

L23: Course Evaluation & Review

CS3200 Database design (fa18 s2)

<https://northeastern-datalab.github.io/cs3200/>

Version 12/03/2018

Announcements!

- Today
 - Course evaluations
 - Review of all 5 class topics
- Admin:
 - all last HWs are due this Wednesday: HW8, HW10, Q11, Optional PPTX, peer evaluation
 - Extended office hours: Wolfgang: TH and MON during regular class time
 - See course web page
- For final exam:
 - topics: everything that was covered in the course: slides, homeworks, solutions & FMs, in-class discussions, discussion on Piazza, Gradiance, Jupyter
 - "Professor: I am starting to prepare for the final exam. How should I best study?"
 - I will ask some random people in class how they would answer.

The world is increasingly
driven by data...

This class teaches the basics of
how to use & manage data.

Lectures outline

1. **SQL:** Relational data models & Queries

- ~ 5-6 lectures -> **6.5 lectures**
- How to manipulate data with SQL, a declarative language
 - reduced expressive power but the system can do more for you

2. **Database Design:** Design theory and constraints

- ~ 5-6 lectures -> **6 lectures**
- Designing relational schema to keep your data from getting corrupted

3. **Transactions:** Syntax & supporting systems

- ~ 3-4 lectures -> **3 lectures**
- A programmer's abstraction for data consistency

Lectures outline

4. NoSQL

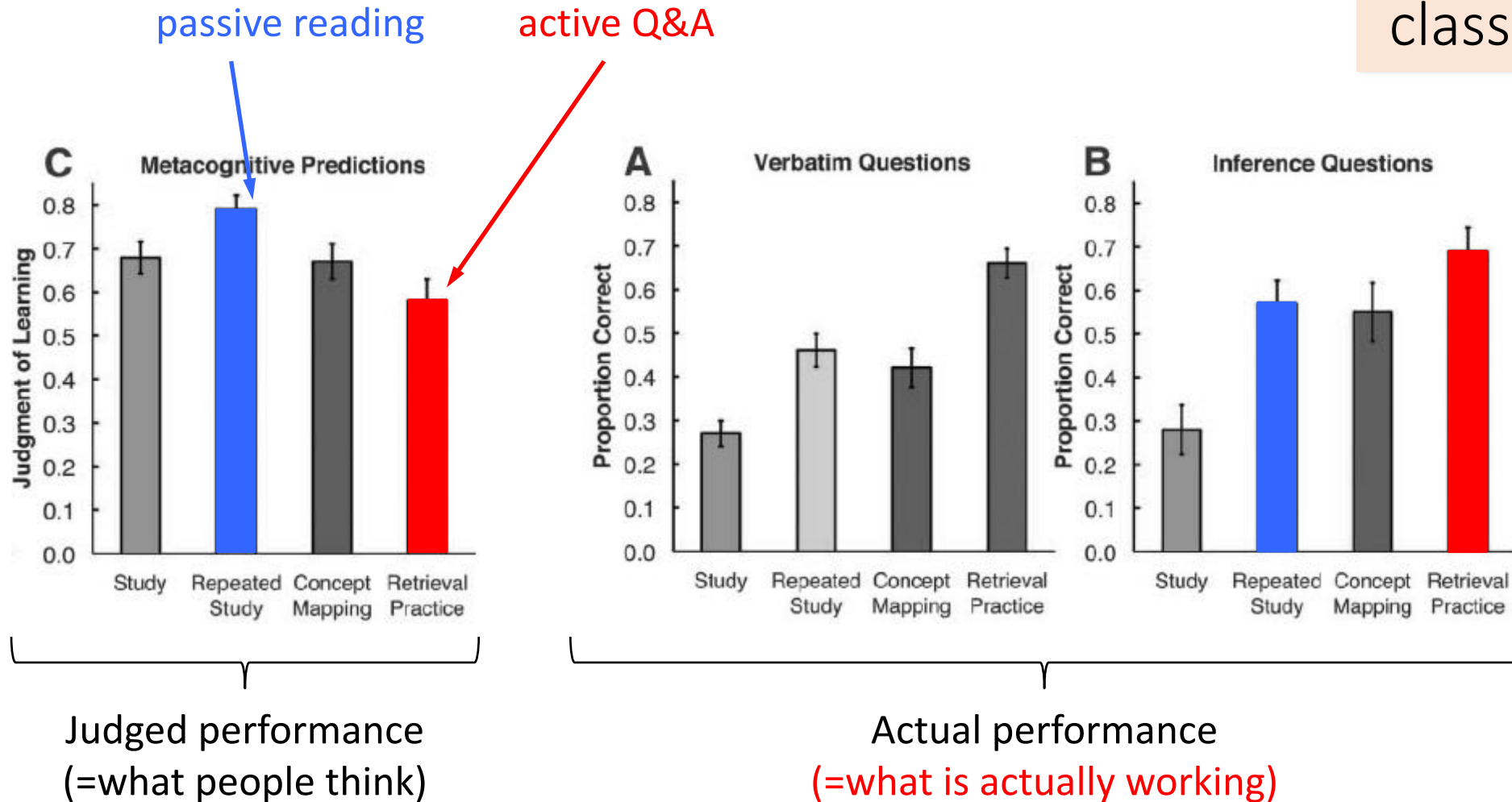
- ~2-3 lectures -> 2.5 lectures
- Key-Value Stores, Column Stores, Document stores, Graph DBs
- (More in CS6240: Large-Scale Parallel Data Processing)

5. Database internals: Query Processing

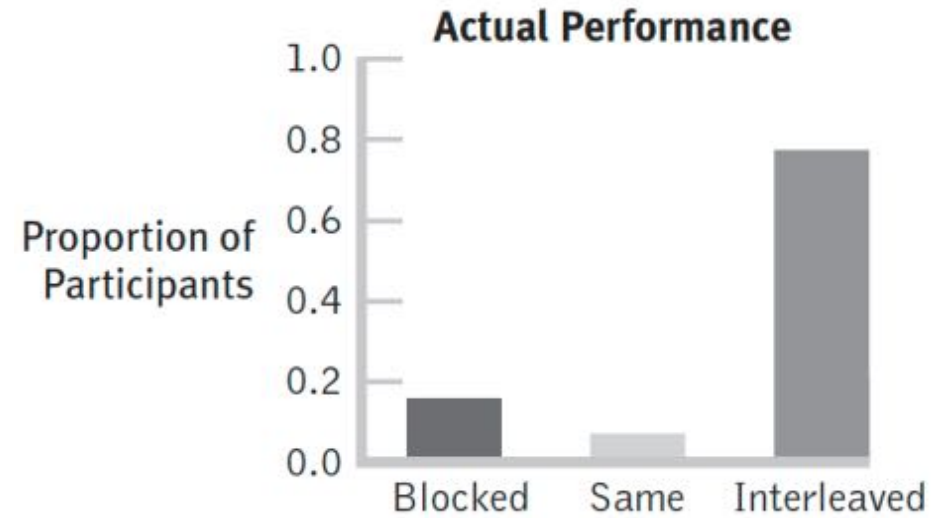
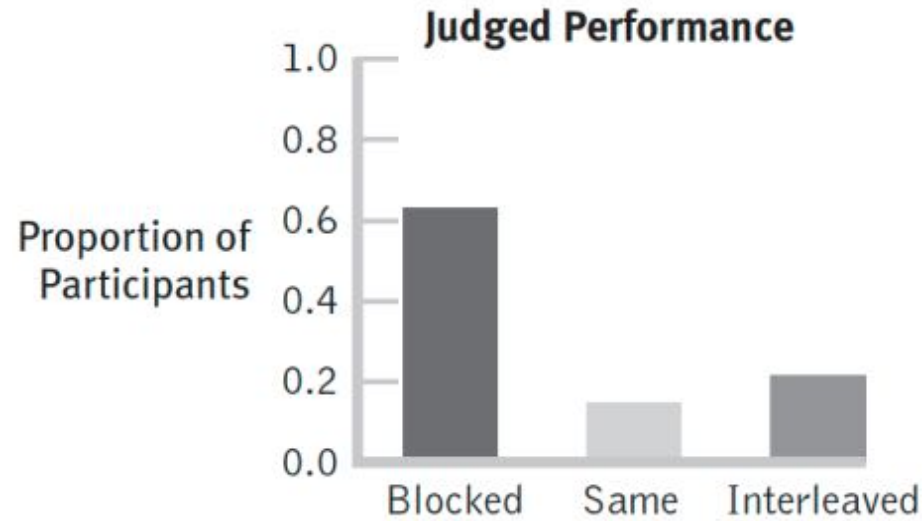
- ~ 3-4 lectures -> 2.5 lectures
- Indexing
- ~~– External Memory Algorithms (IO model) for sorting and joins~~
- Basics of query optimization (Cost Estimates)
- Relational algebra

Studying new material: "Under which study condition do you think you learn better?"

⇒ name plates,
class participation



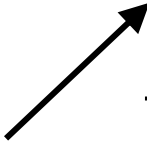
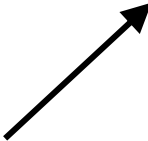
Sequencing Material: "Under which teaching condition do you think you learn better?"



⇒ exams are comprehensive

The mix of chapter and cases is also meant to provide a holistic view of how technology and business interrelate. Don't look for an "international" chapter, an "ethics" chapter, a "mobile" chapter, or a "systems development and deployment" chapter. Instead, you'll see these topics woven throughout many of our cases and within chapter examples. This is how professionals encounter these topics "in the wild, so we ought to study them not in isolation but as integrated parts of real-world examples. Examples are consumer-focused and Internet-heavy for approachability, but the topics themselves are applicable far beyond the context presented.

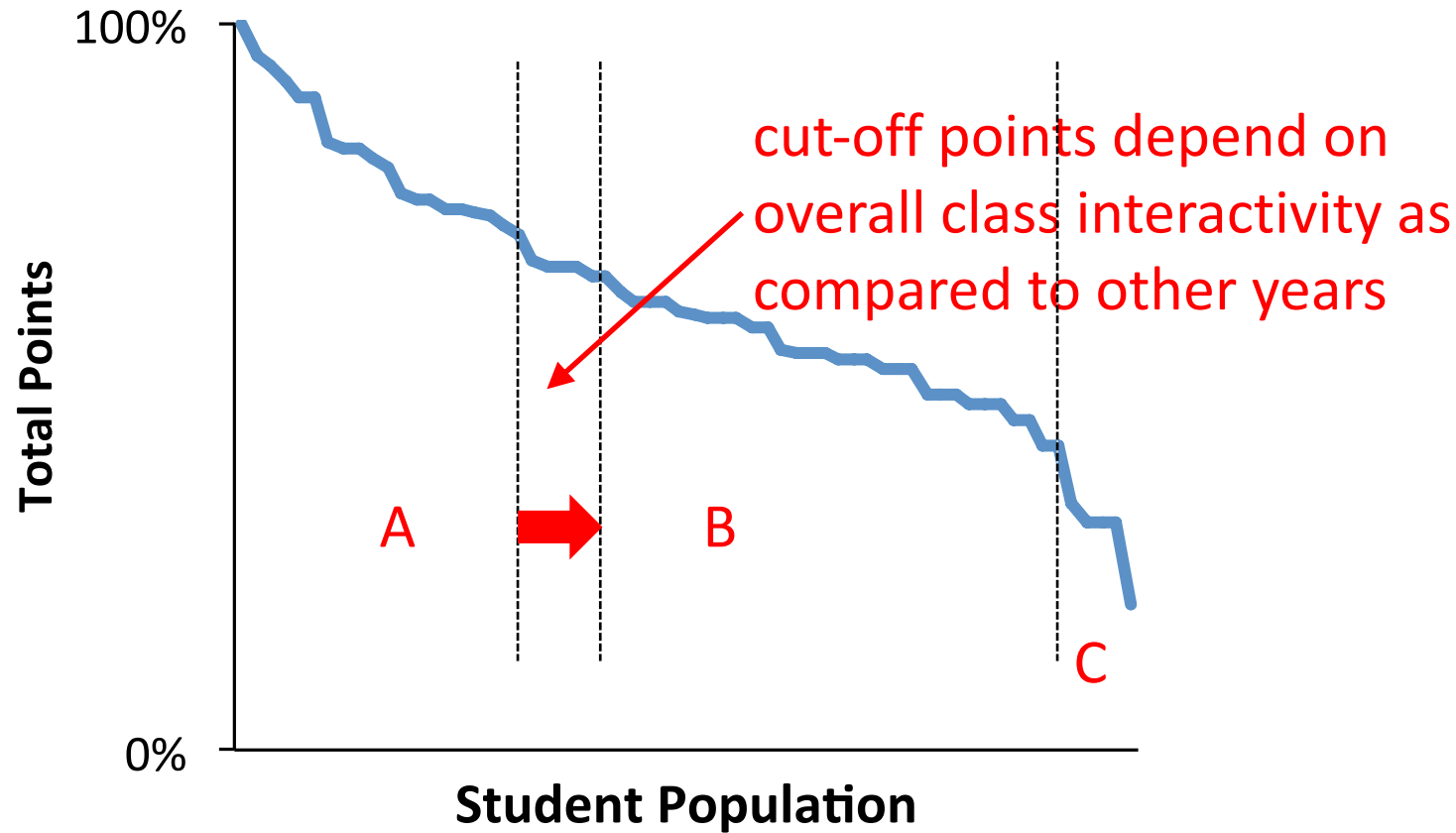
My pedagogic goals for classroom effectiveness

Goal	Increased learning	Fair assessment
Metric	 $\frac{\Delta \text{ learning}}{\text{time invested}} \text{ ratio}$	 $\frac{\text{signal}}{\text{noise}} \text{ ratio}$
Implications	minimize chores, have group HWs, "soft" graded HWs, no attendance check, in-class problems, class contributions, interleaved, discuss student solutions, ...	exam: hard, comprehensive, individual, time-constrained
Risks	"Slacking off"	Stress, "not fun"

Grading Philosophy

Actual point distribution from a past final exam: long, but fair!

- no fixed percentages (e.g., top 30% get A)
- no fixed cut-offs (e.g., 80/100 points for A)



I will not disclose the actual cut-off points. Don't ask for an exception.

Ideas for next year

- Allowing arbitrary groups to work on HWs (as long as acknowledged)
- Renaming Office Hours (focus on TA) to study meetings (TA available)
- All submissions via Gradescope
- More hands-on Jupyter notebooks, even for SQL
- Topics:
 - 1 SQL: no change
 - 2 Database design: shorten and completely replace the Stanford arrow notation with crow foot / UML; requires working with Gradiance to change examples
 - 3 Transactions: extend and include hands-on exercises
 - 4 NoSQL: extend with hands-on with all 4 types of NoSQL databases; all in Jupyter, but MongoDB all setting up on their computers in addition
 - 5 Database internals: Re-introduce basics on IO-aware algorithms

Reminder: Faculty Course Evaluations

- Please take the next 10 minutes to complete your Faculty Course Evaluation ("TRACE") for this course.
- Your feedback:
 - Helps me improve the course
 - Helps your fellow students make better decisions about courses and professors
 - Is anonymous – I get a report with results and comments 2-3 weeks after grades are in
 - Should only take 10 minutes to complete (I will walk outside)
 - Written comments with explanations and suggestions are especially valuable

Thanks for leaving *detailed* Feedback 😊

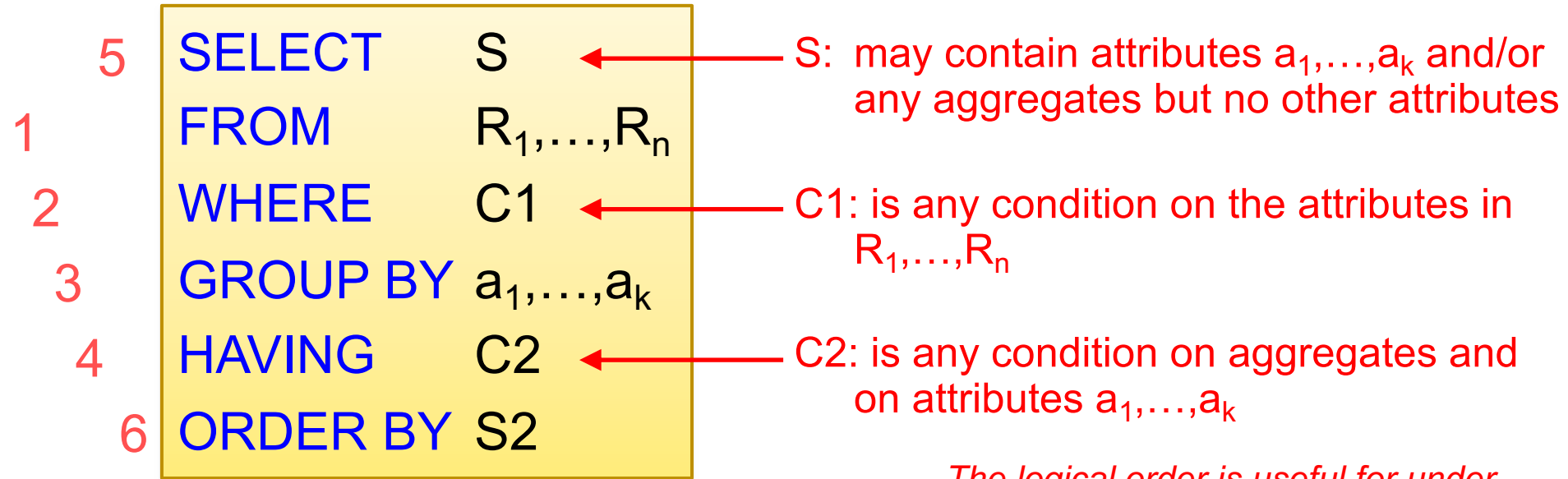
1. Topics: what was most interesting, what not, why, what would you have liked to have covered?
2. Pace: what parts most difficult / too slow or too fast?
3. Class organization / Website: did you find what you were looking for? / what was difficult to find or follow? What would have helped? Suggestions website or Piazza or Gradescope?
4. Gradescope: in the future, everything graded on Gradescope
5. Assessment & cheating: I am following a more British style: homeworks = practice / final = test is "hard" (lower points). "hard exams" & soft cut-offs: what is a good alternative and why?
6. Class diversity: from freshman to senior; thus no groups project; suggestions of splitting up the class?
7. Homeworks: keep random assignments. Perhaps allow arbitrary collaborations? But then homework points give *no* predictions for grade. Thus homework = practice, exams = assessment
8. Jupyter notebooks: what went well or wrong? how to improve?
9. Pedagogy: What aspects helped you learn and not forget: cold calling / group exercises / short slide exercises (SQL) / hands-on SQL typing vs SQL animations / FMs on homework solutions / Office Hours / TA Office Hours / animated slides vs whiteboard drawings (attention vs time needed) / reading material
10. Use of computers & social media in class: yes / no
11. Best practice from other classes / what to copy *to* other classes. Other ways I can help: office hours / anonymous feedback form / 5min breaks / 5min "social breaks" where I assign you to talk to somebody
12. How to make you engage more actively? SQL worked really well. More random calling from class list?
13. New exercise next year: Creating and answering MCQs
14. Textbooks: no single textbook to buy, but all digitally available from different sources

Review

- A quick tour d'horizon through the 5 topics we discussed in class

1. SQL

General form of SQL Query



Evaluation

1. Evaluate FROM
2. WHERE, apply condition C1
3. GROUP BY the attributes a_1, \dots, a_k
4. Apply condition C2 to each group (may have aggregates)
5. Compute aggregates in S and return the result
6. Sort rows by ORDER BY clause

*The logical order is useful for understanding, but not always correct. The ANSI SQL standard does not require a specific processing order and leaves that to the implementation. Recall our intro example with **SELECT DISTINCT** and order by! Notice that that example can't be explained with the order shown here*

From → Where → Group By → Select



Purchase

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10

Product	TotalSales
Bagel	40
Banana	20

- Select contains
- grouped attributes
 - and aggregates

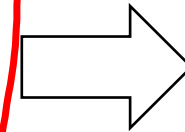
```
1 4 SELECT product, sum(quantity) as TotalSales
2 FROM Purchase
3 WHERE price > 1
4 GROUP BY product
```

Let's confuse the database engine



Purchase

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10



Product	Quantity
Bagel	?
Banana	?

