

SQL

SQL

- ❖ Based primarily on Relational Algebra with some features from Relational Calculus
- ❖ Components: Data definition language, Manipulation language
- ❖ Other SQL features
 - Transaction definition: end of query by default
 - Security, Views, Index
- ❖ Embedded SQL – embed SQL commands in a general purpose language
- ❖ Database connectivity packages allow queries to be passed to DB from applications – JDBC

SQL-Data Definition language

- ❖ We've seen the DDL component of SQL
 - CREATE TABLE
 - ALTER TABLE
 - Key constraints
 - PRIMARY KEY(<>)
 - UNIQUE (<>)
 - Foreign Key/Referential Integrity
 - FOREIGN KEY (<>) REFERENCES <table name>
 - Other constraints?
 - Will see later

Basic SQL Query

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
```

- ❖ relation-list A list of relation names (possibly with a *range-variable, i.e., tuple variable*, after each name).
- ❖ attribute-list A list of attributes of relations in *relation-list*
- ❖ Qualification/predicate Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of <, >, =, ≤, ≥, ≠) combined using AND, OR and NOT.
- ❖ **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

Conceptual Evaluation Strategy

- ❖ *Semantics* of an SQL query defined in terms of the following conceptual evaluation strategy:
- ❖ Compute the cross-product of *relation-list*.
- ❖ Discard resulting tuples if they fail *predicate qualifications*.
- ❖ Delete attributes that are not in *target attribute-list*.
 - If **DISTINCT** is specified, eliminate duplicate rows.
 - SQL allows duplicates in relations (unlike Rel. Algebra)
- ❖ This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

CS 178

5

Equivalence with Rel. Algebra

- ❖ **Select** <attribute-list> = Project $\Pi_{(\text{attribute-list})}$
- ❖ **From** < r_1, r_2, \dots > = Cartesian Product $r_1 \times r_2 \dots$
- ❖ **Where** <predicate> = Select $\sigma_{(\text{predicate})}$
 - consisting of one or more conditions connected by logical connectives (and, or, not)
 - If no condition then ALL tuples selected
- ❖ SQL query = $\Pi_{(\text{attr-list})} \sigma_{(\text{predicate condition})} (R \times S)$
- ❖ NOTE: SQL has no natural join operator
 - Need to build cross product and then specify predicate (qualifiers) that forces the join condition

CS 178

6

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT *
FROM Product
WHERE category='Gadgets'
```



PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks

“selection”

CS 178

7

Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Price > 100
```



PName	Price	Manufacturer
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

“selection” and
“projection”

CS 178

8

Eliminating Duplicates

```
SELECT DISTINCT category
FROM Product
```

Category
Gadgets
Photography
Household

Compare to:

```
SELECT category
FROM Product
```

Category
Gadgets
Gadgets
Photography
Household

Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under \$200 manufactured in Japan;
return their names and prices.

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
```

Join
between Product
and Company

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
```

PName	Price
SingleTouch	\$149.99

A Join Subtlety

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all countries that manufacture some product in the
'Gadgets' category.

```
SELECT Country
FROM Product, Company
WHERE Manufacturer=CName AND Category='Gadgets'
```

A Join Subtlety

Product				Company		
Name	Price	Category	Manufacturer	Cname	StockPrice	Country
Gizmo	\$19.99	Gadgets	GizmoWorks	GizmoWorks	25	USA
Powergizmo	\$29.99	Gadgets	GizmoWorks	Canon	65	Japan
SingleTouch	\$149.99	Photography	Canon	Hitachi	15	Japan
MultiTouch	\$203.99	Household	Hitachi			

```
SELECT Country
FROM Product, Company
WHERE Manufacturer=CName AND Category='Gadgets'
```

↓

Country
??
??

What is the problem?
What's the solution?

CS 178

13

Tuple/Range variables

- ❖ Concept of tuple/range variables borrowed from relational calculus
 - Tuple t of type R : $t \in R$
 - *What about $x \in R, y \in R$*
- ❖ It performs the job of the rename operator in relational algebra
 - One variable with name x and one with name y
- ❖ Need to worry about scope of tuple variables when we have nested queries

CS 178

14

Tuple Variables

Person(pname, address, worksfor)
Company(cname, address)

```
SELECT DISTINCT pname, address
FROM Person, Company
WHERE worksfor = cname
```

Which address?

→

```
SELECT DISTINCT Person.pname, Company.address
FROM Person, Company
WHERE Person.worksfor = Company.cname
```

→

```
SELECT DISTINCT x.pname, y.address
FROM Person AS x, Company AS y
WHERE x.worksfor = y.cname
```

CS 178

15

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, ..., ak
FROM R1 x1, R2 x2, ..., Rn xn
WHERE Conditions
```

```
Answer = {}
for x1 in R1 do
  for x2 in R2 do
    .....
    for xn in Rn do
      if Conditions
        then Answer = Answer ∪ {(a1, ..., ak)}
```

CS 178

16

A Note on Tuple/Range Variables

- ❖ Really needed only if the same relation appears twice in the FROM clause. The query can be written as:

```
SELECT C.name
FROM Customer C, Deposit D
WHERE C.custid=D.custid
```

OR

```
SELECT name
FROM Customer, Deposit
WHERE
Customer.custid=Deposit.custid
```

It is good style, however, to use range variables always!

