
Biological Database Design

Week 1

Winter 2007
Melanie Nelson, Ph.D.

Introductions

- About me
 - Ph.D. in biochemistry (1999)
 - Bioinformatics/IT in Biotech
 - Physiome Sciences (now Predix Pharmaceuticals)
 - GeneFormatics (now Cengent Therapeutics)
 - SAIC
 - Experience with
 - Databases
 - XML
 - Perl
 - Protein Structure/Function Analysis
 - E-mail: m-nelson-1@alumni.uchicago.edu (1 day turnaround)
- Class introductions

Course Overview

■ Goals

- Intro to DBs
- Overview of common types of biological data
- Introduction to biology-specific problems/issues

■ Grading

- 2% participation
- 10% homework
- 38% midterm
- 50% final project

■ Schedule: Thursdays, 7-10 p.m. In practice, most classes will finish by 9 or 9:30.

- No class Jan. 25
- Will work through examples, answer questions, etc. either from 6 or 6:30 (if room is available) or from ~9-10.

Course Overview

- No Required Book. Some Suggestions:
 - Handbook for Relational Database Design (Fleming and von Halle): old, but covers basics well
 - An Introduction to Database Systems (Date): the classic: frequently updated but overkill for this course
 - Database in Depth (Date): concise summary of database theory, geared at people who already work with databases
- I strongly recommend students without a database background choose a book and read it!
- Readings are supplemented by articles supplied on website.
 - I strongly recommend students without a bio background plan on reading recommended bio resources!
- Online resources
 - My website:
www.32geeks.com/classes/biodb_design_2007

Course Overview

■ Week 1

- Introduction to databases
- Fundamentals of the relational model (includes a brief intro to SQL)

■ Week 2

- Database design process
- ER diagrams
- Normalization

■ Week 3

- Bio data 1: Gene and protein sequences and metadata
- Midterm

Course Overview

■ Week 4

- Bio data 2: gene expression
- Bio data 3: LIMS
- Project plans due

■ Week 5

- Biology-specific issues in database design

■ Week 6

- Biology-specific issues in database design and/or special topics (time permitting)
- Final project presentations

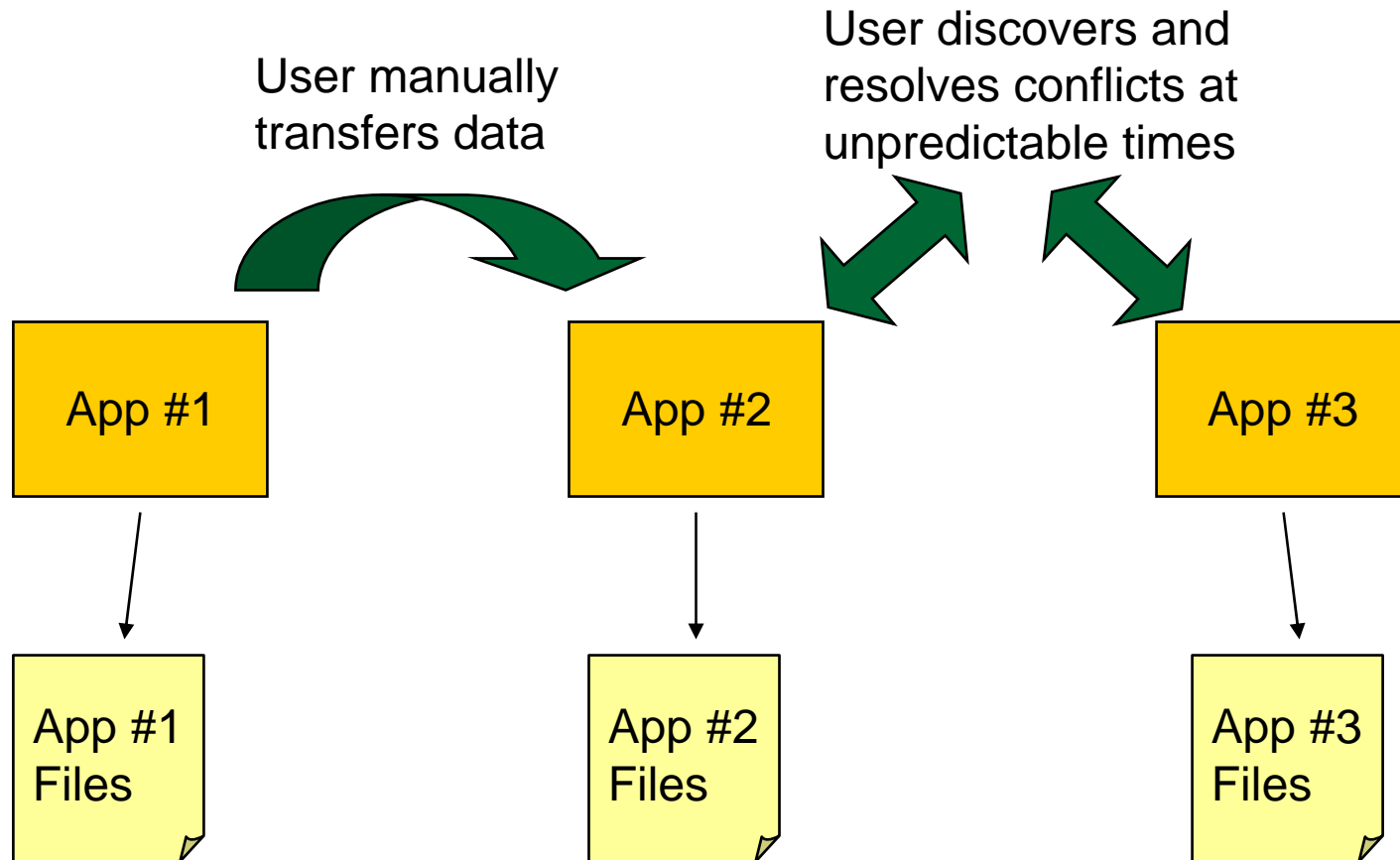
Final Project

- Design a database to store biological data. The database must integrate at least two sources of data.
- Can work alone or in teams of up to three members
- Week 4: Hand in plan
 - What type of biological data will be stored
 - Scope statement (what aspects of the data are to be covered by your database)
- Week 6: Hand in and present design
 - Requirements document
 - ER or UML diagram
 - Short (1-2 page) report describing any difficult or unusual design decisions
 - Make 10-15 minute presentation about DB to class

