Review

Consider a database of warships. Each warship has the following information associated with it:

(a) Its name.

(b) Its displacement (weight), in tons.

(c) Its type, e.g., battleship, destroyer.

In addition, there are the following special kinds of ships that have some other information:

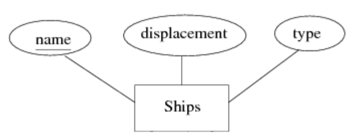
(a) *Gunships* are ships that carry large guns, such as battterships or cruisers. For these ships, we wish to record the number and bore of the main guns.

(b) *Carriers* hold aircraft. For these, we wish to record the length of the flight deck and the set of air groups assigned to them.

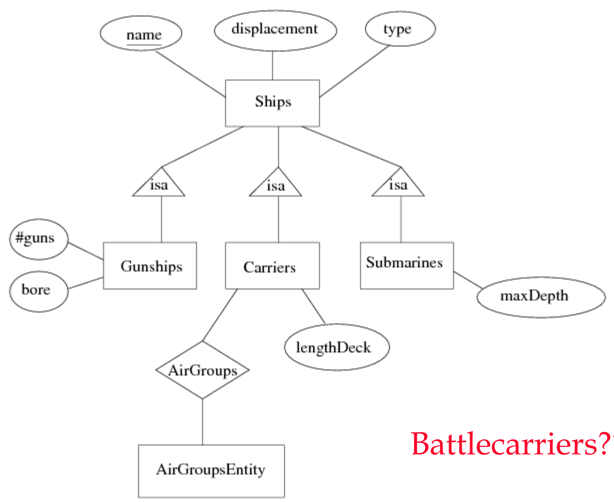
(c) *Submarines* which can travel under water. For these, we wish to record their maximum safe depth. You may assume no gunship or carrier is a submarine.

(d) *Battlecarriers* are both gunships and carriers, and have all the information associated with either.

Design 1

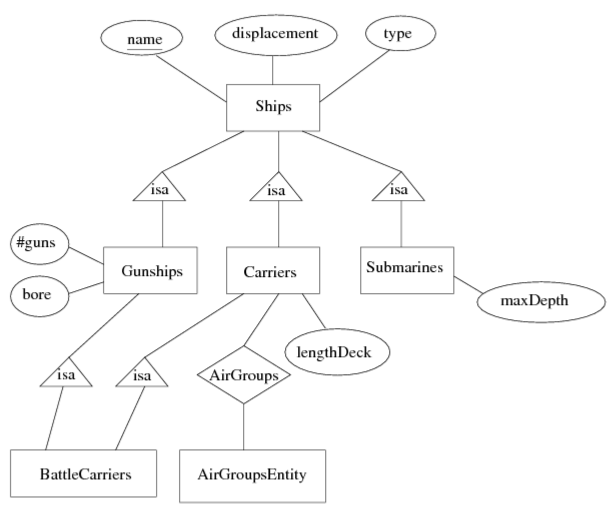


Design 1



Battlecarriers??

Design 2



Schema Refinement, Normalization and Query Languages

It is normal to design and then refine your design.

Evils of Redundancy

Hourly_Emps (ssn, name, lot, rating, hrly_wages, hrs_worked)
Also used SNLRWH to refer to the table

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

- *Redundant storage*
- *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
- *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
- *Deletion anomaly*: What if we delete all employees with rating 5?

A BAD Relational Schema

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

An Improved Schema

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

R	W
8	10
5	7

Refinements

- Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.
- Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).
- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?

Functional Dependencies (FDs)

- A *functional dependency* X → Y (X determines Y) holds over relation R if, for *every* allowable instance *r* of R:
  - given two tuples in *r*, if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)
- K is a *candidate key* for relation R if:
  1. K determines *all* attributes of R.
  2. For no proper subset of K is (1) true.
     - If K satisfies only (1), then K is a superkey.
- Primary key

Example

- Consider relation Hourly_Emps:
  - Hourly_Emps (*ssn*, *name*, *lot*, *rating*, *hrly_wages*, *hrs_worked*)
- FDs S → SNLRWH
  - *ssn* is the key
- FDs give more detail than the mere assertion of a key
  - *rating* determines *hrly_wages*
- R → W

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	40
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	30
612-67-4134	Madayan	35	8	10	40

Who Determines Keys/FDs?

- An FD is a statement about *all* allowable relations.
  - Must be identified based on semantics of application.
  - Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we *cannot* tell if *f* holds over R!
- We can define a relation schema with a single key K.
  - Then the only FD asserted are $K \to A$ for every attribute A.
- Or, we can assert some FDs and deduce one or more keys or other FDs.

Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - $ssn \to did$, $did \to lot$ implies $ssn \to lot$
- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - F^+ = *closure of F* is the set of all FDs that are implied by *F*.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - *Reflexivity*: If $Y \subseteq X$, then $X \to Y$
  - *Augmentation*: If $X \to Y$, then $XZ \to YZ$ for any Z
  - *Transitivity*: If $X \to Y$ and $Y \to Z$, then $X \to Z$
- These are *sound* and *complete* inference rules for FDs!

Reasoning About FDs (Cont.)

- Example: Contracts(*cid,sid,jid,did,pid,qty,value*)
  - C is the key: $C \rightarrow CSJDPQV$
  - Project purchases each part using single contract: $JP \rightarrow C$
  - Dept purchases at most one part from a supplier: $SD \rightarrow P$
- $JP \rightarrow C$, $C \rightarrow CSJDPQV$ imply $JP \rightarrow CSJDPQV$
- $SD \rightarrow P$ implies $SDJ \rightarrow JP$
- $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ imply $SDJ \rightarrow CSJDPQV$
- So, JP and SDJ are candidate keys!

Reasoning About FDs (Cont.)

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure of a set of FDs F. An efficient check:
  - Compute *attribute closure* of X (denoted X^+) wrt F:
    - Set of all attributes A such that $X \rightarrow A$ is in F^+
    - There is a linear time algorithm to compute this.
  - Check if Y is in X^+
- Does $F = \{A \rightarrow B,\ B \rightarrow C,\ C\,D \rightarrow E\}$ imply $A \rightarrow E$?
  - i.e, is $A \rightarrow E$ in the closure F^+? Equivalently, is E in A^+?

Algorithm to Compute Attribute Closure

- Define Y^+ = closure of Y.
- Basis: $Y^+ = Y$
- Induction: If $X \subseteq Y^+$, and $X \to A$ is a given FD, then add A to Y^+
- End when Y^+ cannot be changed. Then Y functionally determines all members of Y^+, and no other attributes.

- $A \to B$, $BC \to D$
  - $A^+ = AB$
  - $C^+ = C$
  - $(AC)^+ = ABCD$
- Thus, AC is a key.

Finding All Implied FDs

- Motivation: Suppose we have a relation ABCD with some FDs *F*. If we decide to decompose ABCD into ABC and AD, what are the FDs for ABC, AD?
- Example: $F = AB \to C, C \to D, D \to A$. It looks like just $AB \to C$ holds in ABC, but in fact $C \to A$ follows from *F* and applies to relation ABC.
- Problem is exponential in worst case.
- Algorithm to find F^+:
  - For each set of attributes X of R, compute X^+.

A	B	C	D
1	1	2	3
1	2	2	3
2	2	2	4

A	B	C
1	1	2
1	2	2
2	2	2

A	D
1	3
2	4

Example

- F = AB → C, C → D, D → A. What FDs follow?
  - A^+ = A; B^+ = B (nothing)
  - C^+ = ACD (add C → A)
  - D^+ = AD (nothing new)
  - $(AB)^+$ = ABCD (add AB → D; skip all supersets of AB).
  - $(BC)^+$ = ABCD (nothing new; skip all supersets of BC).
  - $(BD)^+$ = ABCD (add BD → C; skip all supersets of BD).
  - $(AC)^+$ = ACD; $(AD)^+$ = AD; $(CD)^+$ = ACD (nothing new).
  - $(ACD)^+$ = ACD (nothing new).
  - All other sets contain AB, BC, or BD, so skip.
  - Thus, the only interesting FDs that follow from F are:
    - C → A, AB → D, BD → C.

Projection of set of FDs

- If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs U → V in F^+ (*closure of F*) such that U, V are in X.
- Using the same example,
  - R1(ABC): AB → C, C → A
  - R2(AD): D → A

A BAD Relational Schema

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

An Improved Schema

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

R	W
8	10
5	7

What's a Good Design?

- Three properties:
  - No anomalies.
  - Can reconstruct all original information.
  - Ability to check all FDs within a single relation.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given A →B: Several tuples could have the same A value, and if so, they'll all have the same B value!

Decomposition of a Relation Scheme

- Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs $S \rightarrow SNLRWH$ and $R \rightarrow W$
  - W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
    - i.e., we decompose SNLRWH into SNLRH and RW
- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

- There are three potential problems to consider:
  1. Some queries become more expensive.
     - e.g., How much did sailor Joe earn? (salary = W*H)
  2. Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
     - Fortunately, not in the SNLRWH example.
  3. Checking some dependencies may require joining the instances of the decomposed relations.
     - Fortunately, not in the SNLRWH example.
- *Tradeoff*: Must consider these issues vs. redundancy.

