More on SQL

Juliana Freire

Some slides adapted from J. Ullman, L. Delcambre, R. Ramakrishnan, G. Lindstrom and Silberschatz, Korth and Sudarshan

Interpreting a Query

SELECT A1, A2, …, Am
FROM R1, R2, …, Rn
WHERE C1, C2, …, Ck

Translate to Relational Algebra

1. Start with the Cartesian 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.

Interpreting a Query

SELECT A1, A2, …, Am
FROM R1, R2, …, Rn
WHERE C1, C2, …, Ck

Nested loops

- 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.

Challenge Question

- Suppose R, S and T are unary relations, each having one attribute A. We want to compute R ∩ (S ∪ T).
- Does the following query do the job?

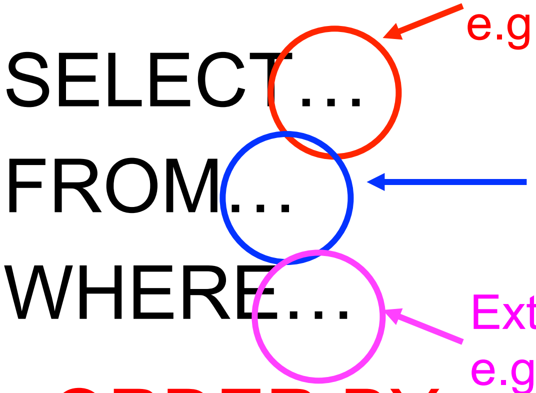
SELECT R.A

FROM R, S, T

WHERE R.A = S.A OR R.A = T.A

???

SQL … Extensions



SELECT…

FROM…

WHERE…

Extension to the SELECT clause
e.g., DISTINCT, SUM, COUNT, MIN, MAX, AVG and AS

Extension to the FROM clause
e.g., correlation names and various kinds of JOINs

Extension to the WHERE clause
e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY

ORDER BY…

GROUP BY…

HAVING …

Several additional clauses
e.g., ORDER BY, GROUP BY, and HAVING

(SELECT…FROM…WHERE…)
UNION
(SELECT…FROM…WHERE…)

And operators that expect two or more complete SQL queries as operands
e.g., UNION and INTERSECT

Sample Database

For this discussion, we will use the following database:

Customer (Number, Name, Address, CRating, CAmount, CBalance, RegisterDate, SalespersonNum)

Foreign key: Customer.SalespersonNum references Salesperson.Number

Salesperson(Number, Name, Address, Office)

Eliminating Duplicates

Consider the following two queries:

```
SELECT    DISTINCT Name
FROM      Customer;
```

Name
W. Wei
J. Smith

```
SELECT    Name
FROM      Customer;
```

Name
J. Smith
W. Wei
J. Smith

The first query eliminates duplicate rows from the answer.

- Although the relational model is based on set, by default RDBMSs operate on *multisets (bags)*
- *The query writer gets to choose whether duplicates are eliminated*

Eliminating Duplicates: A Word of Caution

- In theory, placing a DISTINCT after select is harmless
- In practice, it is very expensive
  - **The time it takes to sort a relation so that duplicates are eliminated can be greater than the time to execute the query itself!**

Use DISTINCT only when you really need it

Aggregates

- *Summarize or "aggregate" the values in a column*
- Operators: COUNT, SUM, MIN, MAX, and AVG
  - Apply to sets or bags of atomic values
- SUM and AVG: produce sum and average of a column with numerical values
- MIN and MAX:
  - applied to column with numerical values, produces the smallest and largest value
  - applied to column with character string values, produces the lexicographically first or last value
- COUNT: produces the number of values in a column
  - Equivalently the number of tuples in a relation, *including duplicates*

How is this query evaluated?

```
SELECT      AVG (CBalance)
FROM        Customer
WHERE       age > 35;
```

Aggregates and NULLs

- General rule: aggregates ignore NULL values
  - Avg(1,2,3,NULL,4) = Avg(1,2,3,4)
  - Count(1,2,3,NULL,4) = Count(1,2,3,4)
- But…
  - Count(*) returns the total number of tuples, regardless whether they contain NULLs or not

Aggregates and Duplicates

- Aggregates apply to bags
- If you want sets instead, use DISTINCT

```
SELECT COUNT(Name)
FROM   Customer;
```

Answer: 3

```
SELECT  COUNT(DISTINCT Name)
FROM    Customer;
```

Answer: 2

Name
J. Smith
W. Wei
J. Smith

Note: Full-Relation Operations

- DISTINCT and aggregates act on relations as a whole, rather than on individual tuples

- More on aggregates later!

