Lecture 4: Defining database Schema using SQL DDL

CSX3006 DATABASE SYSTEMS

ITX3006 DATABASE MANAGEMENT SYSTEMS

Outline

- **How to define Database Schema using SQL DDL**
  - **Domain (Data type) Specification, Integrity Constraints Specification**
- How to express Queries using SQL DML
  - Basic Structures of SQL Query expression

Relational Algebra vs. SQL

RELATIONAL ALGEBRA

- A concise, formal notation for representing queries for relational database
- DML expressions for data retrieval and manipulation only
- **No formal syntax for specifying schema or constraints**

SQL

(STRUCTURED QUERY LANGUAGE)

- **More "user-friendly" and pragmatic** than relational algebra
- **Based on relational algebra and relational calculus**
- **Support both the definition and manipulation of database**
  - **DDL (Data Definition Language)** – for definition of DB schema
  - **DML (Data Manipulation Language)** – for data retrieval and manipulations

SQL DDL for Defining Database Schema

- Allows the **specification of relations** (tables) and **other control** information
  - **The schema for relations**
    - **Domain of attributes**
    - **Data Integrity constraints**
      - Primary Key, Candidate Key, Foreign Key, etc
  - **Indices** for each relation
  - **Security and authorization** information for each relation
  - **Physical storage structure** of each relation on disk

Issues

- How to create tables and views
  - How to specify the **domains** of attributes; the data types supported
  - How to specify various data consistency and integrity **constraints**
- How to remove tables and views
- How to modify the schema of the tables

Create a Table

Syntax:

create table ***r*** (A_1 D_1,
A_2 D_2,
...,
A_n D_n,
(integrity-constraint$_1$),
...,
(integrity-constraint$_k$))

- ***r*** is the **relation name**
- each A_i is an **attribute name** in the schema of relation ***r***
- D_i is the **data type** of values for the **domain of attribute** A_i

Some Basic Data Types

- Numeric data types[1]
  - Integer numbers: `INTEGER`, `INT`, and `SMALLINT`
  - Floating-point (real) numbers: `FLOAT` or `REAL`, and `DOUBLE PRECISION`
  - Formatted numbers: `DECIMAL(i,j)`, `DEC(i,j)` or `NUMERIC(i,j)`
    - `i`: total number of decimal digits
    - `j`: the number of digits after the decimal point
    - E.g., the maximum allowed value for decimal(5,2) is 999.99
  - Auto increment number: `SERIAL – in postgresql`
- Character-string data types[2]
  - Fixed length, blank padded: `CHAR(n)`, `CHARACTER(n)`
  - Varying length with limit : `VARCHAR(n)`, `CHARACTER VARYING(n)`,
  - Variable unlimited length: `TEXT`, `varchar`
- Boolean data type
  - Values of `TRUE` or `FALSE` or `NULL`

[1]: http://www.postgresqltutorial.com/postgresql-integer/
[2]: http://www.postgresqltutorial.com/postgresql-char-varchar-text/

Another Data Type: Date and Time

- Date literal: ‘yyyy-mm-dd’
  - E.g., May 20, 2021 is represented in SQL as ‘2021-05-20’
- A commonly used function: current_date
  - Return the current date
- Time literal ‘hh:mm:ss’

Caution

- Each statement in SQL should be ended with a **semicolon.**
- **String literal is case sensitive.**
  - E.g., ‘Smith’ is not equals to ‘smith’

Relations in Banking Enterprise Database

branch	branch_name	branch_city	assets
account	account_number	branch_name	balance
depositor	customer_name	account_number	
customer	customer_name	customer_street	customer_city
loan	loan_number	branch_name	amount
borrower	customer_name	loan_number	

SQL Command to Create Tables - 1

```
create table branch (
   branch_name       char(15),
   branch_city       char(15),
   asset             numeric(18,2)
);

create table customer (
    customer_name  char(15),
    customer_street  char(15),
    customer_city     char(15)
);
```

branch_name	*branch_city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

branch relation

customer_name	*customer_street*	*customer_city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

customer relation

SQL Command to Create Tables - 2

```
create table account (
    account_number char(10),
    branch_name      char(15),
    balance              numeric(18,2)
);
```

account_number	branch_name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

account relation

```
create table depositor (
    customer_name  char(15),
    account_number char(10)
);
```

customer_name	account_number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

depositor relation

SQL Command to Create Tables - 3

```
create table loan (
    loan_number  char(10),
    branch_name char(15),
    amount          numeric(18,2)
);

create table borrower (
   customer_name char(15),
   loan_number     char(10)
);
```

loan_number	branch_name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

customer_name	loan_number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

Problems with the Simple Tables - 1

- Accidentally add duplicate records into the table
  - insert into branch values ('Downtown', 'Brooklyn', 9000000);
  - insert into branch values ('Downtown', 'Brooklyn', 9000000);
  - select * from branch;

	branch character (15)	branch_city character (15)	asset numeric (18,2)
1	Downtown	Brooklyn	9000000.00
2	Downtown	Brooklyn	9000000.00

  - How to solve this problem?

Problems with the Simple Tables - 2

- Accidentally add duplicate records into the table
  - insert into branch values ('Downtown', 'Brooklyn', 9000000);
  - insert into branch values ('Downtown', 'Brooklyn', 9000000);
  - select * from branch;

	branch character (15)	branch_city character (15)	asset numeric (18,2)
1	Downtown	Brooklyn	9000000.00
2	Downtown	Brooklyn	9000000.00

  - How to solve this problem?
  - Use Primary Key Constraint

Problems with the Simple Tables - 3

- Insert Records to Simple Tables Created
  - insert into account values ('A-101', 'Downtownn', 500);

    /*Typo happens on 'Downtown'*/

- Still able to add to the database but may have problem later on ← How to solved?
  - select * from account where branch = 'Downtown';
    - 0 row affected

Problems with the Simple Tables - 4

- insert into branch values ('Downtown', 'Brooklyn', 9000000);
- insert into account values ('A-101', 'Downtown**n**', 500);

/*Typo happens on 'Downtown'*/

- Still able to add to the database but may have problem later on ← How to solved?
  - Use Referential Integrity Constraint

Problems with the Simple Tables - 5

- Balance is a ‘required’ value but forget to put it.
  - insert into account values ('A-102', 'Perryridge');
  - select * from account;

	account_numbe character (10)	branch_name character (15)	balance numeric (18,2)
1	A-102	Perryridge	[null]

- Violates business logic!! (still able to add to the database) ← How to solved?

