DBM1
Part 2: SQL

Vincent Primault

vincent.primault@insa-lyon.fr

Course outline

- ~~Databases fundamentals~~ **Done!**
- ~~Relational algebra~~ **Done!**
- SQL language **Today**
- Database internals
- Distributed databases & NoSQL

Sources of this lecture

- Stanford, CS145 – Introduction to Databases
- Prof. Jeff Ullman

- INSA Lyon, 3IF-MD – Modélisation des données ("*data modeling*")
- Prof. Jean-Marc Petit

SQL vs relational algebra

- We focus on the implementation of relational algebra in a practical language named SQL.
- No more distinction between relation schema and relations: we have tables.
- An SQL table is not a set of tuples, it is a bag of tuples.

- Say "what to do" rather than "how to do it".
- DBMS figures out the "best" way to execute a query. It is called query optimization.

SQL (Structured Query Language)

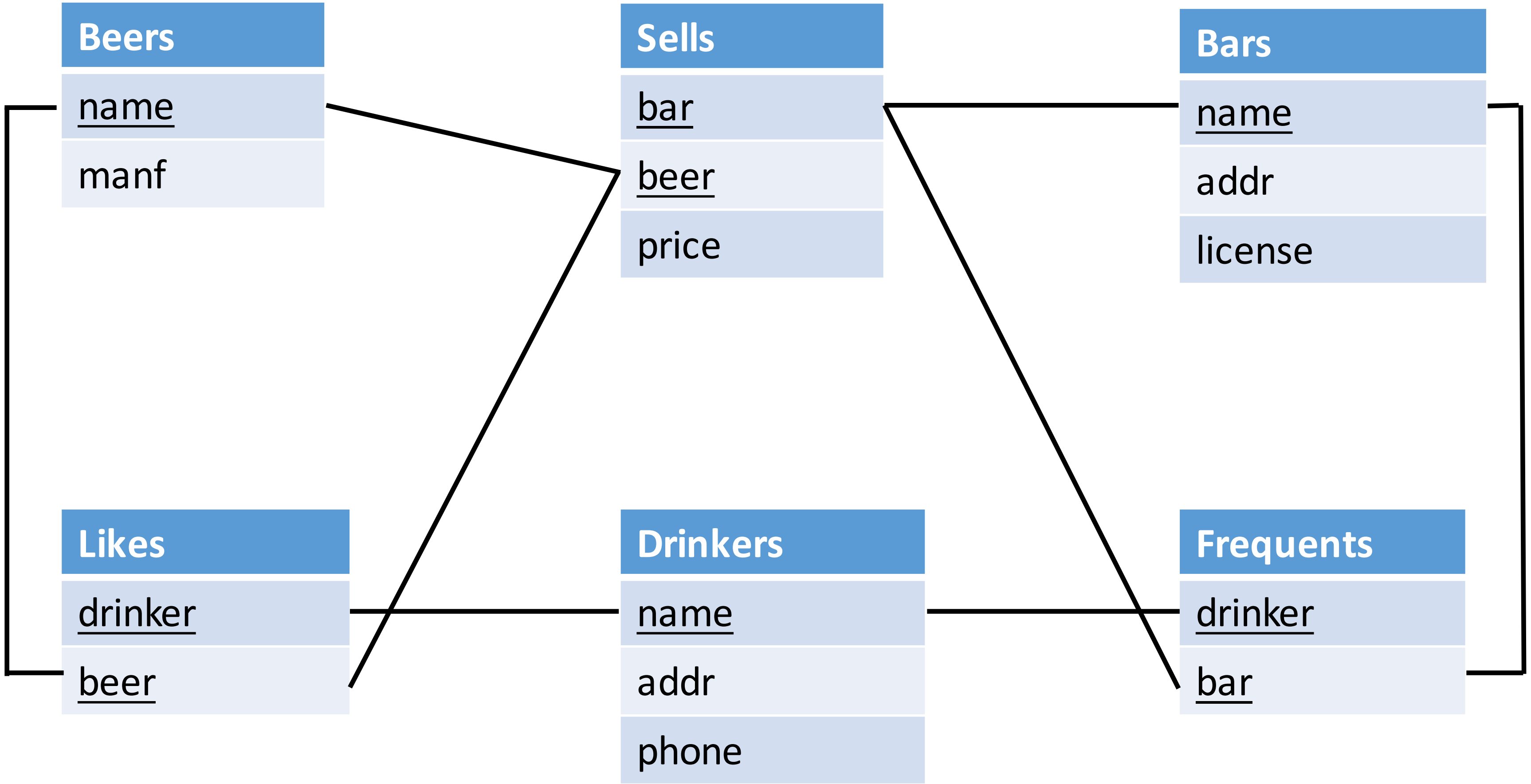
- Very popular data querying language, used almost everywhere.
- It is designed for traditional relational databases, but is also implemented in NoSQL/NewSQL databases.

