Updated 3/12/2022

Topic 3: Efficient query evaluation
Unit 1: Acyclic query evaluation
Lecture 16

Wolfgang Gatterbauer

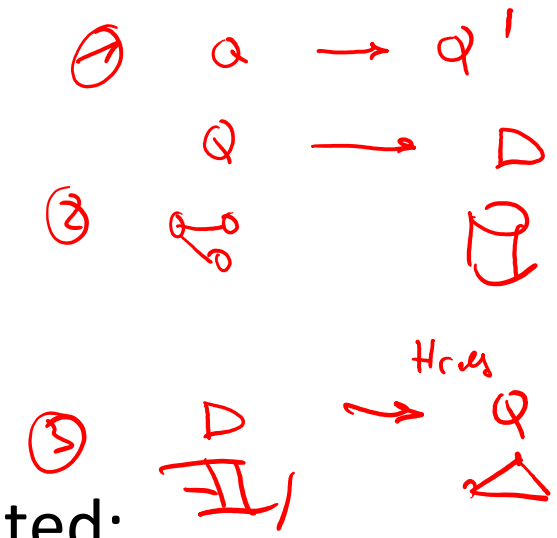
CS7240 Principles of scalable data management (sp22)

https://northeastern-datalab.github.io/cs7240/sp22/

3/11/2022

Islands of Tractability of CQ Evaluation

- Major Research Program: Identify tractable cases of the combined complexity of conjunctive query evaluation.
- Over the years, this program has been pursued by two different research communities:
  - The Database Theory community
  - The Constraint Satisfaction community
- Explanation: Problems in those community are closely related:



Constraint Satisfaction Problem ≡ Homomorphism Problem ≡ CQ evaluation

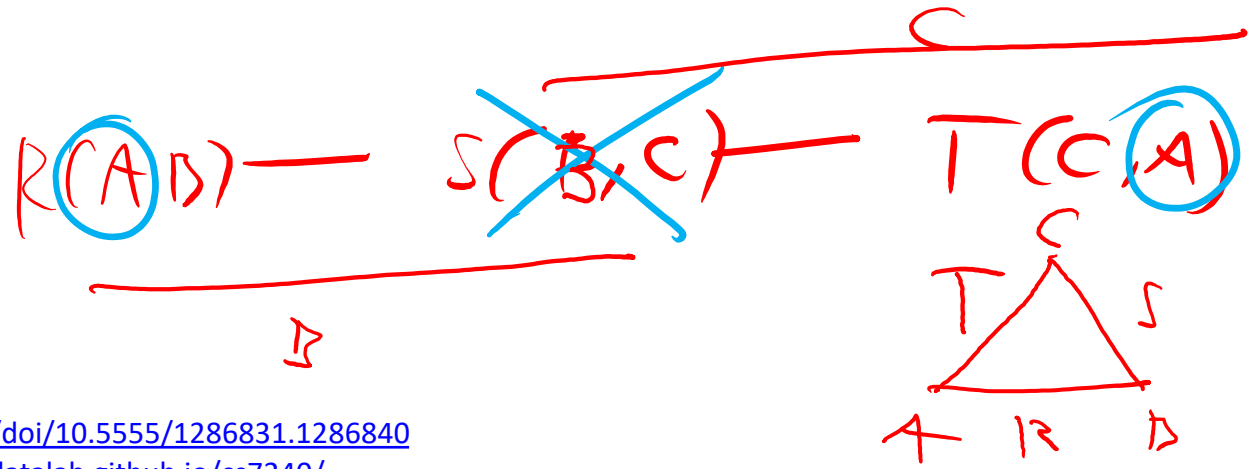
[Feder, Vardi 1993] [Chandra, Merlin 1977]

[Kolaitis, Vardi 2000]

Feder, Vardi: Monotone monadic SNP and constraint satisfaction, STOC 1993 https://doi.org/10.1145/167088.167245 / Kolaitis, Vardi: Conjunctive-Query Containment and Constraint Satisfaction, JCSS 2000 https://doi.org/10.1006/jcss.2000.1713 / Chandra, Merlin. "Optimal implementation of conjunctive queries in relational data bases", STOC 1977. https://doi.org/10.1145/800105.803397
Based on Phokion Kolaitis' "Logic and Databases" series at Simons Institute, 2016. https://simons.berkeley.edu/talks/logic-and-databases
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

