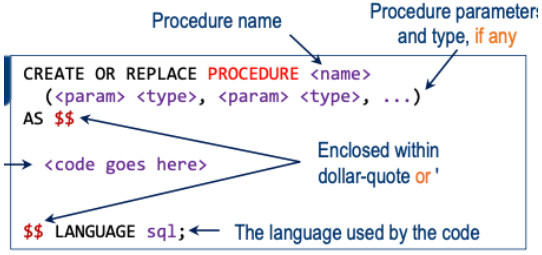
```
CREATE OR REPLACE FUNCTION CountGradeStudents
  (IN Grade CHAR(1),
   OUT Grade CHAR(1),
   OUT Count INT)
RETURNS RECORD AS $$

  SELECT      Grade, COUNT(*)
  FROM        Scores
  WHERE       convert(Mark) = Grade;


$$ LANGUAGE sql;
```

- **SELECT func** - tuple output
- **SELECT * FROM func** - table output
- **SELECT func FROM table** - apply func for all records in table



Grade is unidentified because of update. Executed as an entire transaction. Move update outside as a separate statement.

Procedures




```
CREATE OR REPLACE PROCEDURE AddGradeAttr()
AS $$

  ALTER TABLE Scores ADD COLUMN IF NOT EXISTS
    Grade CHAR(1) DEFAULT NULL;
  SELECT * FROM Scores;

$$ LANGUAGE sql;
```

```
CALL AddGradeAttr();
```

Properties

- No return type
- **PROCEDURE** keyword
- Everything else the same as function
- **CALL** procedure.

Functions	Procedures
Returns value(s)	No return value
FUNCTION keyword	PROCEUDRE keyword
To call, follow normal SQL clauses	To call, use CALL procedure

Variables

```
CREATE OR REPLACE FUNCTION splitMarks
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20),
  OUT mark1 INT, OUT mark2 INT)
RETURNS RECORD AS $$
DECLARE
  temp INT := 0;
BEGIN
  SELECT mark INTO mark1 FROM Scores
  WHERE name = name1;
  SELECT mark INTO mark2 FROM Scores
  WHERE name = name2;

  temp := (mark1 + mark2) / 2;

  UPDATE Scores SET mark = temp
  WHERE name = name1 OR name = name2;
  RETURN; --optional
END;
$$ LANGUAGE plpgsql;
```

Note the assigning of variables : **name TYPE := value**.

Keyword	Definition
DECLARE	declare variables here using name TYPE := value
BEGIN, END	write code here
RETURN	optional
RETURN QUERY ...	appends result to resulting table. **DOES NOT EXIT FUNCTION**
RETURN NEXT	appends tuple to resulting table. **DOES NOT EXIT FUNCTION**

Control Structures

```
IF temp > 60 THEN      temp := temp / 2;
ELSIF temp > 50 THEN   temp := temp - 20;
ELSE                   temp := temp - 10;
END IF;
```

```
WHILE temp > 30 LOOP
    temp := temp / 2;
END LOOP;
```

```
LOOP
    EXIT WHEN temp < 30;
    temp := temp / 2;
END LOOP;
```

in Imperative Language ...

```
while (true) {
  if (temp < 30)
    break;
}
```

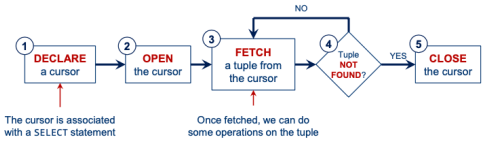
```
FOREACH d IN ARRAY denoms LOOP
    temp := temp / d;
END LOOP;
```

Declaration

```
d INT;
denoms INT[] :=
ARRAY[1, 2, 3];

*index starts at 1.
```

Cursor

Traverses row returned by a SELECT statement



Keyword	Definition
DECLARE	sets var name for cursor
OPEN	opens the cursor
FETCH	fetches cursor into variable, checks if end of table
FETCH RELATIVE	moves cursor to position relative to current and fetch the record
MOVE RELATIVE	moves cursor to position relative to current
CLOSE	closes the cursor

SQL Injection

String queries can be manipulated.

```
char email[100];
scanf("%s", email);
char password[100];
scanf("%s", password);

//query = "SELECT COUNT(*) FROM Users" +
//      "WHERE email = '" + name + "'" +
//      "AND password = '" + password + "';";
```

If user inputs in a password with ', password is almost guaranteed to go through because it closes the string, and condition will always be true. Use a function or procedure to protect the DB. Functions and procedures are compiled. At run time, they change entire input into a string. PREPARE statements to prepare the statement to be executed.

