NOT NULL ICs:

- Example:

```
CREATE TABLE Drinkers(
        name CHAR(30) PRIMARY KEY,
        phone CHAR(16) NOT NULL );
```

- Otherwise, without the NOT NULL IC, we get:

```
INSERT INTO Drinkers(name)
VALUES('Sally');
```

name	phone
Sally	NULL

- Other specifications:
  - NOT NULL: every tuple must have a real value for this attribute
  - DEFAULT value: a value to use whenever no other value for this attribute is known

- Example:

```
CREATE TABLE Drinkers (
        name CHAR(30) PRIMARY KEY,
        addr CHAR(50) DEFAULT '123 Sesame St',
        phone CHAR(16)   );
```

- With the insertion:

```
INSERT INTO Drinkers(name)
VALUES('Sally');
```

name	addr	phone
Sally	123 Sesame St.	NULL

  - Primary key is by default NOT NULL
  - This insertion is perfectly legal

    It is OK to list a subset of the attributes, and values for only this subset
  - Had we declared

    "phone CHAR(16) NOT NULL",

    the insertion could not be done

Changing Columns of a Schema:

- Add an attribute to relation R with

```
ALTER TABLE R ADD <column declaration>;
```

- Example:

```
ALTER TABLE Bars ADD phone CHAR(16)
       DEFAULT 'unlisted';
```

- Columns may also be dropped:

```
ALTER TABLE Bars DROP license;
```

SQL: Queries

- Basic Syntax:

```
SELECT desired attributes
FROM   tuple variables - range over relations
WHERE  condition about tuple variables
```

- Example: Schema:

```
Beers(name, manf)
Bars(name, addr, license)
Drinkers(name, addr, phone)
Likes(drinker, beer)
Sells(bar, beer, price)
Frequents(drinker, bar)
```

- Query: What beers are made by Anheuser-Busch?

```
SELECT name
FROM   Beers
WHERE  manf = 'Anheuser-Busch';
```

(notice the single quotes for strings)

- Answer:

name
Bud
Bud Lite
Michelob

- Conditions in WHERE capture RA selection

  SELECTing only some attributes captures RA projection

- Extensions of the Basic Syntax:
- A star can be used to retrieve all the attributes:

Beers(name, manf)

```
SELECT *
FROM   Beers
WHERE  manf = 'Anheuser-Busch';
```

- Answer:

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch

- Renaming Columns: 'name' into 'beer' in Beers(name, manf)

```
SELECT name AS beer
FROM   Beers
WHERE  manf = 'Anheuser-Busch';
```

- Answer:

beer
Bud
Bud Lite
Michelob

Table useful by itself

Can be combined with other queries

- Expressions as Values in Columns:
- Sells(bar, beer, price)

```
SELECT bar, beer, price*150 AS priceInYen
FROM   Sells;
```

'price*150': expression 'priceInYen': name for it

- Answer:

bar	beer	priceInYen
Joe's	Bud	300
Sue's	Miller	360

- Answers with a particular string in each row: use that constant as an expression Likes(drinker, beer)

```
SELECT drinker, 'likes Bud' AS whoLikesBud
FROM  Likes
WHERE beer = 'Bud';
```

'likes Bud': expression 'whoLikesBud': name for it

- Answer:

drinker	whoLikesBud
Sally	likes Bud
Fred	likes Bud

- Example: Find the price Joe's Bar charges for Bud
- `Sells(bar, beer, price)`

```
SELECT price
FROM   Sells
WHERE bar = 'Joe''s Bar' AND beer = 'Bud';
```

- Two single-quotes in a string represent a single quote
- Conditions in `WHERE` clause can use logical connectives

  `AND`, `OR`, `NOT` and parentheses as usual
- SQL is case insensitive

  Keywords like `SELECT` or `AND` in upper or lower case

  Only inside quoted strings does case matter

- Patterns:

- '%' stands for any string
- '_' stands for any (single) character
- "Attribute `LIKE` pattern" is a condition that is true if the string value of the attribute matches the pattern

