INTRODUCTION TO SQL

CS 564- Fall 2021

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT

- The Relational Model
- SQL: Basics
  - creating a table
  - primary keys
- SQL: Single-table queries
  - SELECT-FROM-WHERE structure
  - DISTINCT/ORDER BY/LIMIT
- SQL: Multi-table queries
  - foreign keys
  - joins

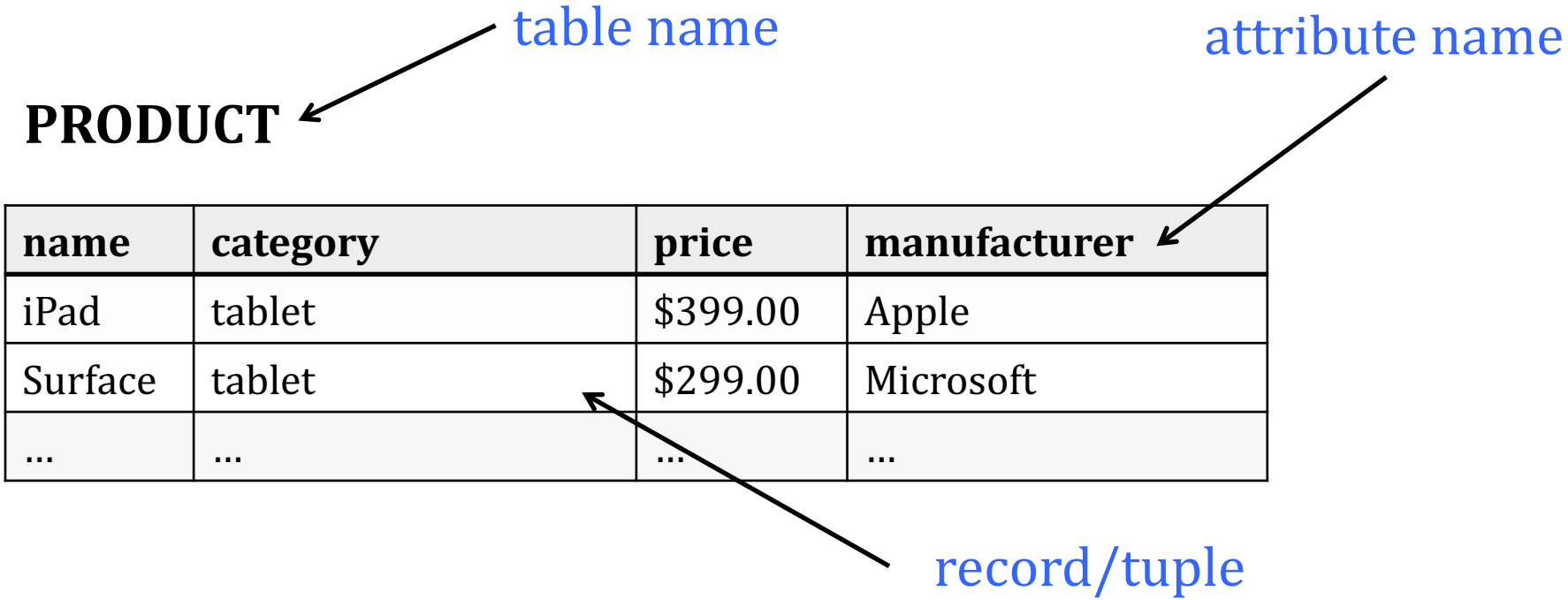
RELATIONAL MODEL

RELATIONAL MODEL

- first proposed by Codd in 1969
- has just a single concept: relation
- the world is represented as a collection of tables
- well-suited for efficient manipulations on computers

RELATION

The data is stored in **tables** (or relations)



DOMAINS

- Each attribute has an atomic type called domain
- A **domain** specifies the set of values allowed
- Examples:
  - integer
  - string
  - real

PRODUCT(name: *string*,
category: *string*,
price: *real*,
manufacturer: *string*)

SCHEMA

The **schema** of a *relation*:

- relation name + attribute names
- **Product** (name, price, category, manufacturer)
- In practice we add the domain for each attribute

