CSE 4020/5260
Database Systems

Instructor: Fitzroy Nembhard, Ph.D.

Week 8 - 9

DML

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Database Systems (CSE 4020/5260) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

Structured Query Language (SQL)

- Basic SQL Query Structure
- Set Operations
- Aggregate Functions
- Nested Subqueries
- Derived Relations
- Views
- Modification of a Database
- Specialized Join Operation

Banking Example

- SQL is a “standardized” language, but most vendors have their own version.
- Queries are typically submitted on the command-line, using a client query tool, via program code, or through an API.
- Now is the time to start issuing queries, just to get the hang of it!
- White space will be used liberally throughout.

Banking Example

- Recall the banking database:

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

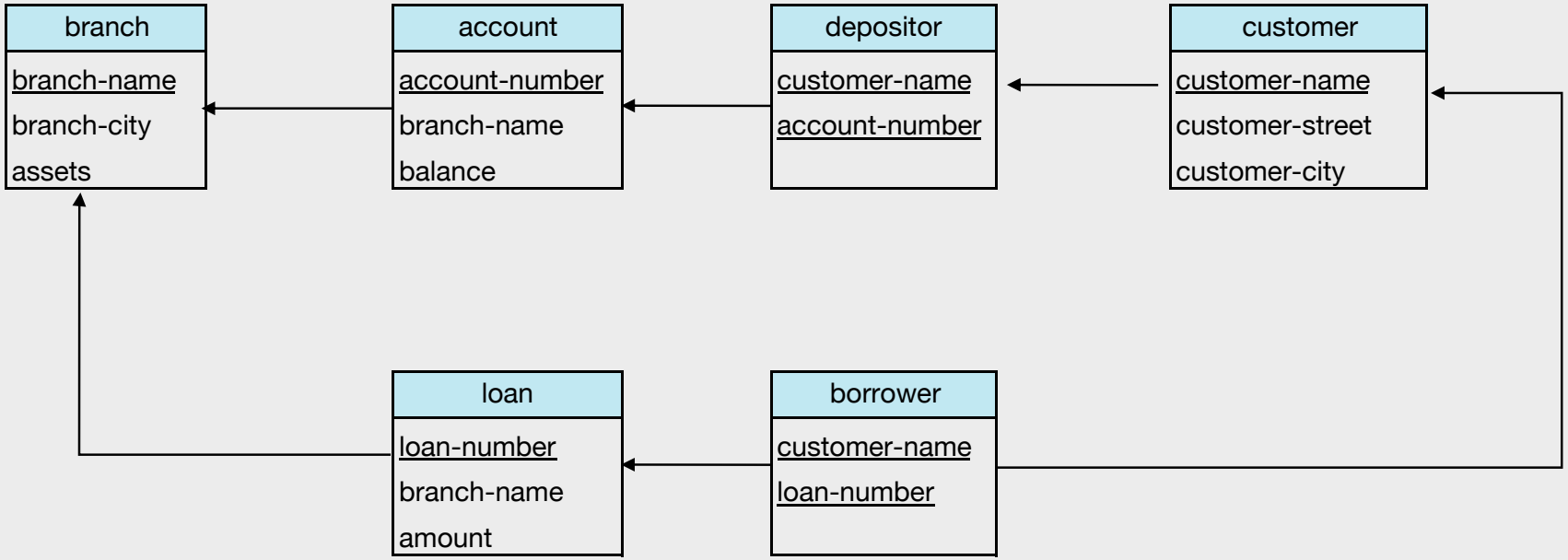
account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Schema Used in Examples



Basic Structure

- Typical SQL statement/query structure:

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

- Equivalent (sort of) to:

$$\Pi_{A1, A2, ..., An}(\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

The select Clause

- **select** clause - lists desired attributes (corresponds to projection).

  "Find the names of those branches that have outstanding loans."

select *branch-name*
from *loan*

$$\Pi_{\text{branch-name}}(loan)$$

select *branch-name, loan-number*
from *loan*

$$\Pi_{\text{branch-name,loan-number}}(loan)$$

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The select Clause (Cont.)

Generalized projection:
$\Pi_{F1, F2, \ldots, Fn}(E)$

- An asterisk denotes all attributes:

  select *
  from *loan*

- **select** can contain expressions (corresponds to generalized projection).

  select *loan-number, branch-name, amount* ∗ 100
  from *loan*

- Note that the above does not modify the table.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The select Clause (Cont.)