What quantity should the DB return for Banana?

```
SELECT    product, quantity
FROM      Purchase
GROUP BY  product
```

The DB engine is confused, there is no single quantity for banana (it's an ill-defined query). It should thus return an error (only SQLite misbehaves and returns something, but which makes no sense). Please think this through carefully!

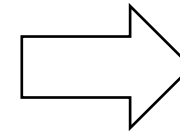
Don't use new Alias in HAVING clause



308

What does this query return over the given database?

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10



Product	SumQ
Bagel	40
Banana	50

Error in SQL server!
Reason: HAVING is
evaluated before SELECT!
(However, SQLite works:
different implementation)

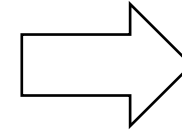
```
SELECT product, sum(quantity) as SumQ
FROM Purchase
WHERE quantity > 15
GROUP BY product
HAVING SumQ > 35
```

Don't use new Alias in HAVING clause



What does this query return over the given database?

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10



Product	SumQ
Banana	50
Bagel	40

Works! Notice
that new sorting

```
SELECT product, sum(quantity) as SumQ
FROM Purchase
WHERE quantity > 15
GROUP BY product
HAVING sum(quantity) > 35
ORDER BY sumQ desc
```

Illustration



English

eText	<u>eid</u>
One	1
Two	2
Three	3
Four	4
Five	5
Six	6

French

<u>fid</u>	fText
1	Un
3	Trois
4	Quatre
5	Cinq
6	Siz
7	Sept
8	Huit

An "inner join":

```
SELECT *  
FROM   English, French  
WHERE  eid = fid
```

Same as:

```
SELECT *  
FROM   English JOIN French  
ON     eid = fid
```

etext	eid	fid	ftext
One	1	1	Un
Three	3	3	Trois
Four	4	4	Quatre
Five	5	5	Cinq
Six	6	6	Siz

"JOIN"
same as
"INNER JOIN"

Illustration



English

eText	<u>eid</u>
One	1
Two	2
Three	3
Four	4
Five	5
Six	6

French

<u>fid</u>	fText
1	Un
3	Trois
4	Quatre
5	Cinq
6	Six
7	Sept
8	Huit

"FULL JOIN"
same as
"FULL OUTER JOIN"

SELECT *
FROM English FULL JOIN French
ON English.eid = French.fid

~~SELECT *
FROM English JOIN French
ON eid = fid~~

etext	eid	fid	fText
One	1	1	Un
Two	2	NULL	NULL
Three	3	3	Trois
Four	4	4	Quatre
Five	5	5	Cinq
Six	6	6	Six
NULL	NULL	7	Sept
NULL	NULL	8	Huit

SQLite does not support "FULL OUTER JOIN"s ☹ (but "LEFT JOIN")

Illustration

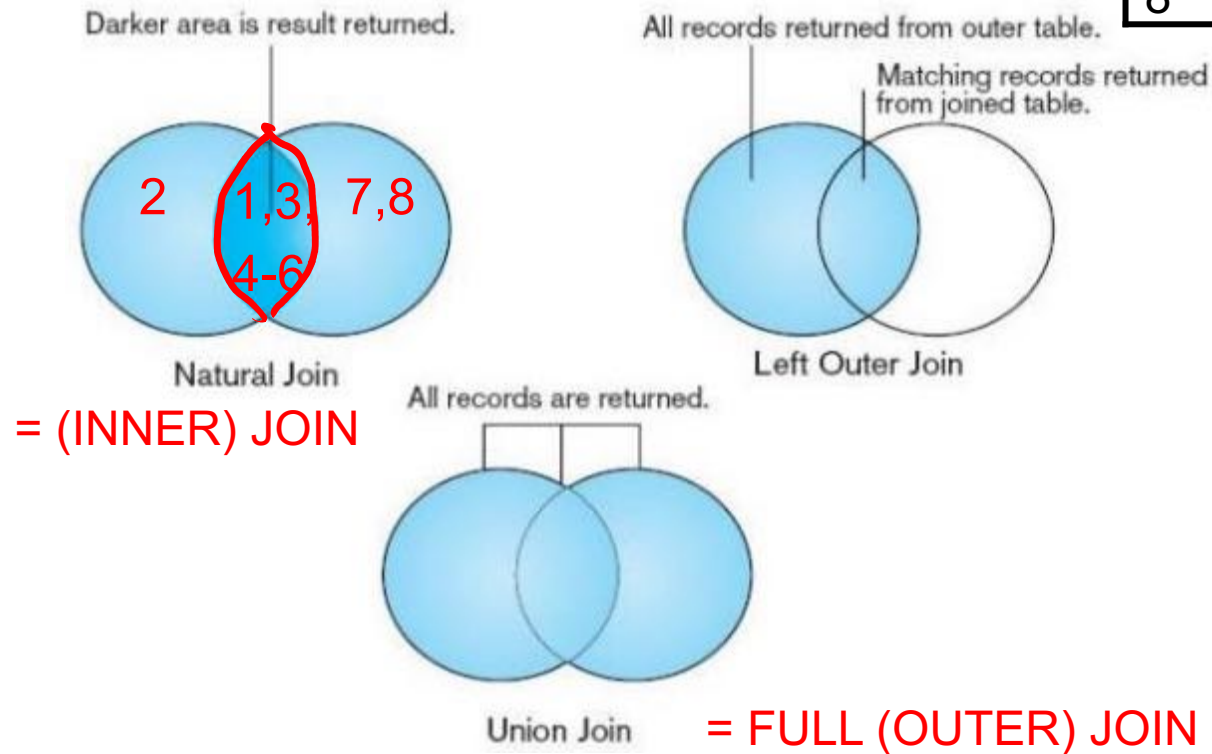


English

eText	<u>eid</u>
One	1
Two	2
Three	3
Four	4
Five	5
Six	6

French

<u>fid</u>	fText
1	Un
3	Trois
4	Quatre
5	Cinq
6	Six
7	Sept
8	Huit



Empty Group Problem

Item(name, category)



334

Purchase2(iName, store, month)

Compute, for each product, the total number of sales in Sept (= month 9)

```
SELECT    name, count(*)  
FROM      Item, Purchase2  
WHERE     name = iName  
          and    month = 9  
GROUP BY name
```

What's wrong?

Empty Group Problem

Item(name, category)



334

Purchase2(iName, store, month)

Compute, for each product, the total number
of sales in Sept (= month 9)

We need to use an attribute from
"Purchase2" to get the correct 0
count. Try "name" from "Item".

```
SELECT    name, count(store)
FROM      Item LEFT JOIN Purchase2 ON
          name = iName
          and    month = 9
GROUP BY  name
```

Now we also get the products with 0 sales

2. DB design

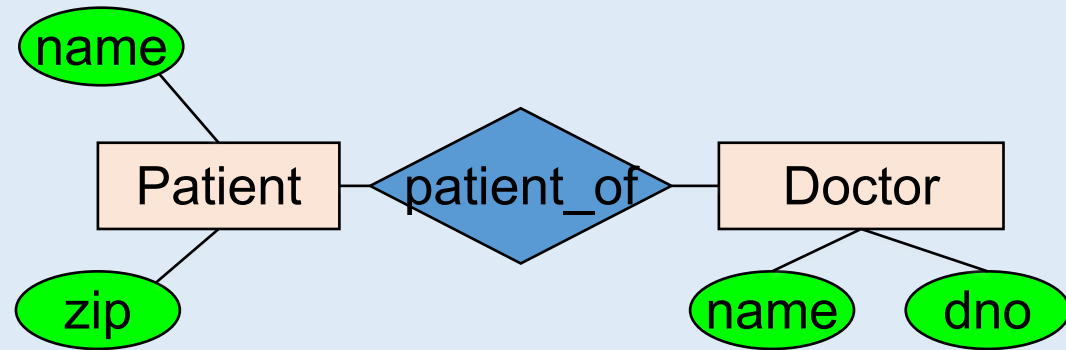
Data modeling and Database Design Process

1. ER Diagram

Conceptual Model:

("technology independent")

describe main data items



2. Relational Database Design

Logical Model

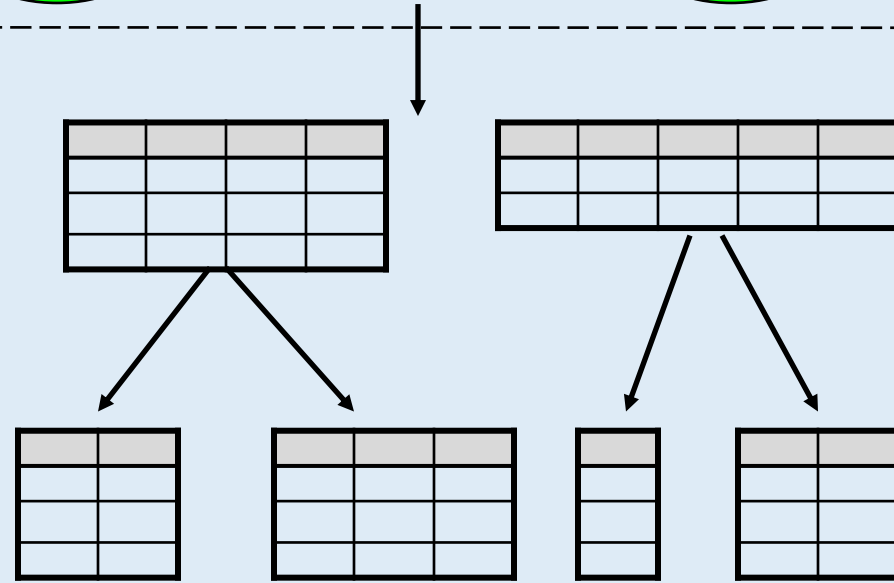
("for relational databases"):

Tables, Constraints

Functional Dependencies

Normalization:

Eliminates anomalies

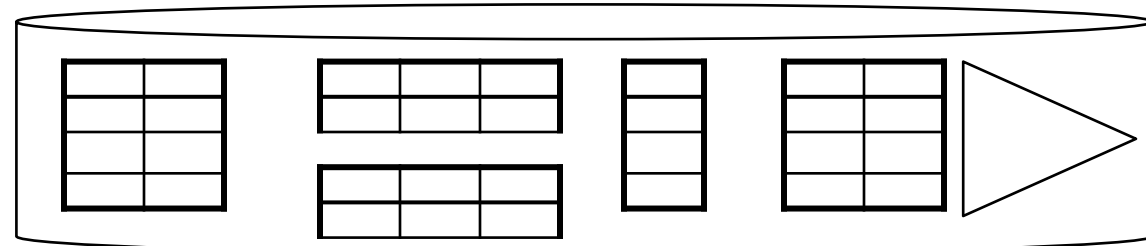


3. Database Implementation

Physical Model

Physical storage details

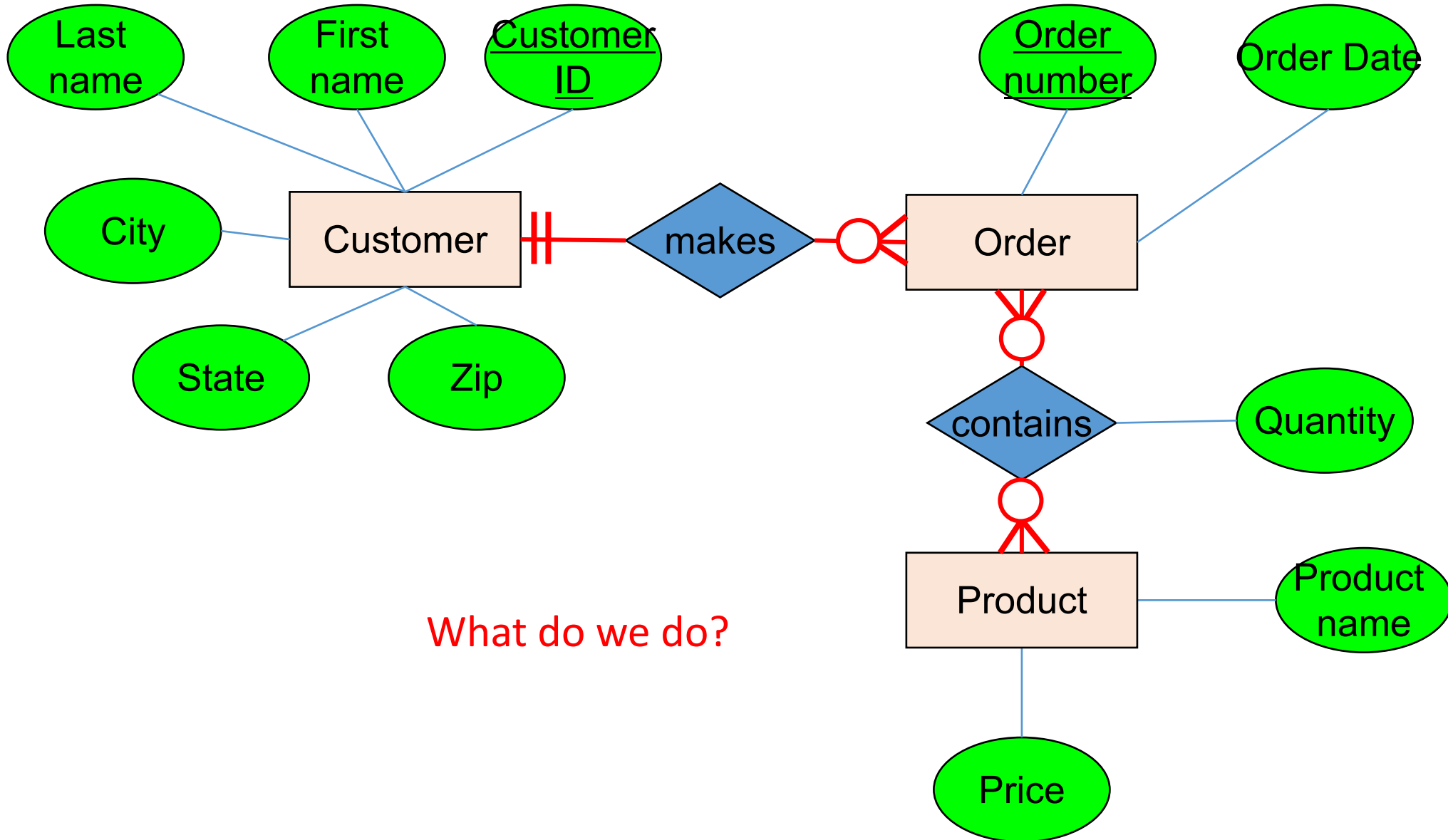
Result: Physical Schema



From E/R Diagrams to Relational Schema

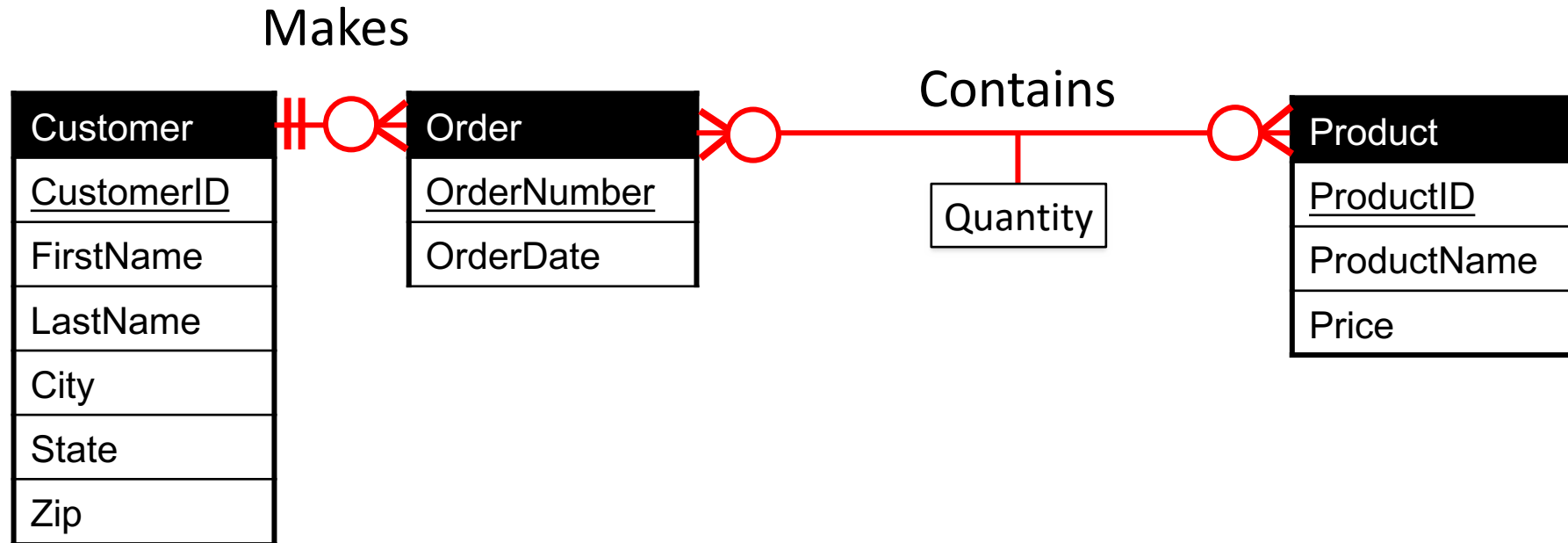
- Key concept
 - Entity sets become relations, Relationships can become relations (tables in RDBMS)
 - Tables are connected with foreign key constraints
- A database schema
 - A map of the tables and fields (attributes) in the database
 - This is what is implemented in the database management system
 - Part of the “design” process

