DATABASTEKNIK - 1DL116

Spring 2004

An introductury course on database systems

http://user.it.uu.se/~udbl/dbt-vt2004/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden
Introduction to SQL

Elmasri/Navathe ch 8

Kjell Orsborn

Department of Information Technology
Uppsala University, Uppsala, Sweden
Contents
(Elmasri/Navathe ch. 8)

• SQL
  – 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
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

- DDL
- Interactive DML
  - Queries: SELECT
  - Updates: INSERT, DELETE, UPDATE
- Embedded DML
- View definition
- Authorization
- Integrity
- Transaction control
SELECT expressions in SQL

- The `select` expression is the most common way to express queries in SQL. Using the schema `customer(cname,balance)` this might look like:
  - “Find the names of customers that owe you money.”
    - `select cname from customer where balance < 0`
    - `select customers.cname from customer where customer.balance < 0`
    - `select cname as kundnamn from customer as c where c.balance < 0`
Basic structure

- SQL is based on set and relational operations with certain modifications and enhancements.
- A typical SQL query has the form:
  ```sql
  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., 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:
  ```sql
  \Pi_{A_1, A_2, ..., A_n} (\Pi_P (r_1 \bowtie r_2 \bowtie ... \bowtie 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.

- Find the names of all branches in the *loan* relation:
  
  ```sql
  select branch-name
  from loan
  ```

- In “pure” relational algebra syntax, this query would be:
  
  ```sql
  ∏branch-name(loan)
  ```

- An asterisk (*) in the select clause denotes “all attributes”
  
  ```sql
  select *
  from loan
  ```
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 relation, and remove duplicates:
  
  ```sql
  select distinct branch-name
  from loan
  ```

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

  ```sql
  select all branch-name
  from loan
  ```
The select clause (cont.)

• The select clause can contain arithmetic expressions involving the operators, +, -, *, and /, and operating on constants or attributes of tuples.

• The following query would return a relation which is the same as the loan relation, except that the attribute amount is 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:

  ```sql
  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.
The where clause (cont.)

- 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 $\leq 100,000$)

  ```sql
  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 ⨝ loan:
  ```sql
  select * from borrower, loan
  ```
- Find the name and loan number of all customers having a loan at the Perryridge branch:
  ```sql
  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 \texttt{as} clause:

\begin{verbatim}
old-name as new-name
\end{verbatim}

• 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.

\begin{verbatim}
select distinct customer-name, borrower.loan-number as lid
from borrower, loan
where borrower.loan-number = loan.loan-number and
   branch-name = “Perryridge”
\end{verbatim}
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.

```sql
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.

```sql
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”:

  ```sql
  select customer-name 
  from customer 
  where customer-street like "%Main%"
  ```

- Match the name “Main%”:

  ```sql
  like "Main\%" escape "\"
  ```
Ordering the display of tuples

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

\[
\text{select distinct customer-name} \\
\text{from borrower, loan} \\
\text{where borrower.loan-number = loan.loan-number and} \\
\text{branch-name = "Perryridge"} \\
\text{order by customer-name}
\]

• We may specify \text{desc} for descending order or \text{asc} for ascending order, for each attribute; ascending order is the default.

• SQL must perform a sort to fulfill an \text{order by} request. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.
Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- Multiset versions of some of the relational algebra operators – given multiset relations r1 and r2:
  - 1. If there are $c_1$ copies of tuple $t_1$ in $r_1$, and $t_1$ satisfies selection $\Box q_1$, then there are $c_1$ copies of $t_1$ in $\Box q_1(r_1)$.
  - 2. For each copy of tuple $t_1$ in $r_1$, there is a copy of tuple $\Box_A(t_1)$ in $\Box_A(r_1)$, where $\Box_A(t_1)$ denotes the projection of the single tuple $t_1$.
  - 3. If there are $c_1$ copies of tuple $t_1$ in $r_1$ and $c_2$ copies of tuple $t_2$ in $r_2$, there are $c_1 \uplus c_2$ copies of the tuple $t_1.t_2$ in $r_1 \uplus r_2$. 
Duplicates cont.

• Suppose relations $r_1$ with schema $(A,B)$ and $r_2$ with schema $(C)$ are the following multisets:

$$r_1 = \{ (1,a),(2,a) \} \quad r_2 = \{ (2),(3),(3) \}$$

• Then $\square_B(r_1)$ would be $\{ (a),( a) \}$, while $\square_B(r_1) \quad r_2$ would be $\{ (a,2),( a,2),( a,3),( a,3),( a,3),( a,3) \}$

• SQL duplicate semantics:

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

• is equivalent to the multiset version of the expression:

$$\square A_1, A_2, ..., A_n (\square_P (r_1 \quad r_2 \quad ... \quad r_m))$$
Set operations

• The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, 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** $s$
  - $\max(0, m)$ in $r$ **except all** $s$
Set operations cont.

- Find all customers who have a loan, an account, or both:
  
  \[
  (\text{select } \text{customer-name from depositor}) \\
  \text{union} \\
  (\text{select } \text{customer-name from borrower})
  \]

- Find all customers who have both a loan and an account:
  
  \[
  (\text{select } \text{customer-name from depositor}) \\
  \text{intersect} \\
  (\text{select } \text{customer-name from borrower})
  \]

- Find all customers who have an account but no loan:
  
  \[
  (\text{select } \text{customer-name from depositor}) \\
  \text{except} \\
  (\text{select } \text{customer-name from borrower})
  \]
Aggregate functions

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

- \texttt{avg}: average value
- \texttt{min}: minimum value
- \texttt{max}: maximum value
- \texttt{sum}: sum of values
- \texttt{count}: number of values
Aggregate functions cont.

• Find the average account balance at the Perryridge branch.
  
  \[ \text{select} \ \text{avg} (balance) \]
  \[ \text{from} \ \text{account} \]
  \[ \text{where} \ \text{branch-name} = \text{"Perryridge"} \]

• Find the number of tuples in the customer relation.
  
  \[ \text{select} \ \text{count} (*) \]
  \[ \text{from} \ \text{customer} \]

• Find the number of depositors in the bank
  
  \[ \text{select} \ \text{count} (\text{distinct} \ \text{customer-name}) \]
  \[ \text{from} \ \text{depositor} \]
Aggregate functions - group by

• Find the number of depositors for each branch:

```sql
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

```sql
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*.
- 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 **where** clause predicate is treated as false if it evaluates to unknown
- “P is unknown” evaluates to *true* if predicate P evaluates to *unknown*. 
Null values cont.

- Find all loan numbers which appear in the loan relation with null values for amount:

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

- Total all loan amounts:

  `select sum(amount) from loan`

- Above statement 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.
SQL continued . . . (ch. 8)
(Elmasri/Navathe ch. 8)

• SQL continued . . .
  – Nested Subqueries
  – Derived Relations
  – Views
  – Modification of the Database
  – Joined Relations
  – Data Definition
  – Schema Evolution
  – Additional SQL Features
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

- $F \text{ in } r \ [t \ [r \ ] (t = F)$

$\begin{bmatrix} 0 & 4 & 5 \end{bmatrix}$

$(5 \text{ in } \begin{bmatrix} 0 & 4 & 5 \end{bmatrix}) = \text{true}$

$\begin{bmatrix} 0 & 4 & 6 \end{bmatrix}$

$(5 \text{ in } \begin{bmatrix} 0 & 4 & 6 \end{bmatrix}) = \text{false}$

$\begin{bmatrix} 0 & 4 & 6 \end{bmatrix}$

$(5 \text{ not in } \begin{bmatrix} 0 & 4 & 6 \end{bmatrix}) = \text{true}$
Example query

• Find all customers who have both an account and a loan at bank.
  
  \[
  \text{select distinct customer-name} \\
  \text{from borrower} \\
  \text{where customer-name in (select customer-name} \\
  \text{from depositor)}
  \]

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

  \[
  \text{select distinct customer-name} \\
  \text{from borrower} \\
  \text{where customer-name not in (select customer-name} \\
  \text{from depositor)}
  \]
Example query

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

```sql
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

- $F <\text{comp}> \text{some} r \sqsubseteq t$ ($t \sqsubseteq r \sqsubseteq [F = <\text{comp}> t]$)
  where $<\text{comp}>$ can be: $<, \leq, >, \geq, \langle\rangle, =$

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

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

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

- $(5 \langle\rangle \text{some} \begin{array}{c} 0 \\ 5 \end{array}) = \text{true}$ (since $0 \neq 5$)