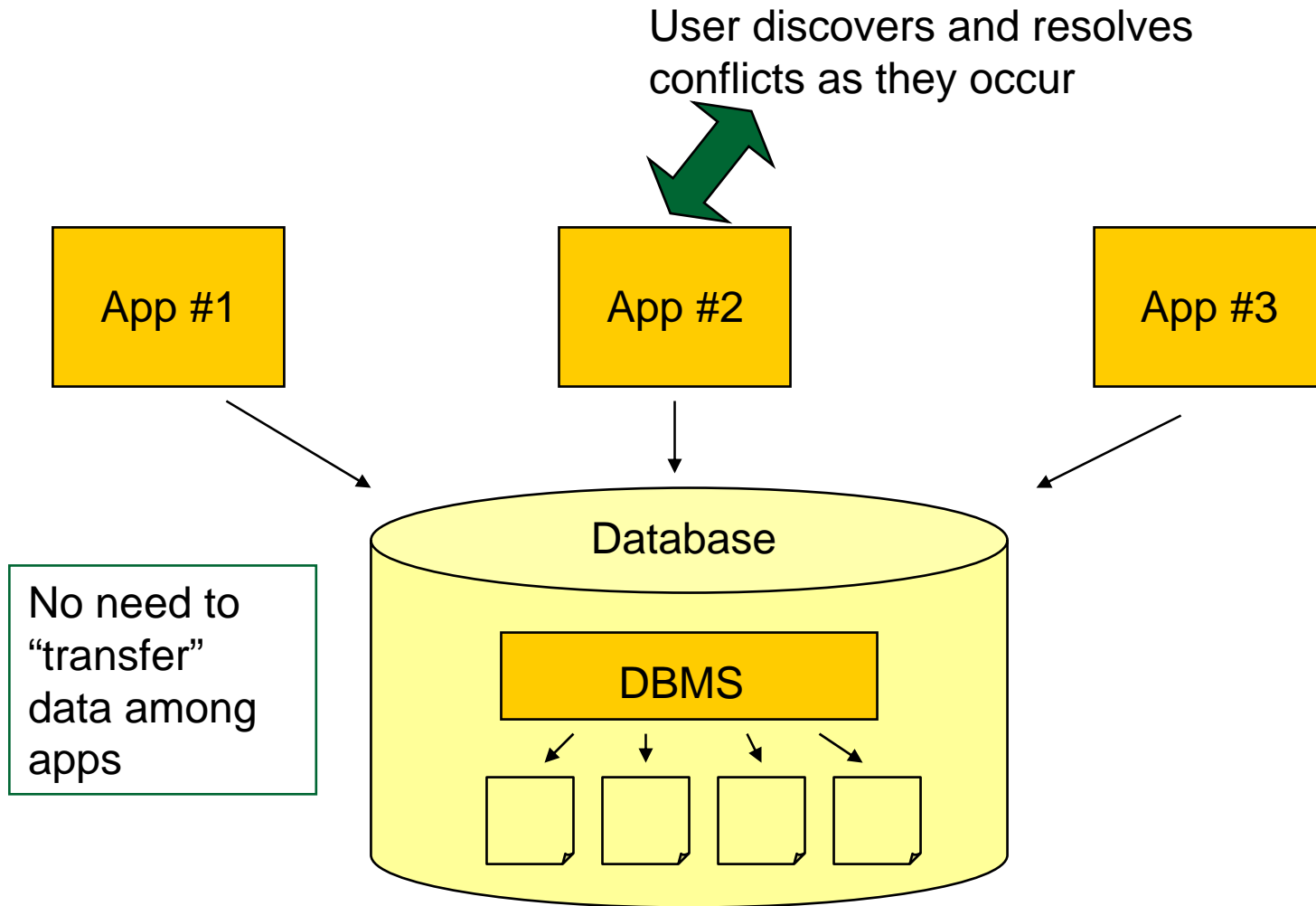
Introduction to Databases

- Information in a database is
 - Structured (searchable)
 - Capable of being shared with multiple applications (multiple uses)
 - Databases are supported by a database management system
 - Layer between applications using the data and the raw data
 - Handles requests for data
 - Manages concurrency
 - Protects data integrity
- } consistency

Data Management without Databases



Data Management with Databases



Some Advantages of Databases

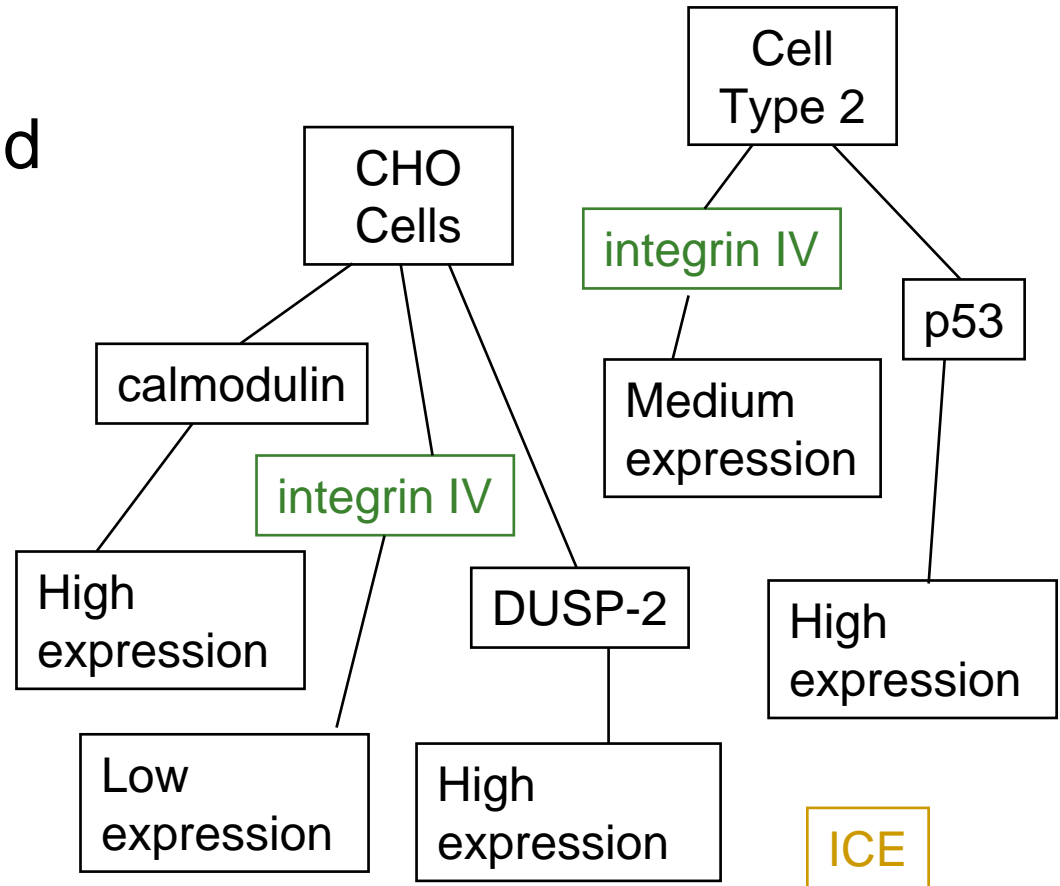
- **Improve interoperability:** app #1 has “sequence”, app #2 has “prot_seq”. Are they the same thing?
- **Reduce inconsistency:** app #1 says protein A binds drug Z, app #2 says it doesn't. Which is right?
- **Improve efficiency:** scientists/programmers don't have to gather data for each application/question