  Also `NOT LIKE` can be used

- Example: Find drinkers whose phone starts with 555

```
Drinkers(name, addr, phone)

SELECT name
FROM   Drinkers
WHERE phone LIKE '%555-____';
```

- Patterns must be quoted, like strings

- Multi-Relation Queries:
- A list of relations in `FROM` clause
- Notation: *Relation.Attribute*

  Disambiguates shared attributes from several relations
- Example: Find the beers that the frequenters of Joe's Bar like

  Likes(drinker, beer) Frequents(drinker, bar)

  'drinker' is in two different tables

```
SELECT beer
FROM   Frequents, Likes
WHERE  bar = 'Joe''s Bar' AND
       Frequents.drinker = Likes.drinker;
```

- We can express products and joints of RA
- Here: Likes $\bowtie$ Frequents, selection, projection (on beer)
- The common class of "select-project-join" (SPJ) queries

  Also known as "conjunctive queries"

- Explicit Tuple Variables:

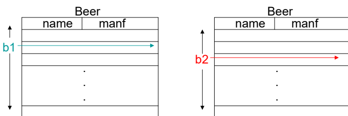
- Sometimes we need to refer to two or more copies of a relation
  E.g. for comparing values within a same relation

- Use explicit variables for copies of relations (or tuples thereof)
  Intuitively: Use *tuple variables* as aliases for the relations

- Example: Find pairs of beers by the same manufacturer

  Beers(name, manf)

```
→  SELECT b1.name, b2.name
   FROM   Beers b1, Beers b2
   WHERE  b1.manf = b2.manf  AND
          b1.name <> b2.name;
```



b1.manf = b2.manf ?

- b1, b2: tuple variables, aliases for relations
  In FROM clause: selection from copies b1 and b2 of Beers

- 'b1.name <> b2.name' needed to avoid producing (Bud, Bud), and a same pair in both orders

- Before we used AS to rename attributes or expressions
- SQL permits AS between a relation and its tuple variable:

```
SELECT b1.name, b2.name
FROM   Beers AS b1, Beers AS b2
WHERE  b1.manf = b2.manf AND b1.name <> b2.name;
```

- Same query in RC:

$$Ans(x,u):\ \exists y(Beer(x,y) \wedge Beer(u,y) \wedge x \neq u)$$

- A conjunctive query in RC, with a built-in $(\neq)$
- Exercise: Pose the query in RA

- Sub-Queries:
- Result of a *select-from-where* query can be used in the WHERE clause of another query

  The Simplest Case: Sub-query returns a single tuple

- Example: Find bars that serve Miller at the same price Joe charges for Bud

  Sells(bar, beer, price)

- First we find what Joe charges for Bud

  Next, the bars that sell Miller for that same price

```
SELECT bar
FROM   Sells
WHERE  beer = 'Miller'  AND  price =
       (SELECT price
       FROM    Sells
       WHERE   bar = 'Joe''s Bar' AND beer = 'Bud'
       );
```

- Nested SELECTs

```
SELECT bar
FROM   Sells
WHERE  beer = 'Miller'  AND  price =
       (SELECT price
       FROM    Sells
       WHERE   bar = 'Joe''s Bar' AND beer = 'Bud'
       );
```

- *Scoping rule*: An attribute refers to the closest nested relation with that attribute
- Parentheses around sub-queries are essential
- `price` is uniquely determined (by the key)
  The sub-query returns a single value
- What if the sub-query returns a set of tuples?
- Use "`IN`" instead of '$=$'
  It is true iff the tuple is in the extension of the relation defined by sub-query

- Example: Find the name and manufacturer of beers that Fred likes

  Beers(name, manf)   Likes(drinker, beer)

```
SELECT *
FROM   Beers
WHERE  name IN
       (SELECT beer
       FROM    Likes
       WHERE   drinker = 'Fred');
```

- Main (final) selection is on Beers, with attributes (name,manf)

  But the condition depends on table Likes (in the sub-query)

- The sub-query creates a set (a unary relation of names)