SQL … Extensions

SELECT…

FROM…

WHERE…

Extension to the SELECT clause
e.g., DISTINCT, SUM, COUNT, MIN, MAX, AVG and AS

Extension to the FROM clause
e.g., correlation names, subqueries and various kinds of JOINs

Joins

There are a number of join types that can be expressed in the FROM clause:

- inner join (the theta join)
- cross join (Cartesian product)
- natural join
- left outer join
- right outer join
- full outer join
- union join

Joins

There are a number of join types that can be expressed in the FROM clause:

- inner join (the theta join)
- cross join
- natural join
- left outer join
- right outer join
- full outer join
- union join

These are syntactic sugar … they can be expressed in a basic SELECT..FROM..WHERE query.

Joins

There are a number of join types that can be expressed in the FROM clause:

- inner join (the regular join)
- cross join
- natural join
- left outer join
- right outer join
- full outer join
- union join

There are new operators … but can be expressed in a complex SQL query involving the union operator.

ON clause for the join

Join condition in the ON clause (vs. the WHERE clause)

These two queries are equivalent:

```
SELECT      C.Name, S.Name
FROM        Customer C JOIN Salesperson S
            ON C.SalespersonNum = S.Number
WHERE       C.CRating < 6;

SELECT      C.Name, S.Name
FROM        Customer C, Salesperson S
WHERE       C.SalespersonNum = S.Number AND
            C.CRating < 6;
```

Customer (Number, Name, Address, CRating, CAmount,
CBalance, RegisterDate, SalespersonNum)
Salesperson(Number, Name, Address, Office)

Basic Join ≡ INNER JOIN

Customer (Number, Name, Address, CRating, CAmount,
CBalance, RegisterDate, SalespersonNum)
Salesperson(Number, Name, Address, Office)

These queries are equivalent.

```
SELECT    C.Name, S.Name
FROM      Customer C JOIN Salesperson S
          ON SalespersonNum;
```

Note the use of foreign key to simplify expression

```
SELECT    C.Name, S.Name
FROM      Customer C INNER JOIN Salesperson S
          ON SalespersonNum;
```

For the INNER JOIN, the query answer does not include:
a Customer that doesn't have a Salesperson ….
or
a Salesperson that is not assigned to any customers.

Equijoin and Natural Join

Equijoin: Join condition has only equality

- Result will contain two attributes with identical values, perhaps different names

Natural join: Equijoin on attributes with same names

- No need to specify join attributes
- Result will contain each join attribute only once

What if there are no common attribute names?

- In that case, Natural Join ≡ Cross Product

Natural Join

- Joins attributes with same name and eliminates one of them from the result

```
SELECT        *
FROM   Customer NATURAL JOIN SalesPerson;
```

How can we write an equivalent query without using a join clause?

Customer (Number, Name, Address, CRating, CAmount, CBalance, SalespersonNum)

Salesperson (Number, Name, Address, Office)

Natural Join

Original query:

```
SELECT          *
FROM            Customer C NATURAL JOIN SalesPerson S;
```

Equivalent query:

```
SELECT C.Number, C.Name, C.Address, C.CRating,
                C.CAmount, C.CBalance,
       C.SalespersonNum, S.Office
FROM            Customer C, Salesperson S
WHERE C.Number = S.Number and C.Name = S.Name,
                          and C.Address = S.Address;
```

Project out repeated columns

Customer (<u>Number</u>, Name, Address, CRating, CAmount, CBalance, SalespersonNum)

Salesperson (<u>Number</u>, Name, Address, Office)

BTW, what does this query compute?

Natural Join: Some Notes

equivalent

```
SELECT      *
FROM        Customer C, Salesperson S
WHERE       C.SalespersonNum = S.Number;

SELECT      *
FROM        Customer C JOIN Salesperson S on
                                  SalespersonNum;
```

```
SELECT      *
FROM        Customer NATURAL JOIN Salesperson
```

this query is not equivalent to above two queries, why?

Customer (Number, Name, Address, CRating, CAmount, CBalance, SalespersonNum)

Salesperson (Number, Name, Address, Office)

How would you write the following query?

- Student(sid, name, address)
- Spouse(sid, name), *sid references Student.sid*
- List the names of all students and their spouses, if they have one.

```
SELECT Student.name,Spouse.name
FROM Student, Spouse
WHERE Student.sid=Spouse.sid
```

- Does this SQL query do the job?

How would you write the following query?

- Student(sid, name, address)
- Spouse(sid, name), *sid references Student.sid*
- List the names of all students and their spouses, if they have one.

```
SELECT Student.name,Spouse.name
FROM Student, Spouse
WHERE Student.sid=Spouse.sid
```

- Does this SQL query do the job?

  No! Students without spouses will *not* be listed.

