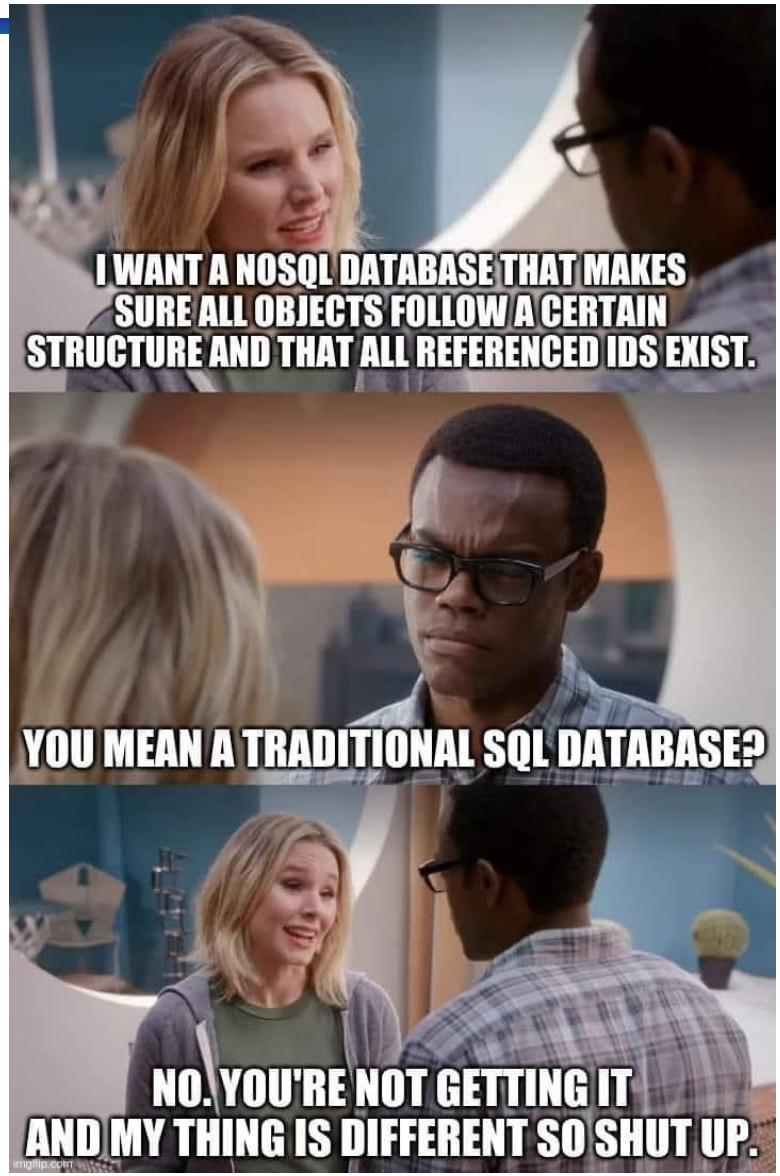
CPSC 304 – Administrative notes Nov. 29 and Dec. 3, 2024

- I may have to accept a call from my family doctor if she calls early. She should call after class is over, but if she calls early, I will need to step away for a few minutes. Sorry!
- Today: Apriori exercise & review (no new material) Clickers today not for points
- Friday is office hours
- Project: don't forget to go to your demo!
- Regular office hours end with the end of classes
  - Additional office hours will be posted
- Final exam: December 16 @12pm Osborne A
  - On paper, closed book, closed notes, closed neighbour



Structure of Review

- Overview
- Most requested topics. We'll see how many we get through.
  - Apriori exercise
  - 3NF Minimal covers
  - Division in Relational Algebra & Datalog
  - SQL: group by and having
  - HRU
  - Pipe Sort
  - Common mistakes

Reading list (from 3rd edition)

- Chapter 1: Intro
- Chapter 2: ER diagrams
- Chapter 3: The relational model
- Chapter 19: Normal forms: Sections 19.1 – 19.6
- Chapter 4: Relational Algebra all except 117-122, and No DRC
- Chapter 24: Sections 24.1 - 24.3 (Datalog)
- Chapter 5: SQL (No syntax for check constraints, assertions, or triggers)
- Chapter 25: Data Warehousing
- Chapter 26: Data mining

Overview of the final

- All material in the course is fair game, but expect it to be weighed more heavily on material from after the midterm
- On paper: closed book, closed notes, closed neighbour

What it looks like when we grade

- Question:

[6 marks] Minimal Keys

Consider the relation R (A, B, C, D, E, F, G, H) and the following FDs:

A –> D
AC –> EF
CGB –> F
EG –> B
E –> C
F –> D
GB –> CE

Find all the minimal keys for R (A, B, C, D, E, F, G, H). Write your final answer in the box, but show your work below. Please list the attributes in alphabetical order within a key and the keys in alphabetical order.

Answer:

- What we see:

:

How to study

- Do the practice exercises on Canvas
  - Do NOT just look at the answers and convince yourself that you can do the question.
- Look over the slides/videos/reveal concepts
- Play with the material

Reminder: Apriori algorithm formalized

1. Find the frequent itemsets of size 1; call this F_1
2. For k=1 until there are no more frequent itemsets
   1. Form candidate itemsets of size k+1: C_{k+1} is the set of itemsets of size k+1 where all subsets of C_{k+1} are frequent itemsets
   2. Count support of items in C_{k+1}
   3. F_{k+1} = itemsets in C_{k+1} that are frequent itemsets
3. Answer is the union of all F_k

Transaction	Items
T1	apple, dates, rice, corn
T2	corn, dates, tuna
T3	apple, corn, dates, tuna
T4	corn, tuna

Minimum support = 75%

Great! Your turn!

Use the Apriori algorithm to find frequent itemsets with a support threshold of 3/7 transactions. Write down all steps!

Algorithm reminder:

1. Find the frequent itemsets of size 1; call this F_1
2. For k=1 until there are no more
    1. Form candidate itemsets of size k+1: C_{k+1} is the set of itemsets of size k+1 where all subsets of C_{k+1} are frequent itemsets
    2. Count support of items in C_{k+1}
    3. F_{k+1} = itemsets in C_{k+1} that are frequent itemsets
3. Answer is the union of all F_k

Transaction	Items
T1	cake, jam, rolls, tea
T2	cake, jam, tea
T3	cake, jam
T4	jam, rolls, tea
T5	jam, rolls
T6	rolls, tea
T7	jam, tea

Apriori Algorithm Clicker question

Which of the following is in C_3?

A. {cake, jam, rolls}

B. {cake, jam, tea}

C. {jam, rolls, tea}

D. All are in C_3

E. None are in C_3

Transaction	Items
T1	cake, jam, rolls, tea
T2	cake, jam, tea
T3	cake, jam
T4	jam, rolls, tea
T5	jam, rolls
T6	rolls, tea
T7	jam, tea

Walking through the example

- Minimum support = 3/7
- Support for C_1:
  {cake} = 3
  {jam} = 6
  {rolls} = 4
  {tea} = 5
- F_1={{cake},{jam},{rolls},{tea}}

Transaction	Items
T1	cake, jam, rolls, tea
T2	cake, jam, tea
T3	cake, jam
T4	jam, rolls, tea
T5	jam, rolls
T6	rolls, tea
T7	jam, tea

Walking through the example

- Minimum support = 3/7
- Support for C_2:
  {cake, jam} = 3
  {cake, rolls} = 1
  {cake, tea} = 2
  {jam, rolls} = 3
  {jam, tea} = 4
  {rolls, tea} = 3
- F_2={{cake, jam}, {jam, rolls}, {jam, tea}, {rolls, tea}}
- Support for C_3: {jam, rolls, tea} = 2
- F_3 = {}

Transaction	Items
T1	cake, jam, rolls, tea
T2	cake, jam, tea
T3	cake, jam
T4	jam, rolls, tea
T5	jam, rolls
T6	rolls, tea
T7	jam, tea

Reminder: F_1={{cake},{jam},{rolls},{tea}}

3NF: minimal covers

To decompose to 3NF we rely on the Minimal Cover for a Set of FDs

Goal: Transform FDs to be as small as possible

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G (i.e., imply the same FDs)
  - Right hand side of each FD in G is a single attribute
  - If we delete an FD in G or delete attributes from an FD in G, the closure changes
- Intuitively, every FD in G is needed, and is "*as small as possible*" in order to get the same closure as F
- e.g., A→B, ABCD→E, EF→GH, ACDF→EG has the following minimal cover:
  - A→B, ACD→E, EF→G and EF→H

