CMSC424: Relational Algebra; JDBC; Remaining SQL

Instructor: Amol Deshpande
amol@cs.umd.edu

Today's Class

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL: JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Integrity Constraints
  - Advanced Aggregation Features
- Relational Algebra
  - Formal Semantics of SQL (i.e., how to deal with duplicates)
- Other things
  - Exam Wednesday -- everything covered so far, including today
  - Project 3: JDBC; Some advanced SQL; Query Plans
    - Will post a iPython notebook on the last one in a couple of days

Client-server Architectures

Many different possibilities to build an end-to-end application, but often see 2-tier or 3-tier architectures

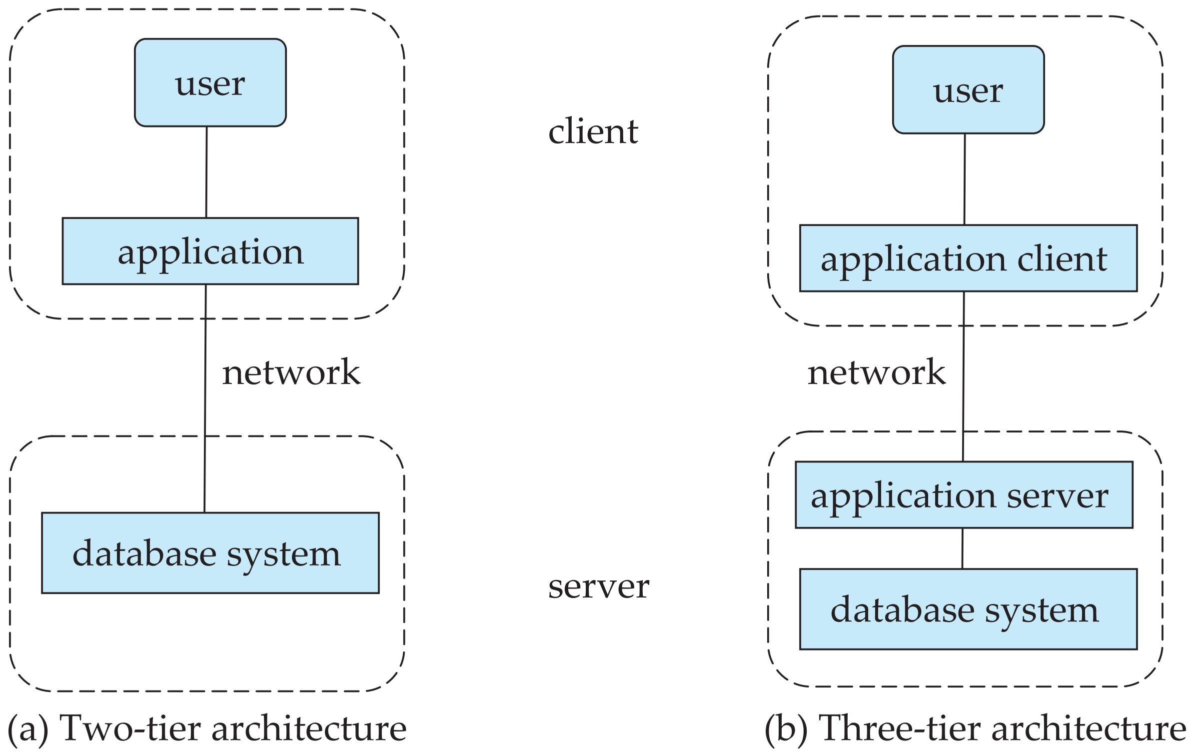
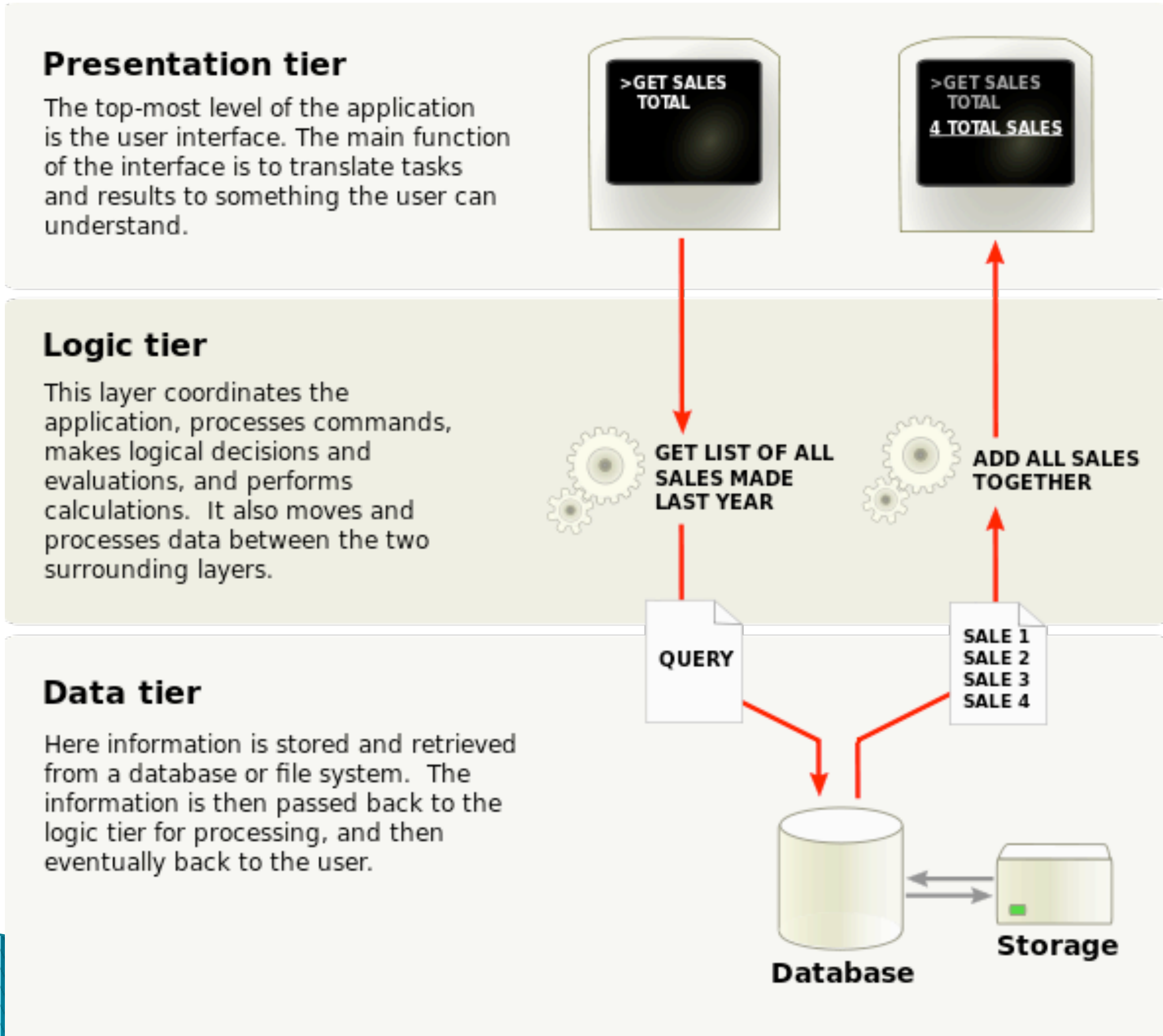


