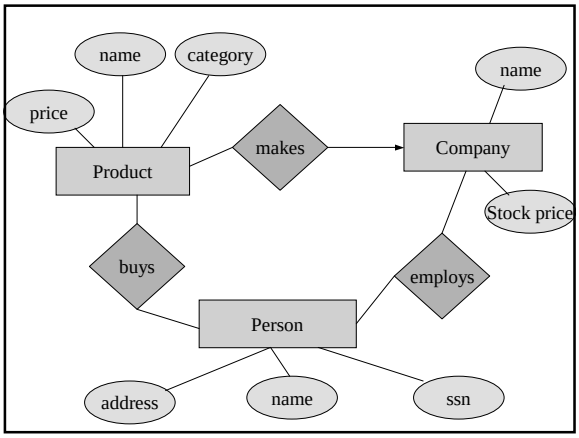
Building a Database Application

- Conceptual design (mostly pictures, a lot of hand waving).
- Design a schema in some data model:
  - relational
  - object-oriented, object relational
  - XML (semi-structured)
- Build the application. Populate the DB.



Outline

- The basics
- Specifying more semantic information: integrity constraints
- The relational algebra: operations on relations.
- A query language based on relational algebra (and more): SQL.

The Relational Model (Codd)

Attribute names

Product

Name	Price	Category	Manufacturer
gizmo	$19.99	gadgets	GizmoWorks
Power gizmo	$29.99	gadgets	GizmoWorks
SingleTouch	$149.99	photography	Canon
MultiTouch	$203.99	household	Hitachi

tuples

(Arity=4)

Product(name: string, Price: real, category: enum, Manufacturer: string)

More Terminology

Every attribute has an atomic type (hmm…)

Relation Schema: relation name + attribute names + attribute types

Relation instance: a set of tuples. Only one copy of any tuple! **(not)**

Database Schema: a set of relation schemas.

Database instance: a relation instance for every relation in the schema.

What can't we say in the relational model?

More on Tuples

Formally, a mapping from attribute names to (correctly typed) values:

name → gizmo
price → $19.99
category → gadgets
manufacturer → GizmoWorks

Sometimes we refer to a tuple by itself: (note order of attributes)

(gizmo, $19.99, gadgets, GizmoWorks) or

Product (gizmo, $19.99, gadgets, GizmoWorks).

Integrity Constraints

An important functionality of a DBMS is to enable the specification of integrity constraints and to enforce them.

Knowledge of integrity constraints is also useful for query optimization.

Examples of constraints:

keys, superkeys
foreign keys
domain constraints, tuple constraints.
Functional dependencies, multi-valued dependencies.

Keys

A minimal set of attributes that uniquely the tuple (I.e., there is no pair of tuples with the same values for the key attributes):

Person: social security number
name
name + address
name + address + age

Perfect keys are often hard to find, but organizations usually invent something anyway.
Superkey: a set of attributes that contains a key.
A relation may have multiple keys: (but only one primary key)

employee number, social-security number

Foreign Key Constraints

Purchase:

buyer	price	product
Joe	$20	gizmo
Jack	$20	E-gizmo

Product:

name	manufacturer	description
gizmo	G-sym	great stuff
E-gizmo	G-sym	even better

An attribute of a relation R is must refer to a key of a relation S.

Functional Dependencies

Definition:

If two tuples agree on the attributes

$A_1, A_2, \ldots A_n$

then they must also agree on the attributes

$B_1, B_2, \ldots B_m$

Formally: $A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$

Key of a relation: all the attributes are either on the left or right.

Some Obvious Properties of FD's

$A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$ Is equivalent to

$A_1, A_2, \ldots A_n \longrightarrow B_1$
$A_1, A_2, \ldots A_n \longrightarrow B_2$
…
$A_1, A_2, \ldots A_n \longrightarrow B_m$

Splitting rule and Combing rule

$A_1, A_2, \ldots A_n \longrightarrow A_i$ Always holds.

Comparing Functional Dependencies

Entailment: a set of functional dependencies S1 entails a set S2 if:

any database that satisfies S1 much also satisfy S2.

