CS 145 Midterm Review

Materials to know

Know your SQL	Select-Project-Aggregate-Having, Nested queries, NULL values Set vs multi-sets
How to design ‘good’ tables?	Find bad FDs, Compute closures/superkeys, Do BCNF decompositions, Find MVDs
ACID transactions	Performance #s (disk seeks, memory access) WAL logs for a transaction? How to recover post crash? (Recap: WAL & performance, Why is a log COMMIT fast?)
	Is a schedule conflict serializable? How to build Conflict graph? What happens during 2PL execution? (Recap: With a 5 Transaction example)

1. Summary recap on a few key topics (see main lectures !!)
2. Mainly, honing a few popular questions in piazza/OH

(Start with transactions and go backwards)

Transactions

Summary

"Need to Master Extreme Transactions" (Forbes (Insights))

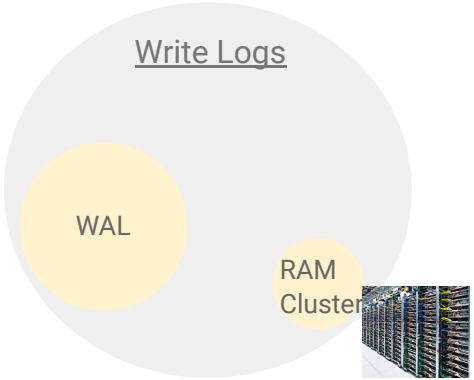
Why study Transactions?

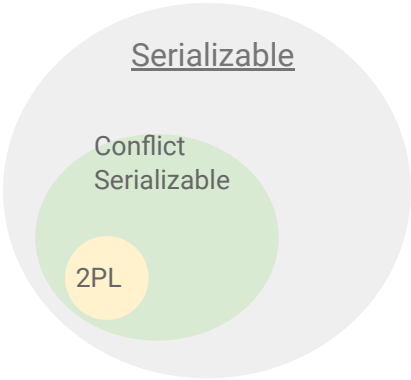
Good programming model for parallel applications on shared data !

- Atomic
- Consistent
- Isolation
- Durable

Design choices?

- Write update Logs (e.g., WAL logs)
- Serial? Parallel, interleaved and serializable?





Latency numbers every engineer should know

Ballpark timings

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

(~0.25 msecs)

(~10 msecs)

(~20 msecs)

Write-Ahead Logging (WAL)

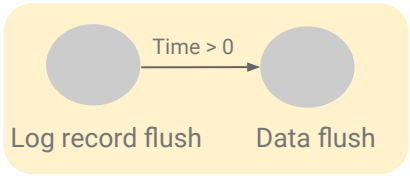
Algorithm: WAL

1. Must *force log record* for an update *before* the corresponding data page goes to storage

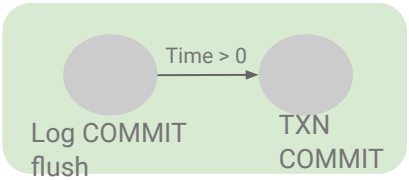
2. Must write all log records for a TX before commit

→ **Atomicity**

→ **Durability**



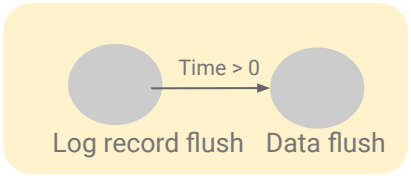
For each record update



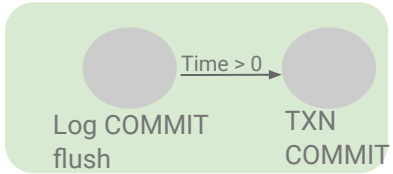
At end of TXN

Example WAL scenarios

TXN commit before Log COMMIT on disk?	No
Data page on disk before Log flush for that record?	No
TXN commit before data page flushed to disk?	Yes, often. Especially for large transactions. For TXN Commit, should have Flushed... - All record updates to Log - COMMIT record to Log
Later transaction needing updated data, not flushed to disk	Check memory for latest. If not in memory, read latest from disk
Parallel transaction looking at updated data, not flushed to disk	(Later, 2PL or Locking kicks in for RW, WW, WR conflicts)



For each record update



At end of TXN

Example

Monthly bank interest transaction

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

‘T-Monthly-423’

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run

```
START TRANSACTION
        UPDATE Money
        SET Balance = Balance * 1.1
COMMIT
```

Example

Monthly bank interest transaction

With crash

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@10:45 am)

Account	....	Balance ($)	
3001		550	??
4001		110	
5001		22	
6001		66	
3002		88	
4002		-200	??
5002		320	??
...			
30108		-110	
40008		110	
50002		22	??