Types of Database Systems

- Four main types of databases:
 - Hierarchical
 - Network
 - Relational
 - Object-Oriented

Hierarchical Databases

- Information organized into tree, or parent-child relationships
- Data gets **duplicated** when one child has more than one parent
- Data gets **lost** when a child doesn't have a parent

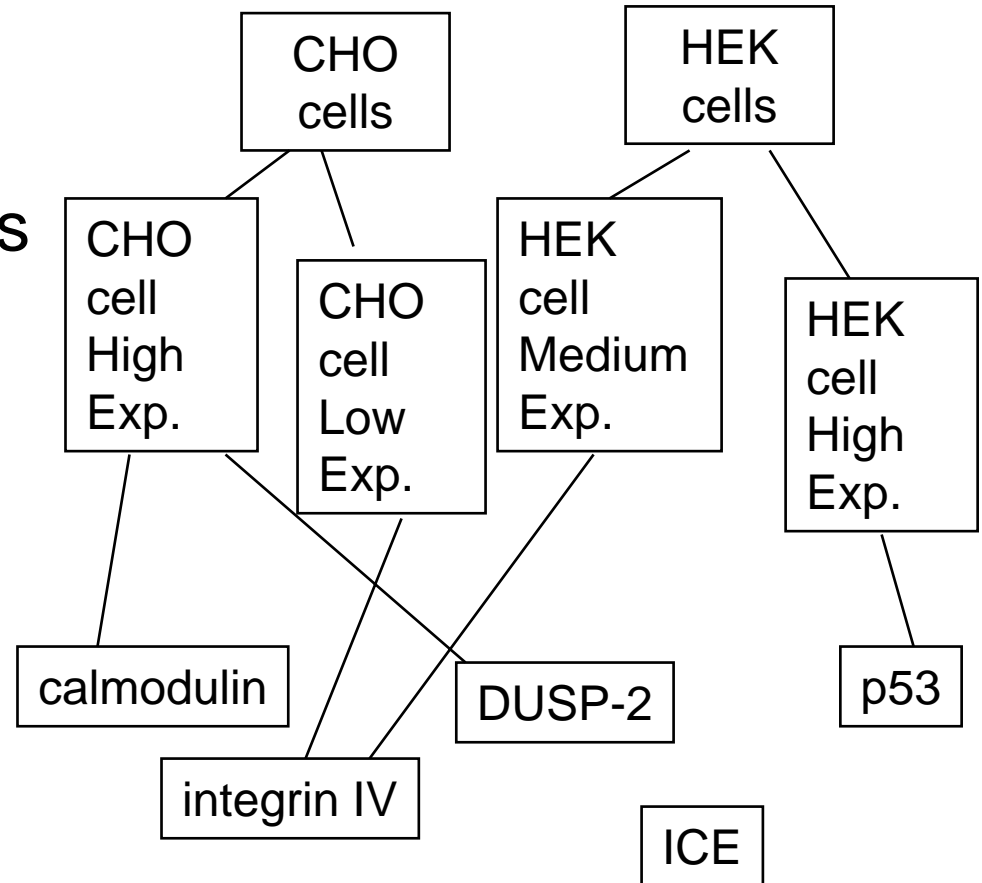


Hierarchical Databases

- Historically, the first type of database
 - IBM's Information Management System (IMS)
 - Introduced in 1968
- XML can be viewed as a hierarchical database
 - Information is organized into tree
 - Collections of XML files can be used as a database

Network Databases

- Extends hierarchical model to allow children to have multiple parents
- Model has:
 - Records
 - Links between records
- Careful design can avoid data duplication
- Complicated design and data access



Network Databases

- Emerged in the 70s
- Conference on Data Systems Languages (CODASYL) produced guidelines for databases
- XML with XLink can be viewed as a network database
 - XLink allows links across branches in the XML tree

Relational Databases

- Information is modeled as tables (relations) with links between tables
- Rigorous mathematical basis
 - Allows prevention of data duplication and other data integrity problems
 - Simplifies data access

Cell Line

Cell line ID	Cell line type
Cell line 1	CHO cells
Cell line 2	HEK cells

Protein Expression

Cell line ID	Protein ID	Expression level
Cell line 1	Protein 1	High
Cell line 1	Protein 2	Low
Cell line 1	Protein 3	High
Cell line 2	Protein 2	Medium
Cell line 2	Protein 4	High