- It is a concrete language built on several formal languages (including relational algebra), plus syntaxic sugar.
- Normalized by ANSI in 1992 (SQL-92) and in 1999 (SQL-99)…
- … but each DBMS has its own implementation for aspects not described in the norm. Always trust the DBMS' documentation!

Bags vs sets

- A bag (also called a multiset) is a generalization of the concept of set that allows multiple occurrences of the same element.
- Bags are needed for aggregate operations (e.g., sum, count).
- A bag is a function D → $\mathbb{N}$, giving for each possible element its multiplicity, i.e., the number of times it appears inside the bag.
  - bag = {a, b, b, d}
  - bag: {a, b, c, ..., z} → $\mathbb{N}$, bag(a) = 1, bag(b) = 2, bag(c) = 0, bag(d) = 1, etc.
- SQL operations are by default evaluated on bags.
  - Motivation is efficiency.

Our running example



Beers
- name
- manf

Sells
- bar
- beer
- price

Bars
- name
- addr
- license

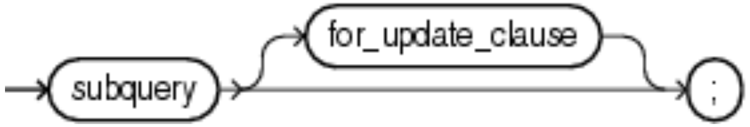
Likes
- drinker
- beer

Drinkers
- name
- addr
- phone

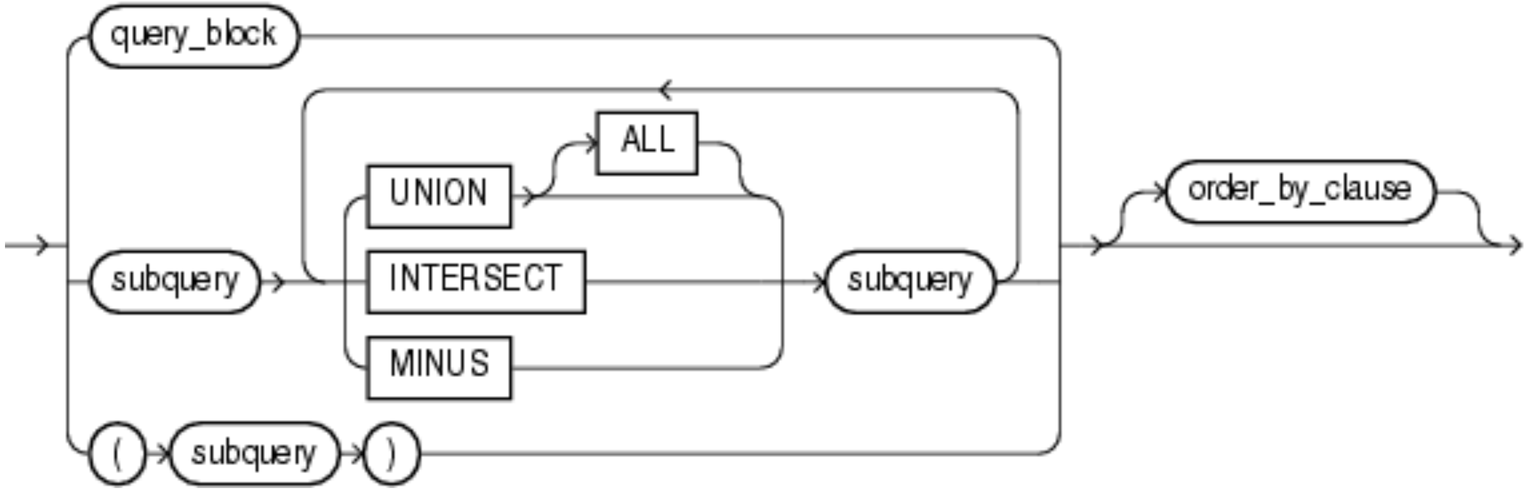
Frequents
- drinker
- bar

SELECT syntax

select ::=

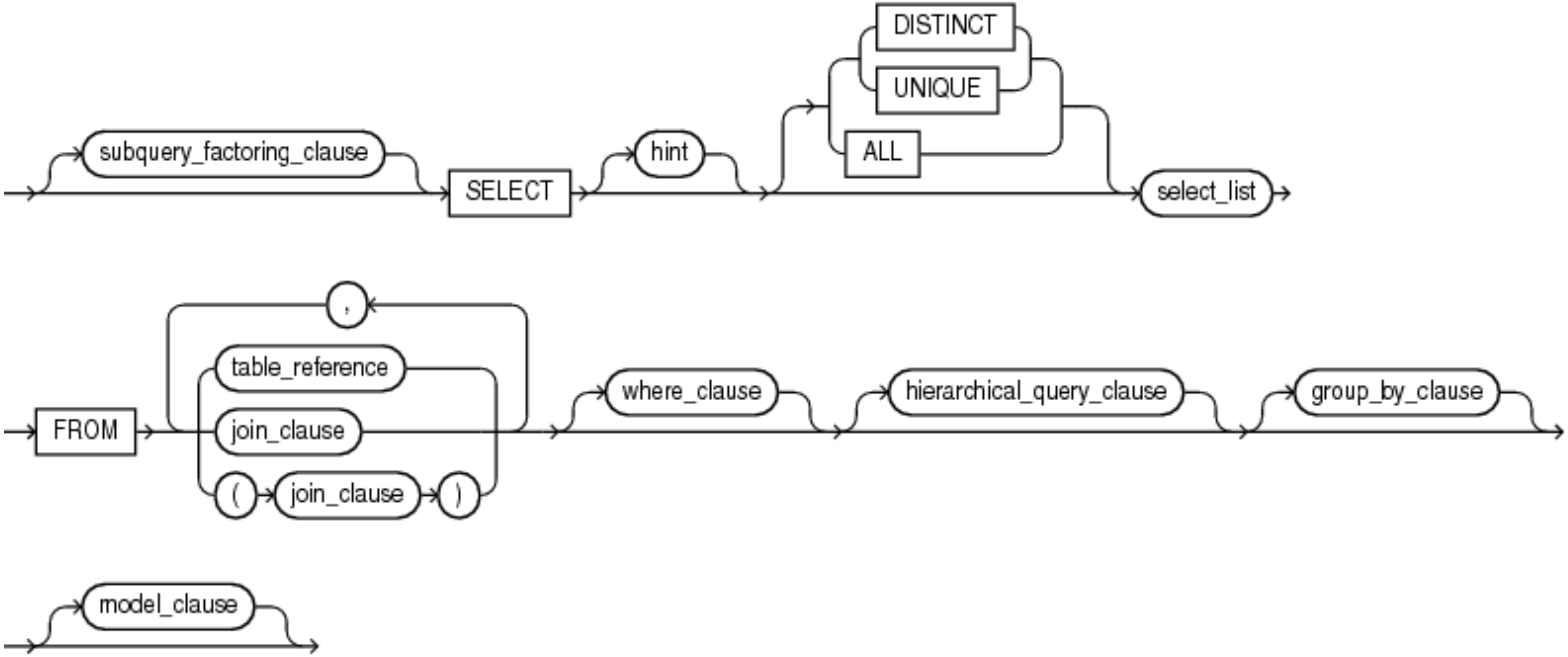


subquery ::=



SELECT syntax (cont'd)

query_block::=



SELECT syntax (cont’d)

```
[ subquery_factoring_clause ]
SELECT [ hint ] [ { { DISTINCT | UNIQUE } | ALL } ] select_list
FROM { table_reference | join_clause | ( join_clause ) }
        [ , { table_reference | join_clause | (join_clause) } ] ...
[ where_clause ]
[ hierarchical_query_clause ]
[ group_by_clause ]
[ model_clause ]
```

