COMP 330/543: Relational Databases 1

Luis Guzman

Sinan Kockara

Chris Jermaine

Rice University

What is a Database?

A collection of data

Plus, a set of programs for managing that data

Back in the Day...

The dominant data model was the network or navigational model (60's and 70's)

Data were a set of records with pointers between them

Much DB code was written in COBOL

Big problem was lack of physical data independence

- Code was written for specific storage model
- Want to change storage? Modify your code
- Want to index your data? Modify your code
- Led to very little flexibility
  - ▷ Your code locked you into a physical database design!

Some People Realized This Was a Problem

By 1970, EF Codd (IBM) was looking at the so-called relational model

- Landmark 1970 paper, "A relational model of data for large shared data banks"
- Led to the 1981 Turing Award
  - Highest honor a computer scientist receives
  - Analogous to a Nobel Prize

Idea: data stored in "relations"

- A relation is a table of tuples or records
- Attributes of a tuple have no sub-structure (are atomic)

No pointers!

Querying in the Relational Model

Querying is done via a "relational calculus"

Declarative

- You give a mathematical description of the tuples you want
- System figures out how to get those for you

Why is this good?

Querying in the Relational Model

Querying is done via a "relational calculus"

Declarative

- You give a mathematical description of the tuples you want
- System figures out how to get those for you

Why is this good?

- Data independence!
- Your code has no data access specs
- So can change physical org, no code re-writes

Relation Schema

All data are stored in tables, or relations

A relation schema consists of:

- A relation name (e.g., LIKES)
- A set of (attribute_name, attribute_type) pairs
  - Each pair is referred to as an "attribute"
  - Or sometimes as a "column"
- Usually denoted using LIKES (DRINKER string, BEER string)
- Or simply LIKES (DRINKER, BEER)

A Relation

A relation schema defines a set of sets

- Specifically, if $T_1, T_2, ..., T_n$ are the n attribute types
- Where each T_i is a set of possible values
  - ▷ Ex: string is all finite-length character strings
  - ▷ Ex: integer is all numbers from -2^{31} to $2^{31} - 1$
- Then a realization of the schema (aka a "relation") is a subset of
  - ▷ $T_1 \times T_2 \times ... \times T_n$
  - ▷ where $\times$ is the Cartesian product operator

Attribute Types Forming a Relation

T_1
0
1
...

X

T_2
A
B
...

X ... X

T_n
Red
Blue
...

=

T_1	T_2	...	T_n
0	A	...	Red
0	A	...	Blue
0	B	...	Red
0	B	...	Blue
1	A	...	Red
1	A	...	Blue
1	B	...	Red
1	B	...	Blue
...	...	...	...

A Relation (continued)

So for the relation schema LIKES (DRINKER string, BEER string)

A corresponding relation might be

$$\{(\text{“Luis”}, \text{“Modelo”}),$$

$$(\text{“Sinan”}, \text{“PBR”})\}$$

This is also referred to as a “table”

The entries in the relation are referred to as

- “rows”
- “tuples”
- “records”

Keys

In the relational model, given $R(A_1, A_2, ..., A_n)$

A set of attributes $K = \{K_1, ..., K_m\}$ is a KEY of R if:

- For any valid realization R' of R...
- For all t_1, t_2 in R'...
- If $t_1[K_1] = t_2[K_1]$ and $t_1[K_2] = t_2[K_2]$ and ... $t_1[K_m] = t_2[K_m]$...
- Then it must be the case that $t_1 = t_2$

Note: every relation schema MUST have a key... why?

Keys

What is a key for

STUDENT (NETID, FNAME, LNAME, AGE, COLLEGE)?

What is a key for

LIKES (DRINKER, BEER)?