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ABCD→E, EF→G, EF→H, ACDF → EG

- Replace last rule with
  - ACDF → E
  - ACDF → G

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ABCD→E, EF→G, EF→H, ACDF → E, ACDF → G

- Can we take anything away from the LHS?
  - ACD^{+}= ABCDE, (crucially includes B) so remove B from the FD

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ACD→E, EF→G, EF→H, ACDF → E, ACDF → G

- Let's find $ACDF^+$ **without** considering the highlighted FDs
  - $ACDF^+$= ACDFEBGH, so I can remove the highlighted rules

- Final answer: A→B, ACD→E, EF→G, EF→H

Division in Relational Algebra & Datalog

Division

- Notation: ***r / s*** ***or*** ***r ÷ s***
- Useful for expressing queries that include a notion of "**for all**" or "**for every**", e.g., *Find movie stars who were in **all** movies*.
- Let *r* and *s* be relations on schemas R and S respectively where
  - r = (A_1, …, A_m, B_1, …, B_n)
  - s = (B_1, …, B_n)

  Then r / s is a relation on schema

  $r / s = (A_1, \ldots, A_m)$

  defined as

  $$r / s = \{ t \mid t \in \prod_{r-s}(r) \wedge \forall u \in s (tu \in r) \}$$

  - i.e., ***A/B* contains all *x* tuples (MovieStars) such that for *every* *y* tuple (movies) in *B*, there is an *x,y* tuple in *A*.**

Examples of Division A/B

A

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Example: building up division subtract off disqualified answers

$A=R$

X	Y
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

$B2 = S$

Y
P2
P4

$\pi_X(R)$

X
S1
S2
S3
S4

All possible values given R

$\pi_X(R) \times S$

X	Y
S1	P2
S1	P4
S2	P2
S2	P4
S3	P2
S3	P4
S4	P2
S4	P4

Values needed for $\pi_X(R)$

$\pi_X(R) \times S - R$

X	Y
S2	P4
S3	P4

These values aren't in R

$\pi_X(R) - \pi_X(\pi_X(R) \times S - R) = A/B2$

A/B2 =

X
S1
S4

Answers not disqualified

Find the name of actors who have been in ***all*** movies

Be careful in choosing the input relations!

InAll← $\pi_{\text{StarID, MovieID}}$StarsIn/ π_{MovieID}(Movie)

π_{Name}(InAll ⋈ MovieStar)

Find the names of actors who have been in all movies after 1950

$$\text{LateMovieIds} \leftarrow \pi_{\text{MovieID}}(\sigma_{\text{year} > 1950}(\text{Movie}))$$

$$\text{InAll} \leftarrow (\pi_{\text{StarID, MovieID}}(\text{StarsIn}) / \text{LateMovieIds})$$

$$\pi_{\text{Name}}(\text{InAll} \bowtie \text{MovieStar})$$

Division in Datalog

Assume schema

Cust(cid,cname,rating,salary)

Order(iid,cid,day,qty)

Query: find items (iid) that are ordered by every customer

The query in Relational Algebra

$\pi_{iid,cid}(order)/\pi_{cid}(cust)$

What does this look like in Datalog?

```
Witness(I,C):-Order(I,C,_,_)
Bad(I)       :- Cust(_,C,_,_), Order(I,_,_,_),  NOT Witness(I,C)
Good(I)      :- Order(I,C, _, _), NOT Bad(I)
```

Technically, you don't need witness, can just use another "order". It's just cleaner to do it this

- Witness finds all items that have been ordered
- Bad finds all items that have not been ordered by some customer
- Good: finds all items for that have not been not ordered by some customer (i.e., all items that have been ordered by all customers)

More generally, in the simplest case

Relations: A(A1, B1, …), B(B1, …)

Witness(a1,b1):-A(a1,b1,…)

Bad(a1) :- A(a1,…), B(b1,…), NOT Witness(a1,b1)

Good(a1) :- A(a1,…), NOT Bad(a1)

Returning to our previous example

A(Sno, Pno)

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B(Pno)

Pno
P2
P4

Witness(a1,b1):-
A(a1, b1)

a1	b1
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

A/B =

a1
S1
S4

Bad-no-π(a1,b1):-
A(a1,…), B(b1,…)
NOT Witness(a1,b1)

a1	b1
S2	P4
S3	P4

Bad(a1):-
A(a1,…), B(b1,…)
NOT Witness(a1,b1)

Bad

a1
S2
S3

Good(a1):-A(a1,…),
NOT

SQL: Group By and Having

GROUP BY and HAVING

- Divide tuples into groups and apply aggregate operations to each group.
- Example: *Find the age of the youngest student for each major.*

For *i* = 'Computer Science',
'Civil Engineering'…

```
SELECT  MIN (age)
FROM    Student
WHERE   major = i
```

- Problem:

  We don't know how many majors exist, not to mention this is not good practice

Grouping Examples

Find the age of the youngest student who is at least 19, for each major

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
…	…	…

No Animal Science

Major	Age
Computer Science	19
Accounting	19
Civil Engineering	21
…	…

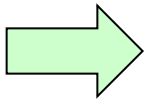
Grouping Examples with Having

Find the age of the youngest student who is at least 19, for each major with at least 2 such students

```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING    COUNT(*) > 1
```

Snum	Major	Age
115987938	Computer Science	20
112348546	Computer Science	19
280158572	Animal Science	18
351565322	Accounting	19
556784565	Civil Engineering	21
…	…	…

Major	Age
Computer Science	20
Computer Science	19
Accounting	19
Civil Engineering	21
…	…



Major	
Computer Science	19

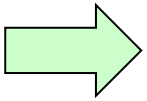
And there are rules

Find the age of the youngest student who is at least 19, for each major with at least 2 such students

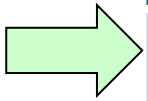
```
SELECT    major, MIN(age)
FROM      Student
WHERE     age >= 19
GROUP BY  major
HAVING    COUNT(*) > 1
```

- Would it make sense if I select age instead of MIN(age)?
- *Would it make sense if I select snum to be returned?*
- *Would it make sense if I select major to be returned?*

Major	Age
Computer Science	20
Computer Science	19
Accounting	19
Civil Engineering	21
…	…



Major	Age
Computer Science	20
Computer Science	19



Major	Age
Computer Science	19

GROUP BY and HAVING (cont)

- Example1: *For each class, find the age of the youngest student who has enrolled in this class:*

```
SELECT      cname,  MIN(age)
FROM        Student S, Enrolled E
WHERE       S.snum= E.snum
GROUP BY    cname
```

- Example2: *For each course with more than 1 enrollment, find the age of the youngest student who has taken this class:*

```
SELECT      cname,  MIN(age)
FROM        Student S, Enrolled E
WHERE       S.snum = E.snum
GROUP BY    cname
HAVING      COUNT(*) > 1        ← per group qualification!
```

Grouping Examples (cont')

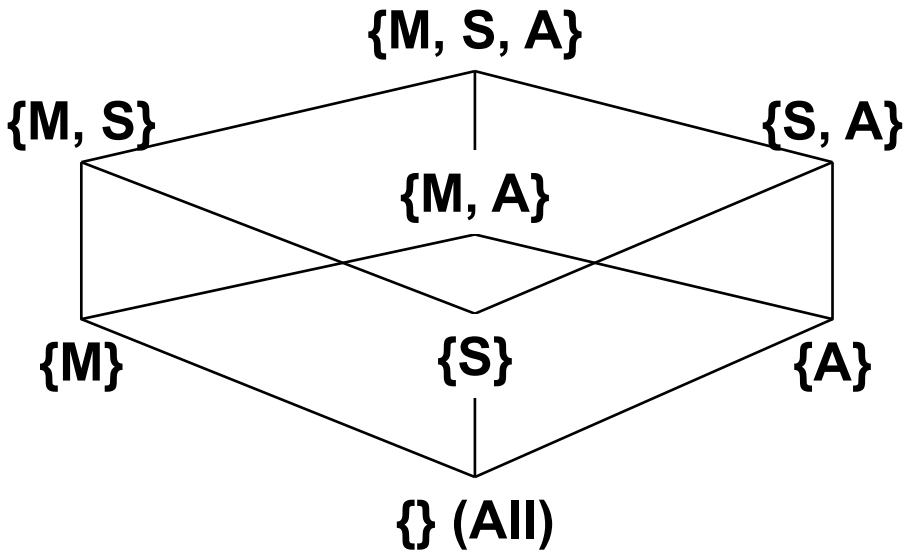
Find the age of the youngest student with age > 18, for each major with at least 2 students(of age > 18)

```
SELECT  S.major, MIN(S.age)
FROM    Student S
WHERE   S.Age > 18
GROUP BY  S.major
HAVING  COUNT(*) > 1
```

