# CS150A Database

Lu Sun

School of Information Science and Technology

ShanghaiTech University

Sept. 6, 2022

Today:

- Introduction to SQL
- DDL & DML

Readings:

- Database Management Systems (DBMS), Chapter 5
- Lecture note SQL I



*Acknowledgement: Joe Hellerstein@UCBerkeley's scourse notes*

# SQL Roots

- Developed @IBM Research in the 1970s
  - System R project
  - Vs. Berkeley's Quel language (Ingres project)
- Commercialized/Popularized in the 1980s
  - "Intergalactic Dataspeak"
  - IBM beaten to market by a startup called Oracle

# SQL's Persistence

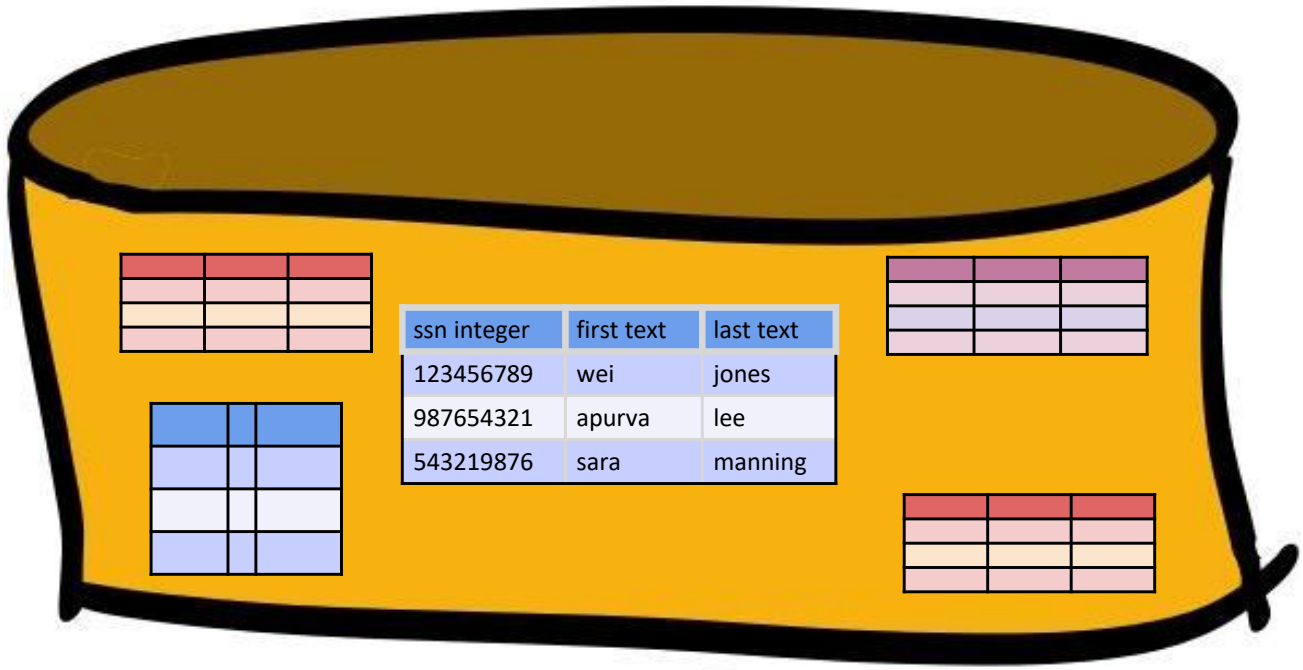
- Over 40 years old!
- Questioned repeatedly
  - 90's: Object-Oriented DBMS (OQL, etc.)
  - 2000's: XML (Xquery, Xpath, XSLT)
  - 2010's: NoSQL & MapReduce
- SQL keeps re-emerging as the standard
  - Even Hadoop, Spark etc. mostly used via SQL
  - May not be perfect, but it is useful

# SQL Pros and Cons

- Declarative!
  - Say *what* you want, not *how* to get it
- Implemented widely
  - With varying levels of efficiency, completeness
- Constrained
  - Not targeted at Turing-complete tasks
- General-purpose and feature-rich
  - many years of added features
  - extensible: callouts to other languages, data sources

# Relational Terminology

- *Database*: Set of named Relations



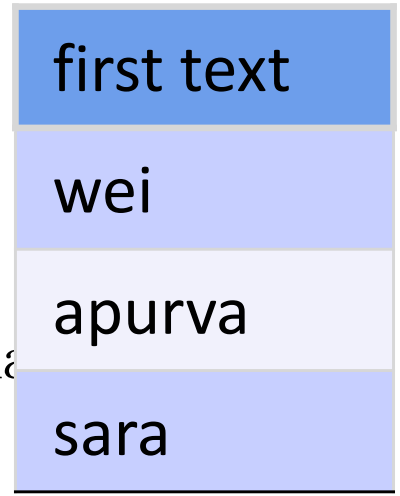
# Relational Terminology, Pt 2.