The **schema** of a *database:*

- a collection of relation schemas

INSTANCE

The **instance** of a *relation*:

- a set of tuples or records

The **instance** of a *database:*

- a collection of relation instances

EXAMPLE

PRODUCT(name: *string*,
category: *string*,
price: *real*,
manufacturer: *string*)

schema

instance

name	category	price	manufacturer
iPad	tablet	$399.00	Apple
Surface	tablet	$299.00	Microsoft
…	…	…	…

SCHEMA VS INSTANCE

- Analogy with programming languages:
  - schema ~ type
  - instance ~ value
- Important distinction
  - schema: stable over long periods of time
  - instance: changes constantly, as data is inserted/updated/deleted

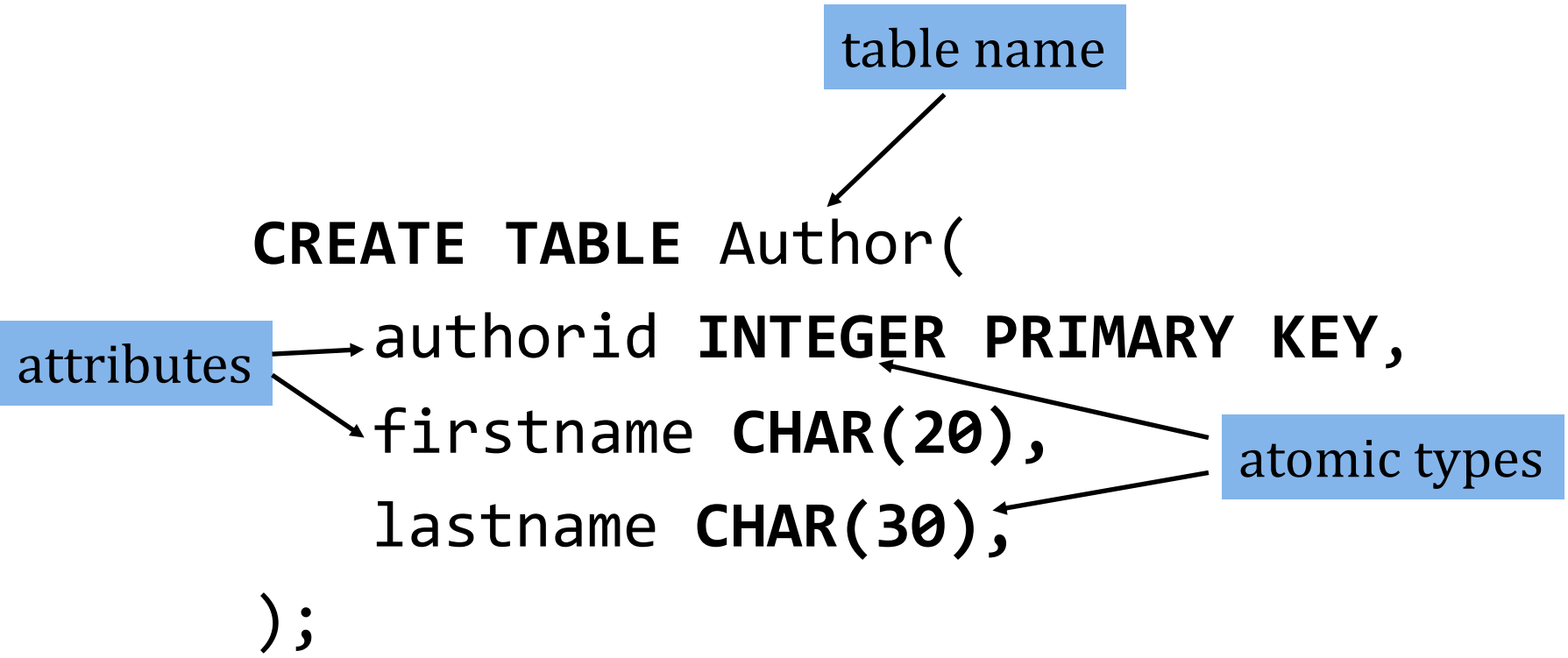
SQL: BASICS

WHAT IS SQL?