- The basic SQL select statement does NOT eliminate duplicates.
- Keyword **distinct** is used to eliminate duplicates.

"Find the names of those branches that have outstanding loans (no duplication)."

select distinct *branch-name*
from *loan*

- Keyword **all** can be used (redundantly) when duplicates desired.

select all *branch-name*
from *loan*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The where Clause

- **where** clause - specifies conditions on the result (corresponds to selection).

  "Find the loan numbers for all loans over $1200 made at the Perryridge branch."

  select *loan-number*
  from *loan*
  where *branch-name* = 'Perryridge' **and** *amount* > 1200

SELECT column1, column2, ...
FROM table_name
WHERE **NOT** condition;

- Logical connectives **and, or,** and **not** can be used**.**

- Comparisons can be applied to results of arithmetic expressions.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The from Clause

- **from** clause - lists required relations (corresponds to Cartesian product).

  "Find the Cartesian product borrower x loan."

  select ∗ **from** *borrower, loan*

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

  select *borrower.customer-name, borrower.loan-number, loan.amount*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number* **and**
  loan.branch-name = 'Perryridge'

- Note the use of expanded name notation in the above.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The from Clause

- Sometimes mixed-use notation is used:

select *customer-name, borrower.loan-number, amount*
from *borrower, loan*
where *borrower.loan-number = loan.loan-number* **and**
branch-name = ‘Perryridge’

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

The Rename Operation

- Attribute renaming (**as**):

- In the **select** clause (for column renaming):

  "Find the name, loan number and loan amount of all customers; rename the loan-number column loan-id."

  select *customer-name, borrower.loan-number* **as** *loan-id, amount*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Tuple Variables

- In the **from** clause (for abbreviating):

"Find the customer names, their loan numbers and loan amounts for all customers having a loan at the Perryridge branch."

select *T.customer-name, T.loan-number, S.amount*
from *borrower* **as** *T, loan* **as** *S*
where *T.loan-number = S.loan-number*
and *S.branch-name = 'Perryridge'*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Tuple Variables

- It can also be used to resolve ambiguous relation names:

  "Find the names of all branches that have greater assets than some branch located in Brooklyn."

```
select distinct T.branch-name
        from branch as T, branch as S
        where T.assets > S.assets and S.branch-city = 'Brooklyn'
```

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

String Operations

- So how about strings?
- SQL supports a variety of string processing functions…surprise!!!
- Example:

"Find the names of all customers whose street includes the substring 'Main'."

```
select customer-name
        from customer
        where customer-street like '%Main%'
```

String Operations

- Other SQL string operations:
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

- Most COTS DBMS query processors augment SQL string processing with even more operations; the list is typically very long.

Ordering the Display of Tuples

- Sorting:

  "List in alphabetic order the names of all customers having a loan at the Perryridge branch."

  select distinct *customer-name*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number* **and**
  branch-name = 'Perryridge'
  order by *customer-name*

- **desc** or **asc** (the default) can be specified:
  - **order by** *customer-name* **desc**

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Ordering the Display of Tuples

- Sorting on multiple attributes (with both **asc** and **desc**):

- Example: add loan amount to the previous query:

select distinct *customer-name, amount*
from *borrower, loan*
where *borrower.loan-number = loan.loan-number* **and**
branch-name = 'Perryridge'
order by *customer-name* **asc**, *amount* **desc**

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Set Operations

- **union, intersect,** and **except** ($\cup$, $\cap$, $-$, respectively):
  - *r* **union** *s*
  - *r* **intersect** *s*
  - *r* **except** *s*

  where *r* and *s* are either relations or sub-queries.

- The above operations all automatically eliminate duplicates.

- Note: MySQL does not support the **except** clause (unlike Oracle, which uses the minus keyword in place of except, MySQL does not support except at all). You can use the **not in** clause instead.

Set Operations

"Find all customers who have a loan, an account, or both."

(select *customer-name* **from** *depositor*)
union
(select *customer-name* **from** *borrower)*

"Find all customers who have both a loan and an account."

(**select** *customer-name* **from** *depositor*)
intersect
(select *customer-name* **from** *borrower)*

"Find all customers who have an account but no loan."

