DATA ANALYTICS USING DEEP LEARNING

GT 8803 // FALL 2019 // JOY ARULRAJ

LECTURE #05: INTRODUCTION TO DATABASE SYSTEMS AND ADVANCED SQL

Georgia Tech

CREATING THE NEXT®

ADMINISTRIVIA

- Assignment 1
  - Due on Sep 18
  - Focuses on topics covered in first four lectures
- Project ideas
  - Share a list next week
  - Start looking for team-mates!

LAST CLASS

- Introduction to neural networks
  - Non-linear activation functions
  - Computational graphs
  - Backpropagation
- Image classification
  - Classification function, Loss function, Optimization
  - KNN, Linear Classifier, Neural networks, etc.

TODAY'S AGENDA

- Introduction to database systems
- Advanced SQL

DATABASE SYSTEMS

DATABASE

- Organized collection of inter-related data that models some aspect of the real-world.
- Databases are core the component of most computer applications.

DATABASE EXAMPLE

- Create a database that models a digital music store to keep track of artists and albums.
- Things we need store:
  - Information about Artists
  - What Albums those Artists released

FLAT FILE STRAWMAN

- Store our database as comma-separated value (CSV) files that we manage in our own code.
  - Use a separate file per entity.
  - The application has to parse the files each time they 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"

"Ice Cube",1989,"USA"

Album(name, artist, year)

"Enter the Wu Tang","Wu Tang Clan",1993

"St.Ides Mix Tape","Wu Tang Clan",1994

"AmeriKKKa's Most Wanted","Ice Cube",1990

FLAT FILE STRAWMAN

- Example: Get the year that Ice Cube went solo.

Artist(name, year, country)

```
"Wu Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"Ice Cube",1989,"USA"
```

FLAT FILE STRAWMAN

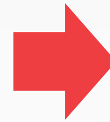
- Example: Get the year that Ice Cube went solo.

Artist(name, year, country)

```
"Wu Tang Clan",1992,"USA"

"Notorious BIG",1992,"USA"

"Ice Cube",1989,"USA"
```



```
for line in file:
  record = parse(line)
  if "Ice Cube" == record[0]:
    print int(record[1])
```

FLAT FILES: DATA INTEGRITY

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?
- How do we store that there are multiple artists on an album?

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?
- How do we store that there are multiple artists on an album?

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?
- How do we store that there are multiple artists on an album?

FLAT FILES: IMPLEMENTATION

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: 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: 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

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?

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 **DBMS** is software that allows applications to store and analyze information in a database.
- A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.

HISTORY REPEATS ITSELF

- 1960s: Hierarchical & Network Data Models
- 1970s: Relational Data Model
- 1980s: Object-Oriented Databases
- 2000s: Data warehouses
- 2010s: NewSQL, Hybrid, and Cloud Systems

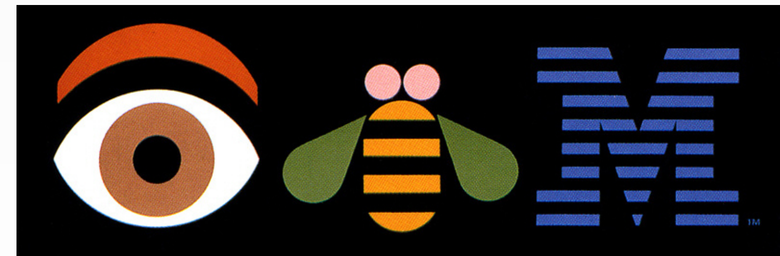
WHAT GOES AROUND COMES AROUND
Readings in DB Systems, 4th Edition, 2006.

HISTORY REPEATS ITSELF

- Old database issues are still relevant today.
- The "SQL vs. NoSQL" debate is reminiscent of "Relational vs. CODASYL" debate.

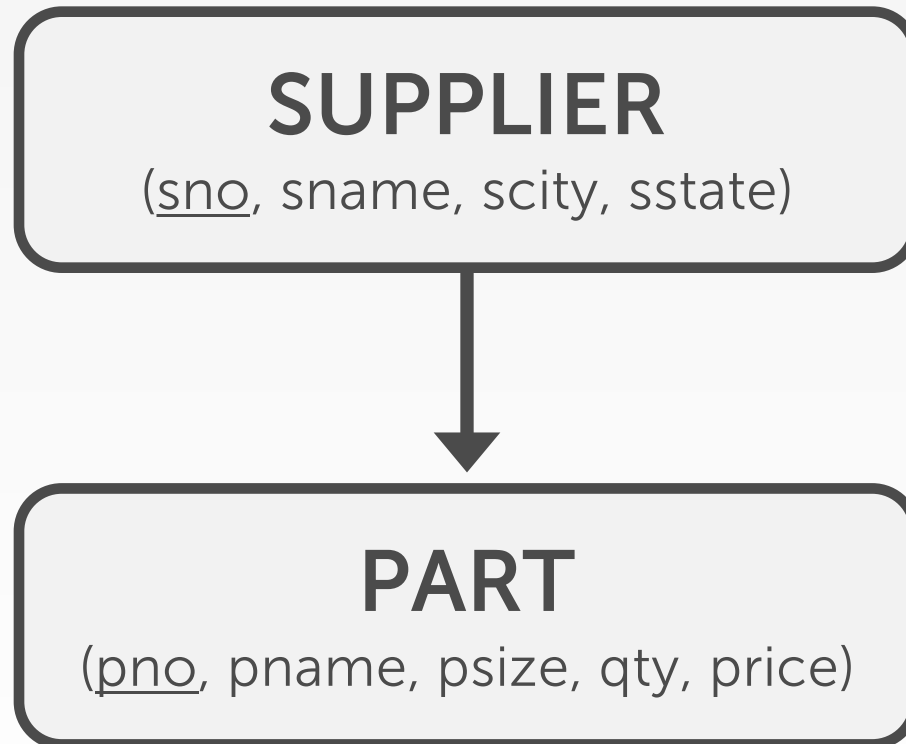
1960S – IBM IMS

- Information Management System
- Early DBMS developed to keep track of purchase orders for Apollo moon mission.
  - Hierarchical data model
  - Programmer-defined physical storage format
  - Tuple-at-a-time queries



