Intro to Databases (COMP_SCI 339)

01 Course Introduction

Northwestern University

WINTER 2024

Andrew Crotty

TODAY'S AGENDA

Relational Model

Relational Algebra

SQL Overview

COURSE OVERVIEW

This course is about the design and implementation of database management systems (DBMSs).

This is **not** a course about how to use a DBMS to build applications or how to administer a DBMS.

DATABASE EXAMPLE

Create a database that models a digital music store to keep track of artists and albums.

Things we need for our store:
→ Information about Artists
→ What Albums those Artists released

FLAT FILE STRAWMAN

Store our database as comma-separated value (CSV) files that we manage ourselves via application code.

→ Use a separate file per entity.

→ The application must parse the files each time we want to read/update records.

FLAT FILE STRAWMAN

Create a database that models a digital music store.

Artist(name, year, country)

```
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

Album(name, artist, year)

```
"Enter the Wu-Tang","Wu-Tang Clan",1993
"St.Ides Mix Tape","Wu-Tang Clan",1994
"Liquid Swords","GZA",1990
```

FLAT FILE STRAWMAN

Example: Get the year that GZA went solo.

Artist(name, year, country)

```
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

```
for line in file.readlines():
  record = parse(line)
  if record[0] == "GZA":
    print(int(record[1]))
```

FLAT FILES: DATA INTEGRITY

How do we ensure that the artist is the same for each album entry?

What if somebody overwrites the album year with an invalid string?

What if there are multiple artists on an album?

What happens if we delete an artist that has albums?

FLAT FILES: IMPLEMENTATION

How do you find a particular record?

What if we now want to create a new application that uses the same database?

What if two threads try to write to the same file at the same time?

FLAT FILES: DURABILITY

What if the machine crashes while our program is updating a record?

What if we want to replicate the database on multiple machines for high availability?

DATABASE MANAGEMENT SYSTEM

A database management system (**DBMS**) is software that allows applications to store and analyze information in a database.

A general-purpose DBMS supports the definition, creation, querying, update, and administration of databases in accordance with some data model.

DATA MODELS

A data model is a collection of concepts for describing the data in a database.

A schema is a description of a particular collection of data, using a given data model.

DATA MODELS

Relational ← Most DBMSs
Key/Value
Graph
Document / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value

DATA MODELS

Relational

Key/Value

Graph

Document / Object

Wide-Column / Column-family

Array / Matrix / Vectors

Hierarchical

Network

Multi-Value

← NoSQL

DATA MODELS

Relational
Key/Value
Graph
Document / Object
Wide-Column / Column-family
Array / Matrix / Vectors ← Machine Learning
Hierarchical
Network
Multi-Value

DATA MODELS

Relational
Key/Value
Graph
Document / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value

← Obsolete / Legacy / Rare

DATA MODELS

$\boxed{\text{Relational}}$ ← This Course

Key/Value

Graph

Document / Object

Wide-Column / Column-family

Array / Matrix / Vectors

Hierarchical

Network

Multi-Value

EARLY DBMSs

Early database applications were difficult to build and maintain on available DBMSs in the 1960s.
→ Examples: IDS, IMS, CODASYL
→ Computers were expensive, humans were cheap.

Tight coupling between logical and physical layers.

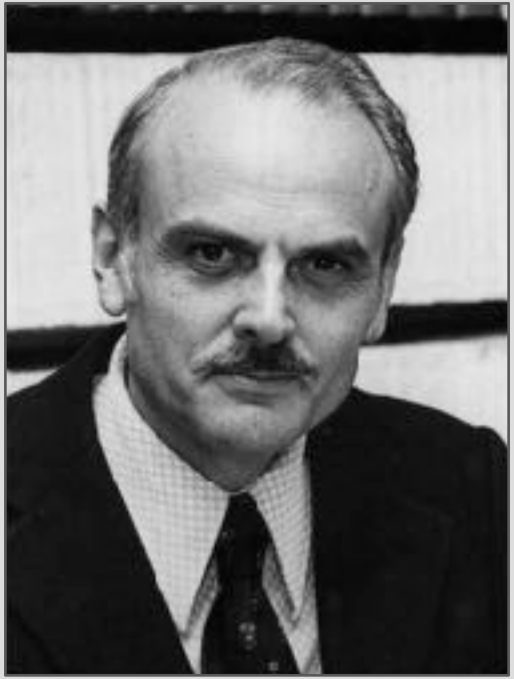
Programmers had to (roughly) know what queries the application would execute before they could deploy the database.

