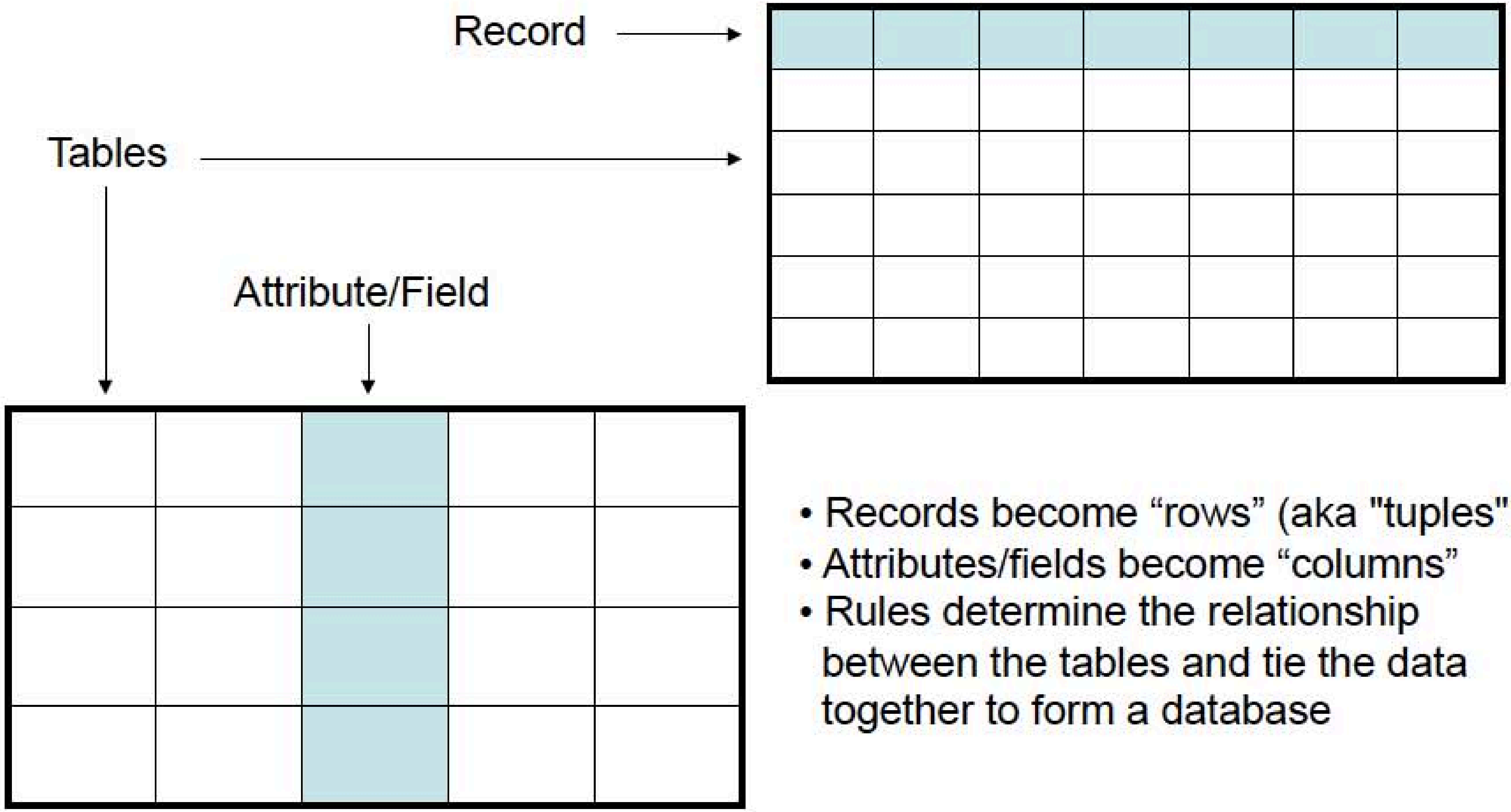
What is a relational database?

- Originally developed by E.F. Codd in 1970
- Organizes data into tables where each item is a row and the attributes of the item are in columns.
- Different from “flat file” databases because you can define “relationships” between items in different tables.
- The data within tables in the same database should all be related somehow.

