DATABASTEKNIK - 1DL116

Fall 2003

An introductury course on database systems

http://user.it.uu.se/~udbl/dbt-ht2003/

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 - S**tructured **Q**uery **L**anguage
- 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-92 (SQL-99 is the last standard released).
- SQL has become the main language in many commercial RDBMS.

Parts of the SQL language

- DDL
- Interactive DML
  - Queries: SELECT
  - Updates: INSERT, DELETE, UPDATE
- Embedded DML
- View definition
- Authorizaton
- 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:

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

  - A_i's represent attributes
  - r_i's represent relations
  - P is a predicate.
- This is equivalent to the relational algebra expression:

  $\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$
- 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.
- Find the names of all branches in the *loan* relation:

  select *branch-name*
  from *loan*

- In "pure" relational algebra syntax, this query would be:

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

- An asterisk (*) in the select clause denotes "all attributes"

  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:

  select distinct *branch-name*
  from *loan*

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

  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:

  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, ≥ \$90,000 and ≤\$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 × 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*

- 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.

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 $\sigma_{\square}$, then there are c_1 copies of t_1 in $\sigma_{\square}(r_1)$.
  - 2. For each copy of tuple t_1 in r_1, there is a copy of tuple $\pi_A(t_1)$ in $\pi_A(r_1)$, where $\pi_A(t_1)$ 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 $c_1 \times c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

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) } r_2 = { (2),(3),(3) }

- Then $\Pi_B(r_1)$ would be {(a),(a)}, while $\Pi_B(r_1) \times r_2$ would be

  { (a,2),(a,2),(a,3),(a,3),(a,3),(a,3) }

- SQL duplicate semantics:

  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:

  $\Pi_{A_1, A_2, ..., A_n}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$

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 *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

Aggregate functions cont.

- 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*.
- 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 **in** r ⇔ ∃*t* ∈ r : (*t* = F)

(5 **in** 

) = true

(5 **in** 

) = false

(5 **not in** 

) = 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 <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.

select *branch-name*
from *branch*
where *assets* **>** **some** (**select** *assets* **from** *branch*
where *branch-city* **=** “Brooklyn”)

The all clause

- F <comp> **all** r ⇔ ∀t (t ∈ r ⇒ [F = <comp> t])

(5 **<** **all** $\{0, 5, 6\}$) **= false**

(5 **<** **all** $\{6, 9\}$) **= true**

(5 **=** **all** $\{4, 5\}$) **= false**

(5 **<>** **all** $\{4, 6\}$) **= true** (since 5 ≠ 4, 6)

- Note that (**<>** **all**) ≡ **not in**, but (**=** **all**) ≠ **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 □ $r \neq \emptyset$
- not exists r □ $r = \emptyset$

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

- 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

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”

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.

  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

  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

- 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

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

- Relation borrower

customer-name	*loan-number*
Jones	1-170
Smith	L-230
Hayes	L-155

Joined relations – examples

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

branch-name	*loan-number*	*amount*	*customer-name*	*loan-number*
Downtown	1-170	3000	Jones	1-170
Redwood	L-230	4000	Smith	L-230

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

branch-name	*loan-number*	*amount*	*customer-name*	*loan-number*
Downtown	1-170	3000	Jones	1-170
Redwood	L-230	4000	Smith	L-230
Perryridge	L-260	1700	*null*	*null*

Joined relations – examples

loan **natural inner join** *borrower*

branch-name	*loan-number*	*amount*	*customer-name*
Downtown	1-170	3000	Jones
Redwood	L-230	4000	Smith

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

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

Joined relations – examples

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

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

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

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

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 contructs such as authorization and physical storage structures.
- Example:

```
create schema company authorization kjell;
```

- SQL2 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 realtion 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 explicitely 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 ,....,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** (A1,...,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 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 priviligies 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*)