Single-table queries

DBM1 – Part 2: SQL

SELECT-FROM-WHERE statements

SELECT desired attributes

FROM one or many tables

[**WHERE** conditions about tuples of the tables];

Clauses

- SQL keywords are always case-insensitive, line breaks have no particular meaning.
- In Oracle, identifiers (e.g., table/attribute names) are by default case-insensitive.
- Queries should end with a semi-colon.

SELECT-FROM-WHERE statements (cont’d)

Which beers are made by Anheuser-Busch?

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

name
Bud
Bud Light
Bud Ice

A relation with a single attribute and tuples with the name of each beer.

SELECT-FROM-WHERE statements (cont'd)

Where and what beers can I find for strictly less than $5?

```
SELECT beer, bar
FROM Sells
WHERE price < 5;
```

beer	bar
Bud	Mikkeller Bar
Bud Ice	Mikkeller Bar
Coors Light	Pi Bar

* in SELECT clauses

- In the SELECT clause, * stands for "all attributes of this relation".
- It can be used instead of typing each one by hand, or if the names of the attributes is unknown.
- It should be used parsimoniously, it is usually better to ask only for the ones really needed.

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

name	manf
Bud	Anheuser-Busch
Bud Light	Anheuser-Busch
Bud Ice	Anheuser-Busch

Data types

- Values (either from tuples or constants) are typed.
- Usually, you only compare two things of the same type, e.g., a number with another number.
- It is possible to change the type of a value (it is called a cast).
- Constant values must be written in a canonical form fitting their type.