(**select** *customer-name* **from** *depositor*)
except
(select *customer-name* **from** *borrower)*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Set Operations

- **union all, intersect all** and **except all** retain duplicates:

  If a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:

  - *m* + *n* times in *r* **union all** *s*
  - min(*m, n)* times in *r* **intersect all** *s*
  - max(0, *m* – *n)* times in *r* **except all** *s*

Aggregate Functions

- Grouping and aggregate functions.
- Basic aggregate functions:

avg	- average value
min	- minimum value
max	- maximum value
sum	- sum of values
count	- number of values

- Aggregate functions operate on groups.

Aggregate Functions, Cont.

"Find the average account balance."

select avg *(balance)*
from *account*

"Find the average account balance at the Perryridge branch."

select avg *(balance)*
from *account*
where *branch-name* = 'Perryridge'

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Aggregate Functions, Cont.

"Find the number of tuples in the depositor relation."

select count (*)
from *depositor*

Or any single or combination of columns:

select count (customer-name)
from *depositor*

select count (account-number)
from *depositor*

select count (customer-name, account-number)
from *depositor*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Aggregate Functions, Cont.

"Find the number of depositors in the bank."

select count (distinct *customer-name)*
from *depositor*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Aggregate Functions – Group By

- Aggregate functions applied to groups:

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

"Find the number of accounts for each branch."

select *branch-name,* **count** *(account-number)*
from *account*
group by *branch-name*

"Find the number of depositors for each branch."

select *branch-name,* **count (distinct** *customer-name)*
from *depositor, account*
where *depositor.account-number = account.account-number*
group by *branch-name*

- Why does the second have *distinct* but not the first?

Aggregate Functions – Group By

- Grouping can be on multiple attributes:

  "For each depositor, determine how many accounts that depositor has at each branch."

  select *customer-name, branch-name,* **count** (*depositor.account-number)*
  from *depositor, account*
  where *depositor.account-number = account.account-number*
  group by *customer-name, branch-name*

- Notes:
  - Should *distinct* have been included?
  - Attributes in the **select** clause outside of the aggregate functions must appear in **group by** list (e.g., delete *branch-name* from the group-by clause).
  - Group-by *might* require a sort.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Aggregate Functions – Group By

- Grouping on multiple attributes, and multiple aggregate functions.

  "For each depositor, determine how many accounts that depositor has at each branch, plus the average, min and max balance for any account at that branch."

```
select customer-name,
       branch-name,
       count (depositor.account-number)
       avg (account.balance)
       min (account.balance)
       max (account.balance)
from depositor, account
where depositor.account-number = account.account-number
group by customer-name, branch-name
```

Schema

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Aggregate Functions – Having Clause

- Groups can be selected or eliminated using the *having* clause.

"Find those branches in Orlando with an average balance over 1200."

select *branch-name*
from *account, branch*
where *account.branch-name = branch.branch-name*
and *branch-city = 'Orlando'*
group by *branch-name*
having avg *(balance)* > 1200

- Predicates in the **having** clause are applied after the formation of groups, but those in the **where** clause are applied before forming groups.

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Null Values

- It is possible for tuples to have a *null* value for some attributes.
- *null* signifies an unknown value or that a value does not exist.
- The rules for null values are consistent with relational algebra (repeated on the following pages), except for the following addition…
- The predicate **is null** can be used to check for null values.

"Find all loan numbers in the loan relation with null values for amount."

select *loan-number*
from *loan*
where *amount* **is null**

Null Values and Three Valued Logic

- Rule #1 - Any comparison with *null* (initially) returns *unknown:*
  - *5 < null or null <> null or null = null*

select *loan-number*
from *loan*
where *amount > 50*

select *borrower-name, branch-name*
from *borrower, loan*
where *borrower.loan-number = loan.loan-number*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

- Rule #2 - The result of any arithmetic expression involving *null* is *null*
  - *5 + null* evaluates to *null*

select *loan-number*
from *loan*
where *amount*100 > 50000*

Null Values and Three Valued Logic

- Rule #3 - A “three-valued logic” is applied to complex expressions:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown* (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown, (false* **and** *unknown) = false, (unknown* **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - “*P* **is unknown**” evaluates to *true* if predicate *P* evaluates to *unknown*

select *loan-number*
from *loan*
where *amount*100 > 5000* **and** *branch-name = “Perryridge”*

- Rule #4 - Final result of a **where** clause predicate is treated as *false* if it evaluates to *unknown.*

select *loan-number*
from *loan*
where *amount*100 > 5000* **and** *branch-name = “Perryridge”*

Null Values, Cont.