- *Database*: Set of named Relations
- *Relation* (*Table*):
  - *Schema:* description (“metadata”)
  - *Instance:* set of data satisfying the schema

| ssn integer | first text | last text |
|---|---|---|
| 123456789 | wei | jones |
| 987654321 | apurva | lee |
| 543219876 | sara | manning |

# Relational Terminology, Pt. 3

- *Database*: Set of named Relations
- *Relation* (*Table*):
  - *Schema:* description ("metadata")
  - *Instance:* set of data satisfying the schema
- *Attribute* (*Column*, *Field*)




equal name.

# Relational Terminology, Pt. 4

- *Database*: Set of named Relations
- *Relation* (Table):
  - *Schema*: description ("metadata")
  - *Instance*: set of data satisfying the schema
- *Attribute* (Column, Field)
- *Tuple* (Record, Row)

equal name.

| | | |
|---|---|---|
| 543219876 | sara | manning |

# Relational Tables

- *Schema* is fixed:
  - unique attribute names, *atomic* types
  - folks (ssn integer, first text, last text)
- *Instance* can change often
  - a *multi*set of "rows" ("tuples")

{(123456789, 'wei', 'jones'),
(987654321, 'apurva', 'lee'),
(543219876, 'sara', 'manning'),
(987654321, 'apurva', 'lee')}

cannot divide.

, or be same

多集

same row, ok. (instance).

# Quick Check 1

- Is this a relation?

| num integer | street text | zip integer |
|---|---|---|
| 84 | Maple Ave | 54704 |
| 22 | High | Street |
| 75 | Hearst Ave | 94720 |

fit scheme

76425

# Quick Check 2

- Is this a relation?

unique col name

| num integer | street text | num integer |
|---|---|---|
| 84 | Maple Ave | 54704 |
| 22 | High Street | 76425 |
| 75 | Hearst Ave | 94720 |

# Quick Check 3

- Is this a relation?

∴ No tuple in tuple
not atomic data type

| first text | last text | addr address |
|---|---|---|
| wei | jones | (84, 'Maple', 54704) |
| apurva | lee | (22, 'High', 76425) |
| sara | manning | (75, 'Hearst', 94720) |

# SQL Language

- Two sublanguages:
  - DDL – Data Definition Language
    - Define and modify schema
  - DML – Data Manipulation Language
    - Queries can be written intuitively.
- RDBMS responsible for efficient evaluation.
  - Choose and run algorithms for declarative queries
    - Choice of algorithm must not affect query answer.

# Example Database

**Sailors**

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

**Boats**

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

**Reserves**

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12/2015 |
| 2 | 102 | 9/13/2015 |

# The SQL DDL: Sailors

```
CREATE TABLE Sailors (
      sid INTEGER,
      sname CHAR(20),
      rating INTEGER,
      age FLOAT
```

Type.

| **sid** | **sname** | **rating** | **age** |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

# The SQL DDL: Sailors, Pt. 2

```
CREATE TABLE Sailors (
      sid INTEGER,
      sname CHAR(20),
      rating INTEGER,
      age FLOAT
PRIMARY KEY (sid));
```

| **sid** | **sname** | **rating** | **age** |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

# The SQL DDL: Primary Keys

```
CREATE TABLE Sailors (
        sid INTEGER,
        sname CHAR(20),
        rating INTEGER,
        age FLOAT
```

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

No duplicate.