CS 178

17

Schema of Bank DB

- ❖ Customer (CustID, Name, street,city,zip)
 - Customer ID, Name, and Address info: street, city, zip
- ❖ Deposit (CustID, Acct-num, balance,Branch-name)
 - Customer ID, Account number, Balance in account, name of branch where account is held
- ❖ Loan (CustID, Loan-num, Amount, Branch-name)
 - Customer ID, loan number, amount of loan...
- ❖ Branch (Branch-name, assets, Branch-city)

CS 178

18

Tuple variables

- ❖ Find customers (ID) who have an account at a branch where CustID 6666 has an account.
- ❖ Need to access Deposit table twice
 - Once to extract branch X of CustID 6666
 - Second time to find accounts at these branches X
 - Define two "variables" A,B of 'type' Deposit
 - B is variable that corresponds to CustID 6666 and its branch-name field is equal to "X"
 - A is a variable whose branch-name is equal to "X"

CS 178

19

Tuple variables

- ❖ Find customers (ID) who have an account at a branch where CustID 6666 has an account.
- ```
SELECT A.CustID
FROM Deposit A, Deposit B
WHERE B.CustID=6666
 AND A.branchname=B.branchname
```
- B is a variable that refers to Customer 6666
  - Variable A can take on any values that satisfy the query

CS 178

20

Find customers who have an account and live in NY



Find customers who have an account and live in NY



```
SELECT C.Custid
FROM Customer C, Deposit D
WHERE C.custid=D.custid
AND C.city=NY
```

- ❖ Would adding DISTINCT to this query make a difference?
- ❖ What is the effect of replacing *C.custid* by *C.name* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

## Expressions and Strings



```
SELECT C.CustID AS cid
FROM Customer C
WHERE C.street LIKE '%Main%'
```

- ❖ Illustrates use of arithmetic expressions and string pattern matching: *Find customers whose street name includes 'Main'.*
- ❖ **AS** and **=** are two ways to name fields in result.
- ❖ **LIKE** is used for string matching. `'_'` stands for any one character and `'%'` stands for 0 or more arbitrary characters.

## Set Operations in SQL

Find all customers (IDs) who have a loan or an account at the Downtown branch



- ❖ **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- ❖ **IMPORTANT NOTE**: In set operations, SQL removes duplicates
- ❖ Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

## Set Operations in SQL

Find all customers (IDs) who have a loan or an account at the Downtown branch



- ❖ **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- ❖ **IMPORTANT NOTE**: In set operations, SQL removes duplicates
- ❖ Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```
SELECT CustID
FROM Loan
WHERE branchname='Downtown'
UNION
SELECT CustID
FROM Deposit
WHERE branchname='Downtown';
```

CS 178

25

Find all customers (IDs) who have a loan and an account at the Downtown branch



- ❖ **INTERSECT**: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- ❖ Included in the SQL/92 standard, but some systems don't support it.

```
SELECT D.CustID
FROM Deposit D, Loan L
WHERE D.CustID=L.CustID
AND D.branchname='Downtown'
AND D.branchname=L.branchname;
```

```
SELECT CustID Key field!
FROM Loan
WHERE branchname='Downtown'
INTERSECT
SELECT CustID
FROM Deposit
WHERE branchname='Downtown';
```

CS 178

26

## Nested Queries



- ❖ Nested query has a sub-query nested in the WHERE clause
  - When query needs to refer to a set of tuples that need to be computed (and are not stored as a relation)
  - Can also appear in FROM clause
  - Need to be careful about scope of tuple variables
    - Scoping rules: local definition and then global
    - In subquery - legal to use only tuple variables defined in subquery itself or in any query that contains the subquery

CS 178

27

## Nested Queries: Semantics



- ❖ Evaluate subquery at each reference
  - Construct cross product of tables in FROM clause
  - For each row when testing predicate conditions in WHERE clause
    - Recompute subquery
      - Is this really necessary?
    - If subquery contains another subquery then apply same principle

CS 178

28

## Subqueries Returning Relations and Set Membership operators

Company(name, city)  
Product(pname, maker)  
Purchase(id, product, buyer)

Return cities of companies that manufacture products bought by Joe Blow

```
SELECT Company.city
FROM Company
WHERE Company.name IN
 (Set of Companies that manufacture
 products bought by Joe Blow);
/* write a SELECT query to obtain this set */
```

CS 178

29

## Subqueries Returning Relations

Company(name, city)  
Product(pname, maker)  
Purchase(id, product, buyer)

Return cities of companies that manufacture products bought by Joe Blow

