

DATABASE TECHNOLOGY - 1DL124

Summer 2007

An introductory course on database systems

<http://user.it.uu.se/~udbl/dbt-sommar2007/>

alt. <http://www.it.uu.se/edu/course/homepage/dbdesign/st07/>

Kjell Orsborn

Uppsala Database Laboratory

Department of Information Technology, Uppsala University,
Uppsala, Sweden



UPPSALA
UNIVERSITET

Introduction to SQL

Elmasri/Navathe ch 8
Padron-McCarthy/Risch ch 7,8,9

Kjell Orsborn

Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

The SQL database language

- **SQL - (Structured Query Language)**
- Was first developed by IBM in the early 70's at their San Jose Research Lab. It was called Sequel and was implemented as part of their System R project.
- Current version of the ISO/ANSI SQL standard is SQL-99 (the earlier SQL-92 is a subset the standard).
- SQL has become the main language in commercial RDBMS.

Parts of the SQL language

- SQL include several subparts
 - DDL
 - Interactive DML
 - Queries: SELECT
 - Updates: INSERT, DELETE, UPDATE
 - Embedded DML
 - View definition
 - Authorizaton
 - Integrity
 - Transaction control
- SQL (E/N chapter 8)
 - Basic Structure
 - Set Operations
 - Aggregate Functions
 - Null Values
 - Nested Subqueries
 - Derived Relations
 - Views
 - Modification of the Database
 - Joined Relations
 - Data Definition
 - Schema Evolution
 - Additional SQL Features



Basic 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_i 's represent attributes
 - r_i 's represent relations
 - P is a predicate.
- This is equivalent to the relational algebra expression:
$$\Pi_{A_1, A_2, \dots, A_n} (\sigma_P (r_1 \times r_2 \times \dots \times r_m))$$
- The result of an SQL query is a relation.



Banking example revisited

- Again we use the bank schema in subsequent examples
- *branch (branch_name,branch_city,assets)*
- *customer (customer_name,customer_street,customer_city)*
- *account (branch_name,account_number,balance)*
- *loan (branch_name,loan_number,amount)*
- *depositor (customer_name,account_number)*
- *borrower (customer_name,loan_number)*

The select clause

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
 - E.g. find the names of all branches in the *loan* relation:
select branch_name from loan
- In “pure” relational algebra syntax, this query would be: $\Pi_{branch_name}(loan)$
- An asterisk (*) in the select clause denotes “all attributes”:
select * from loan
- SQL allows duplicates in relations as well as in query results. To force the elimination of duplicates, insert the keyword **distinct** after select.
 - E.g. find the names of all branches in the loan relation, and remove duplicates:
select distinct branch_name from loan
- The keyword **all** specifies that duplicates will not be removed:
select all branch_name from loan
- The select clause can also contain arithmetic expressions involving the operators, +, -, *, and /, operating on constants or attributes of tuples.
 - E.g. the following query returns the loan relation with the amount attribute multiplied by 100:
select branch_name, loan_number, amount * 100 from loan



The where clause

- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the *from* clause.
 - Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than \$1200:

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

- SQL uses the logical connectives **and**, **or**, (and **not**). It allows the use of arithmetic expressions as operands to the comparison operators.
- SQL includes a **between** comparison operator in order to simplify *where* clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.
 - Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)

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



The from clause

- The **from** clause corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned when evaluating the whole *select* expression.

- Find the Cartesian product borrower \times loan:

select * from *borrower, loan*

- Find the name and loan number of all customers having a loan at the Perryridge branch:

select distinct *customer_name, borrower.loan_number*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number and*
branch_name = "Perryridge"

The rename operation

- The SQL mechanism for renaming relations and attributes is accomplished through the **as** clause:

old-name **as** new-name

- Find the name and loan number of all customers having a loan at the Perryridge branch; replace the column name `loan_number` with the name `lid`.

```
select distinct customer_name, borrower.loan_number as lid  
from borrower, loan  
where borrower.loan_number = loan.loan_number and  
       branch_name = "Perryridge"
```

Tuple variables

- **Tuple variables** are defined in the *from* clause via the use of the *as* clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
select distinct customer_name, T.loan_number  
from borrower as T, loan as S  
where T.loan_number = S.loan_number
```

- 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"
```

String operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring “Main”:

```
select customer_name  
from customer  
where customer_street like “%Main%”
```

- Match the name “Main%”:
like “Main\%” **escape** “\”



Ordering the display of tuples

- List in alphabetic order the names of all customers having a loan at Perryridge branch:

```
select distinct customer_name  
from borrower, loan  
where borrower.loan_number = loan.loan_number and  
       branch_name = "Perryridge"  
order by customer_name asc (desc)
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
- SQL must perform a sort to fulfill an **order by** request. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.



Set operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations \cup , \cap , and $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset (sets with duplicates) versions **union all**, **intersect all** and **except all**.
- Suppose 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



Set operations cont.

- 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*)



Aggregate functions

- These functions operate on the multiset of values of a column of a relation, and return a value

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

- Find the average account balance at the Perryridge branch.

