

L03: SQL

Why I don't post slides *before* lecture

From the Preamble of one of the best physics books there is: „How to read this book“

The best way to use this book is NOT to simply read it or study it, but to read a question and STOP. Even close the book. Even put it away and THINK about the question. Only after you have formed a reasoned opinion should you read the solution. Why torture yourself thinking? Why jog? Why do push-ups?

If you are given a hammer with which to drive nails at the age of three you may think to yourself, “OK, nice.” But if you are given a hard rock with which to drive nails at the age of three, and at the age of four you are given a hammer, you think to yourself, “What a marvelous invention!” You see, you can't really appreciate the solution until you first appreciate the problem.

...

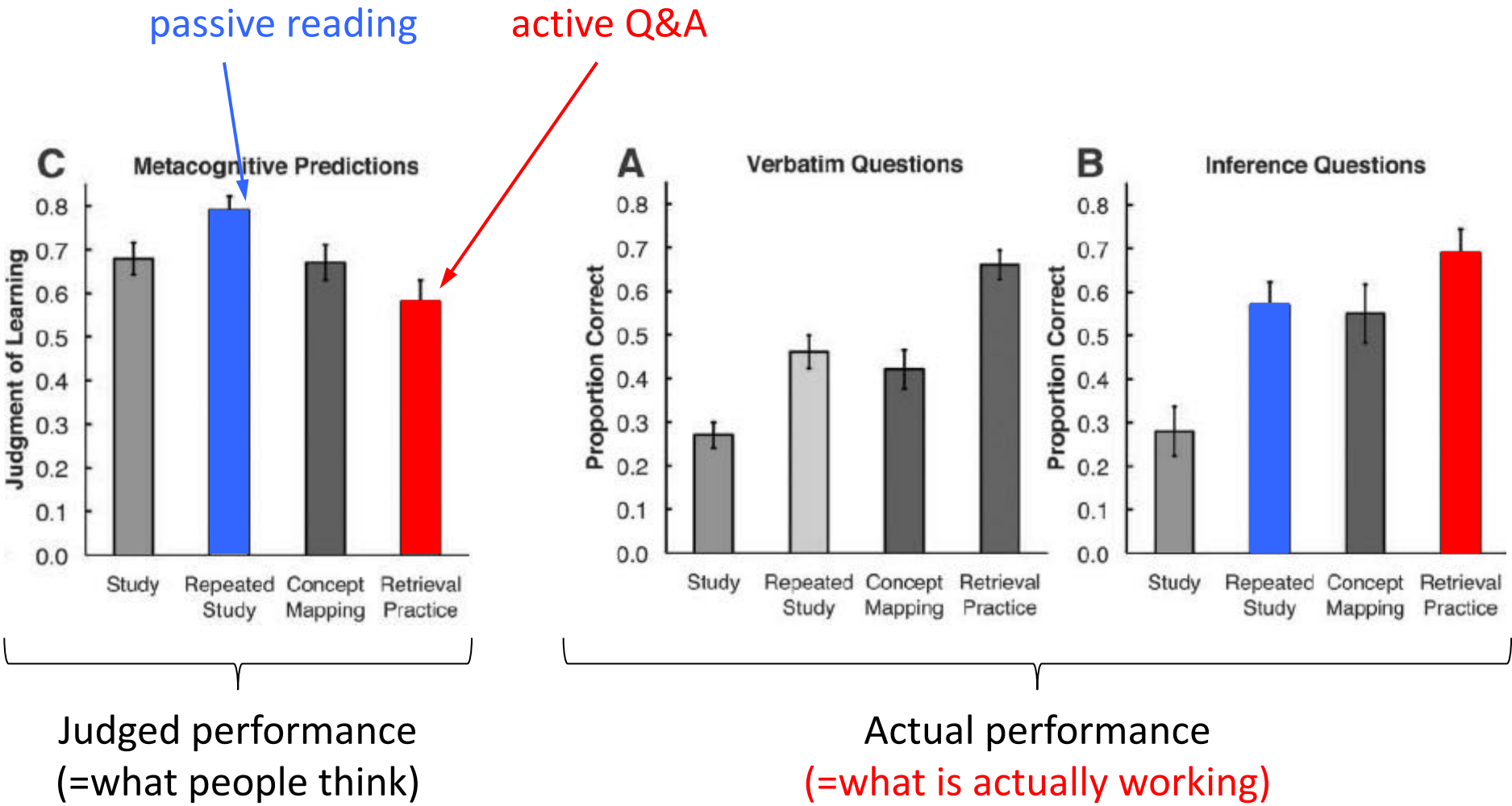
Let this book, then, be your guide to mental push-ups. Think carefully about the questions and their answers *before* you read the answers offered by the author. **You will find many answers don't turn out as you first expect. Does this mean you have no sense for physics? Not at all. Most questions were deliberately chosen to illustrate those aspects of physics which seem contrary to casual surmise. Revising ideas, even in the privacy of your own mind, is not painless work.** But in doing so you will revisit some of the problems that haunted the minds of Archimedes, Galileo, Newton, Maxwell, and Einstein.* The physics you cover here in hours took them centuries to master. Your hours of thinking will be a rewarding experience. Enjoy!