- The most widely used database language
- Used to query and manipulate data

- SQL stands for **S**tructured **Q**uery **L**anguage
  - many SQL standards: SQL-92, SQL:1999, SQL:2011
  - vendors support different subsets
  - we will discuss the common functionality

CREATING A TABLE



PRIMARY KEY

A **primary key** is a minimal subset of attributes that is a unique identifier of tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
- In SQL we specify that an attribute is the primary key with the keyword **`PRIMARY KEY`**

UNIQUE KEY

- We can also define a unique key: a subset of attributes that uniquely defines a row:

```
CREATE TABLE Author(
    authorid INTEGER UNIQUE,
    firstname CHAR(20)) ;
```

- There can be only one primary key, but many unique keys!

NULL VALUES

- tuples in SQL relations can have **`NULL`** as a value for one or more attributes
- The meaning depends on context:
  - missing value: e.g., we know that Greece has some population, but we don't know what it is
  - inapplicable: e.g., the value of attribute *spouse* for an unmarried person

NULL VALUES

When creating a table in SQL, we can assert that a particular attribute takes no **NULL** values

```
CREATE TABLE Author(
        authorid INTEGER PRIMARY KEY,
        firstname CHAR(20) NOT NULL,
        lastname CHAR(30)
    );
```

POPULATING A TABLE

- To insert a single tuple:

  INSERT INTO *<relation>*
  VALUES *(<list of values>);*

- We may add to the relation name a list of attributes (if we forget the order)

```
INSERT INTO Author
VALUES(001, 'Dan', 'Brown');
```

SQL: SINGLE-TABLE QUERIES

BASIC SQL QUERY

optional

SELECT [DISTINCT] attributes
FROM one or more tables
WHERE conditions on the tables

conditions of the form: Attr1 **op** Attr2

EXAMPLE

What is the population of USA?

```
SELECT Population
FROM Country
WHERE Code = 'USA';
```

PROJECTION: keeps only the specified attributes

SELECTION: filters the tuples of the relation

SEMANTICS

1. Think of a *tuple variable* ranging over each tuple of the relation in **`FROM`**
2. Check if the current tuple satisfies the **`WHERE`** clause
3. If so, compute the attributes or expressions of the **`SELECT`** clause using this tuple

* IN SELECT CLAUSES

When there is one relation in the **FROM** clause, * in the **SELECT** clause stands for *"all attributes of this relation"*

```
SELECT *
FROM City
WHERE Population >= '1000000'
AND CountryCode = 'USA';
```

RENAMING ATTRIBUTES

If we want the output schema to have different attribute names, we can use **AS** *<new name>* to rename an attribute

```
SELECT Name AS LargeUSACity
FROM City
WHERE Population >= '1000000'
AND CountryCode = 'USA';
```

ARITHMETIC EXPRESSIONS

We can use any arithmetic expression (that makes sense) in the **SELECT** clause

```
SELECT Name,
 (Population/ 1000000) AS PopulationInMillion
FROM City
WHERE Population >= '1000000’ ;
```

WHAT CAN WE USE IN WHERE CLAUSES?

- attribute names of the relations that appear in the **`FROM`** clause
- comparison operators: =, <>, <, >, <=, >=
- arithmetic operations (+, -, /, *)
- **`AND`**, **`OR`**, **`NOT`** to combine conditions
- operations on strings (e.g. concatenation)
- pattern matching: *s* **`LIKE`** *p*
- special functions for comparing dates and times

PATTERN MATCHING

s **LIKE** p: pattern matching on strings

- % = any sequence of characters
- _ = any single character

```
SELECT Name, GovernmentForm
FROM Country
WHERE GovernmentForm LIKE '%Monarchy%';
```

USING DISTINCT

- The default semantics of SQL is **bag** semantics (duplicate tuples are allowed in the output)
- The use of **`DISTINCT`** in the **`SELECT`** clause removes all duplicate tuples in the result, and returns a **set**

```
SELECT DISTINCT GovernmentForm
FROM Country;
```

ORDER BY

The use of **ORDER BY** orders the tuples by the attribute we specify in decreasing (**DESC**) or increasing (**ASC**) order

```
SELECT Name, Population
FROM City
WHERE Population >= '1000000'
ORDER BY Population DESC;
```