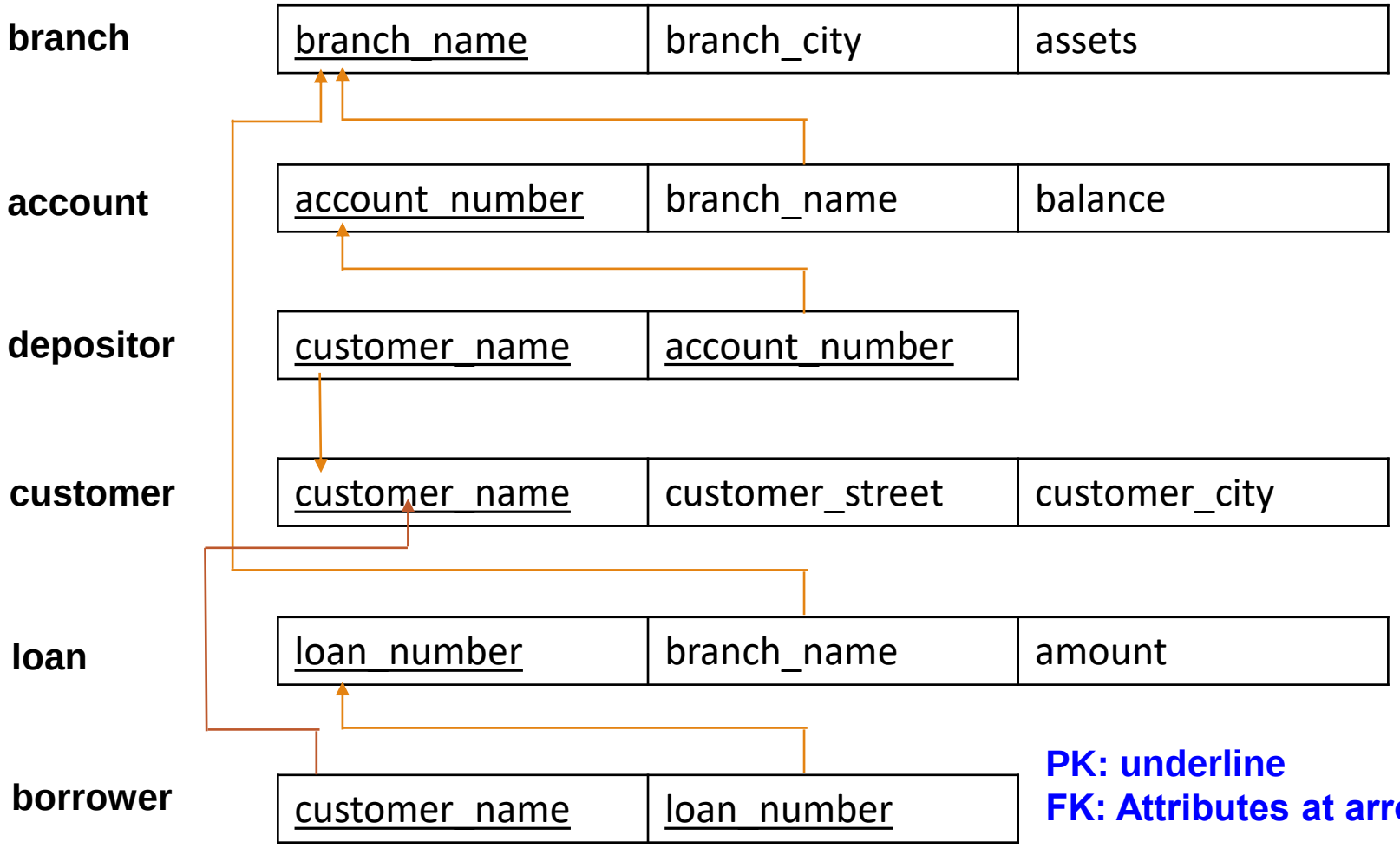
Problems with the Simple Tables - 6

- Balance is a ‘required’ value but forget to put it.
  - insert into account values ('A-102', 'Perryridge');
  - select * from account;

	account_numbe character (10)	branch_name character (15)	balance numeric (18,2)
1	A-102	Perryridge	[null]

- Violates business logic!! (still able to add to the database) ← How to solved?
  - Use Not Null Constraint

Relational Model of Banking Enterprise Database



Revised SQL Command to Create Tables - 1

branch

branch_name	branch_city	assets

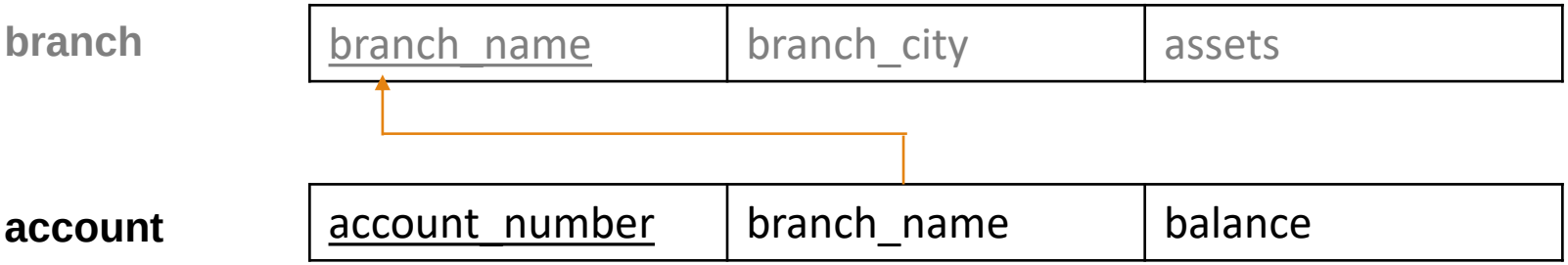
```
create table branch (
  branch_name      char(15),
  branch_city      char(15),
  asset            numeric(18,2)      not null,
  primary key (branch_name)
);
```

customer

customer_name	customer_street	customer_city

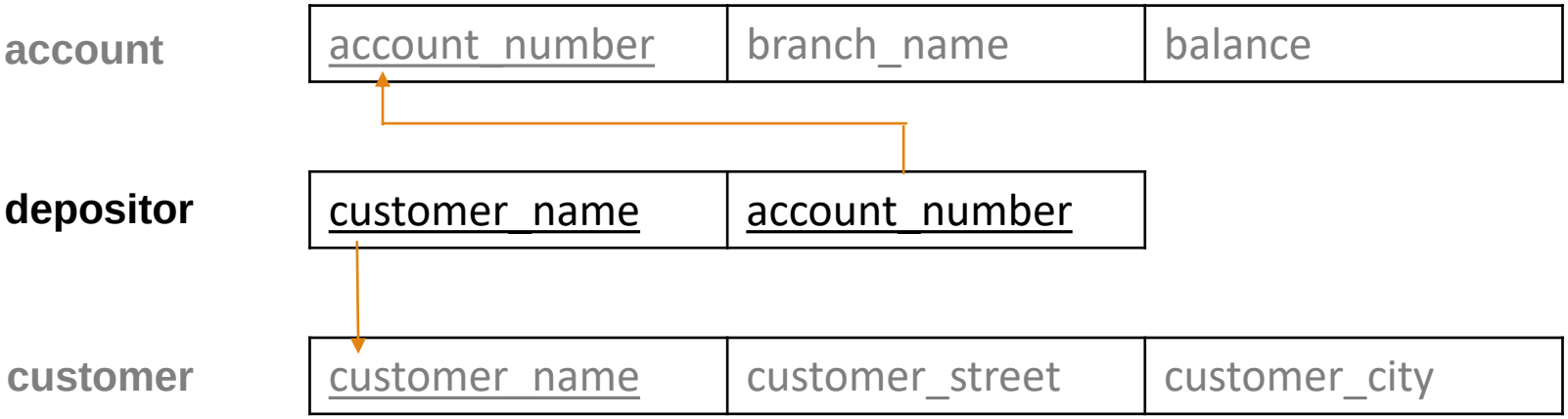
```
create table customer (
   customer_name  char(15),
   customer_street  char(15),
   customer_city              char(15),
   primary key (customer_name)
);
```

Revised SQL Command to Create Tables - 2



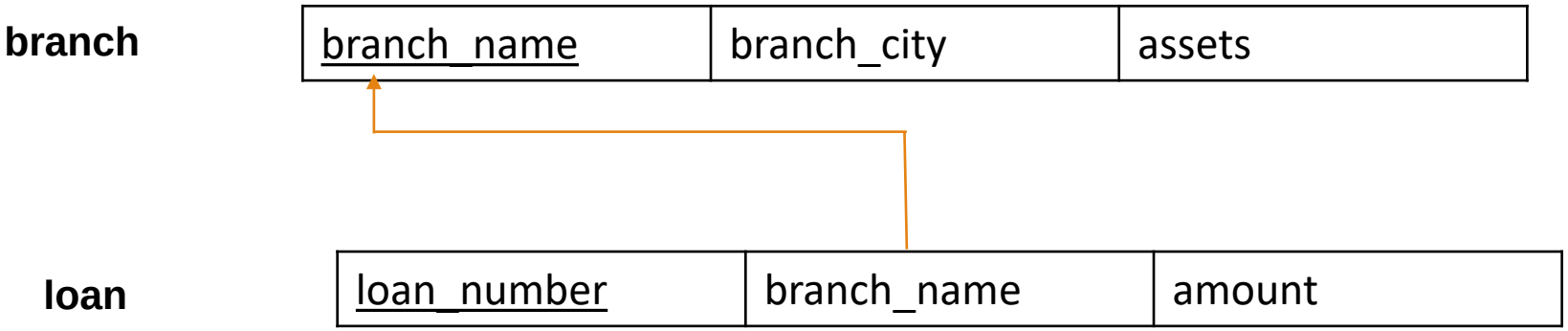

```
create table account (
    account_number  char(10),
    branch_name     char(15),
    balance         numeric(18,2)       not null,
    primary key (account_number),
    foreign key (branch_name) references branch,
    check (balance >= 0)
);
```