Lewis Epstein

Source: "Thinking Physics: Understanding Practical Reality", Lewis Carroll Epstein, 1979-2009.

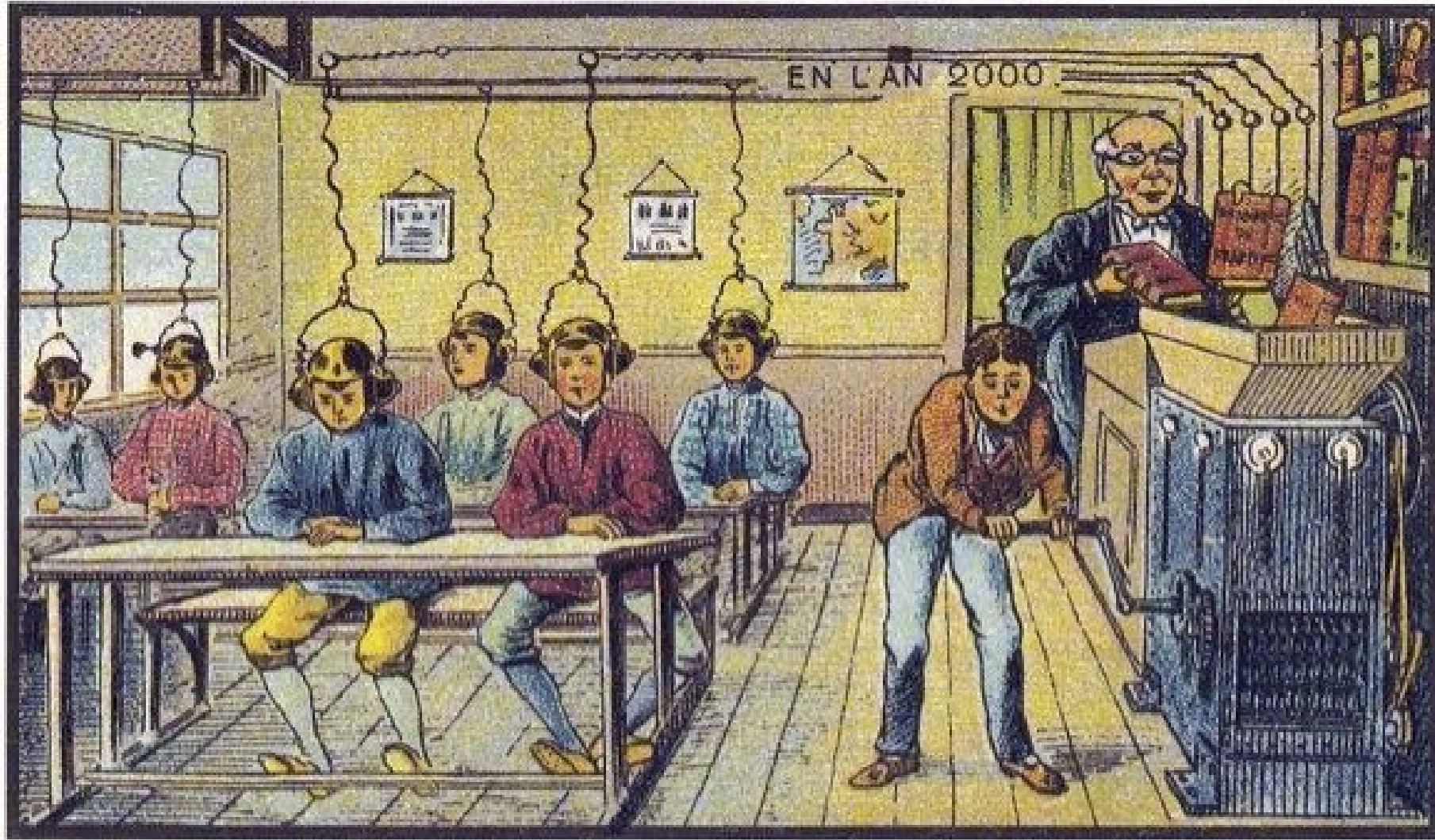
<http://www.amazon.com/Thinking-Physics-Understandable-Practical-Reality/dp/0935218084>

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



Source: Karpicke & Blunt, "Retrieval Practice Produces More Learning than Elaborative Studying with Concept Mapping," Science, 2011.

The year 2000 imagined in 1900



At School

Announcements!