Triggers

Perform an action when a entries in a table is modified. Trigger - checks whether the table modified fulfills some condition. Trigger function - perform some action if triggered.

■ The trigger:

```
CREATE TRIGGER scores_log_trigger
AFTER INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log_func();
```

■ The trigger function:

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS TRIGGER
AS $$
BEGIN
        INSERT INTO Scores_Log(Name, EntryDate)
                VALUES (NEW.Name, CURRENT_DATE);
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

The Trigger

From the above example, this is what the trigger does :

- watch out for some modification on the target table
- depending on the modification, perform some row-wise operation

■ The trigger function:

```
CREATE OR REPLACE FUNCTION scores_log2_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN ...
    ELSEIF (TG_OP = 'DELETE') THEN ...
    ELSEIF (TG_OP = 'UPDATE') THEN ...
    END IF;
    ...
```

■ The trigger:

```
CREATE TRIGGER scores_log2_trigger
AFTER INSERT OR DELETE OR UPDATE ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log2_func();
```

Return type of **TRIGGER** indicates trigger function.**NEW** refers to the last row that inserted. **Only trigger functions can access the NEW keyword**. **OLD** for the last row that was inserted before this current one. Think of it like a linked list, with **NEW** being the current node, and **OLD** being the previous one. Possible to specify the type of operation to perform on in trigger function using **TG_OP**.

Trigger Timing

Keyword	Definition
AFTER	function executed after operation on table
BEFORE	function executed before operation on table
INSTEAD OF	function executed instead of operation on table

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Effect: Elise's mark would be 100 regardless of what we insert

Returning **NULL** tells the db to ignore everything after. ie, exit the query/transaction. Returning **OLD**, is the same as returning **NULL**. Unless the values of **OLD** are updated in the trigger function, then the result will be non-null. **For AFTER operation, return value doesn't matter** - trigger function invoked after operation done.

INSTEAD OF Triggers

Defined in views. Similar properties to returning null or non-null values as above.

Trigger Levels

Row level - apply function on every row that was involved in operation. Statement level - execute once regardless of number of rows involved in op. **INSTEAD OF** - only for row level. Others are allowed for both.

Statement Level

Ignores return value. To ignore subsequent operations, raise **EXECEPTION** instead.

Trigger Condition

Moving condition outside of trigger function using **WHEN**. Only when condition satisfied, then activate trigger. Some conditions of trigger conditions :

- No SELECT in WHEN
- No OLD in WHEN for INSERT
- No NEW in WHEN for DELETE
- No WHEN for INSTEAD OF

Deferred Triggers

Defer the checking of a trigger because some operations require 2 or more separate queries, and in the process, failing trigger requirement. Only works for AFTER and FOR EACH ROW

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION bal_check_func();
```

- CONSTRAINT and DEFERRABLE together indicate that the trigger can be deferred
- INITIALLY DEFERRED indicates that by default, the trigger is deferred
  - Other option: INITIALLY IMMEDIATE, i.e., the trigger is not deferred by default

Trigger activates when committing. If its initially immediate, create constraint to set it to be deferred.

```
BEGIN TRANSACTION;
SET CONSTRAINTS bal_check_trigger DEFERRED;
UPDATE Account SET Bal = Bal – 100 WHERE AID = 1;
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

Multiple Triggers

Order of activation :

- BEFORE statement-level
- BEFORE row-level
- AFTER row-level
- AFTER statement-level

Within each level, triggers separated in lexicographical order. If BEFORE row-level returns null, subsequent triggers wont happen.

