



## Lecture 5 of 42

# SQL Query Syntax and Outer Joins

Friday, 01 September 2006

William H. Hsu

Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:

Sections 3.6 – 3.10, p. 91 - 110, Silberschatz *et al.*, 5<sup>th</sup> edition



## Division Operation: Review

- Notation:  $r \div s$
- Suited to queries that include the phrase “for all”.
- Let  $r$  and  $s$  be relations on schemas  $R$  and  $S$  respectively where

\*  $R = (A_1, \dots, A_m, B_1, \dots, B_n)$

\*  $S = (B_1, \dots, B_n)$

The result of  $r \div s$  is a relation on schema

$R - S = (A_1, \dots, A_m)$

$$r \div s = \{ t \mid t \in \prod_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$$

Where  $tu$  means the concatenation of tuples  $t$  and  $u$  to produce a single tuple





## Another Division Example: Review

- Relations  $r, s$ :

| A        | B | C        | D | E |
|----------|---|----------|---|---|
| $\alpha$ | a | $\alpha$ | a | 1 |
| $\alpha$ | a | $\gamma$ | a | 1 |
| $\alpha$ | a | $\gamma$ | b | 1 |
| $\beta$  | a | $\gamma$ | a | 1 |
| $\beta$  | a | $\gamma$ | b | 3 |
| $\gamma$ | a | $\gamma$ | a | 1 |
| $\gamma$ | a | $\gamma$ | b | 1 |
| $\gamma$ | a | $\beta$  | b | 1 |

$r$

| D | E |
|---|---|
| a | 1 |
| b | 1 |

$s$

- $r \div s$ :

| A        | B | C        |
|----------|---|----------|
| $\alpha$ | a | $\gamma$ |
| $\gamma$ | a | $\gamma$ |



## Bank Example Queries: Review

- Find all customers who have an account from at least the "Downtown" and the Uptown" branches.
- Query 1

$$\Pi_{customer\_name}(\sigma_{branch\_name = \text{"Downtown"}}(depositor \bowtie account)) \cap \Pi_{customer\_name}(\sigma_{branch\_name = \text{"Uptown"}}(depositor \bowtie account))$$

- Query 2

$$\Pi_{customer\_name, branch\_name}(depositor \bowtie account) \div \rho_{temp(branch\_name)}(\{\text{"Downtown"}, \text{"Uptown"}\})$$

Note that Query 2 uses a constant relation.



## Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - \* *null* signifies that the value is unknown or does not exist
  - \* All comparisons involving *null* are (roughly speaking) **false** by definition.
    - ⇒ We shall study precise meaning of comparisons with nulls later



## Outer Join – Example

- Relation *loan*

| <i>loan_number</i> | <i>branch_name</i> | <i>amount</i> |
|--------------------|--------------------|---------------|
| L-170              | Downtown           | 3000          |
| L-230              | Redwood            | 4000          |
| L-260              | Perryridge         | 1700          |

- Relation *borrower*

| <i>customer_name</i> | <i>loan_number</i> |
|----------------------|--------------------|
| Jones                | L-170              |
| Smith                | L-230              |
| Hayes                | L-155              |







## Joined Relations – Examples

- *loan natural inner join borrower*

$r \bowtie s \equiv T_{\text{unique}}(\sigma_{\text{match}}(rs))$

| <i>loan_number</i> | <i>branch_name</i> | <i>amount</i> | <i>customer_name</i> | <i>loan_number</i> |
|--------------------|--------------------|---------------|----------------------|--------------------|
| L-170              | Downtown           | 3000          | Jones                | L-170              |
| L-230              | Redwood            | 4000          | Smith                | L-230              |

- *loan natural right outer join borrower*

| <i>loan_number</i> | <i>branch_name</i> | <i>amount</i> | <i>customer_name</i> |
|--------------------|--------------------|---------------|----------------------|
| L-170              | Downtown           | 3000          | Jones                |
| L-230              | Redwood            | 4000          | Smith                |
| L-155              | <i>null</i>        | <i>null</i>   | Hayes                |



