Basic SQL Queries

Why SQL?

- SQL is a very-high-level language
  - Say "what to do" rather than "how to do it"
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java
- Database management system figures out "best" way to execute query
  - Called "query optimization"

Select-From-Where Statements

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 Albani Bryggerierne?

```
SELECT name
  FROM Beers
  WHERE manf = 'Albani';
```

Result of Query

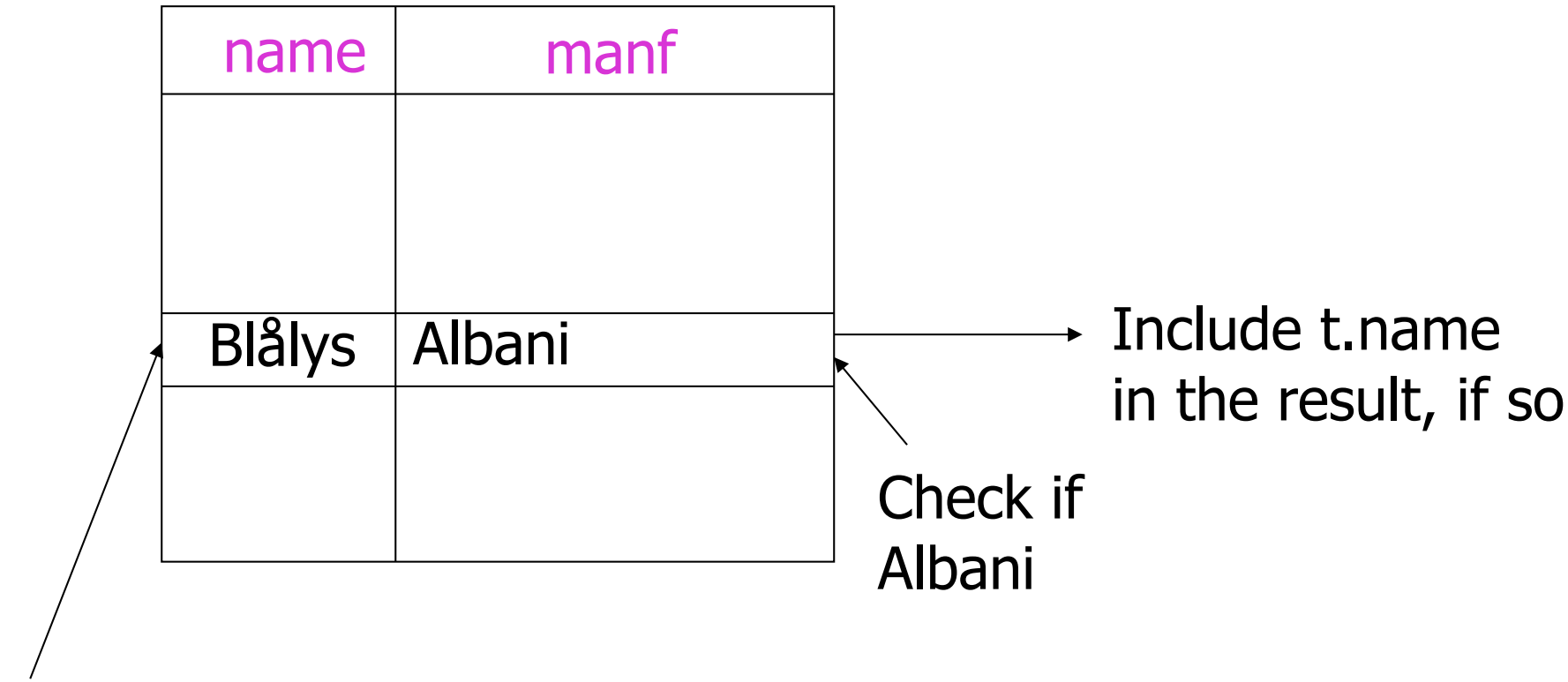
name
Od. Cl.
Eventyr
Blålys
. . .

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Albani Bryggerierne, such as Odense Classic.