Keys (continued)

A relation schema can have many keys

Those that are minimal are CANDIDATE KEYs

- "Minimal" means no subset is a key

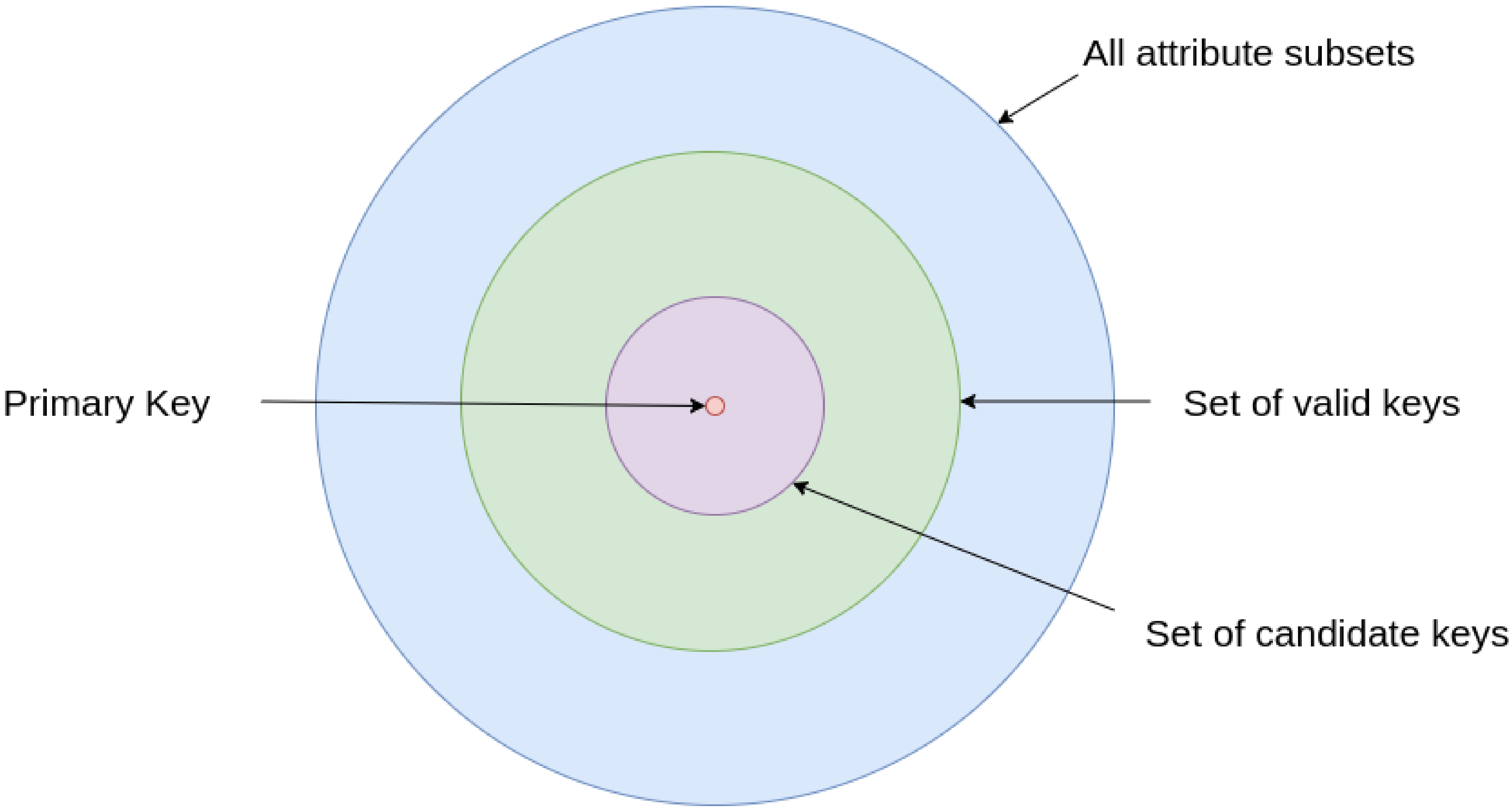
One is designated as the PRIMARY KEY denoted with an underline

- STUDENT (<u>NETID</u>, FNAME, LNAME, AGE, COLLEGE)

