## Data and Queries in the Relational Model

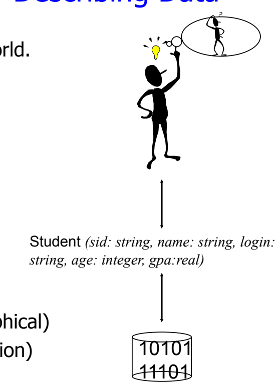
CS 162 Guest Lecture
Mike Franklin
April 6, 2011

A relationship, I think, is like a shark, you know? It has to constantly move forward or it dies. And I think what we got on our hands is a dead shark.

Woody Allen (from Annie Hall, 1979)

## Data Models – Describing Data

- A *Database design* encodes some portion of the real world.
- A *Data Model* is a set of concepts for thinking about this encoding.
- Many models have been proposed.
- We will look at two related models:
  - i) Entity-Relationship (graphical)
  - ii) Relational (implementation)

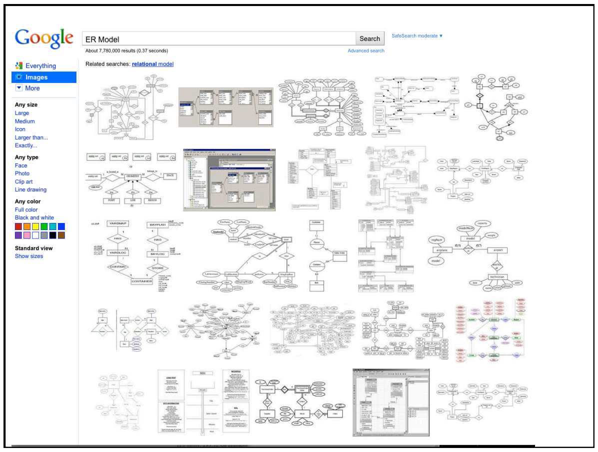



## Steps in Database Design

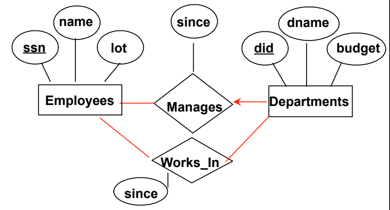
- Requirements Analysis
  - user needs; what must the database capture?
- Conceptual Design
  - high level description (often done w/ER model)
- Logical Design
  - translate ER into DBMS data model
    - Typically: "relational" model as implemented by SQL
- Schema Refinement - consistency, normalization
- Physical Design - indexes, disk layout
- Security Design - who accesses what, and how

## Conceptual Design using ER

- What are the entities and relationships?
- What info about E's & R's should be in DB?
- What *integrity constraints* (*business rules*) hold?
- *ER diagram* is a representation of the `schema'
- Can map an ER diagram into a relational schema.
- Conceptual design is where the SW/data engineering *begins*
  - Rails "models"

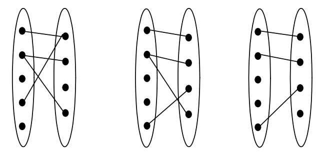



# ER Example




An employee can work in many departments; a dept can have many employees.

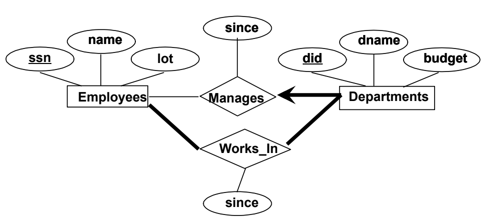
In contrast, each dept has at most one manager, according to the *key constraint* on Manages.