- Rule #5 - Aggregate functions, except **count**, simply ignore nulls.

- Total all loan amounts:

  select sum (*amount)*
  from *loan*

  - Above statement ignores null amounts
  - Result is null if there is no non-null amount

Null Values and Expression Evaluation, Cont.

- This all seems like a pain…couldn't it be simplified?

- Why doesn't a comparison with *null* simply result in *false*?

If *false* was used instead of *unknown*, then:

not (A < 5)

would not be equivalent to:

A >= 5

Why would this be a problem?

Nested Subqueries

- SQL provides a mechanism for nesting queries.
- A sub-query is a **select** statement that is nested in another SQL query.
- Nesting is usually in a **where** clause but may be in a **from** clause.

Nested Subqueries

- Sub-query in a **where** clause typically performs a set test.

in	<comp> some	exists	unique
not in	<comp> all	not exists	not unique

where <comp> can be $<, \leq, >, =, \neq$

Example Query

"Find all customers who have both an account and a loan."

```
select distinct customer-name
        from borrower
        where customer-name in (select customer-name
                                from depositor)
```

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Query

"Find all customers who have a loan but do not have an account."

select distinct *customer-name*
from *borrower*
where *customer-name* **not in (select** *customer-name*
from *depositor)*

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Example Query

"Find the names of all customers who have both an account and a loan at the Perryridge branch."

```
select distinct customer-name
        from borrower, loan
        where borrower.loan-number = loan.loan-number and
              branch-name = "Perryridge" and
             (branch-name, customer-name) in
                    (select branch-name, customer-name
                     from depositor, account
                     where depositor.account-number =
                           account.account-number)
```

=> Note that the above query can be "simplified."

Example Query

"Find the names of all customers who have both an account and a loan at the Perryridge branch."

```
select distinct customer-name
        from borrower, loan
        where borrower.loan-number = loan.loan-number and
              branch-name = "Perryridge" and
             customer-name in
                    (select customer-name
                     from depositor, account
                     where depositor.account-number =
                           account.account-number and
                         branch-name = "Perryridge")
```

```
select distinct customer-name
        from borrower, loan
        where borrower.loan-number = loan.loan-number and
              branch-name = "Perryridge" and
          (branch-name, customer-name) in
                    (select branch-name, customer-name
                     from depositor, account
                     where depositor.account-number =
                         account.account-number)
```

Set Comparison – the "Some" Clause

"Find all branches that have greater assets than some branch located in Brooklyn."

```
select distinct T.branch-name
        from branch as T, branch as S
        where  T.assets > S.assets and
               S.branch-city = 'Brooklyn'
```

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Same query using > **some** clause:

```
select branch-name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch-city = 'Brooklyn')
```

Set Comparison – the "All" Clause

"Find the names of all branches that have greater assets than all branches located in Brooklyn."

```
select branch-name
        from branch
        where assets > all
                (select assets
                 from branch
                 where branch-city = 'Brooklyn')
```

Note that the some and all clauses correspond to existential and universal quantification, respectively.

Schema

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Definition of the "Some" Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ *s.t.* (F <comp> t)

(5< **some** | 0 / 5 / 6 |) = true

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**

However, (≠ **some**) ≢ **not in**

Definition of the "All" Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5< **all** | 0 / 5 / 6 |) = false

(5< **all** | 6 / 10 |) = true

(5 = **all** | 4 / 5 |) = false

(5 ≠ **all** | 4 / 6 |) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≠ **in**

Test for Empty Relations

- The **exists** operator can be used to test if a relation is empty.
- Operator **exists** returns **true** if its argument is nonempty.
  - **exists** $r \Leftrightarrow r \neq \emptyset$
  - **not exists** $r \Leftrightarrow r = \emptyset$
- On a personal note, why not call it **empty**?

Example Query

"Find all customers who have an account at all branches located in Brooklyn."

```
select distinct S.customer-name
        from customer as S
        where not exists (
                (select branch-name
                 from branch
                 where branch-city = 'Brooklyn')
                        except
                (select R.branch-name
                 from depositor as T, account as R
                 where T.account-number = R.account-number and
                       S.customer-name = T.customer-name))
```

Schema

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

- Because of the use of the tuple variable S in the nested query, the above is sometimes referred to as a *correlated* query.
- The above demonstrates that nesting can be almost arbitrarily composed and deep.
- According to the book, the above cannot be written using = **all** or its variants…hmmm…
- Recall that except does not work in MySQL

Test for Absence of Duplicate Tuples

- The **unique** operator tests whether a sub-query contains duplicate tuples.