EARLY DBMSs

Ted Codd was a mathematician working at IBM Research in the late 1960s.

He saw IBM's developers spending their time rewriting database programs every time the database's schema or layout changed.

Devised the relational model in 1969.



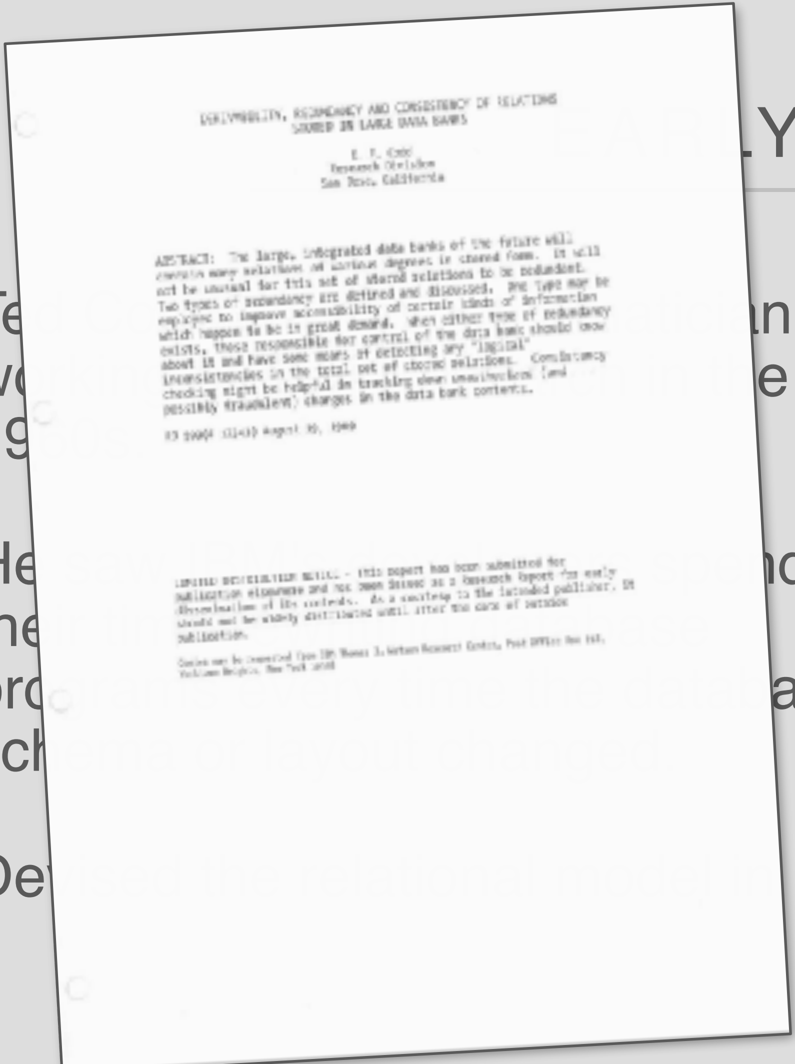
Edgar F. Codd

EARLY DB

Te an
wo e late
19

He nding
the
pro ase's
sch

Dev 1969.



DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS
STORED IN LARGE DATA BANKS

E. F. Codd
Research Division
San Jose, California

ABSTRACT: The large, integrated data banks of the future will contain many relations of various degrees in stored form. It will not be unusual for this set of stored relations to be redundant. Two types of redundancy are defined and discussed. One type may be employed to improve accessibility of certain kinds of information which happen to be in great demand. When either type of redundancy exists, those responsible for control of the data bank should know about it and have some means of detecting any "logical" inconsistencies in the total set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

Information Retrieval

P. BAXENDALE, Editor

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

1. Relational Model and Normal Form

Communications of the ACM

RELATIONAL MODEL

The relational model defines a database abstraction based on relations to avoid maintenance overhead.