Parts of a database



Record

Tables

Attribute/Field

- Records become “rows” (aka "tuples")
- Attributes/fields become “columns”
- Rules determine the relationship between the tables and tie the data together to form a database

Creating a database

- What information are we trying to store?
- How do we describe the information?
- Phone Book/Contact entries
  - Name
  - Address
  - Company
  - Phone Number
  - URL/Web Page
  - Age
  - Height (in meters)
  - Birthday
  - When we added the entry

Data Types

- Binary
  - Database specific binary objects
  - Pictures, digital signatures, etc.
- Boolean
  - True/False values
- Character
  - Fixed width or variable size
- Numeric
  - Integer, Real (floating decimal point), Currency
- Temporal
  - Time, Date, Timestamp

Phone Book/Contact Record

Name	Character
Address	Character
Company	Character
Phone Number	Character
URL/Web Page	Character
Age	Integer
Height	Real (float)
Birthday	Date
When we added the entry	Timestamp

“Normal Forms”

Summarized from Barry Wise’s article on Database Normalization

http://www.phpbuilder.com/columns/barry20000731.php3?page=1

What are the “normal forms”?

- E. F. Codd in 1972 wrote a paper on “Further Normalization of the Data Base Relational Model”
- Normal forms reduce the amount of redundancy and inconsistent dependency within databases.
- Codd proposed three normal forms and through the years two more have been added.

The Zero Form

- No rules have been applied
- Where most people start (and stop)
- No room for growth
- Usually wastes space

Contacts

Name	Company	Address	Phone1	Phone2	Phone3	ZipCode
Joe	ABC	123	5532	2234	3211	12345
Jane	XYZ	456	3421			14454
Chris	PDQ	789	2341	6655		14423

First Normal Form

- Eliminate repeating columns in each table
- Create a separate table for each set of related data
- Identify each set of related data with a primary key

Contacts

Id	Name	Company	Address	Phone	ZipCode
1	Joe	ABC	123	5532	12345
1	Joe	ABC	123	2234	12345
1	Joe	ABC	123	3211	12345
2	Jane	XYZ	456	3421	14454
3	Chris	PDQ	789	2341	14423
3	Chris	PDQ	789	6655	14423

Benefits: Now we can have infinite phone numbers or company addresses for each contact.

Drawback: Now we have to type in everything over and over again. This leads to inconsistency, redundancy and wasting space. Thus, the second normal form…

Keys

A key is a minimal set of attributes that uniquely identifies 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

Second Normal Form

- Create separate tables for sets of values that apply to multiple records
- Each table has its own primary key that uniquely identifies each record in it.
- Relate these tables with a “foreign key”.

People

Id (PK)	Name	Company	Address	Zip
1	Joe	ABC	123	12345
2	Jane	XYZ	456	14454
3	Chris	PDQ	789	14423

PhoneNumbers

PhoneID (PK)	Id (FK)	Phone
1	1	5532
2	1	2234
3	1	3211
4	2	3421
5	3	2341
6	3	6655

Third Normal Form

- Eliminate fields that do not depend on the primary key.

People

Id (PK)	Name	AddressID (FK)
1	Joe	1
2	Jane	2
3	Chris	3

PhoneNumbers

PhoneID (PK)	Id (FK)	Phone
1	1	5532
2	1	2234
3	1	3211
4	2	3421
5	3	2341
6	3	6655

Address

AddressID (PK)	Company	Address	Zip
1	ABC	123	12345
2	XYZ	456	14454
3	PDQ	789	14423

Is this enough? Codd thought so…
What about "many to many"?

Kinds of Relationships

- “One to One”
  - One row of a table matches exactly to another
    - One person, one id number, one address
- “One to Many”
  - One row of a table matches many of another
    - One person, many phone numbers
- “Many to Many”
  - One row may match many of another or many rows match one row of another

Fourth Normal Form

- In a “many to many” relationship, independent entities cannot be stored in the same table.

PhoneNumbers

PhoneID (PK)	Phone
1	5532
2	2234
3	3211
4	3421
5	2341
6	6655

People

Id (PK)	Name	AddressID (FK)
1	Joe	1
2	Jane	2
3	Chris	3

PhoneRelations

PhoneRelID (PK)	Id (FK)	PhoneID (FK)
1	1	1
2	1	2
3	1	3
4	2	4
5	3	5
6	3	6

Address

AddressID (PK)	Company	Address	Zip
1	ABC	123	12345
2	XYZ	456	14454
3	PDQ	789	14423

Fifth Normal Form

- The “very esoteric” one that is probably not required to get the most out of your database.
- “The original table must be reconstructed from the tables into which it has been broken down.”
- The rule ensures that you have not created any extraneous columns and all the tables are only as large as they need to be.

