- Database Modifications:
- Modification means insert, delete or update (change of attribute values)
- Tuple Insertion: "`INSERT INTO` Relation `VALUES` (list of values)"

  Inserts the tuple = list of values
- Values associated with attributes in the order in which they were declared

  Alternative, give list of attributes as arguments of the relation

  If any of them is omitted, default value will be used, e.g. `NULL`
- Example: Insert the fact that Sally likes Bud

```
Likes(drinker, beer)

INSERT INTO   Likes(drinker, beer)
VALUES('Sally', 'Bud');
```

- Insertion of the Result of a Query:

  `INSERT INTO` Relation (SubQuery)

- Example: Create a (unary) table of all Sally's buddies, i.e., the people who frequent bars that Sally also frequents

  `Frequents(drinker, bar)`

```
CREATE  TABLE    Buddies(
        name char(30)
        );
INSERT  INTO     Buddies
(SELECT DISTINCT d2.drinker
FROM    Frequents d1, Frequents d2
WHERE   d1.drinker = 'Sally' AND
        d2.drinker <> 'Sally' AND d1.bar = d2.bar );
```

- Deletions: "`DELETE FROM` Relation `WHERE` Condition"

  Deletes all tuples satisfying the condition from the named relation

- Example: Sally no longer likes Bud

  `Likes(drinker, beer)`

```
DELETE FROM Likes
WHERE  drinker = 'Sally' AND beer = 'Bud';
```

- Example: Make the Likes relation empty

```
DELETE FROM Likes;
```

- Example: Delete all beers for which there is another beer by the same manufacturer

  `Beers(name, manf)`

```
DELETE FROM   Beers b
WHERE  EXISTS
       (SELECT name
       FROM   Beers
       WHERE  manf = b.manf AND name <> b.name);
```

- Tuples of the form `(name, manf)` are deleted from `Beers`

  Or better: Tuples of the form `(b.name, b.manf)` are deleted from `b`

- In last line: `manf` and `name` correspond to `Beers`

  `b.manf` and `b.name` are external variables

- Subquery evaluated once for each row of `b`

- Be careful with the deletion semantics
- What if `AB` makes `Bud` and `Bud Lite` (only)?

  Does the deletion of `Bud` make `BudLite` not satisfy the condition?
- The SQL semantics says that both should be deleted
- SQL Semantics: All conditions about modifications are evaluated by the system before any changes are made
- In the previous example, both beers are first identified as targets

  Next, both are deleted

- Updates: "`UPDATE` Relation `SET` list of assignments `WHERE` Condition"
- To change values in existing tuples in the relation
- Example: Drinker Fred's phone number is now 555-1212

`Drinkers(name, addr, phone)`

```
UPDATE Drinkers
SET    phone = '555-1212'
WHERE  name = 'Fred';
```

- Example: Make $4 the maximum price for beer
- Updates many tuples at once

`Sells(bar, beer, price)`

```
UPDATE Sells
SET    price = 4.00
WHERE  price > 4.00;
```

- NULL Values: Can be used in place of (as) a value for a tuple's attribute
- Many reasons why a NULL value is present

  "missing value", "value inappropriate", "there exists a value but it is not available", "value doesn't exist", "default value", etc
- The semantics is not quite clear

  Correspondingly, the same applies to the operational semantics
- Among other issues:
  - If we do an operation of a NULL value with any other value (NULL or not), we get NULL
  - If a NULL is compared with another value (NULL or not) in a condition, we obtain the third truth value: UNKNOWN
- A query only produces tuples if the WHERE-condition evaluates to TRUE (UNKNOWN is not sufficient)

- Example:

bar	beer	price
Joe's bar	Bud	NULL

```
SELECT bar
FROM   Sells
WHERE  price < 2.00 OR price >= 2.00;
```

- Although "tautological", the two conditions in the disjunction evaluate to `UNKNOWN`

  Therefore, the condition evaluates to `UNKNOWN`

  Joe's bar is not returned

- The combination of the classical values of truth, `TRUE` and `FALSE`, with `UNKNOWN` follow its own logic, a three-valued logic

  There are many more issues with `NULL`

- The SQL Standard is ambiguous/incomplete in this regard

- Different DBMSs may have differences at some point in their operational semantics

- NULL Values and Joins:
- Issues with Natural Join: $\texttt{WINE} \bowtie_{GRAPE} \texttt{LOCATION}$?