Example: $A \rightarrow B$, $B \rightarrow C$ entails $A \rightarrow C$

Equivalence: two sets of FD's are equivalent if each entails the other

$\{A \rightarrow B, B \rightarrow C\}$ is equivalent to $\{A \rightarrow B, A \rightarrow C, B \rightarrow C\}$

Closure: Given a set of attributes ***A*** and a set of dependencies C, we want to find all the other attributes that are functionally determined by ***A***.

Closure Algorithm

Start with Closure=A.

Until closure doesn't change do:

if $A_1, A_2, \ldots A_n \longrightarrow$ B is in C, **and**

$A_1, A_2, \ldots A_n$ Are all in the closure, **and**
B is not in Closure

then

add B to closure.

Problems in Designing Schema

Name	SSN	Phone Number
Fred	123-321-99	(201) 555-1234
Fred	123-321-99	(206) 572-4312
Joe	909-438-44	(908) 464-0028
Joe	909-438-44	(212) 555-4000

Problems:

- redundancy
- update anomalies
- deletion anomalies

Relation Decomposition

Break the relation into two relations:

Name	SSN
Fred	123-321-99
Joe	909-438-44

Name	Phone Number
Fred	(201) 555-1234
Fred	(206) 572-4312
Joe	(908) 464-0028
Joe	(212) 555-4000

Boyce-Codd Normal Form

A simple condition for removing anomalies from relations:

A relation R is in BCNF if and only if:

Whenever there is a nontrivial dependency $A_1, A_2, \ldots A_n \longrightarrow$ B for R , it is the case that $\{ A_1, A_2, \ldots A_n \}$ is a super-key for R.

In English (though a bit vague):

Whenever a set of attributes of *R* is determining another attribute, should determine ***all*** the attributes of *R*.

Relational Algebra

- Operators: sets as input, new set as output
- Basic Set Operators
  - union, intersection, difference, but no complement. (watch comparable sets)
- Selection
- Projection
- Division(not in text)
- Cartesian Product
- Joins, combination of cart product/selection

Set Operations

- Binary operations
  - Result is table(set) with same attributes
- Sets must be compatible!
  - R1(A1,A2,A3)&R2(B1,B2,B3)
  - Domain(Ai)=Domain(Bi)
- Union: all tuples in R1 or R2
- Intersection: all tuples in R1 and R2
- Difference: all tuples in R1 and not in R2
- No complement… what's the universe?

Selection Example

Employee

SSN	Name	DepartmentID	Salary
999999999	John	1	30,000
777777777	Tony	1	32,000
888888888	Alice	2	45,000

SSN	Name	DepartmentID	Salary
888888888	Alice	2	45,000

Projection Example

Employee

SSN	Name	DepartmentID	Salary
999999999	John	1	30,000
777777777	Tony	1	32,000
888888888	Alice	2	45,000

SSN	Name
999999999	John
777777777	Tony
888888888	Alice

Cartesion Product Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

EmployeeSSN	Dname
999999999	Emily
777777777	Joe

Employee_Dependents

Name	SSN	EmployeeSSN	Dname
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Join

- Most often used…
- Combines two relations, selecting only related tuples
- Equivalent to a cross product followed by selection
- Resulting schema has all attributes of the two relations, but one copy of join condition attributes
- Example

Join Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

EmployeeSSN	Dname
999999999	Emily
777777777	Joe

Employee_Dependents

Name	SSN	Dname
John	999999999	Emily
Tony	777777777	Joe

SQL Introduction

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many standards out there: SQL92, SQL2, SQL3, SQL99
Vendors support various subsets of these, but all of what we'll be talking about.

Basic form: (many many more bells and whistles in addition)

Select attributes
From relations (possibly multiple, joined)
Where conditions (selections)

SQL Outline

- select-project-join
- attribute referencing, select distinct
- nested queries
- grouping and aggregation
- updates
- laundry list

Selections

```
SELECT   *
FROM     Company
WHERE    country="USA" AND stockPrice > 50
```

You can use:

attribute names of the relation(s) used in the FROM.
comparison operators: =, <>, <, >, <=, >=
apply arithmetic operations: stockprice*2
operations on strings (e.g., "||" for concatenation).
Lexicographic order on strings.
Pattern matching: s LIKE p
Special stuff for comparing dates and times.