```
SELECT Company.city
FROM Company
WHERE Company.name IN
 (SELECT Product.maker
 FROM Purchase, Product
 WHERE Product.pname=Purchase.product
 AND Purchase.buyer = 'Joe Blow');
```

CS 178

30

## Example Bank Database

- ❖ Customer(CustID, Name, Street, City, Zip)
- ❖ Deposit(CustID, Acct-Num, branch-name, balance)
- ❖ Loan(CustID, Acct-num, branch-name, balance)
- ❖ Branch(branch-name, assets, city)

CS 178

31

## Set Membership Operations: (a)

- ❖ Can check for set membership using **IN** and **NOTIN**
  - $x \text{ IN } A$  or  $x \text{ NOTIN } A$ 
    - Implements Relational Calculus operators
  - **IN** connective tests for membership in the set A
    - Set A may be produced by a **SELECT**
  - **NOTIN** tests for absence of tuples
  - Can test using multiple attribute element
- ❖ Set existence using **EXISTS**
  - Returns true if the argument subquery is nonempty (the converse for the **NOT EXISTS**) thus checking for empty relations

CS 178

32

Find all customers who have both a loan and an account at the Downtown branch

- ❖ Find all account holders at Downtown branch
  - Call this set A
  - SELECT CustID
  - FROM Deposit
  - WHERE branch-name="Downtown"
- ❖ From loan relation, select those customers who also appear in set of account holders
  - If customer is in set A, then select in result
    - Branch-name="Downtown" and CustID IN A

CS 178

33

Find all customers who have both a loan and an account at the Downtown branch

- ❖ Find all account holders at Downtown branch
  - ❖ From loan relation, select those customers who also appear in set of account holders
  - ❖ The embedded/nested select selects customers with accounts
  - ❖ Note that this query was written earlier without using nested queries
- ```
SELECT distinct CustID
FROM Loan
WHERE branchname='Downtown'
AND
CustID IN
(SELECT CustID
FROM Deposit
WHERE
branchname='Downtown');
```

CS 178

34

Set Membership: Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make some products with price < 100

Existential: easy ! ☺

CS 178

35

Set Membership: Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make some products with price < 100

```
SELECT DISTINCT Company.cname
FROM Company, Product
WHERE Company.cname = Product.company and Product.price < 100
```

Existential: easy ! ☺

CS 178

36

Set Membership: Quantifiers

Product (pname, price, company)
Company(cname, city)

Find all companies that make only products with price < 100

same as:

Find all companies such that all of their products have price < 100
Recall equivalence: $\text{Forall } x P(x) = \text{Not Exists } x (\text{Not } P(x))$

Universal: hard ! ☹

Set Membership: Quantifiers

1. Find *the other* companies: i.e. s.t. some product ≥ 100

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
                        FROM Product
                        WHERE Product.price >= 100)
```

2. Find all companies s.t. all their products have price < 100

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname NOT IN (SELECT Product.company
                             FROM Product
                             WHERE Product.price >= 100)
```

More Set Membership Operations

- ❖ Previous operators allowed checking for existence
- ❖ SQL provides operators to test elements of one set A with elements on another set B
 - **SOME: op SOME**
 - Also called as **ANY** in some versions
 - **ALL: op ALL**
 - op can be >=, >, <, <=, =, not=
- ❖ Test single value against members of an entire set
 - $X > \text{ALL} (R)$

Find branches that have greater assets than some branch located in Brooklyn

- ❖ A=Set of branches in Brooklyn
Branch.assets >some
- ❖ Compare assets of branch to members of set A
(assets of branches in Brooklyn)
 - Branch is selected if its assets is greater than assets of SOME branch in Brooklyn

Find branches that have greater assets than some branch located in Brooklyn

```
SELECT branchname
FROM Branch
WHERE assets > some
      (SELECT assets
       FROM Branch
       WHERE
        branchcity='Brooklyn');
```

- ❖ Inner select has set of assets of branches in Brooklyn
- ❖ For branches that have assets greater than all branches in Brooklyn replace >some with >all

Set Comparison Operations: subset

- ❖ Check if one set (query result) contains another set (query result)
 - Is A subset of B?
 - Is A not a subset of B?
- ❖ **contains** and **not contains** operators

*Set Membership Examples:
Find all customers who have an account at all branches located in Brooklyn*

```
SELECT CustID
FROM Deposit S
WHERE
  (set of branchnames where customer has an
   account)
contains
  (set of of all branchnames located in
   Brooklyn)
```

```
SELECT S.CustID
FROM Deposit S
WHERE (SELECT T.branchname
       FROM Deposit T
       WHERE S.CustID=T.CustID)
CONTAINS
  (SELECT branchname
   FROM Branch
   WHERE branchcity='Brooklyn');
```

Set existence example

- ❖ Test if subquery is empty
 - Exists returns true if argument is nonempty
- ❖ Find all customers who have both an account and a loan at Downtown branch
- ❖ First test if customer has account and second test if customer has loan
 - Exists in Deposit and Exists in Loan

```
SELECT C.CustID
FROM Customer C
WHERE exists (SELECT *
              FROM Deposit D
              WHERE D.CustID=C.CustID
                 AND D.branchname='Downtown')
AND exists (SELECT *
            FROM Loan L
            WHERE L.CustID= C. CustID
              AND L.branchname='Downtown');
```

NULLS in SQL

- ❖ Whenever we don't have a value, we can put a NULL
- ❖ Can mean many things:
 - Value does not exist
 - Value exists but is unknown
 - Value not applicable
 - Etc.
- ❖ The schema specifies for each attribute if it can be null (*nullable* attribute) or not
 - NOT NULL after declaring attribute domain
- ❖ How does SQL cope with tables that have NULLs ?

Null Values

- ❖ If $x = \text{NULL}$ then $4*(3-x)/7$ is still NULL
- ❖ If $x = \text{NULL}$ then $x = \text{"Joe"}$ is UNKNOWN
- ❖ In SQL there are three boolean values:
 - FALSE = 0
 - UNKNOWN = 0.5
 - TRUE = 1

Null Values

- ❖ $C1 \text{ AND } C2 = \min(C1, C2)$
- ❖ $C1 \text{ OR } C2 = \max(C1, C2)$
- ❖ $\text{NOT } C1 = 1 - C1$