Example: translate this ERD v1 into tables



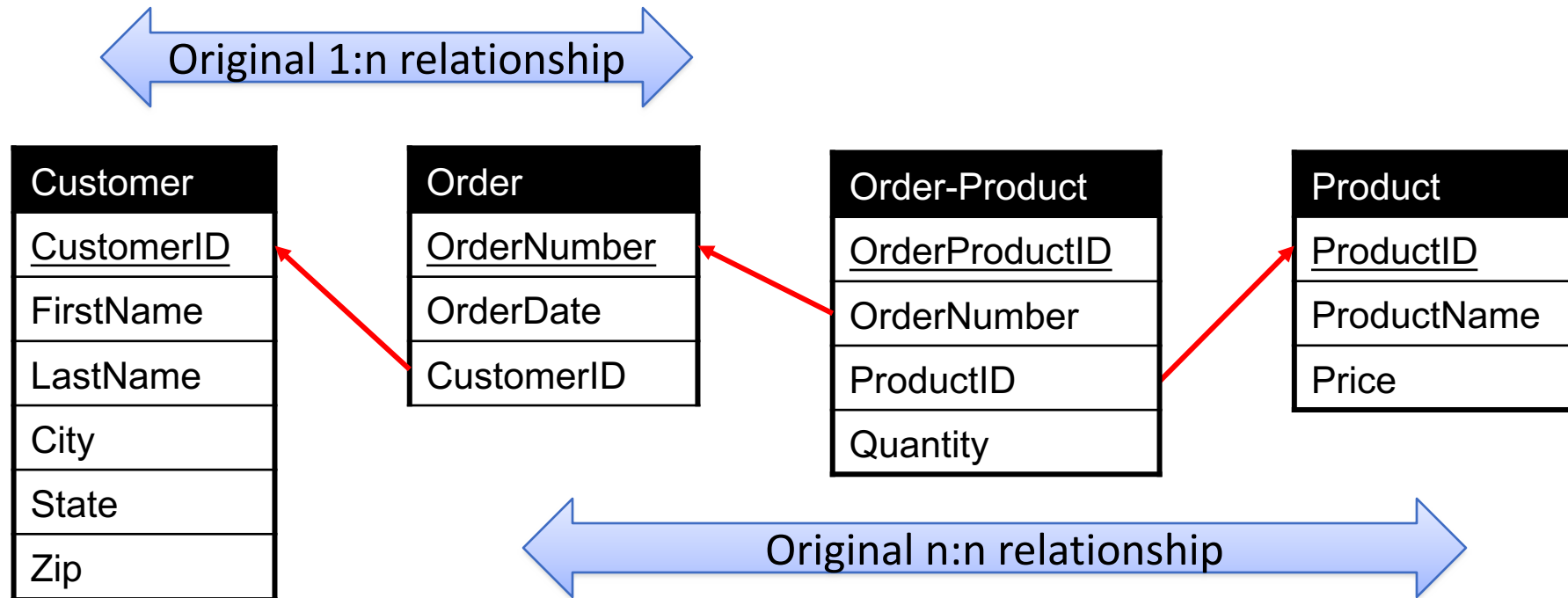
What do we do?

Example: translate this ERD v2 into tables



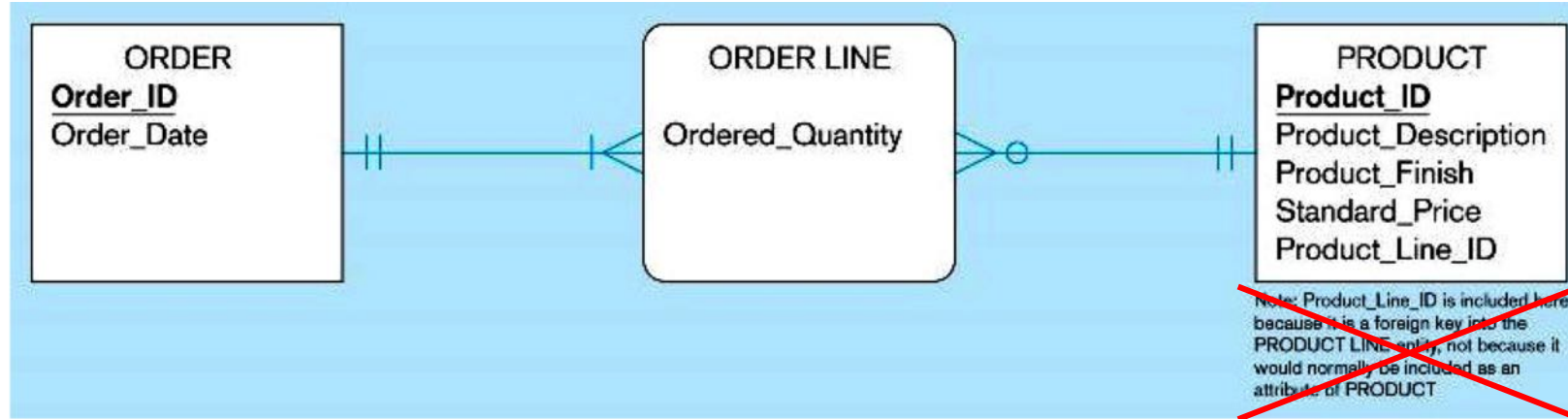
What do we do?

Example: Our Order Database schema

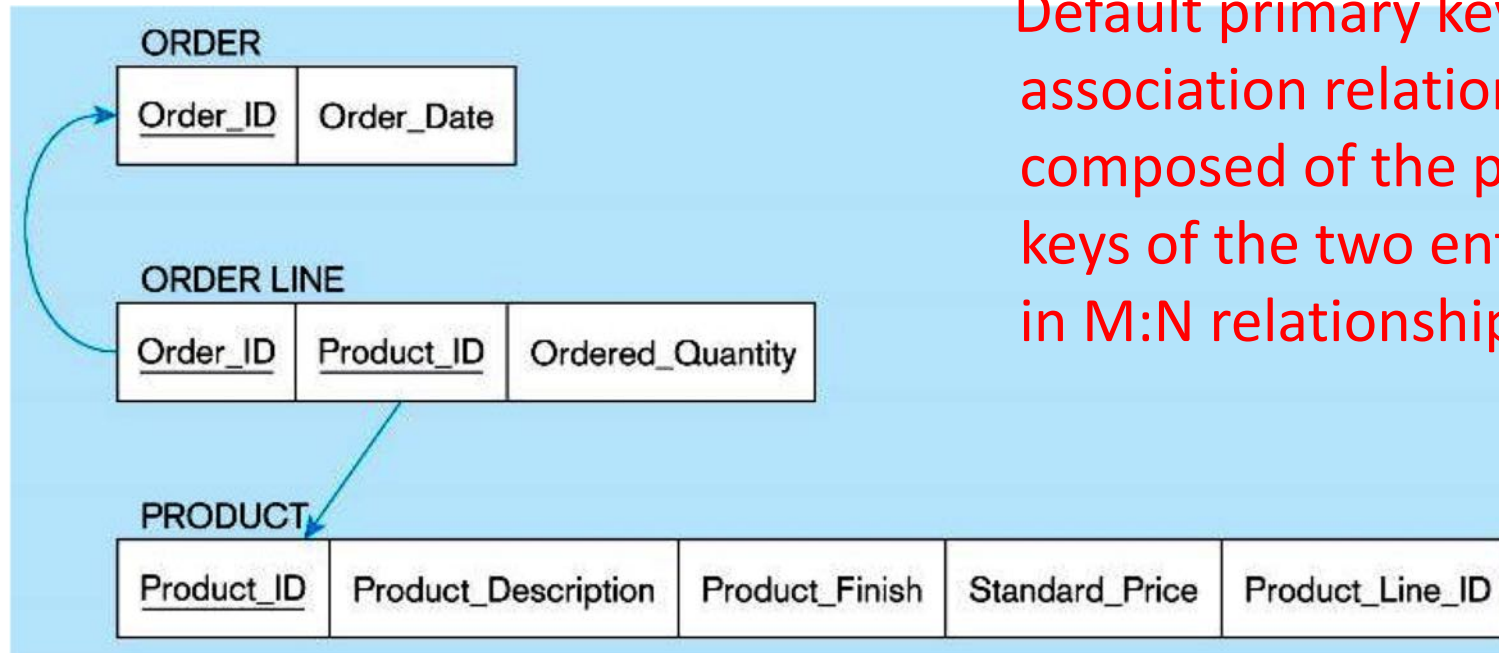
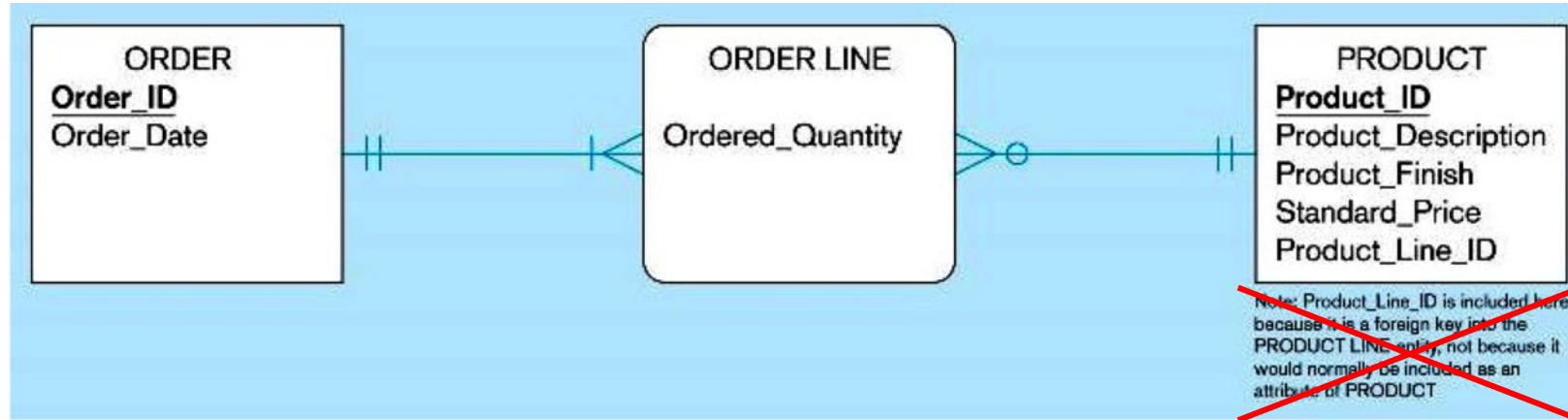


- Order-Product is a decomposed many-to-many relationship
 - Order-Product has a 1:n relationship with Order and Product
 - Now an order can have multiple products, and a product can be associated with multiple orders

A) Associative Entity Relations (No Identifier)



A) Associative Entity Relations (No Identifier)

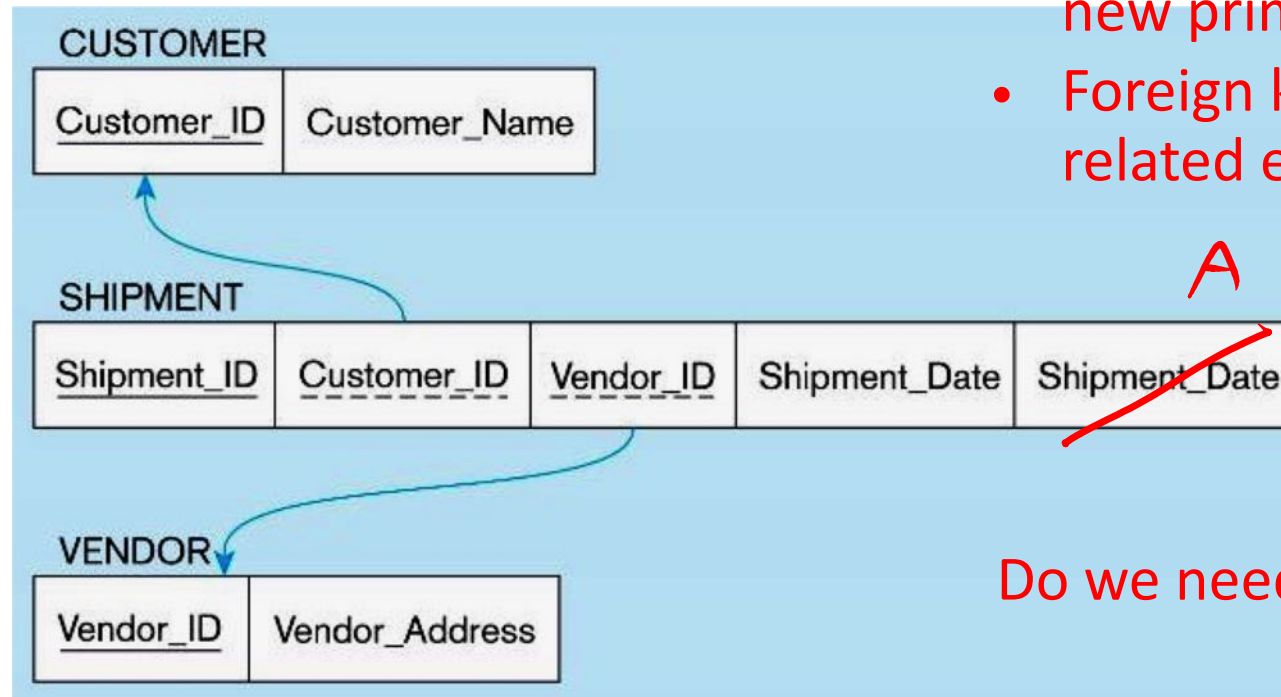


Default primary key for the association relation is composed of the primary keys of the two entities (as in M:N relationship)

B) Associative Entity Relations (With Identifier)



B) Associative Entity Relations (With Identifier)



- Identifier attribute becomes new primary key in relation
- Foreign keys reference all related entities

Do we need the key?

Relational Schema Design



Recall set attributes (persons with several phones):

Employee

Name	<u>SSN</u>	<u>PhoneNumber</u>	City
Fred	123-45-6789	412-555-1234	Boston
Fred	123-45-6789	412-555-6543	Boston
Joe	987-65-4321	908-555-2121	Westfield

- One person may have multiple phones, but lives in only one city
- Primary key is thus (SSN, PhoneNumber)

Do you see any anomalies?

Relational Schema Design



Recall set attributes (persons with several phones):

Employee

Name	<u>SSN</u>	<u>PhoneNumber</u>	City
Fred	123-45-6789	412-555-1234	Boston
Fred	123-45-6789	412-555-6543	Boston
Joe	987-65-4321	908-555-2121	Westfield

- One person may have multiple phones, but lives in only one city
- Primary key is thus (SSN, PhoneNumber)

Do you see any anomalies?

- **Update anomalies:** what if Fred moves to "New York"?
- **Insert anomalies:** what if Joe gets a second phone number
- **Deletion anomalies:** what if Joe deletes his phone number?

(what if Joe had no phone #)

What do we do????

Relation Decomposition



Break the relation into two:

Employee

Name	<u>SSN</u>	<u>PhoneNumber</u>	City
Fred	123-45-6789	412-555-1234	Boston
Fred	123-45-6789	412-555-6543	Boston
Joe	987-65-4321	908-555-2121	Westfield

Employee

Name	<u>SSN</u>	City
Fred	123-45-6789	Boston
Joe	987-65-4321	Westfield

Phone

<u>SSN</u>	<u>PhoneNumber</u>
123-45-6789	412-555-1234
123-45-6789	412-555-6543
987-65-4321	908-555-2121

Anomalies have gone:

- No more repeated data
- Easy to move Fred to "New York" (how ?)
- Easy to delete all Joe's phone numbers (how ?)

Keys and Superkeys

A superkey is a set of attributes A_1, \dots, A_n s.t. for *any other* attribute B in R , we have $\{A_1, \dots, A_n\} \rightarrow B$

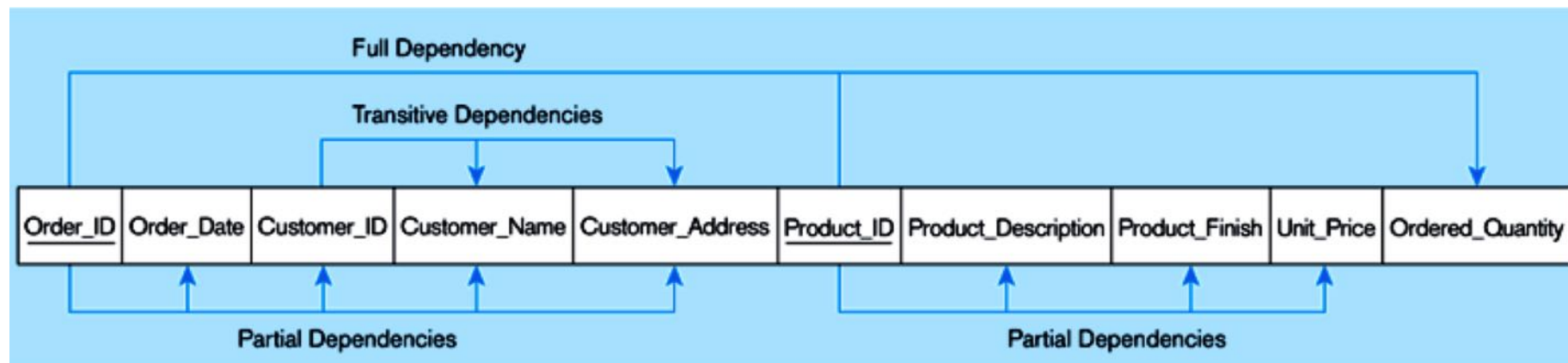
I.e. all attributes are *functionally determined* by a superkey

A key is a *minimal* superkey (also called "candidate key")

This means that no subset of a key is also a superkey (i.e., dropping any attribute from the key makes it no longer a superkey)

Quick recap FDs

- **Functional Dependency (FD)**: The value of one set of attributes (the **determinant**) uniquely determines the value of another set of attributes (the **dependents**)
- A **superkey (SK)** is as a set of attributes of a relation schema upon which all attributes of the schema are functionally dependent.
- A **candidate key (CK)** is a non-redundant (minimal) SK
- **Prime attribute**: belonging to some candidate key
- **Partial FD**: FD in which ^{some} more non-prime attributes are functionally dependent on part (but not all) of any CK
- **Transitive FD**: An FD between two (or more) **nonkey attributes**
- **3NF**: no partial nor transitive FD



Boyce-Codd Normal Form (BCNF)

- Boyce-Codd normal form (BCNF)
 - A relation is in BCNF, if and only if, every (non-trivial) determinant is a superkey.
- The difference between 3NF and BCNF is that for a FD $A \rightarrow B$,
 - 3NF allows this dependency in a relation if B is a primary-key attribute and A is not a candidate key,
 - whereas BCNF insists that for this dependency to remain in a relation, A must be a SK (contain a CK).

