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

**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.

```sql
SELECT A1, A2, ..., Am
FROM R1, R2, ..., Rn
WHERE C1, C2, ..., Ck
```
Challenge Question

• Suppose R, S and T are unary relations, each having one attribute A. We want to compute $R \cap (S \cup T)$.

• Does the following query do the job?
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A

???
Challenge Question

• Suppose R, S and T are unary relations, each having one attribute A. We want to compute \( R \cap (S \cup T) \).

• Does the following query do the job?

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A
```

What happens if T is empty?

Choose one of the interpretations, e.g., Relational Algebra: \( R \times S \times T = \emptyset \)
SQL … Extensions

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

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

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;
```

```
SELECT Name
FROM Customer;
```

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

SELECT AVG (CBalance) FROM Customer WHERE age > 35;
Aggregates and NULLs

• General rule: aggregates ignore NULL values
  – $\text{Avg}(1,2,3,\text{NULL},4) = \text{Avg}(1,2,3,4)$
  – $\text{Count}(1,2,3,\text{NULL},4) = \text{Count}(1,2,3,4)$

• But…
  – $\text{Count}(*)$ 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;

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

Answer: 3  Answer: 2

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>J. Smith</td>
</tr>
<tr>
<td>W. Wei</td>
</tr>
<tr>
<td>J. Smith</td>
</tr>
</tbody>
</table>
Note: Full-Relation Operations

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

- More on aggregates later!
SQL … Extensions

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
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  \equiv  INNER JOIN

These queries are equivalent.

\[
\begin{align*}
\text{SELECT} & \quad \text{C.Name, S.Name} \\
\text{FROM} & \quad \text{Customer C JOIN Salesperson S} \\
& \quad \text{ON SalespersonNum;}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{C.Name, S.Name} \\
\text{FROM} & \quad \text{Customer C INNER JOIN Salesperson S} \\
& \quad \text{ON SalespersonNum;}
\end{align*}
\]

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 $\equiv$ Cross Product
Natural Join

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

```sql
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;
```

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

Salesperson (Number, Name, Address, Office)

BTW, what does this query compute?
Natural Join: Some Notes

- \[ SELECT * \]
  - 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)
Natural Join: Some Notes

NATURAL JOIN requires equal values for all corresponding columns from the two tables that have the same name (or you can list the column name for the join) in the ON clause.

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

```
SELECT C.Name, S.Name
FROM Customer C NATURAL JOIN
(SELECT Name, Number AS SalespersonNum
 FROM Salesperson) S
ON SalespersonNum;
```

These two queries are equivalent.
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.
  
  ```sql
  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**\( (\text{sid}, \text{name}, \text{address}) \)
- **Spouse**\( (\text{sid}, \text{name}) \), \text{sid} \text{ references} \text{ Student} . \text{sid} \\
- List the names of all students and their spouses, if they have one.
  
  SELECT \text{Student.name}, \text{Spouse.name}  
  FROM \text{Student, Spouse}  
  WHERE \text{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**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**Salesperson**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Salesperson

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- **WHERE**...
  - Extension to the WHERE clause
  - e.g., AND, OR, NOT, comparators, EXISTS, IN, ANY
WHERE Clause: Comparison Operators

- \(<\), \(>\), \(=\), \(<>\), \(\geq\), \(\leq\)
  - 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

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

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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

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

Result: ?

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

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

Tests for empty relations
EXISTS: Example

**Customer**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**Salesperson**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

**SQL Query**

```sql
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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Salesperson

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

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

Four predicates can be placed in front of a subquery in SQL:

**UNIQUE** (subquery) -- tests if there are any duplicate tuples
**NOT UNIQUE** (subquery) -- some rows may appear more than once
Set comparison:
ALL or ANY in a Subquery

• Syntax:
  – attribute-name \( \text{comparator} \) ALL (subquery)
  – attribute-name \( \text{comparator} \) ANY (subquery)

• \( A > \text{ALL} \) (subquery S):
  – True if \( A \) is greater than every value returned by \( S \)
  – \((A <> \text{ALL} S) \equiv (A \text{ NOT IN } S)\)

• \( A > \text{ANY} \) (subquery S)
  – True if \( A \) is greater than at least one value returned by \( S \)
  – \((A = \text{ANY} S) \equiv (A \text{ IN } S)\)
ALL or ANY in a Subquery: Example

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

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

What does this query compute?