Revised SQL Command to Create Tables - 3



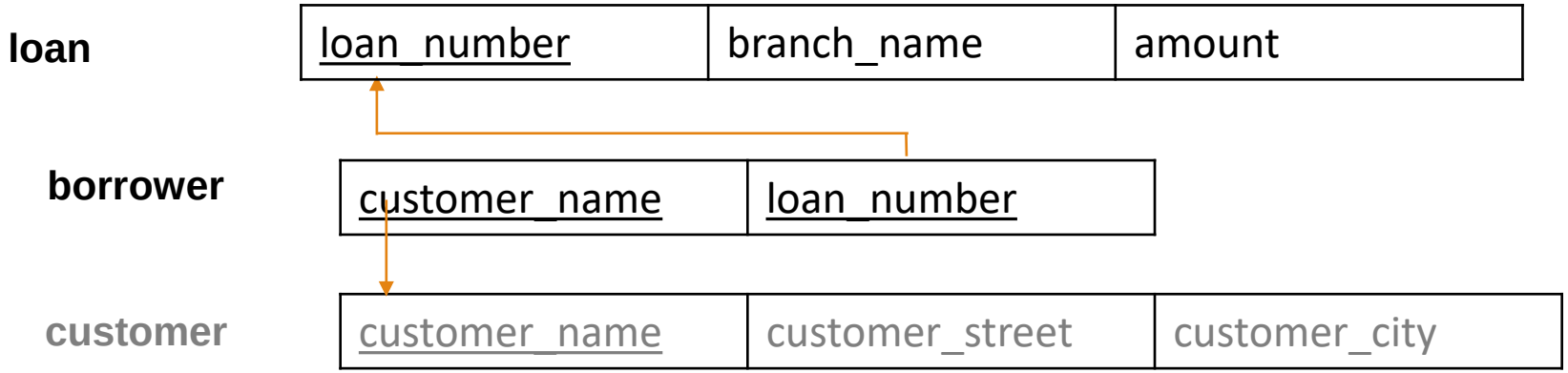

```
create table depositor (
    customer_name  char(15),
    account_number char(10),
    primary key (customer_name, account_number),
    foreign key (customer_name) references customer,
    foreign key (account_number ) references account
);
```

Revised SQL Command to Create Tables - 4




```
create table loan (
   loan_number      char(10),
   branch_name      char(15),
   amount           numeric(18,2)       not null,
   primary key (loan_number ),
   foreign key (branch_name) references branch(branch_name),
   check (amount >= 0)
);
```

Revised SQL Command to Create Tables - 5




```
create table borrower (
    customer_name  char(15),
    loan_number      char(10),
    primary key (customer_name, loan_number ),
    foreign key (customer_name) references customer,
    foreign key (loan_number ) references loan
);
```

After Enforcing Constraints and Trying to Add Records Again - 1

- ‘Jackson’ is not in customer!!
  - insert into borrower values ('Jackson', 'L-14');
    - ERROR: insert or update on table "borrower" violates foreign key constraint "borrower_customer_name_fkey" DETAIL: Key (customer_name)=(Jackson) is not present in table "customer". ********** Error ********** ERROR: insert or update on table "borrower" violates foreign key constraint "borrower_customer_name_fkey" SQL state: 23503 Detail: Key (customer_name)=(Jackson) is not present in table "customer".

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

After Enforcing Constraints and Trying to Add Records Again - 2

loan_number	branch_name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

customer_name	customer_street	customer_city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

customer relation

customer_name	loan_number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

borrower relation

User-Defined Types and Domains - 1

- **create type** construct in SQL creates user-defined type

```
create type status as enum ('single', 'married', 'divorce');

/*use the user-defined type*/
create table customer_status(
   customer_name     char(15)     primary key     not null,
    customer_status    status
);
```

User-Defined Types and Domains - 2

- **create domain** construct in SQL-92 creates user-defined domain types

```
create  domain  person_name     char(20)     not null;


/*use the user-defined domain*/
create table potential_customer(
   customer_name     person_name      primary key,
    customer_street    char(15),
    customer_city       char(15)
);
```

- **Note:** domains can have constraints, such as **not null**, specified on them.

Another Example -- Domain

```
create domain GPA       real      check (value >=0 AND value <= 4.0);

create table student (
        id              integer,
        name            char(15),
        accumGPA        GPA
);

insert into student values (1, 'Sara', 3.00); -- ok
insert into student values (2, 'Mike', 4.02); -- cannot insert
```

```
ERROR:  value for domain gpa violates check constraint "gpa_check"
SQL state: 23514
```

Reminder on Data Types and Domain Specification

- Built-in data types and user-defined types are for specifying domain of attributes at physical level
- Each data type has:
  - Number of bits or bytes used to store the data
    - Affects the set of representable values;
    - Minimum and maximum values
  - Set of valid operations

Additional Reading for More Details

- Postgresql datatype
  - https://www.postgresql.org/docs/9.5/static/datatype.html

Drop a Table - 1

Syntax

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

IF EXISTS: do not throw an error if the table does not exist.

name: the name of the table to drop.

CASCADE: automatically drop objects that depend on the table (such as views or foreign-key constraint).

RESTRICT: refuse to drop the table if any objects depend on it. (default)

- Example 1: **drop table loan;**

ERROR: cannot drop table loan *because other objects depend on it*

DETAIL: constraint borrower_loan_number_fkey on table borrower depends on table loan

HINT: Use DROP ... CASCADE to drop the dependent objects too.

********** Error **********

...

More ref: https://www.postgresql.org/docs/8.2/static/sql-droptable.html

Drop a Table - 2

- Example 2: **drop table loan cascade;**
  - NOTICE: drop cascades to **constraint** borrower_loan_number_fkey *on table borrower*

```
create table borrower (
customer_name  char(15),
loan_number                    char(10),
primary key (customer_name, loan_number ),
foreign key (customer_name) references customer,
foreign key (loan_number ) references loan);
```

This constraint is dropped before removing the table loan.

Alter Table Construct - 1

- **Schema of a relation can be modified when necessary** by using DDL command **alter table**
  - Add/delete/rename column
  - Change data type of column

Note: a particular DBMS MAY NOT support ALL the features and the syntax may vary from product to product

Alter Table Construct - 2

Syntax:

```
alter table table_name
   add column_name data_type

alter table table_name
   drop column column_name

alter table table_name
   rename column old_name to new_name

alter table table_name
   modify column_name data_type
```

Examples:

```
alter table account
     add last_accessed timestamp

alter table account
     drop column branch_name

alter table account
     rename column balance to ac_balance

alter table account
     modify balance number(50,2)
```

Note: a particular DBMS MAY NOT support ALL the features and the syntax may vary from product to product

Specification of Constrictions in SQL DDL

- Example

```
create table account
   ( account_number          char(10),
     branch_name             char(15),
     balance                 numeric(10,2) not null,
     primary key (account_number),
     foreign key (branch_name) references branch,
     check (balance >= 0) );
```

Constraints Specification

- Ensure that changes made to the database by authorized users do not result in a loss of data consistency
- Are based upon the semantics of the real-world enterprise being modelled in the database application
- Specify conditions that must be true for any instance of the database
  - Specified *when schema are defined* by SQL DDL
  - Checked *when relations are modified* by SQL DML