Protein

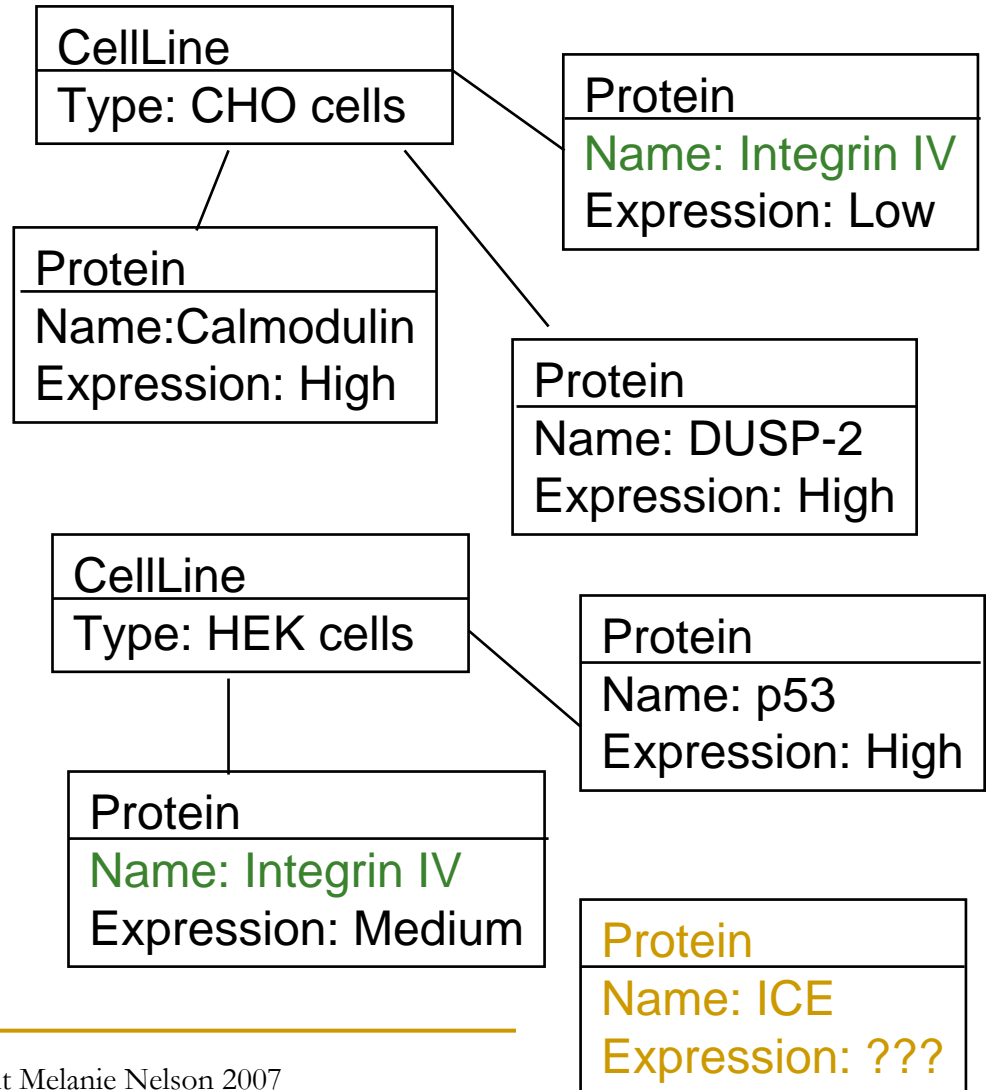
Protein ID	Protein Name
Protein 1	calmodulin
Protein 2	integrin IV
Protein 3	DUSP-2
Protein 4	ICE
Protein 5	p53

Relational Databases

- Developed in 70s by Dr. E.F. Codd at IBM
- Is the dominant model in use today
 - Oracle
 - IBM DB2
 - MS SQL Server
 - PostgreSQL
 - MySQL

Object-Oriented Databases

- OODBs store data in classes, with associations between classes
- Integrates data storage with data manipulation: methods are part of object
- Must be careful to avoid data duplication and “orphan” data



Object-Oriented Databases

- Introduced in 80s, in conjunction with rise in object-oriented programming techniques
 - There are difficulties integrating OO programming and relational DBs
 - OODBs often have the same problems network DBs had
- Lack easy data access of relational DBs
- Major relational DBs have introduced “object extensions”
- Ongoing debate about how best to integrate DBs and OO programming
 - Many solutions now available to assist with this integration

Relational vs the Other Models

- Relational model attempts to correctly represent data, without regard to how it will be used
- In other models, how the data will be used can greatly influence the design
 - If you design to a particular application, you will probably make it easy to answer the questions in that application...
 - But you may make it harder, or even impossible, to answer other types of questions!

Relational vs. Other Models

- Relational DBs were intended to free users from needing a programmer to write new code to answer each new question
- This is particularly useful in science: scientists will *always* think of a new question!
- SQL still too “programming-like” for many users
 - Flexible reporting apps attempt to address this

The Relational Model

- Direct quote from Date:
 - Data is perceived by users as tables (and nothing but tables)
 - The operators at the user's disposal...are operators that generate new tables from old, and those operators include at least `SELECT...`, `PROJECT`, and `JOIN`

The Relational Model

- The relational model speaks to:
 - Data structure
 - Data manipulation
 - Data integrity
- It does not speak to data storage
- Relational model refers to **logical** database design, not **physical** database design

The Relational Model

- Mathematically rigorous
- When correctly implemented, can guarantee accuracy of query results (assuming input was valid!)
- No current DBMS fully implements the relational model

Relational Terms

Relation = Table
Consists of

- heading (a fixed set of attributes)
- body (a set of tuples)

Attribute = Column
Also called a field

Tuple = Row
Also called a record.
A set of attribute:value pairs

Proteins

Protein ID	Protein Name
Protein 1	calmodulin
Protein 2	integrin IV
Protein 3	DUSP-2
Protein 4	ICE
Protein 5	p53

Primary key = Unique identifier
Attribute or combination of attributes that uniquely identifies each tuple

Domain = Valid set of values
“A named set of scalar values”
Each attribute has a domain upon which it is defined

Properties of Relational Tables

- The following properties are a consequence of the definition of relations, attributes, and domains:
 - Each column has a unique name (The heading = a fixed set of attributes)
 - All entries in a given column are of the same kind (Attributes are defined on a domain)

Properties of Relational Tables

- There are no duplicate tuples
 - “Each row is unique”
 - The body of a relation is a mathematical set: sets do not have duplicate elements
 - Primary key ensures this rule is upheld
 - Do not circumvent!
 - Common to use system-assigned numerical value as primary key
 - Should have an “alternate key” that is inherent in the data

Properties of Relational Tables

- The sequence of tuples is unimportant
 - Sets are unordered
 - DBA may change way in which rows are partitioned in storage to improve performance of certain queries
 - Never write code that assumes a query will return results in a given order
 - If tuple order is meaningful, it should be specified by an attribute