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses *null* values to pad *dangling tuples*

LEFT OUTER JOIN

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

INNER JOIN on C.SalespersonNum = S.Number gives us:

“smith” with “johnson” and “jones” with “johnson”

LEFT OUTER JOIN on C.SalespersonNum = S.Number gives us:

INNER JOIN plus “wei” with “<null>” salesperson

- Lists all customers, and their salesperson if any

RIGHT OUTER JOIN

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

INNER JOIN on C.SalespersonNum = S.Number gives us:
"smith" with "johnson" and "jones" with "johnson"

RIGHT OUTER JOIN on C.SalespersonNum = S.Number gives us:
INNER JOIN plus "<null>" customer with "miller"

- Lists customers that have a salesperson, and salespersons that do not have a customer

FULL OUTER JOIN

FULL OUTER JOIN = LEFT OUTER JOIN ∪ RIGHT OUTER JOIN

FULL OUTER JOIN on C.SalespersonNum = S.Number gives us:

INNER JOIN
plus “wei” with “<null>” salesperson
plus “<null>” customer with “miller”

- Lists all customer-salesperson pairs, and customers that do not have a salesperson, and salespersons that do not have a customer

CROSS JOIN

A "CROSS JOIN" is simply a cross product

```
SELECT *
FROM Customer CROSS JOIN Salesperson;
```

How would you write this query without the "CROSS JOIN" operator?

```
SELECT    *
FROM      Customer, Salesperson;
```

SQL … Extensions

SELECT…

FROM…

WHERE…

Extension to the SELECT clause
e.g., SUM, COUNT, MIN, MAX, AVG and AS

Extension to the FROM clause
e.g., correlation names and various kinds of JOINs

Extension to the WHERE clause
e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY

WHERE Clause: Comparison Operators

- <, >, =, <>, >=, <=
  - Compare two values as expected
  - Operates on numbers as well as text values
  - Amount < 50 and CustID <> 1
  - custName = ‘Juliana’ || ‘Freire’ (string concatenation)
- LIKE
  - Compare a text value with a pattern
  - ‘%’ compares with zero or more characters
  - ‘_’ compares with exactly one character
  - custName LIKE ‘%Fr__re’ – matches ‘Juliana Freire’, ‘Freire’, ‘Friere’

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.

Subqueries in the FROM clause

- The FROM clause takes a relation, but *results from SQL queries are themselves relations, so we can use them in the FROM clause*, too!

```
SELECT (N.CRating+1) AS CIncrRating
FROM (SELECT * FROM Customer WHERE CRating = 0) AS N
WHERE N.CBalance = 0
```

- This can often be a more elegant way to write a query, but will be slower. Why?
- Can this be written without nesting?

```
SELECT (CRating+1) AS CIncrRating
FROM Customer
WHERE CRating = 0 AND CBalance = 0
```

Subqueries in the WHERE clause

```
SELECT      C1.Number, C1.Name
FROM        Customer C1
WHERE       C1.CRating IN
                    (SELECT MAX (C2.CRating)
                     FROM   Customer C2,);
```

This is a complete query

Find all customers where their credit rating is equal to the highest credit rating that appears in the database.

To understand semantics of nested queries, think of a *nested loops* evaluation: for each customer tuple, check the qualification by computing the subquery

IN <subquery>: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

```
SELECT          C1.Number, C1.Name
FROM            Customer C1
WHERE           C1.CRating IN
                          (SELECT MAX (C2.CRating)
                           FROM    Customer C2;);
```

```
SELECT          C1.Number, C1.Name
FROM            Customer C1
WHERE           C1.CRating IN  {10}
```

Result: 3, wei

NOT IN <subquery>: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

```
SELECT       C1.Number, C1.Name
FROM         Customer C1
WHERE        C1.CRating NOT IN
                       (SELECT MAX (C2.CRating)
                        FROM    Customer C2;);
```

```
SELECT       C1.Number, C1.Name
FROM         Customer C1
WHERE        C1.CRating NOT IN  {10}
```

Result: ?

Conditions Involving Relations: IN and NOT IN

```
SELECT          C1.Number, C1.Name
FROM            Customer C1
WHERE           C1.CRating IN
                          (SELECT MAX (C2.CRating)
                           FROM    Customer C2;);
```

- <attribute-name A> IN (subquery S): tests set membership
  - A is equal to one of the values in S
- <attribute-name A> NOT IN (subquery S)
  - A is equal to no value in S

Conditions Involving Relations: EXISTS

- EXISTS R is true if R is not empty
- NOT EXISTS R is true if R is empty

```
SELECT   C.Name
FROM     Customer C
WHERE    EXISTS (SELECT     *
                 FROM       Salesperson S
                 WHERE      S.Number =
                     C.SalespersonNum;);
```

What does this query compute?

If the answer to the subquery is not empty - then the EXISTS predicate returns TRUE

Tests for empty relations

EXISTS: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office	
101	johnson	aaa	23	
	102 miller		bbb	26

```
SELECT   C.Name
FROM     Customer C
WHERE    EXISTS (SELECT   *
                 FROM     Salesperson S
                 WHERE    S.Number = C.SalespersonNum;);
```

Conditions involving relations: NOT EXISTS

- NOT EXISTS R is true if R is empty

```
SELECT   C.Name
FROM     Customer C
WHERE    NOT EXISTS (SELECT        *
                     FROM  Salesperson S
                      WHERE        S.Number =
                           C.SalespersonNum;);
```

What does this query compute?