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



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

* 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 = 'Albani';
```

Result of Query:

name	manf
Od.Cl.	Albani
Eventyr	Albani
Blålys	Albani
. . .	. . .

Now, the result has each of the attributes of Beers

Renaming Attributes

- If you want the result to have different attribute names, use "AS <new name>" to rename an attribute
- Example: Using Beers(name, manf):

```
SELECT name AS beer, manf
  FROM Beers
  WHERE manf = 'Albani'
```

Result of Query:

beer	manf
Od.Cl.	Albani
Eventyr	Albani
Blålys	Albani
. . .	. . .

Expressions in SELECT Clauses

- Any expression that makes sense can appear as an element of a SELECT clause
- Example: Using Sells(bar, beer, price):

```
SELECT bar, beer,
    price*0.134 AS priceInEuro
FROM Sells;
```

Result of Query

bar	beer	priceInEuro
C.Ch.	Od.Cl.	2.68
C.Ch.	Er.Wei.	4.69
...	...	...

Example: Constants as Expressions

- Using Likes(drinker, beer):

```
SELECT drinker, ' likes Albani '
   AS whoLikesAlbani
FROM Likes
WHERE beer = 'Od.Cl.';
```

Result of Query

drinker	whoLikesAlbani
Peter	likes Albani
Kim	likes Albani
...	...

Example: Information Integration

- We often build "data warehouses" from the data at many "sources"
- Suppose each bar has its own relation Menu(beer, price)
- To contribute to Sells(bar, beer, price) we need to query each bar and insert the name of the bar

Information Integration

- For instance, at the Cafe Biografen we can issue the query:

```
SELECT 'Cafe Bio', beer, price
FROM Menu;
```

Complex Conditions in WHERE Clause

- Boolean operators AND, OR, NOT
- Comparisons =, <>, <, >, <=, >=
  - And many other operators that produce boolean-valued results

Example: Complex Condition

- Using Sells(bar, beer, price), find the price Cafe Biografen charges for Odense Classic:

```
SELECT price
  FROM Sells
  WHERE bar = 'Cafe Bio' AND
      beer = 'Od.Cl.';
```

Patterns

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern> or
    <Attribute> NOT LIKE <pattern>
- *Pattern* is a quoted string with
  % = "any string"
  _ = "any character"

Example: LIKE

- Using Drinkers(name, addr, phone) find the drinkers with address in Fynen:

```
SELECT name
FROM Drinkers
WHERE phone LIKE '%, 5___ %';
```

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 Cafe Chino 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
- Comparing any value (including NULL itself) with NULL yields UNKNOWN
- A tuple is in a query answer iff 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 = ½
- AND = MIN; OR = MAX; NOT(*x*) = 1-*x*
- Example:

TRUE AND (FALSE OR NOT(UNKNOWN)) =
MIN(1, MAX(0, (1 - ½))) =
MIN(1, MAX(0, ½)) = MIN(1, ½) = ½

Surprising Example

- From the following Sells relation:

bar	beer	price
C.Ch.	Od.Cl.	NULL

SELECT bar

FROM Sells

WHERE price < 20 OR price >= 20;

2-Valued Laws != 3-Valued Laws

- Some common laws, like commutativity of AND, hold in 3-valued logic
- But not others, e.g., the *law of the excluded middle:* *p* OR NOT *p* = TRUE
  - When *p* = UNKNOWN, the left side is MAX(½, (1 – ½)) = ½ != 1

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: Joining Two Relations

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents C. Ch.