Surrogate Key

- No real-world meaning, added to simplify primary key

Keys (continued)



Connecting Relations

The relational model does not have pointers

Why? Two reasons:

- Not nice mathematically
  - ▷ Mathematical elegance key goal in model design
- Implementation difficult
  - ▷ Move an object? All pointers are invalid!
  - ▷ Can have centralized look-up table
  - ▷ But expensive, complicated, plus problem still exists

Connecting Relations

But we still need some notion of between-tuple references

- DRINKERS (DRN_ID, FNAME, LNAME)
- LIKES (DRN_ID, BEER)

Clearly, LIKES.DRN_ID refers to DRINKERS.DRN_ID

- Why not the other way around?

Accomplished via the idea of a FOREIGN KEY

Solution: Foreign Keys

- DRINKERS (DRN_ID, FNAME, LNAME)
- LIKES (DRN_ID, BEER)

Given relation schemas R_1, R_2

- We say a set of attributes K_1 from R_1 is a foreign key to a set of attributes K_2 from R_2 if...
- (1) K_2 is a candidate key for R_2, and...
- (2) For any valid realizations R'_1, R'_2 of R_1, R_2...
- For each $t_1 \in R'_1$, it MUST be the case that there exists $t_2 \in R'_2$ s.t...
- $t_1[K_{1,1}] = t_2[K_{2,1}]$ and $t_1[K_{1,2}] = t_2[K_{2,2}]$ and ... $t_1[K_{1,m}] = t_2[K_{2,m}]$

Intuitively, what does this mean?

Foreign Keys (continued)

Intuitively what does it mean?

- The foreign key (k_1) must be a set of attributes that uniquely identify a record in another table (R_2)
- The combination of attribute values present in every tuple of (R_1) must also be present in (R_2)

Why is this a requirement?

Foreign Keys (continued)

Intuitively what does it mean?

- The foreign key (k_1) must be a set of attributes that uniquely identify a record in another table (R_2)
- The combination of attribute values present in every tuple of (R_1) must also be present in (R_2)

Why is this a requirement?

- To prevent inconsistencies
- To match to a single target

RDBMS enforce these requirements via

- Cascading deletes
- Failed inserts

Foreign Keys (continued)

Which one is R_1 and R_2?

DRINKERS

DRN_ID	**FNAME**	**LNAME**
lg67	Luis	Guzman
sk212	Sinan	Kockara

LIKES

DRN_ID	**BEER**
lg67	Modelo
sk212	PBR
lg67	Corona

Foreign Keys (continued)

What happens here?

DRINKERS

DRN_ID	FNAME	LNAME
lg67	Luis	Guzman
sk212	Sinan	Kockara

LIKES

DRN_ID	BEER
lg67	Modelo
sk212	PBR
lg67	Corona
cmj4	Blue Moon

Queries/Computations in the Relational Model

The original query language was the RELATIONAL CALCULUS

- Fully declarative programming language

next was the RELATIONAL ALGEBRA

- Imperative
- Define a set of operations over relations
- A RA program is then a sequence of those operations
- This is the "abstract machine" of RDBs

Today we use SQL

- Heavily influenced by RC
- Has aspects of RA

Overview of Relational Calculus

RC is a variant on first-order logic

You say: "Give me all tuples t where $P(t)$ holds"

$P(t)$ is a predicate in first-order logic

- A predicate is basically boolean function

Predicates

First order logic allows predicates

- predicate: a function that evals to true/false
- "It's raining on day X" or $Raining(X)$
- "It's cloudy on day X" or $Cloudy(X)$

Can build more complicated preds using logical operations over them

- and ($\wedge$)
- or ($\vee$)
- not ($\neg$)
- implies ($\rightarrow$)
- iff ($\leftrightarrow$)

Predicates (continued)

Example: $Raining(X) \wedge Cloudy(X)$

Evals to true if:

- It is raining and cloudy on day X

Example: $Raining(X) \rightarrow Cloudy(X)$

Evals to true if either:

- It is not raining on day X, or
- It is raining and cloudy on day X

Note the difference between them!