Thing	Oracle type	Example
Number	NUMBER	1234.56
String	VARCHAR2	‘I’’m a string’
Date	DATE	DATE ‘2015-09-31’
Date and time	DATETIME	TIMESTAMP ‘2015-10-01 08:00:00’

A (small) list of types available in Oracle

Building complex conditions

Operator	Meaning
a = b	*a* is equal to *b*
a <> b, a != b	*a* is not equal to *b*
a < b, a <= b	*a* is strictly less than *b*, *a* is less than *b*
a > b, a >= b	*a* is strictly greater than *b*, *a* is greater than *b*
a BETWEEN b AND c	$b \leq a \leq c$
a IN (b_1, b_2, ..., b_n)	*a* is equal to b_1, b_2, ..., or b_n
a IN (b_1, b_2, ..., b_n)	*a* is different from b_1, b_2, ..., and b_n
a LIKE b	*a* matches pattern *b*
a NOT LIKE b	*a* does not match pattern *b*

- *a* is usually an attribute, *b* another attribute or a constant value.
- Boolean operators AND, OR and NOT can be used to compose selections.

Pattern matching

- The **LIKE** operator is used to compare a string-valued attribute to some pattern.
- A pattern is a quoted string with placeholders:
    - % means "any string"
    - _ means "any single character"

Pattern matching (cont’d)

Who are the drinkers whose name starts with an "A"?

```
SELECT name
FROM Drinkers
WHERE name LIKE ‘A%’;
```

name
Arthur
Anna

Pattern matching (cont’d)

I can’t remember the name of a beer in starting with a B, an unknown letter and then a D.

```
SELECT name
FROM Beers
WHERE name LIKE ‘B_D%’;
```

name
Bud
Bud Light
Bud Ice

Building a complex query

Where can I find for at most $4.5 either a "light" beer or a Bud Ice?

bar
Mikkeller Bar
Pi Bar

```
SELECT bar
FROM Sells
WHERE (beer LIKE ‘%Light%’ OR beer = ‘Bud Ice’) AND price <= 4.5;
```

Debugging a query

Where can I find for at most $4.5 either a "light" beer or a Bud Ice?

A student wrote the following query. Is it correct?

```
SELECT bar
FROM Sells
WHERE beer LIKE ‘%Light%’ OR beer = ‘Bud Ice’ AND price <= 4.5;
```

Debugging a query (cont’d)

Where can I find for at most $4.5 either a "light" beer or a Bud Ice?

Can you spot the problem?

SELECT bar, beer, price
FROM Sells
~~**WHERE** beer **LIKE** ‘%Light%’~~
~~**OR** beer = ‘Bud Ice’ **AND** price <= 4.5;~~
WHERE beer **LIKE** (‘%Light%’ **OR** beer = ‘Bud Ice’) **AND** price <= 4.5;

bar	beer	price
Mikkeller Bar	Bud Light	5.0
Mikkeller Bar	Bud Ice	4.5
Pi Bar	Bud Light	5.0
Pi Bar	Coors Light	3.14
ISObeers	Bud Light	5.0
ISObeers	Coors Light	5.0

Renaming attributes

- If you want attributes in the result to have different names, use "**AS** <new name>" to give it a new name.

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = ‘Anheuser-Busch’;
```

beer	manf
Bud	Anheuser-Busch
Bud Light	Anheuser-Busch
Bud Ice	Anheuser-Busch

- The new name is only used for the resulting tuple, e.g., it cannot be used anywhere in the query, e.g., in the **WHERE** clause.

Ordering

- By default, the results are not sorted. This means the order in which tuples are returned is not predictable.
- It is possible to force an specific ordering by using one or many attributes.

```
SELECT select_list
FROM table_reference [ joins ]
[ WHERE conditions ]
[ ORDER BY  { attribute { ASC | DESC } } [, ...] ];
```

Ordering

Give the list of drinkers in alphabetical order.

```
SELECT name
FROM Drinkers
ORDER BY name;
```

name
Anna
Arthur
John

Ordering (cont’d)

Give tastes of drinkers in beers, ordered alphabetically by beer. If two beers the same, order results by drinker name in reverse alphabetic order.

```
SELECT beer, drinker
FROM Likes
ORDER BY  beer, drinker DESC;
```

beer	drinker
Bud Ice	John
Bud Light	John
Bud Light	Arthur
Bud Light	Anna
Coors Light	Arthur

Links with relation algebra so far

- **SELECT** encodes the projection operation.
- **WHERE** encodes the selection operation.
- **AS** encodes the renaming operation.

But…

- Pattern matching is not available in relational algebra.
- "SELECT *" is not available in relational algebra.
- **ORDER BY** is not available in relational algebra.

Multi-tables queries

DBM1 – Part 2: SQL

Inner joins

- Inner joins encompass joins we have studied in relational algebra.

```
SELECT select_list
FROM table_reference
[ {
  JOIN table_reference ON condition
  | NATURAL JOIN table_reference
  | CROSS JOIN table_reference
} [, ... ] ]
[ WHERE conditions ]
[ ORDER BY  attributes ];
```

← Theta-join

← Natural join

← Cross product

← Can be repeated

Cross join

List all pairs of beers and bars.