3NF to BCNF

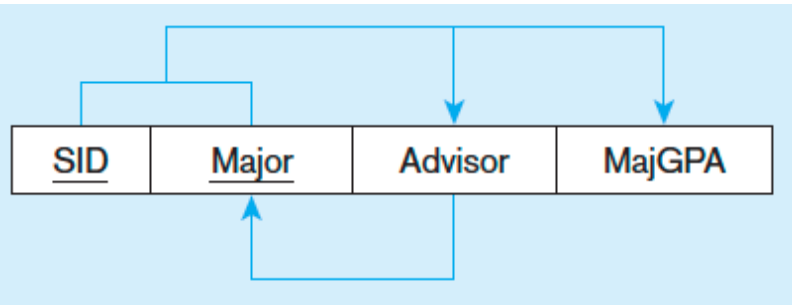
STUDENT ADVISOR

<u>SID</u>	<u>Major</u>	Advisor	MajGPA
123	Physics	Hawking	4.0
123	Music	Mahler	3.3
456	Literature	Michener	3.2
789	Music	Bach	3.7
678	Physics	Hawking	3.5

3NF to BCNF

STUDENT ADVISOR

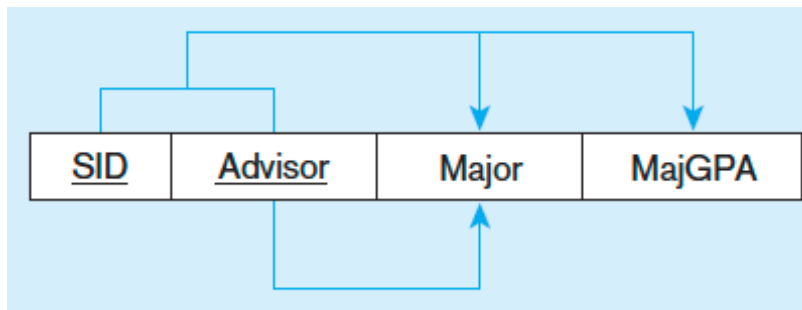
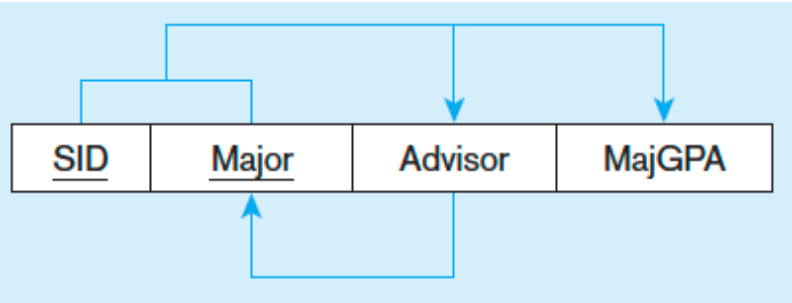
<u>SID</u>	<u>Major</u>	Advisor	MajGPA
123	Physics	Hawking	4.0
123	Music	Mahler	3.3
456	Literature	Michener	3.2
789	Music	Bach	3.7
678	Physics	Hawking	3.5



3NF to BCNF

STUDENT ADVISOR

<u>SID</u>	<u>Major</u>	Advisor	MajGPA
123	Physics	Hawking	4.0
123	Music	Mahler	3.3
456	Literature	Michener	3.2
789	Music	Bach	3.7
678	Physics	Hawking	3.5

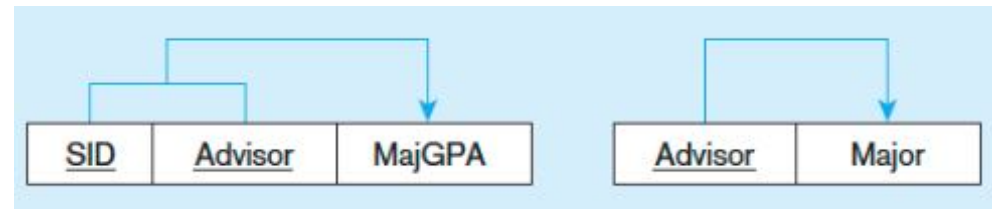


STUDENT

<u>SID</u>	<u>Advisor</u>	MajGPA
123	Hawking	4.0
123	Mahler	3.3
456	Michener	3.2
789	Bach	3.7
678	Hawking	3.5

ADVISOR

<u>Advisor</u>	<u>Major</u>
Hawking	Physics
Mahler	Music
Michener	Literature
Bach	Music



BCNF vs 3NF

- **BCNF**: For **every** functional dependency $X \rightarrow Y$ in a set F of functional dependencies over relation R , either:
 - X is a **superkey** of R
 - (or Y is a subset of X , thus the FD is trivial)
- **3NF**: For every functional dependency $X \rightarrow Y$ in a set F of functional dependencies over relation R , either:
 - X is a superkey of R
 - or Y is a subset of K for some CK (Y is prime)
 - *N.b.*, no subset of a key is a key
 - (or Y is a subset of X , thus the FD is trivial)

A problem with BCNF

$\{Unit\} \rightarrow \{Company\}$ $\{Company, Product\} \rightarrow \{Unit\}$

Unit	Company	Product
...

<u>Unit</u>	Company
...	...

Unit	Product
...	...

$\{Unit\} \rightarrow \{Company\}$

We do a BCNF decomposition
on a “bad” FD:
 $\{Unit\}^+ = \{Unit, Company\}$

We lose the FD $\{Company, Product\} \rightarrow \{Unit\}!!$

So Why is that a Problem?

$\{\text{Unit}\} \rightarrow \{\text{Company}\}$
 $\{\text{Company}, \text{Product}\} \rightarrow \{\text{Unit}\}$

<u>Unit</u>	Company
Galaga99	NEU
Bingo	NEU

Unit	Product
Galaga99	Databases
Bingo	Databases

No problem so far.
All *local* FD's are satisfied.

$\{\text{Unit}\} \rightarrow \{\text{Company}\}$

Unit	Company	Product
Galaga99	NEU	Databases
Bingo	NEU	Databases

Let's put all the data back into a single table again:

Violates the FD $\{\text{Company}, \text{Product}\} \rightarrow \{\text{Unit}\}!!$

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables R_1, R_2, \dots with their own FDs F_1, F_2, \dots
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD across tables!

Practical Problem: To enforce FD, must reconstruct R —*on each insert!*

3. Transactions

ACID

- **A**tomicity
 - Either all operations applied or none are (hence, we need not worry about the effect of incomplete / failed transactions)
- **C**onsistency
 - Each transaction can start with a consistent database and is required to leave the database consistent (bring the DB from one to another consistent state)
- **I**solation
 - The effect of a transaction should be as if it is the only transaction in execution (in particular, changes made by other transactions are not visible until committed)
- **D**urability
 - Once the system informs a transaction success, the effect should hold without regret, even if the database crashes (before making all changes to disk)

Transfer Example

Y/

R,(A)

W,(A)

```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```

txn₁

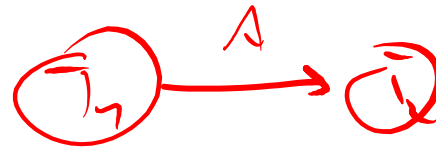
```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```

txn₂

```
Begin
Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y=y+100
Write(C,y)
Commit
```

- *Scheduling* is the operation of interleaving transactions
 - *Why is it good?*
- A *serial schedule* executes transactions one at a time, from beginning to end
- A good (“*serializable*”) scheduling is one that *behaves like some serial scheduling* (typically by locking protocols)

Scheduling Example 1



txn₁

```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```



```
Read(A,v)
v = v-100
Write(A,v)
```

```
Read(B,w)
```

```
w=w+100
```

```
Write(B,w)
```

```
Read(A,x)
x = x-100
Write(A,x)
```

```
Read(C,y)
```

```
y=y+100
```

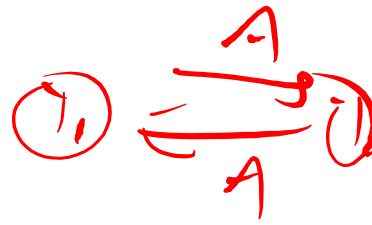
```
Write(C,y)
```



txn₂

```
Begin
Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y=y+100
Write(C,y)
Commit
```

Scheduling Example 2



txn₁

```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```



```
Read(A,v)
v = v-100

Write(A,v)

Read(B,w)

w=w+100

Write(B,w)
```



txn₂

```
Begin
Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y=y+100
Write(C,y)
Commit
```

```
Read(A,x)
x = x-100
Write(A,x)

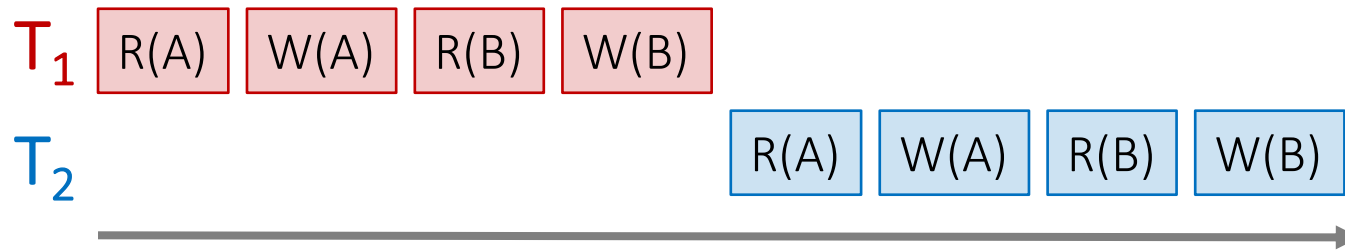
Read(C,y)

y=y+100

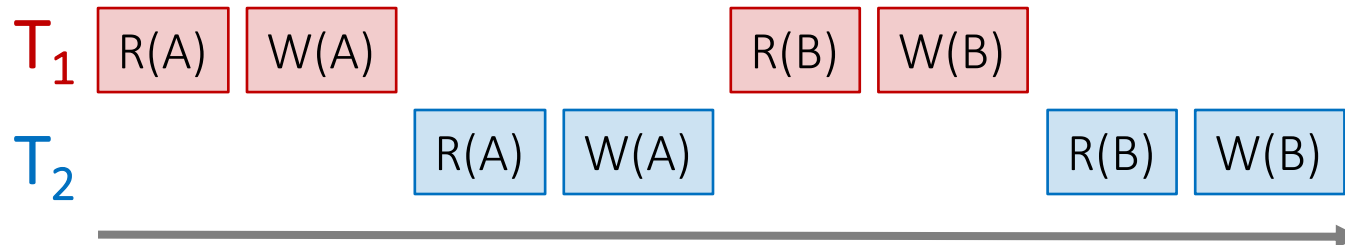
Write(C,y)
```


Recall: Concurrency as Interleaving TXNs

Serial Schedule:



Interleaved Schedule:

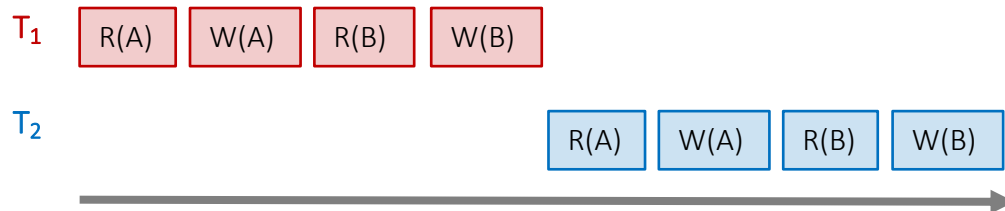


- For our purposes, having TXNs occur concurrently means interleaving their component actions (R/W)

We call the particular order of interleaving a schedule

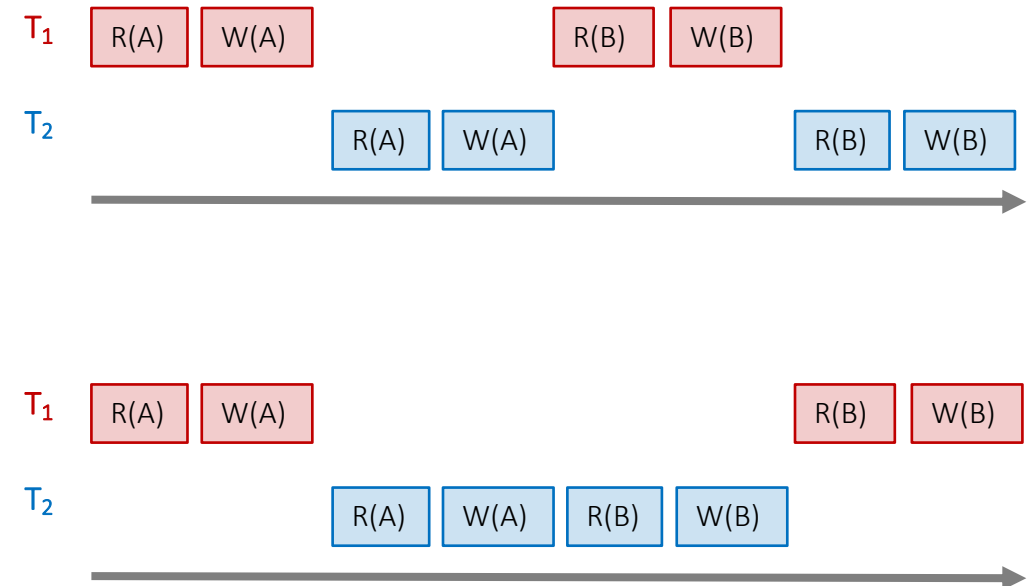
Recall: “Good” vs. “bad” schedules

Serial Schedule:



Why?

Interleaved Schedules:



We want to develop ways of discerning “good” vs. “bad” schedules

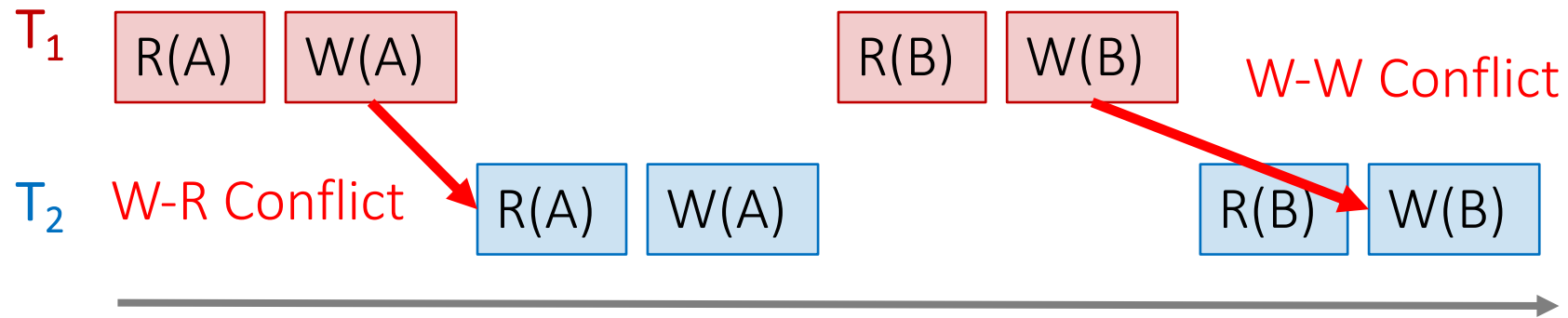
Ways of Defining “Good” vs. “Bad” Schedules

- Recall: we call a schedule serializable if it is equivalent to some serial schedule
 - We used this as a notion of a “good” interleaved schedule, since a serializable schedule will maintain isolation & consistency
- Now, we’ll define a stricter, but very useful variant:
 - **Conflict serializability**

We’ll need to define *conflicts* first..