Figure 1.6 Two-tier and three-tier architectures.

Three-tier Architecture



e.g., Web servers

e.g., Ruby on Rails, Java EE, ASP.NET, PHP, ColdFusion, Perl or Python frameworks

e.g., PostgreSQL, Oracle, MySQL, etc…

Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
      - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Advanced Aggregation Features
  - Integrity Constraints
  - Recursion
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)

JDBC and ODBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables

- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
  - Other API's such as ADO.NET sit on top of ODBC

- JDBC (Java Database Connectivity) works with Java

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
  {
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
      Connection conn = DriverManager.getConnection(
             "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
       Statement stmt = conn.createStatement();
           … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
         System.out.println("SQLException : " + sqle);
    }
  }
```

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
                        "select dept_name, avg (salary)
                         from instructor
                         group by dept_name");
while (rset.next()) {
     System.out.println(rset.getString("dept_name") + " " + rset.getFloat(2));
}
```

JDBC Code Details

- Getting result fields:
  - **rs.getString("dept_name") and rs.getString(1) equivalent if dept_name is the first argument of select result.**
- Dealing with Null values
  - **int a = rs.getInt("a");**

    if (rs.wasNull()) Systems.out.println("Got null value");

Prepared Statement

- PreparedStatement pStmt = conn.prepareStatement(

```
                              "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");     pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");   pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- For queries, use pStmt.executeQuery(), which returns a ResultSet
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - **NEVER create a query by concatenating strings which you get as inputs**

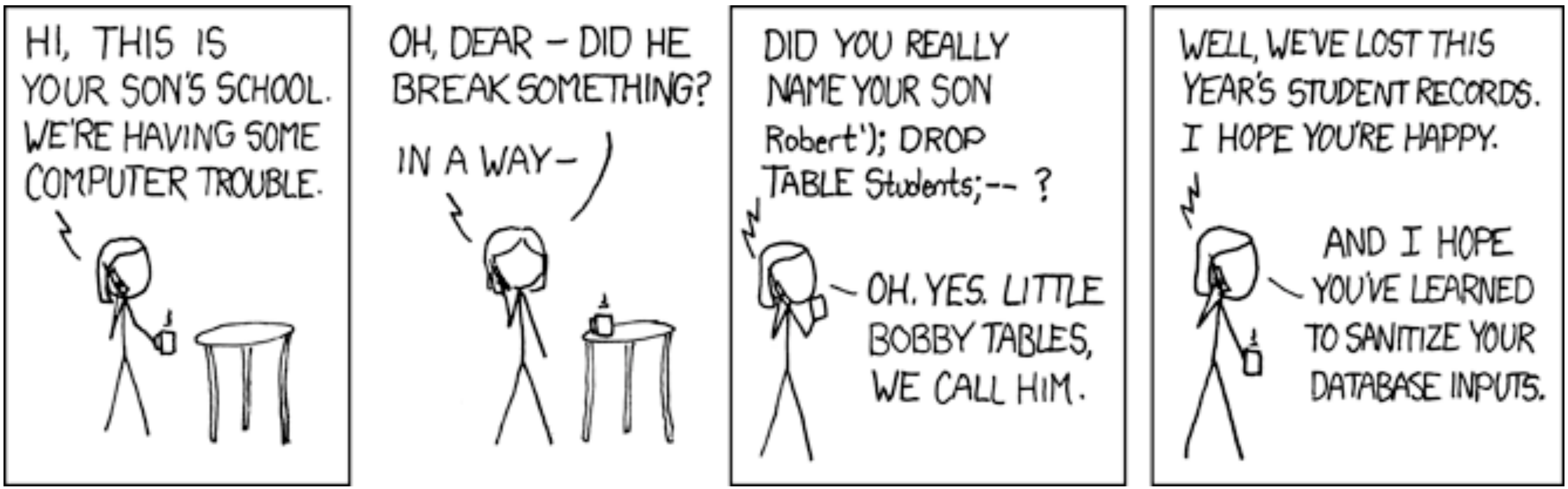
SQL Injection

- String query = "select * from instructor where name = ' " + name + "' "
- User enters: X' or 'Y' = 'Y
- We execute:
  - "select * from instructor where name = ' " + "X' or 'Y' = 'Y" + "' "
  - which is: select * from instructor where name = 'X' or 'Y' = 'Y'
- Worse: user enters:
  - X' ; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'

Always use prepared statements, with user inputs as parameters

https://en.wikipedia.org/wiki/SQL_injection

SQL Injection: XKCD



Metadata Features

- ResultSet metadata
- E.g., after executing query to get a ResultSet rs:
  - ```
    ResultSetMetaData rsmd = rs.getMetaData();
      for(int i = 1; i <= rsmd.getColumnCount(); i++) {
          System.out.println(rsmd.getColumnName(i));
              System.out.println(rsmd.getColumnTypeName(i));
        }
    ```
- Look up the manual etc. for much more
    ```

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

    EXEC SQL <embedded SQL statement > END_EXEC

Note: this varies by language (for example, the Java embedding uses # SQL { …. };)

Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
      - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases)

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
  returns integer
begin
    declare d_count integer;
    select count (* ) into d_count
    from instructor
    where instructor.dept_name = dept_name
    return d_count;
end
```

- Find the department name and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 1
```

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

create function *dept_count* (*dept_name* **varchar**(20))
returns integer
begin
declare *d_count* **integer;**
select count (*) **into** *d_count*
from *instructor*
where *instructor.dept_name = dept_name*
return *d_count;*
end

- Syntax doesn't seem to work with PostgreSQL; see here for examples:
  http://www.postgresql.org/docs/9.1/static/sql-createfunction.html

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

create function *instructors_of* (*dept_name* **char**(20)

returns table (*ID* **varchar**(5),
name **varchar**(20),
dept_name **varchar**(20),
salary **numeric**(8,2))

return table
(**select** *ID, name, dept_name, salary*
from *instructor*
where *instructor.dept_name = instructors_of.dept_name*)

- Usage

select *
from table (*instructors_of* ('Music'))

Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example:

```
declare n integer default 0;
for r as
    select budget from department
    where dept_name = 'Music'
do
    set n = n - r.budget
end for
```

Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
      - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

with recursive *rec_prereq*(*course_id*, *prereq_id*) **as** (
        select *course_id*, *prereq_id*
        from *prereq*
    union
        select *rec_prereq.course_id*, *prereq.prereq_id*
        from *rec_prereq*, *prereq*
        where *rec_prereq.prereq_id* = *prereq.course_id*
)
select ∗
from *rec_prereq*;

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
      - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)

Ranking

- Rank instructors by salary.

```
select *, rank() over (order by salary desc) as s_rank
from instructor;
```

- An extra **order by** clause is needed to get them in sorted order
- Ranking may leave gaps (two with rank 5, none with rank 6)
- **Use dense_rank** to leave no gaps
- Can be done without using new keywords, but probably inefficient

```
select ID, (1 + (select count(*)
                 from instructors i2
                 where i2.salary > i1.salary)) as s_rank
from instructor i1
order by s_rank;
```

Ranking (Cont.)