Warehousing: Choosing which views to materialize (HRU)

Let's revisit the student table: Student(snum,sname,major,standing,age)

- We can aggregate the student table based on major, standing, or age



Group by all 3 – any ordering gives you same # of tuples

```
SELECT major, standing, age, count(*)
FROM         student
GROUP BY major, standing, age
ORDER BY major, standing, age
```

MAJOR	ST	AGE	COUNT(*)
Accounting	JR	19	1
Animal Science	FR	18	1
Architecture	SR	22	1
Civil Engineering	SR	21	1
Computer Engineering	FR	18	1
Computer Engineering	SR	19	1
Computer Science	JR	18	1
Computer Science	JR	20	1
Computer Science	SO	17	1
Computer Science	SO	19	1
Economics	JR	20	1
Education	SR	21	1
Electrical Engineering	FR	17	2
English	SR	21	1
Finance	FR	18	2
History	SR	20	1
Kinesiology	SO	19	1
Law	JR	20	1
Mechanical Engineering	SO	19	1
Psychology	JR	20	1
Psychology	SO	18	1
Veterinary Medicine	SR	21	1

22 rows selected.

```
SELECT age, standing, major, count(*)
FROM student
GROUP BY age, standing, major
ORDER BY age, standing, major
```

AGE	ST	MAJOR	COUNT(*)
17	FR	Electrical Engineering	2
17	SO	Computer Science	1
18	FR	Animal Science	1
18	FR	Computer Engineering	1
18	FR	Finance	2
18	JR	Computer Science	1
18	SO	Psychology	1
19	JR	Accounting	1
19	SO	Computer Science	1
19	SO	Kinesiology	1
19	SO	Mechanical Engineering	1
19	SR	Computer Engineering	1
20	JR	Computer Science	1
20	JR	Economics	1
20	JR	Law	1
20	JR	Psychology	1
20	SR	History	1
21	SR	Civil Engineering	1
21	SR	Education	1
21	SR	English	1
21	SR	Veterinary Medicine	1
22	SR	Architecture	1

22 rows selected.

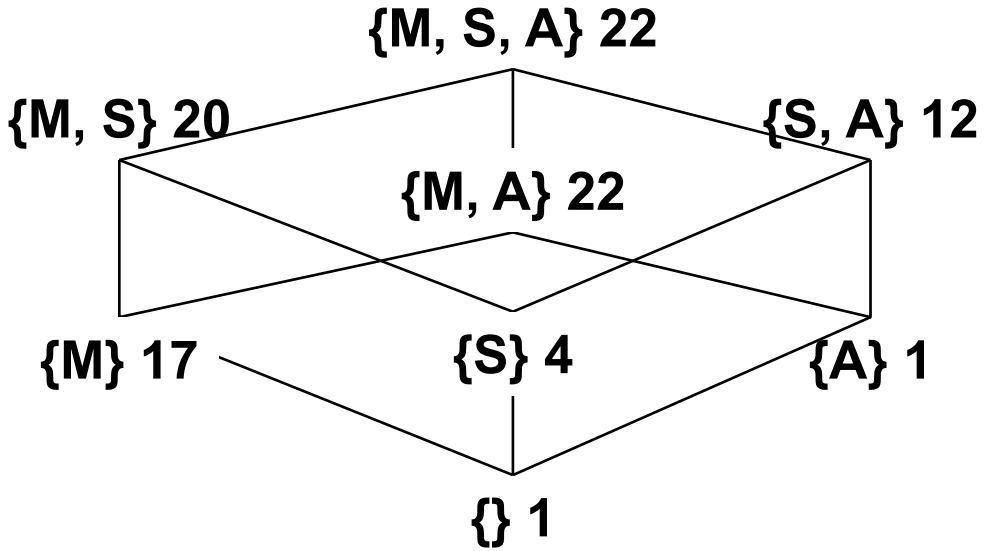
Group by 2 option 1: Major, Standing

```
SELECT major, standing, count(*)
FROM student
GROUP BY major, standing
ORDER BY major, standing

MAJOR                     ST   COUNT(*)
------------------------- -- ----------
Accounting                JR          1
Animal Science            FR          1
Architecture              SR          1
Civil Engineering         SR          1
Computer Engineering      FR          1
Computer Engineering      SR          1
Computer Science          JR          2
Computer Science          SO          2
Economics                 JR          1
Education                 SR          1
Electrical Engineering    FR          2
English                   SR          1
Finance                   FR          2
History                   SR          1
Kinesiology               SO          1
Law                       JR          1
Mechanical Engineering    SO          1
Psychology                JR          1
Psychology                SO          1
Veterinary Medicine       SR          1

20 rows selected.
```

In computing costs to query, we consider the # of tuples that you have to look at



The # is the # of tuples

Assuming all of the views were materialized, executing the query

```
SELECT standing, count(*)
FROM student
GROUP BY standing
```

would cost 4 because that's the cheapest way to access the necessary tuples

We can use {Major, Standing} to compute {Standing} for a cost of 20

```
SELECT major, standing, count(*)
FROM student
GROUP BY major, standing
ORDER BY major, standing
```

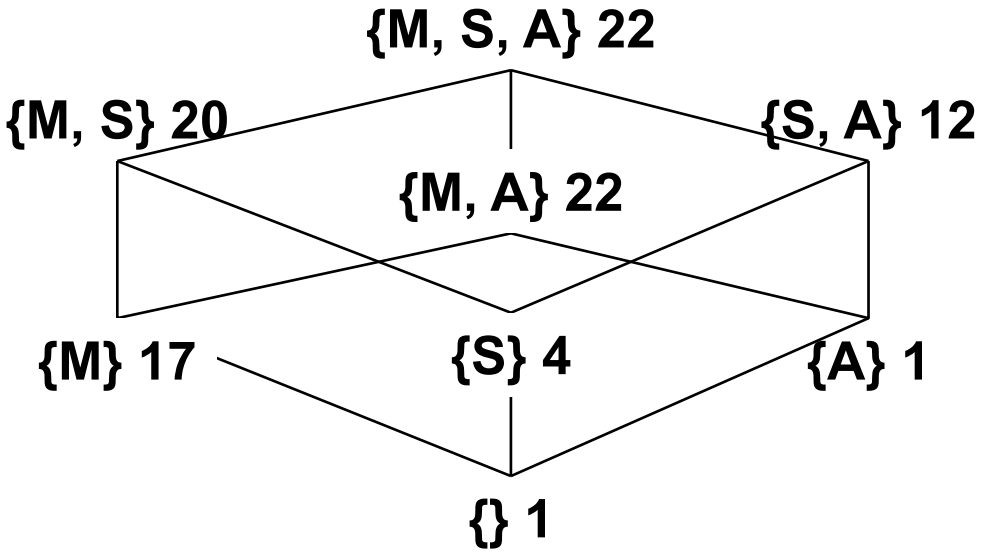
```
SELECT standing, count(*)
FROM student
GROUP BY standing
ORDER BY standing
```

```
MAJOR                     ST   COUNT(*)
------------------------- -- ----------
Accounting                JR          1
Animal Science            FR          1
Architecture              SR          1
Civil Engineering         SR          1
Computer Engineering      FR          1
Computer Engineering      SR          1
Computer Science          JR          2
Computer Science          SO          2
Economics                 JR          1
Education                 SR          1
Electrical Engineering    FR          2
English                   SR          1
Finance                   FR          2
History                   SR          1
Kinesiology               SO          1
Law                       JR          1
Mechanical Engineering    SO          1
Psychology                JR          1
Psychology                SO          1
Veterinary Medicine       SR          1

20 rows selected.
```

```
ST   COUNT(*)
-- ----------
SR          7
SO          5
FR          6
JR          6
```