- Primary Key column(s)
  - Provides a unique "lookup key" for the relation
  - Cannot have any duplicate values
  - Can be made up of >1 column
    - E.g. (firstname, lastname)

# The SQL DDL: Boats

```
CREATE TABLE Sailors (
      sid INTEGER,
      sname CHAR(20),
      rating INTEGER,
      age FLOAT

CREATE TABLE Boats (
    bid INTEGER,
    bname CHAR (20),
    color CHAR(10),
    PRIMARY KEY (bid));
```

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

# The SQL DDL: Reserves

```
CREATE TABLE Sailors (
        sid INTEGER,
        sname CHAR(20),
        rating INTEGER,
        age FLOAT

CREATE TABLE Boats (
    bid INTEGER,
    bname CHAR (20),
    color CHAR(10),
    PRIMARY KEY (bid));

CREATE TABLE Reserves (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day);
```

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12 |
| 2 | 102 | 9/13 |

# The SQL DDL: Reserves Pt. 2

```
CREATE TABLE Sailors (
        sid INTEGER,
        sname CHAR(20),
        rating INTEGER,
        age FLOAT

CREATE TABLE Boats (
    bid INTEGER,
    bname CHAR (20),
    color CHAR(10),
    PRIMARY KEY (bid));

CREATE TABLE Reserves (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (sid) REFERENCES Sailors,
```

"pointers"

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12 |
| 2 | 102 | 9/13 |

# The SQL DDL: Foreign Keys

```
CREATE TABLE Sailors (
        sid INTEGER,
        sname CHAR(20),
        rating INTEGER,
        age FLOAT

CREATE TABLE Boats (
    bid INTEGER,
    bname CHAR (20),
    color CHAR(10),
    PRIMARY KEY (bid));

CREATE TABLE Reserves (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (sid) REFERENCES Sailors,
        FOREIGN KEY (bid) REFERENCES Boats);
```

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12 |
| 2 | 102 | 9/13 |

# The SQL DDL: Foreign Keys Pt. 2

- Foreign key references a table
  - Via the primary key of that table
- Need not share the name of the referenced primary key

```
CREATE TABLE Reserves (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (sid) REFERENCES
    Sailors,
        FOREIGN KEY (bid) REFERENCES
        Boats);
```

ref the prime

foreign keys

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

| bid | bname | color |
|---|---|---|
| 101 | Nina | red |
| 102 | Pinta | blue |
| 103 | Santa Maria | red |

| sid | bid | day |
|---|---|---|
| 1 | 102 | 9/12 |
| 2 | 102 | 9/13 |

# The SQL DML

- Find all 27-year-old sailors:

```
SELECT *
FROM  Sailors AS S
WHERE S.age=27;
```

short for

No: NaN

- To find just names and rating, replace the first line to:

```
SELECT S.sname,
S.rating
```

**Sailors**

| sid | sname | rating | age |
|---|---|---|---|
| 1 | Fred | 7 | 22 |
| 2 | Jim | 2 | 39 |
| 3 | Nancy | 8 | 27 |

# Basic Single-Table Queries

- **SELECT** [*DISTINCT*] *<column expression list>*
  **FROM** *<single table>*
  [**WHERE** *<predicate>*]

- Simplest version is straightforward
  - Produce all tuples in the table that satisfy the predicate
  - Output the expressions in the SELECT list
  - Expression can be a column reference, or an arithmetic expression over column refs