If the answer to the subquery is empty -
then the NOT EXISTS predicate returns TRUE

Tests for non-empty relations

NOT EXISTS: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

```
SELECT   C.Name
FROM     Customer C
WHERE    NOT EXISTS (SELECT      *
                     FROM       Salesperson S
                     WHERE      S.Number = C.SalespersonNum;);
```

Set comparison: ALL or ANY in a Subquery

- Syntax:
  - attribute-name *comparator* ALL (subquery)
  - attribute-name *comparator* ANY (subquery)
- A > ALL (subquery S):
  - True if A is greater than every value returned by S
  - (A <> ALL S) ≡ (A NOT IN S)
- A > ANY (subquery S)
  - True if A is greater than at least one value returned by S
  - (A = ANY S) ≡ (A IN S)

ALL or ANY in a Subquery: Example

```
SELECT  S.Number, S.Name
FROM    Salesperson S
WHERE   S.Number = ALL (SELECT C.SalespersonNum
                        FROM    Customer;);
```

This predicate must be true for all SalespersonNums returned by the subquery!

What does this query compute?

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	101

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

ALL or ANY in a Subquery: Example

```
SELECT  C.Name
FROM    Customer C
WHERE   C.Crating >= ALL (SELECT C1.Crating
                          FROM    Customer C1);
```

What does this query compute?

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	101

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

Correlated Subqueries

- The simplest subqueries can be evaluated once and for all and the result used in a higher-level query
- More complicated subqueries must be evaluated once for each assignment of a value to a term in the subquery that comes from a tuple outside the subquery

```
SELECT  S.Number, S.Name
FROM    Salesperson S
WHERE   S.Number IN     (SELECT     C.SalespersonNum
                               FROM       Customer C
                               WHERE      C.Name = S.Name;);
```

Because the subquery mentions an attribute from a table in the outer query

Subquery that is not correlated

```
SELECT          C1.Number, C1.Name
FROM            Customer C1
WHERE           C1.CRating IN
                          (SELECT MAX (C2.CRating)
                           FROM    Customer C2;);
```

The subquery only uses attributes from the the table mentioned in the subquery

Correlated Subqueries: Scoping

- An attribute in a subquery belongs to one of the tuple variables corresponding to the *closest* relation
  - In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's FROM clause
  - If not, look at the immediately surrounding subquery, then to the one surrounding that, and so on.

Correlated Subqueries: Semantics

- Analyze from the inside out
  - For each tuple in the outer query, evaluate the innermost subquery, and replace that with the resulting relation
  - Repeat

```
SELECT S.Number, S.Name
FROM    Salesperson S
WHERE S.Number IN   (SELECT C.SalespersonNum
                     FROM    Customer C
                     WHERE  C.Name = S.Name;);
```

Correlated Subqueries: Semantics

```
SELECT S.Number, S.Name
FROM    Salesperson S
WHERE S.Number IN
```

Can’t evaluate, don’t know
The value for S.Name

```
(SELECT       C.SalespersonNum
        FROM          Customer C
        WHERE         C.Name = S.Name;);
```

As we range through the Salesperson tuples, each tuple provides a value for S.Name

De-correlate: another way to write this query:

```
SELECT S.Number, S.Name
FROM      Salesperson S, Customer C
WHERE S.Number = C.SalespersonNum AND
             C.Name = S.Name;
```

These two queries are equivalent. Is one preferable to the other?

SQL … Extensions

SELECT…

FROM…

WHERE…

ORDER BY…

GROUP BY…

HAVING …

Extension to the SELECT clause
e.g., SUM, COUNT, MIN, MAX, AVG and AS

Extension to the FROM clause
e.g., correlation names and various kinds of JOINs

Extension to the WHERE clause
e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY

Several additional clauses
e.g., ORDER BY, GROUP BY, and HAVING

ORDER BY

Sort the result on one or more attributes
Can specify ASC, DESC--default is ASC

```
SELECT Name, Address
FROM Customers
ORDER BY Name
```

```
SELECT *
FROM Customer C JOIN Salesperson S
          ON C.SalespersonNum
ORDER BY CRating DESC, C.Name, S.Name
```

ORDER BY: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

SELECT Name, Address
FROM Customers
ORDER BY Name

Answer:
Jones, yyy
Smith, xxx
Wei, zzz

ORDER BY: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	102

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

```
SELECT *
FROM Customer C JOIN Salesperson S
                    ON C.SalespersonNum
ORDER BY CRating DESC, C.Name, S.Name
```

Answer:

3, wei, zzz, 10 …

2, jones, yyy, 7

1, smith, xxx, 5

ORDER BY: Example

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	Ann	aaa	7	3,000	20,000	102
4	wei	zzz	10	10,000	10,000	102