Why normalize?

- Increases the integrity of the data
- Reduces redundancy
- Improves efficiency
- Although normalization can be hard, it is worth it in the long run.

What do I need to remember?

- Keep normalization in mind.
- Don’t replicate data in a table.
- If you break the rules, know why you are breaking the rules and do it for a good reason.

More about SELECT

“Normal Forms” and SELECT

- Good database design using the normal forms requires data to be separated into different tables
- SELECT allows us to join the data back together
- We can use “views” to create virtual tables

The Normal Forms

- First Form
  - Eliminate replicated data in tables
  - Create separate tables for each set of related data
  - Identify each set of related data with a primary key
- Second Form
  - Create separate tables for sets of values that apply to multiple records
  - Relate the tables with a foreign key
- Third Form
  - Eliminate fields that do not depend on the primary key
- Fourth Form
  - In many-to-many relationships, independent entities cannot be stored in the same table

Joining together tables

```
SELECT name,phone,zip
FROM people, phonenumbers, address
WHERE
     people.addressid=address.addressid
     AND people.id=phonenumbers.id;
```

People

Id	Name	AddressID
1	Joe	1
2	Jane	2
3	Chris	3

PhoneNumbers

PhoneID	Id	Phone
1	1	5532
2	1	2234
3	1	3211
4	2	3421
5	3	2341
6	3	6655

Address

AddressID	Company	Address	Zip
1	ABC	123	12345
2	XYZ	456	14454
3	PDQ	789	14423

Different types of JOINs

- “Inner Join”
  - Unmatched rows in either table aren’t printed
- “Left Outer Join”
  - All records from the “left” side are printed
- “Right Outer Join”
  - All records from the “right” side are printed
- “Full Outer Join”
  - All records are printed
- Multiple Table Join
  - Join records from multiple tables

General form of SELECT/JOIN

Syntax

SELECT *columns,…*
    FROM *left_table*
    join_type JOIN *right_table* ON *condition* ;

Example

```
SELECT name, phone
   FROM people
   JOIN phonenumbers ON people.id=phonenumbers.id;
```

Other versions

```
SELECT name, phone FROM people
  LEFT JOIN phonenumbers ON
      people.id=phonenumbers.id;

SELECT name, phone FROM people
  RIGHT JOIN phonenumbers ON
      people.id=phonenumbers.id;

SELECT name, phone FROM people
  FULL JOIN phonenumbers ON
      people.id=phonenumbers.id;
```

“Theta style” vs. ANSI

- Theta Style (used in most SQL books)

```
SELECT name, phone, zip
   FROM people, phonenumbers, address
   WHERE people.addressid=address.addressid
      AND people.id=phonenumbers.id;
```