Properties of Relational Tables

- The sequence of attributes is unimportant
 - The heading of a relation is also a set
 - DBA may change physical order of columns to improve performance of certain queries
 - Never assume the columns will be returned in a given order: specify the order in the query

Properties of Relational Tables

- Attribute values are atomic
 - “Entries in columns are single-valued”
 - First normal form

Protein ID	Protein Name
Protein 1	Calmodulin, CaM
Protein 3	DUSP-2, dual specificity phosphatase 2, PAC1

Protein ID	Protein Name
Protein 1	Calmodulin
Protein 1	CaM
Protein 3	DUSP-2
Protein 3	Dual specificity phosphatase 2
Protein 3	PAC1

Protein ID	Protein Name 1	Protein Name 2	Protein Name 3
Protein 1	Calmodulin	Ca	
Protein 2	DUSP-2	Dual specificity phosphatase 2	PAC1

Types of Relations

- **Base relation** = an autonomous relation (i.e., not defined in terms of another relation)
 - What we typically mean when we talk about database tables
- **Derived relation** = a relation defined in terms of other relations
 - Query results, for instance
- **View** = a named derived relation
 - SQL to generate derived relation is stored in database
- **Materialized view** = a view in which data is actually copied
 - “snapshot”
 - Used to improve performance
 - Can lead to integrity issues

Data Integrity

- Data in the database is meant to represent “reality”
- Certain combinations of values are not possible in the real world, so the database should exclude them
- Rules apply to base relations
- Three types:
 - Entity Integrity
 - Referential Integrity
 - “Domain Integrity” (other rules)

Candidate Keys

- Candidate keys
 - A candidate key can uniquely identify each row
 - A candidate key cannot be reduced: i.e., there is no subset of the attributes in the key that also uniquely identify each row
- Primary key is the candidate key chosen to be used
- Alternate keys = candidate keys not chosen to be primary key

Entity Integrity

- No part of the primary key may be NULL
- NULL = absence of value
 - Value doesn't exist
 - Value isn't known
- The primary key uniquely identifies a row
 - If part is NULL, it means that we do not know the value
 - It could be a value that is already represented in the table
 - Therefore, we can't uniquely identify the row

Referential Integrity: Foreign Keys

- Links between two related tables are made via foreign keys
- Foreign key = the primary key of a related table

Species

Species_ID	Species_Sci_Name	Species_Common_Name	Study_in_Lab
1	Homo sapiens	human	Y
4	Mus musculus	house mouse	Y
56	Bos taurus	cow	N

Primary key of parent table

Available_Protein

Protein_ID	Protein_Name	Species_ID
Protein 1	calmodulin	1
Protein 2	integrin IV	56
Protein 3	DUSP-2	4

Foreign key of child table

Referential Integrity

- A foreign key value must either
 - Match a primary key value in the referenced table
 - Be NULL

Species

Species_ID	Species_Sci_Name	Species_Common_Name	Study_in_Lab
1	Homo sapiens	human	Y
4	Mus musculus	house mouse	Y
56	Bos taurus	cow	N

Available_Protein

Protein_ID	Protein_Name	Species_ID
Protein 1	calmodulin	1
Protein 2	integrin IV	56
Protein 3	DUSP-2	4
Protein 4	PTP1B	72

Referential Integrity

- Prevents “orphan” rows in child table
 - Child data usually loses significant meaning without parent information
- In practice, allowing a foreign key to be NULL can create problems
- In practice, NULLs can create problems!
 - What does it mean? Value doesn't exist or value unknown?
 - Consider using defaults instead

Domain Integrity

- Attribute integrity

- Values of an attribute are taken from the specified domain
- Domain support in database management systems is weak

- Business rules

- All the other rules the data must follow
- Implemented in triggers, stored procedures, application logic

Data Definition and Manipulation

- Any functioning DBMS must provide a language for data definition and manipulation
 - Data definition = a way to create relations and store data in them
 - Data manipulation = a way to get data back out
- Codd's papers provided a relational algebra and a relational calculus
- SQL is the standard language by which this is implemented

Properties of Data Manipulation

- Closure: relational operators operate on relations and produce relations
 - Allows nested expressions
- Relational operators are not affected by changes to physical storage of data

