



Databases 1



Queries in SQL: SELECT

SQL

- ▶ SQL: Structured Query Language
- ▶ SQL is a very-high-level language, the main parts:
DDL data-definition language (create, drop, alter)
DML data-manipulation language (insert, delete, update and select). Queries: SELECT statement
- ▶ What makes SQL viable is that its queries are “optimized” quite well, yielding efficient query executions.

Basic statement

- ▶ The principal form of a query is:

SELECT *desired attributes*

FROM *one or more tables*

WHERE *condition about tuples of the tables*

Our Running Example

- ▶ All our SQL queries will be based on the following database schema.
 - ▶ *Underline indicates key attributes.*

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Example

- ▶ Using `Beers(name, manf)`, what beers are made by Anheuser-Busch?

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

- ▶ Relational algebraic expression:

$$\pi_{name} \left(\sigma_{\text{manf} = \text{'Anheuser-Busch'}} (Beers) \right)$$

Result of Query

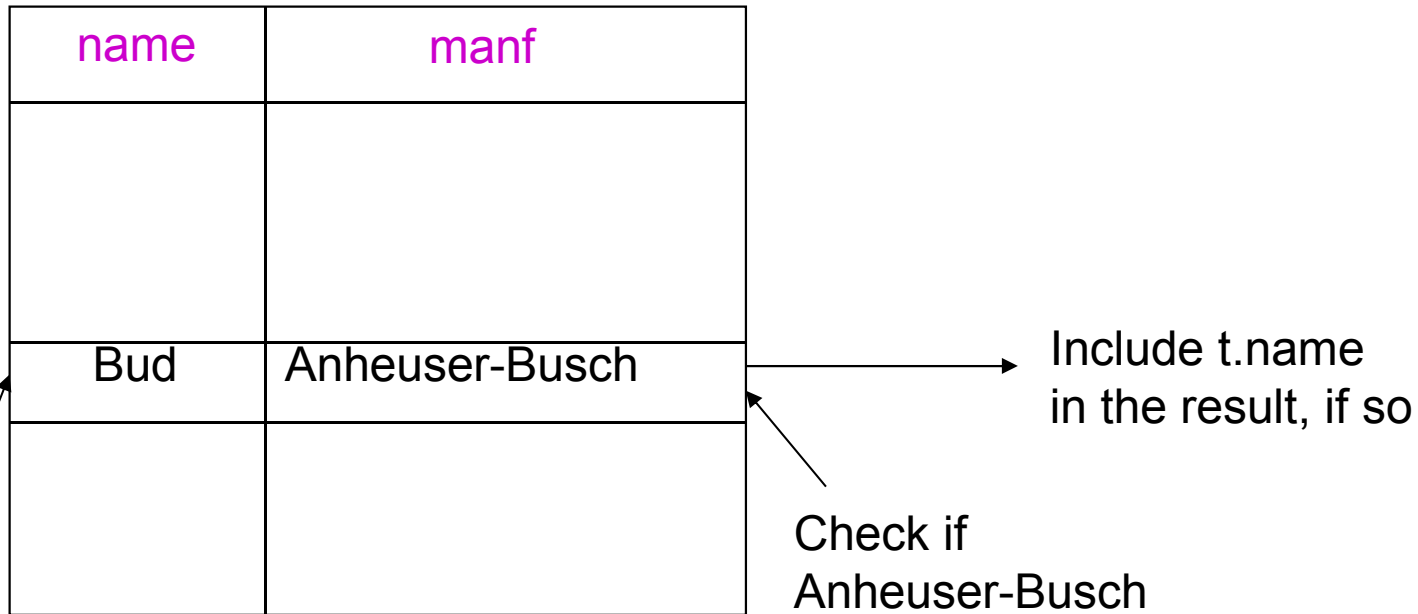
name
Bud
Bud Lite
Michelob

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

Meaning of Single-Relation Query

- ▶ Begin with the relation in the FROM clause.
- ▶ Apply the selection indicated by the WHERE clause.
- ▶ Apply the extended projection indicated by the SELECT clause.