Key tenets:

→ Store database in simple data structures (relations).
→ Physical storage left up to the DBMS implementation.
→ Access data through high-level language, DBMS figures out best execution strategy.

RELATIONAL MODEL

Structure: The definition of the database's relations and their contents.

Integrity: Ensure the database's contents satisfy constraints.

Manipulation: Programming interface for accessing and modifying a database's contents.

RELATIONAL MODEL

A relation is an unordered set that contain the relationship of attributes that represent entities.

A tuple is a set of attribute values (also known as its domain) in the relation.

→ Values are (normally) atomic/scalar.

→ The special value **NULL** is a member of every domain (if allowed).

Artist(name, year, country)

name	year	country
Wu-Tang Clan	1992	USA
Notorious BIG	1992	USA
GZA	1990	USA

n-ary Relation

=

Table with *n* columns

RELATIONAL MODEL: PRIMARY KEYS

A relation's primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if a table does not define one.

Auto-generation of unique integer primary keys:
→ **SEQUENCE** (SQL:2003)
→ **AUTO_INCREMENT** (MySQL)

Artist(id, name, year, country)

id	name	year	country
123	Wu-Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	GZA	1990	USA

RELATIONAL MODEL: FOREIGN KEYS

A foreign key specifies that an attribute from one relation has to map to a tuple in another relation.

RELATIONAL MODEL: FOREIGN KEYS

Artist(<u>id</u>, name, year, country)

id	name	year	country
123	Wu-Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	GZA	1990	USA

Album(<u>id</u>, name, artists, year)

id	name	artists	year
11	Enter the Wu-Tang	123	1993
22	St.Ides Mix Tape	???	1994
33	Liquid Swords	789	1995

RELATIONAL MODEL: FOREIGN KEYS

ArtistAlbum(artist_id, album_id)

artist_id	album_id
123	11
123	22
789	22
456	22

Artist(id, name, year, country)

id	name	year	country
123	Wu-Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	GZA	1990	USA

Album(id, name, year)

id	name	year
11	Enter the Wu-Tang	1993
22	St.Ides Mix Tape	1994
33	Liquid Swords	1995

DATA MANIPULATION LANGUAGES (DML)

Methods to store and retrieve information from a database.

Procedural:
→ The query specifies the (high-level) strategy to find the desired result based on sets / bags.

← Relational Algebra

Non-Procedural (Declarative):
→ The query specifies only what data is wanted and not how to find it.

← Relational Calculus

RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

SQL is the *de facto* standard (many dialects).

```
for line in file.readlines():
  record = parse(line)
  if record[0] == "GZA":
    print(int(record[1]))
```

```
SELECT year FROM artists
 WHERE name = 'GZA'
```

SQL HISTORY

In 1971, IBM created its first relational query language called SQUARE.

IBM then created "SEQUEL" in 1972 for IBM System R prototype DBMS.
→ Structured English Query Language

IBM releases commercial SQL-based DBMSs:
→ System/38 (1979), SQL/DS (1981), and DB2 (1983).

SQL HISTORY

In 1971, IBM created
language called SQ

IBM then created "S
System R prototype
→ Structured English Q

IBM releases comm
→ System/38 (1979), SQL/DS (1981), and DB2 (1983).

Q2. Find the average salary of employees in the Shoe Department.

$$AVG\,(\ {}_{SAL}\,EMP'_{DEPT}\,('SHOE'))$$

Mappings may be *composed* by applying one mapping to the result of another, as illustrated by Q3.

Q3. Find those items sold by departments on the second floor.

$$ITEM\ {}^{SALES}{}_{DEPT}\ \circ\ {}_{DEPT}{}^{LOC}{}_{FLOOR}\ (2)$$

The floor '2' is first mapped to the departments located there, and then to the items which they sell. The range of the inner mapping must be compatible with the domain of the outer mapping, but they need not be identical, as illustrated by Q4.