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>101</td>
</tr>
</tbody>
</table>

Salesperson

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>
Subquery: 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 C1.Number, C1.Name
FROM Customer C1
WHERE C1.CRating IN (SELECT MAX (C2.CRating)
                        FROM Customer C2)
```

(10)
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;);
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 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

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

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

De-correlate: another way to write this query:

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

- 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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

### Salesperson

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

SELECT Name, Address
FROM Customers
ORDER BY Name

Answer:
Jones, yyy
Smith, xxx
Wei, zzz
### ORDER BY: Example

**Customer**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>102</td>
</tr>
</tbody>
</table>

**Salesperson**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

**Answer:**

3, wei, zzz, 10 …
2, jones, yyy, 7
1, smith, xxx, 5

**SQL Query:**

```sql
SELECT *
FROM Customer C JOIN Salesperson S
ON C.SalespersonNum = S.Number
ORDER BY CRating DESC, C.Name, S.Name
```
### ORDER BY: Example

**Customer**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>Ann</td>
<td>aaa</td>
<td>7</td>
<td>3,000</td>
<td>20,000</td>
<td>102</td>
</tr>
<tr>
<td>4</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>102</td>
</tr>
</tbody>
</table>

**Salesperson**

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>johnson</td>
<td>aaa</td>
<td>23</td>
</tr>
<tr>
<td>102</td>
<td>miller</td>
<td>bbb</td>
<td>26</td>
</tr>
</tbody>
</table>

**Answer:**

4, wei, zzz, 10 …
3, ann, aaa, 7…
2, jones, yyy, 7…
1, smith, xxx, 5…

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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>102</td>
</tr>
</tbody>
</table>

Group 1

<table>
<thead>
<tr>
<th>SalespersonNum</th>
<th>TotCust</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
</tr>
</tbody>
</table>
Challenge Question

• What is the answer for the query:

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

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>102</td>
</tr>
</tbody>
</table>

???
Challenge Question

• What is the answer for the query:
SELECT SalespersonNum
FROM Customer
GROUP BY SalespersonNum

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Address</th>
<th>CRating</th>
<th>CAmount</th>
<th>CBalance</th>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smith</td>
<td>xxx</td>
<td>5</td>
<td>1,000</td>
<td>1,000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>jones</td>
<td>yyy</td>
<td>7</td>
<td>5,000</td>
<td>4,000</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>wei</td>
<td>zzz</td>
<td>10</td>
<td>10,000</td>
<td>10,000</td>
<td>102</td>
</tr>
</tbody>
</table>

Answer

<table>
<thead>
<tr>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>
Another Challenge Question

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

SELECT SalespersonNum
FROM Customer
GROUP BY SalespersonNum

Answer

<table>
<thead>
<tr>
<th>SalespersonNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>

SELECT DISTINCT SalespersonNum
FROM Customer
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

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Answer

<table>
<thead>
<tr>
<th>SalespersonNum</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
</tr>
</tbody>
</table>
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)

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

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

Incorrect!

```sql
SELECT SalespersonNum
FROM Customer
GROUP BY SalespersonNum
HAVING Count(*) > 10;
```
## Readable SQL Queries

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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  
| SELECT * FROM Customer | Keep it compact  
  - If query or subquery is short, write in a single line |
Order of Clauses in SQL Queries

- 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

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

(_SELECT...FROM...WHERE...) UNION (SELECT...FROM...WHERE...)
**UNIONing Subqueries**

<table>
<thead>
<tr>
<th>SELECT</th>
<th>C.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Customer C</td>
</tr>
<tr>
<td>WHERE</td>
<td>C.Name LIKE “B%”</td>
</tr>
</tbody>
</table>

**UNION**

<table>
<thead>
<tr>
<th>SELECT</th>
<th>S.Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Salesperson S</td>
</tr>
<tr>
<td>WHERE</td>
<td>S.Name LIKE “B%”</td>
</tr>
</tbody>
</table>

Two complete queries - with UNION operator in between.

Unlike other operations, UNION eliminates duplicates!
UNION ALL preserves duplicates
EXCEPT (=difference)

(SELFCET S.Number
FROM Salesperson) EXCEPT

(SELECT C.SalespersonNum Number
FROM Customer C);

EXCEPT ALL retains duplicates

Two complete queries - with EXCEPT operator in between.

What is this query looking for?
EXCEPT (=difference)

(SELECT S.Number
FROM Salesperson);

MINUS

(SELECT C.SalespersonNum Number
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.
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
```
Practice Exercise
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
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*

  ```sql
  DELETE FROM account
  WHERE branch-name = 'Perryridge'
  ```

