Basic Standard SQL
(Structured Query Language)

1. Schema Creation/Modification: DDL (Data Definition Language) Features of SQL

Built-in Domain types

- **char(n)**: fixed length
- **varchar(n)**: variable length string upto n characters
- **int** (or **integer**): machine dependent,
- **smallint**: machine dependent,
- **numeric(p, d)**: fixed-point number (d of p to the right side of the decimal point)
- **real** and **double precision**: floating-point number and double-precision floating-point number (the precision is machine dependent)
- **float(n)**: floating-point number (user-defined precision: at least n digits)
- **date**: 4 year digits+2 month digits + 2 day digits
- **time**: 2 hour digits + 2 minute digits + 2 second digits
- **interval**: date1-date2 or time1-time2 Note, date1 (or time1) >= date2 (or time2). interval + (or -) time (or date) is time (or date)
Creating a domain type (rename a domain type)

e.g., `create domain` `fullname` `char(30)`

Create Relation Schema

`create table` `relation-name` (  
  A1 D1 <not-null>,  
  A2, D2 <not-null>, ...,  
  An Dn <not-null>,  
  <constraint1>,  
  <constraint2>, ...,  
  <constraintn>
)

A1, A2, ..., An are attribute names and D1, D2, ..., Dn are domain types.  
Note, the parameters in angle brackets are optional!  
Note, there are three types of constraints: **primary key** (Ai, Aj, ..., Al), **foreign key** (Ai, Aj, ..., Al) **references** `relation-name` <on delete cascade> <on update cascade>, and **check** (predicate)

Note, if an insertion, deletion, or update tuple incurs a failure of any constraint or not-null on the tuple --> DBMS reject the operation and an error is flagged.

E.g. `create table` `employee` (  
  e-num `integer` not-null,  
  e-name `fullname`,  
  e-age `integer`,  
  dept-num `integer`,  
  primary key (e-num),  
  foreign key (dept-num) references `department`  
  on update cascade  
  on delete cascade,  
  check (e-age <= 100)
Note, the available predicates in "check" is discussed in Section 3 (the "where" clause of SQL DML).

**Modify Relation Schema (Add or Delete an attribute)**

```sql
alter table relation-name add A D
alter table relation-name drop A
```

Note, "A" is an attribute and "D" is a domain type.

Note, when an attribute is added to a relation, all tuples in the relation are assigned "null" as the value for the new attribute.

We cannot give any constraint (or not-null) on the new attribute.

e.g.  
```sql
alter table employee add e-office char(30)
alter table employee drop e-office
```

**Drop (Delete) Relation Schema**

```sql
drop table relation-name
```

Delete the relation schema and instance!

e.g.  
```sql
drop table employee
```

**Create (Define)/Drop a View on one or more Relations**

```sql
create view view-name <(attribute names)> as query-expression
```

The query expression will be discussed in Section 3 (select ~ from ~ where ~). Note, *(attribute names)* is optional.

```sql
drop view view-name
```

e.g.  
```sql
create view dept-emp (d-num, d-name, emp-num, emp-name) as
    select department.dept-num, dept-name, e-num, e-name
    from department, employee
    where department.dept-num = employee.dept-num
```
drop view dept-emp

Again, the query-expression will be discussed in Section 3.

**Create (Build)/Drop an Index on a Relation (Not in standard SQL-92)**

`create <unique> index index-name on relation-name <(attribute names)>`

`drop index index-name`

Note, the option "**unique**" is used only when the given attribute-list is a superkey (i.e., each tuple has unique values on the attribute-list). Therefore, when create an index or insert/update a tuple, if this condition fails, an error will be flagged.

e.g.  
`create unique index b-tree1 on employee (e-num)`

`create index b-tree2 on employee (e-name)`

`drop index b-tree2`

Note, "create index" is not in SQL-92 standard. However, many commercial DBMS support this.
2. Instance Modification: DML (Data Manipulation Language) Features of SQL (1 of 2)

**Inserting Tuples into a Relation**

Type 1: `insert into` relation-name `<(attribute names)>` values (tuple value)

Type 2: `insert into` relation-name `<(attribute names)>` query-expression

Note, "(attribute names)" is optional. This option enables us to change the order of domains.