  "Find all customers who have at most one account at the Perryridge branch."

```
select T.customer-name
      from customer as T
      where unique (
            select D.customer-name
            from account as A, depositor as D
            where T.customer-name = D.customer-name and
                  A.account-number = D.account-number and
                  A.branch-name = 'Perryridge')
```

- What if the inner query selected the account number?
  - count(…) <= 1

Example Query

"Find all customers who have at least two accounts at the Perryridge branch."

```
select distinct T.customer-name
from customer T
where not unique (
        select R.customer-name
        from account, depositor as R
        where T.customer-name = R.customer-name and
              R.account-number = account.account-number and
              account.branch-name = 'Perryridge')
```

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Nesting in the From-Clause

"Find the average account balance of those branches where the average account balance is greater than $1200."

```
select branch-name, avg-balance
        from (select branch-name, avg (balance)
                from account
                group by branch-name)
               as result (branch-name, avg-balance)
        where avg-balance > 1200
```

Note that previously we saw an equivalent query that used a *having* clause.

Views

- Purpose of a view:
  - Hide certain data from the view of certain users
  - Provide pre-canned, named queries
  - Simplify complex queries

- Syntax of a view:

  create view *v* **as** <query expression>

  where:
  - *v* - view name
  - <query expression> - view definition (SQL)

Example Views

- A view consisting of branches and their customers:

```
create view all-customer as
                (select branch-name, customer-name
                 from depositor as D, account as A
                 where D.account-number = A.account-number)
                      union
                (select branch-name, customer-name
                 from borrower as B, loan as L
                 where B.loan-number = L.loan-number)
```

"Find all customers of the Perryridge branch."

```
select customer-name
                from all-customer
                where branch-name = 'Perryridge'
```

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Modification of the Database – Insertion

- Basic insert*:*

  insert into *account* **values** (‘A-9732’, ‘Perryridge’,1200)

- Ordering values:

  insert into *account (branch-name, balance, account-number)*
  values (‘Perryridge’, 1200, ‘A-9732’)

- Inserting a null value:

  insert into *account* **values** (‘A-777’,‘Perryridge’, *null*)

Schema
branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Modification of the Database – Insertion

"Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new account."

```
insert into account
        select loan-number, branch-name, 200
        from loan
        where branch-name = 'Perryridge'

insert into depositor
        select customer-name, loan-number
        from loan, borrower
        where branch-name = 'Perryridge'
              and loan.account-number = borrower.account-number
```

- The above would typically be a transaction.

Schema

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Modification of the Database – Insertion

- Most DBMSs provide a command-line, bulk-load command:

```
LOAD DATA LOCAL INFILE '<file-path>' INTO TABLE part
       FIELDS TERMINATED BY '<file-separator>'
       LINES TERMINATED BY '<line-separator>';
```

- Example:

```
LOAD DATA LOCAL INFILE '../drfitz/department.csv' INTO TABLE department
       FIELDS TERMINATED BY ',' -- Specify the delimiter
       ENCLOSED BY '"'   -- When strings are enclosed in quotes
       LINES TERMINATED BY '\n'
       IGNORE 1 LINES -- or ROWS
```

Modification of the Database – Deletion

"Delete all tuples in the depositor table."

delete from *depositor*

"Delete all depositor records for Smith."

delete from *depositor*
where *customer-name* = 'Smith'

Modification of the Database – Deletion

"Delete all accounts at every branch located in Needham city."

```
delete from depositor
where account-number in
                (select account-number
                  from branch as B, account
                    where branch-city = 'Needham' and B.branch-name = A.branch-name)

delete from account
where branch-name in (select branch-name
                              from branch
                             where branch-city = 'Needham')
```

Example Query

"Delete the record of all accounts with balances below the average at the bank."

delete from *account*
where *balance* < (**select avg** *(balance)*
from *account)*

Modification of the Database – Updates

"Set the balance of all accounts at the Perryridge branch to 0."

update *account*
set *balance = 0*
where *branch-name = "Perryridge"*

"Set the balance of account A-325 to 0, and also change the branch name to "Mianus."

update *account*
set *balance = 0, branch-name = "Mianus"*
where *account-number* = "A-325"

Modification of the Database – Updates

"Increase all accounts with balances over $10,000 by 6%, all other accounts by 5%."