```
SELECT *  
FROM Person  
WHERE (age < 25) AND  
      (height > 6 OR weight > 190)
```

E.g.
age=20
height=NULL
weight=200

Rule in SQL: include only tuples that yield TRUE

Null Values

Unexpected behavior:

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- $x \text{ IS NULL}$
- $x \text{ IS NOT NULL}$

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25 OR age IS NULL
```

Now it includes all Persons

Outerjoins

Explicit joins in SQL = "inner joins":

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store  
FROM Product JOIN Purchase ON  
      Product.name = Purchase.prodName
```

Same as:

```
SELECT Product.name, Purchase.store  
FROM Product, Purchase  
WHERE Product.name = Purchase.prodName
```

But Products that never sold will be lost !

Outerjoins

Left outer joins in SQL:
Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM Product LEFT OUTER JOIN Purchase ON
Product.name = Purchase.prodName
```

CS 178

53

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

CS 178

54

Outer Joins

- ❖ Left outer join:
 - Include the left tuple even if there's no match
- ❖ Right outer join:
 - Include the right tuple even if there's no match
- ❖ Full outer join:
 - Include the both left and right tuples even if there's no match

CS 178

55

Course Administrivia...

- ❖ HW 3 posted this week – SQL queries
 - Queries from HW2 plus a few more
- ❖ Team assignments will be posted soon.
 - Complete survey
- ❖ Phase 1 project will be posted next week
 - You must work alone during Phase 1

CS 178

56

Next: Advanced SQL

- ❖ We've done basic SQL
 - Returns data stored in database
 - Combination of Rel algebra and calculus
- ❖ More advanced SQL:
 - Aggregate and Group-by operations
 - Return result of applying some operations on the data
 - General constraints on schema
 - Constraints, CHECKs, Assertions, Triggers
 - Stored procedures

CS 178

57

Basic SQL Query

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
```

- ❖ relation-list A list of relation names (possibly with a *range-variable, i.e., tuple variable*, after each name).
- ❖ attribute-list A list of attributes of relations in *relation-list*
- ❖ Qualification/predicate Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of $<$, $>$, $=$, \leq , \geq , \neq) combined using AND, OR and NOT.
- ❖ **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!

CS 178

58

SQL- Aggregate Operations

- ❖ Thus far SQL (and Relational Algebra/Calculus) only fetched data stored in database tables
- ❖ What if we need some basic 'statistics' on the data?
 - Number of rows?
 - Maximum value in a field?
- ❖ Aggregate Operators: apply a function to a set of tuples
 - Function defined on one (or more) field

CS 178

59

Aggregate Operators

- ❖ Compute functions on set of tuples selected by where clause
- ❖ Semantics: if SELECT clause contains aggregate operations then it can contain *only* aggregate operations
 - **Except when groupby construct is used**
 - Functions on sets of values but result is single value
 - Average, minimum, maximum, sum, count(size)

CS 178

60

Aggregate Operators

- ❖ Significant extension of relational algebra.

COUNT (*)
 COUNT ([DISTINCT] A)
 SUM ([DISTINCT] A)
 AVG ([DISTINCT] A)
 MAX (A)
 MIN (A)

single column

```
SELECT AVG(balance)
FROM Deposit;
```

```
SELECT D.CustID
FROM Deposit D
WHERE D.balance = (SELECT MAX(S.balance)
FROM Deposit S)
```

```
SELECT AVG (D.balance)
FROM Deposit D
WHERE D.branchname='GW';
```

```
SELECT COUNT (DISTINCT CustID)
FROM Deposit
```

CS 178

61

More Examples

Purchase(product, date, price, quantity)

```
SELECT Sum(price * quantity)
FROM Purchase
```

What do they mean ?

```
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

CS 178

62

Simple Aggregations

Purchase

Product	Date	Price	Quantity
Bagel	10/21	1	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Bagel	10/25	1.50	20

```
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```



50 (= 20+30)

CS 178

63

Motivation for Grouping

- ❖ So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- ❖ Consider: Find the average balance for each branch in the bank.
 - In general, we don't know how many branches exist, and what the balances are!
 - Suppose we know that 10 branchnames exist; then we can write 10 queries that look like this (!):

For $x = 1, 2, \dots, 10$:

```
SELECT AVG(balance)
FROM Deposit D
WHERE D.branchname='x'
```

CS 178

64

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Let's see what this means...

CS 178

65

Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.
2. Group by the attributes in the **GROUPBY**
3. Compute the **SELECT** clause: grouped attributes and aggregates.

CS 178

66

1&2. FROM-WHERE-GROUPBY

Product	Date	Price	Quantity	
Bagel	10/21	1	20	Group 1
Bagel	10/25	1.50	20	
Banana	10/3	0.5	10	Group 2
Banana	10/10	1	10	

CS 178

67

3. SELECT

Product	Date	Price	Quantity	
Bagel	10/21	1	20	→
Bagel	10/25	1.50	20	
Banana	10/3	0.5	10	→
Banana	10/10	1	10	

Product	TotalSales
Bagel	50
Banana	15

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

CS 178

68

(a) Find average balance at each branch and (b) find number of customers with account at each branch