- Textbooks (v2): link to Amazon international ed
- Python, Jupyter
- Keep up the great class interactions 😊
- Microphone
- Continue giving feedback
- Talk announcement today at 3pm

DISTINGUISHED SPEAKER: RETHINKING QUERY EXECUTION ON BIG DATA

JANUARY 18 3:00 PM - 4:30 PM EST



Title: Rethinking Query Execution on Big Data

Speaker: Dan Suciu, Professor of Computer Science at the University of Washington

Location: Northeastern University, 45 Forsyth St., Cargill Hall, Lower Level, Room #97, Boston, Massachusetts 02115

Abstract

Database engines today use the same approach to evaluate a query as they did forty years ago: convert the query into a query plan, then execute each operator individually, e.g. a join, followed by another join, followed by duplicate elimination. It turns out that converting a query into binary joins is theoretically suboptimal, and this can lead to poor performance over very large datasets. The theoretical database research community has studied a new query evaluation paradigm, which in some cases leads to provably optimal algorithms. In this talk I will give a brief survey of this new paradigm: I will review the AGM bound on the query size (Atserias, Grohe and Marx), the worst-case optimal “generic join” algorithm for full conjunctive queries (Ngo, Re, and Rudra), and our new algorithm for aggregate queries, called PANDA, which matches the best known running times for certain graph problems.

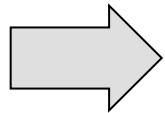
Table Alias (Tuple Variables)



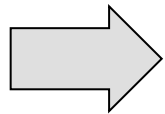
Person (pName, address, works_for)
University (uName, address)

```
SELECT DISTINCT pName, address
FROM   Person, University
WHERE  works_for = uName
```

which address?
Error!



```
SELECT DISTINCT Person.pName, University.address
FROM   Person, University
WHERE  Person.works_for = University.uName
```



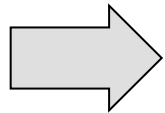
```
SELECT DISTINCT X.pName, Y.address
FROM   Person as X, University Y
WHERE  X.works_for = Y.uName
```

Note that "as" is optional !!

Column Alias (rename attributes)



Person (pName, address, works_for)
University (uName, address)



```
SELECT DISTINCT X.pName as name, Y.address adr  
FROM      Person as X, University Y  
WHERE     X.works_for = Y.uName
```

```
SELECT DISTINCT X.pName, Y.address  
FROM      Person as X, University Y  
WHERE     X.works_for = Y.uName
```


Quiz 2

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)



Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Q: Find all US companies that manufacture products in the 'Gadgets' category!

```
SELECT cName
FROM
WHERE
```

Quiz 2

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)



Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Q: Find all US companies that manufacture products in the 'Gadgets' category!

```
SELECT cName
FROM Product P, Company
WHERE country = 'USA'
      and P.category = 'Gadgets'
      and P.manufacturer = cName
```



Cname
GizmoWorks
GizmoWorks

Quiz 2

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)



302

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Q: Find all US companies that manufacture products in the 'Gadgets' category!

```
SELECT DISTINCT cName
FROM   Product P, Company
WHERE  country = 'USA'
       and Pcategory = 'Gadgets'
       and P.manufacturer = cName
```



Cname
GizmoWorks

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

Q: Find all US companies that manufacture both a product below \$20 and a product above \$25.

```
SELECT DISTINCT cName  
FROM  
WHERE
```

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

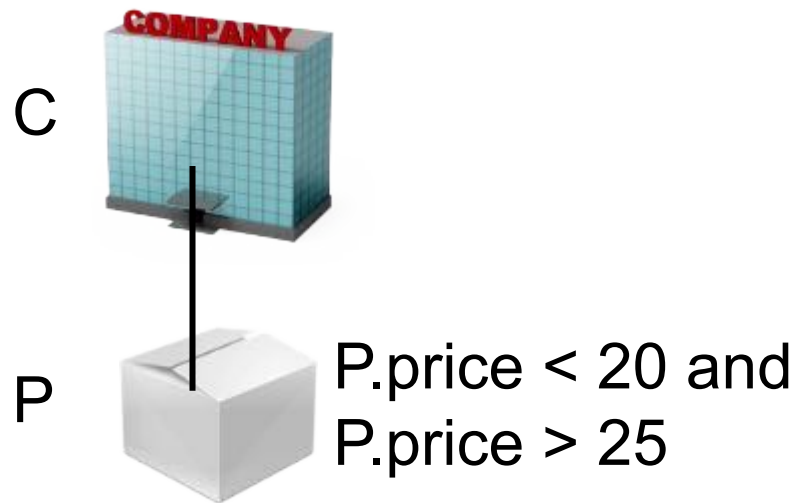
Q: Find all US companies that manufacture both a product below \$20 and a product above \$25.

```
SELECT DISTINCT cName  
FROM Product as P, Company  
WHERE country = 'USA'  
      and P.price < 20  
      and P.price > 25  
      and P.manufacturer = cName
```