Kinds of Constraints

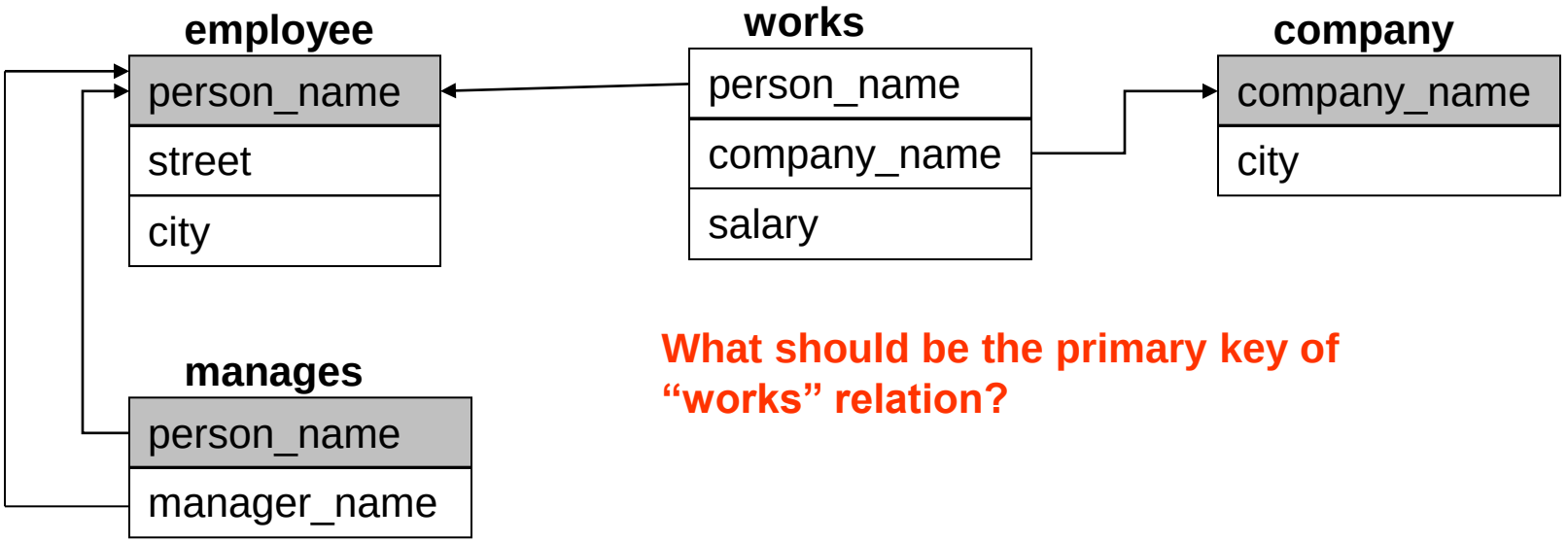
- Key Constraints
  - Primary Keys and Candidate Keys
- Referential Integrity
  - Foreign Keys
- Attribute and Tuple based Constraints
  - not null, default values, check conditions
- Assertions

Primary Key Constraints in SQL DDL

```
create table account
        ( account_number        char(10),
          branch_name           char(15),
          balance               numeric(18,2) not null,
          primary key (account_number),
          foreign key (branch_name) references branch,
          check (balance >= 0)                          );
```

- Note: Only one primary key specification for a relation is allowed
- Primary key attributes are implicitly not null and unique

Primary Key and Business Logic



What should be the primary key of "works" relation?

- Option 1: person_name
- Option 2: { person_name, company_name}
- What does it mean in terms of our business logic?

Reminder: Constraints come from business rules and logic!

Candidate Key Constraints in SQL DDL

```
create table employee
        ( employee_id          char(10),
          name                 varchar(40),
          email                varchar(50)
          street               varchar(45),
          city                 varchar(30),
          primary key (employee_id),
          unique (email)                    );
```

- Note: **Unique** constraint **allows null values** unless it has explicitly been declared to be not null.

  email varchar(50) not null

Not Null Constraint

- null value signifies either **the value does not exist** or **is unknown**
- **null value is a member of all possible domain**
- Business Logic may prohibit certain attributes to contain null values

```
create table account
        ( account_number        char(10),
          branch_name           char(15),
          balance               numeric(22,2) not null,
          primary key (account_number),
          foreign key (branch_name) references branch,
          check (balance >= 0)                          );
```

Check Constraint

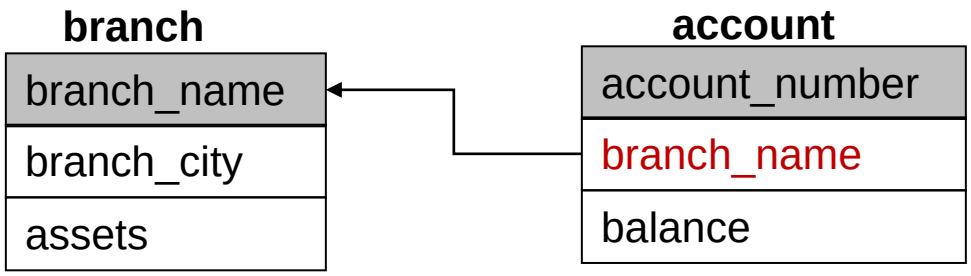
- Use check constraint to ensure attribute values are in certain range of values

```
create table account
        ( account_number        char(10),
          branch_name           char(15),
          balance               number(22,2) not null,
          primary key (account_number),
          foreign key (branch_name) references branch,
          check (balance >= 0)                          );
```

```
create table student
        ( student_id            char(10),
          name                  char(30) not null,
          degree_level          char(15),
          primary key (student_id),
          check (degree_level in ('Bachelors', 'Masters', 'PhD'))  );
```

Referential Integrity and Foreign Keys - 1

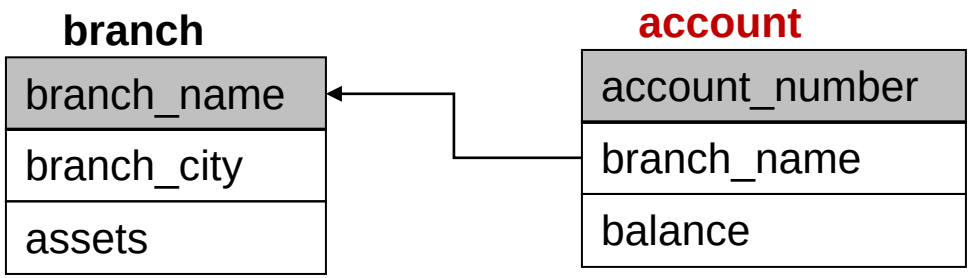
- **Foreign Key**: Set of attributes in one relation that is used to 'refer' to a tuple in another relation
  - **Must correspond to primary or candidate key of the 'referred' relation**




```
create table account
        ( account_number        char(10),
          branch_name           char(15),
          balance               number(22,2) not null,
          primary key (account_number),
          foreign key (branch_name) references branch(branch_name),
          check (balance >= 0)                              );
```

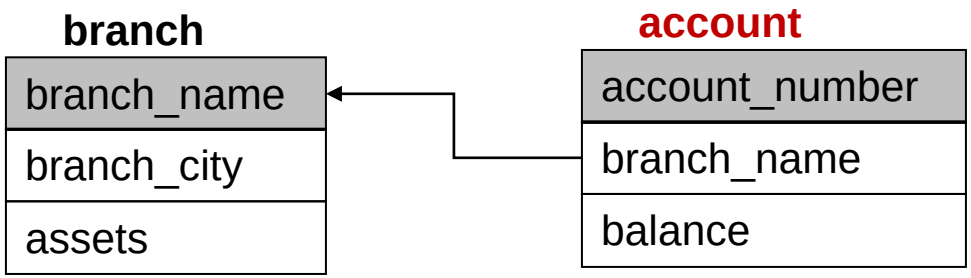
Only branch names listed in the branch relation should be appeared as branch name of accounts in account relation

Referential Integrity and Foreign Keys - 2



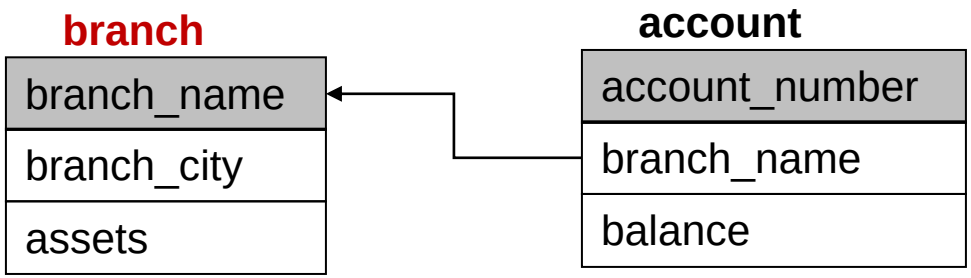
- **Assume a new account** (‘A-789’,**’Hellington’**, 1200.00) is opened, but the bank does **not have a branch known as ‘Hellington’**. What should we do?