- Ranking can be done within partition of the data.
- "Find the rank of instrcutors within each department."

select *ID*, *dept_name*,
rank () **over** (**partition by** *dept_name* **order by** *salary* **desc**)
as *dept_rank*
from *instructor*
order by *dept_name*, *dept_rank*;

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

    select *date*, ***sum***(*value*) **over**
    (**order by** *date* **between rows** 1 **preceding and** 1 **following**)
    from *sales*
- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Outline

- Advanced SQL
  - Accessing SQL From a Programming Language
    - Dynamic SQL
      - JDBC and ODBC
    - Embedded SQL
  - Functions and Procedural Constructs
  - Recursion
  - Advanced Aggregation Features
  - Integrity Constraints
- Relational Algebra
- Formal Semantics of SQL (i.e., how to deal with duplicates)

IC's

- Predicates on the database
- Must always be true (checked whenever db gets updated)

- There are the following 4 types of IC's:
  - Key constraints (1 table)
    e.g., *2 accts can't share the same acct_no*
  - Attribute constraints (1 table)
    e.g., *accts must have nonnegative balance*
  - Referential Integrity constraints (2 tables)
    E.g. *bnames* associated w/ *loans* must be names of real branches
  - Global Constraints (*n* tables)
    E.g., all *loans* must be carried by at least 1 *customer* with a savings acct

Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. Primary Key:

```
CREATE TABLE branch(
    bname  CHAR(15)  PRIMARY KEY,
    bcity     CHAR(20),
    assets   INT);
```

or

```
CREATE TABLE depositor(
    cname   CHAR(15),
    acct_no  CHAR(5),
    PRIMARY KEY(cname, acct_no));
```

2. Candidate Keys:

```
CREATE TABLE customer (
    ssn      CHAR(9)    PRIMARY KEY,
    cname  CHAR(15),
    address CHAR(30),
    city         CHAR(10),
    UNIQUE (cname, address, city));
```

Key Constraints

Effect of SQL Key declarations
PRIMARY (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, **reject insertion**

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently
   (e.g.: create an index on PK)

How would you implement something like this ?

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)
- In SQL:

1. NOT NULL

e.g.:
```
CREATE TABLE branch(
    bname   CHAR(15)  NOT NULL,
    ....
)
```

Note: declaring bname as primary key also prevents null values

2. CHECK

e.g.:
```
CREATE TABLE depositor(
    ....
    balance int NOT NULL,
    CHECK(  balance >= 0),
    ....
)
```

affect insertions, update in affected columns

Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g: instead of:

```
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK  (balance >= 0)
    )
```

One can write:

```
CREATE DOMAIN  bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));


CREATE TABLE depositor (
    .....
    balance    bank-balance,
    )
```

Advantages?

Attribute Constraints

Advantage of associating constraints with domains:

1. can avoid repeating specification of same constraint for multiple columns

2. can name constraints

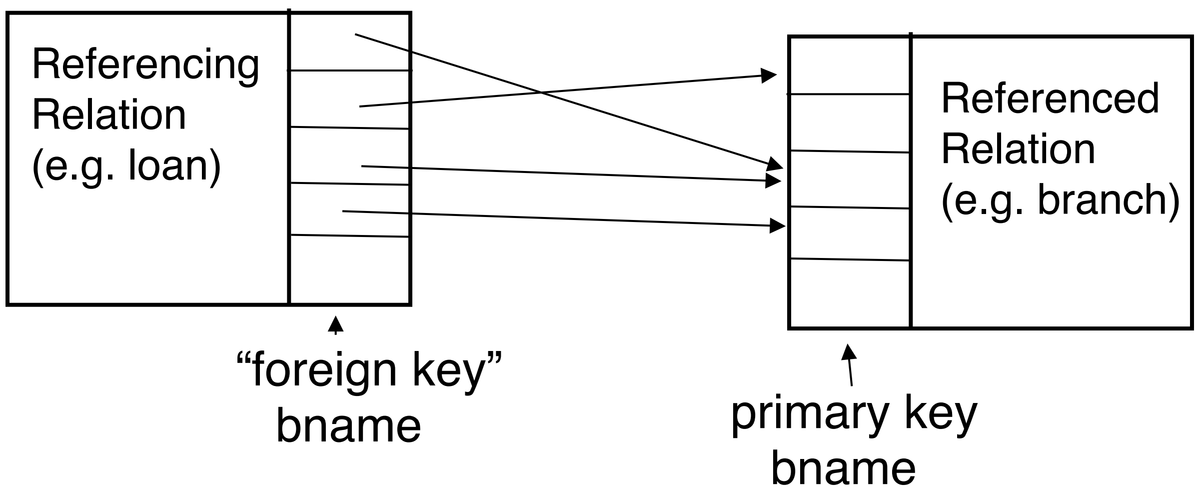
   e.g.:
```
CREATE DOMAIN bank-balance INT (
    CONSTRAINT not-overdrawn
        CHECK (value >= 0),
    CONSTRAINT not-null-value
        CHECK( value NOT NULL));
```