Lossless Join Decompositions

- Decomposition of R into X and Y is *lossless-join* w.r.t. a set of FDs F if, for every instance *r* that satisfies F, "reassembling" X and Y will give R and nothing else.
- It is always true that reassembling X and Y gives exactly R or a superset of R.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem (2).)*

More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$
- In particular, the decomposition of R into UV and R - V is lossless-join if $U \rightarrow V$ holds over R.

A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, $JP \rightarrow C$ and $SD \rightarrow P$.
  - (BCNF) Decomposition: CSJDQV and SDP
  - Problem: Checking $JP \rightarrow C$ requires a join!
- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3).)*

Dependency Preserving Decompositions (Cont.)

- Decomposition of R into X and Y is *dependency preserving* if $(F_X \text{ union } F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.
- Important to consider F^+, not F, in this definition:
  - ABC, $A \to B$, $B \to C$, $C \to A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \to A$ preserved?????
- Dependency preserving does not imply lossless join:
  - ABC, $A \to B$, decomposed into AB and BC.
- And vice-versa! (Example?)

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.

Boyce-Codd Normal Form (BCNF)

- Reln R with FDs *F* is in BCNF if, for all X →A in F^+
  - A ∈ X (called a *trivial* FD), or
  - X contains a key for R.
- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
- Why?
  - Guarantees no redundancy due to FDs.
  - Guarantees no insert/update/delete anomalies.
  - Guarantees no loss of information.
- But …
  - May destroy the ability to check FDs within a single relation

Example

- Consider relation Beers(*name*, *manf*, *manfAddr*).
  - FDs = *name* → *manf*, *manf* → *manfAddr*
  - Only key is *name*.
    - *manf* → *manfAddr* violates BCNF with a left side unrelated to any key.
      - Redundancy (every manf has the same manfAddr)
      - Update anomalies (if manf moves, all manfAddr in ALL tuples)
      - Deletion anomalies (deleting all beers produced by a particular manf will lose info on manf and manfAddr)
- Not in BCNF.

Decomposition into BCNF

- Consider relation R with FDs F. If X → Y violates BCNF,
  - Expand left side to include X^+.
  - Decompose R into (R - X^+) U X and X^+.
  - Find the FDs for the decomposed relations.
- Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.
- In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

Example

- R(A, C, B, D, E)
- F = A → B, A → E, C → D
- Since AC is a key, not in BCNF.
- Pick A → B for decomposition.
- Expand left side: A → B E
- Decomposed relations: R1(A,B,E) and R2(A,C,D).
- Projected FDs (skipping a lot of work …)
  - R1: A → B, A → E
  - R2: C → D
- BCNF violations?
  - For R1, A is key and all left sides are superkeys.
  - For R2, AC is key, and C → D violates BCNF.
- Decompose R2
  - R3(C,D)
  - R4(A,C)
- Resulting relations are all in BCNF.
  - R1(A,B,E)
  - R3(C,D)
  - R4(A,C)

BCNF and Dependency Preservation

- The example decomposition is dependency preserving!
- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS $\rightarrow$ Z, Z $\rightarrow$ C
  - Can't decompose while preserving 1st FD; not in BCNF.

Summary of Schema Refinement

- If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
  - Must consider whether all FDs are preserved. If a lossless-join, dependency preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), should consider decomposition into 3NF.
  - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.

SQL – Structured Query Language

Relational Database

- Example Data

Census **Table** (**Relation**/File)

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

← **Row** (**Tuple**/Record)

↑ **Column** (**Attribute**/Field/Domain)

- Database Schema

Census

State	Abbr	Year	Population	Cars

Student

Name	Major	Year	Home

SQL - Relational Calculus Query Language

Census

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

- Structured Query Language (SQL)
  - Most common/standard language (IBM, Oracle, Sybase, Informix, Microsoft)

Q1. Population in MASSACHUSETTS (all available years)

select year, population ← Domain name(s)

from census ← Relation name(s)

where state = "MASSACHUSETTS" ← Tuple restriction(s)

Q2. Names of states with more than 9 million people in 2000.