```
(a)SELECT branchname, avg(balance)
    FROM Deposit
    GROUPBY branchname;
(b) Query (a) keeps duplicates
    SELECT branchname, count(distinct CustID)
    FROM Deposit
    GROUPBY branchname;
```

Condition on the Groups

❖ What if we are only interested in groups that satisfy a condition ?

Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Let's see what this means...

Grouping and Aggregation

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

Product	TotalSales
Bagel	50
Banana	15

What if we are only interested in products that sold quantity >30?

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

HAVING Clause

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 buyers.

```
SELECT product, Sum(price * quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

```
SELECT S
FROM R1,...,Rn
WHERE C1
GROUP BY a1,...,ak
HAVING C2
```

Why?

S = may contain attributes a_1, \dots, a_k and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in R_1, \dots, R_n

C2 = is any condition on aggregate expressions

General form of Grouping and Aggregation

```
SELECT S
FROM R1,...,Rn
WHERE C1
GROUP BY a1,...,ak
HAVING C2
```

Evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes a_1, \dots, a_k
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Generalized SELECT: Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
GROUP BY grouping-list
HAVING group-qualification/predicate
```

❖ The *attribute-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).

- The attribute list must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation



- ❖ The cross-product of *relation-list* is computed, tuples that fail *qualification* in WHERE clause are discarded, 'unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- ❖ The *group-qualification* specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause must have a *single value per group!*
 - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- ❖ One answer tuple is generated per qualifying group.
- ❖ Any aggregate function can be applied to a group
 - Final SELECT can have function over each selected group

CS 178

77

Find branches where average account balance is greater than \$1200



```
SELECT branchname, avg(balance)
FROM Deposit
GROUPBY branchname
HAVING avg(balance) >1200
```

CS 178

78

Find those branches with the highest average balance



- ❖ Aggregate functions cannot be composed
 - Max(min(...)) not allowed
- ❖ Select groups that have average balance greater than or equal to ALL average balances at all branches.
 - Need nested query to compute set of balances at each branch

CS 178

79

```
SELECT branchname
FROM Deposit
GROUPBY branchname
HAVING AVG(balance) >= ALL
      (SELECT AVG(balance)
       FROM Deposit
       GROUPBY branchname);
```

CS 178

80

Find average balance of all depositors (account holders) who live in New York and have at least three accounts

- ❖ Select tuples where Customer lives in New York
- ❖ Form groups in Deposit based on Customer ID
 - GROUPBY CustID
- ❖ Select only those groups with 3 or more tuples
 - COUNT number of tuples in each group

```
SELECT AVG(balance)
FROM Deposit D, Customer C
WHERE D.CustID=C.CustID AND
      C.City='New York'
GROUPBY D.CustID
HAVING COUNT(distinct account-number)>=3
```

*A quick Note:
Group-by v.s. Nested Query*

Author(login,name)

Wrote(login,url)

- ❖ Find authors who wrote ≥ 10 documents:
- ❖ Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM Author
WHERE count(SELECT Wrote.url
            FROM Wrote
            WHERE Author.login=Wrote.login)
      > 10
```

This is
SQL by
a novice

Group-by v.s. Nested Query

- ❖ Find all authors who wrote at least 10 documents:
- ❖ Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) > 10
```

This is
SQL by
an expert

No need for **DISTINCT**: automatically from **GROUP BY**

CS 178

85

Quick Note: Outer Joins Application

Compute, for each product, the total number of sales in 'September'
Product(name, category)
Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
and Purchase.month = 'September'
GROUP BY Product.name
```

What's wrong ?

CS 178

86

Outer Join Application

Compute, for each product, the total number of sales in 'September'
Product(name, category)
Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM Product LEFT OUTER JOIN Purchase ON
Product.name = Purchase.prodName
and Purchase.month = 'September'
GROUP BY Product.name
```

Now we also get the products who sold in 0 quantity

CS 178


87

Modifying the Database

- ❖ INSERT
- ❖ UPDATE
- ❖ DELETE

CS 178

88




```
DELETE r
WHERE P
Predicate in P can be as complex as any select clause
Delete all accounts located in New York
DELETE Deposit
WHERE branchname in (SELECT branchname
                      FROM Branch
                      WHERE branchcity='New York');
```


How about this query ?