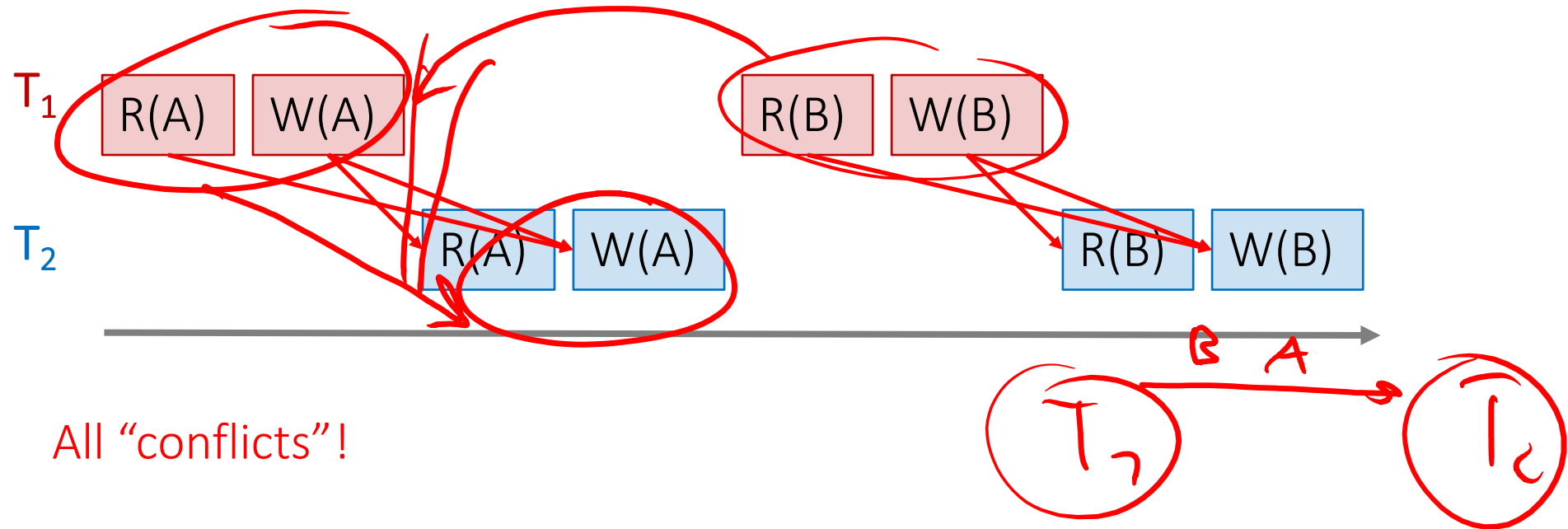
Conflicts

Two actions conflict if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflicts

Two actions conflict if they are part of different TXNs, involve the same variable, and at least one of them is a write



Conflict Serializability

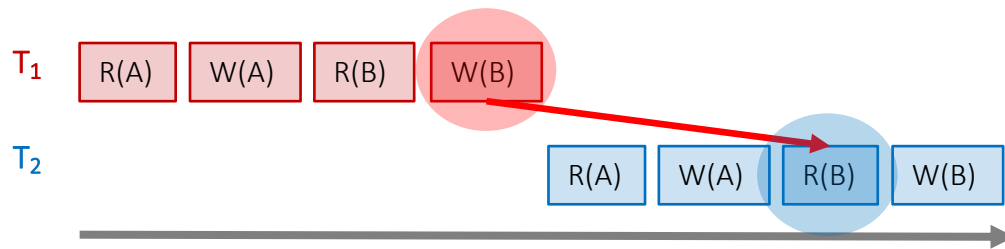
- Two schedules are conflict equivalent if:
 - They involve the same actions of the same TXNs
 - Every pair of conflicting actions of two TXNs are ordered in the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Conflict serializable \Rightarrow serializable

So if we have conflict serializable, we have consistency & isolation!

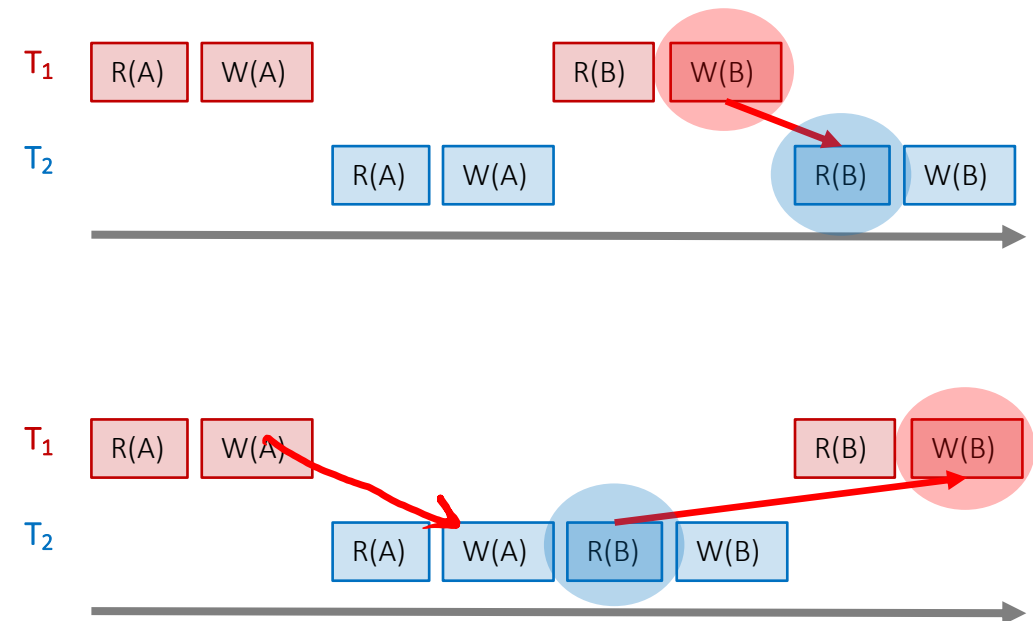
Recall: “Good” vs. “bad” schedules

Serial Schedule:



Note that in the “bad” schedule, the *order of conflicting actions is different than the above (or any) serial schedule!*

Interleaved Schedules:



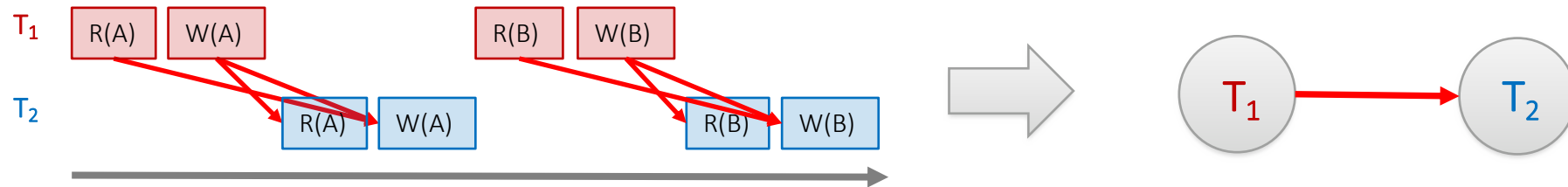
Conflict serializability also provides us with an operative notion of “good” vs. “bad” schedules!

Note: Conflicts vs. Anomalies

- Conflicts are things we talk about to help us characterize different schedules
 - Present in both “good” and “bad” schedules
- Anomalies are instances where isolation and/or consistency is broken because of a “bad” schedule
 - We often characterize different anomaly types by what types of conflicts predicated them

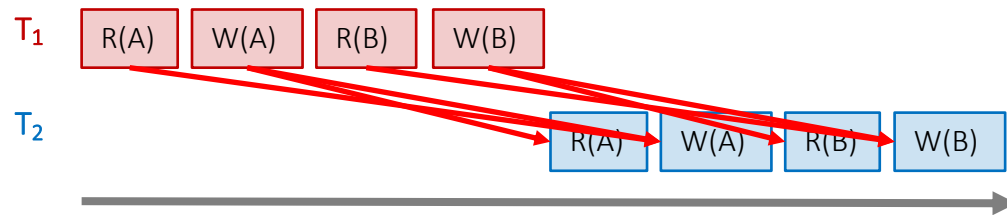
The Conflict Graph

- Let's now consider looking at conflicts at the TXN level
- Consider a graph where the nodes are TXNs, and there is an edge from $T_i \rightarrow T_j$ if any actions in T_i precede and conflict with any actions in T_j



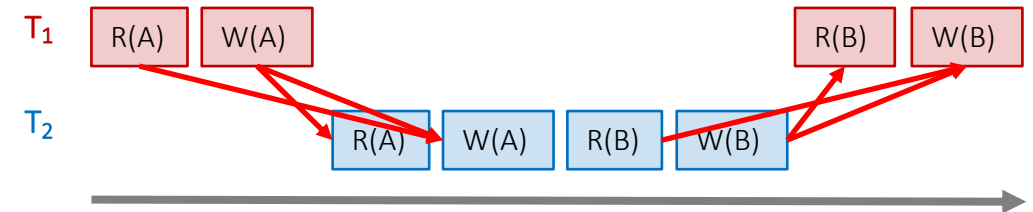
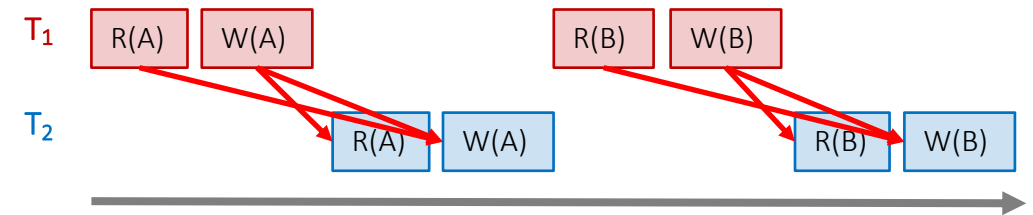
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



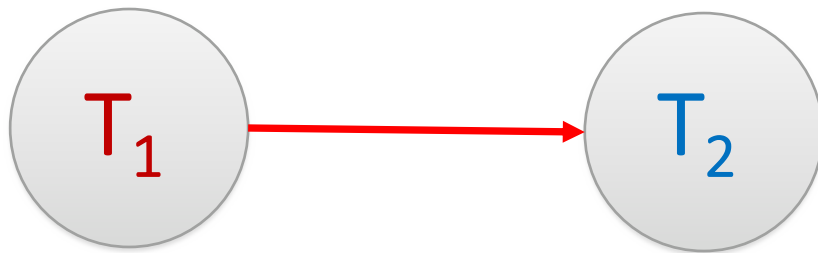
A bit complicated...

Interleaved Schedules:



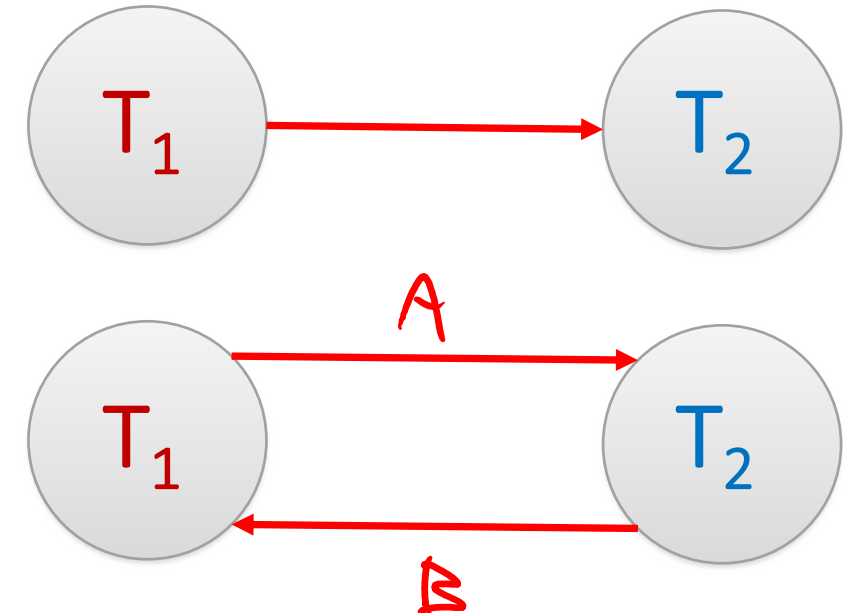
What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:



Simple!

Interleaved Schedules:



Theorem: Schedule is **conflict serializable** if and only if its conflict graph is acyclic

Examples of Transactions

- Airline ticketing
 - Verify that the seat is vacant, with the price quoted, then charge credit card, then reserve
- Textbook example: bank money transfer
 - Read from acct#1, verify funds, update acct#1, update acct#2
- Online purchasing
 - Similar
- “Transactional file systems” (MS NTFS)
 - Moving a file from one directory to another: verify file exists, copy, delete

Strict Two-Phase Locking ("2PL")

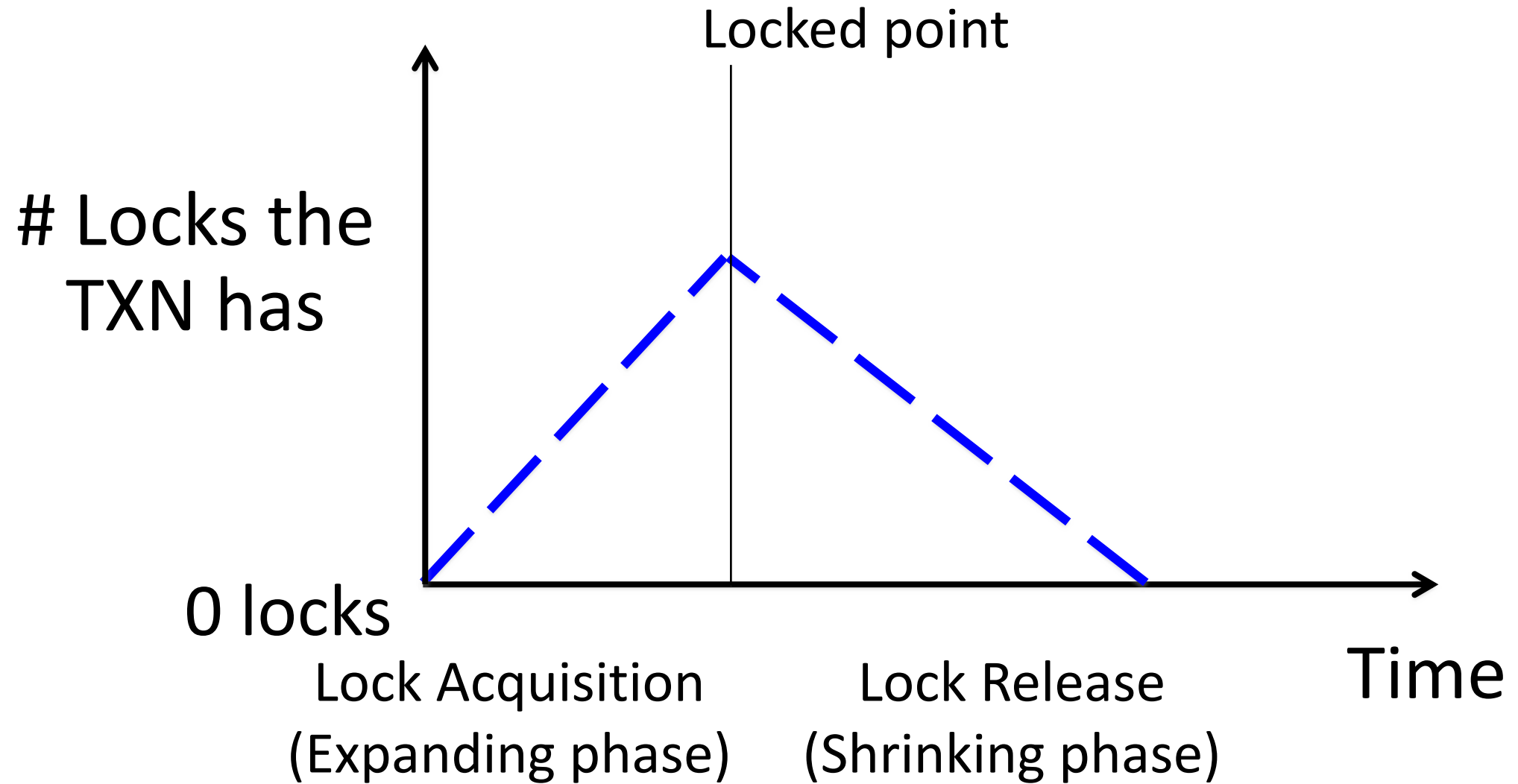
- Two-phase locking is a way to deal with concurrency, because it guarantees conflict serializability (if it completes...)
- Also (conceptually) straightforward to implement, and transparent to the user!

Strict Two-Phase Locking (Strict 2PL) Protocol:

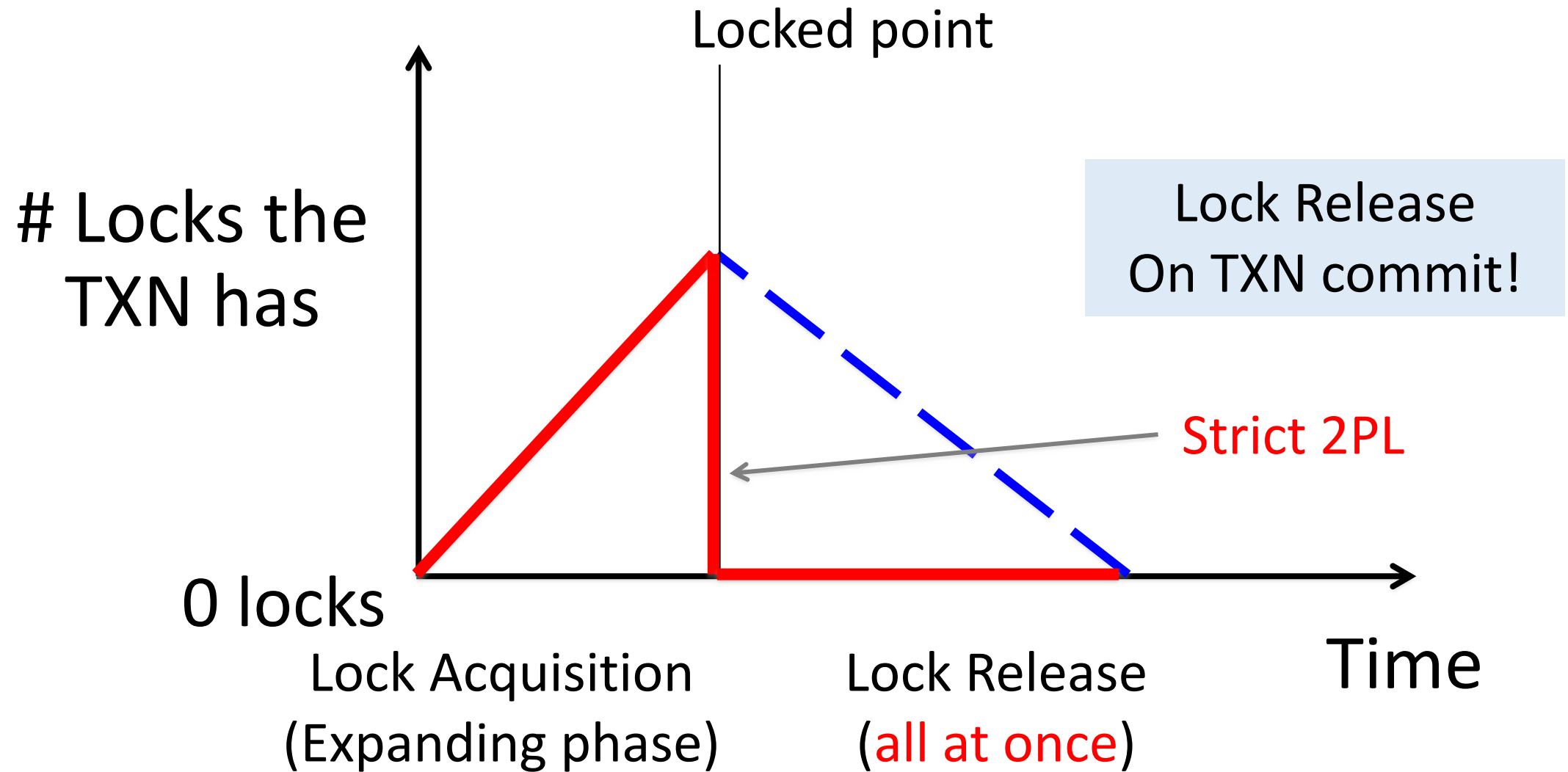
- TXNs obtain:
- An X (exclusive) lock on object before writing
 - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An S (shared) lock on object before reading
 - If a TXN holds, no other TXN can get an X lock on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- “exclusive”, “shared”- meant to be intuitive- no tricks!