**Many-to-Many** **1-to-Many** **Many-to-1** **1-to-1**

# Participation Constraints

- Does every employee work in a department?
- If so: a *participation constraint*
  - participation of Employees in Works_In is *total* (vs. *partial*)
  - What if every department has an employee working in it?
- Basically means "at least one"




# Implementation: The Relational Model

- The E-R model is not directly implemented by most DBMSs.
- Fairly easy to map an E-R design to a Relational Schema
- The Relational Model is Ubiquitous
  - MySQL, PostgreSQL, Oracle, DB2, SQLServer, …
  - Note: some "Legacy systems" use older models
    - e.g., IBM's IMS
- Object-oriented concepts have been merged in
  - Early work: POSTGRES research project at Berkeley
  - Informix, IBM DB2, Oracle 8i
- As has support for XML (semi-structured data)

## Relational Database: Definitions

- *Relational database:* a set of *relations*
- *Relation:* made up of 2 parts:
  *Schema* : specifies name of relation, plus name and type of each column

  **Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real)**

  *Instance* : the actual data at a given time
  - #rows = *cardinality*
  - #fields = *degree / arity*

## Some Synonyms

| Formal | Not-so-formal 1 | Not-so-formal 2 |
|---|---|---|
| Relation | Table | |
| Tuple | Row | Record |
| Attribute | Column | Field |
| Domain | Type | |

## Ex: Instance of Students Relation

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

- Cardinality = 3, arity = 5 , all rows distinct
- Do all values in each column of a relation instance have to be distinct?

## SQL - A language for Relational DBs

- Say: "ess-cue-ell" or "sequel"
  - But spelled "SQL"
- Data Definition Language (DDL)
  - create, modify, delete relations
  - specify constraints
  - administer users, security, etc.
- Data Manipulation Language (DML)
  - Specify queries to find tuples that satisfy criteria
  - add, modify, remove tuples

## Creating Relations in SQL

- Create the Students relation:

```
CREATE TABLE Students
    (sid CHAR(20),
     name CHAR(20),
     login CHAR(10),
     age INTEGER,
     gpa FLOAT)
```

## Table Creation (continued)

- Another example: the Enrolled table holds information about courses students take.

```
CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2))
```

## Constraints - Keys

- Keys are a way to associate tuples in different relations
- Keys are one form of integrity constraint (IC)

Enrolled

| sid | cid | grade |
|---|---|---|
| 53666 | Carnatic101 | C |
| 53666 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

Students

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

FOREIGN Key

PRIMARY Key

## Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.
- Keys must be used carefully!
- "For a given student and course, there is a single grade."

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))
```

vs.

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade))
```

"Students can take only one course, and no two students in a course receive the same grade."

## Foreign Keys, Referential Integrity

- *Foreign key*: a "logical pointer"
  - Set of fields in a tuple in one relation that `refer' to a tuple in another relation.
  - Reference to *primary key* of the other relation.
- All foreign key constraints enforced?
  - *referential integrity*!
  - i.e., no dangling references.

## Foreign Keys in SQL

- **E.g. Only students listed in the Students relation should be allowed to enroll for courses.**
  - *sid* is a foreign key referring to Students:

```
CREATE TABLE Enrolled
(sid CHAR(20),cid CHAR(20),grade CHAR(2),
 PRIMARY KEY (sid,cid),
 FOREIGN KEY (sid) REFERENCES Students);
```

Enrolled

| sid | cid | grade |
|---|---|---|
| 53666 | Carnatic101 | C |
| 53666 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |
| ~~11111~~ | ~~English102~~ | ~~A~~ |

Students

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

## Enforcing Referential Integrity

- *sid* in Enrolled: foreign key referencing Students.
- Scenarios:
  - Insert Enrolled tuple with non-existent student id?
  - Delete a Students tuple?
    - Also delete Enrolled tuples that refer to it? (Cascade)
    - Disallow if referred to? (No Action)
    - Set sid in referring Enrolled tuples to a *default* value? (Set Default)
    - Set sid in referring Enrolled tuples to *null*, denoting `*unknown*' or `*inapplicable*'. (Set NULL)
- Similar issues arise if primary key of Students tuple is updated.

## Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database
  - e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

## Where do ICs Come From?

- Semantics of the real world!
  - Should be determined during Requirements Analysis and/or Conceptual Design phases
- Note:
  - We can check IC violation in a DB instance
  - We can NEVER infer that an IC is true by looking at an instance.
    - An IC is a statement about all possible instances!
  - From example, we know name is not a key, but the assertion that sid is a key is given to us.
- Key and foreign key ICs are the most common
- More general ICs supported too.

## Adding and Deleting Tuples

- Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
 VALUES ('53688', 'Smith', 'smith@ee', 18, 3.2)
```

- **Can delete all tuples satisfying some condition (e.g., name = Smith):**

```
DELETE
  FROM Students S
 WHERE S.name = 'Smith'
```

**Powerful variants of these commands are available;**

## Relational Query Languages

- Feature: Simple, powerful *ad hoc querying*
- Declarative languages
  - Queries precisely specify *what* to return
  - DBMS is responsible for efficient evaluation (*how*).
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.
    - Key to data independence!

## The SQL Query Language

- The most widely used relational query language.
  - Current std is SQL:2008; SQL92 is a basic subset
- To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
```

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

- To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```

## Querying Multiple Relations

- What does the following query compute?

```
SELECT S.name, E.cid
  FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='A'
```

Given the following instances

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

we get:

| S.name | E.cid |
|---|---|
| Smith | Topology112 |

## Cross-product of Students and Enrolled Instances

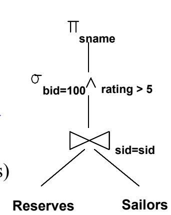
| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |
| 53650 | Smith | smith@math | 19 | 3.8 | 53831 | Carnatic101 | C |
| 53650 | Smith | smith@math | 19 | 3.8 | 53831 | Reggae203 | B |
| **53650** | Smith | smith@math | 19 | 3.8 | **53650** | Topology112 | **A** |
| 53650 | Smith | smith@math | 19 | 3.8 | 53666 | History105 | B |

## Query Optimization Overview

- Query can be converted to relational algebra
- Rel. Algebra converted to tree, joins as branches
- Each operator has implementation choices
- Operators can also be applied in different order!

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
  R.bid=100 AND S.rating>5
```

$\pi_{(sname)}\sigma_{(bid=100 \wedge rating>5)}$ (Reserves ⋈ Sailors)

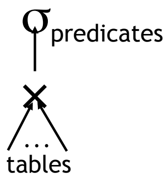



## Relational Operations

- Some "logical" operators:
  - *Selection* ( σ ) Selects a subset of rows from relation.
  - *Projection* ( π ) Deletes unwanted columns from relation.
  - *Join* (⋈) Allows us to combine two relations.
  - *Set-difference* ( - ) Tuples in reln. 1, but not in reln. 2.
  - *Union* ( ∪ ) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

## A Really Simple Query Optimizer

- For each Select-From-Where query block
  - Create a plan that:
    - Forms the cartesian product of the FROM clause
    - Applies the WHERE clause
    - Incredibly inefficient
      - Huge intermediate results!

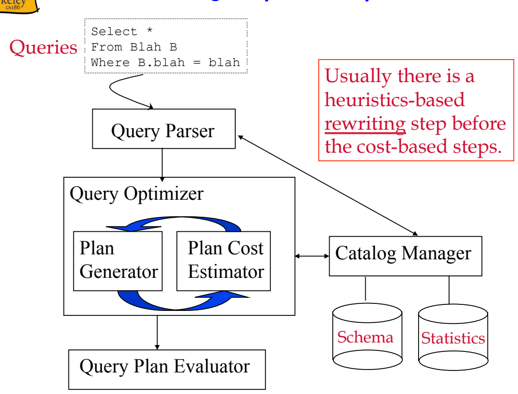



- Then, as needed:
  - Apply the GROUP BY clause
  - Apply the HAVING clause
  - Apply any projections and output expressions
  - Apply duplicate elimination and/or ORDER BY

## The Query Optimization Game

- "Optimizer" is a bit of a misnomer…
- Goal is to pick a "good" (i.e., low expected cost) plan.
  - Involves choosing access methods, physical operators, operator orders, …
  - Notion of cost is based on an abstract "cost model"

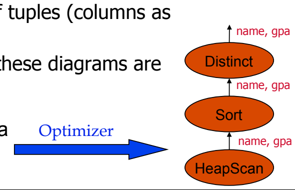
## Cost-based Query Sub-System



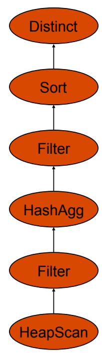

## Query Processing Overview

- The *query optimizer* translates SQL to a special internal "language"
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as "box-and-arrow" *dataflow* diagrams
  - Each box implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

SELECT DISTINCT name, gpa
FROM Students



## Iterators




- The relational operators are all subclasses of the class `iterator`:

```
class iterator {
   void init();
   tuple next();
   void close();
   iterator inputs[];
   // additional state goes here
}
```

- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

## Example: Scan

```
class Scan extends iterator {
   void init();
   tuple next();
   void close();
   iterator inputs[1];
   bool_expr filter_expr;
   proj_attr_list proj_list;
}
```

- `init()`:
  - Set up internal state
  - call init() on child – often a file open
- `next()`:
  - call next() on child until qualifying tuple found or EOF
  - keep only those fields in "proj_list"
  - return tuple (or EOF -- "End of File" -- if no tuples remain)
- `close()`:
  - call close() on child
  - clean up internal state

Note: Scan also applies "selection" filters and "projections" (without duplicate elimination)

## Example: Sort

```
class Sort extends iterator {
   void init();
   tuple next();
   void close();
   iterator inputs[1];
   int numberOfRuns;
   DiskBlock runs[];
   RID nextRID[];
}
```

- `init()`:
  - generate the sorted runs on disk
  - Allocate `runs[]` array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs
- `next()`:
  - nextRID array tells us where we're "up to" in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- "End of File" -- if no tuples remain)
- `close()`:
  - deallocate the runs and nextRID arrays

## Schema for Examples

Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)

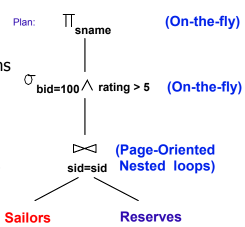
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Let's say there are 100 boats.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - Let's say there are 10 different ratings.
- Assume we have 5 pages in our buffer pool.

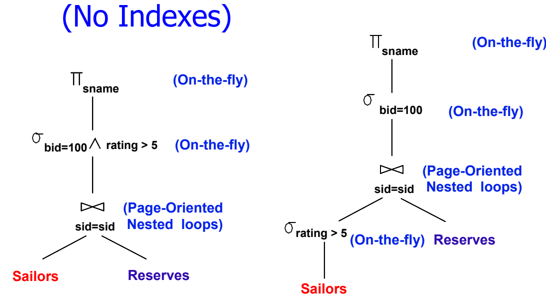
## Motivating Example

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.