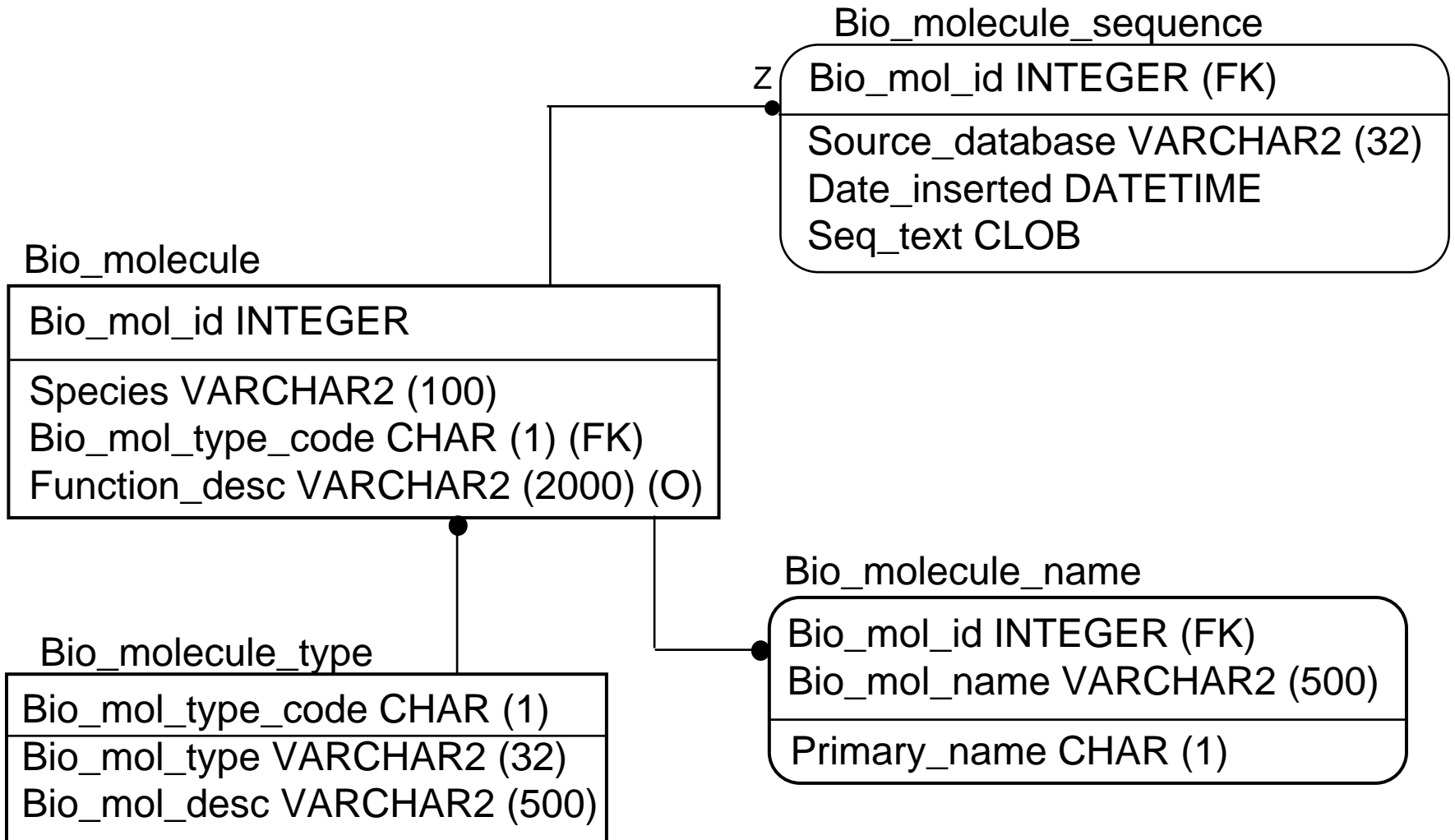
Introduction to SQL

- SQL = Structured Query Language
 - Except that the spec says SQL doesn't stand for anything
- Standard language for storing and accessing data in relational databases
- A nonprocedural language
 - Say what you want, not how to get it
 - A RDBMS has a query optimizer that figures out how to get the data
- RDBMS purists point out that it is not fully compliant with relational database theory
 - Poor support of domains
 - Allows tables without keys

Introduction to SQL

- Data Definition Language (DDL)
 - CREATE TABLE, DROP TABLE
 - CREATE INDEX
 - Constraints: UNIQUE, PRIMARY KEY, FOREIGN KEY, NOT NULL
- Data Manipulation Language (DML)
 - INSERT, UPDATE, DELETE
 - SELECT
 - UNION, INTERSECT, EXCEPT

Example Tables



CREATE TABLE

- Use to create a table
- **CREATE TABLE** table1
(column1 datatype **PRIMARY KEY**,
column2 datatype)
- Each table should have a primary key constraint on one or more columns
- Use **UNIQUE** to enforce alternate keys

CREATE TABLE

Create a table to store biological molecules

```
CREATE TABLE Bio_molecule (  
    Bio_mol_id INTEGER PRIMARY KEY,  
    Species VARCHAR2 (50) NOT NULL,  
    Bio_mol_type_code CHAR (1) NOT NULL,  
    Function_desc VARCHAR2 (2000)  
)
```

PRIMARY KEY is equivalent to **UNIQUE, NOT NULL**

Other DDL Commands

■ ALTER TABLE

- ❑ Add/drop/modify a column of a table
- ❑ Not all DBMS support drop and modify

■ CREATE INDEX

- ❑ Create an index on a column or combination of columns
- ❑ Implementation detail: indexes are used by DBMS to enforce constraints and optimize lookup
- ❑ UNIQUE constraints automatically create index

■ DROP TABLE, DROP INDEX

INSERT

- Use INSERT to get data into a table
- **INSERT INTO** table1 (column list)
VALUES (value list)
- Column list is optional, but should specify it if the statement is included in application code
 - Remember, the columns in a table are not in any particular order!

INSERT

Insert the name “PTP1B” for biological molecule #1456. It is a primary name.

```
INSERT INTO Bio_molecule_name  
  (Bio_mol_id, Bio_mol_name, Primary_name)  
VALUES (1456, 'PTP1B', 'Y')
```

Text is surrounded by single quotes.

UPDATE

- Use to alter data in a table
- **UPDATE** table1
SET column1 = new value,
column2 = new value
WHERE column3 = condition
- WHERE clause is optional. Without it, the UPDATE will apply to all rows in the table

UPDATE

Change calmodulin to be the primary name.

```
UPDATE Bio_molecule_name
SET Primary_name = 'Y'
WHERE Bio_mol_name = 'calmodulin'
AND Bio_mol_id = 1
```

Bio_mol_id portion of where clause is probably unnecessary.

DELETE

- Removes row(s) from table
- **DELETE FROM** table1
WHERE column1 = condition
- **WHERE** clause is optional. Without it, **DELETE** will remove all rows from the table.
 - Won't remove table
 - To do this, use **DROP TABLE**

DELETE

Delete all Incyte sequence data

```
DELETE FROM Bio_molecule_sequence  
WHERE Source_database = 'INCYTE'
```

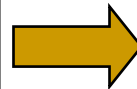
Relational Operators: Select

- Also called restrict
- Retrieve a subset of rows (tuples) from a relation
- Subset is determined by a selection criteria
- SELECT *

```
FROM Bio_molecule_name  
WHERE Bio_mol_id = 1
```

Bio_molecule_name

Bio_mol_id	Bio_mol_name	Primary_name
1	Calmodulin	Y
1	CaM	N
3	DUSP-2	N
3	Dual specificity phosphatase 2	Y
3	PAC1	N



List all the names of Biomolecule 1

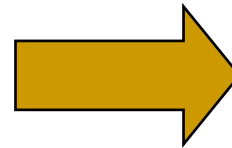
Bio_mol_id	Bio_mol_name	Primary_name
1	Calmodulin	Y
1	CaM	N

Relational Operators: Project

- Retrieve a subset of columns (attributes) from a relation
- `SELECT Bio_mol_id, Bio_mol_name`
`FROM Bio_molecule_name`
`WHERE Primary_name='Y'`

Bio_molecule_name

Bio_mol_id	Bio_mol_name	Primary_name
1	Calmodulin	Y
1	CaM	N
3	DUSP-2	N
3	Dual specificity phosphatase 2	Y
3	PAC1	N



