

Lecture 2 (Week 2)

CS 386/586 Winter 2009

Introducing relational algebra
and then ...

SQL, rel. alg., SQL, rel. alg., ...

Why is a mathematical definition useful?

- Can you think of examples when a mathematical definition improved a practical problem?
- Suppose someone asks you to implement something ... described in English text
- vs. someone asks you to write a method or function that implements modulo 3 arithmetic

Query Languages for Relational DBs

Practical definition of relational database

Operates on tables
(with duplicates)

Practical QL:

SQL

- basic
SELECT/FROM/WHERE
- ...

Mathematical definition of relational database

Operates on relations
(relations are sets)

Mathematical QL:

Relational Algebra

- Intersection
- Union
- ...

Commercial DBMSs – sets or bags?

- The default is to produce a bag of rows as a query answer.
- If you want a set – use DISTINCT in the Select clause.
- Why do you think they do that?
- Some DBMSs use a set-based union, intersection, and difference in SQL.
- Note: even though relational algebra was originally defined as set-based, SQL queries are represented internally using relational algebra (with extra operators).

Despite these differences...

- Relational operators (with appropriate additional operators beyond the original eight) are used to represent queries internally - in a DBMS
- A complete set of operators for bags - that includes the relational algebra operators - plus the additional operators - can be defined formally...and the equivalences can be proven

Overall Plan to Understand SQL Queries

- Present mathematical definition of a relational database
- Intermix relational algebra (a mathematically defined query language for relational databases) and SQL
- ...
 - a little relational algebra ...
 - then a little more SQL ...
 - then a little more relational algebra ...
 - then a little more SQL ...

Query Languages for Relational DBs

Practical definition of relational database

Practical QL: SQL

- basic
SELECT/FROM/WHERE
- Extensions to "SELECT"
- Extensions to "FROM"
- Extensions to "WHERE"
- UNION/INTERSECT/EXCEPT
- GROUP BY/HAVING
- ORDER BY

Mathematical def. of relational database

Mathematical QL: Rel. Alg.

Last week – Lecture 1

Query Languages for Relational DBs

Practical definition of relational database
Practical QL: SQL
<ul style="list-style-type: none">• basic SELECT/FROM/WHERE• Extensions to "SELECT"• Extensions to "FROM"• Extensions to "WHERE"• UNION/INTERSECT/ EXCEPT• GROUP BY/HAVING• ORDER BY

Mathematical def. of relational database
Mathematical QL: Rel. Alg.

Part of the operators

This week –
Lecture 2

Query Languages for Relational DBs

Practical definition of relational database

Practical QL: SQL

- basic SELECT/FROM/WHERE
- Extensions to "SELECT"
- Extensions to "FROM"
- Extensions to "WHERE"
- UNION/INTERSECT/EXCEPT
- GROUP BY/HAVING
- ORDER BY

Mathematical def. of relational database

Mathematical QL: Rel. Alg.

Rest of the operators

Next Week –
Lecture 3

Describing a Relational Database Mathematically

- A relation is a **set** – of **tuples** as in the original definition of the relational model [Codd 1970]
- Query operators are adapted from **set theory**
- **Relational algebra** is a mathematically defined query language

Cross Product: an operator from set theory

Suppose.. $A = \{a, b, c\}$ $B = \{1, 2\}$

then in *set theory*, the cross product is defined as:

$$A \times B = \{(a, 1), (b, 1), (c, 1), (a, 2), (b, 2), (c, 2)\}$$

$A \times B$ is a set consisting of ordered pairs (2-tuples)
where each pair consists of an element from A and
an element from B

Practice Question

Suppose.. $A = \{a, b, c\}$ $B = \{1, 2\}$

What is $B \times B$?

Practice Question

Suppose.. $A = \{a, b, c\}$ $B = \{1, 2\}$

What is $B \times B$?

$B \times B = \{(1, 1), (2, 1), (1, 2), (2, 2)\}$

Definition of a Relation

Suppose we have a relation defined as:

Person(name, salary, num, status) with domains defined as:

Name-values = {all possible strings of 30 characters}

Sal-values = {real numbers between 0 and 100,000}

Status-values = {"f", "p"}

Num-values = {integers between 0 and 9999}

any instance of the relation is always a subset (\subseteq) of:

Name-values X Sal-values X Num-values X Status-values

Note: a domain is a set of simple, atomic values.

Codd's Definition of a Relation

- Suppose we have a relation defined as:

Person(name, salary, num, status) with domains defined as:

Name-values = {all possible strings of 30 characters}

Sal-values = {real numbers between 0 and 100,000}

Status-values = {"f", "p"}

Num-values = {integers between 0 and 9999}

Any instance of a relation is always a subset (\subseteq) of the cross product of its domains:

Name-values X Sal-values X Num-values X Status-values

Mathematical Definition of a Relational DB (cont.)

- Each (instance of a) relation is a subset of the cross product of its domains.
- One element of a relation is called a tuple.
- A relation is ALWAYS a set – by definition.
 - If you add the element 2 to the set $\{1, 2, 3, 4\}$ the resulting set is $\{1, 2, 3, 4\}$
 - If you add the tuple $\{101, \text{"J. Smith"}, 1000.00, \text{"checking"}\}$ to the relation on the next slide, you still only have five tuples.