Projections and Ordering Results

Select only a subset of the attributes

```
SELECT   name, stock price
FROM     Company
WHERE    country="USA" AND stockPrice > 50
```

Rename the attributes in the resulting table

```
SELECT   name AS company, stockprice AS price
FROM     Company
WHERE    country="USA" AND stockPrice > 50
ORDERBY  country, name
```

Joins

```
SELECT   name, store
FROM     Person, Purchase
WHERE    name=buyer
           AND city="Seattle"
           AND product="gizmo"
```

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Company (name, stock price, country)
Person(name, phone number, city)

Disambiguating Attributes

Find names of people buying telephony products:

```
SELECT   Person.name
         FROM     Person, Purchase, Product
         WHERE        Person.name=buyer
                  AND  product=Product.name
                  AND  Product.category="telephony"
```

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Person(name, phone number, city)

Tuple Variables

Find pairs of companies making products in the same category

```
SELECT  product1.maker, product2.maker
        FROM      Product AS product1, Product AS product2
        WHERE         product1.category=product2.category
                AND  product1.maker <> product2.maker
```

Product (name, price, category, maker)

First Unintuitive SQLism

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

Looking for $R \cap (S \cup T)$

But what happens if T is empty?

Union, Intersection, Difference

```
(SELECT  name
 FROM      Person
 WHERE    City="Seattle")

    UNION

(SELECT  name
 FROM      Person, Purchase
 WHERE   buyer=name AND store="The Bon")
```

Similarly, you can use INTERSECT and EXCEPT.
You must have the same attribute names (otherwise: rename).

Subqueries

```
SELECT Purchase.product
FROM     Purchase
WHERE  buyer =
            (SELECT  name
              FROM     Person
              WHERE   social-security-number = "123 - 45 - 6789");
```

In this case, the subquery returns one value.

If it returns more, it's a run-time error.

Subqueries Returning Relations

Find companies who manufacture products bought by Joe Blow.

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

You can also use: s > ALL R
s > ANY R
EXISTS R

Correlated Queries

Find movies whose title appears more than once.

```
SELECT  title
FROM   Movie AS Old
WHERE  year < ANY
                (SELECT  year
                  FROM    Movie
                  WHERE  title =  Old.title);
```

Movie (title, year, director, length)
Movie titles are not unique (titles may reappear in a later year).

Note scope of variables

Removing Duplicates

```
SELECT  DISTINCT Company.name
FROM      Company, Product
WHERE   Company.name=maker
          AND  (Product.name,price)  IN
                  (SELECT product, price)
                    FROM   Purchase
                    WHERE  buyer = "Joe Blow");
```

Conserving Duplicates

The UNION, INTERSECTION and EXCEPT operators operate as sets, not bags.

```
(SELECT  name
 FROM    Person
 WHERE   City="Seattle")

   UNION  ALL

(SELECT  name
 FROM    Person, Purchase
 WHERE   buyer=name AND store="The Bon")
```

Grouping and Aggregation

Product	Date	Price	Quantity
Bagel	10/21	0.85	15
Banana	10/22	0.52	7
Banana	10/19	0.52	17
Bagel	10/20	0.85	20

Example 1: **find total sales for the entire database**

Simple Aggregation

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

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

SQL supports several aggregation operations:

SUM, MIN, MAX, AVG, COUNT

Except COUNT, all aggregations apply to a single attribute

Grouping and Aggregation

Product	Date	Price	Quantity
Bagel	10/21	0.85	15
Banana	10/22	0.52	7
Banana	10/19	0.52	17
Bagel	10/20	0.85	20

Example 2: **find total sales per product.**

Solution: Two Steps

First: group the entries by product.

Product	Date	Price	Quantity
Banana	10/19	0.52	17
Banana	10/22	0.52	7
Bagel	10/20	0.85	20
Bagel	10/21	0.85	15

Example 2: **find total sales per product.**

Then, aggregate

Product	TotalSales
Bagel	$29.75
Banana	$12.48

```
SELECT    product, Sum(price * quantity) AS TotalSales
FROM      Purchase
GROUP BY product
```

Another Example

Product	SumSales	MaxQuantity
Banana	$12.48	17
Bagel	$29.75	20

For every product, what is the total sales and max quantity sold?

```
SELECT    product, Sum(price * quantity) AS SumSales
                   Max(quantity) AS MaxQuantity
FROM      Purchase
GROUP BY product
```

Grouping and Aggregation: Summary

```
SELECT    product, Sum(price)
FROM      Product,  Purchase
WHERE     Product.name = Purchase.product
GROUP BY Product.name
```

1. Compute the relation (I.e., the FROM and WHERE).
2. Group by the attributes in the GROUP BY
3. Select one tuple for every group (and apply aggregation)

SELECT can have (1) grouped attributes or (2) aggregates.