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

‘T-Monthly-423’

Monthly Interest 10%
4:28 am Starts run on 10M bank accounts
Takes 24 hours to run
Network outage at 10:29 am,
System access at 10:45 am

Did T-Monthly-423 complete?
Which tuples are bad?

Case1: T-Monthly-423 was crashed
Case2: T-Monthly-423 completed. 4002 deposited 20$ at 10:45 am

Example

Monthly bank interest transaction

Recovery

Money (@10:45 am)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-200
5002		320
...		
30108		-110
40008		110
50002		22

Money (after recovery)

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

WAL log (@10:29 am)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352

System recovery (after 10:45 am)

1 Rollback uncommitted transactions
- Restore old values from WALlog (if any)
- Notify developers about aborted txn

1.1 Redo Recent committed transactions (w/ new values)
2 Back in business
3 Redo (any pending) transactions

(Sometimes swap 2 and 3, as a tradeoff)

Example

Monthly bank interest transaction

Performance

Money

Account	....	Balance ($)
3001		500
4001		100
5001		20
6001		60
3002		80
4002		-200
5002		320
...		...
30108		-100
40008		100
50002		20

Money (@4:29 am day+1)

Account	....	Balance ($)
3001		550
4001		110
5001		22
6001		66
3002		88
4002		-220
5002		352
...		
30108		-110
40008		110
50002		22

WAL (@4:29 am day+1)

T-Monthly-423	START TRANSACTION		
T-Monthly-423	3001	500	550
T-Monthly-423	4001	100	110
T-Monthly-423	5001	20	22
T-Monthly-423	6001	60	66
T-Monthly-423	3002	80	88
T-Monthly-423	4002	-200	-220
T-Monthly-423	5002	320	352
T-Monthly-423	...	...	...
T-Monthly-423	30108	-100	-110
T-Monthly-423	40008	100	110
T-Monthly-423	50002	20	22
T-Monthly-423	COMMIT		

Cost to update all data

10M bank accounts → 10M seeks? (worst case)

(@10 msec/seek, that's 100,000 secs)

Cost to Append to log

+ 1 seek to get 'end of log'
+ write 10M log entries sequentially (fast!)
(< 1 sec)

[Lazily update data on disk later, when convenient.]

Materials to know

A few questions

Know your SQL	Select-Project-Aggregate-Having, Nested queries, NULL values Set vs multi-sets
How to design ‘good’ tables?	Find bad FDs, Compute closures/superkeys, Do BCNF decompositions, Find MVDs
ACID transactions	Performance #s (disk seeks, memory access) WAL logs for a transaction? How to recover post crash? (Recap: WAL & performance, Why is a log COMMIT fast?)
	Is a schedule conflict serializable? How to build Conflict graph? What happens during 2PL execution? (Recap: With a 5 Transaction example)

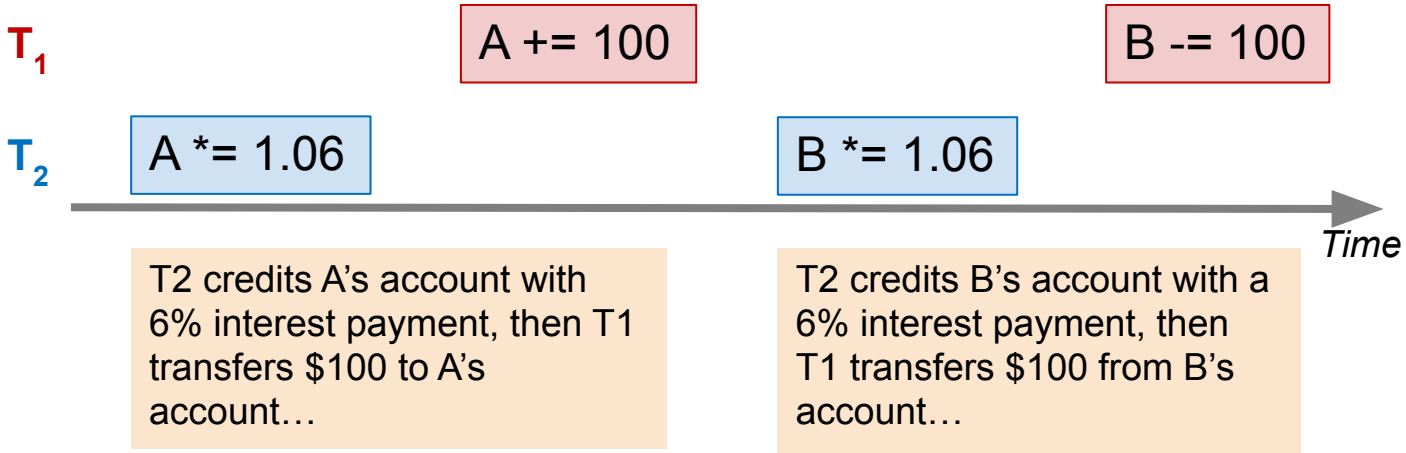
(Recap will start with transactions and go backwards)