```
DELETE Deposit
WHERE balance < (SELECT avg(balance)
                 FROM Deposit);
```

Delete anomalies



- ❖ If delete/update request contains embedded select (sub-query) that references relation where deletions/update take place
- ❖ SQL standard disallows such requests
 - Alternate implementation: mark tuples in first round, and actual delete in second round

- 
- ❖ INSERT
 - Can insert tuple with specified values
 - Can insert set of tuples resulting from query
 - ❖ UPDATE
 - Change a value in tuple without changing all values in the tuple
 - Can update set of tuples by using query to select the set



```
INSERT INTO Deposit VALUES  
(9732, 'Downtown', 1234, 1000);
```

Also can specify attributes

```
INSERT INTO Deposit(Account-Num,  
CustID,Branch-name,Balance)  
VALUES (1234, 9732, 'Downtown', 1000);
```



INSERT

- ❖ Give all customers with a Loan at Downtown branch a \$200 savings account with same account number as Loan number

```
INSERT INTO Deposit  
SELECT CustID, Loan-number, Branch-name,  
200  
FROM Loan  
WHERE branch-name = 'Downtown';
```



UPDATE

- ❖ Update balances of all accounts over \$10,000 to give 5% interest

```
UPDATE Deposit  
SET balance = 1.05*balance  
WHERE balance > 10,000;
```



SQL: Views

- ❖ Create a virtual relation which is result of a query
 - View can be referenced by other queries
- ❖ **CREATE VIEW AS** <query-expression>
 - Query expression is any legal expression

Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee(ssn, name, department, project, salary)

```
CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"
```

Payroll has access to **Employee**, others only to **Developers**

Types of Views

❖ Virtual views:

- Used in databases
- Computed only on-demand – slow at runtime
- Always up to date

❖ Materialized views

- Used in data warehouses
- Pre-computed offline – fast at runtime
- May have stale data

Views: Example

- ❖ Create view of all branch names and IDs of customers with loan or deposit

```
CREATE VIEW all-customer AS
(SELECT branchname, CustID
FROM Deposit)
UNION
(SELECT branchname, CustID
FROM Loan);
```

Example

Person(name, city)

Purchase(buyer, seller, product, store)

Product(name, maker, price, category)

```
CREATE VIEW Seattle-Purchase AS

SELECT y.buyer, y.seller, y.product, y.store
FROM Person x, Purchase y
WHERE x.city = 'Seattle' AND
      x.name = y.buyer
```

Seattle-Purchase(buyer, seller, product, store) “virtual table”

Querying a View

We can later use the view:

```
SELECT v.name, u.store
FROM Seattle-Purchase u, Product v
WHERE u.product = v.name AND
      v.category = 'shoes'
```

CS 178

101

What Happens When We Query a View ?

```
SELECT v.name, u.store
FROM Seattle-Purchase u, Product v
WHERE u.product = v.name AND
      v.category = 'shoes'
```



```
SELECT v.name, y.store
FROM Person x, Purchase y, Product v
WHERE x.city = 'Seattle' AND
      x.name = y.buyer AND
      y.product = v.name AND
      v.category = 'shoes'
```

CS 178

102

Updating Views: Part 1

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

```
CREATE VIEW Expensive-Product AS
SELECT name, maker
FROM Product
WHERE price > 100
```

```
INSERT INTO Expensive-Product
VALUES('Gizmo', 'Gadgets INC.')
```



```
INSERT INTO Product
VALUES('Gizmo', 'Gadgets INC.', NULL, NULL)
```

CS 178

103

Updating Views: Part 2

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

```
CREATE VIEW Toy-Product AS
SELECT price, maker
FROM Product
WHERE category = 'Toys'
```

```
INSERT INTO Toy-Product
VALUES('Gadgets INC.', $100)
```



```
INSERT INTO Product
VALUES(NULL, 'Gadgets INC.', 100, NULL)
```

Anything wrong?

CS 178

104

Updating Views: Part 3

Purchase(buyer, seller, product, store)
Product(name, maker, price, category)

```
CREATE VIEW Buyer-Maker AS
SELECT x.buyer, y.maker
FROM Purchase x, Product y
WHERE x.product = y.name
```

Non-updateable
view

```
INSERT INTO Buyer-Maker
VALUES('John Smith', 'Gadgets INC.')
```

?????

Most views are
non-updateable

CS 178

105

Updating Views

- ❖ Non-updatable views
 - Multiple relation views
 - Primary key NULL
 - Later versions of SQL allow some updates on multiple relation views
- ❖ Interesting “side-effects” on view authorization
 - More when we discuss security/authorization

CS 178

106

Constraints in SQL

- ❖ A constraint = a property that we'd like our database to hold
- ❖ The system will enforce the constraint by taking some actions:
 - forbid an update
 - or perform compensating updates

CS 178

107

Constraints in SQL

Constraints in SQL:

- ❖ Keys, foreign keys
- ❖ Attribute-level constraints
- ❖ Tuple-level constraints
- ❖ Global constraints: assertions

simplest

Most
complex

The more complex the constraint, the harder it is to check and to enforce

CS 178

108

Constraints on Attributes and Tuples

- ❖ Constraints on attributes:
 - NOT NULL -- obvious meaning...
 - CHECK condition -- any condition !
- ❖ Constraints on tuples
 - CHECK condition

CS 178

109

Constraints: Examples

- ❖ Define a minimum balance in an account
- ```
CREATE TABLE Deposit(
 CustID integer
 Branch-name CHAR(20)
 Account-Num integer
 Balance Real
 PRIMARY KEY (CustID, Account-Num)
 FOREIGN KEY (CustID) REFERENCES Customer
 FOREIGN KEY (Branch-name) REFERENCES Branch
 CHECK (Balance >= 100));
```

CS 178

110

## Constraints over multiple tables

- ❖ Account-number cannot be same as zip code

```
CREATE TABLE Deposit(
 CustID integer
 Branch-name CHAR(20)
 Account-Num integer
 Balance Real
 PRIMARY KEY (CustID, Account-Num)
 FOREIGN KEY (CustID) REFERENCES Customer
 FOREIGN KEY (Branch-name) REFERENCES Branch
 CHECK (Balance >= 100)
 CHECK (Account-Num <>
 (SELECT zip
 FROM Customer));
```

CS 178

111

## General Constraints

- ❖ Useful when more general ICs than keys are involved.
- ❖ Can use queries to express constraint.
- ❖ Constraints can be named.

- What is the advantage of this?