Terminology for Relational Databases

Slide repeated from Lecture 1

Account

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

relation

Each entry in the ~~table~~ is called a ~~row~~ or a **tuple**.

Sometimes an entry in the table is called a record.

The *instance* is the current set of rows (or tuples).

Set Theory (VERY brief reminder)

Consider two sets:

Set 1 = {1, 3, 5, 7}

Set 2 = {1, 2, 3, 4}

Set 1 \cap Set 2

Set 1 \cup Set 2

Set 1 - Set 2

Set 1 \times Set 2

Relational Algebra has other operators invented by Codd (discussed a bit later)

Consider two sets:

Set 1 = {1, 3, 5, 7}

Set 2 = {1, 2, 3, 4}

Set 1 $\bowtie_{\text{condition}}$ Set 2

Set 1 \div Set 2

$\sigma_{\text{condition}}$ Set 1

$\pi_{\text{attribute-list}}$ Set 1

$\rho_{\text{renaming-specification}}$ Set 1

Relational Algebra as a Query Language for Relational Databases

Note: we don't normally use relational algebra directly.... products don't allow you to write relational algebra queries.

But...it's used internally in a DBMS – to represent a query plan.

Relational algebra is often used in theoretical work on databases.

One more detail before we start: a simple RA query with zero operators

Write the following query in relational algebra:

```
SELECT    *  
FROM      Student;
```

Answer: **Student**

A relation name, by itself, is a valid relational algebra query. Listing the relation, simply returns all of the tuples in the relation in the query answer.

Select operator (σ) in relational algebra

invented by Codd (not part of set theory)

Given the following relation ... (and instance)

Account

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

Consider the query: $\sigma_{\text{Balance} < 3000}$ Account

Select operator example

$\sigma_{\text{Balance} < 3000}$ Account

The select predicate is evaluated for each tuple

Account			
Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

Select operator in relational algebra

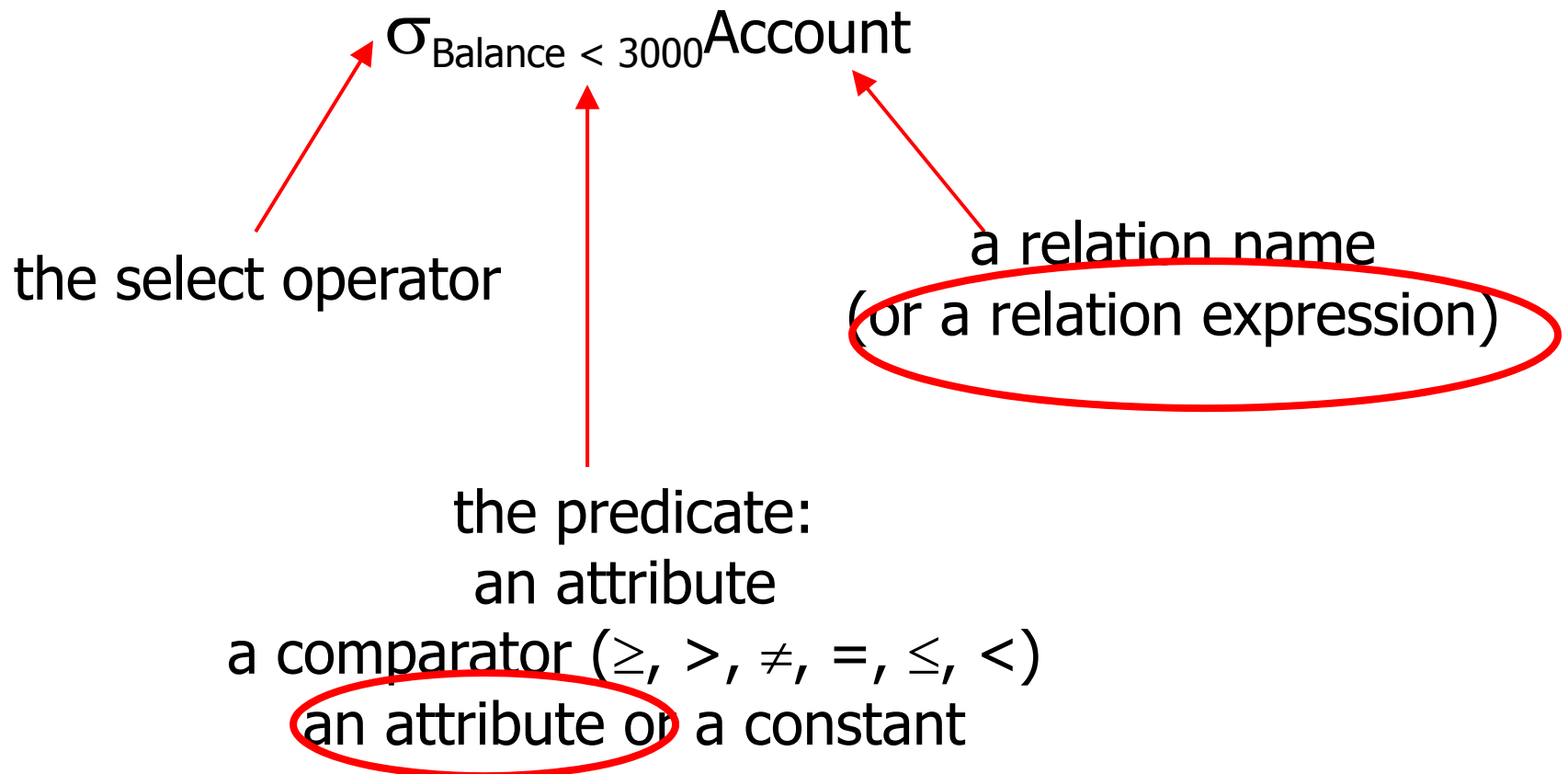
For this query: $\sigma_{\text{Balance} < 3000} \text{Account}$