Example- consider two TXNs:

T_1 A += 100 | B -= 100

T_2 A *= 1.06 | B *= 1.06

The DBMS can **interleave** the TXNs



Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions
- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B
- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some"** makes this def powerful and tricky!

Conflict Types

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

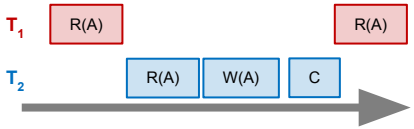
- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Why no "RR Conflict"?

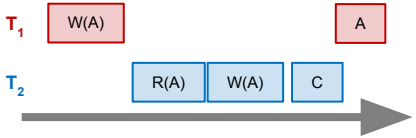
Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

Classic Anomalies with Interleaved Execution

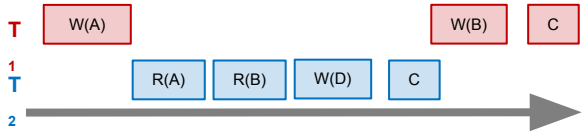
“Unrepeatable read”:



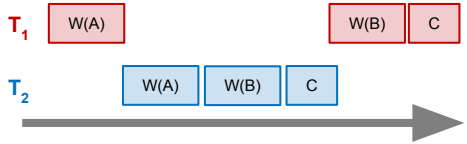
“Dirty read” / Reading uncommitted data:



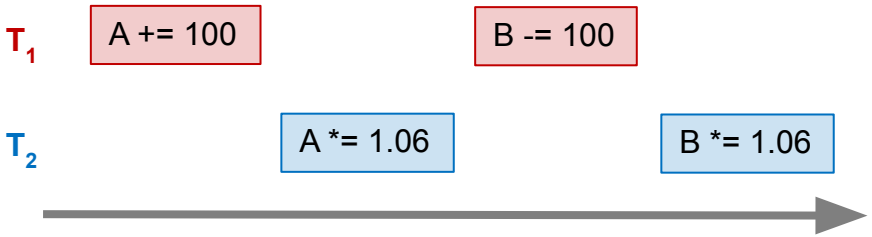
“Inconsistent read” / Reading partial commits:



Partially-lost update:



Serializable?



Serial schedules:

	A	B
$T_1 \rightarrow T_2$	1.06*(A+100)	1.06*(B-100)
$T_2 \rightarrow T_1$	1.06*A + 100	1.06*B - 100

A	B
1.06*(A+100)	1.06*(B-100)

Same as a serial schedule ***for all possible values of A, B*** = **serializable**

Serial Schedules

T1		A += 100	B -= 100		
T2				A *= 1.06	B*= 1.06

S1

T1				A += 100	B -= 100
T2		A *= 1.06	B *= 1.06		

S2

Interleaved Schedules

T1		A += 100		B -= 100	
T2			A *= 1.06		B*= 1.06

S3

T1			A += 100		B -= 100
T2		A *= 1.06		B *= 1.06	

S4

T1			A += 100	B -= 100	
T2		A *= 1.06			B*= 1.06

S5

T1		A += 100			B -= 100
T2			A *= 1.06	B *= 1.06	

S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

Example with 5 Transactions

Schedule S1

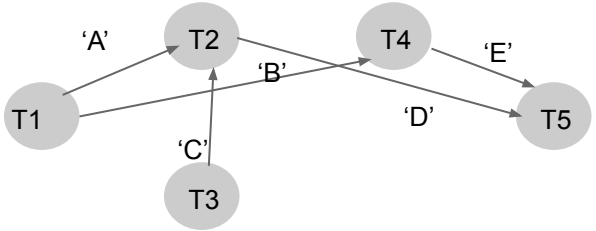
w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Good or Bad schedule?
Conflict serializable?