Clicker question: which views can be used for which query? Part 2

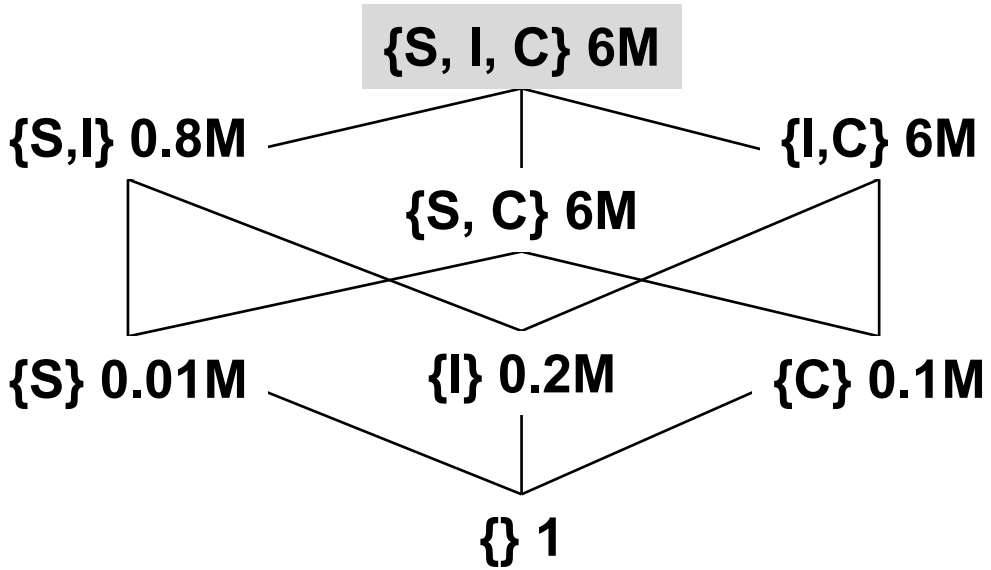


Assume {M,S,A} is only view materialized. If we materialize {S,A}, queries over which lattice parts could be answered using {S,A}?

A. {S,A}
B. {{M,S,A}{S,A}
C. {{S,A},{S},{A},{}}
D. None of the above

The correct answer is C; {S,A} can be used to answer anything that uses an aggregation on {S,A}

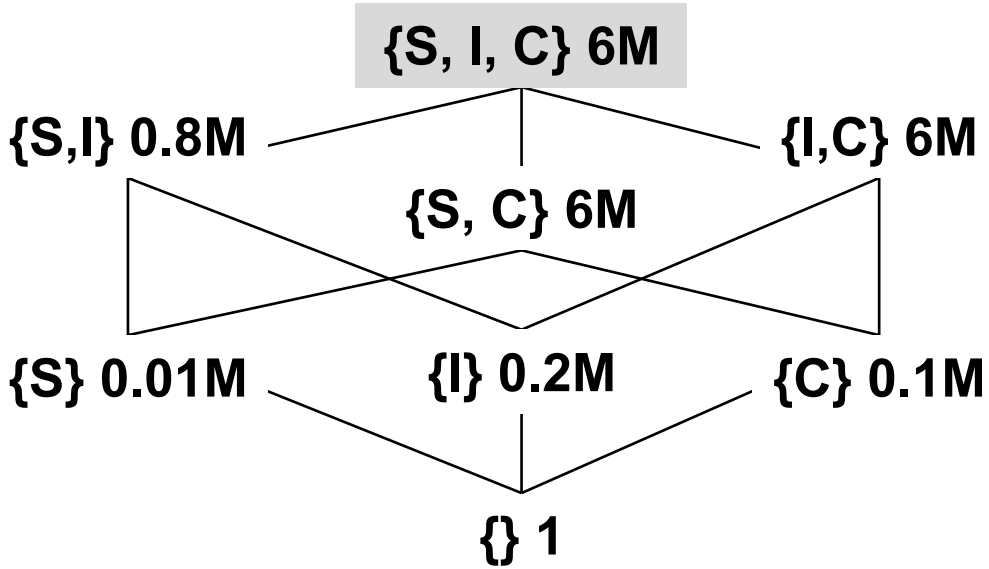
The HRU algorithm for materializing views



- The question is: which views can we materialize to answer queries the cheapest?
- Initially, only the top-most view is materialized

HRU [Harinarayan, Rajaraman, and Ullman, 1996]—SIGMOD Best Paper award—is a greedy algorithm that does not guarantee an optimal solution, though it usually produces a good solution. This solution is a good trade-off in terms of the space used and the average time to answer an OLAP query.

Benefit of Materializing a View



Intuitively, for each view under consideration, determine (1) if it can be used to answer a query and (2) if so, how much does it save?

Formally:

Define the benefit (savings) of view v relative to S as **B(v,S)**.

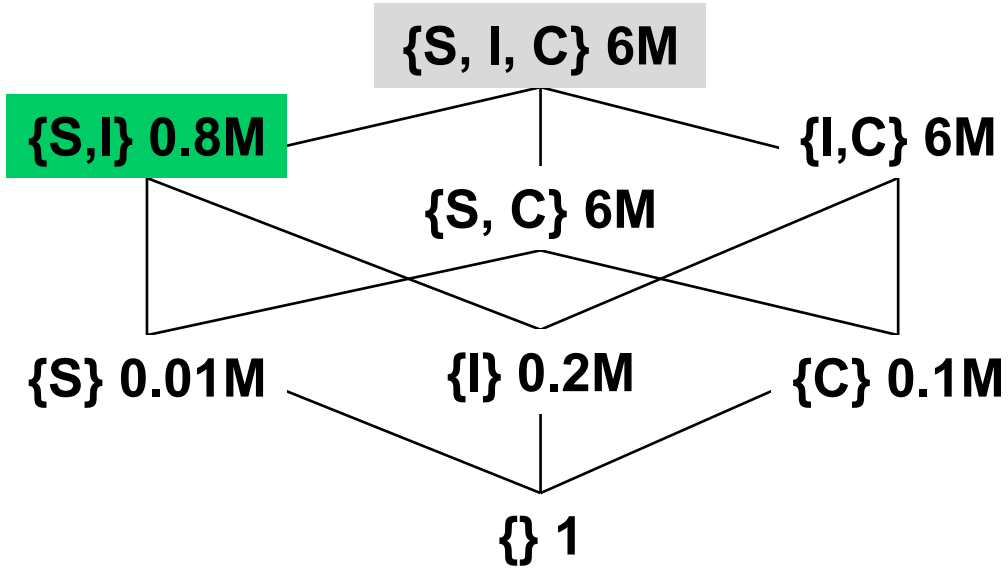
B(v, S) = 0
For each w ≦ v
    u = view of least cost in S such that w ≦ u
    if C(v) < C(u) then B_w = C(u) – C(v)
    else B_w = 0
    B(v,S) = B(v,S) + B_w
end

S = set of views selected for materialization

b ≦ a means b is a descendant of a (including itself) – b can be answered using only a (e.g., {S} ≦ {S,I}

C(v) = cost of view v, which we're approximating by its size

Benefit of Materializing a View



- The number associated with each node represents the number of rows in that view (in millions)
- Initial state has only the top most view materialized

Define the benefit (savings) of view v relative to S as **B(v,S)**.