```
SELECT Beers.name, Bars.name
FROM Beers
CROSS JOIN Bars;
```

Resolves ambiguity about the attribute "name".

Other way to write it:

```
SELECT Beers.name, Bars.name
FROM Beers, Bars;
```

name	name_1
Bud	Mikkeller Bar
Bud	Pi Bar
Bud	ISObeers
Bud Light	Mikkeller Bar
Bud Light	Pi Bar
Bud Light	ISObeers
Bud Ice	Mikkeller Bar
Bud Ice	Pi Bar
Bud Ice	ISObeers
Hamm’s	Mikkeller Bar
Hamm’s	Pi Bar
Hamm’s	ISObeers
Coors Light	Mikkeller Bar
Coors Light	Pi Bar
Coors Light	ISObeers

Natural join

List bars in which drinkers can find beers they like.

SELECT *
FROM Likes
NATURAL JOIN Sells;

drinker	beer	bar	price
John	Bud Light	Mikkeller Bar	5.0
John	Bud Light	Pi Bar	5.0
John	Bud Light	ISObeers	5.0
John	Bud Ice	Mikkeller Bar	4.5
Arthur	Bud Light	Mikkeller Bar	5.0
Arthur	Bud Light	Pi Bar	5.0
Arthur	Bud Light	ISObeers	5.0
Arthur	Coors Light	Pi Bar	3.14
Arthur	Coors Light	ISObeers	5.0
Anna	Bud Light	Mikkeller Bar	5.0
Anna	Bud Light	Pi Bar	5.0
Anna	Bud Light	ISObeers	5.0

Theta-join

List drinkers' name and adress with beers they like.

```
SELECT name, addr, beer
FROM Drinkers
JOIN Likes ON name = drinker;
```

drinker	addr	beer
John	1200 Mission St...	Bud
John	1200 Mission St...	Bud Ice
Arthur	48 Folsom St...	Bud Light
Arthur	48 Folsom St...	Coors Light
Anna	134 E Julian St...	Bud Light

Theta-join (cont’d)

Why do we have duplicates?

List prices at which drinkers can buy beers they like, with their manufacturer.

SELECT drinker, Likes.beer, manf, price
FROM Likes
JOIN Beers **ON** Beers.name = Likes.beer
JOIN Sells **ON** Beers.name = Sells.beer;

drinker	beer	manf	price
John	Bud Light	A-B	5.0
John	Bud Light	A-B	5.0
John	Bud Light	A-B	5.0
John	Bud Ice	A-B	4.5
Anna	Bud Light	A-B	5.0
Anna	Bud Light	A-B	5.0
Anna	Bud Light	A-B	5.0
Arthur	Bud Light	A-B	5.0
Arthur	Bud Light	A-B	5.0
Arthur	Bud Light	A-B	5.0
Arthur	Coors Light	MC	5.0
Arthur	Coors Light	MC	3.14

Bridges between bags and sets

- By default, queries are evaluated on bags, which means duplicate row can be returned.
- It is possible to force the result to be a set by starting the query with **SELECT DISTINCT** ...
- Forcing a set has a cost, it can trigger a sort to deduplicate data.

Theta-join (cont’d)

drinker	beer	manf	price
John	Bud Light	A-B	5.0
John	Bud Ice	A-B	4.5
Anna	Bud Light	A-B	5.0
Arthur	Bud Light	A-B	5.0
Arthur	Coors Light	MC	5.0
Arthur	Coors Light	MC	3.14

List prices at which drinkers can buy beers they like, with their manufacturer.

```
SELECT DISTINCT drinker, Likes.beer, manf, price
FROM Likes
JOIN Beers ON Beers.name = Likes.beer
JOIN Sells ON Beers.name = Sells.beer;
```

Theta-join (cont’d)

List drinkers’ name and phone with bars they visit selling beers at $4.5 or less.

name	phone	bar
John	415-123-4567	Mikkeller Bar
John	415-123-4567	Pi Bar
Arthur	415-482-0312	Mikkeller Bar
Anna	408-127-8205	Pi Bar

```
SELECT DISTINCT Drinkers.name, Drinkers.phone, Frequents.bar
FROM Drinkers
JOIN Frequents ON Frequents.drinker = Drinkers.name
JOIN Sells ON Frequents.bar = Sells.bar
WHERE price <= 4.5;
```

Self-join

Give all pairs of different beers produced by the same manufacturer.
Beers inside the pair must be in alphabetic order.

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

Renames relations to distinguish between them

name	name_1
Bud	Bud Light
Bud	Bud Ice
Bud Ice	Bud Light
Coors Light	Hamm’s

Subqueries

- The result of a SELECT query can be used in the FROM or WHERE clause of another query.
- This is possible because the result of a SELECT query behaves itself as a new (virtual) table.
- Subqueries have a cost, use them when you really need to!

Subqueries (cont’d)

Which beers are manufactured by the same brewery than the Bud?