The query answer is:

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
104	M. Jones	1000.00	checking

Select operator in relational algebra

Always applied to a single relation – a unary operator



Examples using the select operator

$\sigma_{\text{Balance} < 3000}$ Account

$\sigma_{\text{Number} = 103}$ Account

$\sigma_{\text{Balance} = \text{Number}}$ Account *Attribute compared to attribute!*

$\sigma_{\text{Type} = \text{"checking"}}$ ($\sigma_{\text{Balance} < 3000}$ Account) *relation expression!*

Account

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

Project operator (π) in relational algebra

Operator invented by Codd (not part of set theory)

Account

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

Consider the query: $\pi_{\text{Number, Owner}}$ Account

list of attributes (to retain) 

Project operator (π) in relational algebra

Always applied to single relation – a unary operator

For the query: $\pi_{\text{Number, Owner}} \text{Account}$

query answer is:

Number	Owner
101	J. Smith
102	W. Wei
103	J. Smith
104	M. Jones
105	H. Martin

Project operator (π): another example

Account

Number	Owner	Balance	Type
101	J. Smith	1000.00	checking
102	W. Wei	2000.00	checking
103	J. Smith	5000.00	savings
104	M. Jones	1000.00	checking
105	H. Martin	10,000.00	checking

Consider the query: $\pi_{\text{Owner}} \text{Account}$

list of attributes (to retain) 

Project operator example (cont.)

Consider the query: $\Pi_{\text{Owner}} \text{Account}$

Query answer is:

Owner
W. Wei
J. Smith
M. Jones
H. Martin

In relational algebra (as defined mathematically), relations are always sets. Thus the query answer is a set. J. Smith appears just once in the query answer!

Select and Project can be combined

Account		
Number	Owner	Balance
101	J. Smith	1000.0
102	W. Wei	2000.0
103	J. Smith	5000.0
104	M. Jones	1000.0
105	H. Martin	10,000.0

Note: two queries are equivalent if they are guaranteed to return the same query answer for every possible DB instance.

$\Pi_{\text{Owner}}(\sigma_{\text{Balance} < 3000}(\text{Account}))$

$\sigma_{\text{Balance} < 3000}(\Pi_{\text{Owner, Balance}}(\text{Account}))$

$\Pi_{\text{Owner}}(\sigma_{\text{Balance} < 3000}(\Pi_{\text{Owner, Balance}}(\text{Account})))$

Which pairs of these queries are equivalent, if any?

Cross Product an operator from set theory (repeated from earlier tonight)

Suppose.. $A = \{a, b, c\}$ $B = \{1, 2\}$

then in *set theory*, the cross product is defined as:

$$A \times B = \{(a, 1), (b, 1), (c, 1), (a, 2), (b, 2), (c, 2)\}$$

$A \times B$ is a set consisting of pairs (2-tuples) where each pair consists of an element from A and an element from B

Cross Product in Set Theory

Suppose.. $A = \{a, b, c\}$ $B = \{1, 2\}$ $C = \{x, y\}$ then

$$A \times B = \{(a, 1), (b, 1), (c, 1), (a, 2), (b, 2), (c, 2)\}$$

and $(A \times B) \times C =$

$$\begin{aligned} &\{((a, 1), x), ((b, 1), x), ((c, 1), x), ((a, 2), x), ((b, 2), x), \\ &((c, 2), x), ((a, 1), y), ((b, 1), y), ((c, 1), y), ((a, 2), y), \\ &((b, 2), y), ((c, 2), y)\} \end{aligned}$$

Relational Algebra vs. Set Theory

(cross product) (cont.)

Given $A = \{a, b, c\}$ $B = \{1, 2\}$ $C = \{x, y\}$ with the cross product $(A \times B) \times C$ in *set theory* =

$\{((a,1),x), ((b,1),x), ((c,1),x), ((a,2),x), ((b,2),x), ((c,2),x), ((a,1),y), ((b,1),y), ((c,1),y), ((a,2),y), ((b,2),y), ((c,2),y)\}$

Codd simplified it in *relational algebra* to:

$\{(a,1,x), (b,1,x), (c,1,x), (a,2,x), (b,2,x), (c,2,x), (a,1,y), (b,1,y), (c,1,y), (a,2,y), (b,2,y), (c,2,y)\}$

by eliminating parentheses.... "flattening" the tuples.