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

- \( F \text{ <comp> all } r \quad t \) \( (t \quad r \quad [F = \text{ <comp> } t]) \)

\[
(5 < \text{ all } \begin{pmatrix} 0 \\ 5 \\ 6 \end{pmatrix}) = \text{ false}
\]

\[
(5 < \text{ all } \begin{pmatrix} 6 \\ 9 \end{pmatrix}) = \text{ true}
\]

\[
(5 = \text{ all } \begin{pmatrix} 4 \\ 5 \end{pmatrix}) = \text{ false}
\]

\[
(5 \Leftrightarrow \text{ all } \begin{pmatrix} 4 \\ 6 \end{pmatrix}) = \text{ true} \text{ (since } 5 \neq 4, 6)\]

- Note that \((\Leftrightarrow \text{ all}) \equiv \text{ not in, but } (= \text{ all}) \neq \text{ in}\)
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.

- exists $r \neq \emptyset$

- not exists $r = \emptyset$
Example query

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

```sql
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.

```sql
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.

```sql
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.

```sql
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

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

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

where:

– `<query expression>` is any legal expression
– the view name is represented by `v`
Example queries

• A view consisting of branches and their customers

```sql
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

```sql
select customer-name
from all-customer
where branch-name = “Perryridge”
```
Modifying the database – deletion

• Delete all account records at the Perryridge branch.
  `delete from account where branch-name = "Perryridge"`

• Delete all accounts at every branch located in Needham.
  `delete from account where branch-name in (select branch-name`  
  `from branch`  
  `where branch-city = "Needham")`

  `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.

```sql
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)
Modifying the database – insertion

- Add a new tuple to account
  \[\text{insert into account values ("Perryridge", A-9732, 1200)}\]

- or equivalently
  \[\text{insert into account (branch-name, balance, account-number) values ("Perryridge", 1200, A-9732)}\]

- Add a new tuple to account with balance set to null
  \[\text{insert into account values ("Perryridge", A-777,null)}\]
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.

```sql
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

- 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.
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 \((A_1, A_2, ..., A_n)\)
Joined relations
datasets for examples

• Relation loan

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>L-260</td>
<td>1700</td>
</tr>
</tbody>
</table>

• Relation borrower

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>1-170</td>
</tr>
<tr>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>
### Joined relations – examples

**loan inner join borrower on**

\[
\text{loan.loan-number} = \text{borrower.loan-number}
\]

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>1-170</td>
<td>3000</td>
<td>Jones</td>
<td>1-170</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
</tbody>
</table>

**loan left outer join borrower on**

\[
\text{loan.loan-number} = \text{borrower.loan-number}
\]

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>1-170</td>
<td>3000</td>
<td>Jones</td>
<td>1-170</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Perryridge</td>
<td>L-260</td>
<td>1700</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
### Joined relations – examples

**loan natural inner join borrower**

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>1-170</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
<td>Smith</td>
</tr>
</tbody>
</table>

**loan natural right outer join borrower**

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>null</td>
<td>L-155</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>
Joined relations – examples

\textit{loan full outer join borrower using (loan-number)}

<table>
<thead>
<tr>
<th>branch-name</th>
<th>loan-number</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>Perryridge</td>
<td>L-260</td>
<td>1700</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>L-155</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>

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

\textbf{select} customer-name
\textbf{from} (depositor natural full outer join borrower)
\textbf{where} account-number is null or loan-number is null
Data Definition and Schema Evolution

- Data definition include the specification of a database \textit{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;

- SQL2 also uses the \textit{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

- **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.
Data types and domains cont…

- **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.
Data types and domains cont...

- User-defined domain types can be explicitly defined in SQL-92 using a `create domain` statement that can be reused in defining relations:

  ```sql
  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, ..., A_n D_n,
  integrity-constraint_1 i ,
  ..., 
  integrity-constraint_k 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** \((A_1,...,A_n)\)
- **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.

```sql
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.

  \[
  \text{drop schema } \textit{company} \text{ cascade (restrict);}
  \]

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

  \[
  \text{drop table } \textit{dependent} \text{ cascade (restrict);}
  \]
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

  \[
  \text{alter table } r \text{ add } A D
  \]

  where A is the name of the attribute to 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

  \[
  \text{alter table } r \text{ drop } A
  \]

  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)