Step1
Find conflicts
(RW, WW, WR)

T1	w1(A)		w1(B)							
T2		r2(A)			r2(C)		w2(D)			
T3				w3(C)						
T4						r4(B)		w4(E)		
T5									r5(D)	w5(E)

Step2
Build Conflict graph
Acyclic? Topo Sort



Acyclic
⇒ Conflict serializable!
⇒ Serializable

Step3
Example serial schedules
Conflict Equiv to S1

SerialSched (SS1)

T3	T1	T4	T2	T5					
w3(C)	w1(A)	w1(B)	r4(B)	w4(E)	r2(A)	r2(A)	w2(D)	r5(D)	w5(E)

SerialSched (SS2)

T1	T3	T2	T4	T5					
w1(A)	w1(B)	w3(C)	r2(A)	r2(A)	w2(D)	r4(B)	w4(E)	r5(D)	w5(E)

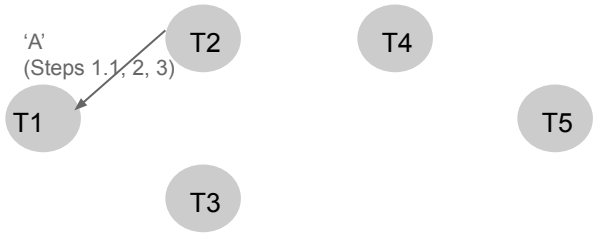
Example with 5 Transactions (2PL)

Schedule S1

w1(A)	r2(A)	w1(B)	w3(C)	r2(C)	r4(B)	w2(D)	w4(E)	r5(D)	w5(E)

Execute with 2PL

	T1	T2	T3	T4	T5
Step 1	X (A) w1(A)				
Step 1.1		Req S(A)			
Step 2	X (B) w1(B) Unl B, A				
Step 3			X (C) w3(C) Unl C		
Steps 4, 5		Get S(A) r2(A) S(C) r2(C)			
Step 6				S(B) r4(B)	
Step 7		X(D) w2(D) Unl A, C, D			
Step 8				X(E) w4(E) Unl B,E	
Step 9					S (D) r5(D)
Step 10					X (E) w5(E) Unl D, E



Waits- For Graph

Materials to know

A few questions

Know your SQL	Select-Project-Aggregate-Having, Nested queries, NULL values Set vs multi-sets
How to design ‘good’ tables?	Find bad FDs, Compute closures/superkeys, Do BCNF decompositions, Find MVDs
ACID transactions	Performance #s (disk seeks, memory access) WAL logs for a transaction? How to recover post crash? (Recap: WAL & performance, Why is a log COMMIT fast?)
	Is a schedule conflict serializable? How to build Conflict graph? What happens during 2PL execution? (Recap: With a 5 Transaction example)

(Recap will start with transactions and go backwards)

Example Enrollment table - “v0”

SID	Class	Room	Time	Lat	Lng
4749732	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
2720942	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4823984	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4287594	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2984994	cs 145	Nvidia Aud	T/R 4:30-6	...	...
8472374	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4723663	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2478239	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4763268	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364532	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364573	cs 145	Nvidia Aud	T/R 4:30-6	...	...
3476382	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2347623	cs 145	Nvidia Aud	T/R 4:30-6	...	...
...	...	...	...	...	...
2364579	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
3476343	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
2322232	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W

~375 cs145 students

~300 cs245 students

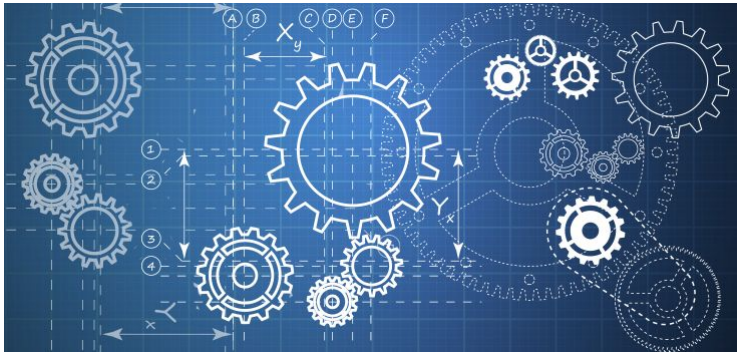
<u>Problems</u>
Repeats?
Room/time change?
Deletes?