Acyclic queries

- Like many areas in computer science, cycles complicate things
  - Same with conjunctive query (CQ) evaluation
- Acyclic CQs are a large and useful tractable case CQs
- A query is acyclic if its relations can be placed in a tree (join tree) s.t.
  - the set of nodes that contain any variable form a connected set
- Yannakakis' algorithm [81]: any acyclic query can be computed in time: O(|Input| + |Output|)



Yannakakis, "Algorithms for acyclic database schemes", VLDB 1981. https://dl.acm.org/doi/10.5555/1286831.1286840
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- T3-2: Cyclic conjunctive queries

Semijoin (⋉): derived RA operator

Actor (aid, name, gender)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$ RA

where A_1, …, A_n are the attributes in R

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: ?

Semijoin (⋉): derived RA operator

Actor (aid, name, gender)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$ RA

where A_1, …, A_n are the attributes in R

SQL: ?

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: $\text{Actor} \ltimes \text{Casts}$

Semijoin (⋉): derived RA operator

Actor (aid, name, gender)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$ RA

where A_1, …, A_n are the attributes in R

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: $\text{Actor} \ltimes \text{Casts}$

SQL:

Semijoins have no "direct" representation in SQL (just like relational division)

```
SELECT DISTINCT *
FROM Actor
WHERE aid IN
        (SELECT aid
        FROM Casts)
```

Alternative WHERE EXISTS

Duplicates in R are preserved!

Equivalent only if no duplicates in R!

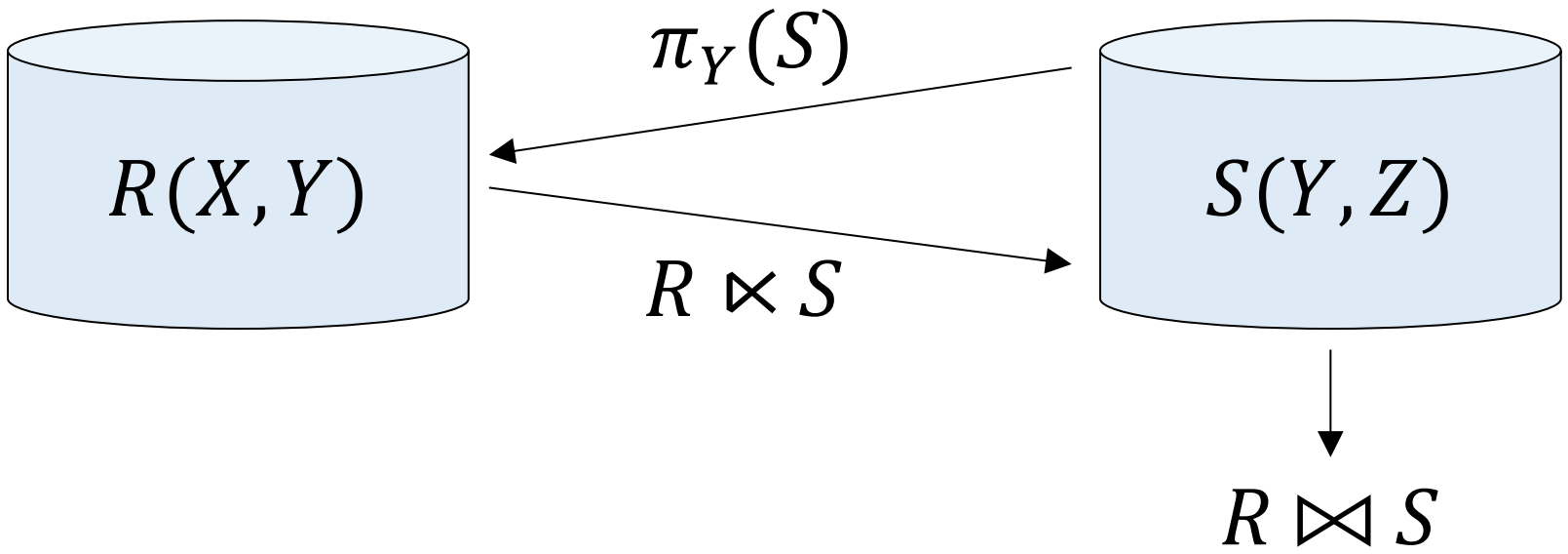
```
SELECT DISTINCT
         A.aid, name, age
FROM Actor A, Casts C
WHERE A.aid = C.aid
```

Semijoins in Distributed Databases

- Semijoins are often used to compute equijoins in distributed databases

$R \bowtie S$

Goal: send less data to reduce network bandwidth!