HIERARCHICAL DATA MODEL

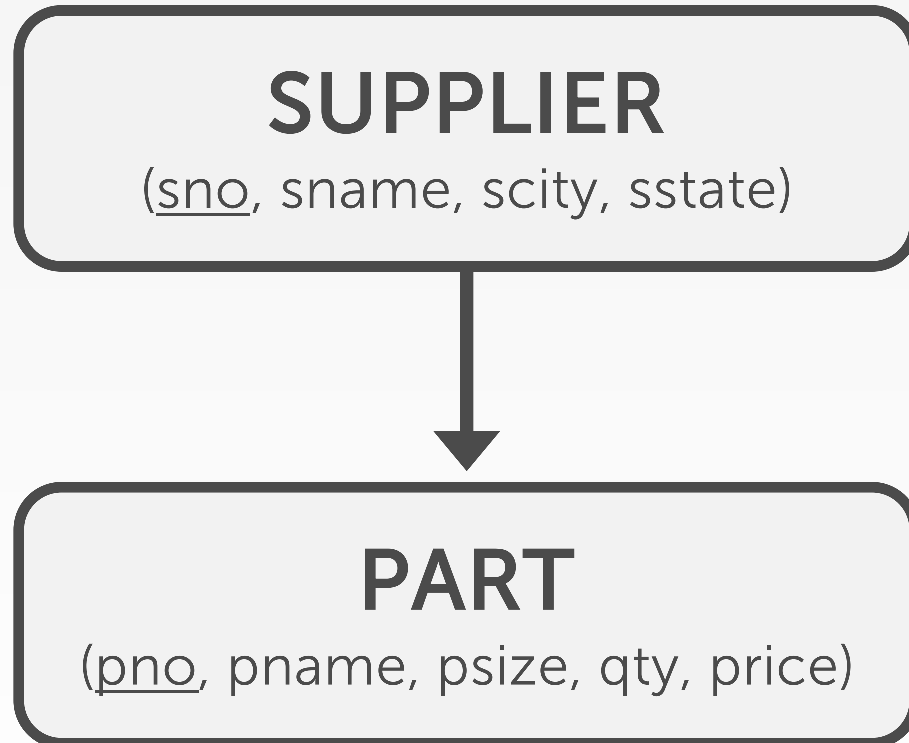
Schema



Instance

HIERARCHICAL DATA MODEL

Schema

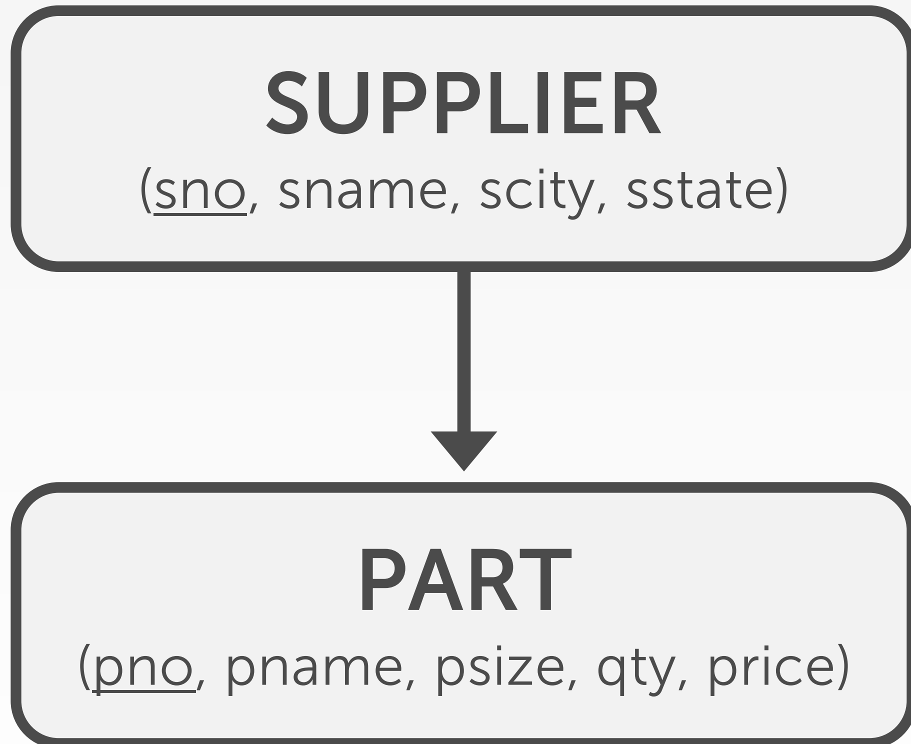


Instance

sno	sname	scity	sstate	parts
1001	Dirty Rick	New York	NY	
1002	Squirrels	Boston	MA	

HIERARCHICAL DATA MODEL

Schema



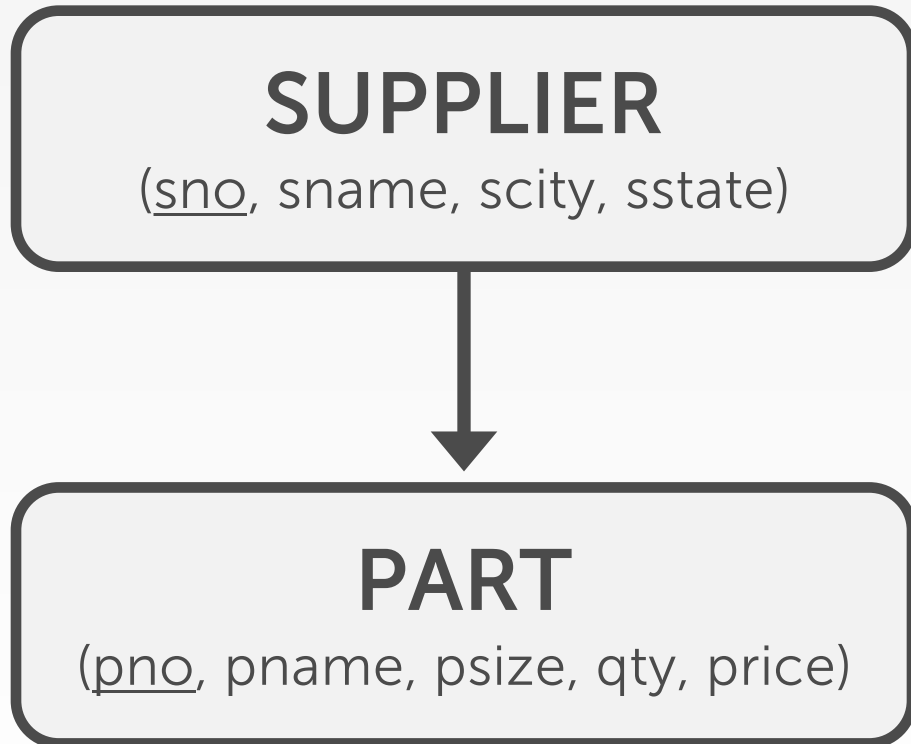
Instance

sno	sname	scity	sstate	parts
1001	Dirty Rick	New York	NY	
1002	Squirrels	Boston	MA	

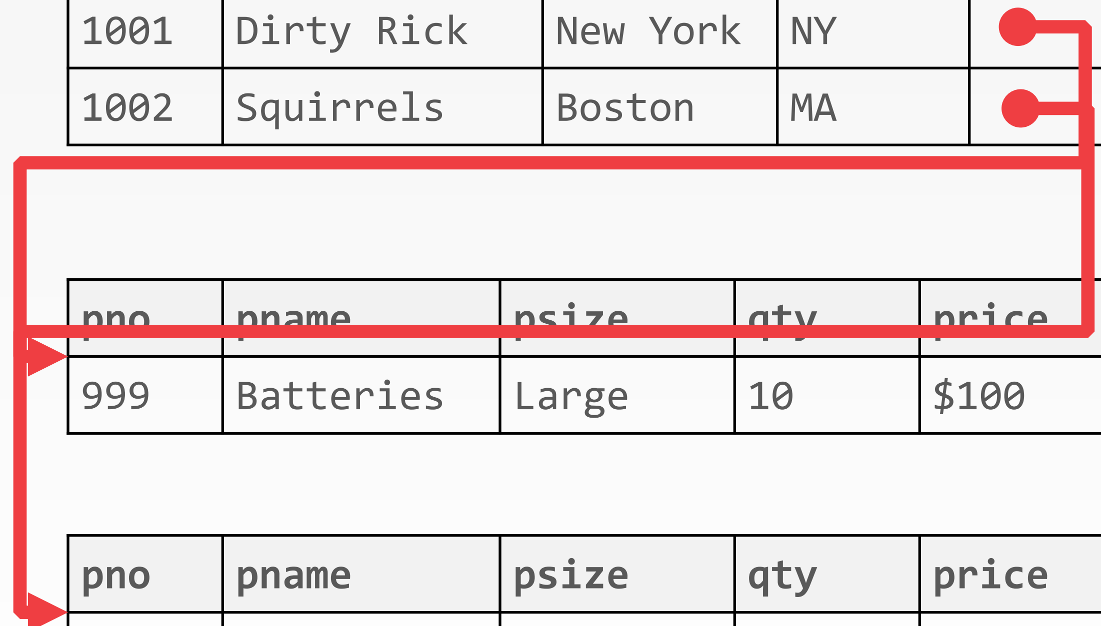
pno	pname	psize	qty	price
999	Batteries	Large	10	$100