Cross product

- Is used in the basic definition of a relation

An instance of a relation is a subset of the cross product of its domains.

- Is also an operator in the relational algebra

So, we can use it in queries.

New Example Database – To Show how the Cross Product Operator works

Imagine that we have these two relations in a university database.

Teacher (t-num, t-name)

Course (c-num, c-name)

In reality, the relations would probably be more detailed with attribute names as follows:

Teacher (Number, Name, Office, E-mail)

Course (Number, Name, Description)

Taught-By (Quarter, Course, Section, Teacher, TimeDays) etc.

X cross product operator

Teacher	t-num	t-name
	101	Smith
	105	Jones
	110	Fong

Course	c-num	c-name
	586	Intro to DB
	533	Intro to OS

Teacher X Course

t-num	t-name	c-num	c-name
101	Smith	586	Intro to DB
105	Jones	586	Intro to DB
110	Fong	586	Intro to DB
101	Smith	533	Intro to OS
105	Jones	533	Intro to OS
110	Fong	533	Intro to OS

Cross product produces:
every possible combination of a teacher and a course

⋈ join operator

Account	Number	Owner	Balance	Type
Deposit	Account	Transaction-id	Date	Amount
Check	Account	Check-number	Date	Amount

$\sigma_{A.Number=Deposit.Account}$ (Account X Deposit)

is equivalent to

Account $\bowtie_{A.Number=Deposit.Account}$ Deposit

Join is defined for convenience

- Note: Join ($\text{Relation1} \bowtie_{A1=A2} \text{Relation2}$, where $A1$ is an attribute of Relation1 and $A2$ is an attribute of Relation2) is defined as $\sigma_{A1=A2}(\text{Relation1} \times \text{Relation2})$
- Thus...we don't really "need" the join operator. Any query with a join that we want to express, we can always use \times and σ . But ... join is used very frequently so it has been defined as an operator, for our convenience.
- It is also possible to define join (\bowtie) first and then define cross product (\times) based on join.

A few details about join

- Each simple Boolean predicate in the join condition must be comparing an attribute from one relation to an attribute in the other relation. In this query:

Account A \bowtie A.Number=D.Account AND D.type="checking" Deposit D

the “D.type = “checking” isn’t a JOIN condition.

- If you have a join with NO condition, then it is a cross product – by definition.

Join .. with all six comparators

- Student $|X|_{\text{advisor=number}}$ Faculty
- Student S $|X|_{\text{S.age} < \text{F.age}}$ Faculty F
- Student S $|X|_{\text{S.salary} \geq \text{F.salary}}$ Faculty F etc.
- Join is sometimes called “theta-join” or “ θ -join” where the θ represents any of the 6 comparators
- The most common join (with equality) is called equi-join

Challenge Question

- How could you convert the basic SQL statement (SELECT...FROM...WHERE...) statement to relational algebra (assuming that we wanted to eliminate all duplicates in the answer)?

```
SELECT  DISTINCT x, y, z  
FROM    Table1, Table2  
WHERE   .....
```

Basic SQL = SPJ

(except SQL uses tables vs. relations)

- `SELECT DISTINCT x, y, z`
`FROM Table1, Table2`
`WHERE`

Is the same as:

$\pi_{x, y, z} (\sigma \dots (\text{Table1 X Table2}))$

The basic `SELECT ... FROM ... WHERE ..`
is sometimes described as equivalent to the SPJ subset of
relational algebra (S = select, P = project, J = join)

SQL Queries – Many options

SELECT...
FROM...
WHERE...

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

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

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

(SELECT...FROM...WHERE...)

UNION

(SELECT...FROM...WHERE...)

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

GROUP BY...

HAVING ...

ORDER BY...

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

remember: each product may differ
in which features of SQL it supports

Sample Database

We use the following database for sample SQL queries:

Credit Rating Credit Amount Credit Balance
↓ ↓ ↓

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

Foreign key: Customer.Salesperson references Salesperson.Number

Salesperson (Number, Name, Address, Office)

Extensions to the SELECT clause

These queries are equivalent except for the attributes names in the query answer "Important-Customer" used in the 2nd query.

```
SELECT Name, Address  
FROM Customer  
WHERE CRating = 10;
```

```
SELECT Name AS Important-Customer, Address  
FROM Customer  
WHERE CRating = 10;
```

The query writer can assign new names to the columns in a query answer with 'AS <new attribute name>' clause.

More Extensions to the SELECT Clause

We can use aggregate operators in the SELECT clause:
COUNT, SUM, MIN, MAX, and AVG

```
SELECT MIN(CBalance), MAX(CBalance), AVG (CBalance)
FROM Customer;
```

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

If **one aggregate operator appears** in the SELECT clause, **then ALL** of the entries in the select clause **must be an aggregate operator** (unless the query includes a **GROUP BY** clause (covered later)).