   allows one to:

   1. add or remove:
```
ALTER DOMAIN bank-balance
    ADD CONSTRAINT capped
        CHECK( value <= 10000)
```
   2. report better errors (know which constraint violated)

Referential Integrity Constraints

Idea: prevent "dangling tuples" (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)



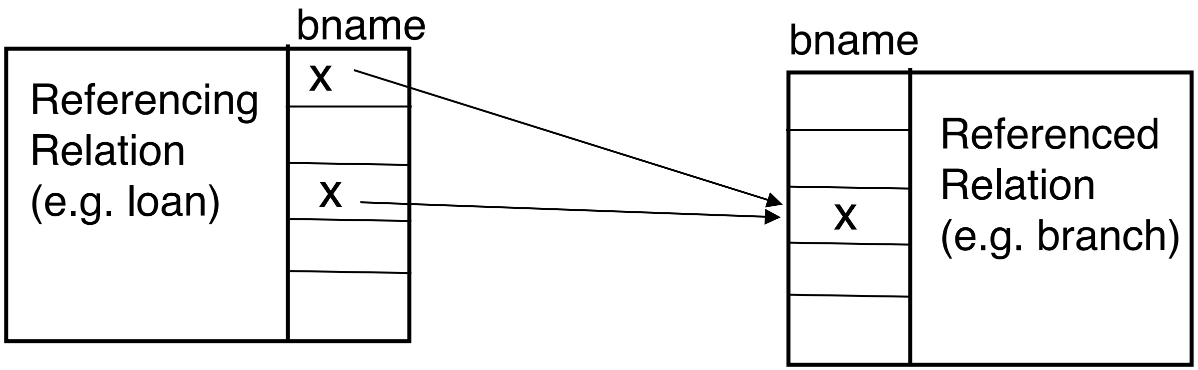
Ref Integrity:

ensure that:

foreign key value → primary key value

(note: don't need to ensure ←, i.e., not all branches have to have loans)

Referential Integrity Constraints



In SQL:

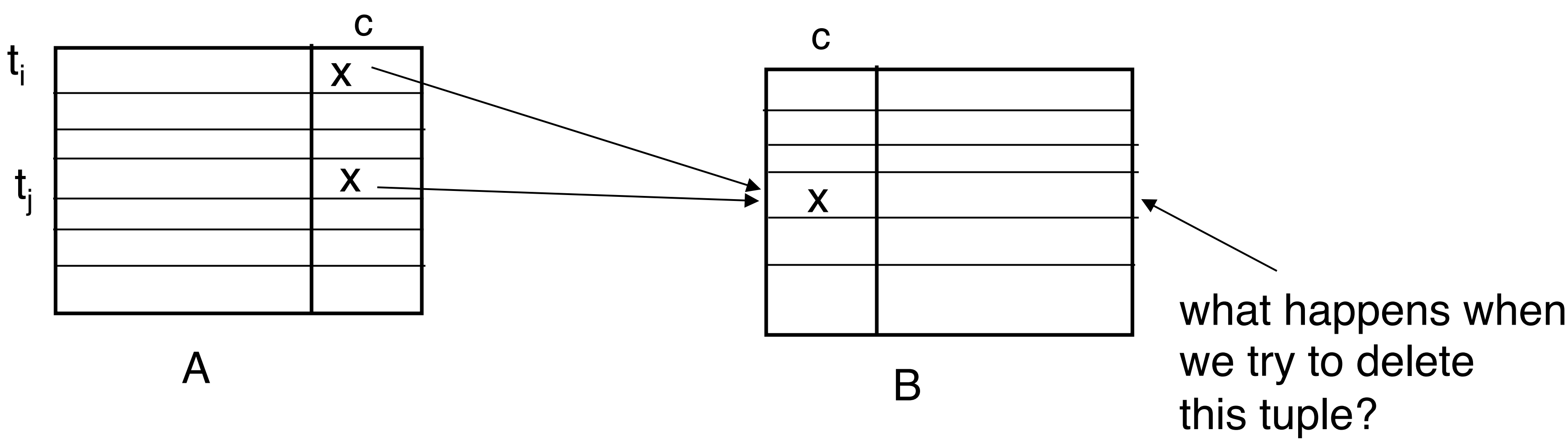
```
CREATE TABLE  branch(
        bname   CHAR(15)   PRIMARY KEY
        ....)

CREATE TABLE loan (
          .........
          FOREIGN KEY bname REFERENCES branch);
```