$R \bowtie S = (R \ltimes S) \bowtie S$ law of semijoins

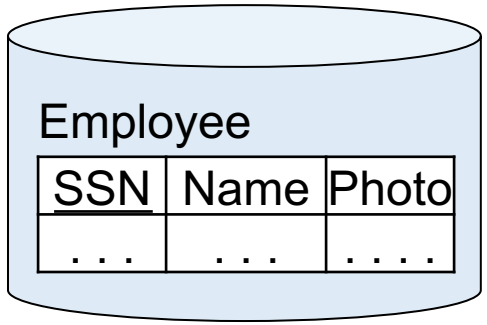
reduced

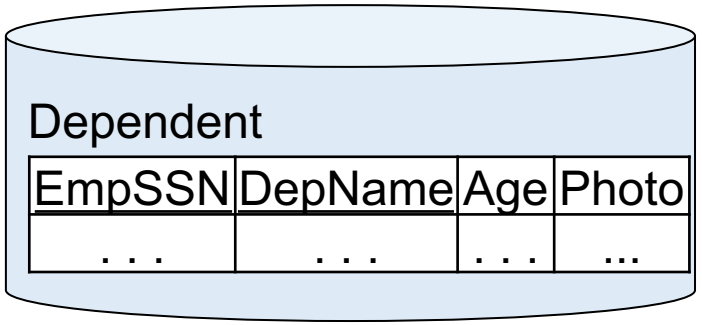
Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71}(\text{Dependent}))$$

Assumptions:
1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. "Photo" is big.





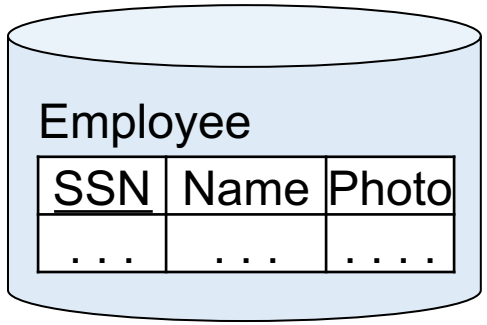
?

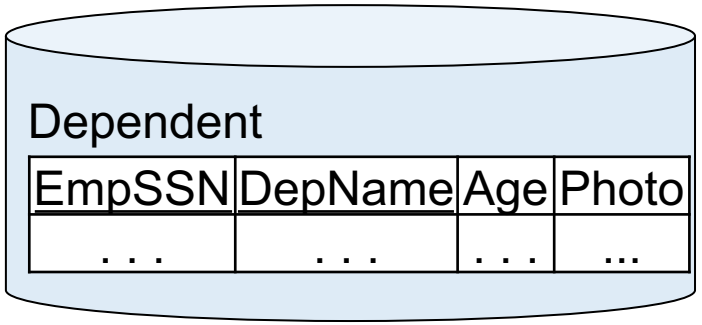
Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent}))$$

Assumptions:
1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. "Photo" is big.





R

$$R(SSN) = \pi_{\text{EmpSSN}} \sigma_{\text{age>71}}(\text{Dependents})$$

$$\begin{aligned} L &= \text{Employee} \bowtie_{\text{SSN=EmpSSN}} R \\ &= \text{Employee} \ltimes_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent})) \\ &= \text{Employee} \ltimes_{\text{SSN=EmpSSN}} R \end{aligned}$$