```
SELECT beer
FROM Likes, Frequents
WHERE bar = 'C.Ch.' AND
   Frequents.drinker =
                  Likes.drinker;
```

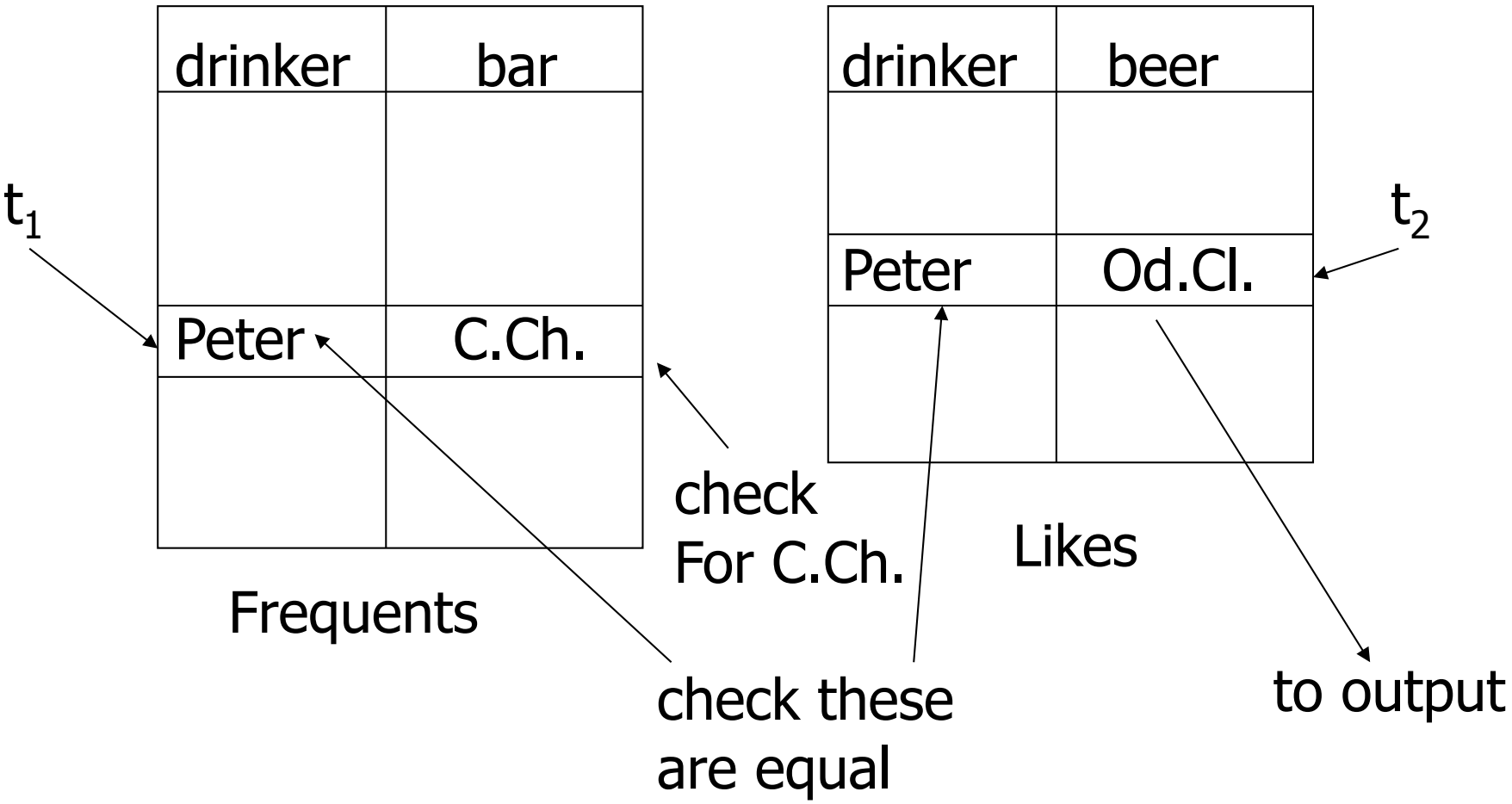
Formal Semantics

- 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


drinker
bar
Peter
C.Ch.
Frequents
drinker
beer
Peter
Od.Cl.
Likes
t1
t2
check
For C.Ch.
check these
are equal
to output

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 (Od.Cl., Od.Cl.)
  - Produce pairs in alphabetic order, e.g., (Blålys, Eventyr), not (Eventyr, Blålys)

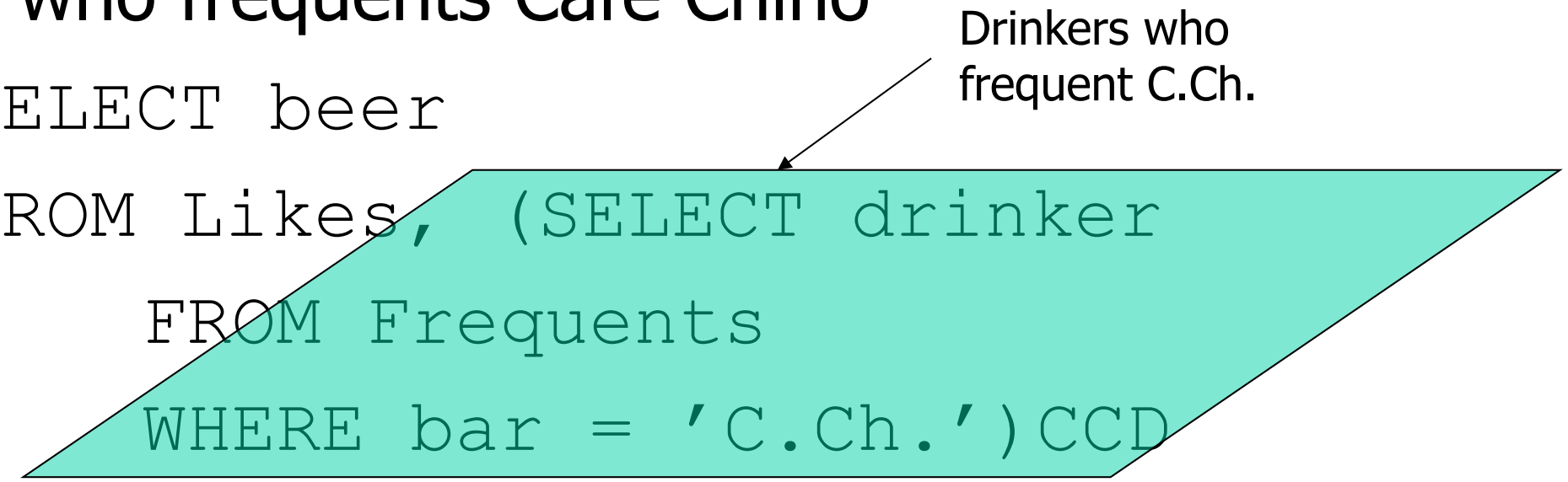
```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND
    b1.name < b2.name;
```

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 use a subquery and then query its result
  - Must use a tuple-variable to name tuples of the result

Example: Subquery in FROM

- Find the beers liked by at least one person who frequents Cafe Chino