Picture of 2-Phase Locking (2PL)

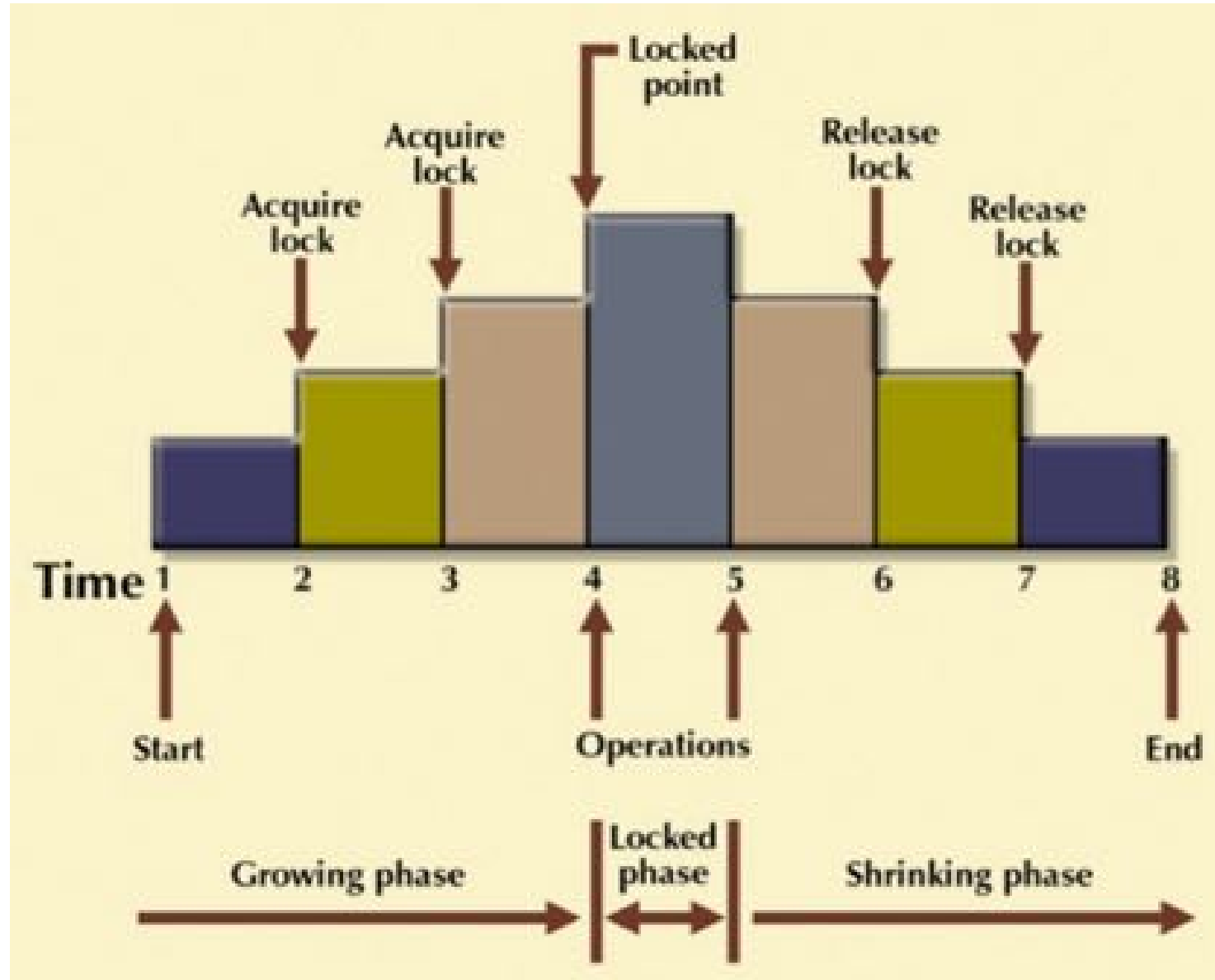


Picture of Strict 2-Phase Locking (S2PL)



Picture of 2-Phase Locking (2PL)

It locks



Strict 2PL

- If a schedule follows strict 2PL and locking, it is conflict serializable...
 - ...and thus serializable
 - ...and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL.
- So let's use strict 2PL, what could go wrong?

Deadlock Detection: Example



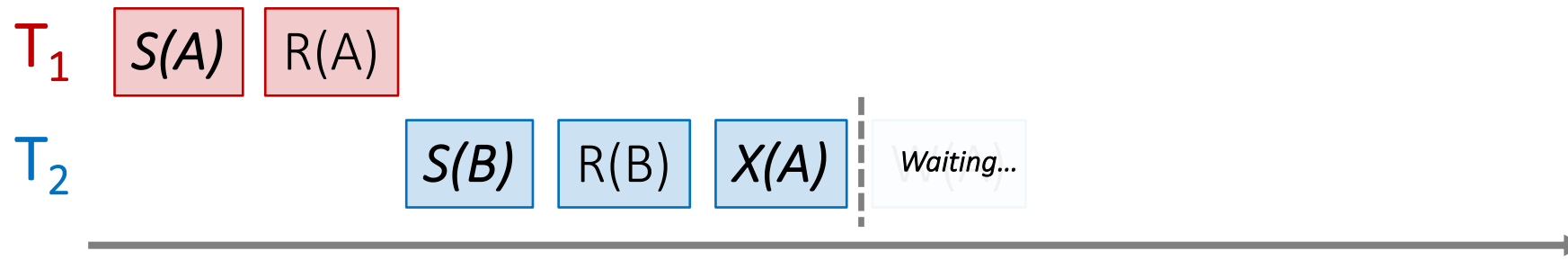
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example



Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

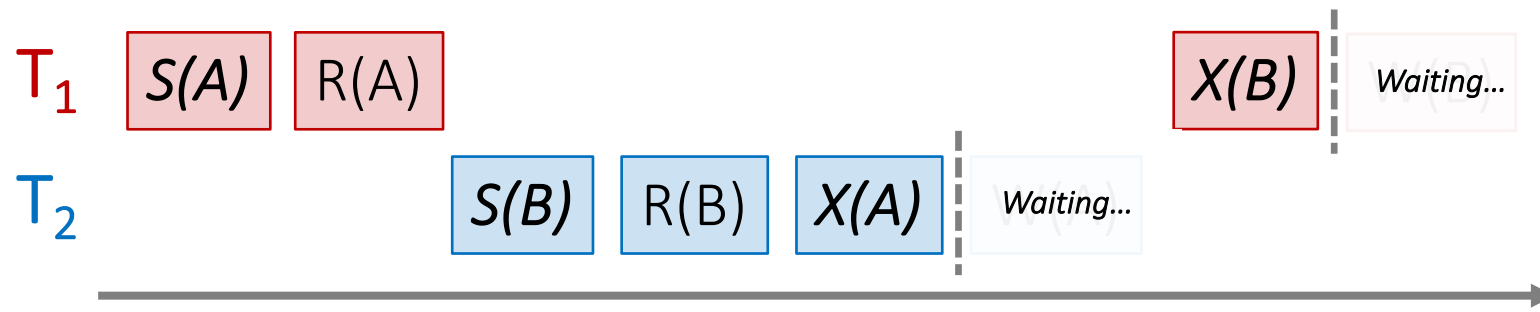


Waits-for graph:



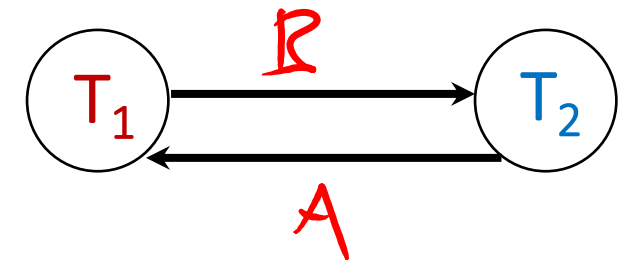
T_2 then requests an exclusive lock on A to write to it- **now** T_2 is waiting on T_1 ...

Deadlock Detection: Example



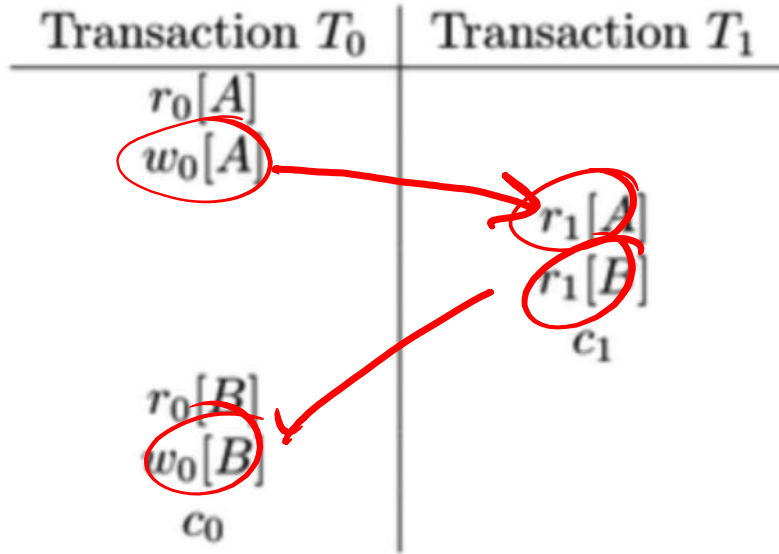
Finally, T_1 requests an exclusive lock on B to write to it- now T_1 is waiting on T_2 ... DEADLOCK!

Waits-for graph:



Cycle =
DEADLOCK

6. Consider the following two transactions and schedule (time goes from top to bottom).
- a. Is this schedule conflict-serializable? Explain why or why not.

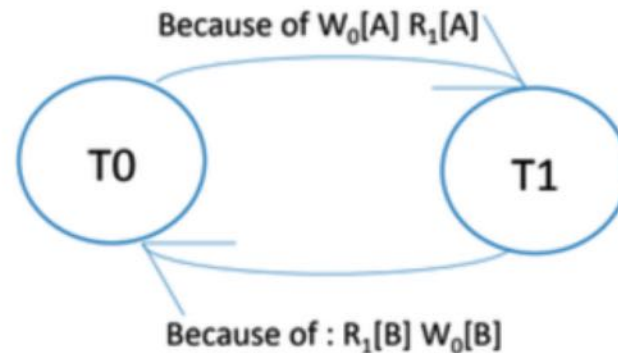


6. Consider the following two transactions and schedule (time goes from top to bottom).
- a. Is this schedule conflict-serializable? Explain why or why not.

Transaction T_0	Transaction T_1
$r_0[A]$	
$w_0[A]$	
	$r_1[A]$
	$r_1[B]$
	c_1
$r_0[B]$	
$w_0[B]$	
c_0	

SOLUTION

The schedule is not conflict serializable because the precedence graph contains a cycle. The graph has an edge $T_0 \rightarrow T_1$ because the schedule contains $w_0[A] \rightarrow r_1[A]$. The graph has an edge $T_1 \rightarrow T_0$ because the schedule contains $r_1[B] \rightarrow w_0[B]$.



6. Consider the following two transactions and schedule (time goes from top to bottom).
- a. Is this schedule conflict-serializable? Explain why or why not.

Transaction T_0	Transaction T_1
$r_0[A]$	
$w_0[A]$	
	$r_1[A]$
	$r_1[B]$
	c_1
$r_0[B]$	
$w_0[B]$	
c_0	

- b. Show how 2PL can ensure a conflict-serializable schedule for the same transactions above. Notice that using locks may change the schedule (e.g. if a transaction attempts to read at a certain moment, but cannot receive the appropriate lock, then it has to wait; this waiting generally leads to a different schedule!) Use the notation $Li(A)$ to indicate that transaction i acquires the lock on element A and $Ui(A)$ to indicate that transaction i releases its lock on A .

6. Consider the following two transactions and schedule (time goes from top to bottom).
- a. Is this schedule conflict-serializable? Explain why or why not.

Transaction T_0	Transaction T_1
$r_0[A]$	
$w_0[A]$	
	$r_1[A]$
	$r_1[B]$
	c_1
$r_0[B]$	
$w_0[B]$	
c_0	

- b. Show how 2PL can ensure a conflict-serializable schedule for the same transactions above. Notice that using locks may change the schedule (e.g. if a transaction attempts to read at a certain moment, but cannot receive the appropriate lock, then it has to wait; this waiting generally leads to a different schedule!) Use the notation $Li(A)$ to indicate that transaction i acquires the lock on element A and $Ui(A)$ to indicate that transaction i releases its lock on A .

SOLUTION

Multiple solutions are possible.

Transaction T_0	Transaction T_1
$L_0[A]$	
$r_0[A]$	
$w_0[A]$	
	$L_1[A] \rightarrow \text{blocks}$
$L_0[B]$	
$r_0[B]$	
$w_0[B]$	
$U_0[A]$	
$U_0[B]$	
c_0	
	$L_1[A] \rightarrow \text{granted}$
	$r_1[A]$
	$L_1[B]$
	$r_1[B]$
	$U_1[A]$
	$U_1[B]$
	c_1

4. NoSQL

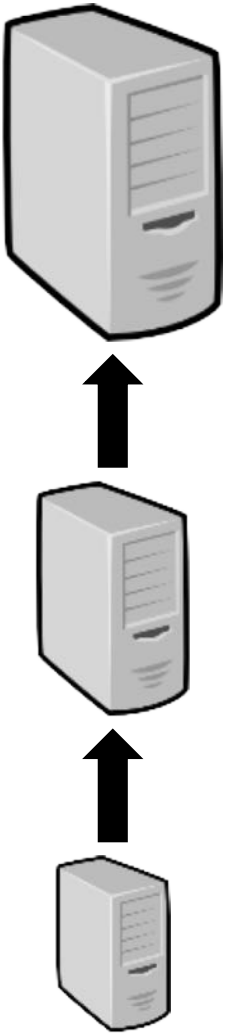
Concluding Remarks on Common NoSQL

- Aim to **avoid join & ACID overhead**
 - Joined within, correctness compromised for quick answers; believe in best effort
- Avoid the idea of a schema
- Query languages are more imperative
 - And less declarative
 - Developer better knows what's going on; less reliance on smart optimization plans
 - More responsibility on developers
- No standard well studied languages (yet)

SQL Means More than SQL

- SQL stands for the query language
- But commonly refers to the traditional RDBMS:
 - Relational storage of data
 - Each tuple is stored consecutively
 - Joins as first-class citizens
 - In fact, normal forms prefer joins to maintenance
 - Strong guarantees on transaction management
 - No consistency worries when many transactions operate simultaneously on common data
- Focus on scaling up
 - That is, make a single machine do more, faster

Vertical vs. Horizontal Scaling



- **Vertical scaling ("scale up"):** you scale by adding more power (CPU, RAM)
- **Horizontal scaling ("scale out"):** you scale by adding more machines



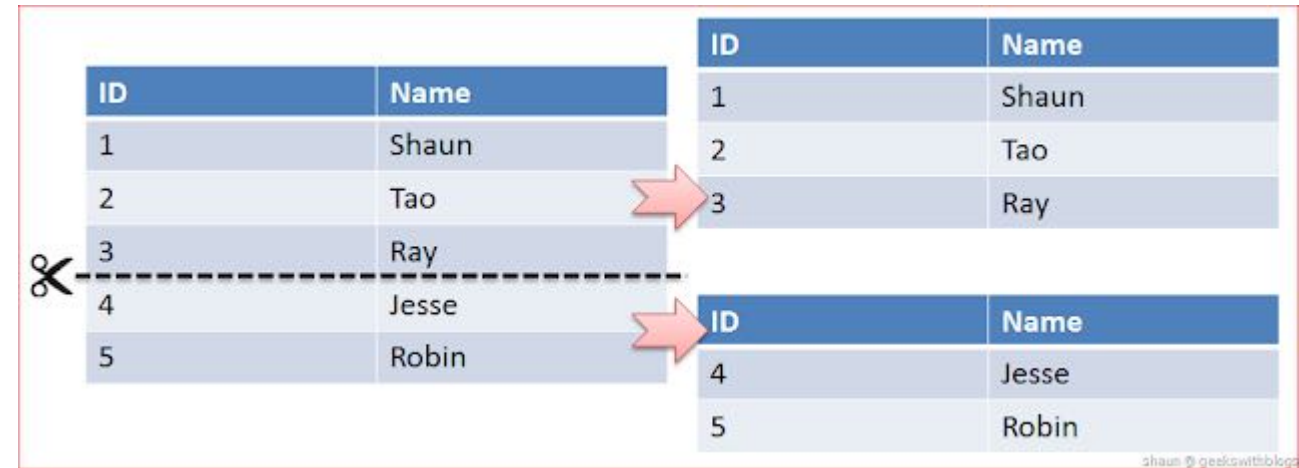
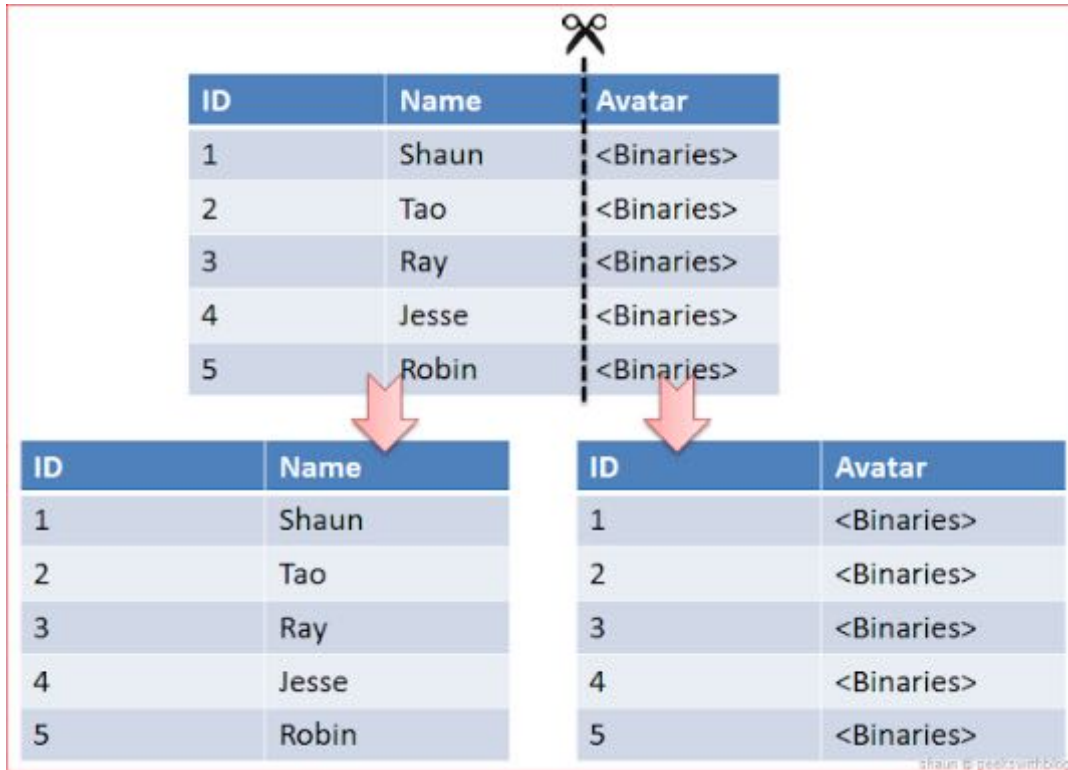
"scaling up"

"scaling out"

Vertical

vs.