Operational Semantics



Tuple-variable t
loops over all
tuples

Operational Semantics --- General

- ▶ Think of a *tuple variable* visiting each tuple of the relation mentioned in FROM.
- ▶ Check if the “current” tuple satisfies the WHERE clause.
- ▶ If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

(star) * in SELECT clauses

- ▶ When there is one relation in the FROM clause, * in the SELECT clause stands for “all attributes of this relation.”
- ▶ Example using Beers(name, manf):

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

Result of Query:

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

Now, the result has each of the attributes of Beers.

Expressions in SELECT Clauses

- ▶ Any expression that makes sense can appear as an element of a SELECT clause.
- ▶ If you want the result to have different attribute names, use “AS <new name>” to rename an attribute.
- ▶ Example: from Sells(bar, beer, price):

```
SELECT bar, beer, price * 120 AS priceInYen  
FROM Sells;
```

Result of Query

Bar	Beer	PriceinYen
Joe's	Bud	300
Joe's	Miller	330
Sue's	Bud	300
Sue's	Miller	360

Complex Conditions in WHERE Clause

- ▶ From Sells(bar, beer, price), find the price Joe's Bar charges for Bud:

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

- ▶ Result of the Query:

price
2.50

Important Points

- ▶ Two single quotes inside a string represent the single-quote (apostrophe).
- ▶ Conditions in the WHERE clause can use AND, OR, NOT, and parentheses in the usual way boolean conditions are built.
- ▶ SQL is *case-insensitive*. In general, upper and lower case characters are the same, **except inside quoted strings.**

Patterns

- ▶ WHERE clauses can have conditions in which a string is compared with a pattern, to see if it matches.
- ▶ General form:
 - ▶ <Attribute> LIKE <pattern>
 - ▶ <Attribute> NOT LIKE <pattern>
- ▶ Pattern is a quoted string
 - ▶ with % = “any string”
 - ▶ with _ = “any character.”

Example

- ▶ From Drinkers(name, addr, phone) find the drinkers with exchange 555:

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

NULL Values

- ▶ Tuples in SQL relations can have NULL as a value for one or more components.
- ▶ Meaning depends on context. Two common cases:
 - ▶ *Missing value* : e.g., we know Joe's Bar has some address, but we don't know what it is.
 - ▶ *Inapplicable* : e.g., the value of attribute *spouse* for an unmarried person.

Comparing NULL's to Values

- ▶ The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- ▶ When any value is compared with NULL, the truth value is UNKNOWN.
- ▶ But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued Logic

- ▶ To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN = $\frac{1}{2}$.
- ▶ AND = MIN; OR = MAX, NOT(x) = $1-x$.

- ▶ **Example:**

TRUE AND (FALSE OR NOT(UNKNOWN)) =
MIN(1, MAX(0, (1 - $\frac{1}{2}$))) =

MIN(1, MAX(0, $\frac{1}{2}$)) = MIN(1, $\frac{1}{2}$) = $\frac{1}{2}$.

Surprising Example

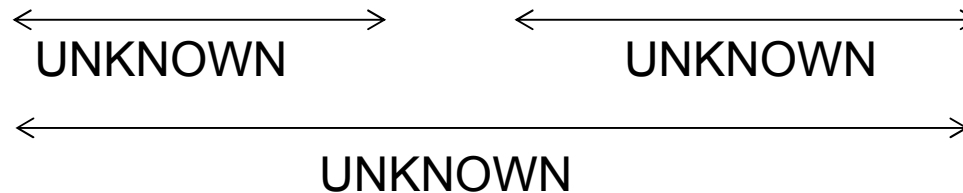
- ▶ From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE price < 2.00 OR price >= 2.00;



Multirelation Queries

- ▶ Interesting queries often combine data from more than one relation.
- ▶ We can address several relations in one query by listing them all in the FROM clause.
- ▶ Distinguish attributes of the same name by “<relation>.<attribute>”

Example

- ▶ Using relations Likes(drinker, beer) and Frequent(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