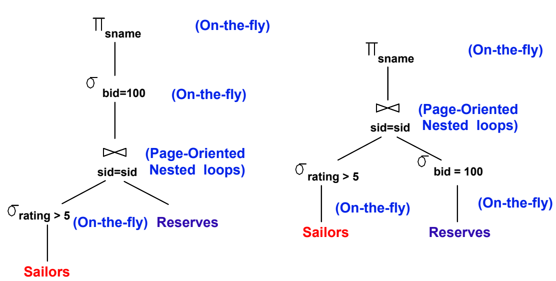

## Alternative Plans – Push Selects (No Indexes)




500,500 IOs

250,500 IOs

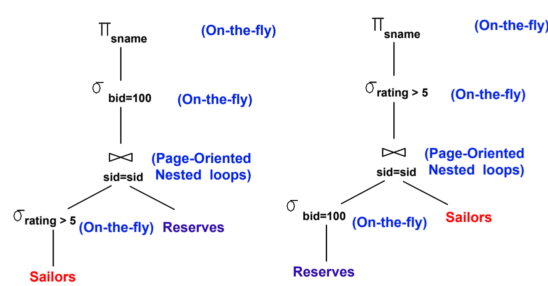
## Alternative Plans – Push Selects (No Indexes)




250,500 IOs

250,500 IOs

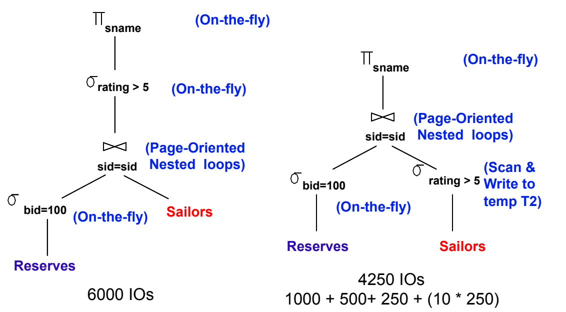
## Alternative Plans – Push Selects (No Indexes)



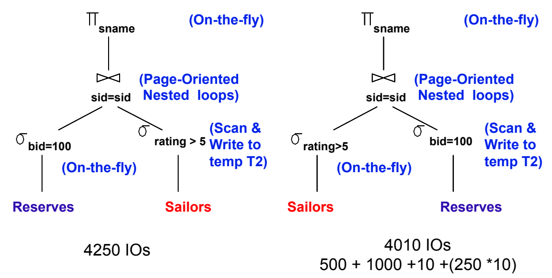

250,500 IOs

6000 IOs

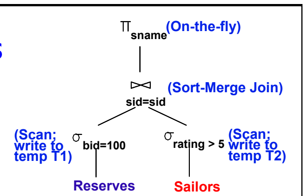
## Alternative Plans – Push Selects (No Indexes)




## Alternative Plans – Push Selects (No Indexes)

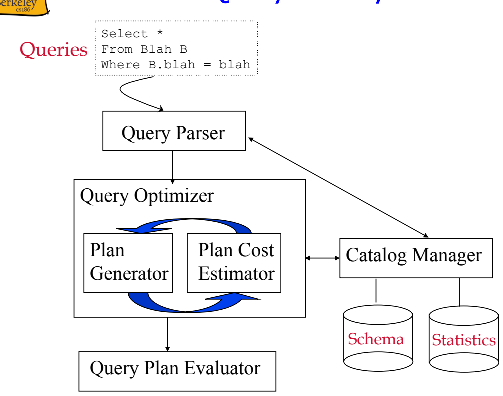



## Alternative Plans




- *Sort Merge Join*
- With 5 buffers, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, w/ 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os. (note: T2 sort takes 4 passes with B=5)
- If use BNL join, join = 10+4*250, total cost = 2770.
- Can also `push' projections, but must be careful!
  - T1 has only *sid*, T2 only *sid*, *sname*:
  - T1 fits in 3 pgs, cost of BNL under 250 pgs, total < 2000.

## Cost-based Query Sub-System



## Relational Model: Summary

- ER is a high-level model that is typically not directly implemented but is "user-friendly"
- Relational Model: A tabular representation of data.
- Simple and intuitive, currently the most widely used
  - Object-relational and XML extensions in most products
- Integrity constraints
  - Specified by the DB designer to capture application semantics.
  - DBMS prevents violations.
  - Some important ICs:
    - primary and foreign keys
    - Domain constraints
- Powerful query languages:
  - SQL is the standard commercial one
    - DDL - Data Definition Language
    - DML - Data Manipulation Language
- Lots of machinery to ensure "declarative"-ness