HIERARCHICAL DATA MODEL

Schema



Instance



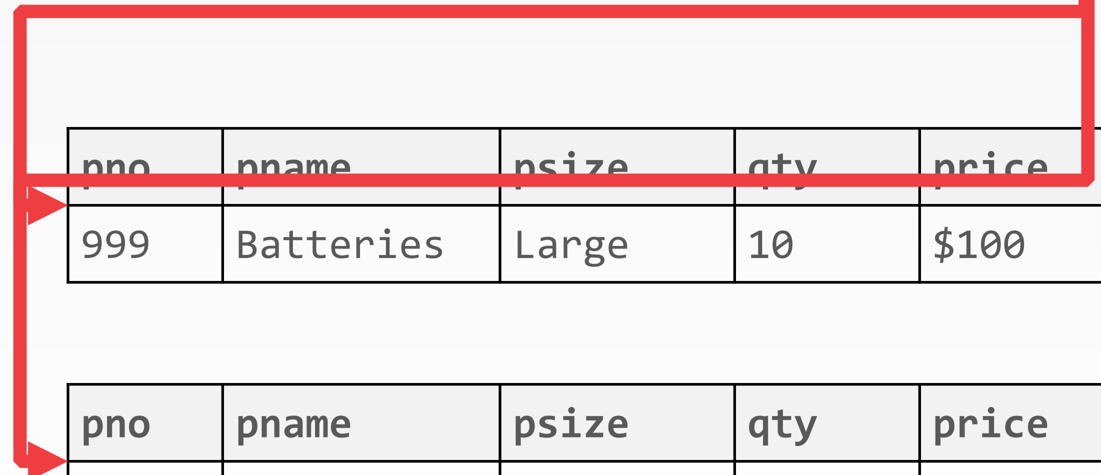
HIERARCHICAL DATA MODEL

Duplicate Data

(sno, sname, scity, sstate)

PART

(pno, pname, psize, qty, price)



				parts
1002	Squirrels	Boston	MA	

pno	pname	psize	qty	price
999	Batteries	Large	10	$100

pno	pname	psize	qty	price
999	Batteries	Large	14	$99

HIERARCHICAL DATA MODEL

Schema *Instance*

Duplicate Data

(sno, sname, scity, sstate)

				parts
1002	Squirrels	Boston	MA	

Data Dependencies

	price
	$100

PART

(pno, pname, psize, qty, price)

pno	pname	psize	qty	price
999	Batteries	Large	14	$99

HIERARCHICAL DATA MODEL

- Advantages
  - **No need to reinvent the wheel** for every application
  - **Logical data independence:** New record types may be added as the logical requirements of an application may change over time.

HIERARCHICAL DATA MODEL

- Limitations
  - **Tree structured data models** are very restrictive
  - **No physical data independence**: Cannot freely change storage organizations to tune a database application because there is no guarantee that the applications will continue to run
  - **Optimization:** A tuple-at-a-time user interface forces the programmer to do manual query optimization, and this is often hard

1960s – IDS

- Integrated Data Store
- Developed internally at GE in the early 1960s.
- GE sold their computing division to Honeywell in 1969.
- One of the first DBMSs:
  - Network data model
  - Tuple-at-a-time queries

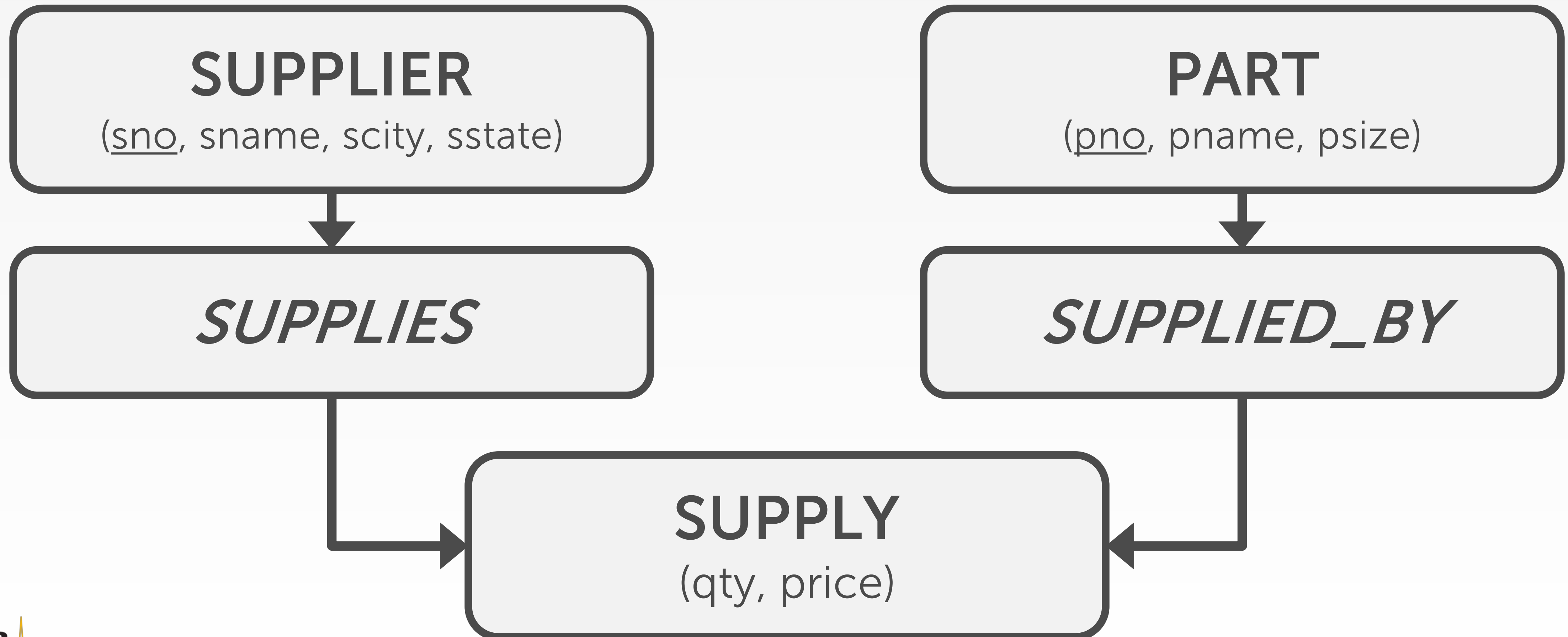
1960s – CODASYL

- COBOL people got together and proposed a standard for how programs will access a database. Lead by Charles Bachman.
  - Network data model.
  - Tuple-at-a-time queries.