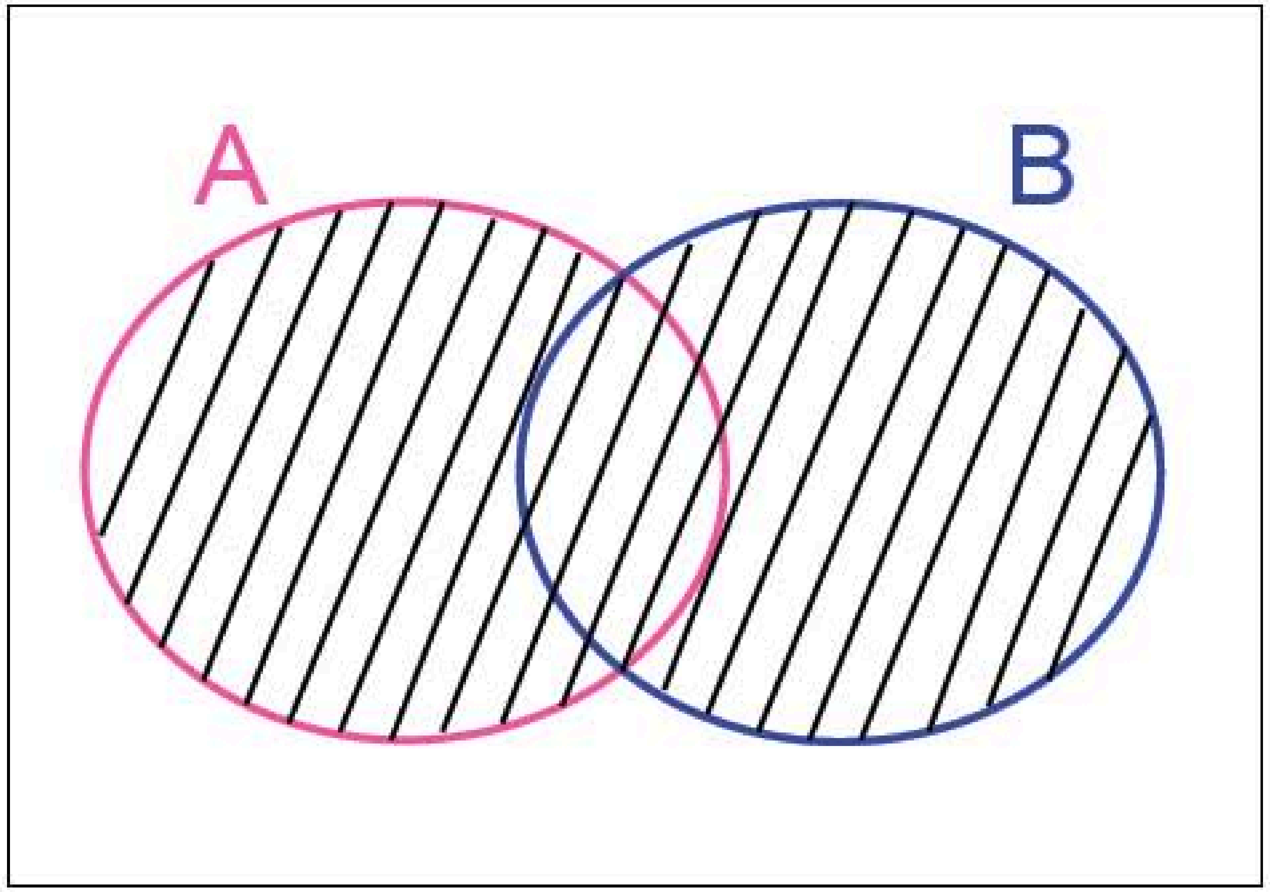
- ANSI Style uses JOIN

```
SELECT name, phone, zip
   FROM people
   JOIN phonenumbers ON people.id=phonenumbers.id
   JOIN address      ON people.addressid=address.addressid;
```

Union, Intersection and Difference of Tables

Union of Tables

The ***union*** of A and B (A∪B)



A table containing all the rows from **A** and **B**.