Does this make sense?

```
SELECT      S.Name, S.Phone, AVG(S.Age)
FROM        Student S
WHERE       S.Major = "CS";
```

Consider the data:

Student	ID	Name	Phone	Age	Major
	1	Joe	123	24	CS
	2	Mary	456	28	CS
	3	Arun	789	32	CS
	4	John	999	18	English

WHAT would the query answer be? This SQL query is NOT well-formed and thus not allowed.

More Extensions to the SELECT Clause

What is the difference between these two queries?

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

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

When will these two queries return the same answer? (That is, for what sort of database instance would these queries return the same answer?)

Challenge Questions

What is the implication of using **DISTINCT** when computing the **SUM** or **AVG** of an attribute?

SUM(DISTINCT(age)) vs.
SUM (age)

What is the implication of using **DISTINCT** when computing the **MIN** or **MAX** of an attribute?

MIN(DISTINCT(age)) vs.
MIN(age)

Challenge Questions Answers

What is the implication of using DISTINCT when computing the SUM or AVG of an attribute?

Answer: The SUM or AVG will be computed only on distinct values (compared to the normal case - without DISTINCT - where the SUM and AVG are computed on all values)

What is the implication of using DISTINCT when computing the MIN or MAX of an attribute?

Answer: There is no difference. The maximum or minimum value is the same whether or not duplicates are eliminated ahead of time.

More Extensions to the SELECT clause

The SELECT clause list can also include simple arithmetic expressions using +, -, *, and /.

```
SELECT (CAmount - CBalance) AS AvailableCredit,  
       Name  
FROM   Customer  
WHERE  CAmount > 0;
```

Note: it is very common to use "AS" to name computed columns in queries.

This query computes the available credit, for those Customers that have CAmount > 0.

SQL ... Extensions

SELECT...
FROM...
WHERE...

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

Note: we have already seen correlation names.

Extensions to the FROM clause - Joins

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

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

Extensions to the FROM clause - Joins

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

- inner join (the regular join) ←
 - cross join ←
 - natural join ←
 - left outer join
 - right outer join
 - full outer join
- ... these joins can be expressed in a basic SELECT..FROM..WHERE query.
- These JOIN operators in the FROM clause are a bit like syntactic sugar.

Extensions to the FROM clause - Joins

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

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

There are
new
operators ...

join clause in FROM (with ON)

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.Salesperson = S.Number
WHERE C.CreditRating < 6;
```

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

Join queries: SQL and equivalent relational algebra

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

$$\Pi_{C.Name, S.Name} (\sigma_{C.CreditRating < 6} (Customer \bowtie_{C.Salesperson=S.Number} Salesperson))$$

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

$$\Pi_{C.Name, S.Name} (\sigma_{C.CreditRating < 6 \text{ AND } C.Salesperson=S.Number} (Customer \times Salesperson))$$

JOIN with USING Clause (when join attributes in the 2 tables have the same name)

Course (CNumber, CName, Description)

Teacher (TNumber, TName, Phone)

Offering (CNumber, TNumber, Time, Days, Room)

These two queries are equivalent:

```
SELECT    C.CNumber, C.CName, Room
FROM      Course C JOIN Offering USING (CNumber);
```

```
SELECT    C.Number, C.Name, Room
FROM      Course C JOIN Offering O
           ON C.CNumber = O.CNumber;
```

Note: USING clause doesn't need (and can't have) a correlation name

Basic Join \equiv INNER JOIN

For the INNER JOIN,

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

the query answer includes all “matches”
but does not include:

- a Customer that doesn't have a Salesperson
- nor
- a Salesperson that is not assigned to any customers.

Basic Join \equiv INNER JOIN is the Default (the word "INNER" is optional)

This INNER JOIN query:

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

is the same as:

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

CROSS JOIN

A “CROSS JOIN” is a cross product

The following queries are equivalent:

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

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

Write this query in relational algebra.

Equi-join vs. Natural Join

equi join: Account \bowtie Number=Account Deposit

When the join is based on equality, then we always have two identical attributes (columns) in the answer.

Number	Owner	Balance	Type	Account	Trans-id	Date	Amount
102	W. Wei	2000.00	checking	102	1	10/22/00	500.00
102	W. Wei	2000.00	checking	102	2	10/29/00	200.00
104	M. Jones	1000.00	checking	104	3	10/29/00	1000.00
105	H. Martin	10,000.00	checking	105	4	11/2/00	10000.00

The **natural join** eliminates the duplicate column for joins based on equality. (This is slightly different terminology from the book.)

More on natural join

- Natural join requires attributes with the same name in the two relations. For the previous example, we would need to rename “Account” to “Number” or vice versa to use natural join.
- Natural join (when you don’t specify which attributes are to be joined on) will join on ALL ATTRIBUTES WITH THE SAME NAME!

NATURAL JOIN

NATURAL JOIN ... like a “macro” that joins tables with an equality check for all attributes **with the same name**.

Consider the following database

Course (CNumber, CName, Description)

Teacher (TNumber, TName, Phone)

Offering (CNumber, TNumber, Time, Days, Room)



Natural Join Drops Columns (automatically)

With any join based on equality, **there will always be pairs of identical columns** (one for each column that is joined).

The NATURAL JOIN **eliminates one** of the duplicate columns.

Challenge question: what is the attribute name for the column that is retained?

Natural Join

List course name and teacher name for all course offerings. This query can be expressed with the NATURAL JOIN or with an INNER JOIN.

These two queries are equivalent.