B(v, S) = 0
For each w ≦ v
    u = view of least cost in S such that w ≦ u
    if C(v) < C(u) then $B_w = C(u) – C(v)$
    else $B_w = 0$
    $B(v,S) = B(v,S) + B_w$
end

Example

S = {{S, I, C}}, **v = {S, T}**

$B_{\{S, I\}}$ **= 5.2 M**

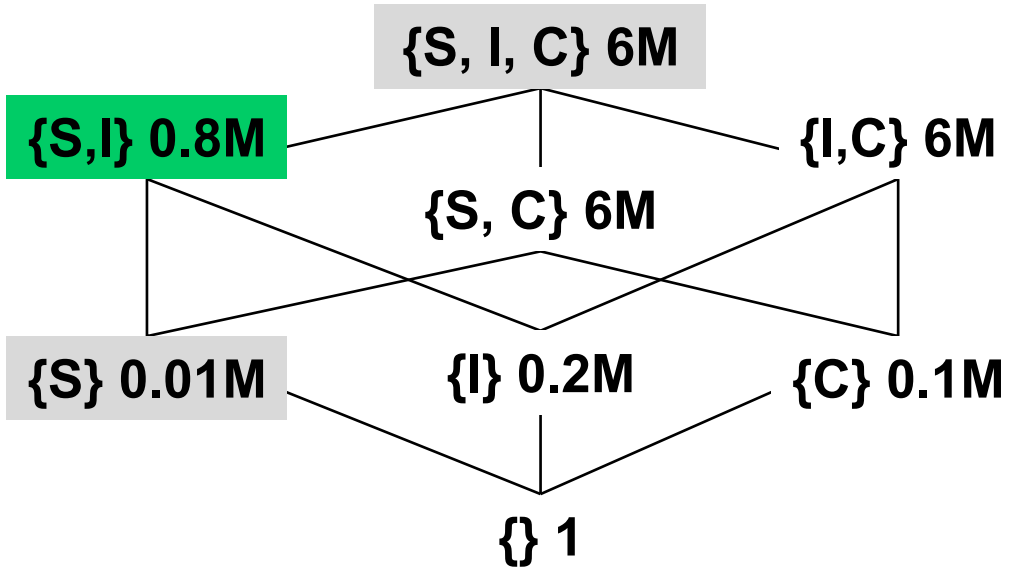
$B_{\{S\}}$ **= 5.2 M**

$B_{\{I\}}$ **= 5.2 M**

$B_{\{\}}$ **= 5.2 M**

B(v,S) = 5.2M *4

Benefit of Materializing a View: Clicker Question



What is the saving of {S,I} relative to {{S,I,C},{S}}?

A. 5.2M
B. 5.2M x 2
C. 5.2M x 4 M
D. .79M x 2
E. None of the above

Define the benefit (savings) of view v relative to S as **B(v,S)**.

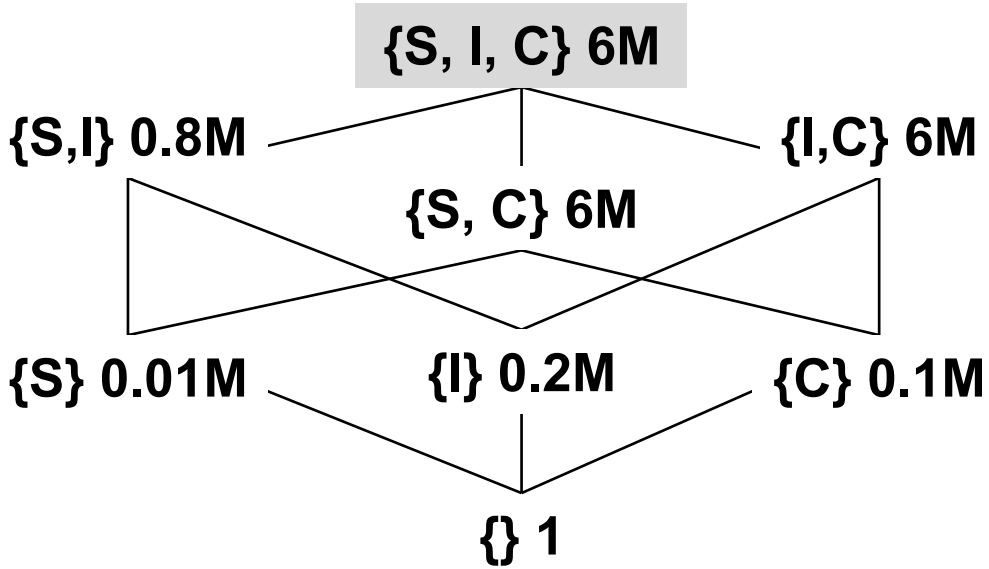
B(v, S) = 0
For each w ≦ v
    u = view of least cost in S such that w ≦ u
    if C(v) < C(u) then $B_w = C(u) – C(v)$
    else $B_w = 0$
    $B(v,S) = B(v,S) + B_w$
end

Calculation:

S = {{S, I, C}, {S}}, **v = {S,I}**

$B_{\{S,I\}}$ **= 5.2 M**
$B_{\{S\}}$ **= 0**
$B_{\{I\}}$ **= 5.2 M**
$B_{\{\}}$ **= 0**
B(v,S) = 5.2M *2

Finding the Best *k* Views to Materialize



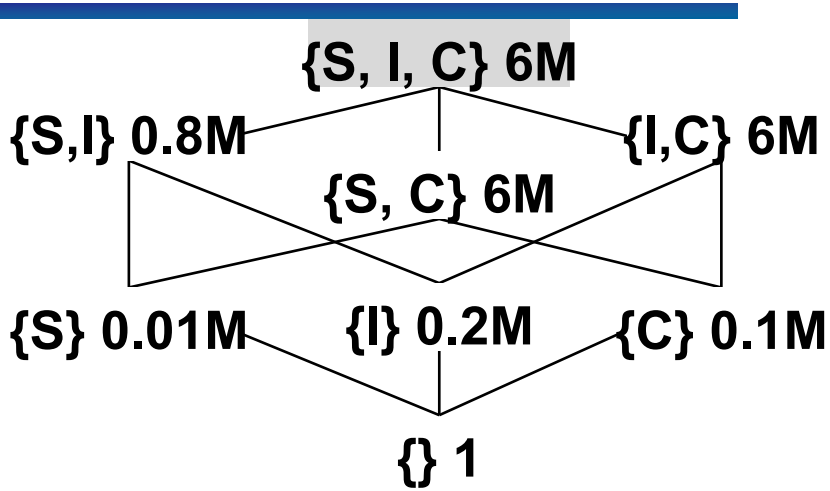
- The number associated with each node represents the number of rows in that view (often in millions)
- Initial state has only the top most view materialized

A greedy algorithm for finding the best *k* views to materialize

```
S = {top view}
for i=1 to k do begin
    select v ⊄ S such that B(v,S) is maximized
    S = S union {v}
end
```

HRU Algorithm Example. Pick the best 2 views to materialize: Round 1

{S,I} offers biggest benefit: materialize it.



View	1st choice
{S, I}	(6-0.8)M *4 = 20.8M
{S, C}	(6-6) *4 = 0
{I, C}	(6-6) *4 = 0
{S}	(6-0.01) M*2 = 11.98M
{I}	(6-0.2) M*2 = 11.6M
{C}	(6-0.1) M*2 = 11.8M
{}	6M – 1

Explanation

Can impact {S,I}, {S}, {I}, {}. Benefit is over {S,I,C}

Can impact {S,C}, {S},{C},{}. Benefit from {S,I,C} is zero

Can impact {I,C}, {I},{C},{}. Benefit from {S,I,C} is zero

Can impact {S}, {}. Benefit is over {S,I,C}

Can impact {I}, {}. Benefit is over {S,I,C}

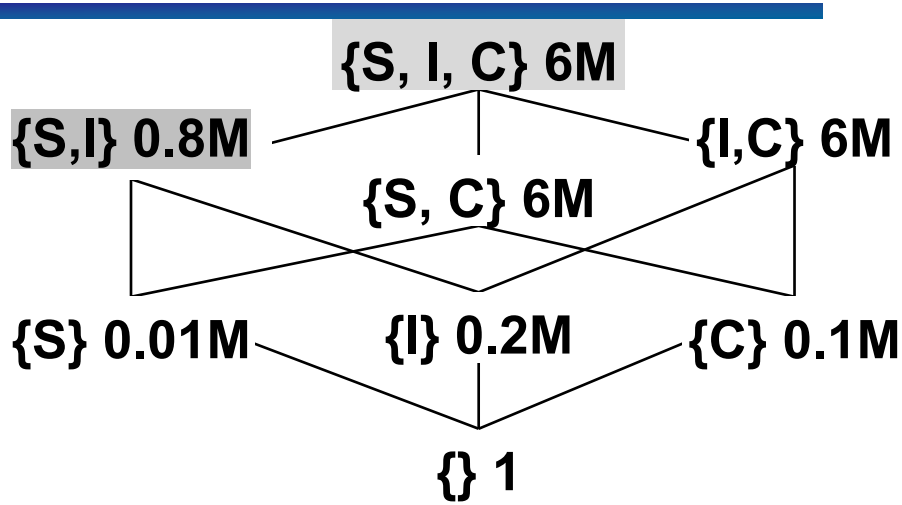
Can impact {C}, {}. Benefit is over {S,I,C}

Can impact {}. Benefit is over {S,I,C}

HRU Algorithm Example. Pick the best 2 views to materialize: Round 2

{S,I} is already materialized.

