Basic format of the `select` command

```
select [distinct] target_list
from tuple_variable_list
where qualification
[order by target_list_subset];
```

Simple query examples use this relational schema:

- `sailors(sid, sname, rating)`
- `boats(bid, bname, colour)`
- `reserve(sid, bid, date)`
SQL: target list

“*” is an abbreviation for all attributes in the `from` list

```
select * 
from sailors s
where order by s.rating
```

Each item in the target list can be as general as `attribute_name = expression`, where the expression is any arithmetic or string expression over indexed tuple variables and constants. It can also contain some built-in functions like `sqrt`, `sin`, `mod`, etc. as well as aggregates (coming up later)
SQL: target list expression example

With rating an integer from 1 to 10, this query gives a rating bonus to sailors who sailed two different boats on the same day.

```sql
SELECT s.sid, s.sname, rating=s.rating+2
FROM sailors s, reserve r, reserve r2
WHERE s.sid=r.sid AND s.sid=r2.sid AND r.date=r2.date AND r.bid != r2.bid
```

What’s wrong with the above?

- What happens if s.rating = 9 before this query?
- Domain constraints might take care of this, but we need to be careful
Qualifications: each item in a qualification (\textit{where} clause) can be as general as \textit{expression}=\textit{expression}

Example:
\begin{verbatim}
select name1 = s1.sname, name2 = s2.sname
from sailors s1, sailors s2
where 2*s1.rating = s2.rating-1
\end{verbatim}
Further elaboration:

tuple variables can be implicit if the system can figure out which relation each attribute belongs to
	table names can be used as tuple variables

Example: find names, ages, and departments of employees who are over 40 and work on the first floor.

```sql
select ename, age, emp.dname
from emp, dept
where age>40 and floor=1 and emp.dname=dept.dname
```
QL provides set operators: union, intersect, and minus

Example: find the names of employees who work in the toy department and make at most 60K

\[
\begin{align*}
\text{(select & ename} \\
\text{from & emp} \\
\text{where & dname=“toy”)} \\
\text{minus} \\
\text{(select & ename} \\
\text{from & emp} \\
\text{where & sal>60K)}
\end{align*}
\]
Note that it is usually possible to phrase a single query in multiple ways. The previous query could have also been written:

```
(SELECT ename
 FROM emp
 WHERE dname="toy")
INTERSECT
(SELECT ename
 FROM emp
 WHERE sal?60K)
```
or also (even simpler):

```sql
select ename
from emp
where dname="toy" and sal>60K
```

Writing a query in different ways will usually change how efficient the query is -- the above query is very likely to be faster than the example using `intersects`, and that one is likely to be faster than the one using `minus`. 
SQL also provides set operators: *contains* (a set being a superset of another) and *exists* (a set not being empty). Both return Boolean results, so may be negated (using *not*).
Example: find the names of employees who manage all the departments on the first floor.

```sql
select mgr
from dept d1
where (select d2.dname
    from dept d2
    where d1.mgr=d2.mgr)
contains
    (select dname
        from dept
        where floor=1)
```
QL allows nested queries using the keyword *in*

Example: find the names of employees who work on the first floor.

```sql
select ename
from emp
where dname in
  (select dname
   from dept
   where floor = 1)
```

The same query in flat form is

```sql
select dname
from emp, dept
where emp.dname=dept.dname and floor=1
```
The connective \textit{in} tests for set membership. Similar connectives are:
\begin{itemize}
  \item not in (set non membership)
  \item op any (op relationship with some tuple in the set)
  \item op all (op relationship with all tuples in the set)
\end{itemize}
where \textit{op} is one of (=, !=, <, >, <=, >=)

Example: find the names of employees who make more than everybody on the first floor.
\begin{verbatim}
select ename
from emp
where sal > all
  (select sal
   from emp, dept
   where emp.dname=dept.dname and floor = 1)
\end{verbatim}
SQL

coping of variables works exactly as in Pascal or C

Example: find the names of students who take a course from their advisor.

```sql
select sname
from student
where s# in
  (select s#
   from enroll
   where c# in
     (select c#
      from class
      where prof=student.advisor))
```
Recap: SQL