Select

SQL - Relational Calculus Query Language

Census

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

- Structured Query Language (SQL)
  - Most common/standard language (IBM, Oracle, Sybase, Informix, Microsoft)

Q1. Population in MASSACHUSETTS (all available years)

select year, population ← Domain name(s)

from census ← Relation name(s)

where state = "MASSACHUSETTS" ← Tuple restriction(s)

Q2. Names of states with more than 9 million people in 2000.

```
Select state, population
      FROM census
      WHERE population>9000 and year=2000;
```

SQL Clauses

- **Where** clause
  - Conditions:

<	Less than	**<=**	Less than or equal
>	Greater than	**>=**	Greater than or eqal
=	Equal to	**!=** or **<>**	Not equal

  - Compound conditions:

not	logical not		
and	logical and	**or**	logical or

- **Order by** clause
  - Sort in either ascending (default) or descending (**desc**) order
  - Can use column name or number

Q3. Names of states with more than 9 million people in 2000 -- ordered from highest to lowest population.

Select . . .

SQL Clauses

- **Where** clause
  - Conditions:

<	Less than	**<=**	Less than or equal
>	Greater than	**>=**	Greater than or eqal
=	Equal to	**!=** or **<>**	Not equal

  - Compound conditions:

not	logical not		
and	logical and	**or**	logical or

- **Order by** clause
  - Sort in either ascending (default) or descending (**desc**) order
  - Can use column name or number

Q3. Names of states with more than 9 million people in 2000 -- ordered from highest to lowest population.

Select . . . Order by population desc

Calculations

- Calculations with SQL query to create "virtual" columns or within "where" clause.
  - Operations include:

    + Addition   **-** Subtraction
    ***** Multiplication   **/** Division

Q4. Which states have highest cars per capita in 2000?

calculation → **cars/population**; name of "virtual" column → **carspercapita**

select state, **cars/population** as **carspercapita**
from census
where year=2000
order by 2 desc

Note: "as" clause not needed.

Calculations

- Calculations with SQL query to create "virtual" columns or within "where" clause.
  - Operations include:

    + Addition   **-** Subtraction
    ***** Multiplication   **/** Division

Q4. Which states have highest cars per capita in 2000?

calculation → **cars/population**; name of "virtual" column → **carspercapita**

select state, **cars/population** as **carspercapita**
from census
where year=2000
order by 2 desc

You can get the correct answer, but the query is "incorrect". What is wrong?

Note: "as" clause not needed.

"Join" Between Relations

Q6. Names of all students from states with more than 9 million people in 2000.

Census

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

Student

Name	Major	Year	Home
GUPTA	ECE	1	WYOMING
MADNICK	EE	3	MASSACHUSETTS
TAN	CS	4	ALABAMA
ZHAO	CS	1	WYOMING

- SQL:

```
select  student.name
from    student, census
where   student.home = census.state
and     census.year = 2000
and     census.population > 9000
```


"Join" Between Relations

Q7. Names of all states where number of cars increased by over 5% between 1999 and 2000.

Census

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

Select
From
Where

"Join" Between Relations

Q7. Names of all states where number of cars increased by over 5% between 1999 and 2000.

Census

State	Abbr	Year	Population	Cars
ALABAMA	AL	1999	4370	3957
ALABAMA	AL	2000	4447	3960
⋮	⋮	⋮	⋮	⋮
WYOMING	WY	2000	494	586

```
Select select c00.state, c99.cars, c00.cars, c00.cars/c99.cars
From census c00, census c99
Where c00.year=2000 and c99.year=1999  and
        c99.state=c00.state and c00.cars/c99.cars>1.05;
```


Relational Algebra

- Relational algebra defines the theoretical way of manipulating table contents using the five basic relational functions: UNION, SELECT, PROJECT, DIFFERENCE, PRODUCT.
- JOIN, INTERSECT, and DIVIDE, etc. helpful, but derivable from five basics.
- Often underlying implementation of relational calculus

Basic Relational Database Operators

- ∪ - **UNION combines all rows from two tables. The two tables must be union compatible.**

Relation A UNION Relation B yields Relation C

Relation A [Supplier A]

P_CODE
123456
123457
123458

UNION

Relation B [Supplier B]

P_CODE
123457
345678
345679

yields

Relation C

P_CODE
123456
123457
123458
345678
345679

Basic Relational Database Operators

- – - **DIFFERENCE yields all rows in one table that are not found in the other table; i.e., it subtracts one table from the other. The tables must be union compatible.**

A [Classmates]

F_NAME
George
Jane
Elaine
Wilfred
Albert

DIFFERENCE

B [Unfriendly]

F_NAME
Jane
Larry
Albert

yields

C [Invite]