Type 1 e.g.

```
insert into employee
values (001, "Sam Murlas", 35, 1101)
```

e.g.

```
insert into employee (e-name, e-num, dept-num, e-age)
values ("Sam Murlas", 001, 1101, 35)
```

Type 2 uses SQL query-expression. SQL query-expression (i.e., `select ~ from ~ where ~`) is discussed in Section 3 in detail. So, this section gives only one simple example.

Type 2 e.g.

```
insert into account (branch-name, account-number, balance)
select branch-name, loan-number, 200
from loan
where branch-name = "Laramie"
```

Note, the result schema of the subquery must be the same as (attribute names) in "into" clause.
Deleting Tuples from a Relation

```
delete from relation-name <where P>
```

Note, P is a predicate. See the "where" clause in Section 3.

**e.g. 1**  
```
delete from employee
```

The above request will delete every tuple from employee relation.

**e.g. 2**  
```
delete from employee where dept-num=1101
delete from employee where e-age > 80
```

Updating Tuples

```
update relation-name set assignment <where P>
```

Note, P is a predicate. See the "where" clause in Section 3.

**e.g. 1**  
```
update employee
    set e-age = e-age+1
```

**e.g. 2**  
```
update employee
    set dept-num = dept-num+1
    where dept-num >= 1101
```

Insert, Delete, and Update operation on a View

In many DBMSs, these operations on a view is allowed only if the view is defined on one relation and the operation satisfies every constraints (and not-null) defined on the original relation.
3. Extraction of Information from the underlying database: DML (Data Manipulation Language)

Features of SQL (2 of 2)

Structure of query-expression

```
select * or L
from relation-name1 (or query-expression) <as temporary-name1>, ..., relation-nameN (or query-expression) <as temporary-nameN>
<where P1>
<group by attribute-names <having P2> >
<order by attr-name asc (or desc), attr-name asc (or desc), ...>
```

"select" clause

- * means there is no projection
- L is a list. L can include constant, A, f(A), and arithmetic expression (e.g., 0.3*A) involving A, where A is an attribute of the cartesian product of the relations in "from" clause and function f is an aggregation function. Aggregation functions: avg(), min(), max(), sum(), count().

"from" clause

If a query-expression is used in "from" clause, "as temporary-name" is mandatory since we need the name of the result relation of the sub-query. This clause produces a cartesian product of every relation in the clause. This single big relation is used by the other clauses as the base relation.

"where" clause

The predicate P1 can involve so many things including +, -, *, /, >, <, >=, <=, <>, =, and, or, not, like, between~and~, is null, in (query-expression),
some (query-expression), all (query-expression), exists (query-expression), unique (query-expression), and every attribute of the "from" clause.

"group by" clause
Group the tuples having the same values on the given attributes. "having" clause is to filter the groups. P2 is applied after the grouping. Thus, rather than raw attributes, aggregation functions are used.

"order by" clause
Ascending order (i.e., asc) is default. Represents the order in which the result tuples are placed in the result relation of the query. Sorting is very expensive.

Simple Selection
select * from employee
select * from employee where dept-num = 1101

Projection and Selection
Assumptions: employee schema = {e-num, e-name, age, dept-num}
department schema = {dept-num, dept-name}
select e-num, e-name, age+1, dept-num from employee
select e-name, e-num from employee where dept-num=1101
select dept-name, count(e-name), 200 from employee, department where department.dept-num = employee.dept-num group by department.dept-name
select e-name from employee where e-name like "John%"
Note, %, _, and \ are used with "like".
I will also explain "%Murlas", "John_ _ _ _ _ _ _", "John_ \% _ Murlas", and "John\Murlas" in the class. (see the related Section in the textbook)
Union, Intersect, and Set-Difference (unlike select, duplicates are eliminated)

Assumptions: checking schema = \{custID, acc-num, balance\}
saving schema = \{custID, acc-num, balance\}
customer schema = \{custID, name, phone, street, city, zip\}

Union Example: (select C.name, C.phone from checking as K, customer as C where K.custID=C.custID) union (select C.name, C.phone from saving as V, customer as C where V.custID=C.custID)

Set-Intersect Example: (select C.name, C.phone from checking as K, customer as C where K.custID=C.custID) intersect (select C.name, C.phone from saving as V, customer as C where V.custID=C.custID)

Set-Difference Example: (select C.name, C.phone from checking as K, customer as C where K.custID=C.custID) except (select C.name, C.phone from saving as V, customer as C where V.custID=C.custID)

Note, "union all", "intersect all", "except all" can be used to allow duplicates.