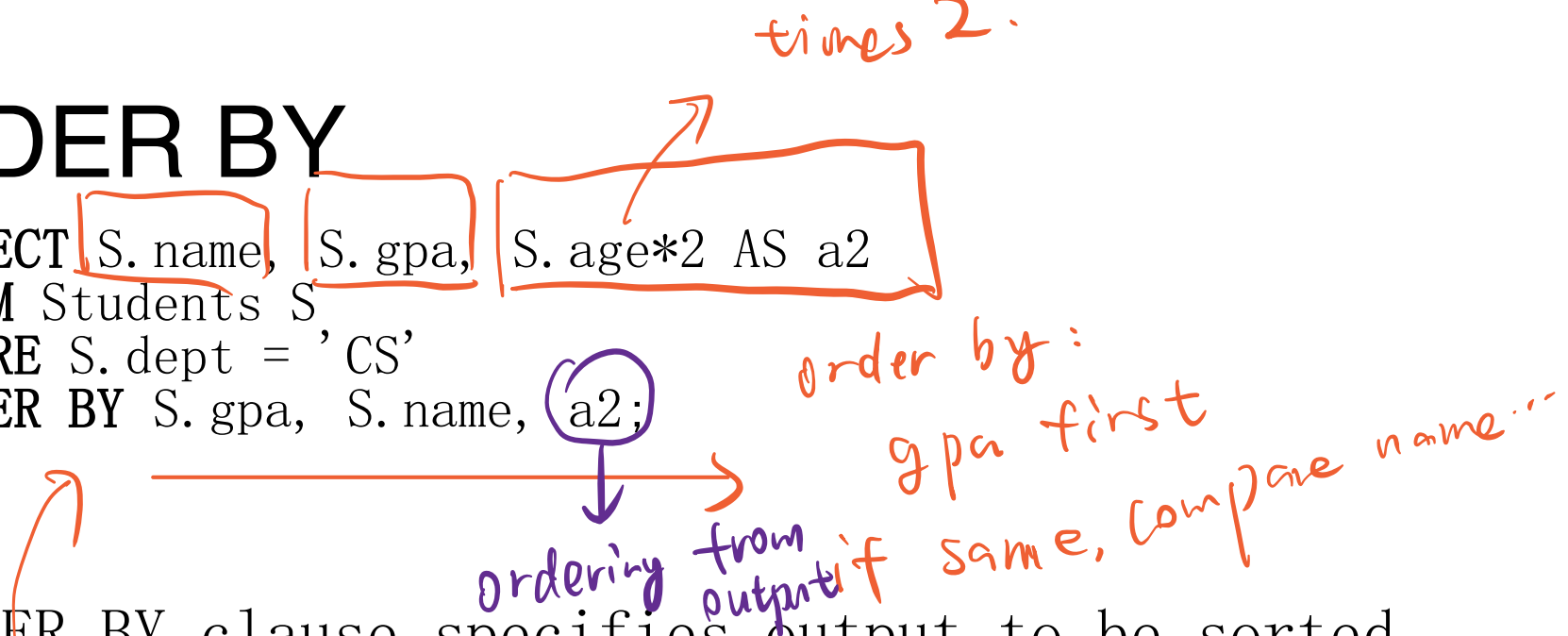
# SELECT DISTINCT

```
SELECT DISTINCT S.name, S.gpa
FROM students S
WHERE S.dept = 'CS'
```

- DISTINCT specifies removal of duplicate rows before output
- Can refer to the students table as "S", this is called an alias

# ORDER BY

```
SELECT S.name, S.gpa, S.age*2 AS a2
FROM Students S
WHERE S.dept = 'CS'
ORDER BY S.gpa, S.name, a2;
```




- ORDER BY clause specifies output to be sorted
  - *Lexicographic* ordering
- Obviously must refer to columns in the output
  - Note the AS clause for naming output columns!

# ORDER BY, Pt. 2

```
SELECT  S.name, S.gpa, S.age*2 AS a2
FROM Students S
WHERE S.dept = 'CS'
ORDER BY S.gpa DESC, S.name ASC, a2;
```

Asc

- Ascending order by default, but can be overridden
  - DESC flag for descending, ASC for ascending
  - Can mix and match, lexicographically

# LIMIT

```
SELECT  S.name, S.gpa, S.age*2 AS a2
FROM Students S,
WHERE S.dept = 'CS'
ORDER BY S.gpa DESC, S.name ASC, a2;
LIMIT 3 ;
```

only see 3.

ORDER BY + LIMIT

- Only produces the first <integer> output rows
- Typically used with ORDER BY
  - Otherwise the output is *non-deterministic*
  - Not a "pure" declarative construct in that case - output set depends on algorithm for query processing

"Declarative"

# Aggregates

Not in where

- SELECT [DISTINCT] AVG(S.gpa)
  FROM Students S
  WHERE S.dept = 'CS'

- Before producing output, compute a summary (a.k.a. an *aggregate*) of some arithmetic expression
- Produces 1 row of output
  - with one column in this case
- Other aggregates: SUM, COUNT, MAX, MIN

1-1 cell

# GROUP BY (with aggregates)

double:

```
SELECT [DISTINCT] AVG(S.gpa), S.dept
FROM Students S
GROUP BY S.dept
```

Group by A, B

两个的数值均相同

才视作一种

- Partition table into groups with same GROUP BY column values
  - Can group by a list of columns
- Produce an aggregate result per group
  - Cardinality of output = # of distinct group values
- Note: can put grouping columns in SELECT list

# HAVING : (Must) after grouping

```
SELECT [DISTINCT] AVG(S.gpa), S.dept
FROM Students S
GROUP BY S.dept
HAVING COUNT(*) > 2
```

- The HAVING predicate filters groups
- HAVING is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list
  - I.e. aggs or GROUP BY columns
- HAVING can only be used in aggregate queries
- It's an optional clause

# Putting it all together

```
4 SELECT S.dept, AVG(S.gpa), COUNT(*)
1 FROM Students S
2 WHERE S.gender = 'F'
3 GROUP BY S.dept
5 HAVING COUNT(*) >= 2
6 ORDER BY S.dept;
```

count record num of each dept

Range.