Salesperson

Number	Name Address Office		
101	johnson	aaa	23
102	miller	bbb	26

```
SELECT *
FROM Customer C JOIN Salesperson S
            ON C.SalespersonNum
ORDER BY CRating DESC, C.Name, S.Name
```

Answer:

4, wei, zzz, 10 …

3, ann, aaa, 7…

2, jones, yyy, 7…

1, smith, xxx, 5…

Grouping

- GROUP BY partitions a relation into groups of tuples that *agree* on the value of one or more columns
- Useful when combined with aggregation – apply aggregation within each group
- Any form of SQL query (e.g., with or without subqueries) can have the answer “grouped”
- The query result contains one output row for each group

GROUP BY

```
SELECT      SalespersonNum, COUNT(*) as TotCust
FROM        Customer
GROUP BY    SalespersonNum;
```

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	102

Group 1

Group 2

Answer

SalespersonNum	TotCust
101	2
102	1

Challenge Question

- What is the answer for the query:

SELECT SalespersonNum

FROM Customer

GROUP BY SalespersonNum

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	102

???

Challenge Question

- What is the answer for the query:

SELECT SalespersonNum

FROM Customer

GROUP BY SalespersonNum

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	102

Answer

SalespersonNum
101
102

Another Challenge Question

- Can you write a simpler SQL stmt for this query?

```
SELECT SalespersonNum
FROM Customer
GROUP BY SalespersonNum
```

```
SELECT DISTINCT SalespersonNum
FROM Customer
```

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	102

Answer

SalespersonNum
101
102

HAVING Clauses

- Select groups based on some aggregate property of the group
  - E.g., Only list a salesperson if he/she has more than 10 customers
- The HAVING clause is a condition evaluated against each group
  - A group participates in the query answer if it satisfies the HAVING predicate

```
SELECT      SalespersonNum
FROM        Customer
GROUP BY    SalespersonNum
HAVING      Count(*) > 10;
```

GROUP BY Clauses and NULLS

- Aggregates ignore NULLs
- On the other hand, NULL is treated as an ordinary value in a grouped attribute
- If there are NULLs in the Salesperson column, a group will be returned for the NULL value

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	NULL

```
SELECT  SalespersonNum, Count(*) AS T
FROM     Customer
GROUP BY  SalespersonNum
```

Answer

SalespersonNum	T
NULL	1
101	2

GROUP BY, HAVING: Note

- The only attributes that can appear in a "grouped" query answer are aggregate operators (that are applied to the group) or the grouping attribute(s)

```
SELECT SalespersonNum, COUNT(*)
FROM          Customer
GROUP BY      SalespersonNum;

SELECT        SalespersonNum
FROM          Customer
GROUP BY      SalespersonNum
HAVING Count(*) > 10;
```

Incorrect!

```
SELECT
SalespersonNum, C.Name, COUNT(*)
FROM            Customer C
GROUP BY        SalespersonNum;
```

Readable SQL Queries

```
SELECT      SalespersonNum
FROM        Customer
GROUP BY    SalespersonNum
HAVING      Count(*) > 10
ORDER BY    SalespersonNum
```

- Offer visual clues to the structure of query
  - Each ‘important’ keyword starts a new line
  - Capitalize keywords
- Keep it compact
  - If query or subquery is short, write in a single line

```
SELECT * FROM Customer
```

Order of Clauses in SQL Queries

SELECT…

FROM…

Required

WHERE…

GROUP BY…

HAVING …

ORDER BY…

- SELECT and FROM are required
- Can't use HAVING without GROUP BY
- Whichever additional clauses appear must be in the order listed

SQL … Extensions

SELECT…

FROM…

WHERE…

ORDER BY…

GROUP BY…

HAVING …

Extension to the SELECT clause
e.g., SUM, COUNT, MIN, MAX, AVG and AS

Extension to the FROM clause
e.g., correlation names and various kinds of JOINs

Extension to the WHERE clause
e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY

Several additional clauses
e.g., ORDER BY, GROUP BY, and HAVING

(SELECT…FROM…WHERE…)
UNION
(SELECT…FROM…WHERE…)

And operators that expect two or more complete SQL queries as operands
e.g., UNION, INTERSECT, MINUS

UNIONing Subqueries

```
(SELECT   C.Name
FROM      Customer C
WHERE     C.Name LIKE “B%”)
UNION
(SELECT   S.Name
FROM      Salesperson S
WHERE     S.Name LIKE “B%”);
```

Two complete
queries -
with UNION
operator
in between.

Unlike other operations, UNION eliminates duplicates!

UNION ALL

```
(SELECT   C.Name
FROM      Customer C
WHERE     C.Name LIKE “B%”)
UNION ALL
(SELECT   S.Name
FROM      Salesperson S
WHERE     S.Name LIKE “B%”);
```

UNION ALL preserves duplicates

EXCEPT (=difference)

```
(SELECT   S.Number
FROM      Salesperson)
EXCEPT
(SELECT   C.SalespersonNum Number
FROM      Customer C);
```

Two complete queries - with EXCEPT operator in between.