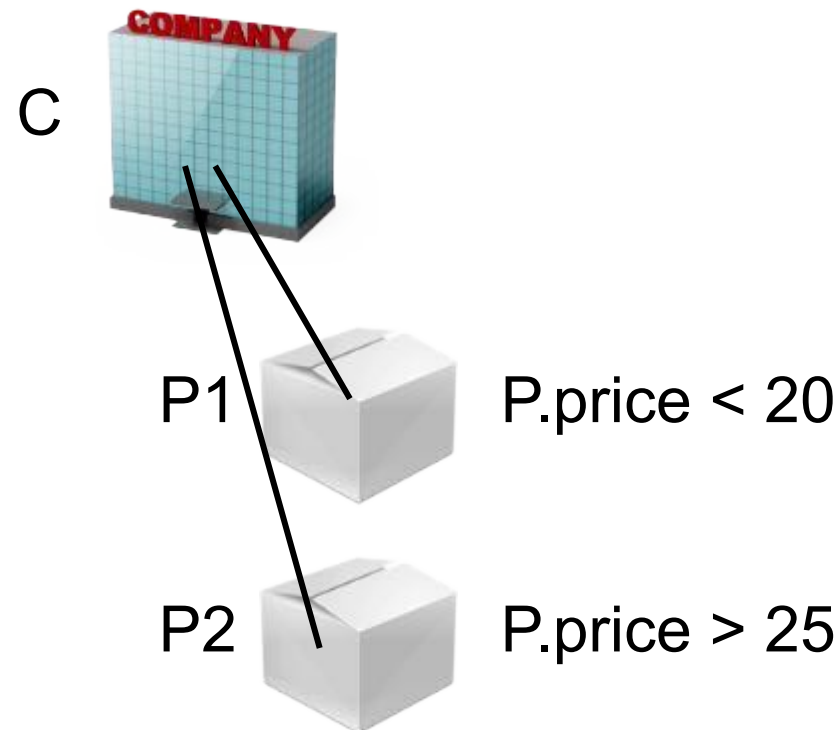
Wrong! Gives empty
result: There is no
product with price
<20 and >25

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

Q: Find all US companies that manufacture both a product below \$20 and a product above \$25.



not possible!
-> Empty result

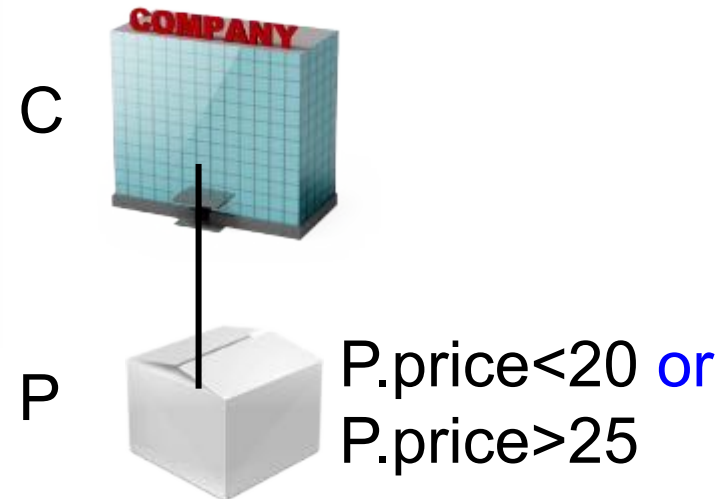


Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

Q: Find all US companies that manufacture both a product below \$20 and a product above \$25.

```
SELECT DISTINCT cName
FROM Product as P, Company
WHERE country = 'USA'
      and (P.price < 20
      or   P.price > 25)
      and P.manufacturer = cName
```

Returns companies
with single product
w/price (<20 or >25)



Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

Q: Find all US companies that manufacture both a product below \$20 and a product above \$25.

```
SELECT DISTINCT cName
FROM   Product as P1, Product as P2, Company
WHERE  country = 'USA'
       and P1.price < 20
       and P2.price > 25
       and P1.manufacturer = cName
       and P2.manufacturer = cName
```

Quiz 3



P1

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
...

P2

PName	Price	Category	Manufacturer
...
Powergizmo	\$29.99	Gadgets	GizmoWorks

Company

CName	StockPrice	Country
GizmoWorks	25	USA
...

```
SELECT DISTINCT cName
FROM Product as P1, Product as P2, Company
WHERE country = 'USA'
      and P1.price < 20
      and P2.price > 25
      and P1.manufacturer = cName
      and P2.manufacturer = cName
```



Cname
GizmoWorks

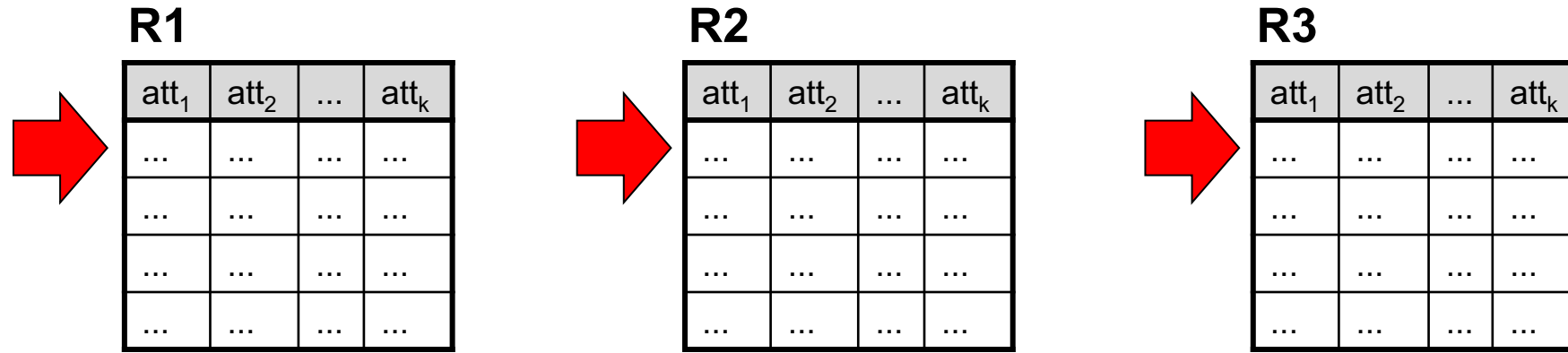
Meaning (Semantics) of conjunctive SQL Queries

```
SELECT  a1, a2, ..., ak  
FROM    R1 as x1, R2 as x2, ..., Rn as xn  
WHERE   Conditions
```