Functional Dependencies

How to decide the goodness of a schema? Which schema fulfills some criteria

Redundancy

Storing repeated info in a schema.

Update Anamolies

In the case of repeated information in a schema, we may update one value for a tuple and forget to update the value for other tuples.

Deletion Anamolies

Primary key deletion not allowed.

Insertion Anamolies

Incomplete information can violate primary key constraint

Normalisation

Break up the schema into smaller tables. Resolves the redundancies and anamolies. (Why not just impose constraints on the table? - talking about schema layout here, measure of how good the schema is without any values.)

Functional Dependencies

- cause of redundancies
- find a counter example to prove this

If $\mathbf{X} \rightarrow \mathbf{Y}$, then if 2 tuples have same X value, they have same Y value. FDs can hold on one table but not on other tables (ie specific to requirements of schema)

FD Reasoning

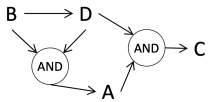
Suppose $\mathbf{X} \rightarrow \mathbf{Y}$, $\mathbf{Y} \rightarrow \mathbf{Z}$

- Reflexivity - $XY \rightarrow X$
- Augmentation - $XZ \rightarrow YZ$, for any values of Z (ie if $X \rightarrow Y$, then $X \rightarrow XY$)
- Transitive - from above, $X \rightarrow Z$
- Decomposition - if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Union - if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- if $X \rightarrow Y$, then by augmentation $X \rightarrow XY$

Closure

Something like circuit board, multiple attributes combined using AND gate. fds are connected via arrows.

- Four attributes: A, B, C, D
- Given: B→D, DB→A, AD→C
- Check if B→C holds



- First, activate B
  - Activated set = { B }
- Second, activate whatever B can activate
  - Activated set = { B, D }, since B→D
- Third, use all activated elements to activate more
  - Activated set = { B, D, A }, since DB→A
- Repeat the third step, until no more activation is possible
  - Activated set = { B, D, A, C }, since AD→C; done
- { B, D, A, C } is referred to as the closure of {B}

To find closure of element in the brackets :

- draw the board connecting the attributes and the necessary AND gates
- for every element in the closure
- see how many elements it can turn on
- all the elements that are on in the end will be in the closure.

1. Initialize the closure to $\{A_1, A_2, \ldots, A_n\}$
2. If there is an FD: $A_i, A_j, \ldots, A_m \rightarrow B$, such that $A_i, A_j, \ldots, A_m$ are all in the closure, then put B into the closure
3. Repeat step 2, until we cannot find any new attribute to put into the closure

- Example
  - A Table with five attributes A, B, C, D, E
  - A → B, C → D, BC → E
  - $\{A\}^+ = \{A, B\}$
  - $\{A, C\}^+ = \{A, B, C, D, E\}$
  - $\{B\}^+ = \{B\}$

To proof whether an fd holds, show whether its inside the closure.

Keys, Superkeys

Similar idea to entity sets, but in this case, it also decides the values of all other attributes in the table. Check FDs, then use closure to derive the keys. How to find keys (naive)?

- Consider every subset of attributes in a table
- Derive closure for each subset
- Identify Superkeys
- identify keys

How to do better?

- Checks small attribute sets first. Others will be superkeys and not keys.
- In the FDs presented, if an attribute doesnt appear on the RHS of any of them, then it can only decide itself. Which means that its guaranteed to be a key

Prime Attributes

Attributes that appears in a key. If XY is a key, then X and Y are prime attributes.

Non-trivial and Decomposed FD

- Decomposed : RHS has **only 1 attribute**. Non-decomposed FD can be translated into decomposed FD using some form of decomposition.
- Non-trivial : Attribute on RHS don't appear in left.

Algorithm to derive :

- Consider **all** attribute subsets
- Compute closure of each subset
- From each closure, remove trivial attributes (ie attributes that appear on both RHS and LHS) from RHS
- Derive non-trivial and decomposed FD from each closure

BCNF

If every non-trivial and decomposed FD has a superkey as its LHS. Basically LHS must always be superkeys. If LHS is not superkey, then LHS appear multiple times in table, then same RHS appear multiple times (redundancy).