{C} offers biggest benefit: materialize it



View	2nd choice
{S, I}	
{S, C}	(6-6) *2 = 0
{I, C}	(6-6) *2 = 0
{S}	(0.8-0.01)M*2 = 1.58M
{I}	(0.8-0.2)M*2 = 1.2M
{C}	**(6-0.1)M + (0.8-0.1)M = 6.6M**
{}	0.8M - 1

Explanation

Already materialized. Not an option
Can impact {S,C}, {S},{C},{}. Benefit of 0 from {S,I,C} for {S,C},{C}. {S,I} is cheaper for {S},{}

Same reasoning as {S,C}

Can impact {S}, {}. Benefit is over {S,I}

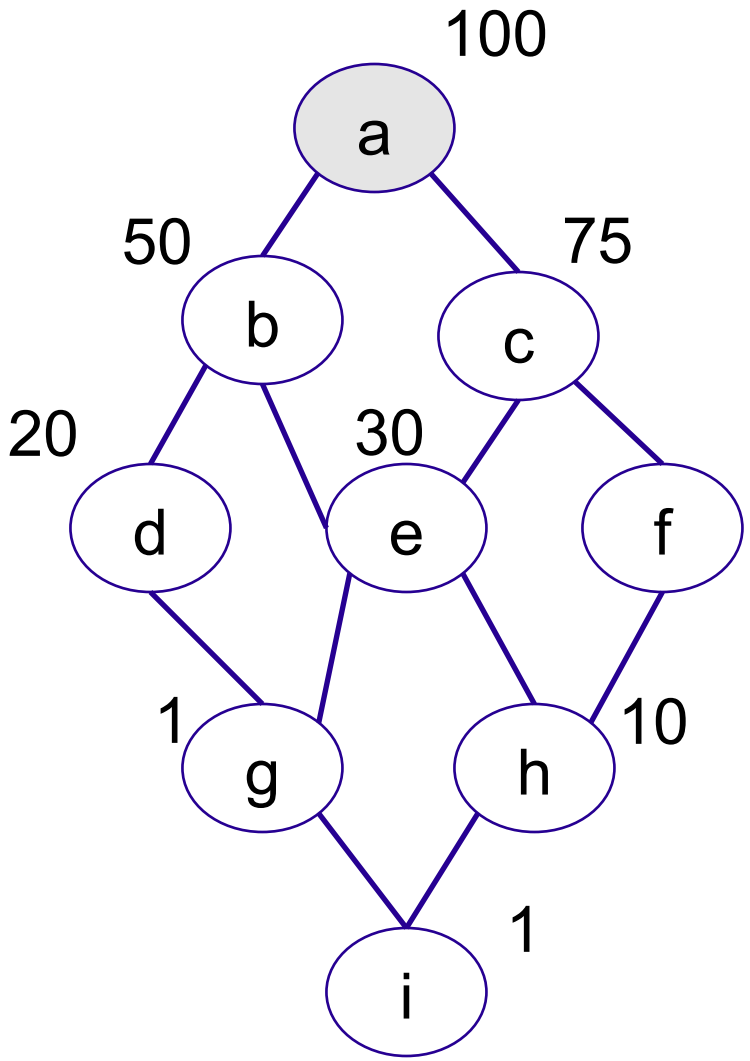
Can impact {I}, {}. Benefit is over {S,I}

Can impact {C}, {}. Benefit over {S,I,C} for {C}, {S,I} for {}

Can impact {}. Benefit is over {S,I}

A note on workload

- Thus far, we've been assuming that the workload is evenly distributed.
- If it's not, then multiplying by the workload expected at each spot in the lattice will give you a more precise answer
- For example, assume that queries at g and h each make up 20% of the workload, and the remainder of the nodes make 10% each

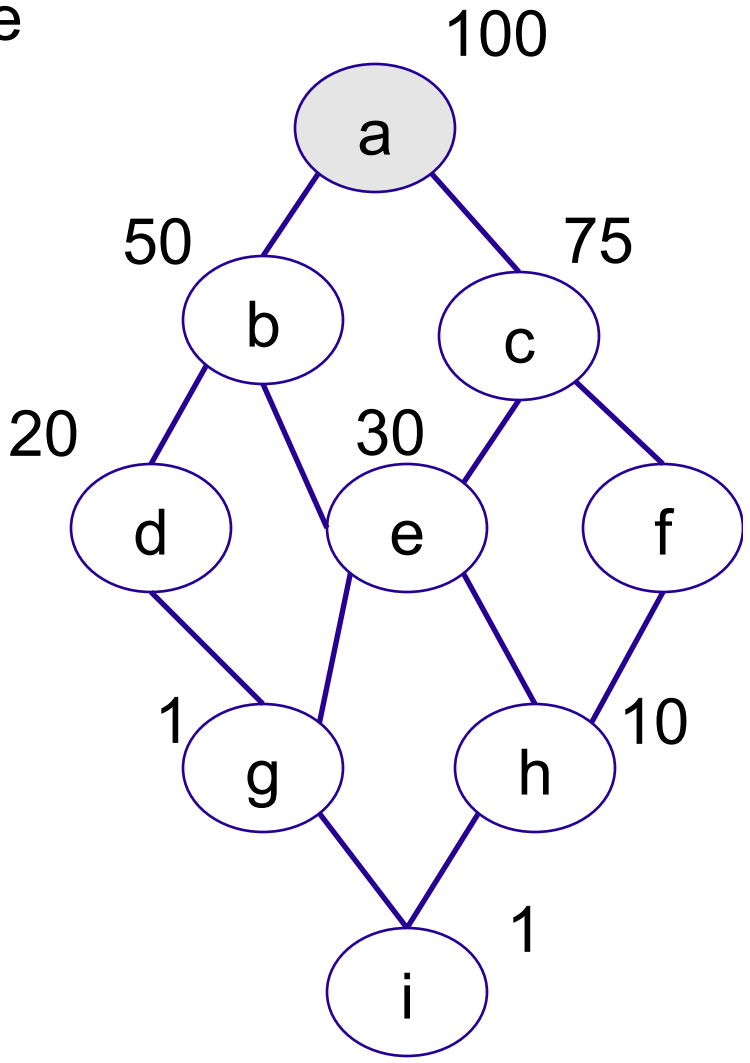


Taking workload into account

- Assume queries at g and h each make up 20% of the workload, and the remainder of the nodes make 10% each