```
SELECT beer
FROM Likes, (SELECT drinker
     FROM Frequents
     WHERE bar = 'C.Ch.')CCD
WHERE Likes.drinker = CCD.drinker;
```

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
  - A run-time error occurs if there is no tuple or more than one tuple

Example: Single-Tuple Subquery

- Using Sells(bar, beer, price), find the bars that serve Pilsener for the same price Cafe Chino charges for Od.Cl.
- Two queries would surely work:
  1. Find the price Cafe Chino charges for Od.Cl.
  2. Find the bars that serve Pilsener at that price

Query + Subquery Solution

```
SELECT bar
FROM Sells
WHERE beer = 'Pilsener' AND
  price = (SELECT price
           FROM Sells
           WHERE bar = 'Cafe Chino'
             AND beer = 'Od.Cl.');
```

The price at Which C.Ch. sells Od.Cl.

The IN Operator

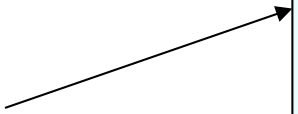
- <tuple> IN (<subquery>) is true if and only if the tuple is a member of the relation produced by the subquery
  - Opposite: <tuple> NOT IN (<subquery>)
- IN-expressions can appear in WHERE clauses

Example: IN

- Using Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Peter likes

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

The set of Beers Peter likes



What is the difference?

```
R(a,b); S(b,c)

SELECT a
FROM R, S
WHERE R.b = S.b;

SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

IN is a Predicate About R's Tuples

```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

Two 2's

One loop, over the tuples of R

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) satisfies the condition; 1 is output once

This Query Pairs Tuples from R, S

```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

Double loop, over the tuples of R and S

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) with (2,5) and (1,2) with (2,6) both satisfy the condition;
1 is output twice

The Exists Operator

- EXISTS(<subquery>) is true if and only if the subquery result is not empty
- Example: From Beers(name, manf), find those beers that are the unique beer by their manufacturer

Example: EXISTS

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

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

Notice the SQL "not equals" operator

The Operator ANY

- *x* = ANY(<subquery>) is a boolean condition that is true iff *x* equals at least one tuple in the subquery result
  - = could be any comparison operator.
- Example: *x* >= ANY(<subquery>) means *x* is not the uniquely smallest tuple produced by the subquery
  - Note tuples must have one component only