```
CREATE TABLE Sailors
 (sid INTEGER,
 sname CHAR(10),
 rating INTEGER,
 age REAL,
 PRIMARY KEY (sid),
 CHECK (rating >= 1
 AND rating <= 10)
);

CREATE TABLE Reserves
 (sname CHAR(10),
 bid INTEGER,
 day DATE,
 PRIMARY KEY (bid,day),
 CONSTRAINT noInterlakeRes
 CHECK ('Interlake' <>
 (SELECT B.bname
 FROM Boats B
 WHERE B.bid=bid)))
```

CS 178

112

## Domain Constraints – user defined types

- ❖ User can define a new domain
  - Restriction of domain types supported by DBMS
  - Example: no grade below C

```
CREATE DOMAIN nice-grade INTEGER DEFAULT 1
CHECK (VALUE >1 AND VALUE <=4)
```

- ❖ In schema definition, attribute grade is defined to be of this type
  - ... grade nice-grade
- ❖ Remember domain constraints
  - These are very useful when designing real applications!!!

What is the difference from Foreign-Key ?

```
CREATE TABLE Purchase (
 prodName CHAR(30)
 CHECK (prodName IN
 SELECT Product.name
 FROM Product),
 date DATETIME NOT NULL)
```

## Constraints Over Multiple Relations

```
CREATE TABLE Sailors
 (sid INTEGER,
 sname CHAR(10),
 rating INTEGER,
 age REAL,
 PRIMARY KEY (sid),
 CHECK
 ((SELECT COUNT (S.sid) FROM Sailors S)
 + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

Number of boats plus number of sailors is < 100

- ❖ Awkward and wrong!
- ❖ If Sailors is empty, the number of Boats tuples can be anything!

## Assertions: Constraints Over Multiple Relations

```
CREATE TABLE Sailors
 (sid INTEGER,
 sname CHAR(10),
 rating INTEGER,
 age REAL,
 PRIMARY KEY (sid),
 CHECK
 ((SELECT COUNT (S.sid) FROM Sailors S)
 + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

Number of boats plus number of sailors is < 100

- ❖ Awkward and wrong!
- ❖ If Sailors is empty, the number of Boats tuples can be anything!
- ❖ **ASSERTION** is the right solution; not associated with either table.

```
CREATE ASSERTION smallClub
CHECK
 ((SELECT COUNT (S.sid) FROM Sailors S)
 + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

## Assertions



- ❖ Assertion is a predicate expressing a condition that the database must satisfy
- ❖ Assertion must check to make sure an update is good
  - on violation it rejects the update
- ❖ Allows constraints over multiple tables

## Final Comments on Constraints



- ❖ Can give them names, and alter later
- ❖ We need to understand exactly *when* they are checked
- ❖ We need to understand exactly *what* actions are taken if they fail

## Triggers




- ❖ Trigger: procedure that starts automatically if specified changes occur to the DBMS
- ❖ Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

## Triggers: Example




- ❖ Overdraft transaction:
  - When balance is negative, create a Loan in the amount of the overdraft plus a fee
    - Loan number is same as account-number
- ❖ When to “act” ?
  - On update of Deposit relation
- ❖ What is the action ?
  - Add new tuple into Loan




```
DEFINE TRIGGER Overdraft
ON UPDATE OF Deposit T
If new.balance < 0
Then (
 INSERT INTO Loan values
 (T.Branch-name, T.Cust-ID, T.account-num,
 new.balance+fine)
 UPDATE Deposit S
 SET S.balance=0
 WHERE T.account-num = S.account-num)
```

## Triggers: Example (SQL:1999)




```
CREATE TRIGGER youngSailorUpdate
 AFTER INSERT ON SAILORS
 REFERENCING NEW TABLE NewSailors
 FOR EACH STATEMENT
 INSERT
 INTO YoungSailors(sid, name, age, rating)
 SELECT sid, name, age, rating
 FROM NewSailors N
 WHERE N.age <= 18
```

## Summary



- ❖ SQL SELECT statements
  - Nested queries
  - Set operations
  - Aggregate operations
  - Grouping
- ❖ Constraints, Triggers, Assertions ?
  - Will get to this after a week
- ❖ Next question: How to design a schema ?