<u>Properties</u>
Class -> Room/time
Room -> Lat, Lng

(more compact)

Design Theory

- Design theory is about how to represent your data to avoid ***anomalies***.
- Simple algorithms for “best practices”

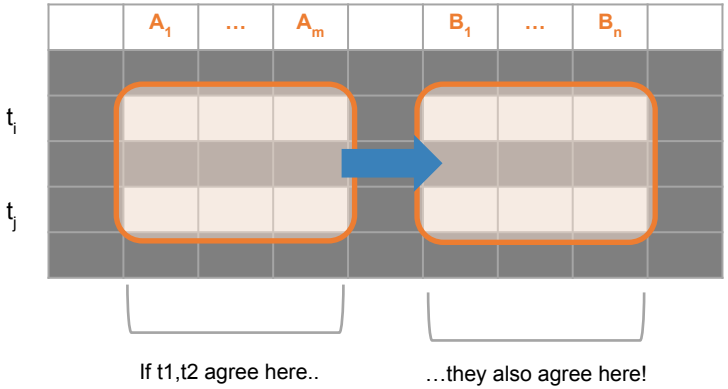


Relational Schema Design

High-level idea

1. Start with some relational *schema*
2. Find out its *functional dependencies (FDs)*
3. Use these to *design a better schema*
   One which minimizes the possibility of anomalies

A Picture Of FDs



Defn (again):
Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R,**

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

if $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

then $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2]=t_j[B_2]$ AND … AND $t_i[B_n] = t_j[B_n]$

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called
Armstrong's Rules.

1. **Split/Combine,**
2. **Reduction, and**
3. **Transitivity…** *ideas by picture*

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Department}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	Trivial
5. {Name, Category} -> {Color}	Transitive (4 -> 1)
6. {Name, Category} -> {Category}	Trivial
7. {Name, Category} -> {Color, Category}	Split/Combine (5 + 6)
8. {Name, Category} -> {Price}	Transitive (7 -> 3)

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

What's an algorithmic way to do this?

Closure of a set of Attributes

Given a set of attributes $\mathbf{A_1, \ldots, A_n}$ and a set of FDs **F:**
Then the **closure**, $\mathbf{\{A_1, \ldots, A_n\}^+}$ is the set of attributes **B** s.t. $\mathbf{\{A_1, \ldots, A_n\} \rightarrow B}$

Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

Example: F =

{name} → {color}
{category} → {department}
{color, category} → {price}

Example Closures:

$\{name\}^+ = \{name, color\}$
$\{name, category\}^+ =$
{name, category, color, dept, price}
$\{color\}^+ = \{color\}$

Keys and Superkeys

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

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

A **key** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

Algorithm: For each set of attributes X

1. Compute X^+
2. If X^+ = set of all attributes then X is a **superkey**
3. If X is minimal, then it is a **key**

Conceptual Design

For a “mega” table

- Search for “bad” dependencies
- If any, *keep decomposing (lossless) the table into sub-tables* until no more bad dependencies
- When done, the database schema is *normalized*

Recall: there are several normal forms…

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is **in BCNF** if:

if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, ..., A_n\}$ **is a superkey** for R