- $\rightarrow$ is like a logical “if-then”

First Order Logic

Just predicates and logical ops?

- You've got predicate logic

But when you add quantification

- $\forall$, $\exists$
- You've got first order logic

Universal Quantification

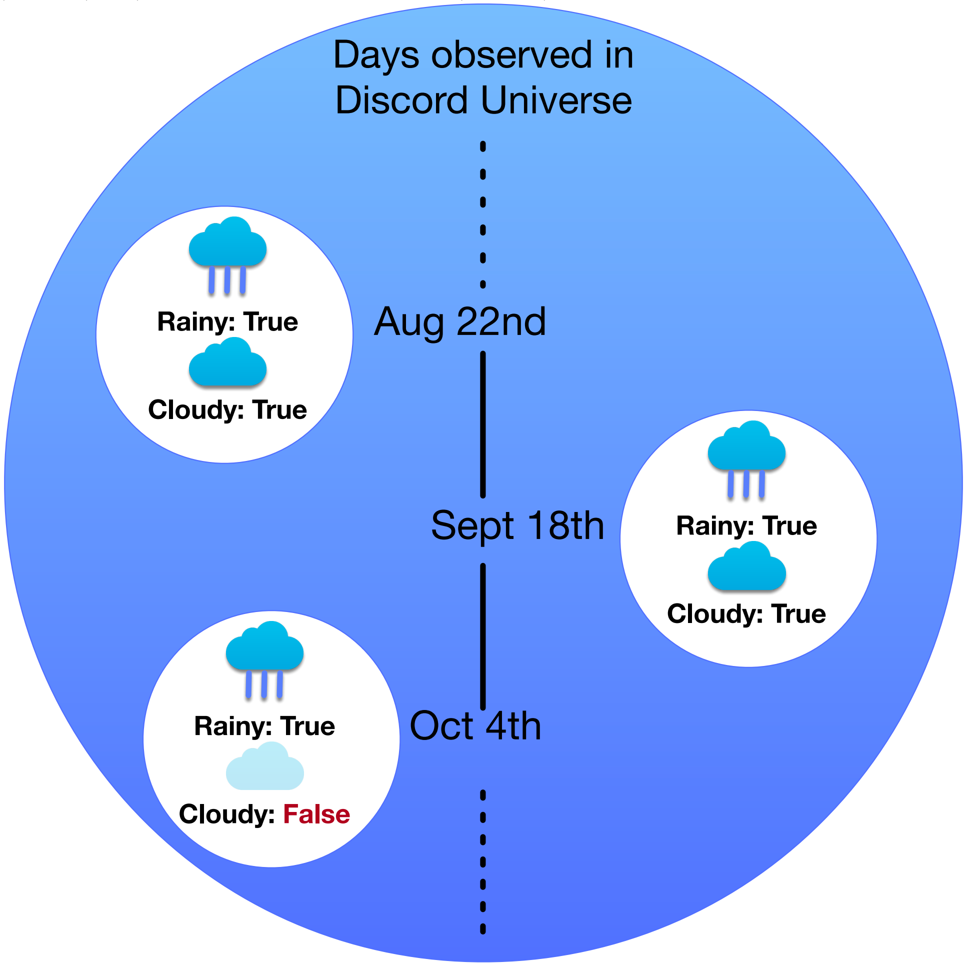
Asserts that a predicate is true all of the time

Example:

- $\forall(X)(Raining(X) \rightarrow Cloudy(X))$
- This is a zero-arg predicate (takes no params)
- Asserts that it only rains when it is cloudy
- Note: idea of universe of discourse is key!