Bachman

NETWORK DATA MODEL

Schema



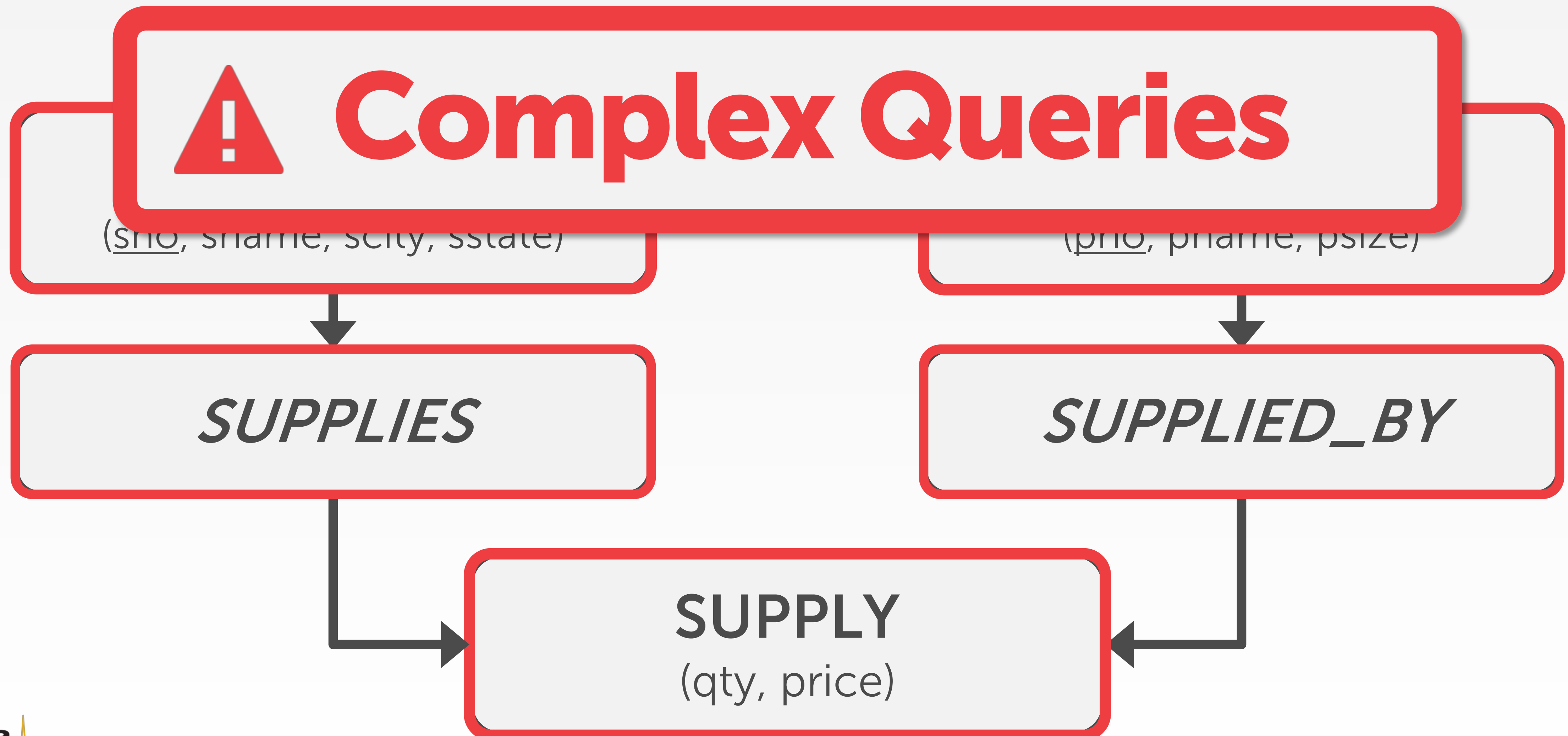
NETWORK DATA MODEL

Schema

NETWORK DATA MODEL

Schema

NETWORK DATA MODEL



NETWORK DATA MODEL

Complex Queries

(sno, sname, scity, sstate)

(pno, pname, psize)

Easily Corrupted

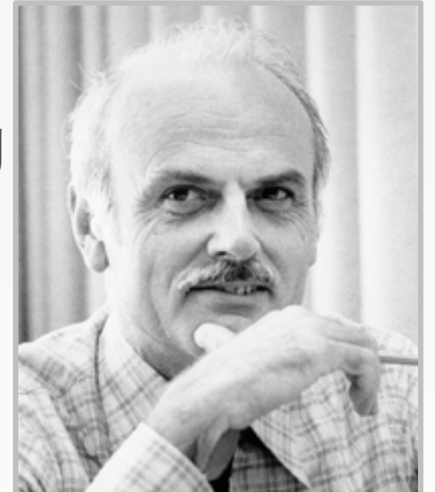
SUPPLY
(qty, price)

NETWORK DATA MODEL

- Advantages
  - **Graph structured data models** are less restrictive
- Limitations
  - **Poorer physical and logical data independence**: Cannot freely change <u>physical data storage organization</u> or change <u>logical application schema</u>
  - **Slow loading and recovery:** Data is typically stored in one large network. This much larger object had to be bulk-loaded all at once, leading to very long load times.

1970s – RELATIONAL MODEL

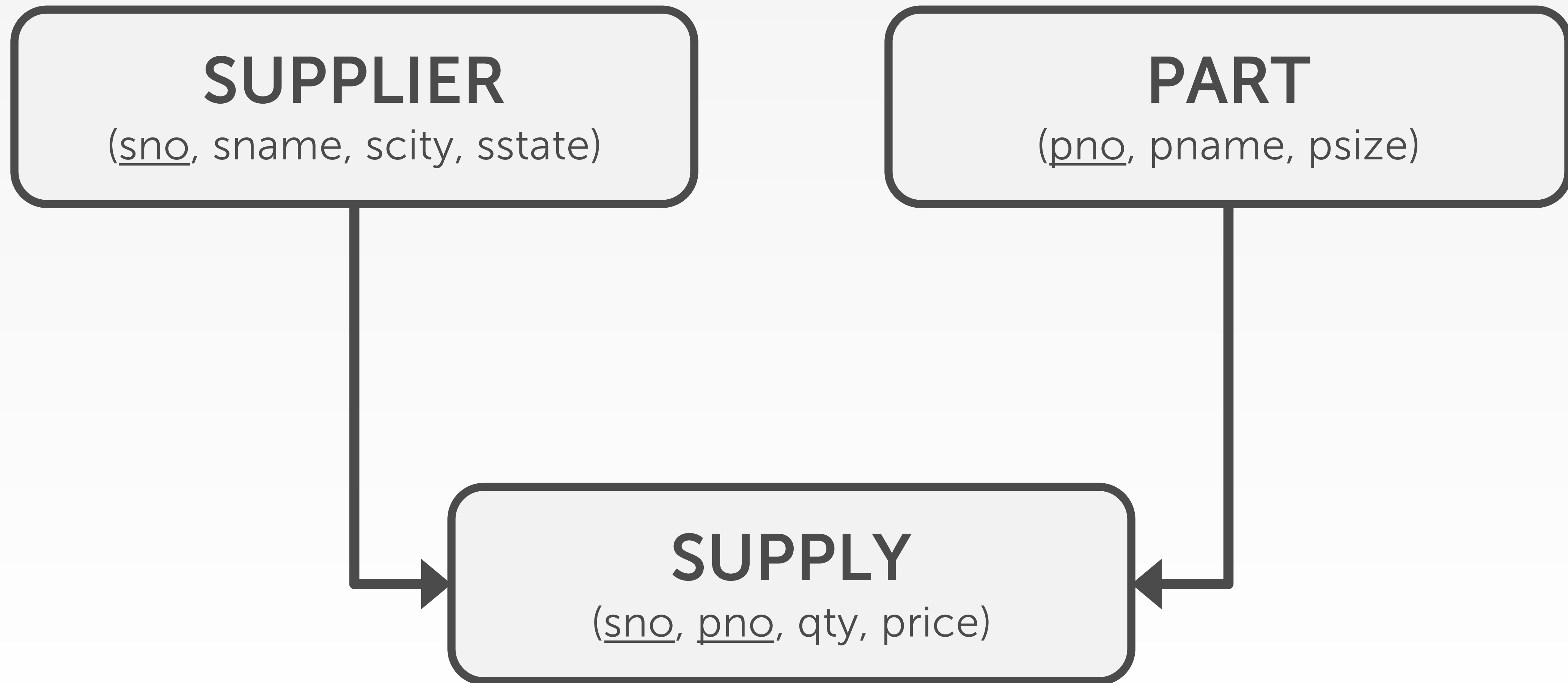
- Ted Codd was a mathematician working at IBM Research.
  - He saw developers spending their time rewriting IMS and Codasyl programs every time the database's schema or layout changed.
- Relational abstraction to avoid this:
  - Store database in simple data structures.
  - Access data via high-level declarative language.
  - Physical storage left up to implementation.