WINE	GRAPE	VINTAGE	QUALITY
	Chenas	1977	Good
	NULL	1980	Excellent
	Chablis	1977	Good
	Chablis	1978	Bad
	Volnay	1980	Average

LOCATION	GRAPE	AREA	AVG-QUALITY
	Chenas	Beaujolais	Good
	NULL	Bourgogne	Average
	Zinfandel	California	Bad

- Tuple $\langle \texttt{NULL}, \texttt{1980}, \texttt{Excellent}, \texttt{Bourgogne}, \texttt{Average} \rangle$ NOT returned
- $\langle \texttt{Chenas}, \texttt{1977}, \texttt{Good}, \texttt{Bourgogne}, \texttt{Average} \rangle$ NOT returned either

  Even when NULL could stand for Chenas
- The null values also have an application with Outer Joins
- Outer Join: Operation to report on the tuples that do not match up

  In the context of joins, a tuple is "dangling" if it does not pair up with any other tuple

- Example:

R	A	B
	8	2
	3	4

S	B	C
	2	5
	2	6
	7	8

- Here tuple $(3, 4)$ of R, and $(7, 8)$ of S are dangling
- Through the natural join they get lost
- A full outer join of R and S does not lose them

  It includes them filling with NULL

R ⟗ S	A	B	C
	8	2	5
	8	2	6
	3	4	NULL
	NULL	7	8

- There are also "left and right outer joins"

  Return third and fourth tuples, resp.
- Can be seen as extension of RA
- Different ways of specifying outer joins in SQL

- Joins in SQL:
- Joins can be specified with or w/o a `select-from-where` clause
- Can be used to define a relation in `FROM` clause (cf. 3 below)
- `R NATURAL JOIN S`: Simplest join

  A cross product requiring attributes in common to be equal

  Attributes in common shown only once in the result
- Examples: (relative to preceding slide)

```
1) (Select A,B FROM R)
           NATURAL JOIN
   (Select B,C FROM S);
2) R NATURAL JOIN S;
3) Select A,B,C
   FROM R NATURAL JOIN S
```

All of them produce the same result:

R ⋈ S	A	B	C
	8	2	5
	8	2	6

- `R JOIN S ON` *condition*

  Do cross product, and choose rows as specified by ON

- Example:

```
4) SELECT *
   FROM R JOIN S ON
        R.B = S.B
```

Result:

R ⋈ S	A	B	C
	8	2	5
	8	2	6

- This query can also be written w/o an explicit join:

```
5) SELECT *
   FROM R, S
   WHERE R.B =S.B;
```

- 4) and 5) produce same result as the natural join

- Conditions can be more complex than that for natural join
- This one compares other attributes:

```
6)  SELECT R.B, R.A, S.B
    FROM R JOIN S ON
         R.A = S.C
```

Result:

R.B	R.A	S.B
2	8	7

- `R CROSS JOIN S`: the cartesian product

```
7)  SELECT *
    FROM R CROSS JOIN S
```

Result:

R.A	R.B	S.B	S.C
8	2	2	5
8	2	2	6
8	2	7	8
3	4	2	5
3	4	2	6
3	4	7	8

- This query can also be written without an explicit join:

```
8)  SELECT *
    FROM R, S
```

- `R OUTER JOIN S:` Non-matching tuples added with `NULL`s
- Options:
  - `NATURAL` optional at the beginning
  - `ON` *condition* optional at the end
  - `LEFT`, `RIGHT`, or `FULL` optional before the `OUTER`
  - `LEFT` : only add the dangling tuples of `R`
  - `RIGHT` : only add the dangling tuples of `S`
- Example:

```
9)  SELECT *
    FROM R LEFT OUTER JOIN S
    ON R.B = S.B;
```