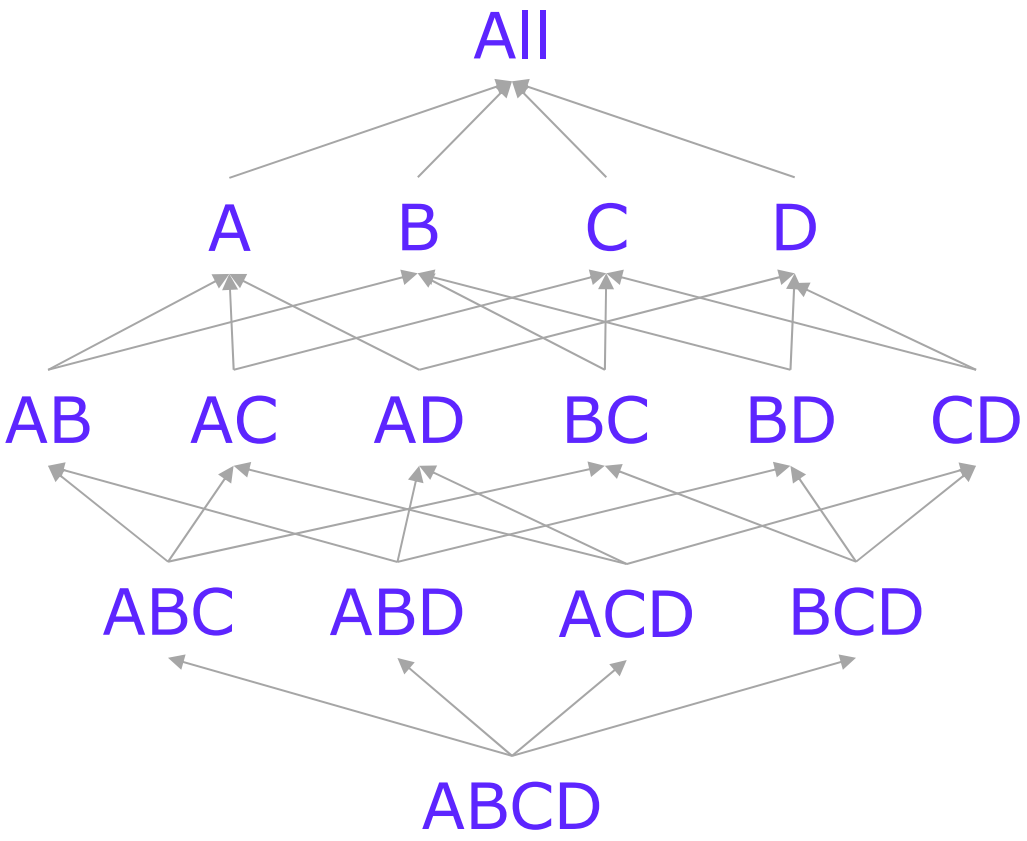
Original: With workload:

View	1st choice	View	1st choice
b	50*6=300	b	.2*2*50+.1*4*50=40
c	25*6=150	c	.2*2*25+.1*4*25=20
d	80*3=240	d	.2*1*80+.1*2*80=32
e	70*4=280	e	.2*2*70+.1*2*70=42
f	60*3=180	f	.2*1*60+.1*2*60=24
g	99*2=198	g	.2*1*99+.1*1*99=29.7
h	90*2=180	h	.2*1*90+.1*1*90=27
i	99	i	.1*99=9.9



Order to materialize *all* views in a warehouse: Pipe Sort

The PipeSort Algorithm [Agarwal et al., VLDB 1996]



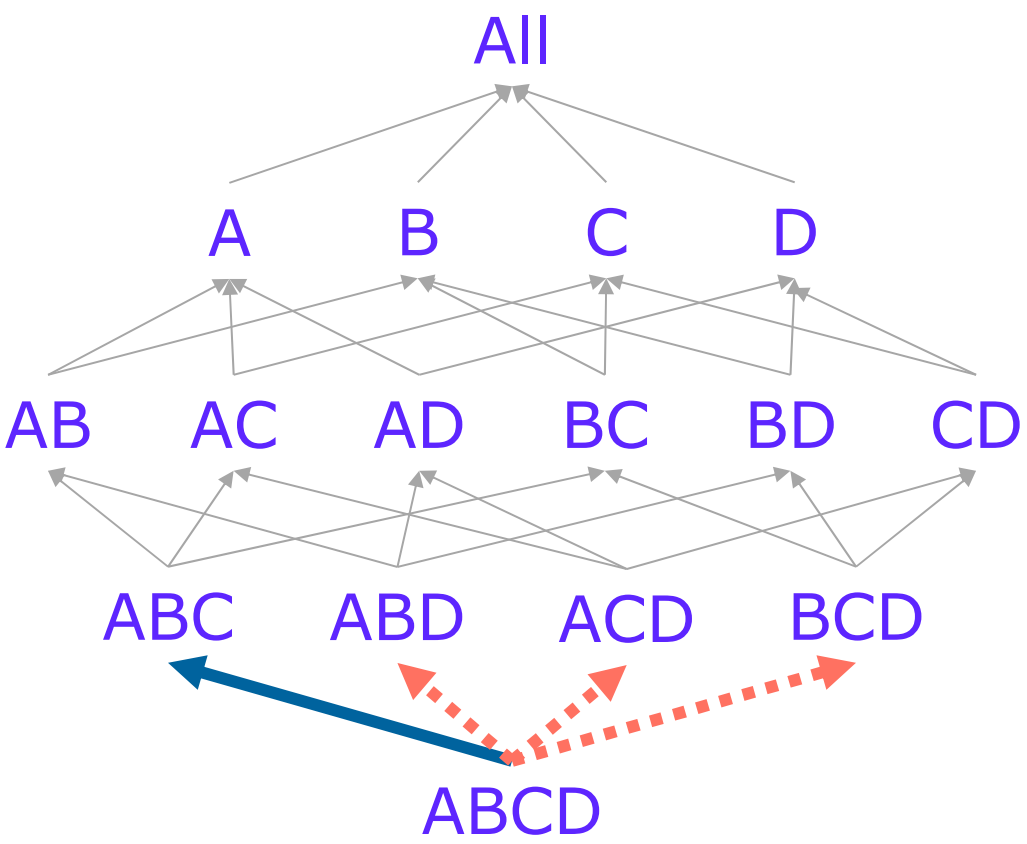
Output:

- Exactly which views to materialize in which order

Key ideas:

- Pipelining the computation is cheaper
- Ordering matters to the pipelining but NOT the cube.
- Compute each view once
- Greedy works well

Greedy works really well

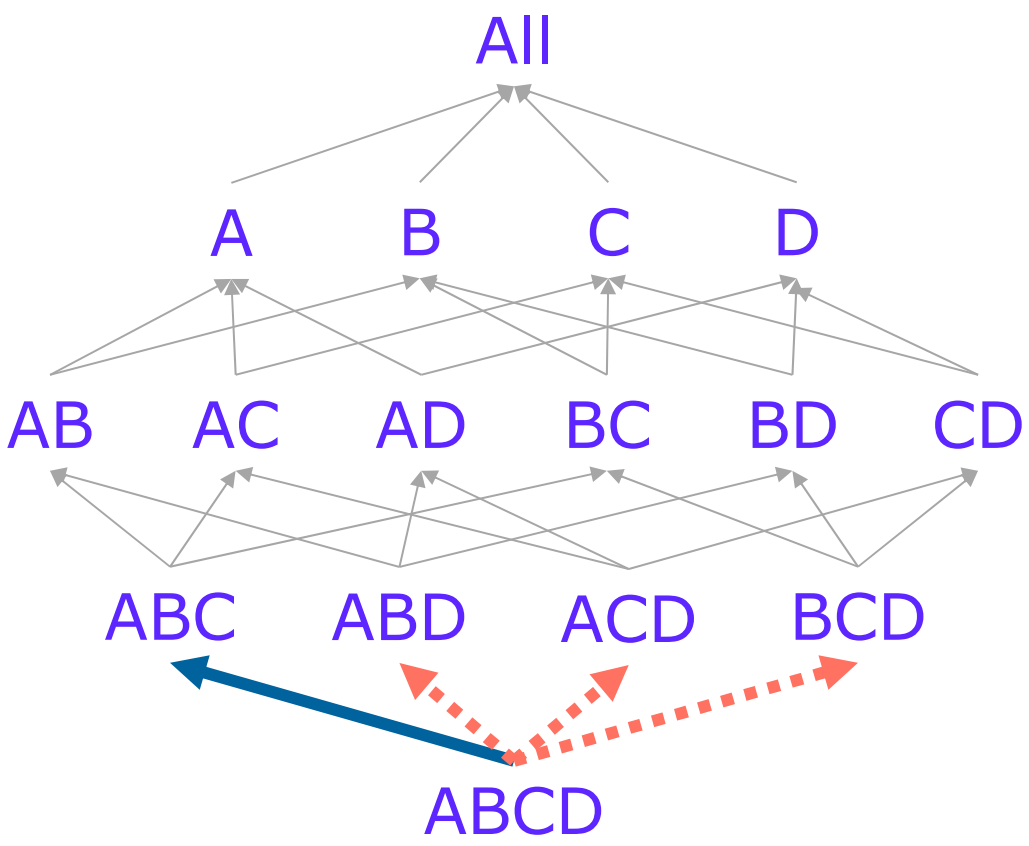


A Edge (sorted)

S Edge (not sorted)

- At each level, look at the costs to compute the next level
- Use A edges if already sorted
- Use S edges if not already sorted
- Each view can be the origin of **one** A edge
- Each view can be the origin of as many S edges as necessary
- Greedy: minimize per-level cost