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

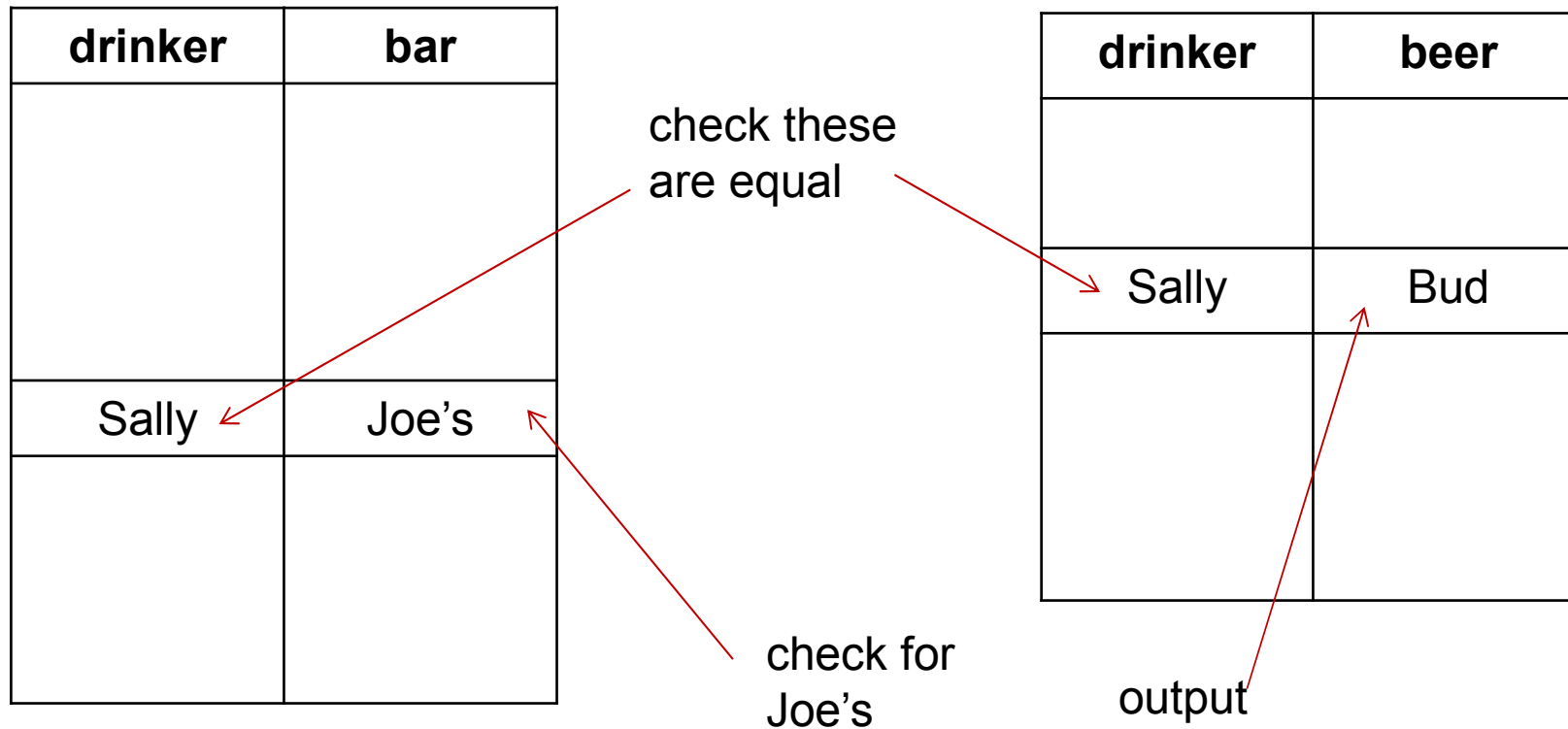
Formal Semantics

- ▶ Almost the same as for single-relation queries:
 1. Start with the product of all the relations in the FROM clause.
 2. Apply the selection condition from the WHERE clause.
 3. Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- ▶ Imagine one tuple-variable for each relation in the FROM clause.
 - ▶ These tuple-variables visit each combination of tuples, one from each relation.
- ▶ If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example



Explicit Tuple-Variables

- ▶ Sometimes, a query needs to use two copies of the same relation.
- ▶ Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- ▶ It's always an option to rename relations this way, even when not essential.

Example: Self-Join

- ▶ From Beers(name, manf), find all pairs of beers by the same manufacturer.