## Relational Model: Definitions Review



- ❖ Relations/tables, Attributes/Columns, Tuples/rows
  - Attribute domains
- ❖ Superkey
- ❖ Key
  - No two tuples can have the same value in the key attribute
  - Primary key, candidate keys
  - No primary key value can be null
- ❖ Referential integrity constraints
  - Foreign key

## Relational Schema Design

- ❖ Logical Level
  - Whether schema has intuitive appeal for users
- ❖ Manipulation level
  - Whether it makes sense from an *efficiency* or *correctness* point of view

CS 178

125

## Functional Dependencies and Normal Forms

- ❖ Guidelines for database schema design: how to design a “good” schema ?
- ❖ Example of a COMPANY database: two possible designs to represent Employees and Department information

S1: EMPLOYEE(LNAME,FNAME,SSN,DNO)  
DEPT(DNUM, DNAME, MGRSSN)

S2:  
EMPDEPT(LNAME,FNAME,SSN,DNUM,DNAME,  
MGRSSN)

Which one is better ?? S1 or S2 ?

CS 178

126

## Functional Dependencies and Normal Forms

- ❖ Informal methods
  - Rules of thumb, intuitive reasoning, experience
- ❖ Formal methods
  - Provable properties
  - Involve concept of **Functional Dependencies**
  - Develop theoretical model to define what we mean by “good schema”

CS 178

127

## Informal Guidelines: 1

- ❖ 1: Try to make user interpretation easy

S1: EMP(FNAME, LNAME, SSN)  
WORKS\_ON (SSN, PNO)  
PROJECT\_LOC(PNO, PLOC)

S2: EMP(FNAME,LNAME,SSN, PNO,PLOC)

- ❖ Perhaps S2 has too much information to absorb per tuple ?

CS 178

128

## Informal Guidelines: 2

- ❖ Try to reduce redundancy
  - In S2 in previous example, suppose only few projects
    - PLOC is unnecessarily repeated too often
  - On the other hand, S1 repeats SSN in WORKS\_ON
    - But SSN is a smaller attribute than PLOC (which may be a large string)

CS 178

129

## Informal Guidelines: 3

- ❖ Try to avoid update anomalies
  - Avoid having to search through entire table during update operation
    - Insert, delete, update/modify
  - Avoid losing information

CS 178

130

## Example

S1: EMPLOYEE (ENAME, SSN, BDATE, ADDR, DNO)  
DEPT (DNUM, MGRSSN, DNAME)  
WORKS\_ON (SSN, PNO, HOURS)  
PROJECT (PNUM, PLOC)

S2: EMP\_DEPT(ENAME,SSN,BDATE, ADDR,DNO, DNAME,MGRSSN)  
EMP\_PROJ (SSN, PNUM, HOURS, ENAME, PNAME, PLOC)

Both schemas have same attributes....  
Problems with S2 ??

CS 178

131

## Insertion Anomalies in S2

- ❖ Consider inserting information "John Smith works in Department 5"
  - i.e., < John Smith, 123456789,...,5,Research,111223333>
  - What do we need to check in the table to maintain correctness ?
- ❖ Check DNAME is correct - i.e., all tuples with DNO=5 have Research and 111223333
  - Scan the whole relation since DNO is not a key!

CS 178

132

## *Insertion problems...contd.*



- ❖ Consider creating a new department in the Company: DNO=9, DNAME = 'Sales'
- ❖ Only one way to do this: create NULL values for employee info
  - But that means NULL value in primary key (SSN)!!

## *Deletion and Modification Anomalies*



- ❖ What if we delete the last employee in 'Research' department
  - Eg. (John Smith, 123456789,...,5,'Research',...)
- ❖ We lose the information that Department 5 is Research department
  - Problem: We can then insert a tuple with DNO=5 and DNAME= Sales!!!
- ❖ Similar cases for Modification
  - Change Manager SSN of department 5 = change for all Department 5 employees ...scan of entire table

## *Informal Guidelines*



- ❖ Avoid too many NULL values
  - Space is wasterd
  - Problems occur when using aggregate functions like count or sum
  - NULLs can have different intentions
    - Attribute does not apply
    - Value unknown and will remain unknown
    - Value unknown at present

## *Informal Guidelines: Spurious Tuples*



- ❖ Split a table into smaller tables (with fewer columns in each) - sometimes a better design in our examples
  - How to split ?
- ❖ When reconstructing the "original" data, should not introduce spurious tuples

## Example: Spurious Tuples

S1: CAR (ID, Make, Color)

|     |        |      |
|-----|--------|------|
| 123 | Toyota | Blue |
| 456 | Audi   | Blue |
| 789 | Toyota | Red  |

S2: CAR1 (ID, Color)

CAR2 (Color, Make)

What happens when we  
join CAR1 and CAR2 ?

|     |      |      |        |
|-----|------|------|--------|
| 123 | Blue | Blue | Toyota |
| 456 | Blue | Blue | Audi   |
| 789 | Red  | Red  | Toyota |

|     |      |        |
|-----|------|--------|
| 123 | Blue | Toyota |
| 123 | Blue | Audi   |
| 456 | Blue | Toyota |
| 456 | Blue | Audi   |
| 789 | Red  | Toyota |

## Summary of Problems

- ❖ Insertion, Deletion, modification anomalies
- ❖ Too many NULLs
- ❖ Spurious tuples - called non-additive join
- ❖ We need a theory of schema design
  - Functional dependencies and normalization
- ❖ Using functional dependencies define "normal forms" of schema
  - A schema in a "Third Normal Form" will avoid certain anomalies

*Next...*



❖ Functional Dependencies and Normal Forms