Universal Quantification: Example 1

$$\forall(X)(Raining(X) \rightarrow Cloudy(X))$$



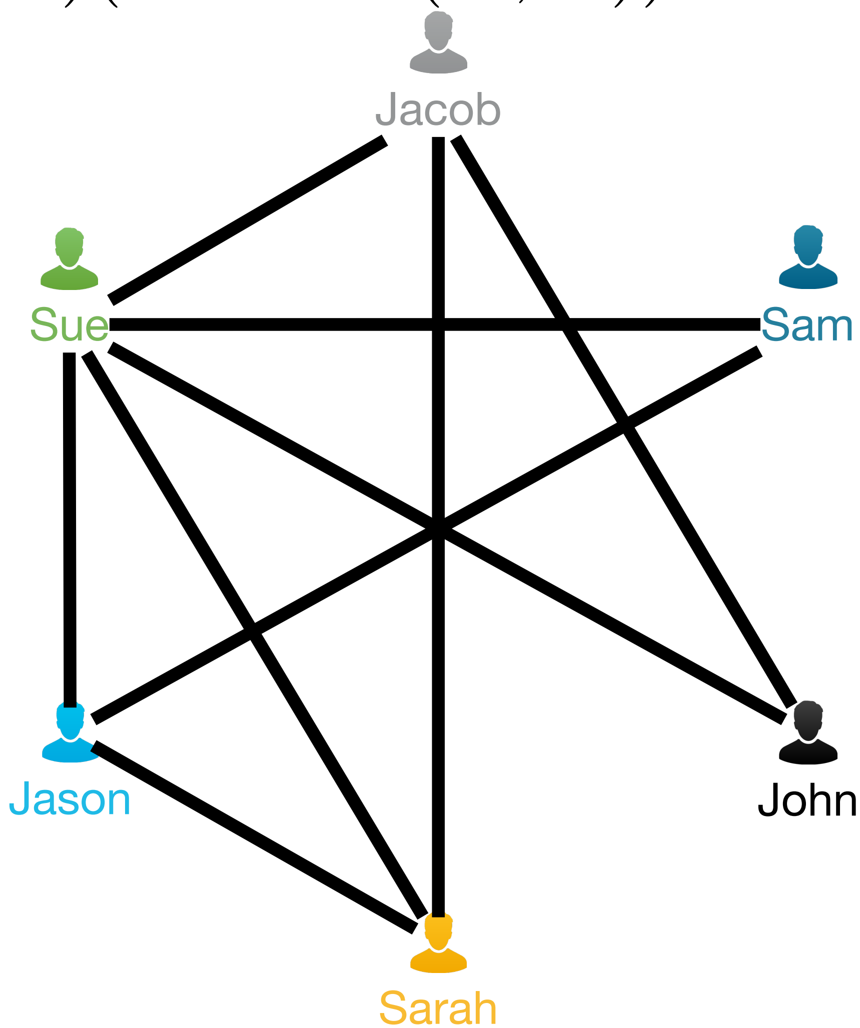
Universal Quantification

Asserts that a predicate is true all of the time

- $\forall(X)(Friends(X, Y))$
- This is a predicate over Y
- Evals to true if the person Y is friends with everyone