In other words: there are no “bad” FDs

BCNF Decomposition Algorithm

Algorithm: BCNFDecomp(R):
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

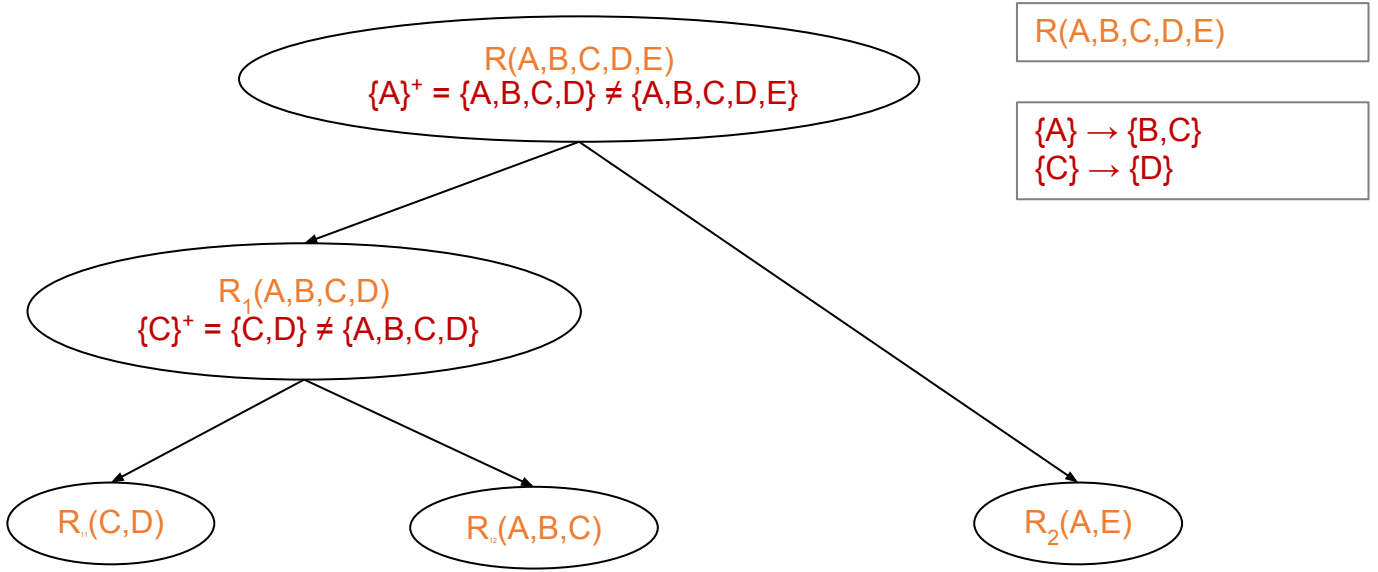
if (not found) **then Return** R

decompose R into $\mathbf{R_1(X^+)}$ and $\mathbf{R_2(X \cup Rest)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no more “bad” FDs!

Example


R(A,B,C,D,E)
{A}+ = {A,B,C,D} ≠ {A,B,C,D,E}
R1(A,B,C,D)
{C}+ = {C,D} ≠ {A,B,C,D}
R11(C,D)
R12(A,B,C)
R2(A,E)
R(A,B,C,D,E)
{A} → {B,C}
{C} → {D}

Example Enrollment table - “v0”

	SID	Class	Room	Time	Lat	Lng
	4749732	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
	2720942	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
	4823984	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
	4287594	cs 145	Nvidia Aud	T/R 4:30-6	...	...
~375 cs145 students	2984994	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	8472374	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	4723663	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	2478239	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	4763268	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	2364532	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	2364573	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	3476382	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	2347623	cs 145	Nvidia Aud	T/R 4:30-6	...	...
	...	...	...	...	...	...
	2364579	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
~300 cs245 students	3476343	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
	2322232	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W

FDs

Class -> Room,Time

Room -> Lat, Lng

(more compact)

Example Enrollment table - “v0”

BCNF decomposition

SID	Class	Room	Time	Lat	Lng
4749732	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
2720942	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4823984	cs 145	Nvidia Aud	T/R 4:30-6	37.4277° N	122.1742° W
4287594	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2984994	cs 145	Nvidia Aud	T/R 4:30-6	...	...
8472374	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4723663	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2478239	cs 145	Nvidia Aud	T/R 4:30-6	...	...
4763268	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364532	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2364573	cs 145	Nvidia Aud	T/R 4:30-6	...	...
3476382	cs 145	Nvidia Aud	T/R 4:30-6	...	...
2347623	cs 145	Nvidia Aud	T/R 4:30-6	...	...
...	...	...	...	...	...
2364579	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
3476343	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W
2322232	cs 245	Nvidia Aud	T/R 3-4:30	37.4277° N	122.1742° W

Schema: SID, Class, Room, Time, Lat, Lng

Key

SID, Class

FDs

Class -> Room,Time

Room -> Lat, Lng

BCNF decomposition

1. Find bad FD #1: Class$^+$ -> Class, Room,Time, Lat, Lng
   Decomposed: R1(Class, Room, Time, Lat, Lng) and R2(SID, Class)
2. Find bad FD #2: Room$^+$ -> Room, Lat, Lng
   Decompose R1 into R11(Room, Lat, Lng) and R12(Class, Room, Time)

⇒ BCNF schema: R2(SID, Class), R12(Class, Room, Time), R11(Room, Lat, Lng)

Example Enrollment table - “v1”