Codd

RELATIONAL DATA MODEL

Schema



RELATIONAL DATA MODEL

Schema

RELATIONAL DATA MODEL

Schema

RELATIONAL DATA MODEL

Schema

A Relational Model of Data for Large Shared Data Banks

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

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

1. Relational Model and Normal Form

1.1. Introduction

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for non-inferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. Data Dependencies in Present Systems

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

Georgia Tech

RELATIONAL DATA MODEL

- Advantages
  - **Set-a-time languages** are good, regardless of the data model, since they offer physical data independence
  - **Logical data independence** is easier with a simple data model than with a complex one.
  - **Query optimizers** can beat all but the best tuple-at-a-time DBMS application programmers

RELATIONAL DATA MODEL

- Early implementations of relational DBMS:
  - **System R** – IBM Research
  - **INGRES** – U.C. Berkeley
  - **Oracle** – Larry Ellison

Gray

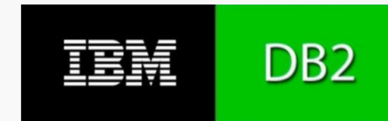
Stonebraker

Ellison

RELATIONAL DATA MODEL

- The relational model wins.
  - IBM comes out with DB2 in 1983.
  - “SEQUEL” becomes the standard (SQL).

- Many new “enterprise” DBMSs but Oracle wins marketplace.



ORACLE

Informix

TANDEM

SYBASE

TERADATA

INGRES

InterBase

1980s – OBJECT-ORIENTED DATABASES

- Avoid "relational-object impedance mismatch" by tightly coupling objects and database.
- Few of these original DBMSs from the 1980s still exist today but many of the technologies exist in other forms (JSON, XML)

OBJECT-ORIENTED MODEL

Application Code

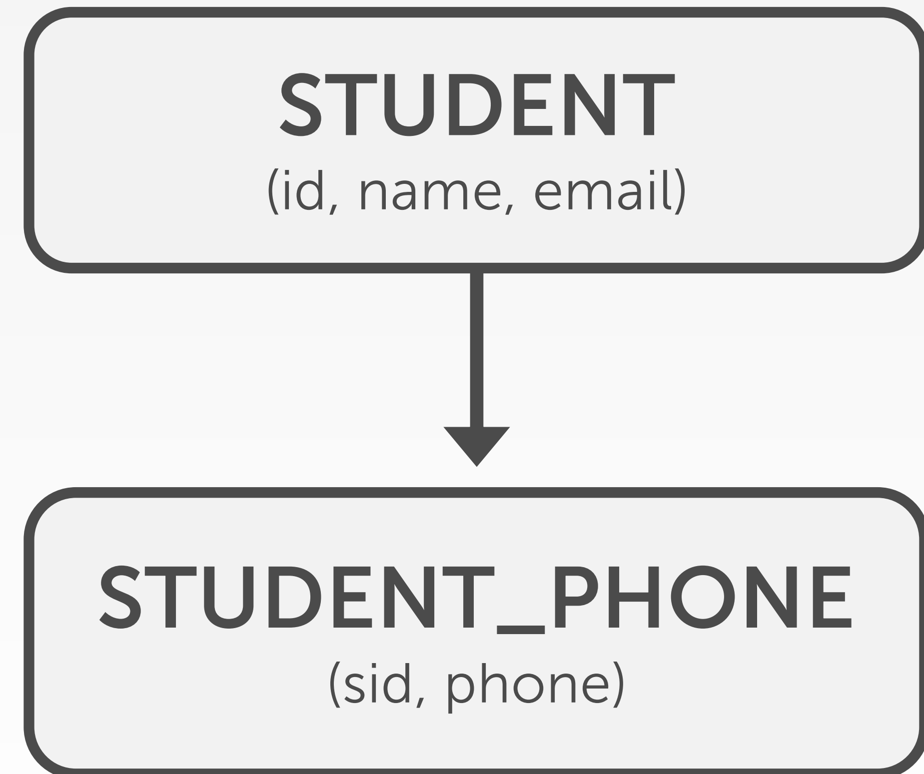
```
class Student {
   int id;
   String name;
   String email;
   String phone[];
}
```

OBJECT-ORIENTED MODEL

Application Code

```
class Student {
   int id;
   String name;
   String email;
   String phone[];
}
```

Relational Schema



OBJECT-ORIENTED MODEL

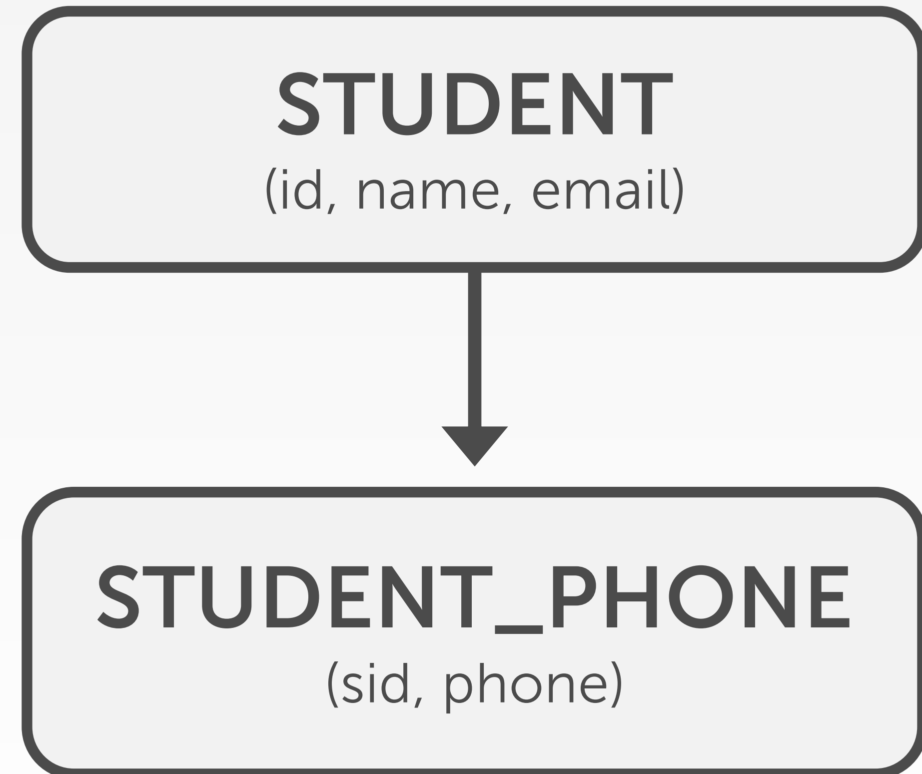
Application Code

```
class Student {
    int id;
    String name;
    String email;
    String phone[];
}
```

id	name	email
1001	M.O.P.	ante@up.com

sid	phone
1001	444-444-4444
1001	555-555-5555

Relational Schema



OBJECT-ORIENTED MODEL

Application Code

```
class Student {
   int id;
   String name;
   String email;
   String phone[];
}
```

id	name	email
1001	M.O.P.	ante@up.com

sid	phone
1001	444-444-4444
1001	555-555-5555

Relational Schema

STUDENT
(id, name, email)

STUDENT_PHONE
(sid, phone)

OBJECT-ORIENTED MODEL

Application Code

```
class Student {
   int id;
   String name;
   String email;
   String phone[];
}
```

OBJECT-ORIENTED MODEL

Application Code

```
class Student {
    int id;
    String name;
    String email;
    String phone[];
}
```

```
Student
{
  "id": 1001,
  "name": "M.O.P.",
  "email": "ante@up.com",
  "phone": [
      "444-444-4444",
      "555-555-5555"
   ]
}
```

OBJECT-ORIENTED MODEL

Complex Queries

```
c
    String email;
    String phone[];
}
```

```
  "email": "ante@up.com",
  "phone": [
      "444-444-4444",
      "555-555-5555"
   ]
}
```

OBJECT-ORIENTED MODEL

Complex Queries

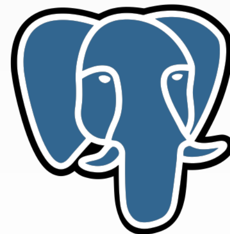
No Standard API

```
String email;
String phone[];
}
```

```
"email": "ante@up.com",
```

1990s – BORING DAYS

- No major advancements in database systems or application workloads.
  - Microsoft forks Sybase and creates SQL Server.
  - MySQL is written as a replacement for mSQL.
  - Postgres gets SQL support.
  - SQLite started in early 2000.