① Group by — Agg

② Group by's target - Select's.

output: S.dept dept's average GPA

COUNT of Female in dept.

access_log

```
+-----+---------+-------+------------+
| aid | site_id | count | date       |
+-----+---------+-------+------------+
|   1 |       1 |    45 | 2016-05-10 |
|   2 |       3 |   100 | 2016-05-13 |
|   3 |       1 |   230 | 2016-05-14 |
|   4 |       2 |    10 | 2016-05-14 |
|   5 |       5 |   205 | 2016-05-14 |
|   6 |       4 |    13 | 2016-05-15 |
|   7 |       3 |   220 | 2016-05-15 |
|   8 |       5 |   545 | 2016-05-16 |
|   9 |       3 |   201 | 2016-05-17 |
+-----+---------+-------+------------+
```

## SQL COUNT(column_name) 实例

下面的 SQL 语句计算 "access_log" 表中 "site_id"=3 的总访问量:

**实例**

```
SELECT COUNT(count) AS nums FROM access_log
WHERE site_id=3;
```

## SQL COUNT(*) 实例

下面的 SQL 语句计算 "access_log" 表中总记录数:

**实例**

```
SELECT COUNT(*) AS nums FROM access_log;
```

执行以上 SQL 输出结果如下:

```
mysql> SELECT COUNT(*) AS nums FROM access_log;
+------+
| nums |
+------+
|    9 |
+------+
1 row in set (0.00 sec)
```

## SQL COUNT(DISTINCT column_name) 实例

下面的 SQL 语句计算 "access_log" 表中不同 site_id 的记录数:

**实例**

```
SELECT COUNT(DISTINCT site_id) AS nums FROM access_log;
```

执行以上 SQL 输出结果如下:

```
mysql> SELECT COUNT(DISTINCT site_id) AS nums FROM access_log;
+------+
| nums |
+------+
|    5 |
+------+
1 row in set (0.00 sec)
```

SQL Script

```
CREATE TABLE mysql_test_a (
gpa INT(6) UNSIGNED ,
dept VARCHAR(30) NOT NULL,
gender VARCHAR(30) NOT NULL,
name VARCHAR(30) NOT NULL
);

INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('4', 'sist', 'F','A');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('6', 'sist', 'M','B');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('6', 'sist', 'F','C');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('8', 'spst', 'F','D');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('10', 'spst', 'M','E');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('10', 'spst', 'F','F');
INSERT INTO `mysql_test_a` (`gpa`, `dept`, `gender`,`name`) VALUES ('10', 'sfst', 'F','G');
```

SQL Query

```
SELECT S.dept, AVG(S.gpa), COUNT(*)
FROM mysql_test_a S
WHERE S.gender = 'F'
GROUP BY S.dept
HAVING COUNT(*)>1
ORDER BY S.dept
```

Result

| dept | AVG(S.gpa) | COUNT(*) |
|---|---|---|
| sist | 5.0000 | 2 |
| spst | 9.0000 | 2 |

# DISTINCT Aggregates

Are these the same or different?

```
SELECT COUNT(DISTINCT S.name)
FROM Students S
WHERE S.dept = 'CS';
```

before counting remove duplicate

useless

only 1 row

```
SELECT DISTINCT COUNT(S.name)
FROM Students S
WHERE S.dept = 'CS';
```

illegal.

# What Is This Asking For?

```
SELECT S.name, AVG(S.gpa)
FROM Students S
GROUP BY S.dept;
```

use col in select that not in group by.

# SQL DML:
# General Single-Table Queries

- **SELECT [DISTINCT]** *<column expression list>*
  **FROM** *<single table>*
  [**WHERE** *<predicate>*]
  [**GROUP BY** *<column list>*
  [**HAVING** *<predicate>*] ]
  [**ORDER BY** *<column list>*]
  [**LIMIT** <integer>];

output

# Summary

- Relational model has **well-defined query semantics**
- Modern SQL extends "pure" relational model

  *(some extra goodies for duplicate row, non-atomic types… more in next lecture)*
- Typically, many ways to write a query
  - DBMS figures out a fast way to execute a query, regardless of how it is written.