Name	NRIC	Phone	Address
Alice	1234	67899876	Jurong East
Alice	1234	83848384	Jurong East
Bob	5678	98765432	Pasir Ris

- Key: {NRIC, Phone}
- We have NRIC → Name, which violates BCNF
- Since NRIC is not a superkey, the same NRIC can appear multiple times in the table
- Every time the same NRIC is repeated, the corresponding Name and Address are also be repeated
- This leads to redundancy
- BCNF prevents this

Algorithm to check whether table in BCNF :

- Derive non-trivial and decomposed fd using algorithm above
- Check to see if **all** satisfy requirement of BCNF

Combine derivation of keys, non-trivial and decomposed FDs and requirement checking. Check closures :

- Closure contains more attributes than LHS
- Since LHS not superkey, won't contain **all** attributes in table.

More but not all condition - violation of BCNF if theres an fd that satisfies this.

Normalisation

If a table is not in BCNF, recursively decompose them into smaller tables. Once a table is in BCNF, we can stop. Criteria for decomposition is choosing a common attribute, X , that violates BCNF, and split them

- If table is not in BCNF, decompose into smaller tables
- If table is in BCNF, stop
- Choose an attribute, X, from table that violates
- Split table into 2 where R_1 has all the attributes in the closure of X, and R_2 has attributes not in closure of X and X.

Normalisation result may not be unique. If table has 2 attributes, confirm satisfies BCNF. If the child table has no clear fds, derive the fds from the parent and project onto child. Basically, if we derive all fds from the root, we can decompose freely. Cross out the attributes that are not in the current child table. **Why does this work?** - since there are finite number of violations and every decomposition removes at least 1 violation, the number of violations will eventually reach 0. **Lossless Join Decomposition** - because of common attributes in the decomposed child tables, we can join them back based on the common attributes.

Properties

- No update or deletion or insertion anomalies (+)
- Small redundancy (+)
- Orginal table can be reconstructed from the decomposed tables (+)
- Dependencies may not be preserved (-)

Third Normal Form

For every non-trivial and decomposed FD

- Either LHS is superkey
- Or RHS is a prime attribute (appear in key)

If BCNF is satisfied,then 3NF is satisfied, but not necessarily vice versa. (3NF can be satisfied by the second condition, which is not never satisfied in BCNF). If 3NF is violated, then BCNF is violated, but not necessarily vice versa. (Violation of BCNF can mean that the second condition of 3NF is fulfilled).

3NF Check

- Compute closure for each subset of attributes
- Derive keys
- For each closure check if closure satisfies "more but not all" **and** there's attribute in RHS thats not in LHS and its not a prime attribute
- If no closure exists, then in 3NF

3NF Decomposition

Only 1 split to all relevant children that satisfy 3NF. Algo :

- Derive **minimial basis** from given FDs
- In minimal basis, combine FDs whose LHS are the same
- Create table for each FD that remains
- If none of the table contain a key of original table, create a table that contains a key (**any key**).

Minimal Basis

Criteria :

- Every FD in basis can be derived from original and vice versa
- Every FD in basis is non-trivial and decomposed
- No redundant FDs in minimal basis
- For each FD in basis, no redundant attributes on **LHS** - **Ensures lossless join**

Algorithm :

- Create decomposed FDs
- Remove redundant attributes on LHS (compare with decomposed FDs)
- Remove redundant FDs (wrt other FDs, once removed, not considered in calculation of others)

Steps 2 and 3 cannot be swapped. Order matters. To remove redundant attributes, determine if the FD (less 1 attribute can be derived) has a redundant attribute. If it does, take the simplified one. To check for redundant FDs, see if by removing it, whether the closure of the LHS contains its RHS. If it does, then its redundant.

BCNF or 3NF?

Go for BCNF if there is a decomposition that preserves all FDs. If no such decomposition, then depending on needs

- BCNF if preserving FDs not important
- 3NF otherwise