Conceptual evaluation strategy (nested for loops):

```
Answer = {}  
for x1 in R1 do  
    for x2 in R2 do  
        .....  
            for xn in Rn do  
                if Conditions  
                    then Answer = Answer ∪ {(a1, ..., ak)}  
return Answer
```

Meaning (Semantics) of conjunctive SQL Queries



```
Answer = {}  
for  $x_1$  in  $R_1$  do  
    for  $x_2$  in  $R_2$  do  
        .....  
            for  $x_n$  in  $R_n$  do  
                if Conditions  
                    then Answer = Answer  $\cup$   $\{(a_1, \dots, a_k)\}$   
return Answer
```

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
 - FROM: Compute the cross-product of relation-list.
 - WHERE: Discard resulting tuples if they fail qualifications.
 - SELECT: Delete attributes that are not in target-list.
 - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.

Inner Joins



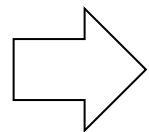
Employee

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

Department

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

```
SELECT *  
FROM Employee E, Department D  
WHERE E.DepartmentID = D. DepartmentID
```



E.LastName	E.DepartmentID	D.DepartmentID	D.DepartmentName
Robinson	34	34	Clerical
Jones	33	33	Engineering
Smith	34	34	Clerical
Steinberg	33	33	Engineering
Rafferty	31	31	Sales

Cross Joins: usually not what you want ☹️

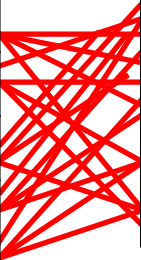


Employee

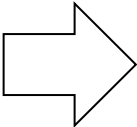
LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

Department

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing



```
SELECT *  
FROM Employee E, Department D  
WHERE E.DepartmentID = D.DepartmentID
```



E.LastName	E.DepartmentID	D.DepartmentID	D.DepartmentName
Rafferty	31	31	Sales
Jones	33	31	Sales
Steinberg	33	31	Sales
Smith	34	31	Sales
Robinson	34	31	Sales
Rafferty	31	33	Engineering
Jones	33	33	Engineering
Steinberg	33	33	Engineering
Smith	34	33	Engineering
Robinson	34	33	Engineering
Rafferty	31	34	Clerical
Jones	33	34	Clerical
Steinberg	33	34	Clerical
Smith	34	34	Clerical
Robinson	34	34	Clerical
Rafferty	31	35	Marketing
Jones	33	35	Marketing
Steinberg	33	35	Marketing
Smith	34	35	Marketing
Robinson	34	35	Marketing

Definitions (for job interviews?)

- An equi-join is a join in which the joining condition is based on equality between values in the common columns; common columns appear redundantly in the result table
- A natural join is an equi-join in which one of the duplicate columns is eliminated in the result table
- A cross join returns the Cartesian product of rows from tables in the join
 - (i.e. it will produce rows which combine each row from the first table with each row from the second table, that's usually **not** what you want)

Definitions (for job interviews?)

Equi-join

E.LastName	E.DepartmentID	D.DepartmentID	D.DepartmentName
Robinson	34	34	Clerical
Jones	33	33	Engineering
Smith	34	34	Clerical
Steinberg	33	33	Engineering
Rafferty	31	31	Sales

Natural join

E.LastName	DepartmentID	D.DepartmentName
Robinson	34	Clerical
Jones	33	Engineering
Smith	34	Clerical
Steinberg	33	Engineering
Rafferty	31	Sales

Cross join

E.LastName	E.DepartmentID	D.DepartmentID	D.DepartmentName
Rafferty	31	31	Sales
Jones	33	31	Sales
Steinberg	33	31	Sales
Smith	34	31	Sales
Robinson	34	31	Sales
Rafferty	31	33	Engineering
...

Alternative JOIN Syntax



Employee

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

Department

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

```
SELECT *
FROM Employee E, Department D
WHERE E.DepartmentID = D. DepartmentID
AND E.DepartmentID = 34
```

```
SELECT *
FROM Employee E JOIN Department D
      ON E.DepartmentID = D. DepartmentID
WHERE E.DepartmentID = 34
```

E.LastName	E.DepartmentID	D.DepartmentID	D.DepartmentName
Robinson	34	34	Clerical
Smith	34	34	Clerical

NATURAL JOIN Syntax



Employee

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

Department

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

```
SELECT *  
FROM Employee E, Department D  
WHERE E.DepartmentID = D. DepartmentID  
AND E.DepartmentID = 34
```

```
SELECT *  
FROM Employee E NATURAL JOIN Department D  
WHERE E.DepartmentID = 34
```

Syntax is not
supported by all
DBMS's

LastName	DepartmentID	DepartmentName
Robinson	34	Clerical
Smith	34	Clerical

Using the Formal Semantics

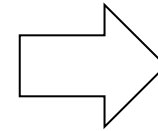
$R(a), S(a), T(a)$



What do these queries compute?