SID	Class
4749732	cs 145
2720942	cs 145
4823984	cs 145
4287594	cs 145
2984994	cs 145
8472374	cs 145
4723663	cs 145
2478239	cs 145
4763268	cs 145
2364532	cs 145
2364573	cs 145
3476382	cs 145
2347623	cs 145
...	...
2364579	cs 245
3476343	cs 245
2322232	cs 245

375 cs145 students

300 cs245 students

Class	Room	Time
cs 145	Nvidia Aud	T/R 4:30-6
cs 245	Nvidia Aud	T/R 3-4:30
cs 246	Nvidia Aud	M/W 3-4:30

Room	Lat	Lng
Nvidia Aud	37.4277° N	122.1742° W

Example: Student profile

(MVDs)

"Student 4749732 is taking
Classes = {cs145, cs 245, cs 222},
Hobbies = {Surfing, Music, Astronomy}"

SID	Class	Hobby
4749732	cs 145	Surfing
4749732	cs 245	Surfing
4749732	cs 222	Surfing
4749732	cs 145	Music
4749732	cs 245	Music
4749732	cs 222	Music
4749732	cs 145	Astronomy
4749732	cs 245	Astronomy
4749732	cs 222	Astronomy
8472374	cs 145	-
8472374	cs 336	-
...		

Problem

FDs?

Lots of redundancy

Root cause

Conditional independence

given SID: Classes & Hobbies are independent

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

<u>Conditional independence</u> (between film and snack given movie theatre)!

{A} ↠ **{B}** if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

there is a tuple t_3 s.t.
- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R\backslash B] = t_2[R\backslash B]$

Where R\B is "R minus B" i.e. the attributes of R not in B.

Example: there is a tuple t_3
$t_3[A] = t_1[A]$ = 'Rains 216'
$t_3[B] = t_1[B]$ = 'Star Trek: Wrath of Kahn'
$t_3[R\backslash B = C] = t_3[C]$ = 'Burrito'

Notes:
1. you can also check the other tuple
2. With more columns, [R\B] will be C + more columns, and $t_3[R\backslash B]$ should = $t_2[R \backslash B]$

Materials to know

A few questions

Know your SQL	Select-Project-Aggregate-Having, Nested queries, NULL values Set vs multi-sets
How to design ‘good’ tables?	Find bad FDs, Compute closures/superkeys, Do BCNF decompositions, Find MVDs
ACID transactions	Performance #s (disk seeks, memory access) WAL logs for a transaction? How to recover post crash? (Recap: WAL & performance, Why is a log COMMIT fast?)
	Is a schedule conflict serializable? How to build Conflict graph? What happens during 2PL execution? (Recap: With a 5 Transaction example)

(Recap will start with transactions and go backwards)

General form of Grouping and Aggregation

```
SELECT     S
FROM       R1,…,Rn
WHERE      C1
GROUP BY   a1,…,ak
HAVING     C2
```

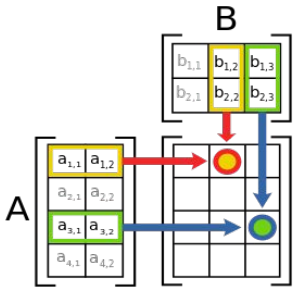
Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply HAVING condition C_2 to each group (may have aggregates)
4. Compute aggregates in SELECT, S, and return the result

General clarification: Sets vs. Multisets

- In theory, and in any more formal material, **by definition** all relations are ***sets of tuples***
- In SQL, relations (i.e. tables) are **multisets**, meaning you can have duplicate tuples
  - We need this because intermediate results in SQL don't eliminate duplicates
- If you get confused: just state your assumptions & we'll be forgiving!

Linear Algebra, Declaratively



- Matrix multiplication & other operations = just **joins**!
- The shift from **procedural** to **declarative** programming

```
C = [[0]*p for i in range(n)]

for i in range(n):
  for j in range(p):
    for k in range(m):
      C[i][j] += A[i][k] * B[k][j]
```