Union of Tables

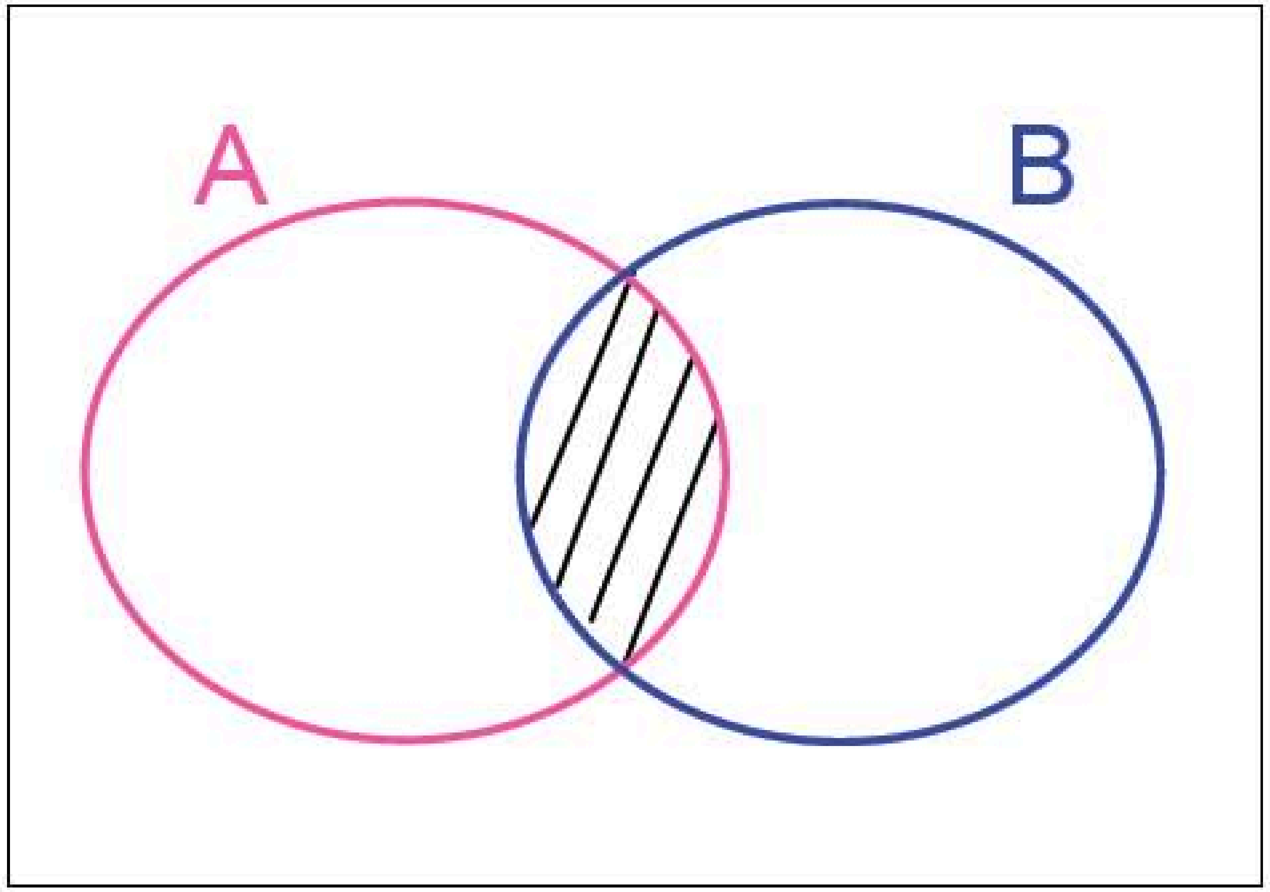
```
SELECT ...... FROM ...... WHERE ......
UNION
SELECT ...... FROM ...... WHERE ......
```

Example: make a list of all people who either play bridge OR chess (UNION)

```
SELECT * FROM bridge
  UNION
  SELECT * FROM chess;
```

Intersection of Tables

The ***intersection*** of A and B (A∩B)



A table containing only rows that appear in both **A** and **B**.

Intersection of Tables

```
SELECT ...... FROM table1
WHERE col IN  ( SELECT col FROM table2 ) ;
```

Example: make a list of all people who play bridge AND chess (INTERSECTION)

```
SELECT name FROM bridge
   WHERE id IN
   ( SELECT id FROM chess );
```