Referential Integrity and Foreign Keys - 3



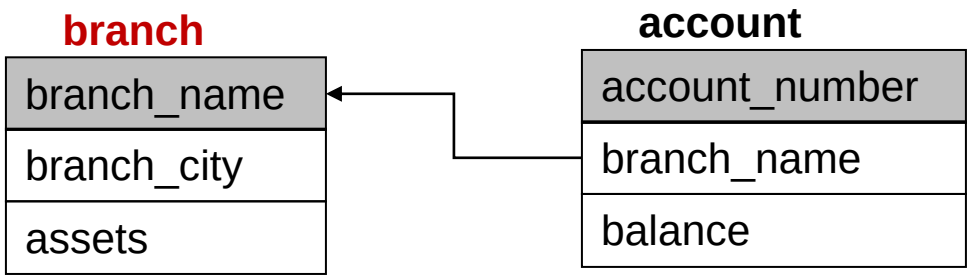
- **Assume a new account** (‘A-789’,**’Hellington’**, 1200.00) is opened, but the bank does **not have a branch known as ‘Hellington’**. What should we do?
  - Reject the operation!; or first Make sure there is a ‘Hellington’ branch

Referential Integrity and Foreign Keys - 4



- What should be done if **a tuple** in branch relation is **deleted** (or update)?

Referential Integrity and Foreign Keys - 5



- What should be done if **a tuple** in branch relation is **deleted** (or update)?
  a) Disallow the deletion or the update of the branch name in branch relation
  b) Delete all account relation having the same branch name → cascade
  c) Set the branch_name of account relation to null → set null
  d) Set the branch_name of account relation to a default value → set default

Referential Integrity - 1

```
create table account
        ( account_number        char(10),
          branch_name           char(15),
          balance               number(22,2) not null,
          primary key (account_number),
          foreign key (branch_name) references branch(branch_name),
          check (balance >= 0)                              );
```

a) **Default** is to **reject** the delete or update **operations** on branch relation *that will cause the referential integrity to be broken*

- when you do not specify any rule on the foreign key specification

Referential Integrity - 2

b) Delete all account relation having the same branch name → cascade

```
foreign key (branch_name) references branch(branch_name)
                                on delete cascade
                                on update cascade,
```

c) Set the branch_name of account relation to null → set null

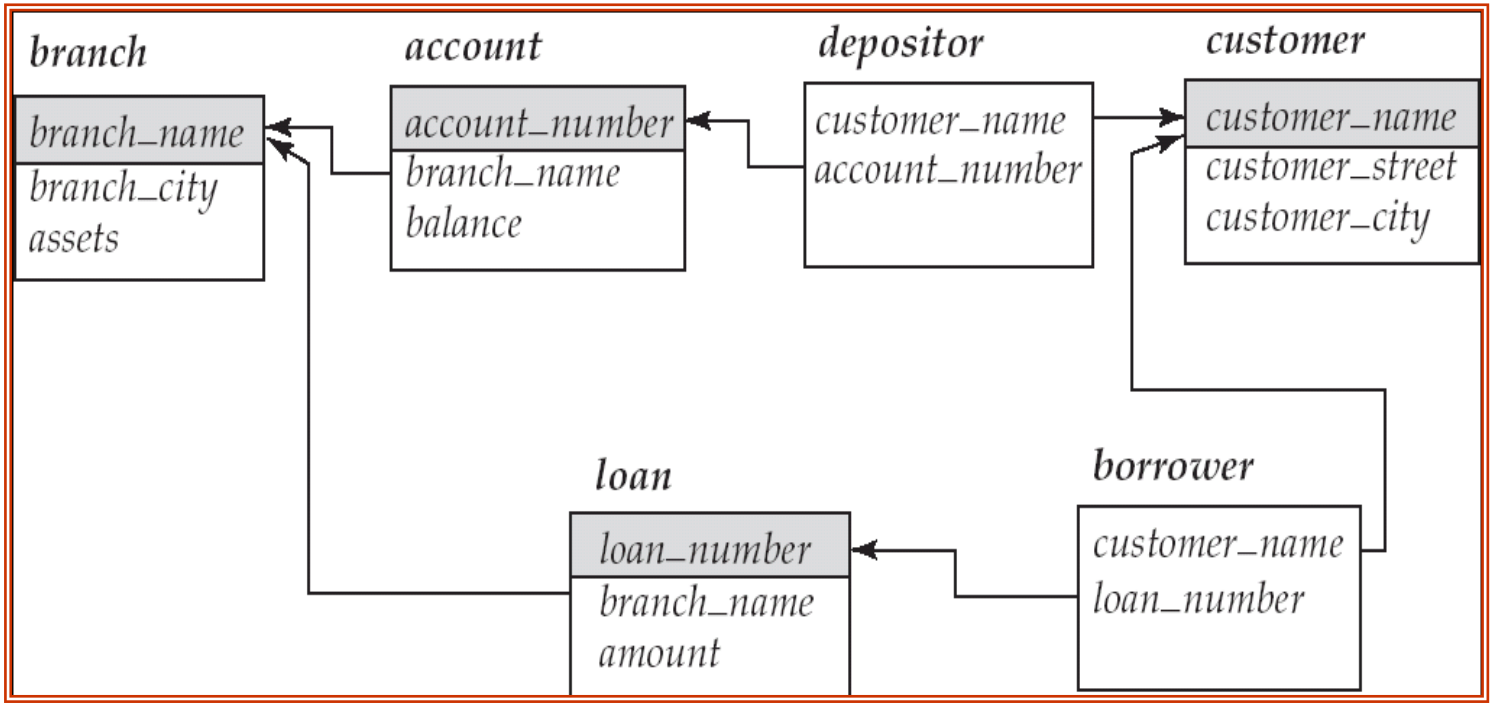
```
foreign key (branch_name) references branch(branch_name)
                                on delete set null
                                on update cascade,
```

d) Set the branch_name of account relation to a default value → set default

```
foreign key (branch_name) references branch(branch_name)
                                on delete set default
                                on update cascade,
```

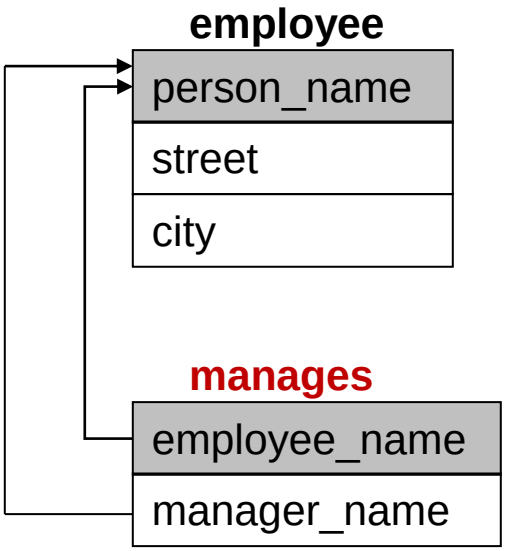
Note: The foreign key constraint is specified in the relation Account.