Option #1:

update *account*
set *balance* = *balance* ∗ 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* ∗ 1.05
where *balance* <= 10000

Modification of the Database – Updates

"Increase all accounts with balances over $10,000 by 6%, all other accounts by 5%."

Option #2:

```
update account
set balance =  case
                   when balance <= 10000 then balance *1.05
                   else balance * 1.06
               end
```

Transactions

- Some of the previous multi-query operations should be made transactions.
- A *transaction* is a sequence of SQL statements executed as a single unit.
- Example - Transferring money from one account to another:
  - deducting the money from one account
  - crediting the money to another account
- If one step succeeds and the other fails, the database is left in an inconsistent state.
- Therefore, either both steps should succeed, or both should fail (note: failing is better than corrupting).

Transaction - Syntax

- Transactions are started either implicitly or explicitly.
- Transactions are terminated by:
  - *commit* - makes all updates of the transaction permanent
  - *rollback* - undoes all updates performed by the transaction
- Commits and rollbacks can also be either implicit or explicit.
- Implicit transactions with implicit commits (no special syntax):
  - DDL statements
  - Individual SQL statements that execute successfully
- Implicit rollbacks:
  - System failure

Transactions, Cont.

- Automatic commit can be turned off, allowing multi-statement transactions.
- Transactions are identified by some variant of:

```
begin transaction                // shuts off auto-commit
   …
end transaction                  // commits the transaction
```

- Within the transaction, partial work can be:
  - made permanent by using the **commit work** statement.
  - undone by using the **rollback work** statement.
- Transactions are, or rather, should be the *rule* for programmers, rather than the exception.

Which MYSQL Package should I Use?

- With which package should I code?

MySQLdb	A thin python wrapper around C module which implements API for MySQL database
mysqlclient	By far the fastest MySQL connector for CPython. Requires the mysql-connector-c C library to work.
PyMySQL	Pure Python MySQL client. Use if you can't use libmysqlclient
mysql-connector-python	MySQL connector developed by the MySQL group at Oracle

Recommended Python Adapters	
Database	**Adapter**
PostgreSQL	Psycopg
SQLite	sqlite3
Oracle	cx_oracle
MySql	mysql-connector

Commit/Rollback Python Example

```
import mysql.connector
try:
    # Connecting to the Database
    conn = mysql.connector.connect(
        host='localhost',
        database=studentdb',
        user='root',
    )
    conn.autocommit = False # turn off autocommit to enforce transaction
    cursor = conn.cursor()
    statement = "UPDATE student SET dept_name ='Comp Sci.' WHERE id ='901000000'"
    cursor.execute(statement)

    # commit changes to the database
    conn.commit()

    # update successful message
    print("Student Department updated")

except mysql.connector.Error as error:
    conn.rollback()
conn.close()
```

Commit/Rollback Python Example

```
import mysql.connector
try:
    # Connecting to the Database
    conn = mysql.connector.connect(
        host='localhost',
        database=studentdb',
        user='root',
        autocommit=True, #The autocommit flag may go here as well

    )
    conn.start_transaction()
    cursor = conn.cursor()
    # these two INSERT statements are executed as a single unit
    sql1 = """ insert into employees(name, salary) value('Jason', 75000) """
    sql2 = """ insert into employees(name, salary) value('Devon', 87000) """
    cursor.execute(sql1)
    cursor.execute(sql2)

    # commit changes to the database
    conn.commit()

    # display successful message
    print("Transaction committed")

except mysql.connector.Error as error:
    conn.rollback()
conn.close()
```

Joined Relations

- Join operations take two relations and return another as a result.
- Specialized join operations are typically used as subquery expressions.
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
  - **natural**
  - **using** (A_1, A_2, ..., A_n) // equi-join
  - **on** <predicate> // theta-join
- Join type – defines how non-matching tuples (based on the join condition) in each relation are treated.
  - **inner join**
  - **left outer join**
  - **right outer join**
  - **full outer join**

Joined Relations – Datasets for Examples

- Relation *loan*

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

- Note that borrower information is missing for L-260 and loan information missing for L-155.