1990s – BORING DAYS

- Multimedia databases
  - Feature engineering
  - Accuracy, robustness, and performance

```
SELECT image_date
 FROM images
 WHERE event = 'Sunrise'
```

2000s – INTERNET BOOM

- All the big players were heavyweight and expensive. Open-source databases were missing important features.
- Many companies wrote their own custom middleware to scale out database across single-node DBMS instances.

2000s – DATA WAREHOUSES

- Rise of the special purpose OLAP DBMSs.
  - Distributed / Shared-Nothing
  - Relational / SQL
  - Usually closed-source.
- Significant performance benefits from using columnar storage organization

2000s – NOSQL SYSTEMS

- Focus on high-availability & high-scalability:
  - Schemaless (i.e., "Schema Last")
  - Non-relational data models (document, key/value, etc.)
  - No ACID transactions
  - Custom APIs instead of SQL
  - Usually open-source

APACHE HBASE
amazon DynamoDB
neo4j
mongoDB
redis
Couchbase
cassandra
AEROSPIKE
riak
CouchDB relax
NOSQL

2010s – NEWSQL SYSTEMS

- Provide same performance for OLTP workloads as NoSQL DBMSs without giving up ACID:
  - Relational / SQL
  - Distributed
  - Usually closed-source

2010s – HYBRID SYSTEMS

- **Hybrid Transactional-Analytical Processing**
- Execute fast OLTP like a NewSQL system while also executing complex OLAP queries like a data warehouse system.
  - Distributed / Shared-Nothing
  - Relational / SQL
  - Mixed open/closed-source.

2010s – CLOUD SYSTEMS

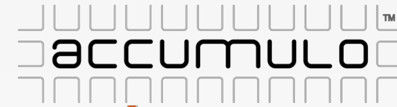
- First database-as-a-service (DBaaS) offerings were "containerized" versions of existing DBMSs.
- There are new DBMSs that are designed from scratch explicitly for running in a cloud environment.

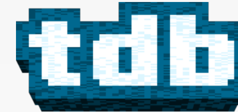
2010s – SPECIALIZED SYSTEMS

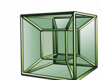
- Shared-disk DBMSs
- Embedded DBMSs
- Times Series DBMS
- Multi-Model DBMSs
- Blockchain DBMSs

2010s – SPECIALIZED SYSTEMS

- Shared-disk DBMSs
- Embedded DBMSs
- Times Series DBMS
- Multi-Model DBMSs
- Blockchain DBMSs







SUMMARY

- There are many innovations that come from both industry and academia:
  - Lots of ideas start in academia but few build complete DBMSs to verify them.
  - IBM was the vanguard during 1970-1980s but now there is no single trendsetter.
  - Oracle borrows ideas from anybody.
- The relational model has won for operational databases.

GOAL: VIDEO ANALYTICS DBMS

- Feature Engineering
- Robustness
- Computational Efficiency
- Usability

CHALLENGES: MULTIMEDIA DBMSs

- Feature Engineering
  - The same multi-media data could mean different things to different people. Second, users typically have diverse information needs.
  - Thus, a single feature may not be sufficient to completely index a given video.
  - Therefore, it becomes difficult to identify the features that are most appropriate in any given environment.

CHALLENGES: MULTIMEDIA DBMSs

- Robustness
  - Works well on one dataset, but completely breaks on another dataset from the same domain
  - Example: Two traffic cameras in different cities
  - Limits the utility of the database system
  - Need inherent support for coping with data drift

CHALLENGES: COMPUTER VISION PIPELINES

- Computational Efficiency
  - These pipelines are computationally infeasible at scale
  - Example: State-of-the-art object detection models run at 3 frames per second (fps) (e.g., Mask R-CNN)
  - It will take 8 decades of GPU time to process 100 cameras over a month of video

CHALLENGES: COMPUTER VISION PIPELINES

- Usability
  - These techniques require complex, imperative programming across many low-level libraries (e.g., Pytorch and OpenCV)
  - This is an ad-hoc, tedious process that ignores opportunity for cross-operator optimization
  - Traditional database systems were successful due to their ease of use (i.e., SQL is declarative)

GOAL: VIDEO ANALYTICS DBMS

ADVANCED SQL

RELATIONAL LANGUAGES

- User only needs to specify **what** answer that they want, not **how** to compute it.
- The DBMS is responsible for efficient evaluation of the query.
  - Query optimizer: re-orders operations and generates query plan

SQL HISTORY

- Originally "SEQUEL" from IBM's **System R** prototype.
  - Structured English Query Language
  - Adopted by Oracle in the 1970s.
- IBM releases DB2 in 1983.
- ANSI Standard in 1986. ISO in 1987
  - Structured Query Language

SQL HISTORY

- Current standard is **SQL:2016**
  - **SQL:2016** → JSON, Polymorphic tables
  - **SQL:2011** → Temporal DBs, Pipelined DML
  - **SQL:2008** → TRUNCATE, Fancy ORDER
  - **SQL:2003** → XML, windows, sequences, auto-generated IDs.
  - **SQL:1999** → Regex, triggers, OO
- Most DBMSs at least support **SQL-92**

RELATIONAL LANGUAGES

- Language
  - Data Manipulation Language (DML)
  - Data Definition Language (DDL)
  - Data Control Language (DCL)
  - View definition
  - Integrity & Referential Constraints
  - Transactions
- Important: SQL is based on **bags** (duplicates) not **sets** (no duplicates).

ADVANCED SQL

- Aggregations + Group By
- Output Control + Redirection
- Nested Queries
- Common Table Expressions
- Window Functions

EXAMPLE DATABASE

student(sid,name,login,gpa)

sid	name	login	age	gpa
53666	Kanye	kayne@cs	39	4.0
53688	Bieber	jbieber@cs	22	3.9
53655	Tupac	shakur@cs	26	3.5

course(cid,name)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

enrolled(sid,cid,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 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 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 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'
```

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

AGGREGATES

- Aggregate functions can 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'
```

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

```
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'
```

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)
3.25	12

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'
```

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)
10

Georgia Tech

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, student AS s
 WHERE e.sid = s.sid
```

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, student AS s
WHERE e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.5	???

GROUP BY

- Project tuples into subsets and calculate aggregates against each subset.

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

GROUP BY

- Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE 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

GROUP BY

- Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
 FROM enrolled AS e, student AS s
WHERE 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

GROUP BY

- Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e, student AS s
 WHERE 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

- Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
 FROM enrolled AS e, student AS s
WHERE 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, 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, 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, student AS s
 WHERE 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
```

Georgia Tech

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
  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_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 REDIRECTION

- Store query results in another table:
  - Table must not already be defined.
  - Table will have the same # of columns with the same types as the input.

SQL-92

```
SELECT DISTINCT cid INTO CourseIds
  FROM enrolled;
```

MySQL

```
CREATE TABLE CourseIds (
SELECT DISTINCT cid FROM enrolled);
```

OUTPUT REDIRECTION

- Insert tuples from query into another table:
  - Inner **SELECT** must generate the same columns as the target table.
  - DBMSs have different options/syntax on what to do with duplicates.

SQL-92

```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM enrolled);
```

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
```

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
 WHERE cid = '15-721'
 ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

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

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

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

sid
53666
53650
53123
53334

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
```

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 query.

NESTED QUERIES

- Queries containing other queries.
- They are often difficult to optimize.

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

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

NESTED QUERIES

- Queries containing other queries.
- They are often difficult to optimize.
- Inner queries can appear (almost) anywhere in query.

Outer Query →

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

← Inner Query

NESTED QUERIES

- Queries containing other queries.
- They are often difficult to optimize.

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

Outer Query →

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

← *Inner Query*

NESTED QUERIES

- *Get the names of students in '15-445'*