EXCEPT ALL retains duplicates

What is this query looking for?

EXCEPT (=difference)

```
(SELECT   S.Number
FROM      Salesperson;)
MINUS
(SELECT   C.SalespersonNum Number
FROM      Customer C;);
```

Oracle provides a MINUS operator to represent difference!

INTERSECT

```
(SELECT   S.Name
FROM      Salesperson)
INTERSECT
(SELECT   C.Name
FROM      Customer C);
```

Two complete
queries -
with
INTERSECT
operator
in between.

INTERSECT ALL retains duplicates

What is this query looking for?

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.
- What about union?

The WITH Clause

- Complex queries are easier to write if you break them up into smaller components
- You can name a query component using the WITH clause
  - It creates a temporary view, which is valid *only* in the query where it is defined

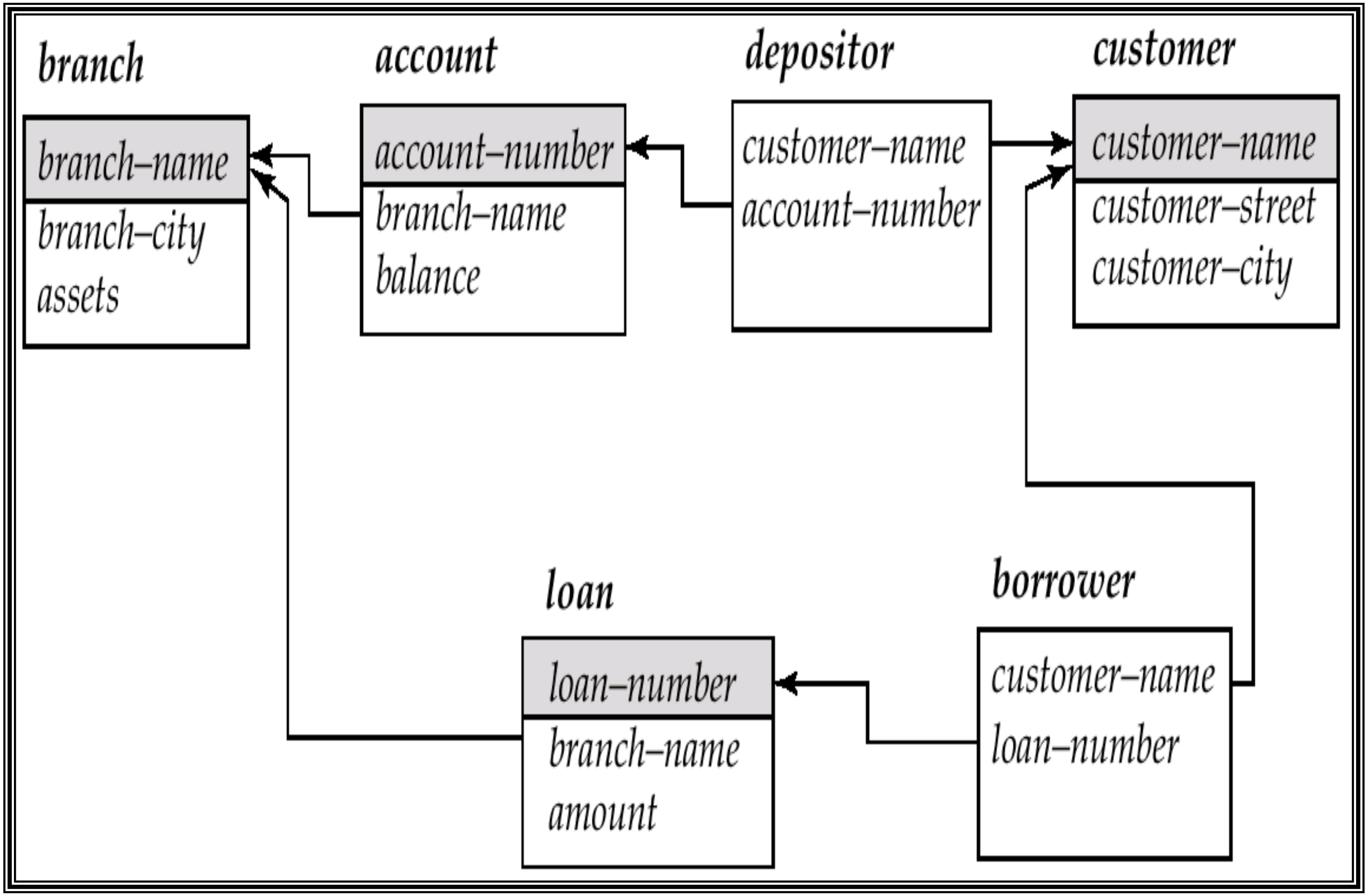
```
WITH max_balance(value) AS
   SELECT MAX(balance)
   FROM Customer
SELECT Cname
FROM Customer C, max_balance
WHERE C.balance = max_balance.value
```