L

$$\text{Answer} = L \bowtie_{\text{SSN=EmpSSN}} \text{Dependent}$$

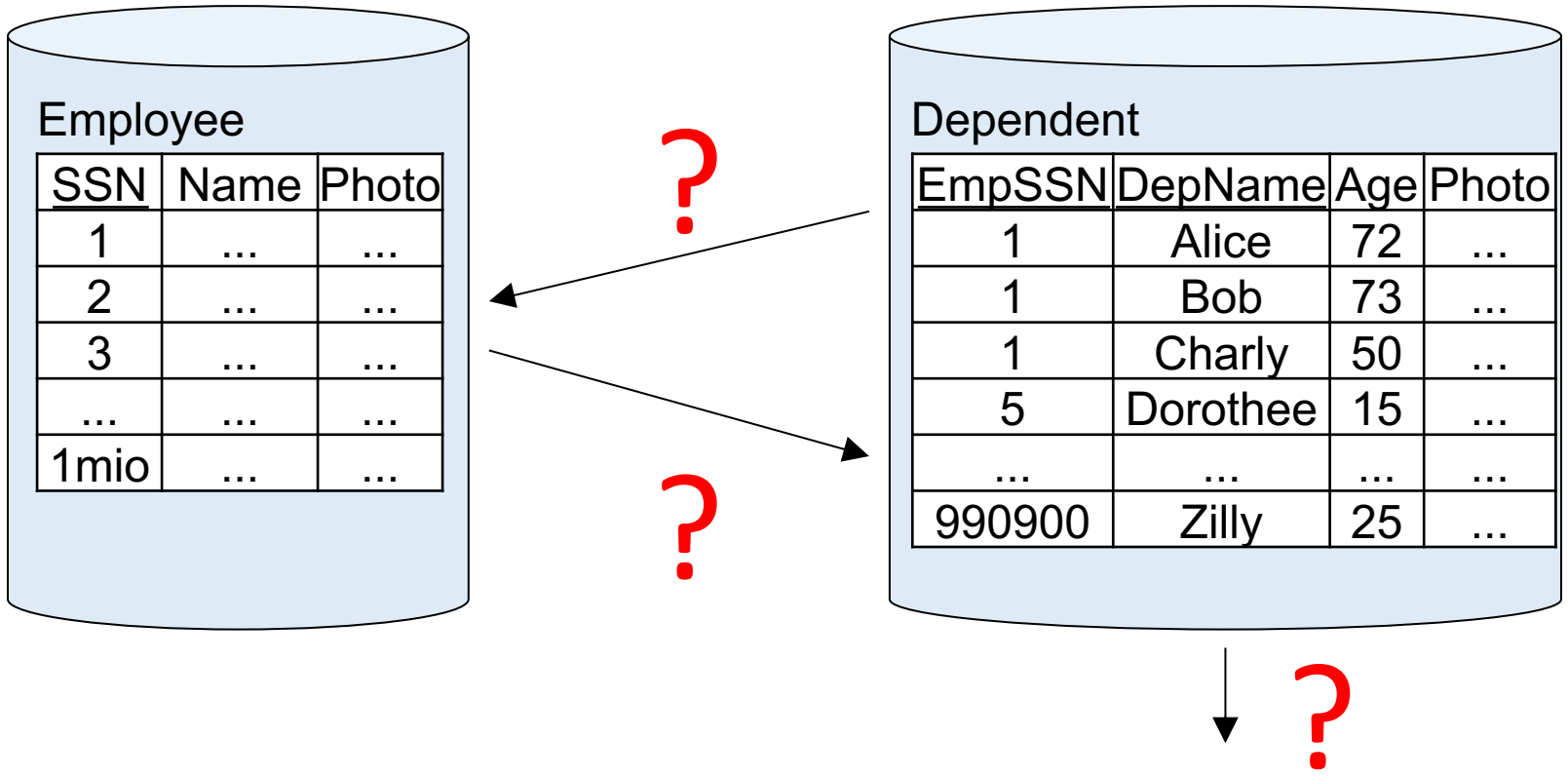
Based upon an example by Dan Suciu

Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71}(\text{Dependent}))$$

Assumptions:
1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. "Photo" is big.

