DATABASE TECHNOLOGY - 1MB025

Fall 2005

An introductory course on database systems

http://user.it.uu.se/~udbl/dbt-ht2005/
alt. http://www.it.uu.se/edu/course/homepage/dbastekn/ht05/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden
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
  - Authorization
  - 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:

\[
\text{select } A_1, A_2, \ldots, A_n \\
\text{from } r_1, r_2, \ldots, r_m \\
\text{where } P
\]

  – \( A_i \)’s represent attributes
  – \( r_i \)’s represent relations
  – \( P \) is a predicate.

• This is equivalent to the relational algebra expression:

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

• 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: \( \Pi_{\text{branch-name}}(\text{loan}) \)
- 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:
    ```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.

• 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 \( \times \) 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:

$$\text{old-name as new-name}$$

• 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.

\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:

```sql
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 $\setminus$.

- 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.
  ```sql
  select avg (balance) from account where branch-name = "Perryridge"
  ```

- Find the number of tuples in the customer relation.
  ```sql
  select count (*) from customer
  ```

- Find the number of depositors in the bank
  ```sql
  select count (distinct customer-name) from 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

\[
\text{select} \ branch-name, \ \text{avg} (balance) \\
\text{from} \ account \\
\text{group by} \ branch-name \\
\text{having} \ \text{avg} (balance) > 1200
\]

• Note: predicates in the \textbf{having} clause are applied \textbf{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*:

\[
\begin{align*}
(true \ or \ unknown) &= true, \\
(false \ or \ unknown) &= unknown, \\
(unknown \ or \ unknown) &= unknown, \\
(true \ and \ unknown) &= unknown, \\
(false \ and \ unknown) &= false, \\
(unknown \ and \ unknown) &= unknown
\end{align*}
\]

– 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

• $F \text{ in } r \iff \exists t \in r \land (t = F)$

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

```sql
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 <comp> some r ⇔ ∃t (t ∈ r ∧ [F = <comp> t])
  where <comp> can be: <, ≤, >, ≥, <>, =

(5 < some 0 5 6) = true (read: 5 < some tuple in the relation)

(5 < some 0 5) = false

(5 = some 0 5) = true

(5 <> some 0 5) true (since 0 ≠ 5)

- Also (= some) ≡ in, but (<> some) ≠ not in
Example query

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

```sql
select branch-name
from branch
where assets > some (select assets from branch
where branch-city = "Brooklyn")
```
The all clause

- \( F <\text{comp}\> \text{all} \ r \iff \exists t \ (t \in r \land [F = <\text{comp}\> \ t]) \)

\[
\begin{array}{c|c}
5 & 0 \\
5 & 5 \\
6 & 6 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c}
6 & 9 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c}
4 & 5 \\
\end{array}
\]

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

\[
\begin{array}{c|c|c}
4 & 6 \\
\end{array}
\]

\( (5 \not< \text{all} \begin{array}{c|c|c}
4 & 6 \\
\end{array}) = \text{true} \) (since \( 5 \neq 4, 6 \))

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

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

\[
\text{select } \text{branch-name} \\
\text{from } \text{branch} \\
\text{where } \text{assets} > \text{all (select } \text{assets from } \text{branch} \\
\hspace{1cm} \text{where } \text{branch-city} = \text{"Brooklyn"})}
\]
Test for empty relations

- The `exists` construct returns the value `true` if the argument subquery is nonempty.

- `exists r ⇔ r ≠ ∅`
- `not exists r ⇔ r = ∅`
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.

```
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

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

```sql
create view viewname as <query expression>
where: <query expression> is any legal expression.
```

• Create 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.
  \texttt{delete from account where branch-name = "Perryridge"}

- Delete all accounts at every branch located in Needham.
  \texttt{delete from account where branch-name in (select branch-name}
  \texttt{  from branch}
  \texttt{  where branch-city = "Needham")}

  \texttt{delete from depositor}
  \texttt{where account-number in}
  \texttt{(select account-number}
  \texttt{  from branch, account}
  \texttt{  where branch-city = "Needham" and}
  \texttt{  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
  
  ```sql
  insert into account
  values ("Perryridge", A-9732, 1200)
  ```

- or equivalently
  
  ```sql
  insert into account (branch-name, balance, account-number)
  values ("Perryridge", 1200, A-9732)
  ```

- Add a new tuple to account with balance set to null
  
  ```sql
  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:
  ```sql
  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 \((A_1, A_2, \ldots, A_n)\)
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

*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><em>null</em></td>
</tr>
<tr>
<td><em>null</em></td>
<td>L-155</td>
<td><em>null</em></td>
<td>Hayes</td>
</tr>
</tbody>
</table>

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

```sql
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:

  
  ```sql
  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:

  ```sql
  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:

  ```sql
  create table branch
  (branch-name char(15) not null,
   branch-city char(30),
   assets integer)
  ```
Integrity constraints create table

- **not null**
- **primary key** \((A_1, \ldots, A_n)\)
- **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.

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

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

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

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

  where A is the name of the attribute be added to relation r and 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*)