Joined Relations – Examples

loan **inner join** *borrower*
on *loan.loan-number = borrower.loan-number*

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

loan **left outer join** *borrower*
on *loan.loan-number = borrower.loan-number*

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Joined Relations – Examples

loan **natural inner join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan **natural right outer join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

loan

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

Joined Relations – Examples

loan **full outer join** *borrower* **using** *(loan-number)*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	null	null	Hayes

"Find all customers who have either an account or a loan (but not both) at the bank."

select *customer-name*
from (*depositor* **natural full outer join** *borrower*)
where *account-number* **is** *null* **or** *loan-number* **is** *null*

Joined Relations – Examples

"Find all customers who have an account at all branches located in Brooklyn."

```
select customer_name
from customer
where customer_name not in (
    select R.customer_name
    from depositor R
    inner join branch T on T.branch_city = 'brooklyn'
    left join account S
        on R.account_number = S.account_number
        and T.branch_name = S.branch_name
    where a.account_number is null )
```

Schema

branch (branch-name, branch-city, assets)
customer (customer-name, customer-street, customer-city)
account (account-number, branch-name, balance)
loan (loan-number, branch-name, amount)
depositor (customer-name, account-number)
borrower (customer-name, loan-number)

Exporting Data - Bonus

Exporting a Table

```
SELECT * INTO OUTFILE 'data.txt'
FIELDS TERMINATED BY ','
FROM table2;
```

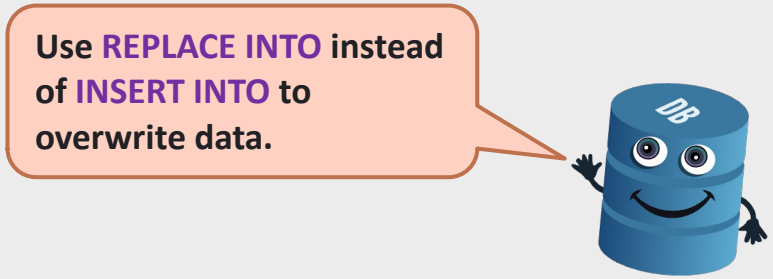
Dumping/Backing up database from the MySQL Shell

```
mysqldump -u [username] -p [database-you-want-to-dump] > [file-path]

mysqldump -u root -p mydatabase > /home/myuser/database-dump.sql
```

Copying Tables Between databases (on the same server)

```
INSERT INTO DB1.TABLE1 SELECT * FROM DB2.TABLE1;
```



Merging Databases - Bonus

Exporting a Table

```
mysqldump -u root -p --no-create-info db1 > db1.sql
mysqldump -u root -p --no-create-info db2 > db2.sql
mysqldump -u root -p --no-data db1 > schema.sql
```

Create a new DB

```
mysql -uroot -p –Ddb3 < schema.sql
mysql -uroot -p -Ddb3 < db1.sql
mysql -uroot -p -Ddb3 < db2.sql
```

Modification of the Database – Insertion

1. *What would the DDL look like for the following where customer ID is to be auto-incremented?*
2. *How would you insert a customer record into the customer table of the following schema where customer ID is to be auto-incremented?*

branch(branch_name, branch_city, assets)

customer (ID, customer_name, customer_street, customer_city)

loan (loan_number, branch_name, amount)

borrower (ID, loan_number)

account (account_number, branch_name, balance)

depositor (ID, account_number)

Customer DDL

create table *customer (*
ID ***int auto_increment****,*
customer-name ***varchar(20)****,*
customer-street ***varchar(25)****,*
customer-city ***varchar(25)****,*
primary key(ID)
)

Modification of the Database – Programmatic Insertion Algorithm 1

Difficulty level: 

This algorithm requires the user to simply create insert statements as strings based on the format for each table. Then issue the insert statement using code.