```
SELECT      CName, TName
FROM        Course NATURAL JOIN Offering
           NATURAL JOIN Teacher;
```

```
SELECT      CName, TName
FROM        Course C, Offering O, Teacher T
WHERE       C.CNumber = O.CNumber AND
           O.TNumber = T.TNumber;
```

Because the join attributes have the same attribute names.

Natural Join

Will the columns with the same name appear in the query answer twice? Or once?

Products may differ ... regarding when they drop duplicate columns ...

```
SELECT *  
FROM   Course NATURAL JOIN Offering  
       NATURAL JOIN Teacher;
```

```
SELECT *  
FROM   Course C, Offering O, Teacher T  
WHERE  C.CNumber = O.CNumber AND  
       O.TNumber = T.TNumber;
```

NATURAL JOIN (risky ...)

What if I had different attribute names?

Course (CNumber, Name, Description)

Teacher (TNumber, Name, Phone)

Offering (CNumber, TNumber, Time, Days, Room)

SELECT *

FROM Course NATURAL JOIN Offering
NATURAL JOIN Teacher;

Then the it would join O.CNumber to C.CNumber AND
O.TNumber AND C.Name to T.Name!!!

INNER JOIN vs OUTER JOIN

For the INNER JOIN,

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

the query answer does not include:

a Customer that doesn't have a Salesperson

nor

a Salesperson that is not assigned to any customers.

INNER JOIN vs OUTER JOIN

an INNER (regular) JOIN includes only those customers that have salespersons (only the matches)

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

a LEFT OUTER JOIN will include **all matches** plus all **Customers that don't have a Salesperson**

a RIGHT OUTER JOIN will include **all matches** plus all **Salespersons that are not assigned to any customers**

a FULL OUTER JOIN will include all of these!

LEFT OUTER JOIN

Customer

Number	Name	Address	CRating	CAmount	CBalance	Salesperson
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.Salesperson = S.Number gives us:

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

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

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

3 wei zzz 10 10,000 10,000 <null> <null> <null> <null> <null>

RIGHT OUTER JOIN

Customer

Number	Name	Address	CRating	CAmount	CBalance	Salesperson
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.Salesperson = S.Number gives:

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

RIGHT OUTER JOIN on C.Salesperson = S.Number gives:

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

<null> <null> <null> <null> <null> <null> <null> 102 miller bbb 26

FULL OUTER JOIN

Customer

Number	Name	Address	CRating	CAmount	CBalance	Salesperson
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	millier	bbb	26

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

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

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

1 smith xxx 5 1,000 1,000 101 101 johnson aaa 23

1 jones yyy 7 5,000 4,000 101 101 johnson aaa 23

3 wei zzz 10 10,000 10,000 <null> <null> <null> <null> <null>

<null> <null> <null> <null> <null> <null> <null> 102 millier bbb 26

LEFT OUTER JOIN

Student				Faculty		
Number	Name	Age	Advisor	Number	Name	Age
1	Bob	22	102	101	Todd	35
2	Sally	19	<null>	102	Beth	22
3	Ted	19	101	103	Anya	<null>

```
SELECT *  
FROM Student S LEFT OUTER JOIN Faculty F  
ON S.Advisor = F.Number
```

--	--	--	--	--	--	--

LEFT OUTER JOIN Exercise

Student				Faculty		
Number	Name	Age	Advisor	Number	Name	Age
1	Bob	22	102	101	Todd	35
2	Sally	19	<null>	102	Beth	22
3	Ted	19	101	103	Anya	<null>

```
SELECT S.Name AS SName, F.Name AS FName  
FROM Student S LEFT OUTER JOIN Faculty F  
  ON S.Advisor = F.Number  
WHERE F.Name <> 'Beth'
```

--	--

RIGHT OUTER JOIN Exercise

Student				Faculty		
Number	Name	Age	Advisor	Number	Name	Age
1	Bob	22	102	101	Todd	35
2	Sally	19	<null>	102	Beth	22
3	Ted	19	101	103	Anya	<null>

```
SELECT *  
FROM Student S RIGHT OUTER JOIN Faculty F  
ON S.Advisor = F.Number
```

--	--	--	--	--	--	--

FULL OUTER JOIN Exercise

Student

Number	Name	Age	Advisor
1	Bob	22	102
2	Sally	19	<null>
3	Ted	19	101

Faculty

Number	Name	Age
101	Todd	35
102	Beth	22
103	Anya	<null>

SELECT *

FROM Student S FULL OUTER JOIN Faculty F

ON S.Age = F.Age

--	--	--	--	--	--	--





Another Extension to the FROM clause

- You can put a complete query expression, in parentheses, in the FROM clause.
- Note: not all DBMSs support this option.