## Joined Relations – Examples

- *loan full outer join borrower using (loan\_number)*

| <i>loan_number</i> | <i>branch_name</i> | <i>amount</i> | <i>customer_name</i> |
|--------------------|--------------------|---------------|----------------------|
| L-170              | Downtown           | 3000          | Jones                |
| L-230              | Redwood            | 4000          | Smith                |
| L-260              | Perryridge         | 1700          | <i>null</i>          |
| L-155              | <i>null</i>        | <i>null</i>   | 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
```



## Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where  $r$  is a relation and  $E$  is a relational algebra query.



## Deletion Examples

- Delete all account records in the Perryridge branch.  
 $account \leftarrow account - \sigma_{branch\_name = "Perryridge"}(account)$

- Delete all loan records with amount in the range of 0 to 50

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{branch\_city = "Needham"}(account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{branch\_name, account\_number, balance}(r_1)$$

$$r_3 \leftarrow \Pi_{customer\_name, account\_number}(r_2 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$





## Insertion

- To insert data into a relation, we either:
  - \* specify a tuple to be inserted
  - \* write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:
 
$$r \leftarrow r \cup E$$
 where  $r$  is a relation and  $E$  is a relational algebra expression.
- The insertion of a single tuple is expressed by letting  $E$  be a constant relation containing one tuple.



## Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$$account \leftarrow account \cup \{("Perryridge", A-973, 1200)\}$$

$$depositor \leftarrow depositor \cup \{("Smith", A-973)\}$$

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

$$r_1 \leftarrow (\sigma_{branch\_name = "Perryridge"}(borrower \bowtie loan))$$

$$account \leftarrow account \cup \Pi_{branch\_name, loan\_number, 200}(r_1)$$

$$depositor \leftarrow depositor \cup \Pi_{customer\_name, loan\_number}(r_1)$$




## Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_i, \dots} (r)$$

- Each  $F_i$  is either
  - \* the  $i^{\text{th}}$  attribute of  $r$ , if the  $i^{\text{th}}$  attribute is not updated, or,
  - \* if the attribute is to be updated  $F_i$  is an expression, involving only constants and the attributes of  $r$ , which gives the new value for the attribute



## Update Examples

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.05} (account)$$

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.06} (\sigma_{BAL > 10000} (account)) \cup \Pi_{account\_number, branch\_name, balance * 1.05} (\sigma_{BAL \leq 10000} (account))$$





## Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations\*\*



## History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - \* SQL-86
  - \* SQL-89
  - \* SQL-92
  - \* SQL:1999 (language name became Y2K compliant!)
  - \* SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - \* Not all examples here may work on your particular system.





## Data Definition Language

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.



## Domain Types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p,d*)**. Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.





## Create Table Construct

- An SQL relation is defined using the **create table** command:

```
create table  $r(A_1 D_1, A_2 D_2, \dots, A_n D_n,$   
              (integrity-constraint1),  
              ...,  
              (integrity-constraintk))
```

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

- Example:

```
create table branch  
  (branch_name char(15) not null,  
   branch_city  char(30),  
   assets       integer)
```



## Integrity Constraints in Create Table

- **not null**
- **primary key** ( $A_1, \dots, A_n$ )

Example: Declare *branch\_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch  
  (branch_name char(15),  
   branch_city  char(30),  
   assets       integer,  
   primary key (branch_name))
```

**primary key** declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89





## Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation:

**alter table  $r$  add  $A D$**

where  $A$  is the name of the attribute to be added to relation  $r$  and  $D$  is the domain of  $A$ .

- \* All tuples in the relation are assigned *null* as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation:

**alter table  $r$  drop  $A$**

where  $A$  is the name of an attribute of relation  $r$

- \* Dropping of attributes not supported by many databases



## Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

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

- \*  $A_j$  represents an attribute
- \*  $R_i$  represents a relation
- \*  $P$  is a predicate.

- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.





## The select Clause

- The **select** clause list the attributes desired in the result of a query
  - \* corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

```
select branch_name  
from loan
```

- In the relational algebra, the query would be:

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

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - \* Some people use upper case wherever we use bold font.



## The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the names of all branches in the *loan* relations, and remove duplicates

```
select distinct branch_name  
from loan
```

- The keyword **all** specifies that duplicates not be removed.

```
select all branch_name  
from loan
```





## Example Query

- Find 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: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.



## Set Comparison

- 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'
```

- Same query using > some clause

```
select branch_name
from branch
where assets > some
(select assets
from branch
where branch_city = 'Brooklyn')
```





## Definition of Some Clause

- $F \langle \text{comp} \rangle \text{ some } r \Leftrightarrow \exists t \in r \text{ such that } (F \langle \text{comp} \rangle t)$   
Where  $\langle \text{comp} \rangle$  can be:  $<$ ,  $\leq$ ,  $>$ ,  $=$ ,  $\neq$

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$  (read: 5 < some tuple in the relation)

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$  (since  $0 \neq 5$ )

$(= \text{ some}) \equiv \text{in}$

However,  $(\neq \text{ some}) \neq \text{not in}$



## Example Query

- 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')

```





## Definition of all Clause

- $F \langle \text{comp} \rangle \text{all } r \Leftrightarrow \forall t \in r (F \langle \text{comp} \rangle t)$

$$(5 < \text{all } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \text{all } \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \text{all } \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \text{all } \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

$(\neq \text{all}) \equiv \text{not in}$   
 However,  $(= \text{all}) \neq \text{in}$



## Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists**  $r \Leftrightarrow r \neq \emptyset$
- **not exists**  $r \Leftrightarrow r = \emptyset$





## Example Query

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

```
select distinct S.customer_name
from depositor 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 ))
```

- Note that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = **all** and its variants



## Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
select T.customer_name
from depositor as T
where 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' )
```





## Example Query

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

```
select distinct T.customer_name
from depositor as 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')
```



## Derived Relations

- SQL allows a subquery expression to be used 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 branch_avg ( branch_name, avg_balance )
where avg_balance > 1200
```

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch\_avg* in the **from** clause, and the attributes of *branch\_avg* can be used directly in the **where** clause.





## With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all accounts with the maximum balance

```
with max_balance (value) as  
  select max (balance)  
  from account  
select account_number  
from account, max_balance  
where account.balance = max_balance.value
```



## Complex Query using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with branch_total (branch_name, value) as  
  select branch_name, sum (balance)  
  from account  
  group by branch_name  
with branch_total_avg (value) as  
  select avg (value)  
  from branch_total  
select branch_name  
from branch_total, branch_total_avg  
where branch_total.value >= branch_total_avg.value
```





## Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

```
delete from account  
where branch_name = 'Perryridge'
```

- Delete all accounts at every branch located in the city 'Needham'.

```
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 )
```

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
  1. First, compute **avg** balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)





## Modification of the Database – Insertion

- Add a new tuple to *account*

```
insert into account  
values ('A-9732', 'Perryridge', 1200)
```

or equivalently

```
insert into account (branch_name, balance, account_number)  
values ('Perryridge', 1200, 'A-9732')
```

- Add a new tuple to *account* with *balance* set to null

```
insert into account  
values ('A-777', 'Perryridge', null)
```



## 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 savings 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 **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like **insert into table1 select \* from table1** would cause problems)





## Modification of the Database – Updates

- Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.
  - \* Write two **update** statements:

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

update account
set balance = balance * 1.05
where balance ≤ 10000
```
  - \* The order is important
  - \* Can be done better using the **case** statement (next slide)



## Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

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





## Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, loan_number
   from borrower, loan
   where borrower.loan_number = loan.loan_number )
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