F_NAME
George
Elaine
Wilfred

Basic Relational Database Operators

- × - **PRODUCT produces a list of all possible pairs of rows from two tables.**

P_CODE	PRICE
AA	5.99
BB	22.75

PRODUCT

STORE	AISLE	SHELF
23	W	5
24	K	9
25	Z	6

yields

P_CODE	PRICE	STORE	AISLE	SHELF
AA	5.99	23	W	5
AA	5.99	24	K	9
AA	5.99	25	Z	6
BB	22.75	23	W	5
BB	22.75	24	K	9
BB	22.75	25	Z	6

Basic Relational Database Operators

- σ - **SELECT yields values for all attributes found in a table. It yields a horizontal subset of a table.**

Original table (X)

P_CODE	P_DESCRIPT	PRICE
213345	9v battery	1.92
311452	Power drill	34.99
254467	100W bulb	1.92

New table or list (T1)

T1 = SELECT X all → will yield

P_CODE	P_DESCRIPT	PRICE
213345	9v battery	1.92
311452	Power drill	34.99
254467	100W bulb	1.92

T1 = SELECT X where PRICE < 2.00

P_CODE	P_DESCRIPT	PRICE
213345	9v battery	1.92
254467	100W bulb	1.92

T1 = SELECT X where P_CODE = 311452

P_CODE	P_DESCRIPT	PRICE
311452	Power drill	34.99

Basic Relational Database Operators

- π - **PROJECT produces a list of all values for selected attributes. It yields a vertical subset of a table.**

Original table (X)

P_CODE	P_DESCRIPT	PRICE
213345	9v battery	1.92
311452	Power drill	34.99
254467	100W bulb	1.92

T1 = PROJECT X on PRICE yields

New table or list (T1)

PRICE
1.92
34.99

T1 = PROJECT X on P_DESCRIPT and PRICE

P_DESCRIPT	PRICE
9v battery	1.92
Power drill	34.99
100W bulb	1.92

T1 = PROJECT X on P_CODE and PRICE

P_CODE	PRICE
213345	1.92
311452	34.99
254467	1.92

Additional Relational Database Operators

- ∩ - **INTERSECT produces a listing that contains only the rows that appear in both tables. The two tables must be union compatible.**

A

F_NAME
George
Jane
Elaine
Wilfred
Fred

[Classmates]

INTERSECT

B

F_NAME
Jane
William
Jorge
Dennis

[Friends]

yields

C

F_NAME
Jane
Fred

[Invite]

- How to accomplish INTERSECT with basic operators?

Additional Relational Database Operators

- $\bowtie$ - **JOIN allows us to combine information from two or more tables, allowing the use of independent tables linked by common attributes.**

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	**231**
1321242	Rodriguez	37134	**125**
1657399	Vanloo	32145	**231**
1312243	Rakowski	34129	**167**
1542311	Smithson	37134	**421**
1217782	Adares	32145	**125**

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124
333	9041234445

JOIN Relational Database Operators

- **Natural JOIN** links tables by selecting only the rows with common values in their common attribute(s). It is the result of a three-stage process.
  - A **PRODUCT** is performed on two tables.
  - **SELECT** is performed to yield only the rows for which the common attribute values match.
  - A **PROJECT** is performed to yield a single copy of each attribute, thereby eliminating duplicate column.

JOIN Example

Table name: CUSTOMER

CUS_CODE	CUS_LNAME	CUS_ZIP	AGENT_CODE
1132445	Walker	32145	**231**
1321242	Rodriguez	37134	**125**
1657399	Vanloo	32145	**231**

Table name: AGENT

AGENT_CODE	AGENT_PHONE
125	6152439887
167	6153426778
231	6152431124

1. Product of both tables

1132445	Walker	32145	231	125	6152439887
1132445	Walker	32145	231	167	6153426778
1132445	Walker	32145	**231**	**231**	6152431124
1321242	Rodrguez	37134	**125**	**125**	6152439887
1321242	Rodrguez	37134	125	167	6153426778
1321242	Rodrguez	37134	125	231	6152431124
1657399	Vanloo	21145	231	125	6152439887
1657399	Vanloo	21145	231	167	6153426778
1657399	Vanloo	21145	**231**	**231**	6152431124

2. Select rows where agent_code match

1132445	Walker	32145	**231**	**231**	6152431124
1321242	Rodrguez	37134	**125**	**125**	6152439887
1657399	Vanloo	21145	**231**	**231**	6152431124

3. Project to eliminate 2nd agent_code

1132445	Walker	32145	**231**		6152431124
1321242	Rodrguez	37134	**125**		6152439887
1657399	Vanloo	21145	**231**		6152431124

Summary

- Relational Calculus:
  - Very user-friendly, easy-to-use
- Relational Algebra:
  - Sound theoretical basis
  - Often underlying implementation of calculus