```
SELECT ...  
FROM Employee E, (SELECT ... FROM ...)  
WHERE ...
```

Relational Algebra Operators

There are eight operators

- π project 
- σ select 
- \cup union
- \cap intersection
- $-$ difference
- \times cross product 
- \bowtie join 
- \div divide
- renaming (to rename a relation and attributes)

Four operators
that you've already
seen

Relational Algebra Operators

There are eight operators

- π project
- σ select
- \cup union
- \cap intersection
- $-$ difference
- \times cross product
- \bowtie join
- \div divide
- renaming (to rename a relation and attributes)

Three operators
from set theory

Intersection, Union, Difference

\cup UNION

\cap INTERSECTION

$-$ SET DIFFERENCE

These operators can only be used with relations that are “union-compatible”.

Union Compatible

- Two relations are *union-compatible* if they have the same degree (i.e., the same no. of attributes) and the corresponding attributes are defined on the same domains. (Some books also require same attr. names.)
- Suppose we have these relations:

Checking-Account (c-num, c-owner, c-balance)

Savings-Account (s-num, s-owner, s-balance)

These are *union-compatible* relations.

- Union, intersection, & difference all require union-compatible relations

Relational Algebra Operators

\cup union Checking-account \cup Savings-account

Checking-account	c-num	c-owner	c-balance
------------------	-------	---------	-----------

101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00

Savings-account	s-num	s-owner	s-balance
-----------------	-------	---------	-----------

103	J. Smith	5000.00
-----	----------	---------

	c-num	c-owner	c-balance
--	-------	---------	-----------

101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00
103	J. Smith	5000.00

Notice: attributes names are from the 1st relation in query.

Intersection Example

\cap intersection

Checking-account \cap Savings-account

What's the answer to this query?

$(\pi_{c\text{-owner}} \text{Checking-account}) \cap (\pi_{s\text{-owner}} \text{Savings-account})$

What's the answer to this new query?

Intersection Example – query answer

Checking-account \cap Savings-account

What's the answer to this query?

It's empty. There are no tuples that are in both relations.

$(\pi_{c\text{-owner}} \text{Checking-account}) \cap (\pi_{s\text{-owner}} \text{Savings-account})$

What's the answer to this new query?

	c-owner J. Smith
--	---------------------

Set difference

— difference

Checking-account — Savings-account

Find all the tuples (rows) that are in the Checking-account relation that are not in the Savings-account relation.

$(\pi_{c\text{-owner}} \text{Checking-account}) -$
 $(\pi_{s\text{-owner}} \text{Savings-account})$

SQL – for Union, intersection, Difference

SELECT...

FROM...

WHERE...

(SELECT...FROM...WHERE...)

UNION

(SELECT...FROM...WHERE...)

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

SQL UNION

```
(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 the UNION operator in between.

SQL INTERSECTION

```
(SELECT C.Name  
FROM Customer C)
```

INTERSECT

```
(SELECT S.Name  
FROM Salesperson S);
```

Two complete queries - with the INTERSECT operator in between.

SQL EXCEPT (or MINUS) (set difference)

```
(SELECT S.SalespersonNum  
FROM Salesperson)
```

EXCEPT

```
(SELECT C.SalespersonNum  
FROM Customer C);
```

Two complete queries - with the EXCEPT (or MINUS) operator in between.

Using the ALL keyword

- UNION vs. UNION ALL
- INTERSECT vs. INTERSECT ALL
- EXCEPT vs. EXCEPT ALL

- Set vs. bag semantics



Union, Intersect, Except (without ALL)

- If you don't specify ALL, then the answer is computed on sets. You can think of it like this:
 - Eliminate duplicates from first operand
 - Eliminate duplicates from second operand
 - THEN ... compute union, intersect, or difference
 - Eliminate duplicates from query answer

Using the ALL keyword

- UNION ALL:
 - A has 3 copies of a row, B has 5
 - $N + M$ (A \cup B has 8 copies of the row)
 - INTERSECT ALL:
 - A has 3 copies of a row, B has 5
 - $\text{MIN}(N, M)$ (A \cap B has 3 copies of the row)
 - EXCEPT ALL:
 - A has 3 copies of a row, B has 5
 - A has 5 copies of a row, B has 3
 - $\text{MAX}(N - M, 0)$ (A - B has 0 copies; A - B has 2)
-

Set Operations Implementation

	SQL Server 2000	SQL Server 2005	Oracle 10g	Postgre SQL	MySQL 5.1
UNION	X	X	X	X	X
UNION ALL	X	X	X	X	X
INTERSECT		X	X	X	
INTERSECT ALL				X	
EXCEPT		X	X	X	
EXCEPT ALL				X	

Using INTERSECT

Table 1

Row A
Row P
Row D
Row R
Row U
...



**If this row is in
the other table,
keep it.**

**Otherwise,
drop it.**

Table 2

Row D
Row Y
Row Z
Row Q
Row J
...



Using EXCEPT

Table 1

Row A
Row P
Row D
Row R
Row U
...



**If this row is in
the other table,
drop it.**

**Otherwise,
keep it.**

Table 2

Row D
Row Y
Row Z
Row Q
Row J
...



Sets vs. Bags (also called multi-sets)

With your partner, construct 3 sets

- Take the union of two of your sets
- Take the intersection of your sets
- Take the set difference

Now construct 3 bags

- Take the union of your two bags, produce a bag
- Take that intersection and set difference of your bags

Why do we use Relational Algebra?