  *Delete all accounts at every branch located in Needham city.*

  ```sql
  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 deposit, 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

\[
\text{INSERT INTO } R(A_1, \ldots , A_n) \text{ VALUES } (v_1, \ldots , v_n)
\]

- A tuple is created using value \( v_i \) for attribute \( A_i \), for \( i=1, \ldots , n \)

```sql
insert into account (branch-name, balance, account-number)
values ('Perryridge', 1200, 'A-9732')
```

\[
\text{INSERT INTO } R \text{ VALUES } (v_1, \ldots , v_n)
\]

- A tuple is created using value \( v_i \) for all attributes \( A \) of \( R \)
  - Order of values must be the same as the standard order of the attributes in the relation

```sql
insert into account
values ('A-9732', 'Perryridge',1200) ---- correct order!
```
Inserting a Tuple into a Relation (cont.)

```sql
insert into account (branch-name, account-number)
values ('Perryridge', 'A-9732')
```

Is equivalent to

```sql
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.

```sql
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
```

Set of tuples to insert
Order of Insertion

```sql
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, why?

- What would happen with the following query?
  ```sql
  insert into table1 select * from table1
  ```
  Infinite loop!
Updates

• Choose tuples to be updated using a query
  
  \textbf{update} \ R \\
  \textbf{set} \ \textit{attribute} = \text{expression} \\
  \textbf{where} \ <\text{condition}> \\

• Pay 5\% interest on accounts whose balance is greater than average
  
  \textbf{update} \ \textit{account} \\
  \textbf{set} \ \textit{balance} = \textit{balance} \ast \ 1.05 \\
  \textbf{where} \ \textit{balance} > \ ( \textbf{select} \ \text{avg}(\textit{balance}) \ \\
  \textbf{from} \ \textit{account})
Update: Example

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

  - Write two `update` statements:
    ```sql
    update account
    set balance = balance * 1.06
    where balance > 10000
    ```

    ```sql
    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%.

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

• Remember correlated subqueries? You can also use correlation in updates.

```sql
update Target_Table
set Target_Field =
    (Select Source_Information
     From Source_Table
     Where Source_Table.Key = Target_Table.Key)
where exists (Select 'x'
    From Source_Table
    Where Source_Table.Key = Target_Table.Key)
```
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 relation.
- 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:

  ```sql
  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:

  ```sql
  create table branch
  (branch-name char(15) not null,
   branch-city char(30),
   assets integer)
  ```
Integrity Constraints in Create Table

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

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

```sql
create table branch
  (branch-name char(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

Will talk about this next class!
Integrity Constraints in Create Table

- **foreign key** \((A_1, \ldots, A_n)\) references \(R\)

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

```sql
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.

```sql
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.

  \[
  \text{alter table } r \text{ add } A \ D
  \]

  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
  
  ```sql
  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** \(<\text{query expression}>\)
  
  where \(<\text{query expression}>\) 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
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)

• To find all customers of the Perryridge branch:
  SELECT cust-name
  FROM AllCustomers
  WHERE branch-name='Perryridge'

Customers with a savings account
Customers with a loan account
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
Updates Through View

• Database modifications expressed as views must be translated to modifications of the actual relations in the database.

• Consider the person who needs to see all loan data in the loan relation except amount. The view given to the person, branch-loan, is defined as:

```sql
CREATE VIEW branch-loan AS
  SELECT branch-name, loan-number
FROM loan
```

• Since we allow a view name to appear wherever a relation name is allowed, the person may write:

```sql
INSERT INTO branch-loan VALUES ("Perryridge", L-37)
```
Updates Through Views (Cont.)

• Need to rewrite insertion into the actual relation _loan_ from which the view _branch-loan_ is constructed.

• An insertion into _loan_ requires a value for _amount_. The insertion can be dealt with by either:
  – rejecting the insertion and returning an error message to the user, or
  – inserting a tuple (‘L-37’, ‘Perryridge’, _null_) into the _loan_ relation

• Others cannot be translated uniquely…
Updates Through Views (Cont.)