LIMIT

- The use of **LIMIT** <*number*> limits the output to be only the specified number of tuples
- It can be used with **ORDER BY** to get the maximum or minimum value of an attribute!

```
SELECT Name, Population
FROM City
ORDER BY Population DESC
LIMIT 2;
```

SQL: MULTI-TABLE QUERIES

FOREIGN KEYS

Suppose that we want to create a table Book, and make sure that *the author of the book exists in the table* Author

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid));
```

FOREIGN KEYS

- Use the keyword **REFERENCES**, as:

FOREIGN KEY *(<list of attributes>)*
REFERENCES *<relation> (<attributes>)*

- Referenced attributes must be declared **PRIMARY KEY** or **UNIQUE**

ENFORCING FK CONSTRAINTS

If there is a foreign-key constraint from attributes of relation ***R*** to the primary key of relation ***S***, two violations are possible:

1. An insert or update to ***R*** introduces values not found in ***S***
2. A deletion or update to ***S*** causes some tuples of ***R*** to dangle

There are 3 ways to enforce foreign key constraints!

ACTION 1: REJECT

- The insertion/deletion/update query is rejected and not executed in the DBMS
- This is the default action if a foreign key constraint is declared

ACTION 2: CASCADE UPDATE

When a tuple referenced is *updated*, the update propagates to the tuples that reference it

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid)
   ON UPDATE CASCADE);
```

ACTION 2: CASCADE DELETE

When a tuple referenced is *deleted*, the deletion propagates to the tuples that reference it

```
CREATE TABLE Book(
   bookid INTEGER PRIMARY KEY,
   title TEXT,
   authorid INTEGER,
   FOREIGN KEY (authorid) REFERENCES
   Author(authorid)
   ON DELETE CASCADE);
```

ACTION 3: SET NULL

- When a delete/update occurs, the values that reference the deleted tuple are set to **NULL**

```
CREATE TABLE Book(
    bookid INTEGER PRIMARY KEY,
    title TEXT,
    authorid INTEGER,
    FOREIGN KEY (authorid) REFERENCES
    Author(authorid)
    ON UPDATE SET NULL);
```

WHAT SHOULD WE CHOOSE?

- When we declare a foreign key, we may choose policies **`SET NULL`** or **`CASCADE`** *independently* for deletions and updates

  ON [UPDATE, DELETE] [SET NULL, CASCADE]

- Otherwise, the default policy (*reject*) is used

MULTIPLE RELATIONS

- We often want to combine data from more than one relation
- We can address several relations in one query by listing them all in the **`FROM`** clause
- If two attributes from different relations have the same name, we can distinguish them by writing *<relation>***.***<attribute>*

EXAMPLE

What is the name of countries that speak Greek?

```
SELECT Name
FROM Country, CountryLanguage
WHERE Code = CountryCode
   AND Language = 'Greek';
```

This is **BAD** style!!

EXAMPLE: GOOD STYLE

```
SELECT Country.Name
FROM Country, CountryLanguage
WHERE Country.Code=CountryLanguage.CountryCode
AND CountryLanguage.Language = 'Greek';

SELECT C.Name
FROM Country C, CountryLanguage L
WHERE C.Code = L.CountryCode
AND L.Language = 'Greek';
```

VARIABLES

Variables are necessary when we want to use two copies of the same relation in the **FROM** clause