R	S	T
a	a	a
1	1	2
2		

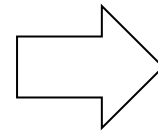
```
SELECT DISTINCT R.a
FROM   R, S
WHERE  R.a=S.a
```



a
1

Returns $R \cap S$
(intersection)


```
SELECT DISTINCT R.a
FROM   R, S, T
WHERE  R.a=S.a
      or R.a=T.a
```



a
1
2

Returns $R \cap (S \cup T)$
if $S \neq \emptyset$ and $T \neq \emptyset$

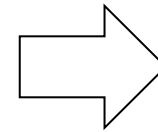
Using the Formal Semantics

$R(a), S(a), T2(a)$  305

What do these queries compute?

R	S	T2
a	a	a
1	1	
2		

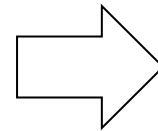
```
SELECT DISTINCT R.a
FROM   R, S
WHERE  R.a=S.a
```



a
1

Returns $R \cap S$
(intersection)

```
SELECT DISTINCT R.a
FROM   R, S, T2 as T
WHERE  R.a=S.a
      or R.a=T.a
```



a

Returns \emptyset
if $S = \emptyset$ or $T = \emptyset$

Can seem counterintuitive! But remember conceptual evaluation strategy: Nested loops. If one table is empty \rightarrow no looping

Illustration with Python



305

```
1 '''
2 Created on 3/23/2015
3 Illustrates nested Loop Join in SQL
4 __author__ = 'gatt'
5 '''
6
7 print "--- 1st nested loop ---"
8 for i in xrange(2):
9     for j in xrange(3):
10         for k in xrange(2):
11             print "i=%d, j=%d, k=%d: " % (i, j, k),
12             if i == j or i == k:
13                 print "TRUE",
14             print
15
16 print "\n--- 2nd nested loop ---"
17 for i in xrange(2):
18     for j in xrange(3):
19         for k in xrange(1):
20             print "i=%d, j=%d, k=%d: " % (i, j, k),
21             if i == j or i == k:
22                 print "TRUE",
23             print
24
25 print "\n--- 3rd nested loop ---"
26 for i in xrange(2):
27     for j in xrange(3):
28         for k in xrange(0):
29             print "i=%d, j=%d, k=%d: " % (i, j, k),
30             if i == j or i == k:
31                 print "TRUE",
32             print
33
```

/Library/Frameworks/Python.framework/Versio

--- 1st nested loop ---

```
i=0, j=0, k=0: TRUE
i=0, j=0, k=1: TRUE
i=0, j=1, k=0: TRUE
i=0, j=1, k=1:
i=0, j=2, k=0: TRUE
i=0, j=2, k=1:
i=1, j=0, k=0:
i=1, j=0, k=1: TRUE
i=1, j=1, k=0: TRUE
i=1, j=1, k=1: TRUE
i=1, j=2, k=0:
i=1, j=2, k=1: TRUE
```

--- 2nd nested loop ---

```
i=0, j=0, k=0: TRUE
i=0, j=1, k=0: TRUE
i=0, j=2, k=0: TRUE
i=1, j=0, k=0:
i=1, j=1, k=0: TRUE
i=1, j=2, k=0:
```

--- 3rd nested loop ---

Process finished with exit code 0

The comparison gets never evaluated

1. Aggregates
2. Groupings
3. Having

Aggregation

Car (name, price, maker)



```
SELECT avg(price)
FROM Car
WHERE maker='Toyota'
```

```
SELECT count(*)
FROM Car
WHERE price > 100
```

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute


Aggregation

```
SELECT avg(price)
FROM Car
WHERE maker='Toyota'
```

Car

<u>Name</u>	Price	Maker
M3	120	BMW
M5	150	BMW
Prius	50	Toyota
Lexus1	75	Toyota
Lexus2	100	Toyota

Database creates new attribute
name (for SQLserver)



(No column name)
75

Aggregation with rename

"as" optional

```
SELECT count(*) as n  
FROM Car  
WHERE price > 100
```

Car

<u>Name</u>	Price	Maker
M3	120	BMW
M5	150	BMW
Prius	50	Toyota
Lexus1	75	Toyota
Lexus2	100	Toyota

Database creates *our*
new attribute name



n
2

Aggregation: Count Distinct



```
SELECT count(maker)
FROM Car
WHERE price > 100
```

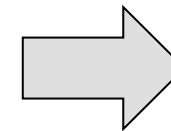
Same as `count(*)`

Car

<u>Name</u>	Price	Maker
M3	120	BMW
M5	150	BMW
Prius	50	Toyota
Lexus1	75	Toyota
Lexus2	100	Toyota

We probably want to ignore duplicates:

```
SELECT count(DISTINCT maker)
FROM Car
WHERE price > 100
```



(No column name)
1

Simple Aggregation 1/3



Purchase (product, price, quantity)

```
SELECT sum(price * quantity)
FROM Purchase
```

```
SELECT sum(price * quantity)
FROM Purchase
WHERE product = 'Bagel'
```

What do these
queries mean?