```
SELECT name
FROM Beers
WHERE manf = (
    SELECT manf
    FROM Beers
    WHERE name = ‘Bud’
) AND name <> ‘Bud’;
```

name
Bud Light
Bud Ice

Subqueries (cont’d)

Who visits a bar serving a Bud?

```
SELECT DISTINCT drinker
FROM Frequents
WHERE bar IN (
    SELECT bar
    FROM Sells
    WHERE beer = ‘Bud’
);
```

name
Arthur
John

Limiting results with a subquery

With Oracle, a virtual attribute named "rownum" (starting from 1) is created for each row **after** having processed the FROM/WHERE clauses and **before** processing all other clauses.

Who is the first drinker in alphabetic order?

```
SELECT *
FROM (
   SELECT name
   FROM Drinkers
   ORDER BY name
) WHERE rownum = 1;
```

name
Anna

Limiting results with a subquery (cont’d)

The following query does **not** return what is expected:

```
SELECT name
FROM Drinkers
WHERE rownum <= 5
ORDER BY name;
```

Because the value of rownum is determined after processing the WHERE clause but before processing the ORDER BY clause, it returns 5 random drinkers ordered by their name.

Set operations

- Union, intersection and difference between to subqueries are expressed with the **UNION**, **INTERSECT** and **MINUS** operators.
- By default, when these operators are used, the evaluation of a query is done on sets (and not bags)!
  - Motivation is, again, efficiency.
- You can force the evaluation on bags by using the keyword **ALL** after the operator name.

Set operations (cont'd)

drinker	beer
John	Bud Ice
Anna	Bud Light
Arthur	Bud Light
John	Bud Light

Give the drinkers and beers they like they can buy in a bar they frequent.

```
SELECT drinker, beer
FROM Likes
INTERSECT
SELECT drinker, beer
FROM Frequents
JOIN Sells ON Sells.bar = Frequents.bar;
```

Beers drinkers like

Beers available in bars drinkers frequent

Intersection equivalence

Give the drinkers and beers they like they can buy in a bar they frequent.

drinker	beer
John	Bud Ice
Anna	Bud Light
Arthur	Bud Light
John	Bud Light

```
SELECT DISTINCT Likes.drinker, Likes.beer
FROM Likes
JOIN Frequents ON Likes.drinker = Frequents.drinker
JOIN Sells ON Sells.bar = Frequents.bar
WHERE Sells.beer = Likes.beer;
```

Union equivalence

Give the light beers and those manufactured by MillerCoors.

Write them as a relational algebra expression tree. Which one is the more efficient?

```
SELECT name FROM Beers WHERE manf = ‘MillerCoors’
UNION
SELECT name FROM Beers WHERE name LIKE ‘%Light%’;
```

```
SELECT name
FROM Beers
WHERE manf = ‘MillerCoors’ OR name LIKE ‘%Light%’;
```

name
Bud Light
Hamm’s
Coors Light

EXISTS

- The operator **EXISTS** (<subquery>) is true iff the subquery's result is not empty, i.e., it contains at least one tuple.

```
SELECT attributes
FROM table
WHERE EXISTS (SELECT * FROM S WHERE C)
```

- The condition in C must involve at least one attribute of S.
- Rename tables if there is ambiguity about which table is involved.

EXISTS (cont’d)

Give the beers that are not liked by anyone.

```
SELECT name
FROM Beers
WHERE NOT EXISTS (
    SELECT *
    FROM Likes
    WHERE beer = name
);
```

name
Bud
Hamm’s

ANY

- The operator **ANY** is a generalization of the **IN** operator. It allows to express queries of the form "there exists...".

- *x* = **ANY**(<subquery>) is true iff there exists at least one tuple in the subquery's result that is equal to *x*.
  - = can be replaced by any boolean comparison operator.
  - *x* >= ANY(<subquery>) means there exist at least one tuple in the subquery's result that is strictly smaller than *x*.

ANY (cont'd)

Give beers that are sold and liked by at least one person.