Get a list of primary names of molecules

Bio_mol_id	Bio_mol_name
1	Calmodulin
3	Dual specificity phosphatase 2

SELECT

- Use to get information out of tables
- **SELECT** column1, column2
FROM table1
WHERE column3 = condition
- **WHERE** clause is optional. Without it, the statement returns all rows in the table

SELECT

- List the primary name and bio_mol_id for all molecules:
 - ```
SELECT Bio_mol_id, Bio_mol_name
FROM Bio_molecule_name
WHERE Primary_name = 'Y'
```
- List all biological molecules stored in the database:
  - ```
SELECT *
FROM Bio_molecule
```

SELECT DISTINCT

- Use to get a list of distinct values
- **SELECT DISTINCT** (column1, column2)
FROM table1
- Can have one or more columns in the select statement
- Multiple columns will provide distinct combinations of values of those columns

SELECT DISTINCT

Find out what types of biological molecules are represented in the Bio_molecule table:

```
SELECT DISTINCT Bio_mol_type_code  
FROM Bio_molecule
```

Relational Operators: Product

- A cartesian product of two relations
- Each row in relation 1 is combined with each row in relation 2
- **SELECT ***
FROM Bio_molecule, Bio_molecule_type

Relational Operators: Product

Bio_molecule

Bio_mol_id	Species	Bio_mol_type_code	Function_desc
1	Homo sapiens	P	Calcium sensor
3	Mus musculus	M	Phosphatase



Bio_molecule_type

Bio_mole_type_code	Bio_mol_type	Bio_mol_desc
P	Protein	Expressed protein
M	mRNA	Messenger RNA

bm. Bio_mol_id	bm. Species	bm. Bio_mol_type_code	bm. Function_desc	bmt. Bio_mol_type_code	bmt. Bio_mol_type	bmt. Bio_mol_desc
1	Homo sapiens	P	Calcium sensor	P	Protein	Expressed protein
1	Homo sapiens	P	Calcium sensor	M	mRNA	Messenger RNA
3	Mus musculus	M	Phosphatase	P	Protein	Expressed protein
3	Mus musculus	M	Phosphatase	M	mRNA	Messenger RNA

Relational Operators

■ Join

- ❑ Combination of product and select
- ❑ Combines row from relation 1 with row from relation 2 only when selection criteria are met
- ❑ Criteria specify when rows are to be combined
- ❑ `SELECT *`

```
FROM Bio_molecule, Bio_molecule_type
```

```
WHERE Bio_molecule.Bio_mol_type_code =  
Bio_molecule_type.Bio_mol_type_code
```

Relational Operators: Join

Bio_molecule

Bio_mol_id	Species	Bio_mol_type_code	Function_desc
1	Homo sapiens	P	Calcium sensor
3	Mus musculus	M	Phosphatase

Bio_molecule_type

Bio_mole_type_code	Bio_mol_type	Bio_mol_desc
P	Protein	Expressed protein
M	mRNA	Messenger RNA



Include biomolecule type in molecule information

bm. Bio_mol_id	bm. Species	bm. Bio_mol_type_code	bm. Function_desc	bmt. Bio_mol_type_code	bmt. Bio_mol_type	bmt. Bio_mol_desc
1	Homo sapiens	P	Calcium sensor	P	Protein	Expressed protein
3	Mus musculus	M	Phosphatase	M	mRNA	Messenger RNA

More meaningful than a product!
More likely to combine with a project and
exclude the bio_mol_type_code.

Relational Operators: Join

- Types of join
 - Equi-join
 - Join criterion is equality of attribute(s) in two tables
 - Natural join
 - Equi-join in which redundant columns are removed from the result set
 - Outer join
 - Returned relation includes rows that are missing from one of the original tables

JOIN

- Joins are used to combine information from multiple tables
- Two types of syntax
- **SELECT** table1.column1, table2.column2
FROM table1, table2
WHERE table1.column3 = table2.column3
- **SELECT** table1.column1, table2.column2
FROM table1
JOIN table 2 **ON** (table1.column3 = table2.column3)

JOIN

Show the biomolecule type, rather than the code, for all types represented in Bio_molecule:

```
SELECT DISTINCT Bio_mol_type
FROM Bio_molecule bm,
     Bio_molecule_type bmt
WHERE bm.Bio_mol_type_code = bmt.Bio_mol_type_code
```

```
SELECT DISTINCT Bio_mol_type
FROM Bio_molecule bm
JOIN Bio_molecule_type bmt
     ON bm.Bio_mole_type_code = bmt.Bio_mol_type_code
```

LIKE and Wildcards

- Wildcards are ‘%’ and ‘_’
 - ‘%’ = any number of characters
 - ‘_’ = exactly one character
- Used with keyword LIKE
- Select information on all biomolecules with the word “kinase” in one of their names
 - ```
SELECT bm.Bio_mol_id, Bio_mol_name, Species
FROM Bio_molecule bm,
 Bio_molecule_name bmn
WHERE bm.Bio_mol_id = bmn.Bio_mol_id
AND Bio_mol_name LIKE '%kinase%'
```

Contents of strings are case-sensitive

---

# ORDER BY

- ORDER BY returns rows in order
- List the names assigned to Biomolecule #478 in alphabetical order:
  - SELECT Bio\_mol\_name  
FROM Bio\_molecule\_name  
WHERE Bio\_mol\_id = 478  
ORDER BY Bio\_mol\_name ASC
- ASC or DESC

---

# Aggregate Functions

## ■ COUNT

- Count number of sequences from RefSeq DB

- `SELECT COUNT (*)`

- `FROM Bio_molecule_sequence`

- `WHERE Source_database = 'RefSeq'`

## ■ GROUP BY

- Count number of sequences from each DB

- `SELECT Source_database, COUNT (*)`

- `FROM Bio_molecule_sequence`

- `GROUP BY Source_database`

---

# Aggregate Functions

- MAX and MIN
  - `SELECT MAX(Date_inserted)`  
`FROM Bio_molecule_sequence`
  - Can be used on numeric and date fields
- SUM
- AVG

---

# String Functions

- DBMS specific implementations
- Usually have at least:
  - Substrings
  - Length

# Subqueries

- Can nest SQL statements:
  - Select all names for human proteins:

```
SELECT Bio_mol_name
FROM Bio_molecule_name
WHERE Bio_mol_id IN (
 SELECT Bio_mol_id
 FROM Bio_molecule
 WHERE Species = 'Homo sapiens'
 AND Bio_mol_type_code = 'P'
)
```