The Operator ALL

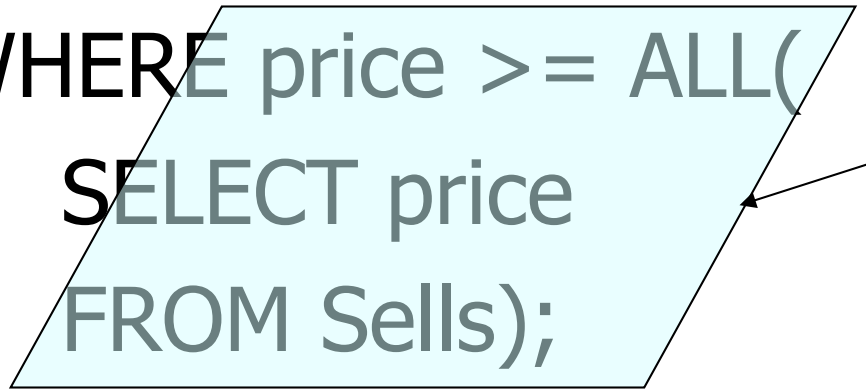
- *x* <> ALL(<subquery>) is true iff for every tuple *t* in the relation, *x* is not equal to *t*
  - That is, *x* is not in the subquery result
- <> can be any comparison operator
- Example: *x* >= ALL(<subquery>) means there is no tuple larger than *x* in the subquery result

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.



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 (<subquery>)

Example: Intersection

- Using Likes(drinker, beer), Sells(bar, beer, price), and Frequents(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.

The drinker frequents a bar that sells the beer.

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

Bag 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, it is easier to avoid eliminating duplicates
  - Just work tuple-at-a-time
- For 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 several versions of (bag) joins
- 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*

Summary 3

More things you should know:

- SELECT FROM WHERE statements with one or more tables
- Complex conditions, pattern matching
- Subqueries, natural joins, theta joins

Extended Relational Algebra

The Extended Algebra

δ = eliminate duplicates from bags

τ = sort tuples

γ = grouping and aggregation

Outerjoin: avoids "dangling tuples" = tuples that do not join with anything

Duplicate Elimination

- $R_1 := \delta(R_2)$
- R_1 consists of one copy of each tuple that appears in R_2 one or more times

Example: Duplicate Elimination

R =

A	B
1	2
3	4
1	2

δ(R) =

A	B
1	2
3	4

Sorting

- $R_1 := \tau_L (R_2)$
  - L is a list of some of the attributes of R_2
- R_1 is the list of tuples of R_2 sorted lexicographically according to the attributes in L, i.e., first on the value of the first attribute on L, then on the second attribute of L, and so on
  - Break ties arbitrarily
- τ is the only operator whose result is neither a set nor a bag

Example: Sorting

R =

A	B
1	2
3	4
5	2

$\tau_B(R) = [(5,2), (1,2), (3,4)]$

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

Example: Aggregation

R =

A	B
1	3
3	4
3	2

SUM(A) = 7
COUNT(A) = 3
MAX(B) = 4
AVG(B) = 3

Grouping Operator

- $R_1 := \gamma_L (R_2)$
  L is a list of elements that are either:
  1. Individual (*grouping*) attributes
  2. AGG(A), where AGG is one of the aggregation operators and A is an attribute
     - An arrow and a new attribute name renames the component

Applying $\gamma_L(R)$

- Group *R* according to all the grouping attributes on list *L*
  - That is: form one group for each distinct list of values for those attributes in *R*
- Within each group, compute AGG(*A*) for each aggregation on list *L*
- Result has one tuple for each group:
  1. The grouping attributes and
  2. Their group's aggregations

Example: Grouping/Aggregation

R =

A	B	C
1	2	3
4	5	6
1	2	5

$\gamma_{A,B,AVG(C)->X}(R)$ = ??

First, group *R* by *A* and *B* :

A	B	C
1	2	3
1	2	5
4	5	6

Then, average *C* within groups:

A	B	X
1	2	4
4	5	6

Outerjoin

- Suppose we join $R \bowtie_C S$
- A tuple of R that has no tuple of S with which it joins is said to be *dangling*
  - Similarly for a tuple of S
- Outerjoin preserves dangling tuples by padding them NULL

Example: Outerjoin

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

(1,2) joins with (2,3), but the other two tuples are dangling

R OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

Summary 4

More things you should know:

- Duplicate Elimination
- Sorting
- Aggregation
- Grouping
- Outer Joins

Back to SQL

Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression
- It is modified by:
  1. Optional NATURAL in front of OUTER
  2. Optional ON <condition> after JOIN
  3. Optional LEFT, RIGHT, or FULL before OUTER
     - LEFT = pad dangling tuples of R only
     - RIGHT = pad dangling tuples of S only
     - FULL = pad both; this choice is the default

Only one of these

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 Odense Classic:

```
SELECT AVG(price)
  FROM Sells
  WHERE beer = 'Od.Cl.';
```

Eliminating Duplicates in an Aggregation

- Use DISTINCT inside an aggregation
- Example: find the number of *different* prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
  FROM Sells
  WHERE beer = 'Od.Cl.';
```

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
  - Exception: COUNT of an empty set is 0

Example: Effect of NULL's

```
SELECT count(*)
FROM Sells
WHERE beer = 'Od.Cl.';
```

The number of bars that sell Odense Classic

```
SELECT count(price)
FROM Sells
WHERE beer = 'Od.Cl.';
```

The number of bars that sell Odense Classic 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;
```

beer	AVG(price)
Od.Cl.	20
…	…

Example: Grouping

- From Sells(bar, beer, price) and Frequents(drinker, bar), find for each drinker the average price of Odense Classic at the bars they frequent:

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

Compute all drinker-bar-price triples for Odense Cl.