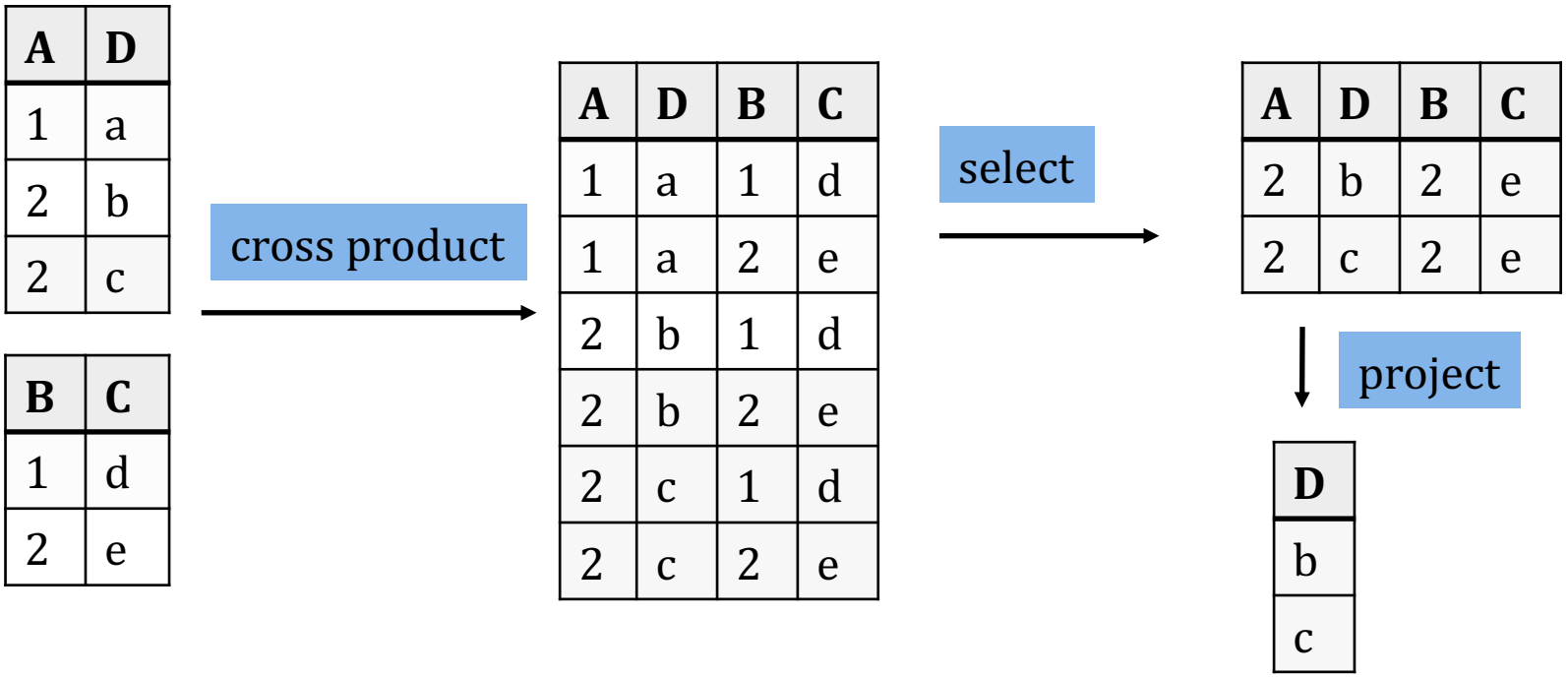
```
SELECT C.Name
FROM Country C, CountryLanguage L1,
CountryLanguage L2
WHERE  C.Code = L1.CountryCode
   AND C.Code = L2.CountryCode
   AND L1.Language = 'Greek’
   AND L2.Language = 'English';
```

SEMANTICS: SELECT-FROM-WHERE

1. Start with the cross product of all the relations in the **`FROM`** clause
2. Apply the conditions from the **`WHERE`** clause
3. Project onto the list of attributes and expressions in the **`SELECT`** clause
4. If **`DISTINCT`** is specified, eliminate duplicate rows

SEMANTICS OF SQL: EXAMPLE

```
SELECT R.D
FROM   R, S
WHERE  R.A = S.B AND S.C = 'e' ;
```



SEMANTICS OF SQL: NESTED LOOP

```
SELECT a1, a2, …, ak
FROM   R1, R2, …, Rn
WHERE  Conditions
```

answer := {}
for t_1 **in** R_1 **do**
  for t_2 **in** R_2 **do**
    …..
      for t_n **in** R_n **do**
        if Conditions
          then *answer := answer* ∪ $\{(a_1,\ldots,a_k)\}$
return *answer*

SEMANTICS OF SQL

- The query processor will almost never evaluate the query this way
- SQL is a declarative language
- The DBMS figures out the most efficient way to compute it
- We will discuss this later in the course when we talk about *query optimization*