Affects:

1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation

Referential Integrity Constraints



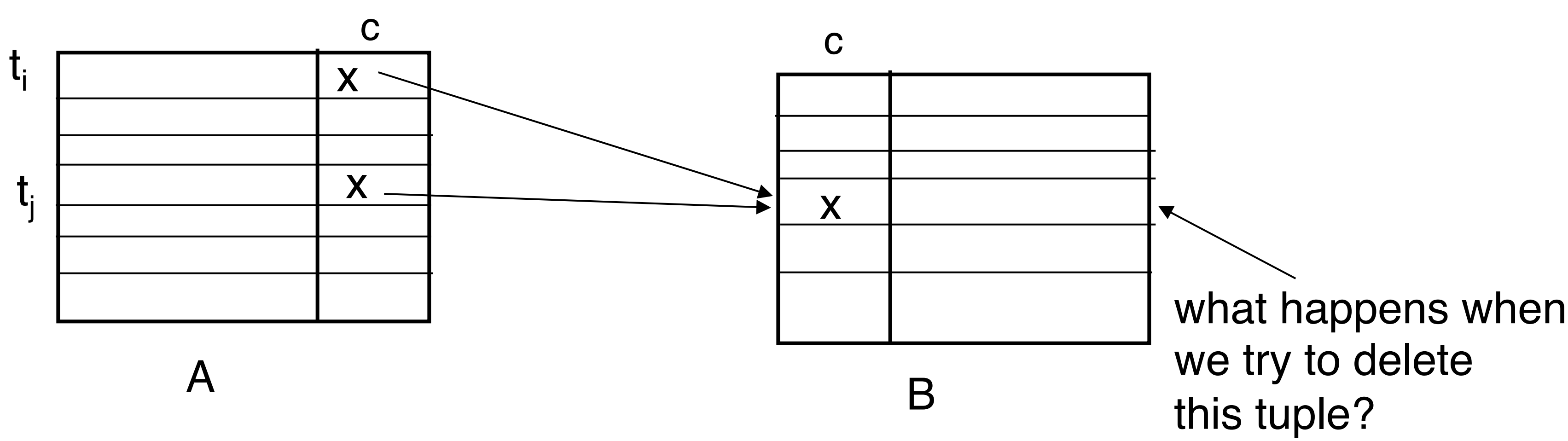
Ans: 3 possibilities

1) reject deletion/ update

2) set $t_i[c]$, $t_j[c]$ = NULL

3) propagate deletion/update
   DELETE: delete ti, tj
   UPDATE: set ti[c], tj[c] to updated values

Referential Integrity Constraints




```
CREATE TABLE A (   .....
          FOREIGN KEY c REFERENCES B  action
          .......... )
```

Action: 1) left blank (deletion/update rejected)

2) ON DELETE SET NULL/ ON UPDATE SET NULL
sets ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
deletes ti, tj
ON UPDATE CASCADE
sets ti[c], tj[c] to new key values

Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   - E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE

2) multiple relations: CREATE ASSERTION

SQL examples:

1) single relation: All Bkln branches must have assets > 5M

```
CREATE TABLE branch (
        ..........
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'Bkln') OR assets > 5M))
```

Affects:

insertions into branch

updates of bcity or assets in branch

Global Constraints

SQL example:

2) Multiple relations: every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
            SELECT   *
            FROM loan AS L
            WHERE  NOT EXISTS(
                  SELECT   *
                  FROM borrower B, depositor D, account A
                  WHERE B.cname = D.cname  AND
                        D.acct_no = A.acct_no  AND
                        L.lno  = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

```
      CREATE ASSERTION loan-constraint
            CHECK(  ..... )
```

Checked with EVERY DB update!
very expensive.....