Valid solutions & Costs

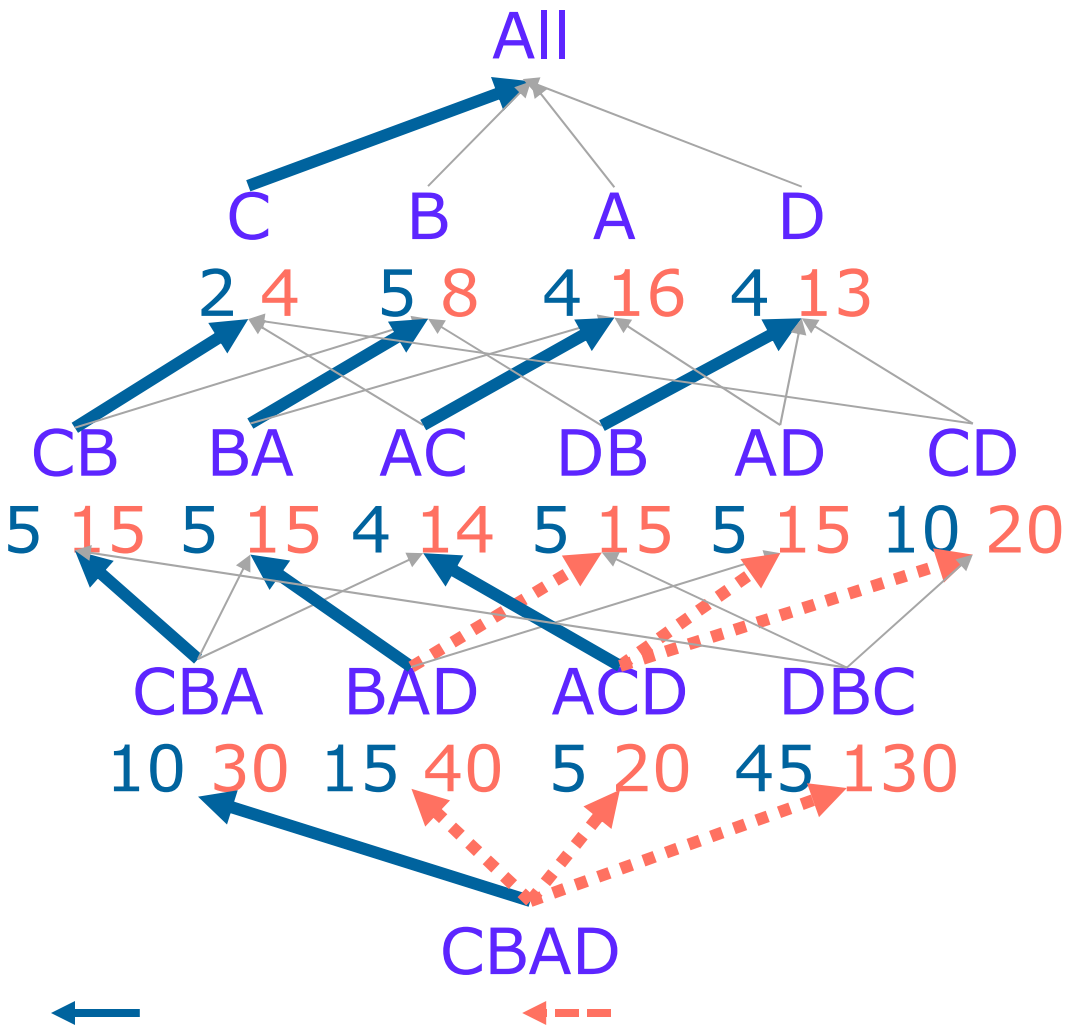


A Edge (sorted)

S Edge (not sorted)

- Valid solution:
  - Each view in the lower level has to be the origin of at most one A edge (multiple S edges are okay)
  - Each view at the upper level has to be the destination of at least one edge (A or S)
- Cost:
  - Sum of the cost of all A and S edges (if more than one S edge leaves a view, need to include the cost twice, because may need to resort)

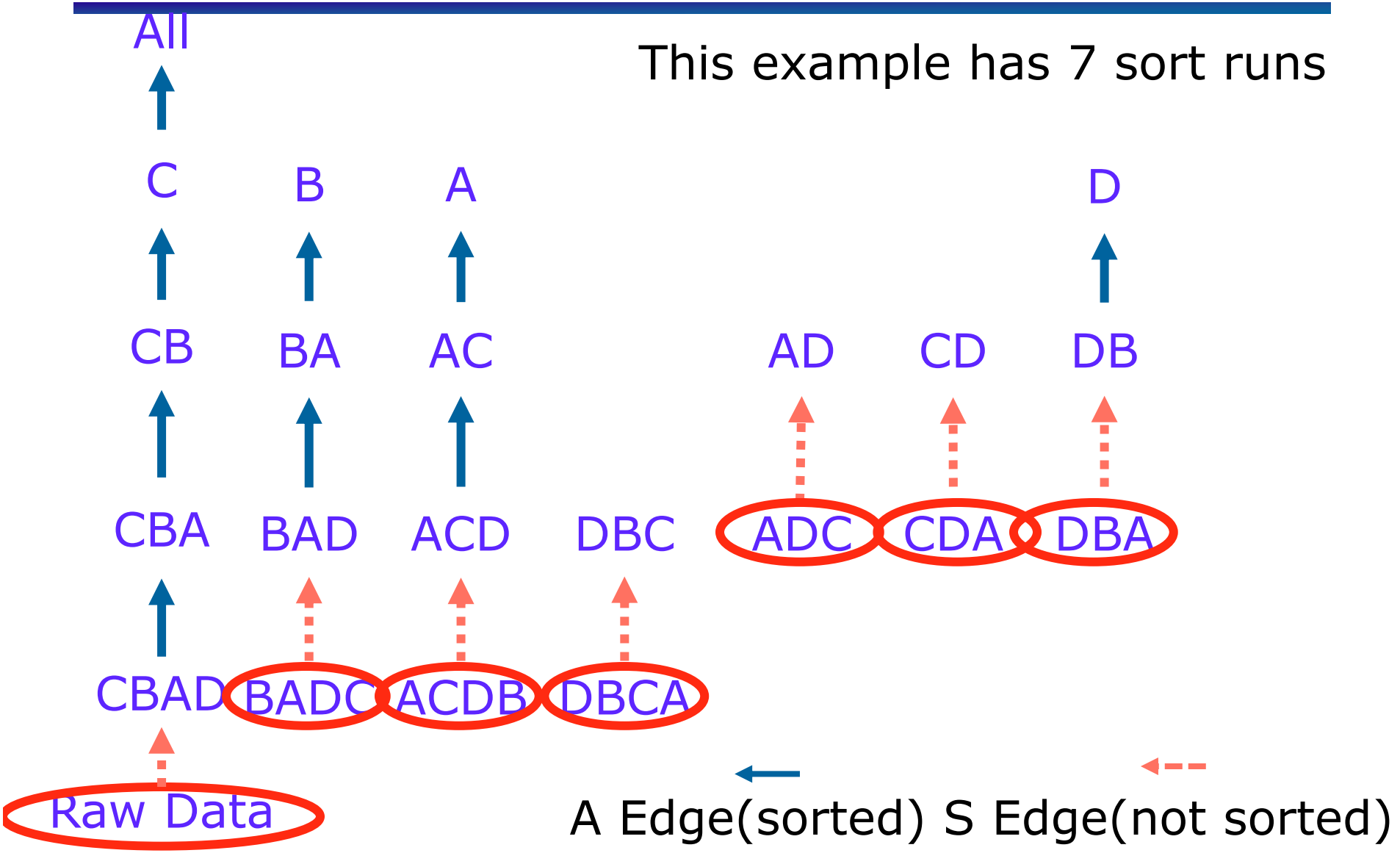
Can greedily compute entire cube materialization



A Edge(sorted) S Edge(not sorted)

Then execute the pipelines

Circles indicate sorts. Each column is a *sort run*



Common mistakes

- Read the problem carefully
- Make sure that you check the name of all attributes when you're doing joins and division in relational algebra
- Make sure that you understand how to do "larger than all" queries in RA and SQL