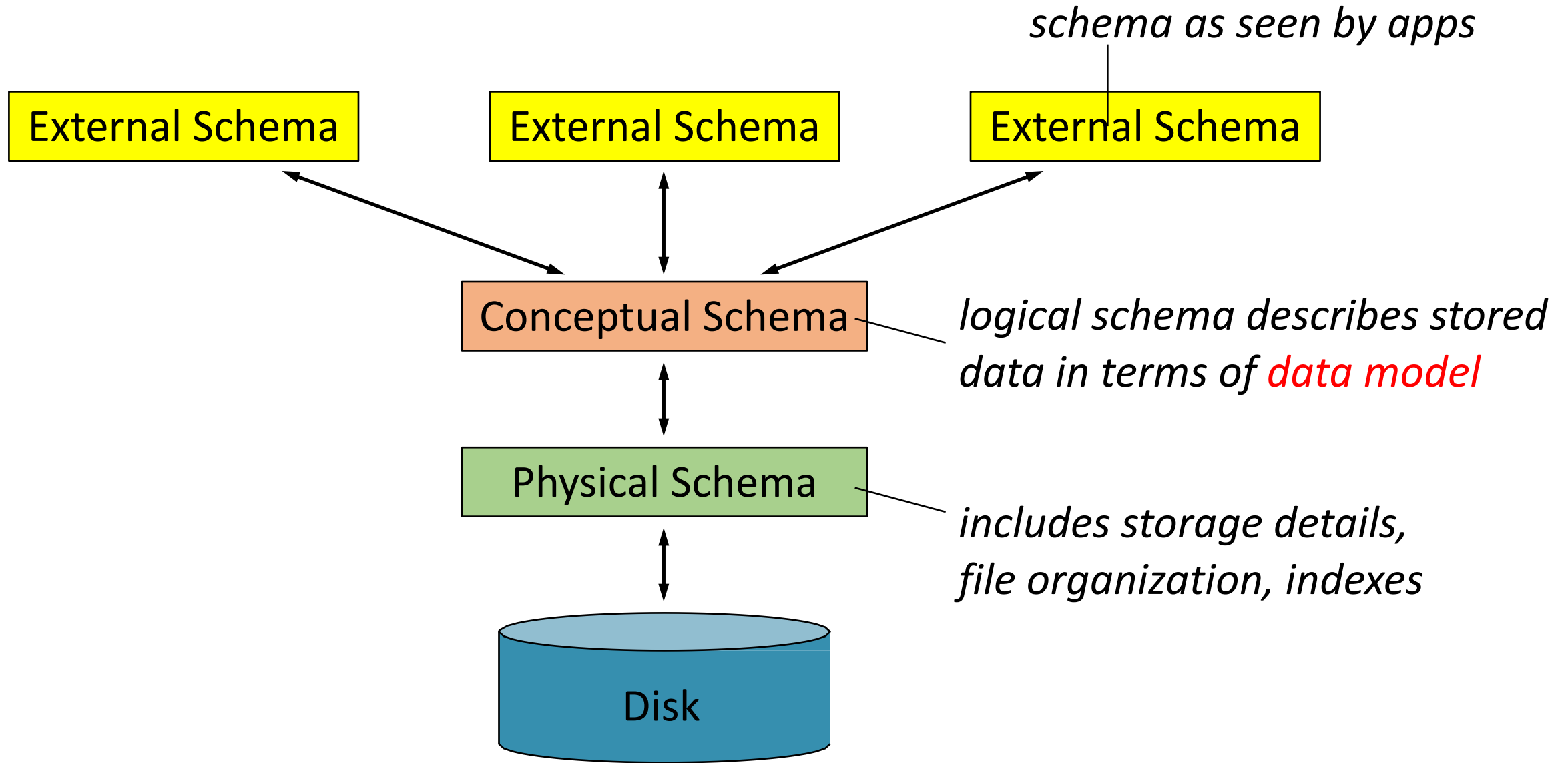
Horizontal partitioning



“Data Model”

- Applications need to model real-world data
 - Typically includes entities and relationships between them
 - Entities: e.g. students, courses, products, clients
 - Relationships: e.g. course registrations, product purchases
- A **data model** enables a user to define the data using high-level constructs without worrying about many low-level details of how data will be stored on disk

Levels of Abstraction



Different Types of Data

- Structured data
- Semistructured data
- Unstructured data

Different Types of Data

- Structured data
 - All data conforms to a schema.
 - Ex: business data
- Semistructured data
 - Some structure in the data but implicit and irregular
 - Ex: resume, ads
- Unstructured data
 - No structure in data.
 - Ex: text, sound, video, images
- What is more important?

Physical Independence

- Definition: Applications are insulated from changes in physical storage details
- Early models (IMS and CODASYL): No
- Relational model: Yes
 - Yes through **set-at-a-time language: algebra or calculus**
 - No specification of what storage looks like
 - Administrator can optimize physical layout

Logical Independence

- Definition: Applications are insulated from changes to logical structure of the data
- Early models
 - IMS: some logical independence
 - CODASYL: no logical independence
- Relational model
 - Yes through views (think fixed SQL queries: give me first and last name of all students)

Great Debate

- Pro relational
 - What where the arguments ?
- Against relational
 - What where the arguments ?

Great Debate

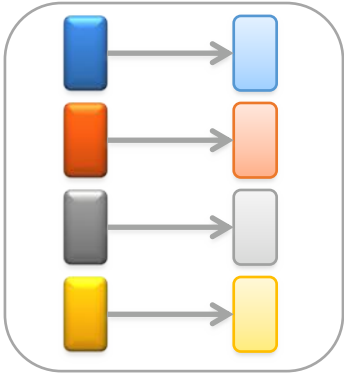
- Pro relational
 - CODASYL is too complex
 - CODASYL does not provide sufficient data independence
 - Record-at-a-time languages are too hard to optimize
 - Trees/networks not flexible enough to represent common cases
- Against relational
 - COBOL programmers cannot understand relational languages
 - Impossible to represent the relational model efficiently
 - CODASYL can represent tables
 - Transitive closure (initially) performance
 - (initially) too complex and mathematical languages
- Ultimately settled by the market place



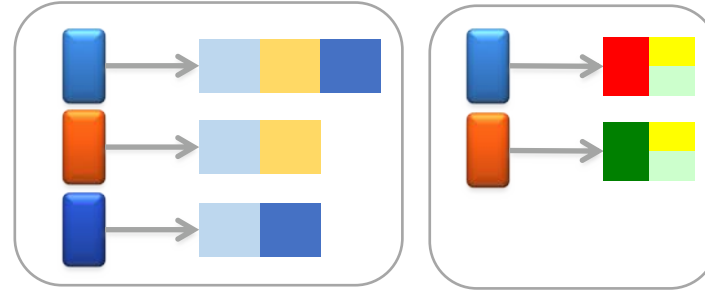
Summary "What goes around ..."

- **Data independence** is desirable
 - Both physical and logical
 - Early data models provided very limited data independence
 - Relational model facilitates data independence
 - **Set-at-a-time** languages facilitate physical indep.
 - Simple data models facilitate logical indep.
- Flat models are also simpler, more flexible
- User should specify what they want not how to get it (declarative)
 - Query optimizer does better job than human
- New data model proposals must
 - Solve a “major pain” or provide significant performance gains

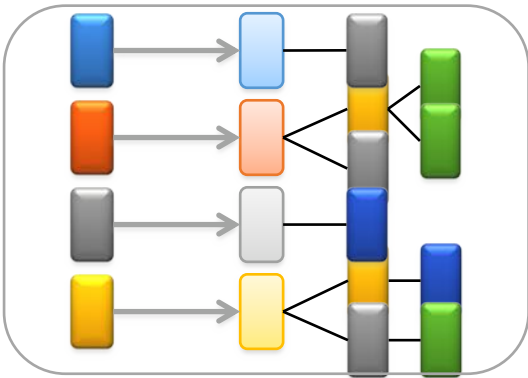
We Will Look at 4 Data Models



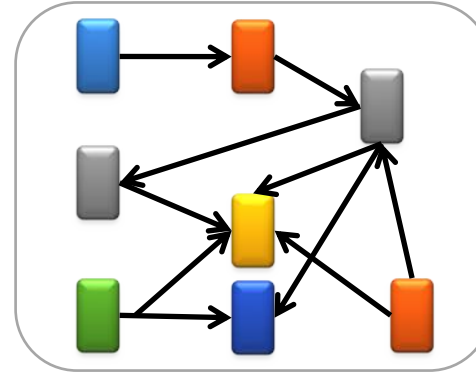
Key/Value Store
(e.g. REDIS)



Column-Family Store
(e.g. Cassandra)



Document Store
(e.g. MongoDB)



Graph Databases (e.g. Neo4J)

ACID May Be Overly Expensive

- In quite a few modern applications:
 - ACID contrasts with key desiderata: high **volume**, high **availability**
 - We can live with **some errors**, to some extent
 - Or more accurately, we prefer to suffer errors than to be significantly less functional
- *Can this point be made more “formal”?*

CAP Service Properties

- **C**onsistency:
 - every read (to any node) gets a response that reflects the most recent version of the data
 - More accurately, a transaction should behave as if it changes the entire state correctly in an instant, Idea similar to serializability
- **A**vailability:
 - every request (to a living node) gets an answer: set succeeds, get returns a value (if you can talk to a node in the cluster, it can read and write data)
- **P**artition tolerance:
 - service continues to function on network failures (cluster can survive
 - As long as clients can reach servers

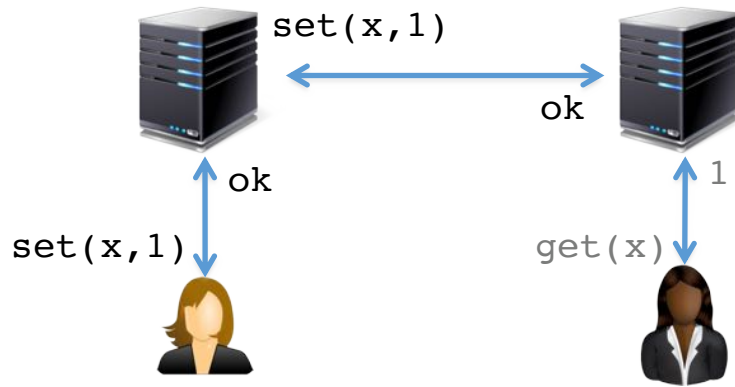
The CAP Theorem

Eric Brewer's CAP Theorem:

*A distributed service
can support **at most two**
out of **C**, **A** and **P***

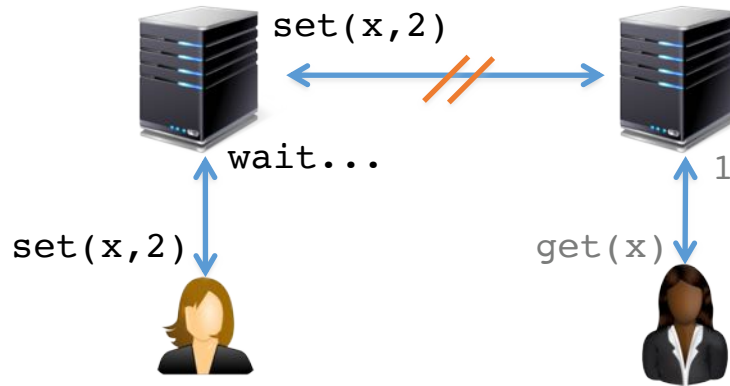
Simple Illustration

Our Relational Database world so far ...