```
select avg (balance)  
from account  
where branch_name = "Perryridge"
```

- Find the number of tuples in the customer relation.

```
select count (*)  
from customer
```

- Find the number of depositors in the bank

```
select count (distinct customer_name)  
from depositor
```



Aggregate functions - group by

- 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
```

- Note: Attributes in select clause outside of aggregate functions **must appear** in **group by** list.

Aggregate functions - having

- Find the names of all branches where the average account balance is more than \$1,200

```
select branch_name , avg (balance)  
from account  
group by branch_name  
having avg (balance) > 1200
```

- Note: predicates in the **having** clause are applied **after** the formation of groups

Null values

- It is possible for tuples to have a **null value**, denoted by *null*, for some of their attributes; null signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving null is *null* and comparisons involving null return *unknown*:
 - (*true or unknown*) = *true*,
 - (*false or unknown*) = *unknown*,
 - (*unknown or unknown*) = *unknown*,
 - (*true and unknown*) = *unknown*,
 - (*false and unknown*) = *false*,
 - (*unknown and unknown*) = *unknown*
- Result of the **where** clause predicate is treated as *false* if it evaluates to *unknown* and “P is unknown” evaluates to *true* if predicate P evaluates to *unknown*.
- Find all loan numbers which appear in the loan relation with null values for amount:
select loan_number from loan where amount is null
- Find the total all loan amounts:
select sum (amount) from loan
- The statement above ignores null amounts; result is null if there is no non-null amount. All aggregate operations except **count(*)** ignore tuples with *null* values on the aggregated attributes.



Nested subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a select-from-where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Set membership

- $F \text{ in } r \Leftrightarrow \exists t \in r \wedge (t = F)$

$$(5 \text{ in } \begin{array}{|c|} \hline 0 \\ 4 \\ 5 \\ \hline \end{array}) = \text{true}$$

$$(5 \text{ in } \begin{array}{|c|} \hline 0 \\ 4 \\ 6 \\ \hline \end{array}) = \text{false}$$

$$(5 \text{ not in } \begin{array}{|c|} \hline 0 \\ 4 \\ 6 \\ \hline \end{array}) = \text{true}$$

Example query

- Find all customers who have both an account and a loan at bank.

```
select distinct customer_name  
from borrower  
where customer_name in (select customer_name  
                                from depositor)
```

- Find all customers who have a loan at the bank but do not have an account at the bank.

```
select distinct customer_name  
from borrower  
where customer_name not in (select customer_name  
                                from depositor)
```



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)
```

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"
```


The some clause

- $F \text{ <comp> some } r \Leftrightarrow \exists t (t \in r \wedge [F = \text{<comp>} t])$

where <comp> can be: <, ≤, >, ≥, <>, =

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

$(5 \text{ < some } \begin{bmatrix} 0 \\ 5 \end{bmatrix}) = \text{false}$

$(5 = \text{some } \begin{bmatrix} 0 \\ 5 \end{bmatrix}) = \text{true}$

$(5 \text{ <> some } \begin{bmatrix} 0 \\ 5 \end{bmatrix}) = \text{true}$ (since $0 \neq 5$)

- Also $(= \text{some}) \equiv \text{in}$, but $(\text{<> some}) \neq \text{not in}$



Example query

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

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

The all clause

- $F \text{ <comp> all } r \Leftrightarrow \forall t (t \in r \wedge [F = \text{<comp>} t])$

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

$$(5 \text{ < all } \begin{array}{|c|} \hline 6 \\ \hline 9 \\ \hline \end{array}) = \text{true}$$

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

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

- Note that $(\text{<=> all}) \equiv \text{not in}$, but $(\text{= all}) \neq \text{in}$



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")
```

Test for empty relations

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



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))
```

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 only 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")
```

Another example query

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

```
select distinct T.customer_name
from depositor 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

- 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 we do not need to use the having clause, since we compute in the from clause the temporary relation result, and the attributes of result can be used directly in the where clause.



Views

- Provides a mechanism to hide certain data from the view of certain users. To create a view we use the command:

create view *viewname* as <query expression>

where: <query expression> is any legal expression.