Result:

A	B	C
8	2	5
8	2	6
3	4	NULL

- The outer join is useful and important:
  - A view is defined through a query, e.g. $V := R \bowtie S$
  - User, may want to (or only) have access to the data through the view
  - In particular, retrieving, e.g. the data of R through the view
  - If is is not an outer join, then the user would be missing information about R (and S)
  - Important topics in DBs: Queries and updates using views
- **In Oracle:** Outer join can be specified in `select-from-where` clause

  By adding a $(+)$ on one of the sides of the equality condition

- Example: Give a list of all the beers sold in Joe's Bar, with the manufacturers

  Include the names of the beers even if the name of the manufacturer is unknown

  `Beers(name, manf)` `Sells(bar, beer, price)`

```
SELECT beer, manf
FROM   Sells, Beers
WHERE  bar = 'Joe''s Bar' AND beer = name(+);
```

- A join of two tables
- Here, beers (i.e. names) from table `Beers` will also be returned

  They may not appear in table `Sells`

  In that case, the manufacturer column (manf) of the result is filled in with a `NULL` when its value is unknown

- Views:
- An expression that describes a table without creating that table in the database (without physical materialization)
- The view is defined in terms of base (material) tables
- Creates a virtual table, defined by an expression
- The expression that creates the views is generally a query
- Useful when a query is going to be used frequently
- A view can be conceived as query with a name
- View definition form is:

```
CREATE VIEW < name > AS
            < query >;
```

- Example: View `CanDrink` is the set of drinker-beer pairs such that the drinker frequents at least one bar that serves the beer

```
CREATE  VIEW   CanDrink AS
        SELECT drinker, beer
        FROM   Frequents, Sells
        WHERE  Frequents.bar = Sells.bar;
```

- This is a "conjunctive view" or an SPJ view: defined by a conjunctive query, i.e. expressed using joins, selections and projections
- A virtual table with two attributes (inherited from base tables)
- Querying (with) Views: Treat the view as if it were a materialized relation (can also be combined with other tables/views)

  Example: What beers can Sally drink?

```
SELECT beer
FROM   CanDrink
WHERE  drinker = 'Sally';
```

- This query can be posed after creating the view

- The view definition and its extension will survive during the interaction session with the DB
- After that, everything will disappear
- Unless the user decides to materialize the view

  Creating a new relation schema and storing its contents
- What about updates during a session?
- Updates on base tables used to define the view?
- Example: `Sells(bar, beer, price)` `Frequents(drinker, bar)`

Sells	bar	beer	price
	Joe's	Bud	5
	Sue's	Miller	6
	Leo's	Duvel	8
	Sue's	Duvel	6
	Roe's	Miller	7

Frequents	drinker	bar
	pete	Joe's
	john	Sue's
	john	Leo's
	ric	Sue's
	mary	Roe's

```
CREATE  VIEW   CanDrink AS
SELECT drinker, beer
FROM   Frequents, Sells
WHERE  Frequents.bar = Sells.bar;
```

CanDrink	drinker	beer
	pete	Bud
	john	Miller
	john	Duvel
	ric	Miller
	ric	Duvel
	mary	Miller

- Relevant updates: May change view contents
  - Insertions and deletions on `Sells`, `Frequents`
  - All changes of attribute values (except for those in `price`)

- Exercise: Show how you would materialize the `CanDrink` view
- Virtual extension of a view must be kept up-to-date during a session

  Synchronized with changes on underlying base tables
- The is the View Maintenance Problem
- Very similar (and related) to the IC Maintenance Problem

  Similar issues and techniques
- A brutal, undesirable approach to keep the view up-to-date:

  Every time a relevant update on base tables is performed, recompute the full view contents from scratch
- Better: Apply incremental view maintenance methods, as the base table undergo updates

  Again, very similar methods for IC maintenance ...