Summary: Integrity Constraints

Constraint Type	Where declared	Affects...	Expense
Key Constraints	CREATE TABLE (PRIMARY KEY, UNIQUE)	Insertions, Updates	Moderate
Attribute Constraints	CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)	Insertions, Updates	Cheap
Referential Integrity	Table Tag (FOREIGN KEY .... REFERENCES ....)	1.Insertions into referencing rel'n 2. Updates of referencing rel'n of relevant attrs 3. Deletions from referenced rel'n 4. Update of referenced rel'n	1,2: like key constraints. Another reason to index/ sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key
Global Constraints	Table Tag (CHECK) or outside table (CREATE ASSERTION)	1. For single rel'n constraint, with insertion, deletion of relevant attrs 2. For assesrtions w/ every db modification	1. cheap 2. very expensive

Outline

- Advanced SQL
- Relational Algebra

Relational Algebra

- Procedural language

- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename

- The operators take one or more relations as inputs and give a new relation as a result.

Select Operation

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

SQL Equivalent:

```
select *
from r
where A = B and D > 5
```

Unfortunate naming confusion

Project

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\Pi_{A,D}(r)$

A	D
α	7
α	7
β	3
β	10

A	D
α	7
β	3
β	10

SQL Equivalent:

select distinct A, D

from r

Set Union, Difference

Relation r, s

r:

A	B
α	1
α	2
β	1

s:

A	B
α	2
β	3

$r \cup s$:

A	B
α	1
α	2
β	1
β	3

$r - s$:

A	B
α	1
β	1

Must be compatible schemas

What about intersection ?

Can be derived

$r \cap s = r - (r - s)$;

SQL Equivalent:

select * from r
union/except/intersect
select * from s;

This is one case where duplicates are removed.

Cartesian Product

Relation r, s

r:

A	B
α	1
β	2

s:

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

$r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

SQL Equivalent:

```
select distinct *
from r, s
```

Does not remove duplicates.

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

Example:

$$\rho_X(E)$$

returns the expression *E* under the name *X*

If a relational-algebra expression *E* has arity *n*, then

$$\rho_{X\,(A1,\,A2,\,\ldots,\,An)}(E)$$

returns the result of expression *E* under the name *X*, and with the attributes renamed to $A_1, A2, \ldots, An$.

Relational Algebra

- Those are the basic operations

- What about SQL Joins ?
  - Compose multiple operators together

$$\sigma_{A=C}(r \; x \; s)$$

- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment

Additional Operators

- Set intersection (∩)
  - $r \cap s = r - (r - s)$;
  - SQL Equivalent: intersect
- Assignment (←)
  - A convenient way to right complex RA expressions
  - Essentially for creating "temporary" relations
    - $temp1 \leftarrow \prod_{R\text{-}S} (r)$
  - SQL Equivalent: "create table as…"

Additional Operators: Joins

- Natural join ($\bowtie$)
  - A Cartesian product with equality condition on common attributes
  - Example:
    - if *r* has schema *R(A, B, C, D)*, and if *s* has schema *S(E, B, D)*
    - Common attributes: *B* and *D*
    - Then:

$$r \bowtie s = \prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

- SQL Equivalent:
  - select r.A, r.B, r.C, r.D, s.E from r, s where r.B = s.B and r.D = s.D, OR
  - select * from r natural join s

Additional Operators: Joins

- Equi-join
  - A join that only has equality conditions

- Theta-join ($\bowtie_\theta$)
  - $r \bowtie_\theta s = \sigma_\theta(r \, x \, s)$

- Left outer join (⟕)
  - Say *r(A, B), s(B, C)*
  - We need to somehow find the tuples in *r* that have no match in *s*
  - Consider: $(r - \pi_{r.A,\, r.B}(r \bowtie s))$

  - We are done:

$$(r \bowtie s) \quad \cup \quad \rho_{temp\ (A,\, B,\, C)}\ (\ (r - \pi_{r.A,\, r.B}(r \bowtie s)) \times \{(NULL)\}\ )$$

Additional Operators: Join Variations

- Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	$r ⟕ s$	left outer join (with “on”)	(see previous slide)
full outer join	$r ⟗ s$	full outer join (with “on”)	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Additional Operators: Division

- Suitable for queries that have "for all"
  - $r \div s$
- Think of it as "opposite of Cartesian product"
  - $r \div s = t$ *iff* $t \times s \subseteq r$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

÷

A	B
α	1
β	2

=

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (i.e. not maximum)
    - using a copy of *instructor* under a new name *d*
  - $\prod_{instructor.salary} (\sigma_{instructor.salary < d,salary} (instructor \times \rho_d (instructor)))$
  - Step 2: Find the largest salary
  - $\prod_{salary} (instructor) - \prod_{instructor.salary} (\sigma_{instructor.salary < d,salary} (instructor \times \rho_d (instructor)))$