- CREATE VIEW all_customer AS (SELECT branch_name, customer_name
  FROM depositor JOIN account
  ON account_number)
  UNION
  (SELECT branch_name, customer_name
  FROM borrower JOIN loan
  ON loan_number)

- Query:
  INSERT INTO all_customers VALUES ('Perryridge', 'John')

*How would you translate this query?*

Have to choose loan or account, and create a new loan/account number!
Updates Through Views (Cont.)

- CREATE VIEW loan_info AS
  SELECT customer_name, amount
  FROM borrower JOIN loan
  ON loan_number

- Query:
  INSERT INTO loan_info VALUES ('John', 1900)

How would you translate this query?

INSERT INTO loan(null,null,1900)
INSERT INTO borrower('John',null)

Does this make sense? Would a SELECT * query over the loan_info view return the newly inserted tuple?
Updatable Views

- SQL provides a formal definition of when modifications to a view are permitted.
- Permit modifications on views only if:
  - FROM clause has only 1 relation, and
  - SELECT clause contains only attribute names of relation and does not have expressions, aggregates, or distinct
  - Any attribute not listed in SELECT can be NULL
  - Query does not have GROUP BY or HAVING
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

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)
Null Values

• It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
• *null* signifies an unknown value or that a value does not exist.
• The predicate **is null** can be used to check for null values.
  – E.g. Find all loan number which appear in the *loan* relation with null values for *amount*.

```sql
select loan-number 
from loan 
where amount is null
```

• The result of any arithmetic expression involving *null* is *null*
  – E.g. 5 + null returns null

• However, aggregate functions simply ignore nulls
  – more on this shortly
Null Values and Three Valued Logic

• Any comparison with null returns unknown
  – E.g. 5 < null or null <> null or null = null

• Result of where clause predicate is treated as false if it evaluates to unknown

• Three-valued logic using the truth value unknown:
  – OR: (unknown or true) = true, (unknown or false) = unknown
    (unknown or unknown) = unknown
  – AND: (true and unknown) = unknown, (false and unknown) = false,
    (unknown and unknown) = unknown
  – NOT: (not unknown) = unknown
  – “P is unknown” evaluates to true if predicate P evaluates to unknown
Nulls and Three-Valued Logic

- Follow the same logic for true and false, and
- Remember unknown is in between!
Null Values and Aggregates

• Total all loan amounts
  
  \[
  \text{select } \text{sum} (\text{amount}) \\
  \text{from } \text{loan}
  \]
  
  – Above statement ignores null amounts
  – result is null if there is no non-null amount
  – All aggregate operations except \text{count}(*) ignore tuples with null values on the aggregated attributes.

Are these queries equivalent?

SELECT AVG(amount) FROM loan
SELECT \text{SUM(amount)}/\text{COUNT(*)} \text{ FROM loan}
Outer Join – Example

- Relation *loan*

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
</tr>
</tbody>
</table>

- Relation *borrower*

<table>
<thead>
<tr>
<th>customer-name</th>
<th>loan-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>L-170</td>
</tr>
<tr>
<td>Smith</td>
<td>L-230</td>
</tr>
<tr>
<td>Hayes</td>
<td>L-155</td>
</tr>
</tbody>
</table>
Outer Join – Example

- **Inner Join**

  \( \text{loan} \bowtie \text{Borrower} \)

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
</tbody>
</table>

- **Left Outer Join**

  \( \text{loan} \leftouterjoin \text{Borrower} \)

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
</tr>
</tbody>
</table>

*How would you represent an outer join using the basic relational algebra operations?*
## Outer Join – Example

### Right Outer Join

\( \text{loan} \bowtie \text{borrower} \)

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>

### Full Outer Join

\( \text{loan} \bowtie \text{borrower} \)

<table>
<thead>
<tr>
<th>loan-number</th>
<th>branch-name</th>
<th>amount</th>
<th>customer-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-170</td>
<td>Downtown</td>
<td>3000</td>
<td>Jones</td>
</tr>
<tr>
<td>L-230</td>
<td>Redwood</td>
<td>4000</td>
<td>Smith</td>
</tr>
<tr>
<td>L-260</td>
<td>Perryridge</td>
<td>1700</td>
<td>null</td>
</tr>
<tr>
<td>L-155</td>
<td>null</td>
<td>null</td>
<td>Hayes</td>
</tr>
</tbody>
</table>