Modifying the Database

Database Modifications

- Some SQL statements do not return any results…
- They change the state of the database
  - Insert tuples into a relation
  - Delete certain tuples from a relation
  - Update values of certain components of existing tuples

Example


branch
branch–name
branch–city
assets
account
account–number
branch–name
balance
depositor
customer–name
account–number
customer
customer–name
customer–street
customer–city
loan
loan–number
branch–name
amount
borrower
customer–name
loan–number

Deletion

DELETE FROM *R*
WHERE <*condition*>

- Delete whole tuples, one relation at a time
- Finds and deletes all tuples *t* in *R* such that *condition(t)* is true

- Examples:

Delete all account records at the Perryridge branch

DELETE FROM *account*
WHERE *branch-name* = 'Perryridge'

Delete all accounts at every branch located in Needham city.

DELETE FROM *account*
WHERE *branch-name* **IN** (
  SELECT *branch-name*
  FROM *branch*
  WHERE *branch-city* = 'Needham')

What does the following statement do?

- **delete from** account

Delete: Example

- Delete the record of all accounts with balances below the average at the bank.

delete from *account*
where *balance* < (**select avg** *(balance)* **from** *account)*

★ Problem: as we delete tuples from *account,* the average balance changes

Solution used in SQL:
1. First, compute **avg** balance and find all tuples to delete
2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Inserting a Tuple into a Relation

INSERT INTO *R(A1,…,An)* VALUES *(v1,…,vn)*

- A tuple is created using value *vi* for attribute *Ai,* for *i=1,…,n*

insert into

account (branch-name, balance, account-number)
values ('Perryridge' , 1200, 'A-9732')

INSERT INTO *R* VALUES *(v1,…,vn)*

- A tuple is created using value *vi* for all attributes *A of R*
  - *Order of values must be the same as the standard order of the attributes in the relation*

insert into *account*
values ('A-9732' , 'Perryridge' ,1200) ---- correct order!

Inserting a Tuple into a Relation (cont.)

insert into

account (branch-name, account-number)
values ('Perryridge' , 'A-9732')

Is equivalent to

insert into

account (branch-name, account-number,balance)
values ('A-9732' , 'Perryridge' ,NULL)

- If a value is omitted, it will become a NULL

Inserting the Results of a Query

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

insert into *account*

select *loan-number, branch-name,* **200**
from *loan*
where *branch-name* = 'Perryridge'

Set of tuples to insert

insert into *depositor*

select *customer-name, loan-number*
from *loan, borrower*
where branch-name = 'Perryridge'
and *loan.account-number* = *borrower.account-number*

Order of Insertion

```
insert into account
    select loan-number, branch-name, 200
    from loan
    where branch-name = ‘Perryridge’
 insert into depositor
    select customer-name, loan-number
    from loan, borrower
    where branch-name = ‘Perryridge’
          and loan.account-number = borrower.account-number
```

- The **select from where** statement is fully evaluated before any of its results are inserted into the relation.

- What would happen with the following query?

```
insert into table1 select * from table1
```

Updates

- Choose tuples to be updated using a query

update R
set *attribute* = expression
where <condition>

- Pay 5% interest on accounts whose balance is greater than average

update *account*
set *balance* = *balance* * 1.05
where *balance* > (**select** avg(balance)
from account)

Update: Example

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

    update *account*
    set *balance* = *balance* * 1.06
    where *balance* > 10000

    update *account*
    set *balance* = *balance* * 1.05
    where *balance* ≤ 10000
  - The order is important, why?

    Accounts with balance > 10000 would be updated twice!
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance =  case
                   when balance <= 10000
                   then balance *1.05
                   else   balance * 1.06
               end
```

SQL as a Data Definition Language (DDL)

Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- *Integrity constraints*
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person-name* **char**(20) **not null**

Date/Time Types in SQL (Cont.)

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. **date** ‘2001-7-27’
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. **time** ’09:00:30’ **time** ’09:00:30.75’
- **timestamp**: date plus time of day
  - E.g. **timestamp** ‘2001-7-27 09:00:30.75’
- **Interval**: period of time
  - E.g. Interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
- Can extract values of individual fields from date/time/timestamp
  - E.g. **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
  - E.g. **cast** <string-valued-expression> **as date**

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table *r* (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i
- Example:

  create table *branch*
  (*branch-name* char(15) **not null,**
  branch-city char(30),
  assets integer)

Integrity Constraints in Create Table

- **not null**
- **primary key** ($A_1, ..., A_n$)
- **check** *(P),* where *P* is a predicate
  - P must be satisfied by all tuples

Example: Declare *branch-name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch
    (branch-namechar(15),
    branch-city     char(30),
    assets          integer,
    primary key (branch-name),
    check (assets >= 0))
```