Then group them 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 Odense Cl. the cheapest by:

```
SELECT bar, MIN(price)
FROM Sells
WHERE beer = 'Od.Cl.';
```

- But this query is illegal in SQL

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

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 Albani Bryggerierne

Solution

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
    beer IN (SELECT name
             FROM Beers
             WHERE manf = 'Albani');
```

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

Beers manu-factured by Albani.

Requirements on HAVING Conditions

- Anything goes in a subquery
- Outside subqueries, they may refer to attributes only if they are either:
  1. A grouping attribute, or
  2. Aggregated

  (same condition as for SELECT clauses with aggregation)

Database Modifications

- A *modification* command does not return a result (as a query does), but changes the database in some way
- Three kinds of modifications:
  1. *Insert* a tuple or tuples
  2. *Delete* a tuple or tuples
  3. *Update* the value(s) of an existing tuple or tuples

Insertion

- To insert a single tuple:

  INSERT INTO <relation>

  VALUES (<list of values>);

- Example: add to Likes(drinker, beer) the fact that Lars likes Odense Classic.

```
INSERT INTO Likes
  VALUES('Lars', 'Od.Cl.');
```

Specifying Attributes in INSERT

- We may add to the relation name a list of attributes
- Two reasons to do so:
  1. We forget the standard order of attributes for the relation
  2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value

Example: Specifying Attributes

- Another way to add the fact that Lars likes Odense Cl. to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
VALUES('Od.Cl.', 'Lars');
```

Adding Default Values

- In a CREATE TABLE statement, we can follow an attribute by DEFAULT and a value
- When an inserted tuple has no value for that attribute, the default will be used

Example: Default Values

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

Example: Default Values

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

Resulting tuple:

name	address	phone
Lars	Vestergade	NULL

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

  INSERT INTO <relation>

  (<subquery>);

Example: Insert a Subquery

- Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Lars "potential buddies", i.e., those drinkers who frequent at least one bar that Lars also frequents

Solution

The other drinker

Pairs of Drinker tuples where the first is for Lars, the second is for someone else, and the bars are the same

```
INSERT INTO PotBuddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Lars' AND
   d2.drinker <> 'Lars' AND
   d1.bar = d2.bar
);
```

Deletion

- To delete tuples satisfying a condition from some relation:

  DELETE FROM <relation>

  WHERE <condition>;

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Lars likes Odense Classic:

```
DELETE FROM Likes
  WHERE drinker = 'Lars' AND
        beer = 'Od.Cl.';
```

Example: Delete all Tuples

- Make the relation Likes empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Some Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

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

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b

Semantics of Deletion

- Suppose Albani makes only Odense Classic and Eventyr
- Suppose we come to the tuple *b* for Odense Classic first
- The subquery is nonempty, because of the Eventyr tuple, so we delete Od.Cl.
- Now, when *b* is the tuple for Eventyr, do we delete that tuple too?

Semantics of Deletion

- Answer: we *do* delete Eventyr as well
- The reason is that deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied
  2. Delete the marked tuples

Updates

- To change certain attributes in certain tuples of a relation:

  UPDATE <relation>

  SET <list of attribute assignments>

  WHERE <condition on tuples>;

Example: Update

- Change drinker Lars's phone number to 47 11 23 42:

```
UPDATE Drinkers
  SET phone = '47 11 23 42'
  WHERE name = 'Lars';
```

Example: Update Several Tuples

- Make 30 the maximum price for beer:

```
UPDATE Sells
  SET price = 30
  WHERE price > 30;
```

Summary 4

More things you should know:

- More joins
  - OUTER JOIN, NATURAL JOIN
- Aggregation
  - COUNT, SUM, AVG, MAX, MIN
  - GROUP BY, HAVING
- Database updates
  - INSERT, DELETE, UPDATE