CA

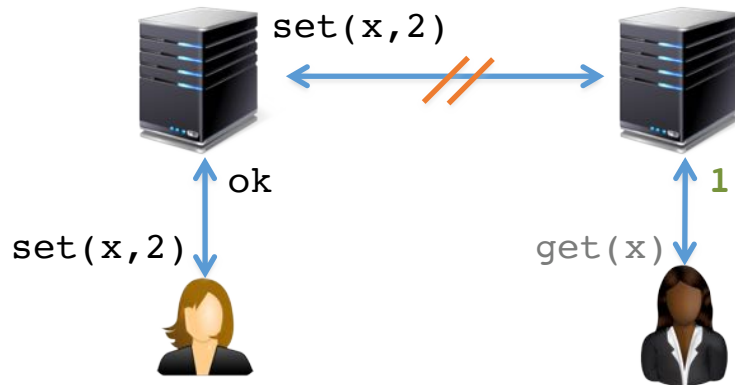
Consistency, Availability



CP

Consistency, Partition tolerance

~~Availability~~

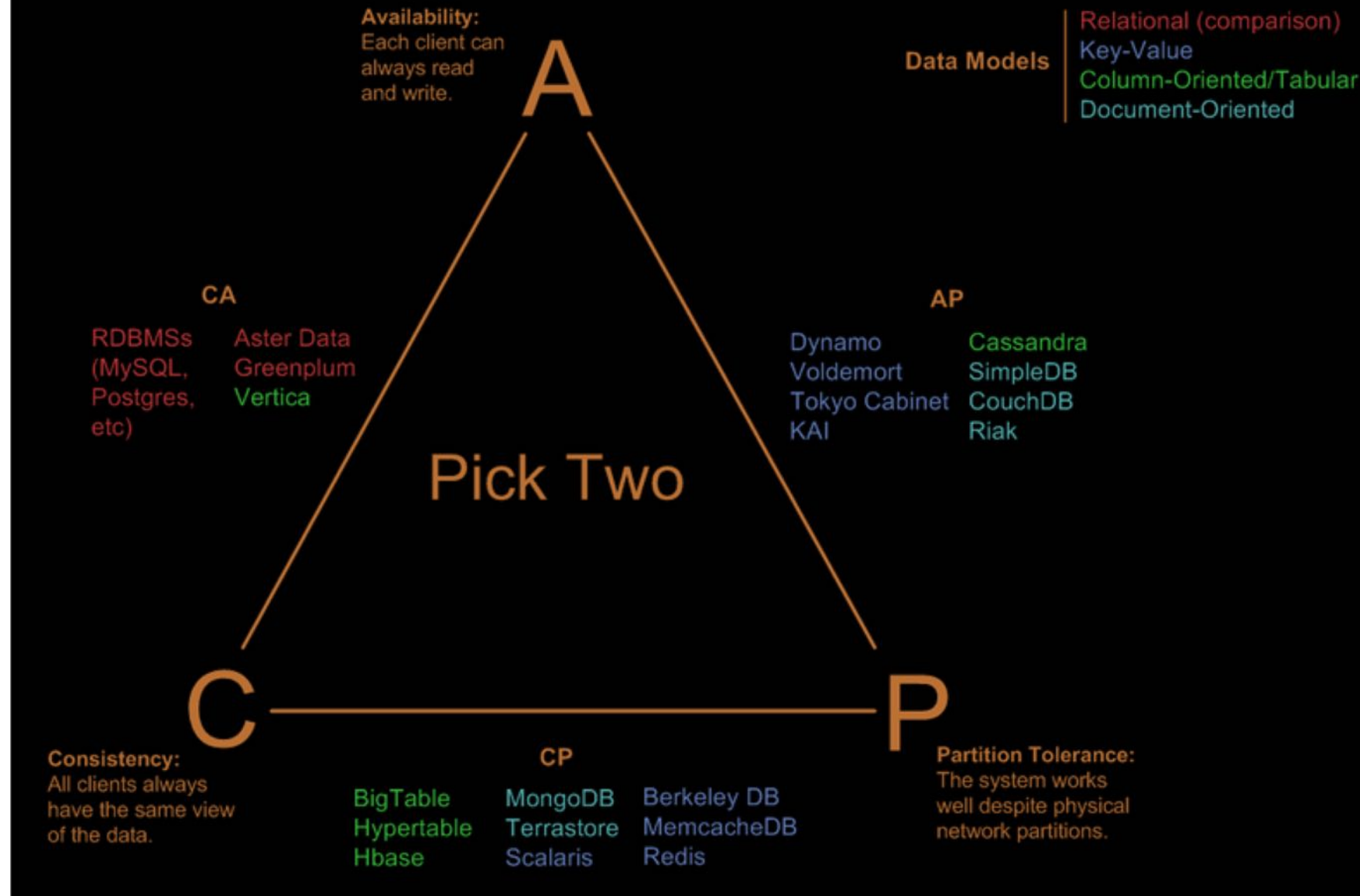


AP

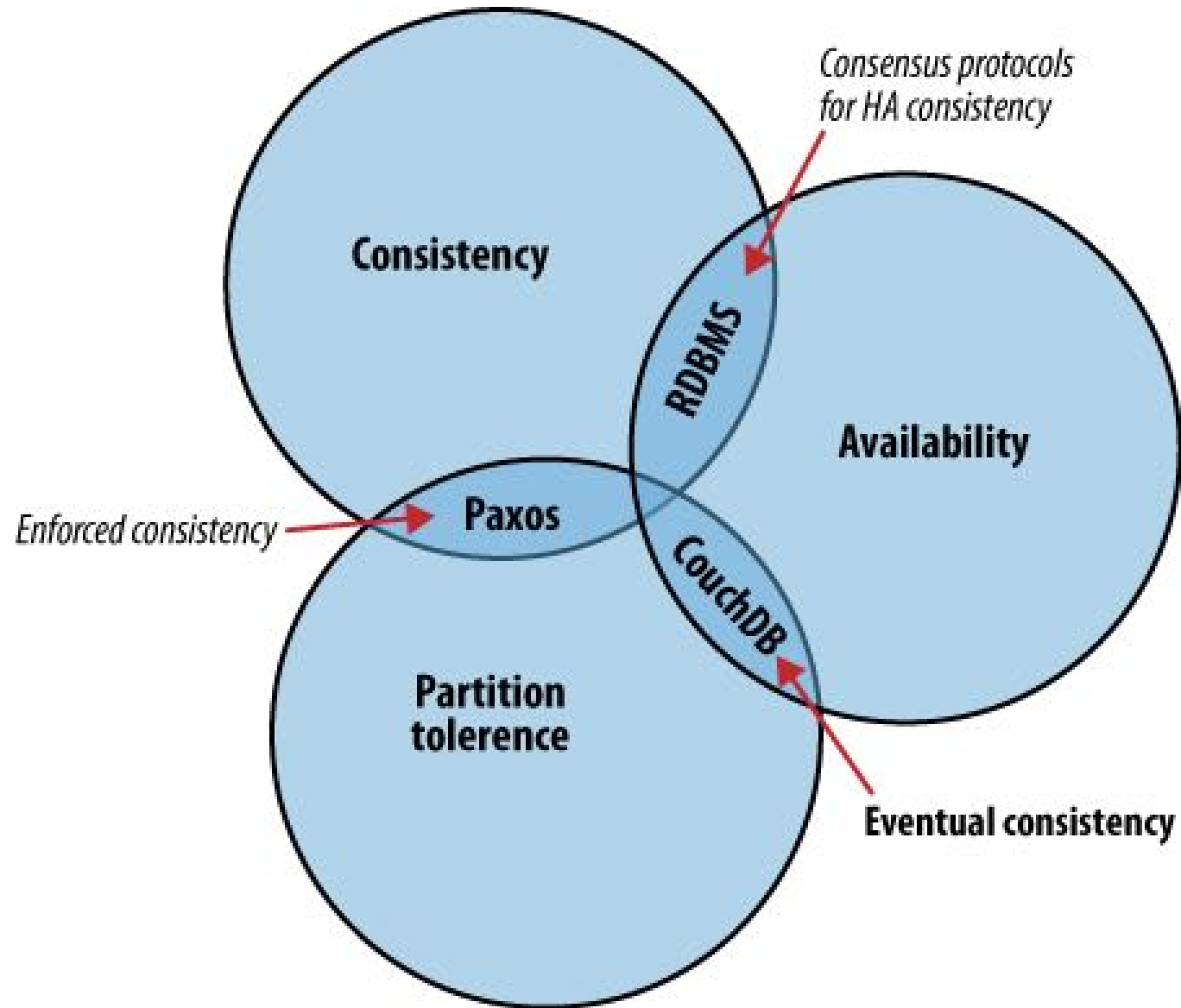
Availability, Partition tolerance

~~Consistency~~

Visual Guide to NoSQL Systems



CAP theorem



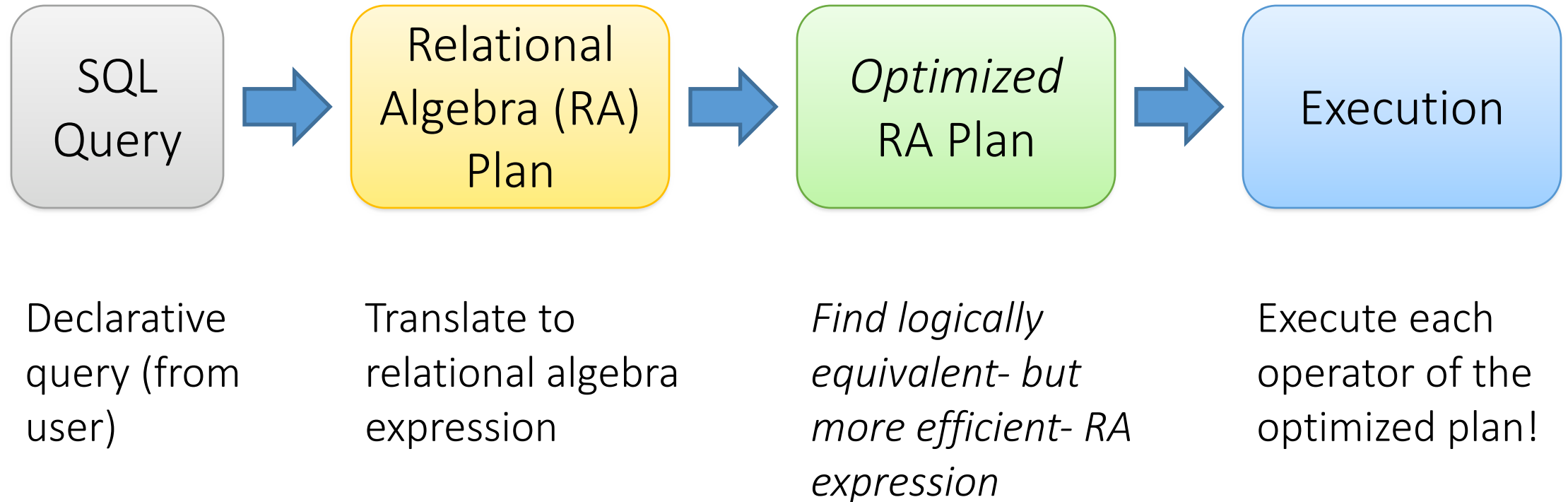
The BASE Model

- Applies to distributed systems of type AP
- **B**asic **A**vailability
 - Provide high availability through distribution: There will be a response to any request. Response could be a 'failure' to obtain the requested data, or the data may be in an inconsistent or changing state.
- **S**oft state
 - Inconsistency (stale answers) allowed: State of the system can change over time, so even during times without input, changes can happen due to 'eventual consistency'
- **E**ventual consistency
 - If updates stop, then after some time consistency will be achieved
 - Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)
- Philosophy: best effort, optimistic, staleness and approximation allowed

5. Relational Algebra

RDBMS Architecture

- How does a SQL engine work ?



Relational Algebra (RA)

- Five basic operators:
 1. Selection: σ
 2. Projection: Π
 3. Cartesian Product: \times
 4. Union: \cup
 5. Difference: $-$
- Auxiliary operators (sometimes counted as basic):
 - Renaming: ρ
- Derived or Intersection, complement
 - Joins (natural, equi-join, theta join, semi-join)
 - Division

Theta Join (\bowtie_{θ})

- A join that involves a predicate
- $R1 \bowtie_{\theta} R2 = \sigma_{\theta}(R1 \times R2)$
- Here θ can be any condition

Note that natural join is a theta join + a projection.

```
Students(sid,sname,gpa)  
People(ssn,pname,address)
```

SQL:

```
SELECT *  
FROM  
    Students, People  
WHERE  $\theta$ ;
```



RA:

$Students \bowtie_{\theta} People$

Semijoin (\bowtie)

- $R \bowtie S = \Pi_{A_1, \dots, A_n} (R \Join S)$
 - where A_1, \dots, A_n are the attributes in R
- Intuition:
 - remove "dangling tuples"
 - Whiteboard: how to specify in SQL?
 - 3 variants
- Example:
 - Employee \bowtie Dependents

```
Students(sid, sname, gpa)  
People(ssn, pname, address)
```

SQL:

```
SELECT DISTINCT  
    sid, sname, gpa  
FROM  
    Students, People  
WHERE  
    sname = pname;
```

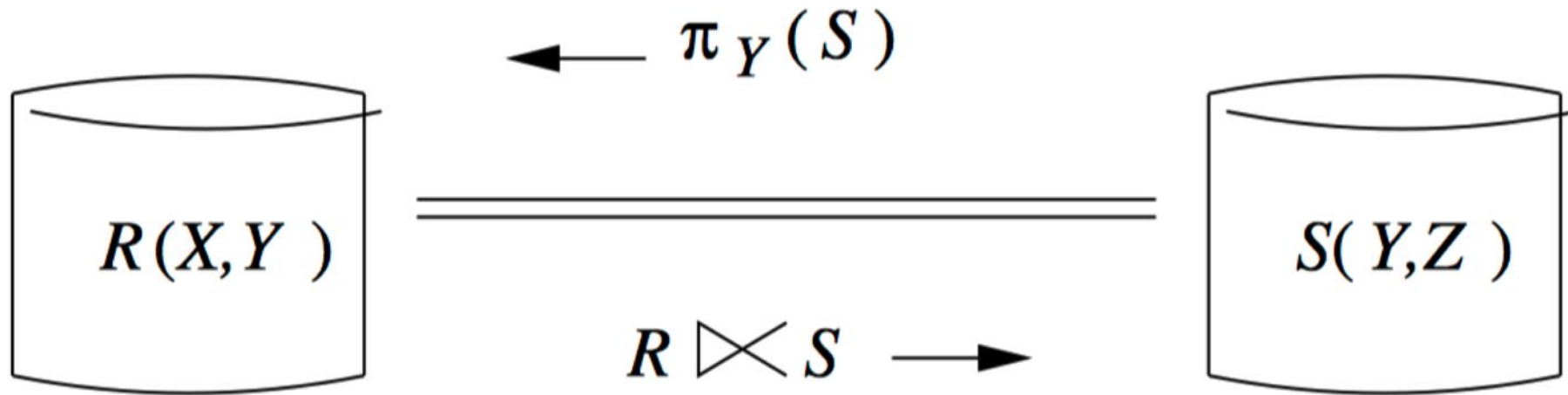


RA:

Students \bowtie People

Semijoins in Distributed Databases

- Semijoins are often used to compute natural joins in distributed databases



Send less data to
reduce network
bandwidth!

Division

- Consider two relations $R(X, Y)$ and $S(Y)$
 - Here, X and Y are tuples of attributes
- $R \div S$ is the relation $T(X)$ that contains all the X s that occur with *every* Y in S

Formal Definition

- **Legal input:** (R, S) such that R has all the attributes of S
- $R \div S$ is the relation T with:
 - The header of R , with all attributes of S removed
 - Tuple set $\{t[X] \mid t[X, Y] \in R \text{ for every } s[Y] \in S\}$
 - *This is an abuse of notation, since the attributes in X need not necessarily come before those of Y*

Questions

sid	student	course
861	Alma	DB
861	Alma	PL
753	Amir	DB
753	Amir	AI
955	Ahuva	PL
955	Ahuva	DB
955	Ahuva	AI

course
AI

÷

course
AI

=

?

course
DB
PL
AI

÷

course
DB
PL
AI

=

?

$$(R \times S) \div S = ?$$

$$(R \times S) \div R = ?$$

Q: If R has 1000 tuples and S has 100 tuples, how many tuples can be in $R \div S$?

Q: If R has 1000 tuples and S has 1001 tuples, how many tuples can be in $R \div S$?

Studies

sid	student	course
861	Alma	DB
861	Alma	PL
753	Amir	DB
753	Amir	AI
955	Ahuva	PL
955	Ahuva	DB
955	Ahuva	AI

CourseType

course	type
DB	core
PL	core
AI	elective
DC	elective

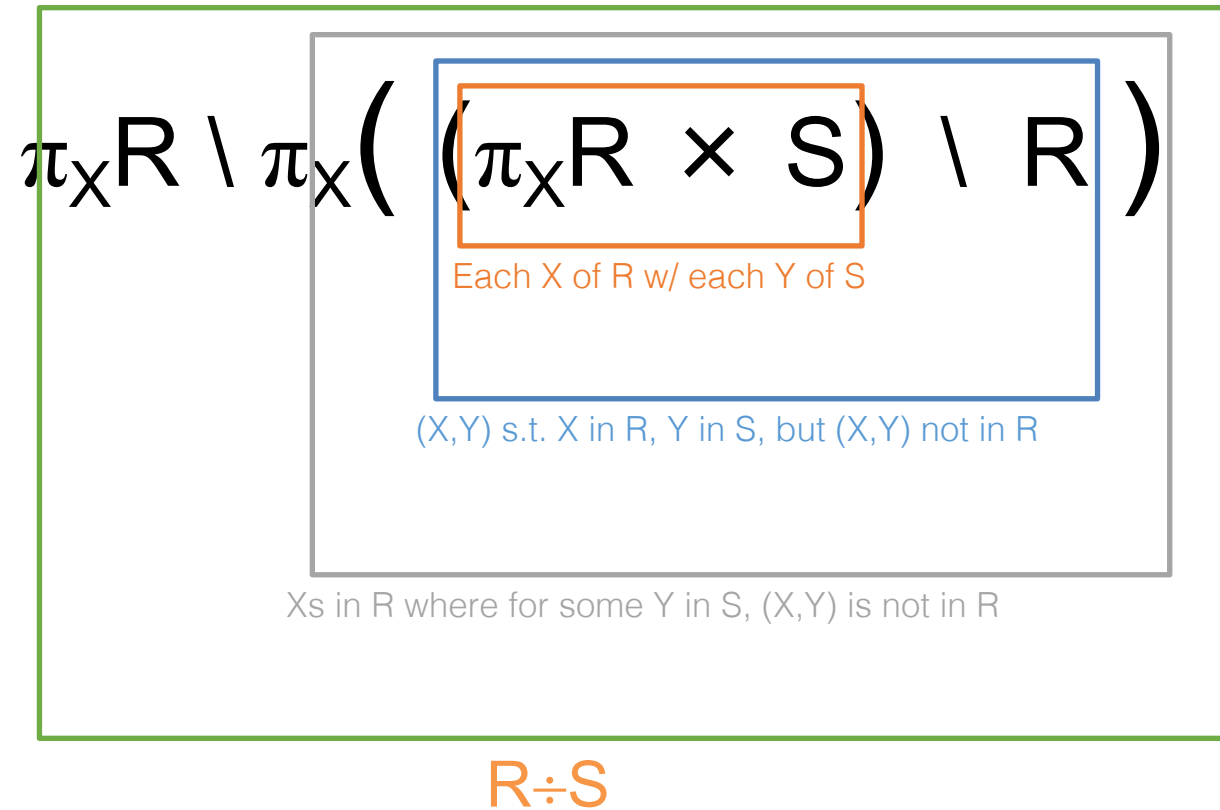
Who took all core courses?

Studies $\div \pi_{\text{course}} \sigma_{\text{type}='core'} \text{CourseType}$

$R \div S$ in Primitive RA

\neg \cap \cup \setminus \rightarrow

$R(X,Y) \quad S(Y)$



Exercise



Student

sid	name	year
861	Alma	2
753	Amir	1
955	Ahuva	2

Course

cid	topic
23	PL
45	DB
76	OS

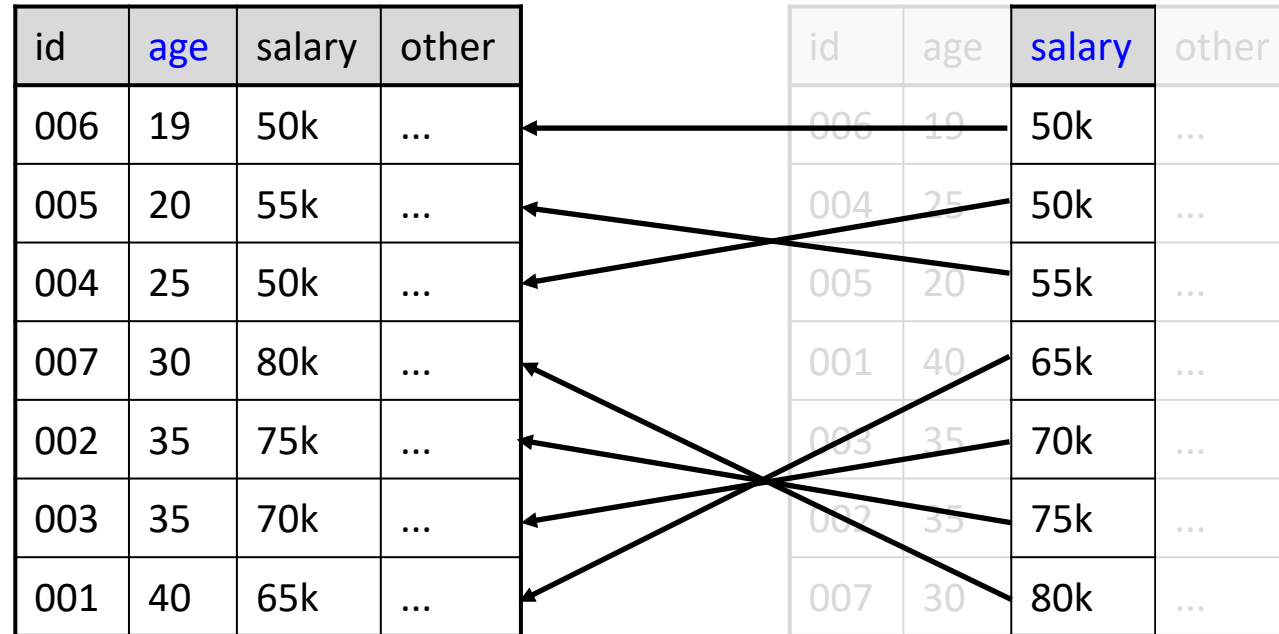
Studies

sid	cid
861	23
861	45
753	45

Write a query in RA that finds the names of students who get "private lessons"

(i.e., the student takes a course that no one else takes)

Recap: High-level overview: indexes



data file = index file
clustered (primary) index

index file
unclustered (secondary) index

Questions on final exam & grading
... or anything else?