Referential Integrity Example - 1



- Assume all foreign keys are having referential integrity rule of 'cascade'
  - What will happen when **a following tuple** is **removed** from **branch** relation?
    - ('Perryridge', 'Horseneck', 1700000)

Referential Integrity Example - 2




```
create table manages
    ( employee_name          char(20),
      manager_name           char(20) default 'No Manager' ,
      primary key (employee_name),
      foreign key (employee_name) references employee(person_name)
                                   on delete cascade
                                   on update cascade,

      foreign key (manager_name) references employee(person_name)
                                   on delete set default
                                   on update cascade
)
```

Alternatively, use **on delete set null** *if there were no default value* defined *and null were allowed* for manager_name field

- In **manages** relation,
  - *employee_name* is the primary key
    - Each employee can have at most one manager
  - There are two foreign key specifications
    - foreign key (employee_name) references employee(person_name)
    - foreign key (manager_name) references employee(person_name)

employee_name	manager_name
A	No Manager
B	A
C	A
D	B

What if 'A' quits the job?

Assertions (in the SQL-92 standard)

- An **assertion** is a predicate expressing a condition that the database will always satisfy.
- Syntax:

  create assertion <assertion-name> **check** <predicate>

- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
- No current SQL product supports CREATE ASSERTION
  - Use CHECK constraints instead.

Why don't DBMS's support ASSERTION:
https://stackoverflow.com/questions/6368349/why-dont-dbmss-support-assertion

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance or $1000.00

```
create assertion balance_constraint check
    (not exists (
        select *
        from loan
        where not exists (
            select *
            from borrower, depositor, account
            where loan.loan_number = borrower.loan_number
                and borrower.customer_name = depositor.customer_name
                and depositor.account_number = account.account_number
                and account.balance >= 1000)));
```

Trigger

- Database operations that are automatically performed when a specified database event occurs.
- Allowed in many DBMS (including postgresql)

https://www2.navicat.com/manual/online_manual/en/navicat/linux_manual/TriggersOthersPGSQL.html

Creating Trigger In PostgreSQL

1. Create a trigger function using CREATE FUNCTION statement.

   CREATE FUNCTION trigger_function() **RETURN trigger AS**

2. Bind this trigger function to a table using CREATE TRIGGER statement.

   CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
   ON table_name
   [**FOR [EACH] {ROW** | STATEMENT}]
   EXECUTE PROCEDURE trigger_function

http://www.postgresqltutorial.com/creating-first-trigger-postgresql/

Trigger Example – 1
Auditing Last Name's Changes

```
CREATE TABLE employees(
  id serial primary key,
  first_name varchar(40) NOT NULL,
  last_name varchar(40) NOT NULL
);
```

```
CREATE TABLE employee_audits (
  id serial primary key,
  employee_id int4 NOT NULL,
  last_name varchar(40) NOT NULL,
  changed_on timestamp(6) NOT NULL
)
```

http://www.postgresqltutorial.com/creating-first-trigger-postgresql/

1. Creating a Trigger Function

```
CREATE OR REPLACE FUNCTION log_last_name_changes() RETURNS trigger AS
$BODY$
BEGIN
   IF NEW.last_name <> OLD.last_name THEN
      INSERT INTO employee_audits(employee_id,last_name,changed_on)
      VALUES(OLD.id,OLD.last_name,now());
   END IF;

   RETURN NEW;
END;
$BODY$
language plpgsql;
```

http://www.postgresqltutorial.com/creating-first-trigger-postgresql/

2. Binding the Trigger Function to a Table

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
     EXECUTE PROCEDURE log_last_name_changes();
```

http://www.postgresqltutorial.com/creating-first-trigger-postgresql/

3. Test the Trigger

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');

INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');

SELECT * FROM employees;

UPDATE employees
SET last_name = 'Brown'
WHERE ID = 2;

SELECT * FROM employees;

SELECT * FROM employee_audits;
```

http://www.postgresqltutorial.com/creating-first-trigger-postgresql/

More Trigger's examples

- https://www.postgresql.org/docs/9.2/static/plpgsql-trigger.html

Outline

- How to define Database Schema using SQL DDL
  - Domain (Data type) Specification, Integrity Constraints Specification
- **How to express Queries using SQL DML**
  - **Basic Structures of SQL Query expression**

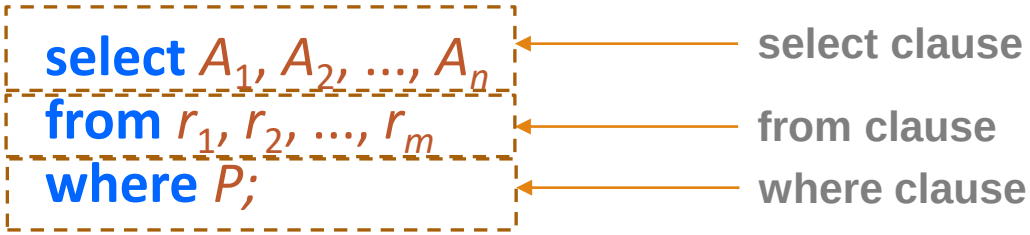
SQL DML and Relational Algebra

- SQL DML: provides convenient and efficient way of retrieving and manipulating data stored on a relational database
- SQL DML is based on Relational Algebra and Relational Calculus
  - Similar to relational algebra, but NOT EXACTLY the same

	Relational Algebra	SQL DML
Operands	Relations (sets of tuples)	Tables (multisets (bag) of rows)
Duplicate tuples	Removed from the output relation	NOT automatically removed from the output

Basic Structure of a Query in SQL

- A typical SQL query has the form:



  - A_i represents an **attribute**
  - r_i represents a **relation**
  - P is a **predicate**.
- This query is equivalent to the relational algebra expression.

$$\Pi_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a table (a relation)

The select clause - 1

- The **select** clause list the attributes desired in the result of a query
  - → the **projection operation** of the relational algebra

- Example: find the names of all branches in the *loan* relation:

SQL

```
select branch_name
from loan;
```

relational algebra

$$\Pi_{branch\_name}(loan)$$

The select clause - 2

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

```
select branch_name
from loan;
```

The select clause - 3

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

```
select branch_name
from loan;
```

branch_name
Round Hill
Downtown
Perryridge
Perryridge
Downtown
Redwood
Mianus

The select clause - 3

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

```
select distinct branch_name
from loan;
```

branch_name
Round Hill
Downtown
Perryridge
Redwood
Mianus

Removal of duplicates can be expensive.

So, use the keyword distinct only when necessary

The select clause - 4

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

loan relation

Keyword all specifies that duplicates are not removed (set by default)

```
select branch_name
from loan;
```

branch_name
Round Hill
Downtown
Perryridge
Perryridge
Downtown
Redwood
Mianus

```
select all branch_name
from loan;
```

The select clause - 5

- An asterisk in the select clause denotes "all attributes"

```
select *
from loan;
```

- The select clause can
  - Contain arithmetic expressions (+, –, ∗, and /)
  - Operate on constants or attributes of tuples.
- The query:

```
select loan_number, branch_name, (amount ∗ 5) / 100
from loan;
```

Result: Same as the *loan* relation *except* that the *amount*'s value *becomes 5% interest*.

  - Equivalent to "Generalized Project" in relational algebra

The from clause - 1

- The from clause lists the relations involved in the query

```
select *
from loan;
```

The from clause - 2

- Can **include multiple relations** in the from clause when data from multiple relations need to be obtained.
  - → the **Cartesian product** operation of the relational algebra.
- Example,

SQL

```
select *
from borrower, loan;
```

relational algebra

(borrower×*loan*)

Find the Cartesian product *borrower X loan*

***borrower* relation**

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

***loan* relation**

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

What is the total number of tuples obtained?

customer_name	borrower. loan_number	loan. loan_number	branch_name	amount
Adams	L-16	L-11	Round Hill	900
Curry	L-93	L-11	Round Hill	900
Hayes	L-15	L-11	Round Hill	900
Jackson	L-14	L-11	Round Hill	900
Jones	L-17	L-11	Round Hill	900
Smith	L-11	L-11	Round Hill	900
Smith	L-23	L-11	Round Hill	900
Williams	L-17	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Curry	L-93	L-14	Downtown	1500
Hayes	L-15	L-14	Downtown	1500
Jackson	L-14	L-14	Downtown	1500
Jones	L-17	L-14	Downtown	1500
Smith	L-11	L-14	Downtown	1500
Smith	L-23	L-14	Downtown	1500
Williams	L-17	L-14	Downtown	1500
…	…	…	…	…
Williams	L-17	L-93	Mianus	500