Because:

- It is mathematically defined (where relations are sets)
- We can prove that two relational algebra expressions are equivalent. For example:

$$\begin{aligned} \sigma_{\text{cond1}} (\sigma_{\text{cond2}} R) &\equiv \sigma_{\text{cond2}} (\sigma_{\text{cond1}} R) \equiv \sigma_{\text{cond1 and cond2}} R \\ &\equiv (\sigma_{\text{cond1}} R) \cap (\sigma_{\text{cond2}} R) \end{aligned}$$

$$R1 \bowtie_{\text{cond}} R2 \equiv \sigma_{\text{cond}} (R1 \times R2)$$

$$\sigma_{\text{cond1 OR cond2}} R \equiv (\sigma_{\text{cond1}} R) \cup (\sigma_{\text{cond2}} R)$$

Equivalences for “AND”, “OR”, and “NOT”

$$\sigma_{\text{cond1 OR cond2}} R \equiv (\sigma_{\text{cond1}} R) \cup (\sigma_{\text{cond2}} R)$$

$$\sigma_{\text{cond1 AND cond2}} R \equiv (\sigma_{\text{cond1}} R) \cap (\sigma_{\text{cond2}} R)$$

$$\sigma_{\text{cond1 AND NOT cond2}} R \equiv (\sigma_{\text{cond1}} R) - (\sigma_{\text{cond2}} R)$$

The WHERE clause (and the predicate for the σ operator) may contain AND, OR, as well as NOT. Such expressions are implicitly using \cup , \cap , and $-$ operators.

Uses of Relational Algebra Equivalences

- To help query writers - they can write queries in several different ways
- To help query optimizers - they can choose among different ways to execute the query

and in both cases **we know for sure** that the two queries (the original and the replacement) are identical...that they will produce the same answer

SQL ... Extensions

SELECT...

FROM...

WHERE...

GROUP BY...

HAVING ...

ORDER BY...

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

Note that “ORDER BY” simply sorts the query answer. It is considered an “implementation specific” feature of SQL

GROUP BY, HAVING

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

```
SELECT      Salesperson  
FROM        Customer  
GROUP BY    Salesperson  
HAVING      Count(*) > 1;
```

Any form of SQL query (e.g., with or without subqueries) can have the answer “grouped”. When a query answer is grouped, there is one output row - for each group.

GROUP BY, HAVING

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

```
SELECT      Salesperson  
FROM        Customer  
GROUP BY    Salesperson  
HAVING      Count(*) > 1;
```

The **HAVING clause** is a **predicate evaluated against each group**. A group participates in the query answer if it satisfies the HAVING predicate

Group by example

Input data:

Customer

(Number, Name, Address, ... Salesperson)

101	Mary	...	5
102	John	...	8
103	Quan	...	NULL
106	Susan	...	5
107	David	...	5
109	Mike	...	2
110	Ying	...	8

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

Example: Group by

Input data:

Customer	(<u>Number</u> , Name, Address, ... Salesperson)
101	Mary ... 5
102	John ... 8
103	Quan... NULL
106	Susan ... 5
107	David ... 5
109	Mike ... 2
110	Ying ... 8

Intermediate result:
4 groups

103	Quan...	Null
101	Mary ...	5
106	Susan ...	5
107	David ...	5
102	John ...	8
110	Ying ...	8
109	Mike ...	2

Example: Group by

Intermediate result:
4 groups

103	Quan...	Null	
101	Mary ...		5
106	Susan	...	5
107	David	...	5
102	John	...	8
110	Ying	...	8
109	Mike	...	2

Query answer:
one row per group!

Salesperson	Count
Null	1
5	3
8	2
2	1

SELECT Salesperson
COUNT(*)
FROM Customer
GROUP BY Salesperson;

Example: Group by with Having

Intermediate result:

4 groups

103	Quan ...		
101	Mary ...		5
106	Susan	...	5
107	David	...	5
102	John	...	8
110	Ying	...	8
109	Mike ...		2

Query answer:

one row per group
that satisfies the
HAVING clause!

Salesperson	Count
5	3
8	2

SELECT Salesperson
FROM Customer
GROUP BY Salesperson
HAVING Count(*) > 1;

GROUP BY, HAVING

SELECT Salesperson, COUNT(*)
FROM Customer
GROUP BY Salesperson;

SELECT Salesperson
FROM Customer
GROUP BY Salesperson
HAVING Count(*) > 1;

The only attributes that can appear in a “grouped” query answer are **the grouping attribute** or **aggregate operators** (that are applied to the group).

Group By - Example

The following query is **not legal**

```
SELECT      Name
FROM        Customer
GROUP BY    SalespersonNUM;
```

...because there can be many names for each group!

French Fries Warmup

- What is the size of the largest size item that each store sells?
- For each store that sells at least 4 distinct beverages*, what is the cost of the highest price item that the store sells?
- For each beverage that comes in at least 3 sizes (ignoring what store it is sold in) , what is its minimum cost (ignoring size) ?

* Two rows represent distinct beverages if their "Beverage" column differs.