Simple Aggregation 2/3

Purchase

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

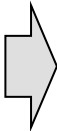
$3 * 20 = 60$

$2 * 20 = 40$

sum: 100

Database creates
new attribute name

```
SELECT sum(price * quantity)
FROM Purchase
WHERE product = 'Bagel'
```



(No column name)
100

Simple Aggregation 3/3



Purchase

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

320

220

sum: 5 * sum: 40 = 200

```
SELECT sum(price) * sum(quantity)
FROM Purchase
WHERE product = 'Bagel'
```

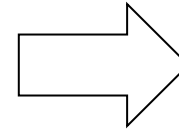
(No column name)
200

Grouping and Aggregation



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

Notice: we use "sales" for total number of products sold

Find total quantities for all purchases with price over \$1 grouped by product.

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

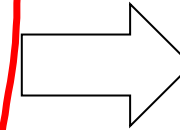
```
1 4 SELECT      product, sum(quantity) as TotalSales
2 FROM        Purchase
3 WHERE       price > 1
4 GROUP BY    product
```

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	?

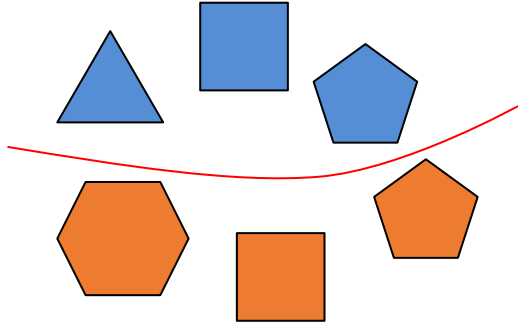
What quantity should the DB return for Banana?

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

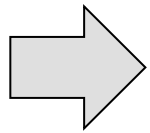
The DB engine is confused, there is no single quantity for banana (it's an ill-defined query). It should thus return an error (only SQLite misbehaves and returns something, but which makes no sense). Please think this through carefully!

Groupings illustrated with colored shapes

group by color

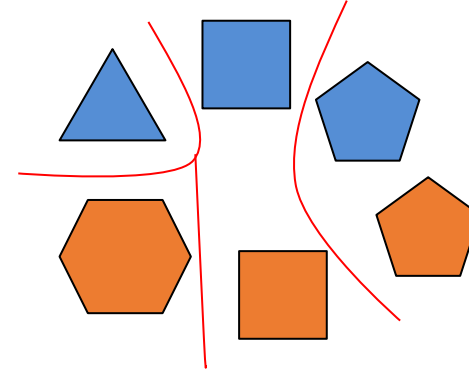


```
SELECT color,  
       avg(numc) anc  
FROM   Shapes  
GROUP BY color
```

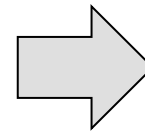


color	anc
blue	4
orange	5

group by numc (# of corners)



```
SELECT numc  
FROM   Shapes  
GROUP BY numc
```

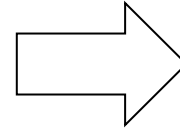


numc
3
4
5
6

Another Example

Purchase

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



Product	SumQ	MaxP
Bagel	40	3
Banana	70	4

```
SELECT product,  
       sum(quantity) as SumQ,  
       max(price) as MaxP  
FROM   Purchase  
GROUP BY product
```

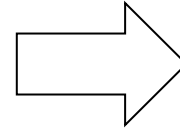
*Next, focus only on
products with at
least 50 sales*

Having Clause



Q: Similar to before, but only products with at least 50 sales.

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



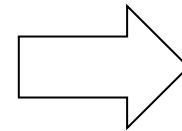
Product	SumQ	MaxP
Banana	70	4

```
SELECT product,  
        sum(quantity) as SumQ,  
        max(price) as MaxP  
FROM Purchase  
GROUP BY product  
HAVING sum(quantity) > 50
```

Quizz

What does this query return over the given database?