Create a list of tables based on integrity constraints (i.e., foreign key order)

Example: tables = [“classroom", "department", "course", ...]

create insert statement templates based on each table structure:

Examples:

insertStatement1 = "insert into table1 values(%s, %s, …);” //known as a parameterized query

insertStatement2 = "insert into table2 values(%s, %s, …);”

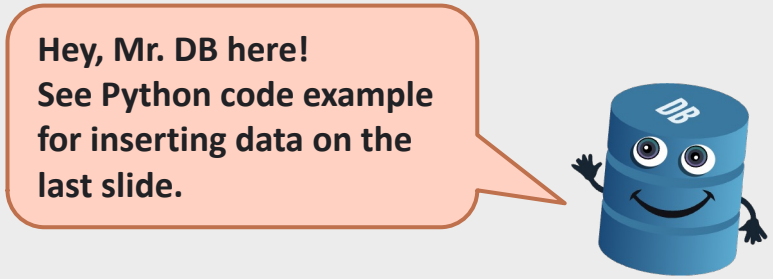
For each csvFile:

csvFile = open(csvFile)

for each **line** in the csv file

create tuple from **line** as record //Python requires tuple whereas Java PreparedStatement

insert record into database table based on insertStatement template for this **csvFile**



Modification of the Database – Programmatic Insertion Algorithm 2

Difficulty level: 

This algorithm creates a DDL file based on a set of CSV Files. The DDL file will have statements of the form:

insert into ***table name*** values('value$_1$', 'value$_2$', … 'value$_n$');

Create a list of tables based on integrity constraints (i.e., foreign key order)

Example: tables = ["classroom", "department", "course", ...]

Determine a **file name** for the output DDL file

fp = open DDL file for writing

for each **table** in the list:

    let ***csvFile*** = open a csv file corresponding to the ***table name*** (eg. "classroom.csv")

        build a string from each record in the file as **record_str** where each attribute except null is enclosed in single quotes

        set insertStatement as "insert into ***table name*** values(***record_str***);"

        write insertStatement to **fp**

Batch execute the DDL file to insert all data into database

Modification of the Database – Programmatic Insertion Algorithm 3

Difficulty level: 

This algorithm reads one or more CSV files, returns records corresponding to each CSV file as a list.
The programmer will loop through each list and write the data to the database.

Create a list of tables based on integrity constraints (i.e., foreign key order)
Example: tables = ["classroom", "department", "course", ...]

Let ***insertStatement*** = "insert into %s values(%s);" //known as a parameterized query
for each ***table*** in the list:
    let ***csvFile*** = open a csv file corresponding to the ***table name*** (eg. "classroom.csv")
    for ***record*** in ***csvFile***
        build a string from each record in the file as ***record_str*** where each attribute except null is enclosed in single quotes
        let **wildcard** = "%s" or "?" based on programming language
        let ***wildcards*** = **wildcard** * (number of attributes in the record)
        fill in the first %s in the ***insertStatement*** with the ***table name***
        fill in the second %s in the ***insertStatement*** with the value in the **wildcards** variable
        issue the insert statement to the database // Note that substituting the wildcard values (i.e., ?,? …) in Java
                                                    // will require some work based on the data type of each attribute.

Python Code to Insert a Record into a Database

```
import mysql.connector
try:
    # Connecting to the Database
    conn = mysql.connector.connect(host='localhost', database='bankdb', user='root')
    # Variables may come from anywhere, a file, user input, etc
    customer_name = ""
    customer_street = ""
    customer_city = ""

    cursor = conn.cursor()
    sql_command = """INSERT INTO customer(customer_name, customer_street, customer_city) VALUES(%s, %s, %s);"""
    record = (customer_name, customer_street, customer_city)  # a record is a tuple

    cursor.execute(sql_command, record)  # the execute method will fill in each record value into the sql_command string

    # commit changes to the database
    conn.commit()

    # update successful message
    print("Customer data added successfully")

except mysql.connector.Error as error:
    conn.rollback()
conn.close()
```

Java Code to Insert Records into a Database

```
import java.sql.*
import java.util.ArrayList;

public class SampleInsert {
    public void insertCustomerData(ArrayList<String[]> records){
        try {
            Connection conn = DriverManager.getConnection(DB, USER, PASS);
            String customerQuery = "INSERT INTO customer(customer_name, customer_street, customer_city) VALUES(?,?,?)";
            for (String[] record : records) {
                PreparedStatement preparedStatement = conn.prepareStatement(customerQuery);
                preparedStatement.setString(1, record[1]);
                preparedStatement.setString(2, record[2]);
                preparedStatement.setString(3, record[3]);
                preparedStatement.execute();
            }
            System.out.println("Inserted " + records.size() + " record(s) into customer table");
            conn.close();

        }catch (SQLException e) {
            System.err.println("An error occurred while trying to write data to the database");
            System.err.println(e.getMessage());
        }
    }
}
```

End of Chapter