HAVING Clause

Same query, except that we consider only products that had at least 100 buyers.

```
SELECT    product, Sum(price * quantity)
FROM      Purchase
GROUP BY product
HAVING    Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

Modifying the Database

We have 3 kinds of modifications: insertion, deletion, update.

Insertion: general form --

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

Insert a new purchase to the database:

```
INSERT  INTO  Purchase(buyer, seller, product, store)
              VALUES  (Joe, Fred, wakeup-clock-espresso-machine,
                       "The Sharper Image")
```

If we don't provide all the attributes of R, they will be filled with NULL.
We can drop the attribute names if we're providing all of them in order.

Data Definition in SQL

So far, SQL operations on the data.

Data definition: defining the schema.

- Create tables
- Delete tables
- Modify table schema

But first:
Define data types.

Finally: define indexes.

Data Types in SQL

- Character strings (fixed of varying length)
- Bit strings (fixed or varying length)
- Integer (SHORTINT)
- Floating point
- Dates and times

Domains will be used in table declarations.

To reuse domains:

```
CREATE DOMAIN  address  AS  VARCHAR(55)
```

Creating Tables

```
CREATE    TABLE Person(

        name                    VARCHAR(30),
        social-security-number  INTEGER,
        age                     SHORTINT,
        city                    VARCHAR(30),
        gender                  BIT(1),
        Birthdate               DATE

 );
```

Creating Indexes

```
CREATE INDEX  ssnIndex ON Person(social-security-number)
```

Indexes can be created on more than one attribute:

```
CREATE INDEX doubleindex ON
                Person (name, social-security-number)
```

Why not create indexes on everything?

Defining Views

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

They are used mostly in order to simplify complex queries and to define conceptually different views of the database to different classes of users.

View: purchases of telephony products:

```
CREATE VIEW  telephony-purchases AS
  SELECT product, buyer, seller, store
  FROM  Purchase, Product
  WHERE  Purchase.product = Product.name
         AND  Product.category = "telephony"
```

A Different View

```
CREATE VIEW  Seattle-view  AS

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

We can later use the views:

```
    SELECT   name, store
    FROM     Seattle-view, Product
    WHERE    Seattle-view.product = Product.name  AND
             Product.category = "shoes"
```

What's really happening when we query a view??

Updating Views

How can I insert a tuple into a table that doesn't exist?

```
CREATE VIEW  bon-purchase  AS
   SELECT   store, seller, product
   FROM     Purchase
   WHERE    store = "The Bon Marche"
```

If we make the following insertion:

```
INSERT INTO  bon-purchase
       VALUES  ("the Bon Marche", Joe, "Denby Mug")
```

We can simply add a tuple
("the Bon Marche", Joe, NULL, "Denby Mug")
to relation Purchase.

Non-Updatable Views

```
CREATE VIEW  Seattle-view  AS

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

How can we add the following tuple to the view?

(Joe, "Shoe Model 12345", "Nine West")

Reusing a Materialized View

- Suppose I have **only** the result of SeattleView:

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

- and I want to answer the query

```
SELECT  buyer, seller
      FROM     Person, Purchase
      WHERE   Person.city = 'Seattle'    AND
                  Person.per-name = Purchase.buyer AND
                  Purchase.product='gizmo'.
```

Then, I can rewrite the query using the view.

Query Rewriting Using Views

Rewritten query:

```
SELECT  buyer, seller
    FROM     SeattleView
    WHERE   product= 'gizmo'
```

Original query:

```
SELECT  buyer, seller
      FROM     Person, Purchase
      WHERE   Person.city = 'Seattle'    AND
                  Person.per-name = Purchase.buyer AND
                  Purchase.product='gizmo'.
```

Another Example

- I still have **only** the result of SeattleView:

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

- but I want to answer the query

```
SELECT  buyer, seller
      FROM     Person, Purchase
      WHERE   Person.city = 'Seattle'    AND
                  Person.per-name = Purchase.buyer AND
                  Person.Phone  LIKE '206 543 %'.
```