SQL HISTORY

ANSI Standard in 1986. ISO in 1987
→ Structured Query Language

Current standard is **SQL:2016**
→ **SQL:2016** → JSON, Polymorphic tables
→ **SQL:2011** → Temporal DBs, Pipelined DML
→ **SQL:2008** → Truncation, Fancy Sorting
→ **SQL:2003** → XML, Windows, Sequences, Auto-Gen IDs.
→ **SQL:1999** → Regex, Triggers, OO

The minimum language syntax a system needs to say that it supports SQL is **SQL-92**.

SQL HIS

ANSI Standard in 1986. ISO
→ Structured Query Language

Current standard is **SQL:201**
→ **SQL:2016** → JSON, Polymorp
→ **SQL:2011** → Temporal DBs, P
→ **SQL:2008** → Truncation, Fanc
→ **SQL:2003** → XML, Windows,
→ **SQL:1999** → Regex, Triggers,

The minimum language syn
say that it supports SQL is SQL-92.

The Rise of SQL › It's become the second programming language everyone needs to know

SQL dominated the jobs ranking in *IEEE Spectrum*'s interactive rankings of the top programming languages this year. Normally, the top position is occupied by Python or other mainstays, such as C, C++, Java, and JavaScript, but the sheer number of times employers said they wanted developers with SQL skills, albeit in addition to a more general-purpose language, boosted it to No. 1.

So what's behind SQL's soar to the top? The ever-increasing use of databases, for one. SQL has become the primary query language for accessing and managing data stored in such databases—specifically relational databases, which represent data in table form with rows and columns. Databases serve as the foundation of many enterprise applications and are increasingly found in other places as well, for example taking the place of traditional file systems in smartphones.

"This ubiquity means that every software developer will have to interact with databases no matter the field, and SQL is the de facto standard for interacting with databases," says Andy Pavlo, a professor specializing in database management at the Carnegie Mellon University (CMU) School of Computer Science and a member of the CMU database group.

RELATIONAL LANGUAGES

Data Manipulation Language (DML)
Data Definition Language (DDL)
Data Control Language (DCL)

Also includes:
→ View definition
→ Integrity & Referential Constraints
→ Transactions

Important: SQL is based on **bags** (duplicates) not **sets** (no duplicates).

RELATIONAL ALGEBRA

Fundamental operations to retrieve and manipulate tuples in a relation.
→ Based on set algebra.

Each operator takes one or more relations as its inputs and outputs a new relation.
→ We can "chain" operators together to create more complex operations.

σ	Select
Π	Projection
$\cup$	Union
$\cap$	Intersection
$-$	Difference
$\times$	Product
$\bowtie$	Join

RELATIONAL ALGEBRA: SELECT

Choose a subset of the tuples from a relation that satisfies a selection predicate.

→ Predicate acts as a filter to retain only tuples that fulfill its qualifying requirement.

→ Can combine multiple predicates using conjunctions / disjunctions.

Syntax: $\sigma_{predicate}(R)$

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\sigma_{a\_id='a2'}(R)$

a_id	b_id
a2	102
a2	103

$\sigma_{a\_id='a2' \wedge b\_id>102}(R)$

a_id	b_id
a2	103

```
SELECT * FROM R
 WHERE a_id='a2' AND b_id>102
```

RELATIONAL ALGEBRA: PROJECTION

Generate a relation with tuples that contains only the specified attributes.

→ Can rearrange attributes' ordering.

→ Can manipulate the values.

Syntax: $\pi_{A1,A2,\ldots,An}(R)$

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\Pi_{b\_id-100,a\_id}(\sigma_{a\_id='a2'}(R))$

b_id-100	a_id
2	a2
3	a2

```
SELECT b_id-100, a_id
  FROM R WHERE a_id = 'a2'
```

RELATIONAL ALGEBRA: UNION

Generate a relation that contains all tuples that appear in either only one or both input relations.