Example Queries

- Find the names of all instructors in the Physics department, along with the *course_id* of all courses they have taught

    - Query 1

$$\Pi_{instructor.ID,course\_id}(\sigma_{dept\_name=\text{“Physics”}}(\sigma_{instructor.ID=teaches.ID}(instructor \times teaches)))$$

    - Query 2

$$\Pi_{instructor.ID,course\_id}(\sigma_{instructor.ID=teaches.ID}(\sigma_{dept\_name=\text{“Physics”}}(instructor) \times teaches))$$

Outline

- SQL Basics
- Relational Algebra
- Formal Semantics of SQL

Duplicates

- By definition, *relations* are *sets*
  - So → No duplicates allowed
- Problem:
  - Not practical to remove duplicates after every operation
  - Why ?
- So…
  - SQL by default does not remove duplicates
- SQL follows *bag* semantics, not *set* semantics
  - Implicitly we keep count of number of copies of each tuple

Formal Semantics of SQL

- RA can only express `SELECT DISTINCT` queries

- To express SQL, must extend RA to a <u>bag</u> algebra
  → *Bags (aka: <u>multisets</u>) like sets, but can have duplicates*

e.g: {5, 3, 3}

e.g: homes =

cname	ccity
Johnson	Brighton
Smith	Perry
Johnson	Brighton
Smith	R.H.

- Next: will define RA^*: a <u>bag</u> version of RA

Formal Semantics of SQL: RA*

1. σ^*_p (r): *preserves copies in* r

e.g: $\sigma^*_{city = Brighton}$ (homes) =

cname	ccity
Johnson	Brighton
Johnson	Brighton

2. $\pi^*_{A1, \ldots, An}$ (r): *no duplicate elimination*

e.g: π^*_{cname} (homes) =

cname
Johnson
Smith
Johnson
Smith

Formal Semantics of SQL: RA*

3. r ∪* s : *additive union*

r

A	B
1	α
1	α
2	β

∪*

s

A	B
2	β
3	α
1	α

=

A	B
1	α
1	α
2	β
2	β
3	α
1	α

4. r -* s: *bag difference*

e.g: r -* s =

A	B
1	α

s -* r =

A	B
3	α

Formal Semantics of SQL: RA*

5. r $\times^*$ s: *cartesian product*

A	B
1	α
1	α
2	β

$\times^*$

C
+
-

=

A	B	C
1	α	+
1	α	-
1	α	+
1	α	-
2	β	+
2	β	-

Formal Semantics of SQL

Query:

```
SELECT      a1, ….., an
FROM        r1, ….., rm
WHERE       p
```

Semantics: $\pi^*_{A1, \ldots, An} (\sigma^*_p (r_1 \times * \ldots \times * r_m))$ (1)

Query:

```
SELECT DISTINCT   a1, ….., an
FROM              r1, ….., rm
WHERE             p
```

Semantics: *What is the only operator to change in (1)?*

$\pi_{A1, \ldots, An} (\sigma^*_p (r_1 \times * \ldots \times * r_m))$ (2)

Set/Bag Operations Revisited

- Set Operations
  - `UNION` ≡ $\cup$
  - `INTERSECT` ≡ $\cap$
  - `EXCEPT` ≡ $-$

Bag Operations

`UNION ALL` ≡ $\cup^*$
`INTERSECT ALL` ≡ $\cap^*$
`EXCEPT ALL` ≡ $-^*$

Duplicate Counting:

Given m copies of t in r, n copies of t in s, how many copies of t in:

`r UNION ALL s?` A: $m + n$

`r INTERSECT ALL s?` *A:* $\min(m, n)$

`r EXCEPT ALL s?` *A:* $\max(0, m-n)$

SQL: Summary

Clause	Eval Order	Semantics (RA/RA*)
`SELECT [(DISTINCT)]`	4	π (or π^*)
`FROM`	1	$\times^*$
`WHERE`	2	σ^*
`INTO`	7	$\leftarrow$
`GROUP BY`	3	Extended relational operator *g*
`HAVING`	5	σ^*
`ORDER BY`	6	Can't express: requires ordered sets, bags
`AS`	-	ρ
`UNION ALL`	8	$\cup^*$
`UNION` `(similarly intersection, except)`		$\cup$

SQL

- Is that it ?
  - Unfortunately No
  - SQL 3 standard is several hundreds of pages (if not several thousands)
  - And expensive too..

- We will discuss a few more constructs along the way

  E.g. *Embedded SQL, creating indexes etc*

- Again, this is what the reference books are for; you just need to know where to look in the reference book