- Create a view consisting of branches and their customers:

```
create view all_customer as
    (select branch_name, customer_name
from depositor, account
where depositor.account_number =
        account.account_number)

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

- Find all customers of the Perryridge branch:

```
select customer_name
from all_customer
where branch_name = "Perryridge"
```



Modification of the database

- INSERT - insert new tuples into a table
- UPDATE - update qualified column values in tables
- DELETE - removes qualified tuples from table

Modifying the database – insertion

- Syntax: **insert into** *table* [(*column_list*)] **values** (*value_list*)
insert into *table* [(*column_list*)] **select** ... **from** ... **where** ...
- Add a new tuple to account:
insert into *account* **values** ("Perryridge", A-9732, 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** ("Perryridge", A-777, null)

Modifying 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 *branch_name, loan_number, 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*



Modifying the database – updates

- Syntax: **update** *table* **set** *column_name* = *value*, ... [**where** ...];
- 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.

Update of a view

- Create a view of all loan data in the loan relation, hiding the amount attribute:
create view *branch_loan* **as**
select *branch_name, loan_number*
from *loan*
- Add a new tuple to *branch_loan*:
insert into *branch_loan*
values ("Perryridge", "L-307")
- This insertion must be represented by inserting into the loan relation the tuple:
("Perryridge", "L-307", null)
- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.



Modifying the database – deletion

- Syntax: **delete from** *table* [**where** ...];
- Delete all account records at the Perryridge branch.
delete from *account* **where** *branch_name* = "Perryridge"
- Delete all accounts at every branch located in Needham.

[illegible]

```
delete from depositor
where account_number in
    (select account_number
     from branch, account
     where branch_city = "Needham" and
           branch.branch_name = account.branch_name)
```



Example query

- Delete the records 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)



Update of a view

- Create a view of all loan data in the loan relation, hiding the amount attribute:
create view *branch_loan* **as**
select *branch_name, loan_number*
from *loan*
- Add a new tuple to *branch_loan*:
insert into *branch_loan*
values ("Perryridge", "L-307")
- This insertion must be represented by inserting into the loan relation the tuple:
("Perryridge", "L-307", null)
- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.



Joined relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause.
- Join **condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join **type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- **Join types**
 - inner join
 - left outer join
 - right outer join
 - full outer join

Join conditions

natural

on <predicate>

using (A_1, A_2, \dots, A_n)



Joined relations

datasets for examples

- Relation loan

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

- Relation borrower

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

Joined relations – examples

loan inner join borrower on

loan.loan_number = borrower.loan_number

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

loan left outer join borrower on

loan.loan_number = borrower.loan_number

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



Joined relations – examples

loan natural inner join borrower

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

loan natural right outer join borrower

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



Joined relations – examples

loan natural full outer join borrower using (loan_number)

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



Data Definition and Schema Evolution

- Data definition include the specification of a database *schema* as well as descriptors for each element in the schema including, tables, constraints, views, domains, indexes, and other constructs such as authorization and physical storage structures.
- Example:

create schema company authorization kjell;

- SQL also uses the *catalog* concept to refer to a named collection of schemas



Creating relations in SQL

- Relations or tables are created using the **CREATE TABLE** command that specifies a relation by name, attributes and constraints.
- Attributes have a name, a data type (its value domain) and possible constraints.
- Key, entity integrity, and referential integrity constraints can also be specified.

Data types and domains

- **Data 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.
 - **date.** Dates, containing a (4 digit) year, month and date.
 - **time.** Time of day, in hours, minutes and seconds.
- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **User-defined domain types** can be explicitly defined in SQL-92 using a **create domain** statement that can be reused in defining relations:
 - **create domain** *person_name* **char(20) not null**



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  $i$  ,  
               ...,  
               integrity-constraintk  $i$  )
```

- 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 create table

- **not null**
- **primary key** (A_1, \dots, A_n)
- **check** (P), where P is a predicate
- E.g. 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) not null,  
   branch_city char(30),  
   assets integer,  
   primary key (branch_name),  
   check (assets >= 0))
```

- primary key declaration on an attribute automatically ensures not null in SQL-92



Schema evolution

- The **drop schema** command deletes all information about the database schema from the database.

drop schema *company* cascade (restrict);

- The **drop table** command deletes all information about the dropped relation from the database.

drop table *dependent* cascade (restrict);

Schema evolution cont...

- The **alter table** command is used to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the alter table command is

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

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

- 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*.



Additional SQL features

- Granting and revoking privileges for database security and authorization (ch 22)
- Embedded SQL and language bindings (C, C++, COBOL, Pascal)
- SQL transaction control commands to provide concurrency control and recovery (ch 19, 20, 21)
- A series of commands for physical database design (*storage definition language - SDL*)