```
SELECT DISTINCT beer
FROM Sells
WHERE beer = ANY(
    SELECT beer
    FROM Likes
);
```

name
Bud Ice
Bud Light
Coors Light

ALL

- The operator **ALL** allows to express queries of the form "for every...".

- *x* > **ALL**(<subquery>) is true iff *x* is greater than every tuple in the subquery’s result.
  - > can be replaced by any boolean comparison operator.
  - *x* <> **ANY**(<subquery>) *x* is not in the subquery’s result. It is actually equivalent to **NOT IN**.

ALL (cont’d)

Find the beer(s) sold at the highest price.

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

beer	bar
Hamm’s	ISObeers

Outer joins & Aggregation

DBM1 – Part 2: SQL

NULL values

- Tuples in SQL tables can have NULL as a value for one or more attributes.
- The database architect can choose to allow or not NULL values on a per-attribute basis.
- The meaning depends on the context. Two common cases are:
  - *Missing value:* e.g., we know Joe’s Bar has an address, but we do not know what this address is.
  - *Inapplicable:* e.g., the value of an attribute "spouse" for an unmarried person.

Comparing NULLs and values

- The logic in SQL is a 3-valued logic: *True*, *False* and *Unknown*.
- Comparing any value (including NULL itself) with a NULL yields an *Unknown* result.
- A tuple is in the result of a query iff the WHERE clause is *True* (thus not *False* or *Unknown*).
- It is possible to check if a value is null or not null with the operators **IS NULL** and **IS NOT NULL**.

3-valued logic

- To understand how **AND**, **OR** and **NOT** work in 3-valued logic, think of:
  - *True* = 1, *False* = 0,*Unknown* = ½
  - **AND** = min,
  - **OR** = max
  - **NOT** x= 1 – x
- Examples:
  - *True* **AND** *Unknown* = min(1, ½) = ½
  - *True* **OR** *Unknown* = max(1, ½) = 1
  - True **AND** (False **OR NOT** Unknown) = min(1, max(0, 1-½)) = min(1, max(0, ½)) = min(1, ½) = ½.

Outer joins

- A theta-join R $\bowtie_{\theta}$ S only return tuples of R that can be associated with a tuple of S.
- A tuple of R that has no tuple of S with which it joins is said to be dangling. And reciprocally for a tuple of S.
- Outer joins preserves dangling tuples by padding them with NULL values.
- You must specify a padding mode:
  - A left (outer) join pads dangling tuples of R only.
  - A right (outer) join pads dangling tuples of S only.
  - A full (outer) join pads both dangling tuples.

Outer joins (cont’d)

```
SELECT select_list
FROM table_reference
[ {
   { LEFT | RIGHT | FULL } JOIN table_reference ON condition
} [, … ] ]
[ WHERE conditions ]
[ ORDER BY  attributes ];
```

Outer joins (cont'd)

Give the list of all beers and people who like them.

```
SELECT name AS beer, drinker
FROM Beers
LEFT JOIN Likes ON beer = name;
```

Equivalent to:

```
SELECT name AS beer, drinker
FROM Likes
RIGHT JOIN Beers ON beer = name;
```

beer	drinker
Bud	*NULL*
Bud Ice	John
Bud Light	Anna
Bud Light	Arthur
Bud Light	John
Coors Light	Arthur
Hamm's	*NULL*

Outer joins (cont'd)

Give the list of all beers with name and address of drinkers who like them.

```
SELECT Beers.name AS beer, drinker, addr
FROM Beers
LEFT JOIN Likes ON beer = name
LEFT JOIN Drinkers ON Likes.drinker = Drinkers.name;
```

beer	drinker	addr
Bud	*NULL*	*NULL*
Bud Ice	John	1200 Mission St...
Bud Light	Anna	134 E Julian St...
Bud Light	Arthur	1432 Valancia St...
Bud Light	John	1200 Mission St...
Coors Light	Arthur	1432 Valancia St...
Hamm's	*NULL*	*NULL*

Basic aggregations

- Compute statistics on values of a relation. Common operations on numeric values include SUM, AVG, COUNT, MIN and MAX.
- COUNT(*) or COUNT(1) can be used to count the number of tuples.
- NULL values are ignored and never contribute to an aggregate.

What is the average, minimum and maximum price of Coors Light?

```
SELECT AVG(price), MIN(price), MAX(price)
FROM Sells
WHERE beer = ‘Coors Light’;
```

AVG(price)	MIN(price)	MAX(price)
4.07	3.14	5

Eliminating duplications in an aggregation

- Like in every SELECT query, aggregations work on bags.
- **DISTINCT** inside an aggregation is used to apply the opration on unique values.

What is the number of different prices charged for Bud Light?

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = ‘Bud Light’;
```

COUNT(DISTINCT price)
1

Grouping

- Previously, aggregation operations where applied on an entire relation and a single tuple was returned.
- We may want to perform the aggregation on sub-groups of a relation and get a tuple for each, e.g., get the average price of each beer.

```
SELECT select_list
FROM table_reference [ joins ]
[ WHERE conditions ]
[ GROUP BY  attributes ]
[ HAVING conditions ]
[ ORDER BY attributes ];
```

Grouping (cont’d)

What is the average price of each beer?

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

beer	AVG(price)
Bud	4.5
Bud Ice	4.5
Bud Light	5
Coors Light	4.07
Hamm’s	5.25

Grouping (cont’d)

The following query is invalid and should yield an error. Can you guess why?

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

Filtering by aggregates

Because the aggregates are computed **after** the selection is done, the following query is invalid:

```
SELECT beer, AVG(price)
FROM Sells
WHERE AVG(price) > 3
GROUP BY beer;
```

This is what the **HAVING** clause is intended be used for.