Do not produce pairs like (Bud, Bud).

Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

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

Union, Intersection, and Difference

- ▶ Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
 - ▶ (subquery) UNION (subquery)
 - ▶ (subquery) INTERSECT (subquery)
 - ▶ (subquery) [EXCEPT | MINUS](subquery)

Example

- ▶ From relations Likes(drinker, beer), Sells(bar, beer, price) and Frequent(drinker, bar), find the drinkers and beers such that:
 1. The drinker likes the beer, and
 2. The drinker frequents at least one bar that sells the beer.

Solution

Notice trick:
subquery is
really a stored
table.

(SELECT * FROM Likes)

INTERSECT

(SELECT drinker, beer
FROM Sells, Frequents
WHERE Frequents.bar = Sells.bar
);

The drinker frequents
a bar that sells the
beer.

Bag (multiset) Semantics

- ▶ Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is set semantics.
- ▶ That is, duplicates are eliminated as the operation is applied.

Motivation: Efficiency

- ▶ When doing projection in relational algebra, it is easier to avoid eliminating duplicates.
 - ▶ Just work tuple-at-a-time.
- ▶ When doing intersection or difference, it is most efficient to sort the relations first.
 - ▶ At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- ▶ Force the result to be a set by
`SELECT DISTINCT . . .`

- ▶ Force the result to be a bag (i.e., don't eliminate duplicates) by `ALL`, as in
`. . UNION ALL . . .`

Example: DISTINCT

- ▶ From Sells(bar, beer, price), find all the different prices charged for beers:

```
SELECT DISTINCT price  
FROM Sells;
```

- ▶ Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.

Example: ALL

- ▶ Using relations `Frequents(drinker, bar)` and `Likes(drinker, beer)`:

```
(SELECT drinker FROM Frequents)
  EXCEPT ALL
  (SELECT drinker FROM Likes);
```

- ▶ Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.

Join Expressions

- ▶ SQL provides a number of expression forms that act like varieties of join in relational algebra.
 - ▶ But using bag semantics, not set semantics.
- ▶ These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Products and Natural Joins

- ▶ Natural join:

`R NATURAL JOIN S;`

- ▶ Product:

`R CROSS JOIN S;`

- ▶ **Example:**

`Likes NATURAL JOIN Sells;`

- ▶ Relations can be parenthesized subqueries, as well.

Theta Join

- ▶ **R JOIN S ON <condition>**
- ▶ **Example:** using **Drinkers(name, addr)** and **Frequents(drinker, bar):**

```
Drinkers JOIN Frequents ON  
    name = drinker;
```

gives us all (d, a, d, b) quadruples such that drinker d lives at address a and frequents bar b .

Subqueries

- ▶ A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- ▶ Example: in place of a relation in the FROM clause, we can place another query, and then query its result.
 - ▶ Better use a tuple-variable to name tuples of the result.

Subqueries That Return One Tuple

- ▶ If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
 - ▶ Usually, the tuple has one component.
 - ▶ Also typically, a single tuple is guaranteed by keyness of attributes.
 - ▶ A run-time error occurs if there is no tuple or more than one tuple.

Example: Single-Tuple Subquery

- ▶ From Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud.
- ▶ Two queries would surely work:
 1. Find the price Joe charges for Bud.
 2. Find the bars that serve Miller at that price.

Query + Subquery Solution

```
SELECT bar  
FROM Sells  
WHERE beer = 'Miller' AND
```

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

The price at
which Joe
sells Bud



The IN Operator

- ▶ `<tuple> IN <relation>` is true if and only if the tuple is a member of the relation.
 - ▶ `<tuple> NOT IN <relation>` means the opposite.
- ▶ IN-expressions can appear in WHERE clauses.
- ▶ The `<relation>` is often a subquery.

Example: IN

- ▶ From Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Fred likes.

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

The set of
beers Fred
likes

The Exists Operator

- ▶ EXISTS(<relation>) is true if and only if the <relation> is not empty.
- ▶ Being a boolean-valued operator, EXISTS can appear in WHERE clauses.
- ▶ Example: From Beers(name, manf), find those beers that are the unique beer by their manufacturer.