The from clause - 3

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

***borrower* relation**

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

***loan* relation**

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

The from clause - 4

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

***borrower* relation**

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

***loan* relation**

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

```
select  customer_name,
        borrower.loan_number, amount
from    borrower, loan
where   borrower.loan_number = loan.loan_number
and     branch_name = 'Perryridge';
```

from borrower, loan

customer_name	borrower. loan_number	loan. loan_number	branch_name	amount
Adams	L-16	L-11	Round Hill	900
Curry	L-93	L-11	Round Hill	900
Hayes	L-15	L-11	Round Hill	900
Jackson	L-14	L-11	Round Hill	900
Jones	L-17	L-11	Round Hill	900
Smith	L-11	L-11	Round Hill	900
Smith	L-23	L-11	Round Hill	900
Williams	L-17	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Curry	L-93	L-14	Downtown	1500
Hayes	L-15	L-14	Downtown	1500
Jackson	L-14	L-14	Downtown	1500
Jones	L-17	L-14	Downtown	1500
Smith	L-11	L-14	Downtown	1500
Smith	L-23	L-14	Downtown	1500
Williams	L-17	L-14	Downtown	1500
…	…	…	…	…
Williams	L-17	L-93	Mianus	500

```
from    borrower, loan
where   borrower.loan_number =
        loan.loan_number
```

customer_name	borrower. loan_number	loan. loan_number	branch_name	amount
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-15	Perryridge	1500
Jackson	L-14	L-14	Downtown	1500
Jones	L-17	L-17	Downtown	1000
Smith	L-11	L-11	Round Hill	900
Smith	L-23	L-23	Redwood	2000
Williams	L-17	L-17	Downtown	1000

```
from    borrower, loan
where   borrower.loan_number =
        loan.loan_number and
        branch_name = 'Perryridge'
```

customer_name	borrower. loan_number	loan. loan_number	branch_name	amount
Adams	L-16	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500

```
select  customer_name,
        borrower.loan_number, amount
from    borrower, loan
where   borrower.loan_number =
        loan.loan_number and
        branch_name = 'Perryridge' ;
```

customer_name	borrower. loan_number	amount
Adams	L-16	1300
Hayes	L-15	1500

The from clause and duplicates

- SQL allows duplicates in relations;
  - SQL works on **bag** of tuples
  - Relational Algebra works on **set** of tuples

B
a
a

r

C
2
3
3

s

select *
from r, s;

B	C
a	2
a	3
a	3
a	2
a	3
a	3

The where clause - 1

- The **where** clause **specifies conditions** that the result must satisfy
  - → **selection predicate** of the relational algebra.
- **Test conditions** are built by using **comparison operators**
  - <, <=, >, >=, =, <> (SQL)
  - $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ (Relational Algebra)
  - **Operands** of the operators are **attributes** and **arithmetic expressions** involving constant values and attributes
  - SQL also supports comparisons of **string**, **time** and **date**
- Comparison results can be combined using the **logical connectives**
  - and, or, not (SQL)
  - $\wedge$, $\vee$, $\neg$ (Relational Algebra)

The where clause - 2

- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

***loan* relation**

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

The where clause - 2

- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

***loan* relation**

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

```
select loan_number
from loan
where branch_name = 'Perryridge'
and amount > 1200;
```

loan_number
L-15
L-16

Keyword between - 1

- SQL includes a **between** comparison operator

Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

```
select loan_number
from loan
where amount >= 90000 and amount <= 100000;
```

Keyword between - 2

- SQL includes a **between** comparison operator

Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

```
select loan_number
from loan
where amount >= 90000 and amount <= 100000;
```

```
select loan_number
from loan
where amount between 90000 and 100000;
```

Keyword between - 3

- not between comparison operator is also provided

Example: Find the loan number of those loans with loan amounts NOT between $90,000 and $100,000 (that is, < $90,000 or > $100,000)

```
select loan_number
from loan
where amount < 90000 or amount > 100000;
```

```
select loan_number
from loan
where amount not between 90000 and 100000;
```

String Comparisons - 1

- SQL specifies string by enclosing them *in single quotes*, e.g.) 'Downtown'
- SQL provides operator like for pattern matching
  - percent (%)
    - Match any substring (of any size including size of 0).
  - underscore (_)
    - Match any single character.
- Find the names of all customers whose street includes the substring "idge".

```
select customer_name
from customer
where customer_street like '%idge%' ;
```

String Comparisons - 2

- ‘%idge%’ → ‘Perryidge’, ‘Rock Ridge’, ‘Mianus Bridge’, ‘Ridgeway’, etc….
- ‘Perry%’ → ‘Perryridge’, ‘Perry’, ‘Perry the gunman’, etc
- ‘_ _ _%’ → Matches any string of at least three characters

String Comparisons - 3

- Escape keyword is used to treat the **special characters** as normal characters
  - like ‘ab\%cd%’ : matches all string beginning with ‘ab%cd’
  - like ‘ab\\cd%’ : matches all string beginning with ‘ab\cd’

Reminder on Data Types and Operations

- Domains of Attributes are specified by SQL Data types and other optional constraints (such as not null, check conditions)
- **Each data type** has a set of **associated operations** that are permitted
  - Arithmetic operations on number types
  - String operations: concatenation, substring matching, etc
- Conversions allowed among different data types

Recap: How is an SQL query evaluated - 1

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;

```
select customer_name,  borrower.loan_number,  amount
from  borrower, loan
where borrower.loan_number = loan.loan_number;
```

- **Step 1**: The **from** clause tells us the input tables.
  - Cartesian product of every relation listed is formed
  - Be reminded duplicates are allowed and maintained

Recap: How is an SQL query evaluated - 2

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

select customer_name, borrower.loan_number, amount
from borrower, loan
where **borrower.loan_number = loan.loan_number;**

- **Step 2**: The **where** clause is evaluated (the predicate conditions) for every tuple formed from the from clause
  - If TRUE, the tuple is selected for the output.
  - Otherwise, the tuple is not included in the output.

Recap: How is an SQL query evaluated - 3

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P

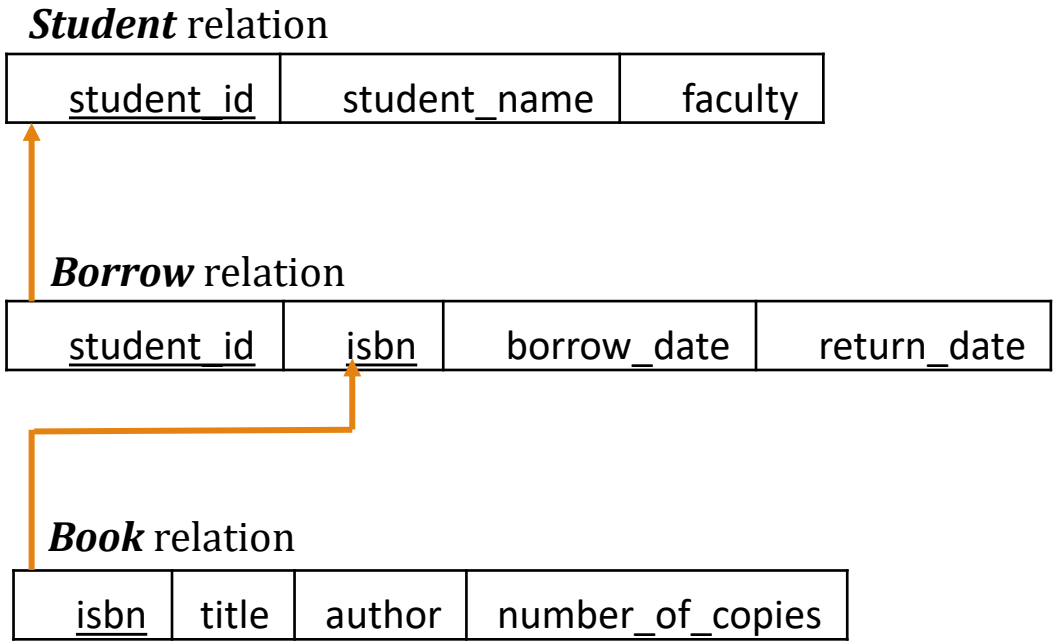
select **customer_name, borrower.loan_number, amount**
from borrower, loan
where borrower.loan_number = loan.loan_number;

- **Step 3**: The **select** clause determines which attributes to keep in the output table
  - Duplicates are allowed in the output unless distinct keyword is specified

Practice 4-1

- Create the Database for the Library in Postgresql to keep records of students who borrow books.
  - Write SQL DDL to create database schema and necessary constraints.
  - Explain your constraints in your own words.