Universal Quantification: Example 2

$$\forall(X)(Friends(X, Y))$$



Existential Quantification

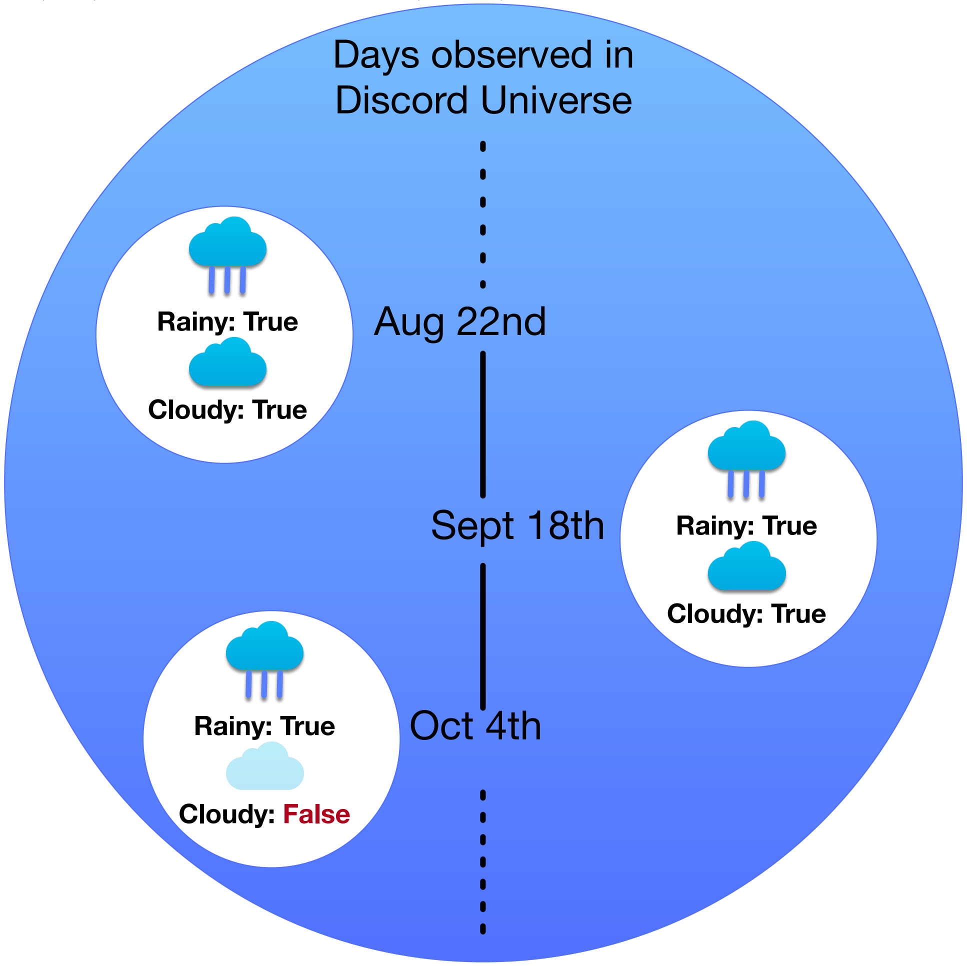
Asserts that a predicate can be satisfied

Example:

- $\exists(X)(Raining(X) \wedge \textsf{not}\ Cloudy(X))$
- Asserts that it can rain when it is not cloudy
- A.K.A a "sun shower"

Existential Quantification: Example 1

$$\exists(X)(Raining(X) \wedge \textbf{not}\ Cloudy(X))$$



Existential Quantification

Asserts that a predicate can be satisfied

Example:

- **not** $\exists(X, Y)(Friends(X, Y) \wedge Friends(X, Z) \wedge Friends(Y, Z))$
- This is a predicate over Z
- Evals to true when?

Existential Quantification

Asserts that a predicate can be satisfied

Example:

- **not** $\exists(X, Y)(Friends(X, Y) \wedge Friends(X, Z) \wedge Friends(Y, Z))$
- This is a predicate over Z
- Evals to true when? (If there is not a pair of friends who are both friendly with Z)

Important Equivalence

$\forall(X)(P(X))$ is equivalent to... **not** $\exists(X)($**not** $P(X))$

- Ex: **not** $\exists(X,Y)(Friends(X,Y) \wedge Friends(X,Z) \wedge Friends(Y,Z))$
- Can be changed to:
- $\forall(X,Y)($**not** $(Friends(X,Y) \wedge Friends(X,Z) \wedge Friends(Y,Z)))$
- Or $\forall(X,Y)($**not** $Friends(X,Y) \vee$ **not** $Friends(X,Z) \vee$ **not** $Friends(Y,Z))$

Which is easier?

- IMO: often easier to reason about $\forall$
- With $\exists$ tend to get into double and triple negatives
- Unfortunately, SQL does not have $\forall$

Questions?