```
SELECT name FROM student
 WHERE ...
```

"sid in the set of people that take 15-445"

NESTED QUERIES

- *Get the names of students in '15-445'*

```
SELECT name FROM student
 WHERE ...
  SELECT sid FROM enrolled
   WHERE cid = '15-445'
```

NESTED QUERIES

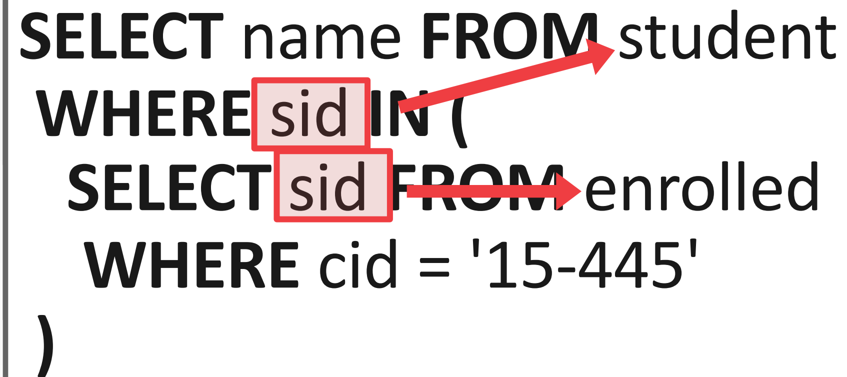
- *Get the names of students in '15-445'*

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

Georgia Tech

NESTED QUERIES

- *Get the names of students in '15-445'*



NESTED QUERIES

- **ANY**→ Must satisfy expression for at least one row in sub-query.
- **IN**→ Equivalent to '**=ANY()**' .
- **EXISTS**→ At least one row is returned.
- **ALL**→ Must satisfy expression for all rows in sub-query

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

- *Get the names of students in '15-445'*

```
SELECT (SELECT S.name FROM student AS S
        WHERE S.sid = E.sid) AS sname
 FROM enrolled AS E
WHERE cid = '15-445'
```

NESTED QUERIES

NESTED QUERIES

```
SELECT MAX(e.sid), s.name
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid;
```

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT MAX(e.sid), s.name
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid;
```

- Won't work in SQL-92.

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT sid, name FROM student
 WHERE ...
```

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT sid, name FROM student
 WHERE ...
```

"Is greater than every other sid"

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT sid, name FROM student
 WHERE sid   is greater than every
  SELECT sid FROM enrolled
```

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT sid, name FROM student
 WHERE sid => ALL(
  SELECT sid FROM enrolled
)
```

sid	name
53688	Bieber

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT sid, name FROM student
 WHE
  SEL
)
```

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

NESTED QUERIES

- *Find student record with the highest id that is enrolled in at least one course.*

SELECT sid, name FROM student

SELECT sid, name FROM student

```
SELECT sid, name FROM student
 WHERE sid IN (
  SELECT sid FROM enrolled
   ORDER BY sid DESC LIMIT 1
)
```

NESTED QUERIES

- *Find all courses that has no students enrolled in it.*

```
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
15-823	Advanced 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 has no students enrolled in it.*

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

NESTED QUERIES

- *Find all courses that has no students enrolled in it.*

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

cid	name
15-823	Advanced Topics in Databases

NESTED QUERIES

- *Find all courses that has no students enrolled in it.*

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

cid	name
15-823	Advanced Topics in Databases

WINDOW FUNCTIONS

- Performs a "sliding" calculation across a set of tuples that are related.
- Like an aggregation but tuples are not grouped into a single output tuples.

```
SELECT ... FUNC-NAME(...) OVER (...)
  FROM tableName
```

WINDOW FUNCTIONS

- Performs a "sliding" calculation across a set of tuples that are related.
- Like an aggregation but tuples are not grouped into a single output tuples.

```
SELECT ... FUNC-NAME(...) OVER (...)
  FROM tableName
```

Aggregation Functions
Special Functions

WINDOW FUNCTIONS

- Performs a "sliding" calculation across a set of tuples that are related.
- Like an aggregation but tuples are not grouped into a single output tuples.

How to "slice" up data
Can also sort

```
SELECT ... FUNC-NAME(...) OVER (...)
  FROM tableName
```

Aggregation Functions
Special Functions

WINDOW FUNCTIONS

- Aggregation functions:
  - Anything that we discussed earlier
- Special window functions:
  - **ROW_NUMBER()**→ # of the current row
  - **RANK()**→ Order position of the current row.

```
SELECT *, ROW_NUMBER() OVER () AS row_num
  FROM enrolled
```

WINDOW FUNCTIONS

- Aggregation functions:
  - Anything that we discussed earlier
- Special window functions:
  - **ROW_NUMBER()**→ # of the current row
  - **RANK()**→ Order position of the current row.

sid	cid	grade	row_num
53666	15-445	C	1
53688	15-721	A	2
53688	15-826	B	3
53655	15-445	B	4
53666	15-721	C	5

```
SELECT *, ROW_NUMBER() OVER () AS row_num
  FROM enrolled
```

WINDOW FUNCTIONS

- Aggregation functions:
  - Anything that we discussed earlier
- Special window functions:
  - **ROW_NUMBER()**→ # of the current row
  - **RANK()**→ Order position of the current row.

sid	cid	grade	row_num
53666	15-445	C	1
53688	15-721	A	2
53688	15-826	B	3
53655	15-445	B	4
53666	15-721	C	5

```
SELECT *, ROW_NUMBER() OVER () AS row_num
  FROM enrolled
```

WINDOW FUNCTIONS

- The **OVER** keyword specifies how to group together tuples when computing the window function.
- Use **PARTITION BY** to specify group.

```
SELECT cid, sid,
   ROW_NUMBER() OVER (PARTITION BY cid)
 FROM enrolled
ORDER BY cid
```

WINDOW FUNCTIONS

- The **OVER** keyword specifies how to group together tuples when computing the window function.
- Use **PARTITION BY** to specify group.

cid	sid	row_number
15-445	53666	1
15-445	53655	2
15-721	53688	1
15-721	53666	2
15-826	53688	1

```
SELECT cid, sid,
   ROW_NUMBER() OVER (PARTITION BY cid)
 FROM enrolled
 ORDER BY cid
```

WINDOW FUNCTIONS

- The **OVER** keyword specifies how to group together tuples when computin[g the window] function.
- Use **PARTITION BY** to specify gro[up]

cid	sid	row_number
15-445	53666	1
15-445	53655	2
15-721	53688	1
15-721	53666	2
15-826	53688	1

```
SELECT cid, sid,
   ROW_NUMBER() OVER (PARTITION BY cid)
 FROM enrolled
 ORDER BY cid
```

WINDOW FUNCTIONS

- You can also include an **ORDER BY** in the window grouping to sort entries in each group.

```
SELECT *,
   ROW_NUMBER() OVER (ORDER BY cid)
 FROM enrolled
ORDER BY cid
```