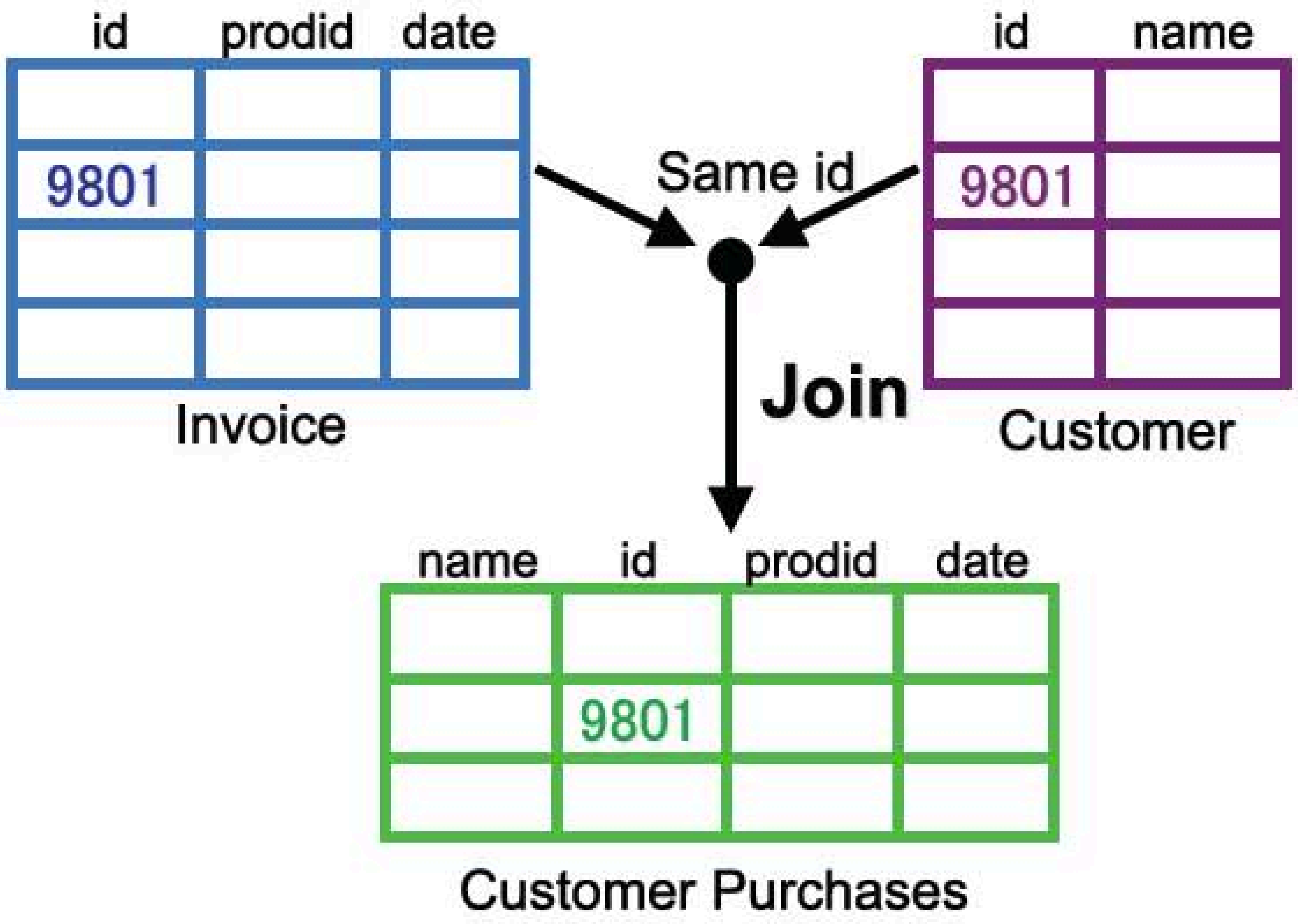
Intersection: Natural Join

A **Natural Join** is a join operation that joins two tables by their common column. This operation is similar to the setting relation of two tables.

```
SELECT a.comcol, a.col1, b.col2, expr1, expr2
FROM table1 a, table2 b
WHERE a.comcol = b.comcol
[ GROUP BY … ] ;
```

Natural Join Example

Make a list of customers and the products they purchased



Query:

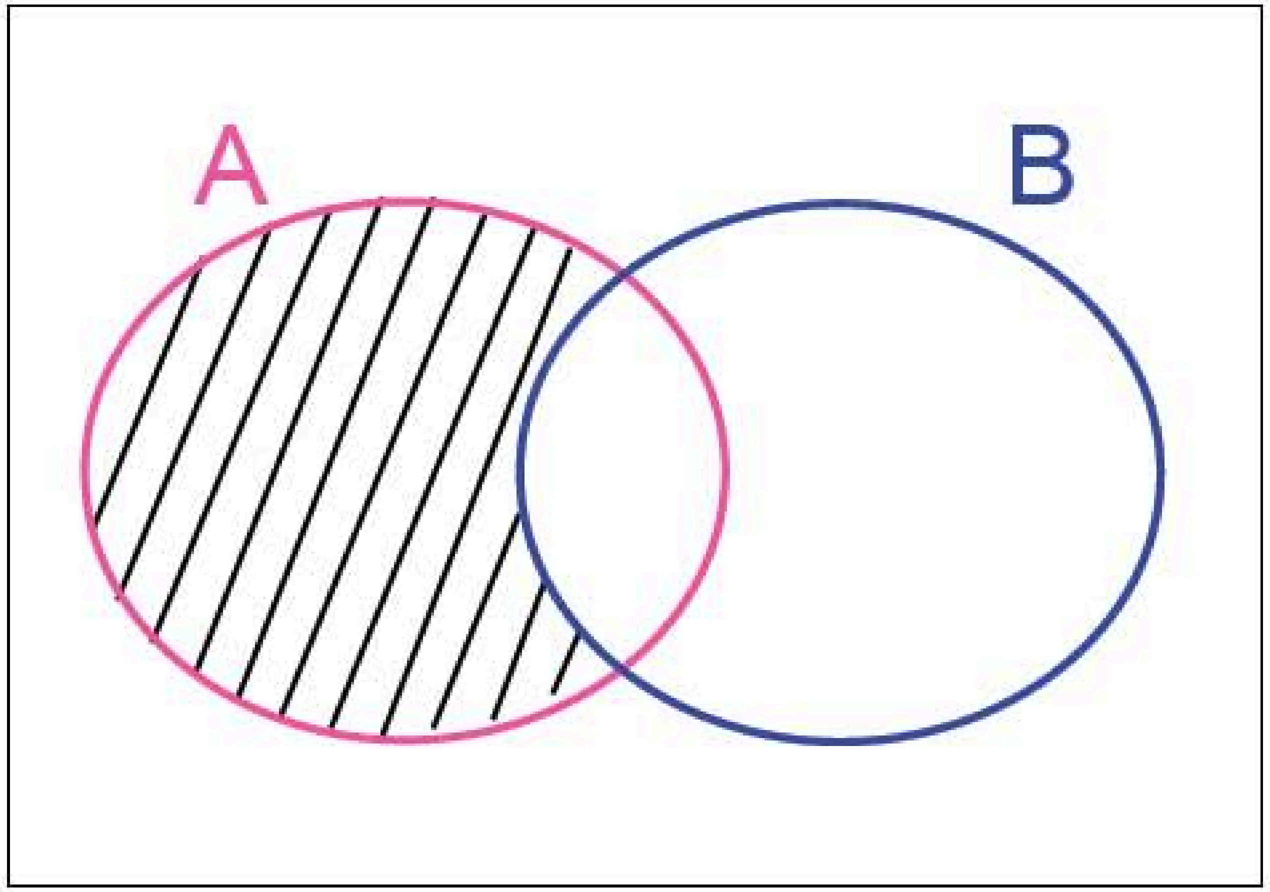
```
SELECT name, a.id, prodid, date
FROM Customer a, Invoice b
WHERE a.id = b.id;
```

Note:

✓disambiguation of id column
✓implicit use of AS

Difference of Tables

The ***difference*** of A and B (A–B)



A table containing rows that appear in **A** but not in **B**.

Difference of Tables

```
SELECT ...... FROM table1 ;
WHERE col NOT IN  ( SELECT col FROM table2 )
```

Example: make a list of all people who ONLY play bridge AND NOT chess (DIFFERENCE)

```
SELECT * FROM bridge
   WHERE id NOT IN
   ( SELECT id FROM chess );
```

Input and Output for whole datasets

To load an entire flat file to a DB table all at once:

```
LOAD DATA INFILE
    '/some/path/accessible/to/mysql._Server/file.txt'
INTO TABLE mytable [ options ];
```

To save the results of a query as a flat file:

```
SELECT * FROM contact
INTO OUTFILE '/home/dir1/dir2/outfilename.txt';
```

Views

You can use “CREATE VIEW” to create a virtual table from a SELECT statement:

```
CREATE VIEW contactview AS
   (SELECT name, phone, zip
    FROM people, phonenumbers, address
    WHERE people.id=phonenumbers.id
       AND   people.addressid=address.addressid);
```

Creating Indexes

Indexes help speed up searches on database tables. The syntax is:

```
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)
```

- ➢Indexes that break the table into several smaller parts are very useful for the database engine.
- ➢If you query on a particular column very often, you should add an index on that column (i.e. the column is used in a WHERE clause).
- ➢If you query multiple columns together quite often, you can create an index on multiple columns.
- ➢If you notice that your queries seem to be taking an unusually long time (this can happen for complex queries), try adding some indexes on your tables.

Why not create indexes on everything?

- *Take up a lot of disk space*
- *Slow down data insertion and updates*
- *May not provide any gain in performance*

Accessing relational databases

There are several ways to connect to RDMSs like MySQL. We'll introduce you to all three in the class.

- Through a command-line interface.
- Through a graphical user interface (GUI).
- Programmatically, through an applications programming interface (API).
  - APIs for many RDMSs exist in most common programming languages to manage these interactions.
  - These APIs typically consist of packages with RDMS-specific drivers, and employ a special syntax beyond simple SQL to handle the database connections and information exchange.
  - We'll learn how to access MySQL from R and Perl in the next few sessions.