Note, some DBMSs don't support these operations.
**Note#1 Duplicates**
"select ~" query does not eliminate duplicates in the result set!
You can use "select distinct ~" to eliminate duplicates. Note, "select all ~" is allowed but "all" is default.
However, set operations, such as union, intersect, and except, on two "select~" queries eliminate duplicates from the result relation automatically.
Use "all" to allow duplicates.

**Note#2 Null values**
```sql
select custID, name from customer where phone is null
```
Aggregation functions ignore null. But, count(*) function counts that since there is a tuple.
e.g.,

relation r

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

select count(*) from r where A=1 ==>result: 2
select count(C) from r where A=1 ==> result: 0
select min(C) from r ==> result: 1
select avg(C) from r ==> result: 1
select avg(C) from r where A=1 ==> result: null
But, in fact, these details are dependent on the underlying DBMS.
Natural Join and Theta Join (inner, left outer, right outer, full outer joins)

Assumptions:

<table>
<thead>
<tr>
<th>relation r</th>
<th>relation s</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Examples:

**r inner join s on r.B = s.B**

(Note, inner join is, in fact, theta join)

<table>
<thead>
<tr>
<th>A</th>
<th>r.B</th>
<th>r.C</th>
<th>s.B</th>
<th>s.C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Same values

**r natural inner join s**

(Note, natural inner join is, in fact, natural join)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
r natural full outer join s

A | B | C | D  
---|---|---|---
1 | 2 | 2 | null  
1 | 2 | 3 | 4  
2 | 3 | 3 | 1  
null | 1 | 1 | 1

r inner join s using (B)
(use "using" to eliminate some common attributes. The parameter of "using"
must be $\subseteq (R \cap S)$, where R and S are the schemas of r and s, respectively)

A | B | r.C | s.C | D  
---|---|---|---|---
1 | 2 | 2 | 3 | 4  
1 | 2 | 3 | 3 | 4  
2 | 3 | 3 | 3 | 1

r inner join s using (B, C)
(This is the original natural join)

A | B | C | D  
---|---|---|---
1 | 2 | 3 | 4  
2 | 3 | 3 | 1

Join Types: inner join, left outer join, right outer join, full outer join

Join Conditions: natural, on P, and using (attribute names), where P is a predicate.

You choose one of join types and one of join conditions
So,
Natural Join: \( r \textit{natural inner join} s = r \textit{inner join} s \) using (every common attribute)
Theta Join: \( r \textit{inner join} s \) on \( P \) (Note, \( P \) is the theta)
Left Outer Join: \( r \textit{natural left outer join} s \)
Right Outer Join: \( r \textit{natural right outer join} s \)
Full Outer Join: \( r \textit{natural full outer join} s \)
Note, the condition "natural" is the same as "using (every common attribute)"

**More Examples**
Assumptions: checking schema = \{custID, acc-num, balance, branchID\}
 saving schema = \{custID, acc-num, balance, branchID\}
customer schema = \{custID, name, phone, street, city, zip\}
 branchs schema = \{branchID, branchName, city, zip, assets\}

select C.name, C.phone 
from customer C, checking K 
where C.custID=K.custID or C.custID in (select custID from saving)

select C.name, C.phone 
from customer C, checking K 
where C.custID=K.custID or C.custID \textbf{in (001, 101, 111)}

select C.name, C.phone 
from customer C, checking K 
where C.custID=K.custID and C.custID \textbf{not in} (select custID from saving)
select branchName, city
from branches
where assets < some (select assets from branches where city="Laramie")
Note, "some" is "at least one".
select branchName, city
from branches
where assets > all (select assets from branches where city="Laramie")

select name
from customer
where exists (select * from checking where
checking.custID=customer.custID)

select name
from customer
where not exists (select * from checking where
checking.custID=customer.custID)

select name
from customer
where unique (select custID from checking where
checking.custID=customer.custID)

select name
from customer
where **not unique** (select custID from checking where checking.custID=customer.custID)

select r, p
from (select name, phone, balance
 from customer C, checking K
 where C.custID=K.custID) as r(n, p, b)
where b >= 2000

Note, a query-expression is treated as a relation. You can use a query expression as a relation variable like "customer".