Syntax: (R ∪ S)

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R ∪ S)

a_id	b_id
a1	101
a2	102
a3	103
a3	103
a4	104
a5	105

```
(SELECT * FROM R)
UNION ALL
(SELECT * FROM S)
```

RELATIONAL ALGEBRA: INTERSECTION

Generate a relation that contains only the tuples that appear in both of the input relations.

Syntax: (R ∩ S)

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R ∩ S)

a_id	b_id
a3	103

```
(SELECT * FROM R)
INTERSECT
(SELECT * FROM S)
```

RELATIONAL ALGEBRA: DIFFERENCE

Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

Syntax: (R – S)

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R – S)

a_id	b_id
a1	101
a2	102

```
(SELECT * FROM R)
EXCEPT
(SELECT * FROM S)
```

RELATIONAL ALGEBRA: PRODUCT

Generate a relation that contains all possible combinations of tuples from the input relations.

Syntax: (R × S)

```
SELECT * FROM R CROSS JOIN S
```

```
SELECT * FROM R, S
```

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R × S)

R.a_id	R.b_id	S.a_id	S.b_id
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

RELATIONAL ALGEBRA: JOIN

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

Syntax: (R ⋈ S)

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R ⋈ S)

a_id	b_id
a3	103

R.a_id	R.b_id	S.a_id	S.b_id
a3	103	a3	103

RELATIONAL ALGEBRA: JOIN

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

Syntax: $(R \bowtie S)$

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

$(R \bowtie S)$

a_id	b_id
a3	103

```
SELECT * FROM R NATURAL JOIN S
```

```
SELECT * FROM R JOIN S USING (a_id, b_id)
```

RELATIONAL ALGEBRA: EXTRA OPERATORS

Rename (ρ)

Assignment ($R \leftarrow S$)

Duplicate Elimination (δ)

Aggregation (γ)

Sorting (τ)

Division ($R \div S$)

EXAMPLE DATABASE

student(<u>sid</u>,name,login,gpa)

sid	name	login	age	gpa
53666	Kanye	kanye@cs	45	4.0
53688	Bieber	jbieber@cs	29	3.9
53655	Tupac	shakur@cs	25	3.5

course(<u>cid</u>,name)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-799	Special Topics in Databases

enrolled(<u>sid</u>,<u>cid</u>,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
53666	15-721	C

AGGREGATES

Functions that return a single value from a bag of tuples:

→ **AVG(col)** → Return the average col value.
→ **MIN(col)** → Return minimum col value.
→ **MAX(col)** → Return maximum col value.
→ **SUM(col)** → Return sum of values in col.
→ **COUNT(col)** → Return # of values for col.

AGGREGATES

Aggregate functions can (almost) only be used in the **SELECT** output list.

Get # of students with a “@cs” login:

```
SELECT COUNT(login) AS cnt
  FROM student WHERE login LIKE '%@cs'
```

AGGREGATES

Aggregate functions can (almost) only be used in the **SELECT** output list.

Get # of students with a “@cs” login:

```
SELECT COUNT(login) AS cnt
```

```
SELECT COUNT(*) AS cnt
  FROM student WHERE login LIKE '%@cs'
```

AGGREGATES

Aggregate functions can (almost) only be used in the **SELECT** output list.

Get # of students with a “@cs” login:

```
SELECT COUNT(login) AS cnt
  F
```

```
SELECT COUNT(*) AS cnt
  F
```

```
SELECT COUNT(1) AS cnt
  FROM student WHERE login LIKE '%@cs'
```

MULTIPLE AGGREGATES

Get the number of students and their average GPA that have a “@cs” login.

```
SELECT AVG(gpa), COUNT(sid)
  FROM student WHERE login LIKE '%@cs'
```

AVG(gpa)	COUNT(sid)
3.8	3

DISTINCT AGGREGATES

COUNT, **SUM**, **AVG** support **DISTINCT**

Get the number of unique students that have an "@cs" login.

```
SELECT COUNT(DISTINCT login)
  FROM student WHERE login LIKE '%@cs'
```

COUNT(DISTINCT login)
3

AGGREGATES

Output of other columns outside of an aggregate is undefined.

Get the average GPA of students enrolled in each course.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.86	???

GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
 GROUP BY e.cid
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	15-721
53439	53439	2.70	15-721
56023	56023	2.75	15-826
59439	59439	3.90	15-826
53961	53961	3.50	15-826
58345	58345	1.89	15-445

AVG(s.gpa)	e.cid
2.46	15-721
3.39	15-826
1.89	15-445

GROUP BY

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause.

```
SELECT AVG(s.gpa), e.cid, s.name
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
```

GROUP BY

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause.

```
SELECT AVG(s.gpa), e.cid, s.name
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
 GROUP BY e.cid, s.name
```

HAVING

Filters results based on aggregation, like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
   AND avg_gpa > 3.9
 GROUP BY e.cid
```

HAVING

Filters results based on aggregation, like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
 HAVING avg_gpa > 3.9;
```

HAVING

Filters results based on aggregation, like a **WHERE** clause for a **GROUP BY**

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
 HAVING AVG(s.gpa) > 3.9;
```

AVG(s.gpa)	e.cid
3.75	15-415
3.950000	15-721
3.900000	15-826

→

avg_gpa	e.cid
3.950000	15-721

OUTPUT CONTROL

ORDER BY <column*> [ASC | DESC]

→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
 WHERE cid = '15-721'
 ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

OUTPUT CONTROL

ORDER BY <column*> [ASC | DESC]

→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
 W
 O
```

```
SELECT sid, grade FROM enrolled
 WHERE cid = '15-721'
 ORDER BY 1
```

```
SELECT sid FROM enrolled
 WHERE cid = '15-721'
 ORDER BY grade DESC, sid ASC
```

sid
53666
53650
53123
53334

OUTPUT CONTROL

ORDER BY <column*> [ASC | DESC]

→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
 W
 O
```

```
SELECT sid, grade FROM enrolled
 WHERE cid = '15-721'
 ORDER BY 1
```

```
SELECT sid FROM enrolled
 W
 O
```

```
SELECT sid FROM enrolled
 WHERE cid = '15-721'
 ORDER BY grade DESC, 1 ASC
```

OUTPUT CONTROL

LIMIT <count> [offset]

→ Limit the # of tuples returned in output.
→ Can set an offset to return a "range"

```
SELECT sid, name FROM student
 WHERE login LIKE '%@cs'
 LIMIT 10
```

```
SELECT sid, name FROM student
 WHERE login LIKE '%@cs'
 LIMIT 20 OFFSET 10
```

NESTED QUERIES

Queries containing other queries.

They are often difficult to optimize.

Inner queries can appear (almost) anywhere in the query.

Outer Query

```
SELECT name FROM student
WHERE
 sid IN (SELECT sid FROM enrolled)
```

Inner Query

NESTED QUERIES

ALL → Must satisfy expression for all rows in the sub-query.

ANY → Must satisfy expression for at least one row in the sub-query.

IN → Equivalent to '**=ANY()**' .

EXISTS → At least one row is returned without comparing it to an attribute in outer query.

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
 WHERE sid IN (
   SELECT sid FROM enrolled
    WHERE cid = '15-445'
 )
```

NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
 WHERE sid = ANY (
   SELECT sid FROM enrolled
    WHERE cid = '15-445'
)
```

NESTED QUERIES

Find all courses that have no students enrolled in them.

```
SELECT * FROM course
 WHERE ...
```

"with no tuples in the enrolled table"

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-799	Special Topics in Databases

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53655	15-445	B
53666	15-721	C

NESTED QUERIES

Find all courses that have no students enrolled in them.

```
SELECT * FROM course
 WHERE NOT EXISTS (
    tuples in the enrolled table
)
```

NESTED QUERIES

Find all courses that have no students enrolled in them.

```
SELECT * FROM course
 WHERE NOT EXISTS (
   SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

cid	name
15-799	Special Topics in Databases

CONCLUSION

Databases are ubiquitous.

Relational algebra defines the primitives for processing queries on a relational database.

We will see relational algebra again when we talk about query optimization + execution.

SQL is not a dead language.

NEXT CLASS

Disk Storage