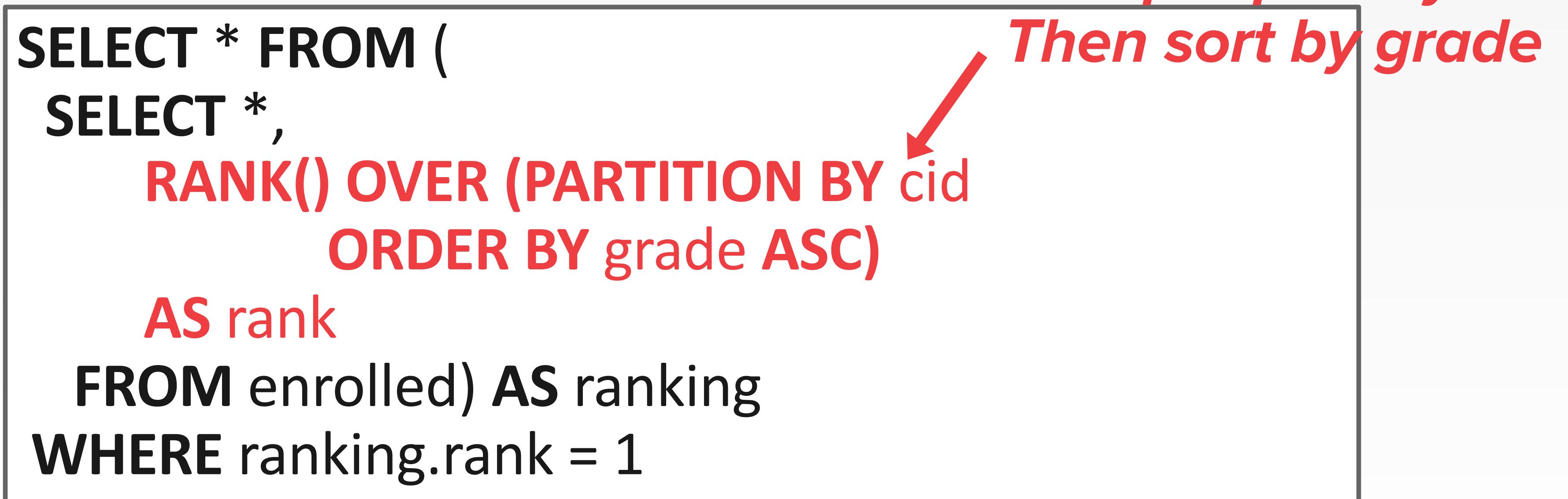
WINDOW FUNCTIONS

- *Find the student with the highest grade for each course.*

```
SELECT * FROM (
 SELECT *,
       RANK() OVER (PARTITION BY cid
                    ORDER BY grade ASC)
       AS rank
  FROM enrolled) AS ranking
 WHERE ranking.rank = 1
```

WINDOW FUNCTIONS

- *Find the student with the highest grade for each course.*



WINDOW FUNCTIONS

- *Find the student with the highest grade for each course.*

Group tuples by cid
Then sort by grade

```
SELECT * FROM (
 SELECT *,
     RANK() OVER (PARTITION BY cid
            ORDER BY grade ASC)
     AS rank
  FROM enrolled) AS ranking
 WHERE ranking.rank = 1
```

COMMON TABLE EXPRESSIONS

- Provides a way to write auxiliary statements for use in a larger query.
  - Improves readability by decomposing the task
  - Think of it like a temp table just for one query.
- Alternative to nested queries and views.

```
WITH cteName AS (
  SELECT 1
)
SELECT * FROM cteName
```

COMMON TABLE EXPRESSIONS

- Provides a way to write auxiliary statements for use in a larger query.
  - Improves readability by decomposing the task
  - Think of it like a temp table just for one query.
- Alternative to nested queries and views.

```
WITH cteName AS (
  SELECT 1
)
SELECT * FROM cteName
```

COMMON TABLE EXPRESSIONS

- You can bind output columns to names before the **AS** keyword.

```
WITH cteName (col1, col2) AS (
  SELECT 1, 2
)
SELECT col1 + col2 FROM cteName
```

COMMON TABLE EXPRESSIONS

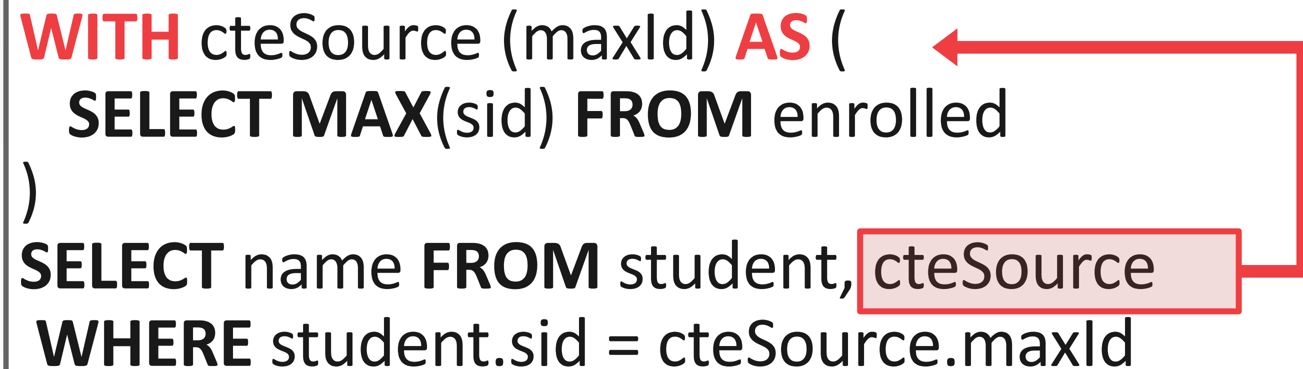
- *Find student record with the highest id that is enrolled in at least one course.*

```
WITH cteSource (maxId) AS (
  SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
 WHERE student.sid = cteSource.maxId
```

Georgia Tech

COMMON TABLE EXPRESSIONS

- *Find student record with the highest id that is enrolled in at least one course.*



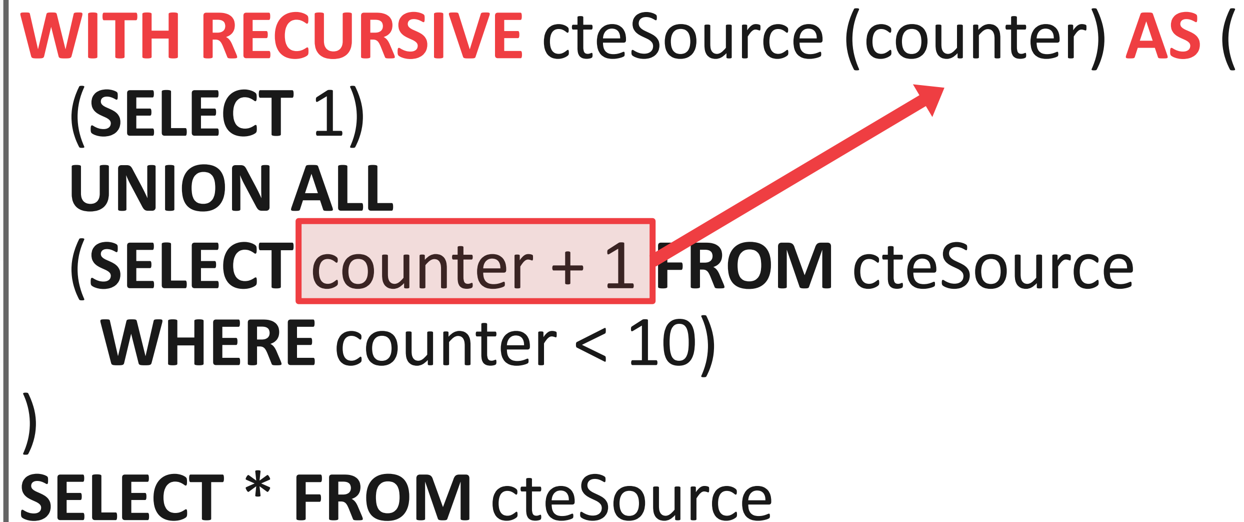
CTE – RECURSION

- Supports recursion unlike nested queries
- *Print the sequence of numbers from 1 to 10.*

```
WITH RECURSIVE cteSource (counter) AS (
   (SELECT 1)
   UNION ALL
   (SELECT counter + 1 FROM cteSource
     WHERE counter < 10)
)
SELECT * FROM cteSource
```

CTE – RECURSION

- Supports recursion unlike nested queries
- *Print the sequence of numbers from 1 to 10.*



SUMMARY

- SQL is not a dead language.
- You should (almost) always strive to compute your answer as a single SQL statement.
- How do these operators generalize to videos?
  - JOIN operator
  - What is a TABLE in this domain?

NEXT LECTURE

- Data Storage