Four basic commands

- select
- insert
- delete
- update
SQL Insert

Insert command format:

\[\text{insert into } \text{relation\_name}\ \text{values}\ (\text{value\_list})\]

or

\[\text{insert into } \text{relation\_name}\ \text{select\_statement}\]

Semantics of insert

– format one: add the tuple corresponding to value\_list into relation\_name

– format two: execute the select statement, then add all the resulting tuples into relation\_name

Example:

insert into student values (1, “Carey”, “CS”, “Stonebraker”)
SQL Insert

Example: relation register(S#, name, paid)

which registered students are recorded. After the end of registration week, we execute:

```sql
insert into student
    select r.s#, r.name
    from register r
    where r.paid="yes"
```
Delete command format:

```
   delete relation_name where qualification
```

Semantics of delete: execute the corresponding select command:
```
   select full_target_list (or "*")
   from relation_name
   where qualification
```
and then remove the resulting tuples from relation_name
Example: with the following schema

student(s#, name, major, advisor)

enroll(s#, c#, grade)

course(c#, dept)

The following command expels CS majors who received a grade of less than 2.5 in a CS course:

```
delete student
where major="CS" and s# in

(select  s#
from enroll, course
where enroll.s#=student.s# and grade<2.5
and enroll.c#=course.c# and dept="CS")
```
SQL Update

Update format

update relation_name
set target_list
where qualification

Semantics of update: it is equivalent to executing:

– insert into del_temp
  select *
  from relation_name
  where qualification
SQL Update

Semantics of update (cont): … then executing

– insert into app_temp
  select ext_target_list
  from relation_name
  where qualification

Ext_target_list is identical to target_list in the original update command, but augmented with tuple_variable.attribute_name for all attributes of the range of tuple_variable that don’t appear in target_list.

– delete the tuples in del_temp from relation_name

– add the tuples in app_temp to relation_name
Example: give a 10% grade raise to every CS major in CS564

```sql
update enroll
set grade=1.1*grade
where c#="CS564" and s# in
(select s#
from student
where major="CS")
```
SQL Update

Which is equivalent to:

```
insert into del_temp
    select s#, c#, grade
from enroll
where c# = "CS564" and s# in
    (select s#
     from student
     where major="CS")
```

```
insert into app_temp
    select s#, c#, grade=1.1*grade
from enroll
where c# = "CS564" and s# in
    (select s#
     from student
     where major="CS")
```
SQL Aggregates

Aggregate functions are functions that take a collection of values as input and return a single value. SQL supports five built-in aggregate functions:

- average: `avg`
- minimum: `min`
- maximum: `max`
- total: `sum`
- cardinality: `count`

Using `distinct` to aggregate only unique values is often important with `avg`, `sum`, and `count`
SQL Aggregates

Example: find the number of students
select num_of_students = count(s#)
from student

why do we not need to use distinct in this example?

Example: find the number of employee records
select count(*)
from emp

if an employee appears more than once in the emp relation, for example if he had switched jobs or had two jobs, then this command would count that employee once for each record
SQL Aggregates

Qualified Aggregates:
Example: find the average age of employees in the toy department

select avg(age)
from emp
where dname="toy"
SQL: Group By clause

Group aggregates: groups of tuples are computed using the `group by` clause
- the attributes given in the clause are used to form groups
- tuples with the same value on all attributes in the clause are placed in one group

Example: in each department, find the minimum age of employees who make more than 50K
```sql
select dname, min(age)
from emp
where sal>50K
group by dname
```
Sometimes it is useful to state a condition that applies to groups in `group by` rather than to tuples. We do that in SQL with the `having` clause. SQL applies predicates of `having` after groups have been formed.

Example: find the average salary for employees under 30 for each department having more than 10 such employees

```sql
select dname, avg(sal)
from emp
where age<30
group by dname
having count(*)>10
```