# Subqueries

## ■ EXISTS

- Another way to express subsets

```
SELECT Bio_mol_name
FROM Bio_molecule_name bmn
WHERE EXISTS (
 SELECT *
 FROM Bio_molecule bm
 WHERE Species = 'Homo sapiens'
 AND Bio_mol_type_code = 'P'
 AND bm.Bio_mol_id = bmn.Bio_mol_id
)
```

---

# Subqueries

- Can also use NOT IN and NOT EXISTS
- Choice between using JOIN, IN, or EXISTS is a performance tuning issue
- Optimizer will usually “convert” for you, but sometimes it pays to optimize, or “tune” the query yourself
- For more details:
  - SQL Performance Tuning, by P. Gulutzan and T. Pelzer

# Subqueries

- Can join back to the same table
- Show the primary name for all biomolecules for which there are no other names:

```
SELECT Bio_mol_name
FROM Bio_molecule_name bmn1
WHERE Primary = 'Y'
AND NOT EXISTS (
 SELECT *
 FROM Bio_molecule_name bmn2
 WHERE Primary <> 'Y'
 AND bmn2.Bio_mol_id = bmn1.Bio_mol_id
)
```

---

# CLOBs

- **CLOB** = **C**haracter **L**arge **O**bject
- Implementation is very DBMS specific
- Usually do not have access to many functions
  - No substring or length functions
  - Can't use in WHERE clause
  - Can even be difficult to load in and select out

---

# Relational Operators: Union

- Merges two relations
- Result is a set that contains all rows in relation 1 and all rows in relation 2
- Useful for combining subsets
- `SELECT *`  
`FROM Protein_Sequence`  
`UNION`  
`SELECT *`  
`FROM Nucleotide_Sequence`

# Relational Operators: Union

Protein\_Sequence

| Biopol_ID | Sequence        |
|-----------|-----------------|
| Protein 1 | ALVCYFMIEGD.... |
| Protein 2 | KLMIKAGGKLV.... |

Nucleotide\_Sequence

| Biopol_ID | Sequence        |
|-----------|-----------------|
| DNA 1     | ATTGCATTAGC.... |
| DNA 2     | GCGGTATGCC....  |



Get a list of all sequences

| Biopol_ID | Sequence        |
|-----------|-----------------|
| Protein 1 | ALVCYFMIEGD.... |
| Protein 2 | KLMIKAGGKLV.... |
| DNA 1     | ATTGCATTAGC.... |
| DNA 2     | GCGGTATGCC....  |

More likely to be used in combination with projection

# Relational Operators: Union

**Protein\_Sequence**

| Biopol_ID | Sequence        | pI  |
|-----------|-----------------|-----|
| Protein 1 | ALVCYFMIEGD.... | 4.5 |
| Protein 2 | KLMIKAGGKLV.... | 7.3 |

**Nucleotide\_Sequence**

| Biopol_ID | Sequence        | Promoter |
|-----------|-----------------|----------|
| DNA 1     | ATTGCATTAGC.... | TATA     |
| DNA 2     | GCGGTATGCC....  | TAAA     |



Get a list of all sequences

| Biopol_ID | Sequence        |
|-----------|-----------------|
| Protein 1 | ALVCYFMIEGD.... |
| Protein 2 | KLMIKAGGKLV.... |
| DNA 1     | ATTGCATTAGC.... |
| DNA 2     | GCGGTATGCC....  |

```
SELECT Biopol_ID, Sequence
FROM Protein_Sequence
UNION
SELECT Biopol_ID, Sequence
FROM Nucleotide_Sequence
```

---

# Relational Operators: Intersection

- Returns rows common to both relations
- Used to identify overlapping subsets

- `SELECT *`

`FROM Protein_Stock`

`INTERSECT`

`SELECT *`

`FROM Plasmid_Stock`



# Relational Operators: Intersection

Protein\_Stock

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 1  | Box 2          |
| Protein 2  | Box 5          |

Plasmid\_Stock

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 1  | Box 2          |
| Protein 3  | Box 3          |



Find proteins for which  
lab has both plasmid and  
protein prep in stock

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 1  | Box 2          |

Again, more likely to be  
used in combination with  
projection

---

# Relational Operators: Difference

- Subtraction: returns rows found in relation 1 but not in relation 2
- Used to identify non-overlapping subsets
- `SELECT *`  
`FROM Protein_Stock`  
`EXCEPT`  
`SELECT *`  
`FROM Plasmid_Stock`

# Relational Operators: Difference

Protein\_Stock

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 1  | Box 2          |
| Protein 2  | Box 5          |

Plasmid\_Stock

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 1  | Box 2          |
| Protein 3  | Box 3          |



Find proteins for which  
lab has plasmid but no  
protein prep in stock  
(time to make more!)

| Protein_ID | Stock_location |
|------------|----------------|
| Protein 3  | Box 3          |

Again, more likely to be  
used in combination with  
projection

---

# Relational Operators: Division


- Returns column values from one relation for which there are matching column values for every row in another relation
- A fancy sort of intersection:
  - Finds the subset of relation 1 that “meets criteria” established by relation 2
- No simple SQL implementation. See:  
<http://www.developersdex.com/gurus/articles/113.asp>

# Relational Operators: Division

**Available\_Protein**

| Protein_ID | Protein_Name | Species_Sci_Name | Species_Common_Name |
|------------|--------------|------------------|---------------------|
| Protein 1  | calmodulin   | Homo sapiens     | human               |
| Protein 2  | integrin IV  | Bos taurus       | cow                 |
| Protein 1  | calmodulin   | Mus musculus     | house mouse         |
| Protein 3  | ICE          | Homo sapiens     | human               |

Find proteins that are available in all species studied in the lab



| Protein_ID | Protein_Name |
|------------|--------------|
| Protein 1  | calmodulin   |

**Lab\_Species**

| Species_Sci_Name | Species_Common_Name |
|------------------|---------------------|
| Homo sapiens     | human               |
| Mus musculus     | house mouse         |

---

# Reading and Homework

- Recommended reading for this week's class:  
Chapters 1-3 of Fleming and von Halle
- Homework handout
  
- Fleming and von Halle:
  - Recommended reading for next week's class: Chapter 4
  - Optional reading: Chapters 5-7
- The Trip-Packing Dilemma article (on website)
- Optional: Writing Quality Requirements article (on website)