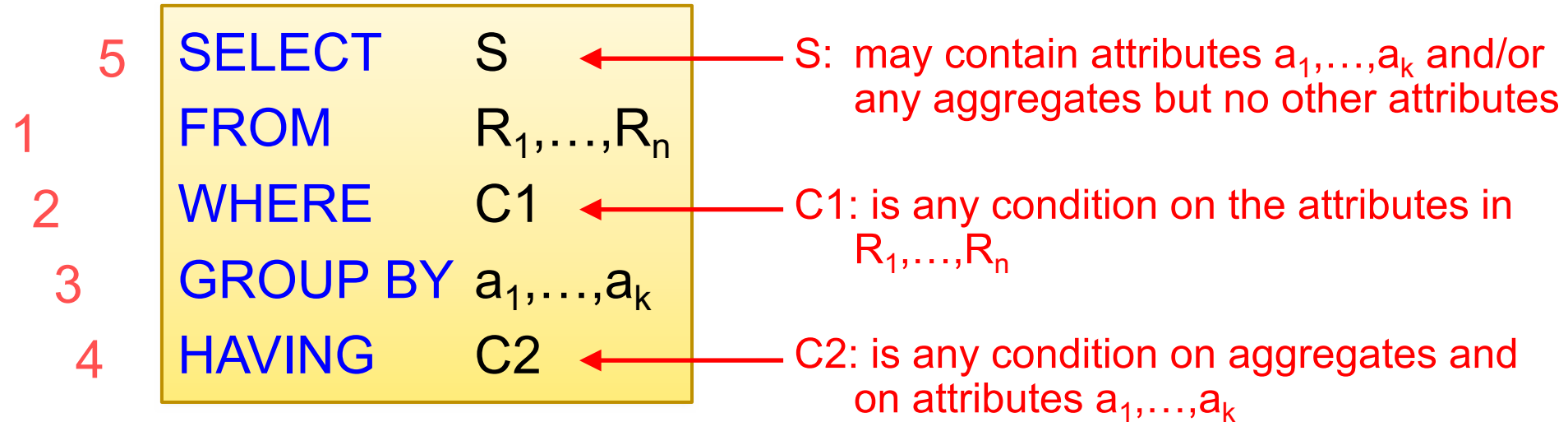
Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10



Product	SumQ
Bagel	40
Banana	50

```
SELECT product, sum(quantity) as SumQ
FROM Purchase
WHERE quantity > 15
GROUP BY product
HAVING sum(quantity) > 40
```

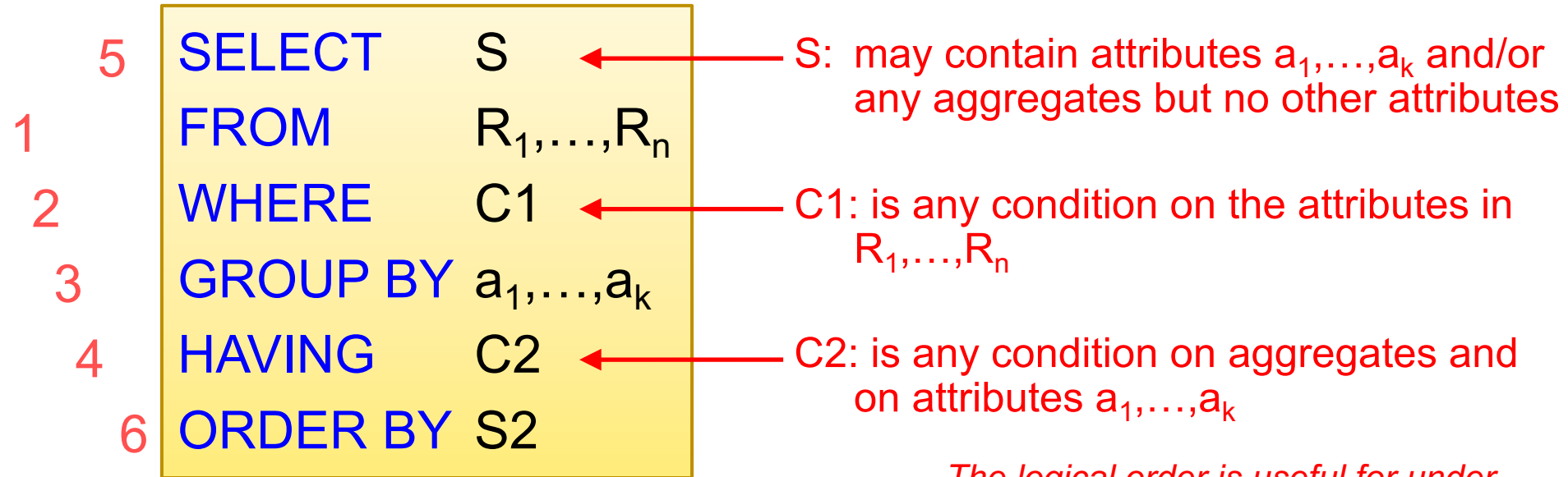
General form of Grouping and Aggregation



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

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*

Conceptual Evaluation Strategy

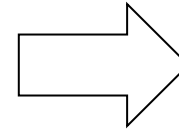
- The cross-product of relation-list is computed (FROM), tuples that fail qualification are discarded (WHERE), then:
- GROUP BY: the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- HAVING: The group-qualification is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group!
 - In effect, an attribute in group-qualification that is not an argument of an aggregate op must also appear in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

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
Bagel	40
Banana	50

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

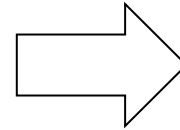
```
SELECT product, sum(quantity) as SumQ
FROM Purchase
WHERE quantity > 15
GROUP BY product
HAVING SumQ > 35
```


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
```