Library Database



Library Relations - 1

Student relation

student_id	student_name	faculty
5725001	Paul Smith	Science and Technology
5815002	Alice Summerville	Science and Technology
5817013	Masha Winston	Laws
5819020	Tom Lee	Biology
5811051	Mark Cooper	BBA
5915004	Peter Highlander	BBA

Borrow relation

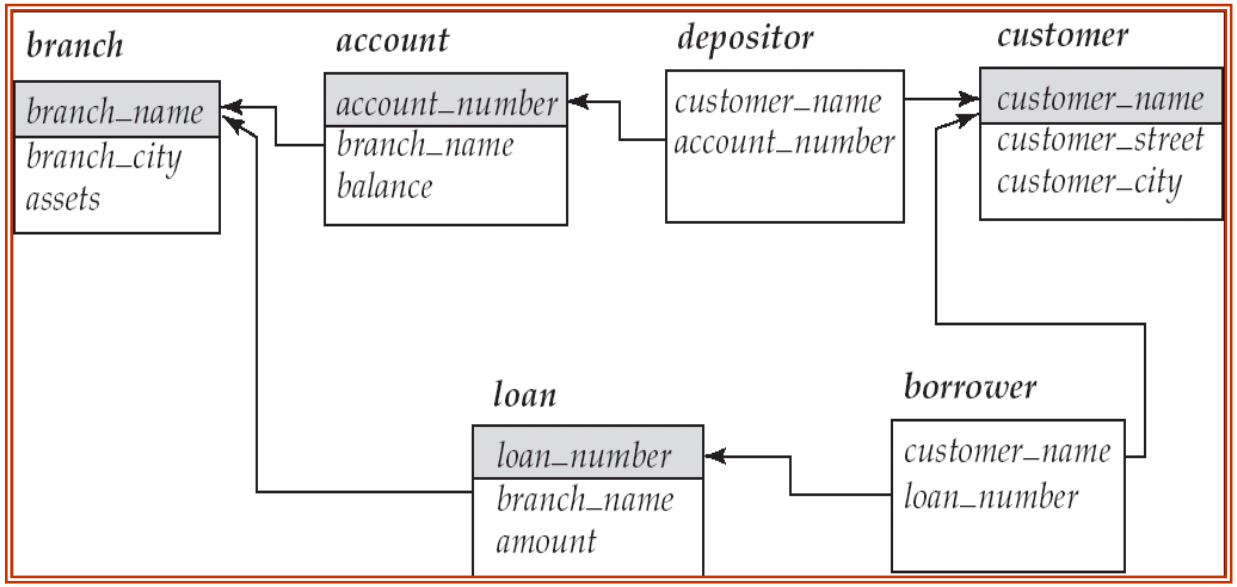
student_id	isbn	borrow_date	return_date
5725001	0760555841236	2018-01-10	2018-01-14
5815002	0760555841236	2018-01-11	2018-01-15
5817013	0251462157459	2018-01-12	2018-01-18
5819020	0760482456211	2018-01-15	2018-01-18
5815002	0584215622477	2018-01-15	
5815002	0584215622477	2018-01-22	2018-01-26
5819020	0154215871222	2018-01-23	
5915004	0154215871222	2018-01-25	

Library Relations - 2

Book relation

isbn	title	author	number_of_copies
0584215622477	The Foundation in Mathematics	Ian Stewart	5
0154215871222	English Grammar in Use	Ray Murphy	5
0251462157459	Civil Laws	H. Patrick Glenn	2
0760482456211	Bioinformatics	Dan Gusfield	2
0760555841236	Python Programming	Guido van Rossum	3

Practice 4-2



- Create the Database for the sample banking enterprise on your Postgresql
  - Define the Database Schema using SQL DDL
  - Run the given sql script to insert records into tables
  - Write simple query to perform the following tasks and show the results generated (next slide)

Write SQL commands to perform the following tasks and **show the results** generated

1. Find all customers' name who have deposit account.
2. Find all customer's name who have both deposit and loan accounts.
3. Find number and amount of all loans that have multiple owners.
4. Find name of all customers who lives in Stamford.
5. Find name and account number of all customers who lived in Harrison and have account at Brighton.

More DBMS-based Data Types

(Physical) Domain Types in SQL-92

Data Type	Alternative format	Description
char(n)	character(n)	**Fixed length character string with user-specified length n**
varchar(n)	character varying (n)	**Variable length character string with user-specified length n**
clob(s)		For storing huge text information
int	integer	An integer (machine dependent range); (int in C)
smallint		A smaller range version (machine dependent) (short in C)
numeric(p,d)		**A fixed-point number with user-specified precision**
dec(p,d)	decimal(p,d)	Almost identical to numeric (Exact precision)
real		Machine dependent single precision floating point number
double precision		Machine dependent double precision floating point number
float(n)		A floating point number, with precision of at least n digits
boolean		True or false
date		Date containing a year (4 digit), month and day; no time information
time(p)		Time of a day in hours, minutes and seconds; no date information
timestamp(p)		Date plus time
interval		Period of time
bit(n)		Fixed length bit vector
bit varying(n)		Variable length bit vector
BLOB(s)		For strong large binary information

PostgreSQL Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(n)]		fixed-length bit string
bit varying [(n)]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [(n)]	char [(n)]	fixed-length character string
character varying [(n)]	varchar [(n)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [fields] [(p)]		time span
json		textual JSON data
jsonb		binary JSON data, decomposed
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount
numeric [(p, s)]	decimal [(p, s)]	exact numeric of selectable precision
path		geometric path on a plane
pg_lsn		PostgreSQL Log Sequence Number
point		geometric point on a plane
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(p)] [without time zone]		time of day (no time zone)
time [(p)] with time zone	timetz	time of day, including time zone
timestamp [(p)] [without time zone]		date and time (no time zone)
timestamp [(p)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot
uuid		universally unique identifier
xml		XML data

Ref: https://www.postgresql.org/docs/9.5/static/datatype.html

Oracle (10gR2) Data Types

char(n)	**Fixed length character string with user-specified length n**
varchar(n)	**Deprecated**; **use varchar2(n) below**
varchar2(n)	**Variable length character string with user-specified length n**
clob	Character large object
blob	binary Large object
bfile	Pointer to binary file on disk
number(p,s)	**A fixed-point number with user-specified precision**
binary_float	32-bit single precision floating point number (each value requires 5 bytes)
binary_double	64-bit double precision floating point number (each value requires 9 bytes)
date	Date and time, but no fractional seconds
timestamp	Date and time with fractional seconds, but no time zone information
timestamp with time zone	Date and time with fractional seconds, include explicit time zone information
timestamp with local time zone	Date and time with fractional seconds, include relative time zone information
rowid	Hexadecimal string representing the unique address of a row in its table
urowid	Hexadecimal string representing the logical address of a row of an index-organized table
XMLType	For representing XML data
Uritype	Pointer to data inside or outside the database
raw(n)	Raw binary data of length of n bytes
long	**Deprecated**; use clob instead
long raw	**Deprecated**; use blob or bfile instead

Some useful links

- Add PostgreSQL not-null constraint to columns of existing table
  - http://www.postgresqltutorial.com/postgresql-not-null-constraint/
- Alter Table (e.g., ADD table constraint)
  - https://www.postgresql.org/docs/8.2/static/sql-altertable.html