Example Query with EXISTS

```
SELECT name
FROM Beers b1
WHERE NOT EXISTS(
```

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute.

Set of beers with the same manf as b1, but not the same beer

```
SELECT *
FROM Beers
WHERE manf = b1.manf AND
      name <> b1.name);
```

Notice the SQL "not equals" operator

The Operator ANY

- ▶ $x = \text{ANY}(\langle \text{relation} \rangle)$ is a boolean condition meaning that x equals at least one tuple in the relation.
- ▶ Similarly, $=$ can be replaced by any of the comparison operators.
- ▶ Example: $x \geq \text{ANY}(\langle \text{relation} \rangle)$ means x is not smaller than all tuples in the relation.
 - ▶ Note tuples must have one component only.

The Operator ALL

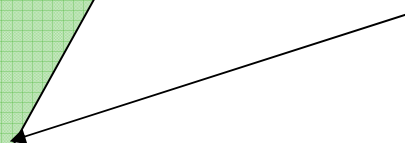
- ▶ Similarly, $x \neq \text{ALL}(\text{<relation>})$ is true if and only if for every tuple t in the relation, x is not equal to t .
 - ▶ That is, x is not a member of the relation.
- ▶ The \neq can be replaced by any comparison operator.
- ▶ Example: $x \geq \text{ALL}(\text{<relation>})$ means there is no tuple larger than x in the relation.

Example: ALL

- ▶ From Sells(bar, beer, price), find the beer(s) sold for the highest price.

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

price from the outer
Sells must not be
less than any price.



Aggregation Operators

- ▶ Aggregation operators are not operators of relational algebra.
- ▶ Rather, they apply to entire columns of a table and produce a single result.
- ▶ The most important examples: SUM, AVG, COUNT, MIN, and MAX.

Aggregations

- ▶ SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- ▶ Also, COUNT(*) counts the number of tuples.

Example: Aggregation

- ▶ From Sells(bar, beer, price), find the average price of Bud:

```
SELECT AVG (price)
FROM Sells
WHERE beer = 'Bud' ;
```

Eliminating Duplicates in an Aggregation

- ▶ **DISTINCT** inside an aggregation causes duplicates to be eliminated before the aggregation.
- ▶ **Example:** find the number of different prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud';
```

NULL's Ignored in Aggregation

- ▶ NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- ▶ But if there are no non-NULL values in a column, then the result of the aggregation is NULL.

Example: Effect of NULL's

```
SELECT count(*)  
FROM Sells  
WHERE beer = 'Bud';
```

← The number of bars
that sell Bud.

```
SELECT count(price)  
FROM Sells  
WHERE beer = 'Bud';
```

← The number of bars
that sell Bud at a
known price.

Grouping

- ▶ We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- ▶ The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.

Example: Grouping

- ▶ From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer;
```

Example: Grouping

- ▶ From Sells(bar, beer, price) and Frequent(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequent, Sells
WHERE beer = 'Bud' AND
      Frequent.bar = Sells.bar
GROUP BY drinker;
```

Compute
drinker-bar-
price of Bud
tuples first,
then group
by drinker.

Restriction on SELECT Lists With Aggregation

- ▶ If any aggregation is used, then each element of the SELECT list must be either:
 1. Aggregated, or
 2. An attribute on the GROUP BY list.

Illegal Query Example

- ▶ You might think you could find the bar that sells Bud the cheapest by:

```
SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Bud';
```

- ▶ But this query is illegal in SQL.
 - ▶ Why? Note bar is neither aggregated nor on the GROUP BY list.

HAVING Clauses

- ▶ **HAVING <condition>** may follow a **GROUP BY** clause.
- ▶ If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Requirements on HAVING Conditions

- ▶ These conditions may refer to any relation or tuple-variable in the FROM clause.
- ▶ They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
 1. A grouping attribute, or
 2. Aggregated.

Example: HAVING

- ▶ From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
```

Beer groups with at least 3 non-NULL bars and also beer groups where the manufacturer is Pete's.

```
HAVING COUNT(bar) >= 3 OR
```

```
beer IN (SELECT name
FROM Beers
WHERE manf = 'Pete's');
```

Beers manufactured by Pete's.