  Main query checks if name belongs to it

- In general (not here), NOT IN can also be used

- Quantifiers:
- Inherited from Relational Calculus

  "EXISTS (Relation)" is true iff Relation is non-empty

- Example: Find the beers that are the only beer made by its company

  Beers(name, manf)

```
SELECT b1.name
FROM   Beers b1
WHERE  NOT EXISTS(
       SELECT *
       FROM   Beers
       WHERE  manf = b1.manf
              AND name <> b1.name );
```

  "Choose the name of the beers such that there is no other beer produced by the same manufacturer"

- Think of having a generic, variable, tuple in Beers, b1, and checking a condition on it

  A condition in terms of a quantifiers (NOT EXISTS) applied to a sub-query

  Keep the b1s that satisfy the condition (their names)

```
SELECT b1.name
FROM   Beers b1
WHERE  NOT EXISTS(
       SELECT *
       FROM   Beers
       WHERE  manf = b1.manf
              AND name <> b1.name );
```

- Variable `b1` goes over tuples in `Beers` (alias for `Beers`)
- `b1.name` is just another name to refer to `name`
- In the subquery, with `b1` fixed (as a variable outside the subquery):

  We get the tuples `(name, manf)` with:

  $$\texttt{manf} = \texttt{b1.manf} \quad \& \quad \texttt{name} \neq \texttt{b1.name}$$

- *Scoping rule*: To refer to external `Beers` in the sub-query, give to the external tuple a variable (`b1` in this example)
- A subquery that refers to values from a surrounding query is called a *correlated sub-query*
- Exercise: Express the query in RC and RA

- ANY and ALL behave as existential and universal quantifiers, respectively
- Beware: In English, "any" and "all" sometimes act as synonyms

  For example, "I am fatter than any of you" vs. "I am fatter than all of you"

  Not in SQL!
- They can be used to express numerical maxima and minima

  "A value is a maximum if *all* values are not higher"
- Example: Find the beers sold for the highest price

Sells(bar, beer, price)

```
SELECT beer
FROM   Sells
WHERE  price >= ALL(SELECT price
                    FROM   Sells);
```

(the sub-query returns real numbers)

- Exercise: Find the beers not sold for the lowest price (use ANY)

- Bank Example: Schema:

```
Branch(branch-name, branch-city, assets)
Customer(customer-name, customer-street, customer-city)
Account(branch-name, account#, balance)
Depositor(customer-name, account#)
Loan(branch-name, loan#, amount)
Borrower(customer-name, loan#)
```

- Example 1: Find all customers who have both an account and a loan at the bank

```
SELECT customer-name
FROM   Borrower
WHERE  customer-name IN (SELECT customer-name
                         FROM   Depositor);
```

- Example 2: Find all customers who have a loan but not an account at the bank

```
SELECT customer-name
FROM   Borrower
WHERE  customer-name NOT IN (SELECT customer-name
                             FROM   Depositor);
```

- Example 3: Find the customers at the "First Street" branch with the highest account balance at that branch

```
SELECT customer-name
FROM   Depositor, Account
WHERE  Depositor.account#=Account.account AND
       branch-name= 'First Street' AND
       Account.balance >= ALL
               (SELECT balance
                FROM Account
                WHERE branch-name='First Street');
```

- Union, Intersection, Difference:
- One can bring RA operators explicitly into SQL

  UNION, INTERSECT as usual

  EXCEPT for the difference of the two relations
- They require a shared schema
- Example: Find the drinkers and beers such that the drinker likes the beer and frequents a bar that serves it

```
Likes(drinker, beer)    Sells(bar, beer, price)
Frequents(drinker, bar)
```

```
(SELECT * FROM Likes)
                INTERSECT
(SELECT drinker, beer
 FROM   Sells, Frequents
 WHERE  Frequents.bar = Sells.bar );
```

(a way to pick tuples from LIKES with conditions coming from other relations)

(selection of drinkers that frequent a bar where those beers are served)

- In SELECT: A join between Sells and Frequents, and a projection over drinker and beer (so, same schema as LIKES)
- Exercise: Solve it without INTERSECT