Proceed through a series of instructions

```
SELECT A.i, B.j, SUM(A.x * B.x)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Declare a desired output set

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

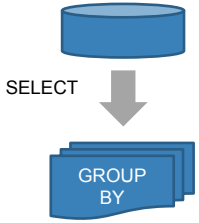
Think about **order*!**

**of the semantics, not the actual execution*

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
   FROM precipitation
   GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

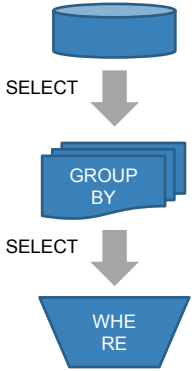


Get the max precipitation **by day**

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
          COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



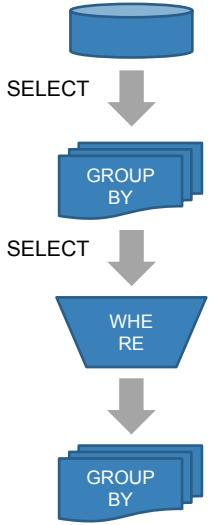
Get the max precipitation **by day**

Get the station, day pairs where / when this happened

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



Get the max precipitation **by day**

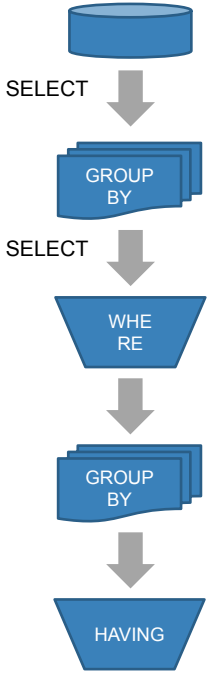
Get the station, day pairs where / when this happened

Group by stations

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
            COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip)
    FROM precipitation
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



Get the max precipitation **by day**

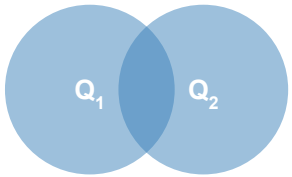
Get the station, day pairs where / when this happened

Group by stations

Having > 1 such day

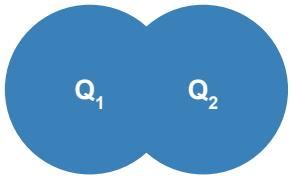
INTERSECT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```



UNION

```
SELECT  R.A
FROM   R, S
WHERE  R.A=S.A
UNION
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```



EXCEPT

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

Nested queries: Sub-queries Returning Relations

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
    SELECT pr.maker
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
           AND  p.buyer = 'Joe Blow')
```

"Cities where one can find companies that manufacture products bought by Joe Blow"

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

```
SELECT name
FROM   Product
WHERE  price > ALL(
  SELECT price
  FROM   Product
  WHERE  maker = 'G')
```

Find products that are more expensive than ***all products*** produced by "G"

ANY

```
SELECT name
FROM   Product
WHERE  price > ANY(
  SELECT price
  FROM   Product
  WHERE  maker = 'G')
```

Find products that are more expensive than ***any one product*** produced by "G"

EXISTS

```
SELECT name
FROM   Product p1
WHERE  EXISTS (
  SELECT *
  FROM   Product p2
  WHERE  p2.maker = 'G'
    AND  p1.price =
p2.price)
```

Find products where ***there exists some*** product with the same price produced by "G"

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

```
SELECT name
FROM   Product
WHERE  price > ALL(X)
```

Price must be > *all* entries in multiset X

ANY

```
SELECT name
FROM   Product
WHERE  price > ANY(X)
```

Price must be > *at least one* entry in multiset X

EXISTS

```
SELECT name
FROM   Product p1
WHERE  EXISTS (X)
```

X must be non-empty

**Note that p1 can be referenced in X (correlated query!)*

Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL
- *For boolean operations,* in SQL there are three values:

FALSE = 0
UNKNOWN = 0.5
TRUE = 1

  - If x= NULL then x="Joe" is UNKNOWN

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190)
```

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE / 1.0

Null Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25
  OR  age >= 25
```

Some Persons are not included !

```
SELECT *
FROM   Person
WHERE  age < 25
  OR  age >= 25
  OR  age IS NULL
```

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

Materials to know

A few questions

Know your SQL	Select-Project-Aggregate-Having, Nested queries, NULL values Set vs multi-sets
How to design ‘good’ tables?	Find bad FDs, Compute closures/superkeys, Do BCNF decompositions, Find MVDs
ACID transactions	Performance #s (disk seeks, memory access) WAL logs for a transaction? How to recover post crash? (Recap: WAL & performance, Why is a log COMMIT fast?)
	Is a schedule conflict serializable? How to build Conflict graph? What happens during 2PL execution? (Recap: With a 5 Transaction example)

(Recap will start with transactions and go backwards)