Filtering by aggregates (cont’d)

What beers have an average price greater than $5?

```
SELECT beer
FROM Sells
GROUP BY beer
HAVING AVG(price) >= 5;
```

beer
Bud Light
Hamm’s

Filtering by aggregates (cont’d)

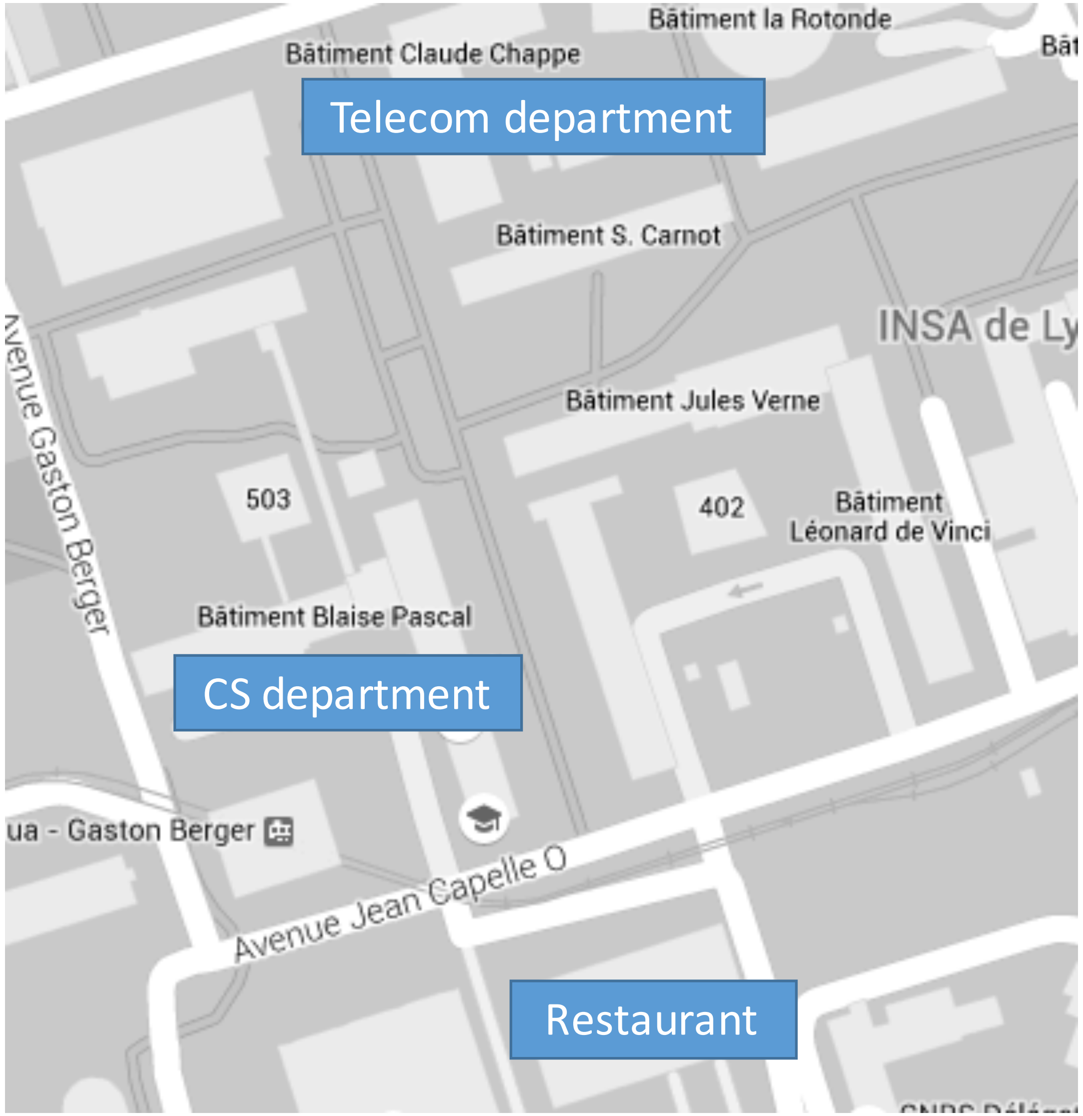
Give the drinkers visiting more at least two different bars and the number of different beers they like.

```
SELECT name, COUNT(DISTINCT beer)
FROM Drinkers
JOIN Likes ON name = Likes.Drinker
JOIN Frequents ON name = Frequents.Drinker
GROUP BY name
HAVING COUNT(DISTINCT bar) > 1;
```

name	COUNT(DISTINCT beer)
Anna	1
John	2

Next time: Practice

- 4h of lab work (evaluated)
- CS department, 2nd floor.


Bâtiment la Rotonde
Bât
Bâtiment Claude Chappe
Telecom department
Bâtiment S. Carnot
INSA de Ly
Avenue Gaston Berger
Bâtiment Jules Verne
503
402
Bâtiment Léonard de Vinci
Bâtiment Blaise Pascal
CS department
ua - Gaston Berger
Avenue Jean Capelle O
Restaurant