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89

Integrity Constraints in Create Table

- **foreign key** (A_1, ..., A_n) **references** *R*

Example: Create the borrower table which captures the relationship between borrower and customer, and between borrower and loan

```
create table borrower (customer_name varchar(30),
                        loan_number number(8),
        CONSTRAINT fk1
          FOREIGN KEY (customer_name)
          REFERENCES customer (customer_name),
        CONSTRAINT fk2
          FOREIGN KEY (loan_number)
          REFERENCES loan )
```

Integrity Constraints in Create Table

- ON DELETE CASCADE
- Specifies that if an attempt is made to delete a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted.

```
create table borrower (customer_name varchar(30),
                  loan_number number(8),
   CONSTRAINT fk1
   FOREIGN KEY (customer_name)
   REFERENCES customer (customer_name),
   CONSTRAINT fk2
   FOREIGN KEY (loan_number)
   REFERENCES loan ON DELETE CASCADE)
```

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation.

  alter table *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- Examples:
  - ALTER TABLE borrower ADD b_date DATE
  - DROP TABLE borrower

Drop and Alter Table Constructs (cont.)

- The **alter table** command can also be used to drop attributes of a relation

  alter table *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*
  - E.g., ALTER TABLE borrower DROP b_date
- Dropping of attributes not supported by many databases
- The **alter table** command can also be used to drop or add constraints
  - More about this later!

Default Values

- Any place we declare an attribute we may add the keyword DEFAULT followed by NULL or a constant
- Example:
  - Gender CHAR(1) DEFAULT ‘?’
  - Birthdate DATE DEFAULT DATE ‘0000-00-00’

Views

- Relation that is not part of the logical model

  create view *v* **as** <query expression>

  where <query expression> is any legal relational algebra query expression. The view name is represented by *v*
- Once a view is defined, its name can be used to refer to the virtual relation that the view generates
- View definition is not the same as creating a new relation by evaluating the query expression
- A *view definition causes the saving of an expression*; the expression is substituted into queries using the view

View: Examples

```
CREATE VIEW AllCustomers AS
    (SELECT branch-name, cust-name
    FROM depositor D, account A
    WHERE D.account-no=A.number)
UNION
     (SELECT branch-name, cust-name
    FROM borrower B, loan L
    WHERE B.loan-no=L.number)
```

Customers with a savings account

Customers with a loan account

- To find all customers of the Perryridge branch:

```
SELECT cust-name
FROM AllCustomers
WHERE branch-name= 'Perryridge'
```

Views: Renaming Attributes

```
CREATE VIEW AllCustomers(bname,cname) AS
    (SELECT branch-name, cust-name
    FROM depositor D, account A
    WHERE D.account-no=A.number)
UNION
     (SELECT branch-name, cust-name
    FROM borrower B, loan L
    WHERE B.loan-no=L.number)
```

Customers with a savings account

Customers with a loan account

- To find all customers of the Perryridge branch:

```
SELECT cname
FROM AllCustomers
WHERE bname= ‘Perryridge’
```

Interpreting Queries that use Views

- To find all customers of the Perryridge branch:

```
SELECT cust-name
FROM AllCustomers
WHERE branch-name= ‘Perryridge’
---->
(SELECT branch-name, cust-name
FROM depositor D, account A
WHERE D.account-no=A.number AND branch-name= ‘Perryridge’ )
UNION
 (SELECT branch-name, cust-name
FROM borrower B, loan L
WHERE B.loan-no=L.number AND branch-name= ‘Perryridge’ )
```

The database will perform this rewriting

Materialized Views

- Create a *real* table

```
CREATE MATERIALIZED VIEW hr.employees AS
        SELECT * FROM hr.employees@orc1.world;
```

Challenge Questions

Are these queries equivalent?

SELECT AVG(amount) FROM loan

SELECT SUM(amount)/COUNT(*) FROM loan

Challenge Question

Are these queries equivalent?

```
SELECT SalespersonNum, AVG(CBalance)
FROM  Customer
GROUP BY SalespersonNum
HAVING AVG(CBalance) > 200;
```

```
SELECT SalespersonNum, AVG(CBalance)
FROM  Customer
WHERE CBalance > 200
GROUP BY SalespersonNum;
```

Challenge Question

equivalent

```
SELECT      *
FROM        Customer C, Salesperson S
WHERE       C.SalespersonNum = S.Number;

SELECT      *
FROM        Customer C JOIN Salesperson S on
                                  SalespersonNum;
```

```
SELECT      *
FROM        Customer NATURAL JOIN Salesperson
```

this query is not equivalent to above two queries, why?

Customer (Number, Name, Address, CRating, CAmount, CBalance, SalespersonNum)

Salesperson (Number, Name, Address, Office)