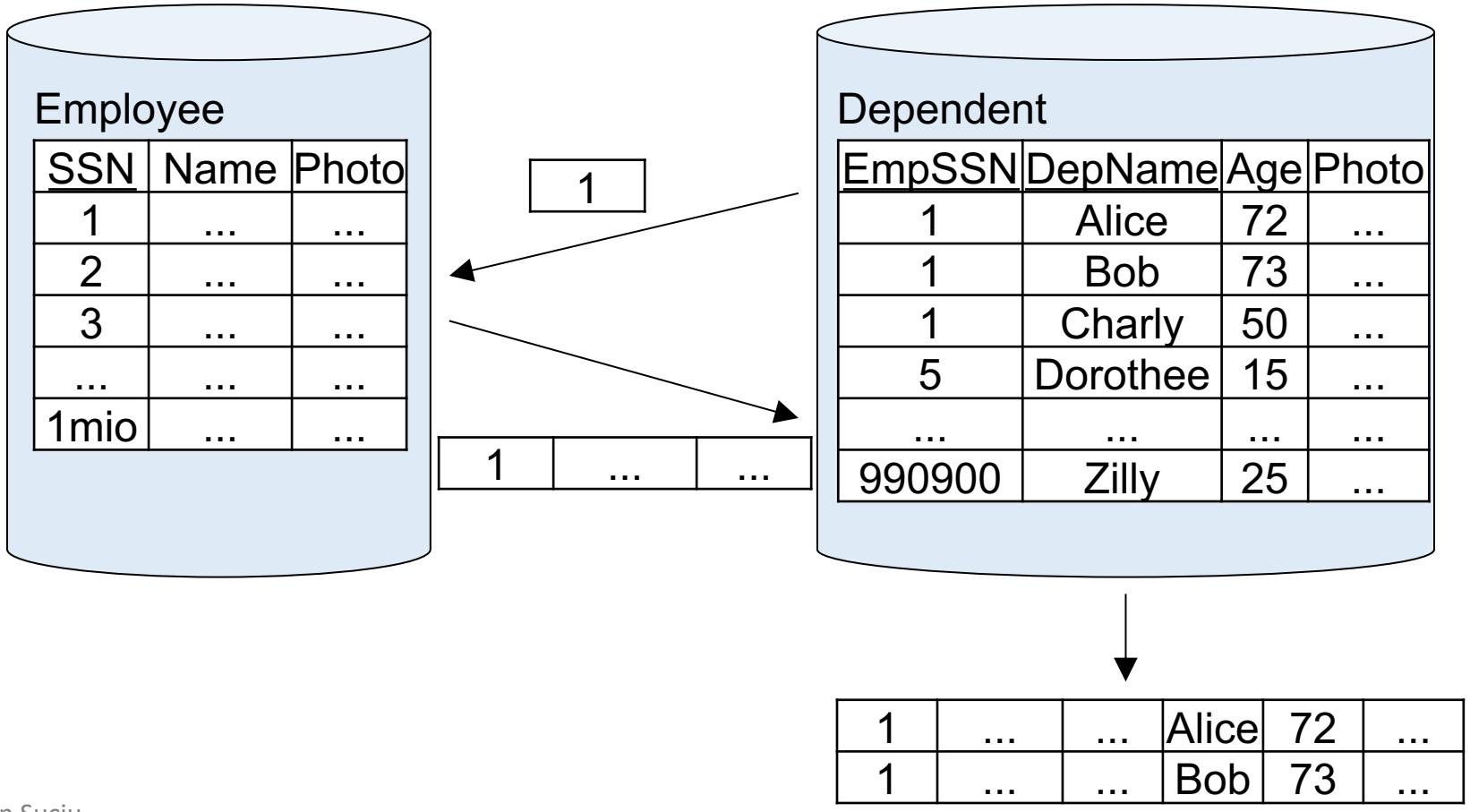


Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71}(\text{Dependent}))$$

Assumptions:
1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. "Photo" is big.



Based upon an example by Dan Suciu

Summary: Law of Semijoins

- Definition: the semi-join operation is:

$$R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$

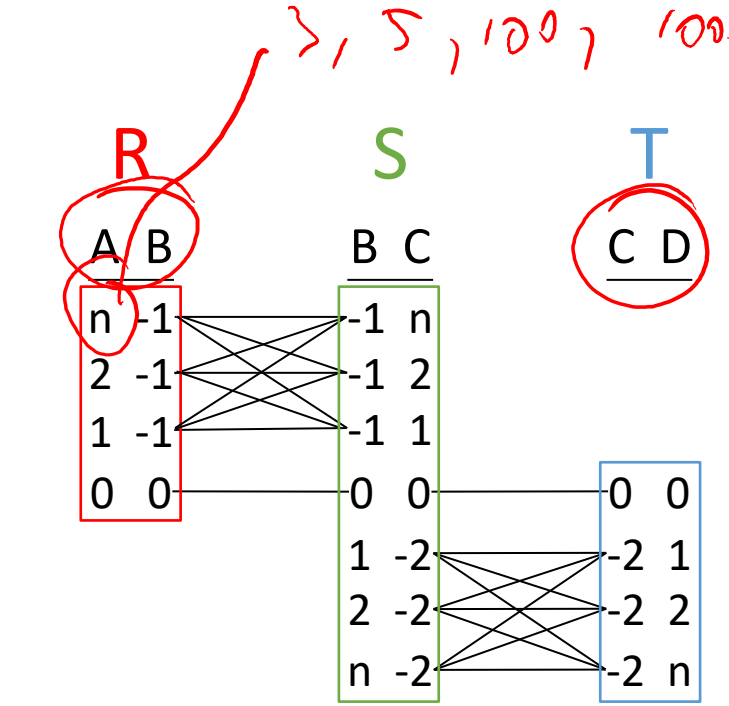
  - Formally, R ⋉ S means: retain from R only those tuples that have some matching tuple in S (in bag semantics: duplicates in R are preserved / Duplicates in S don't matter)
  - Data complexity: O(|R| + |S|) ignoring log-factors
  - Input: $R(A_1,\ldots,A_n)$, $S(B_1,\ldots,B_m)$, Output: $T(A_1,\ldots,A_n)$

- The law of semijoins is:

$$R \bowtie S = (R \ltimes S) \bowtie S$$

  - Thus, removing "dangling tuples" from a table does not change the query result

Semi-joins can also help if data are local



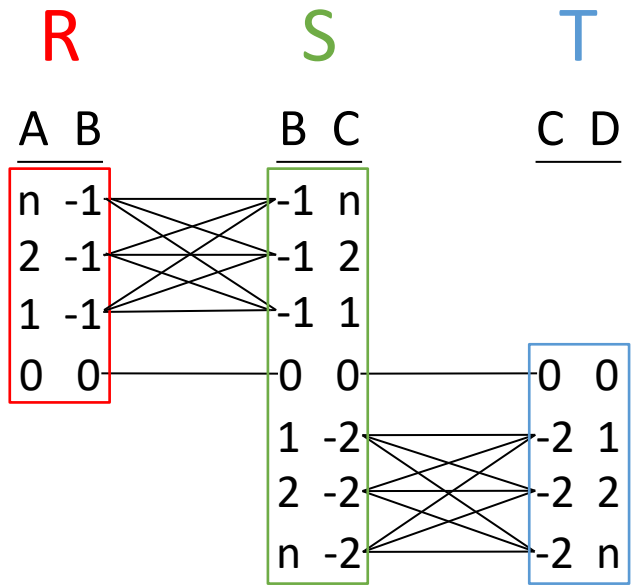
$$Q = (R \bowtie_B S) \bowtie_C T$$

What is the problem?

?

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Semi-joins can also help if data are local



$$Q = R \bowtie_B S \bowtie_C T$$

What is a typical query plan?

What is the problem?

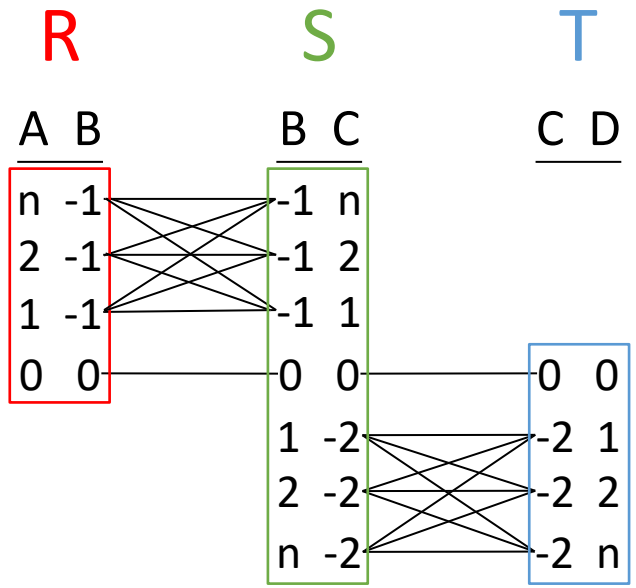
- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes $O(n^2)$ despite constant output ☹

?

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Semi-joins can also help if data are local



$$Q = R \bowtie_B S \bowtie_C T$$

What is the problem?

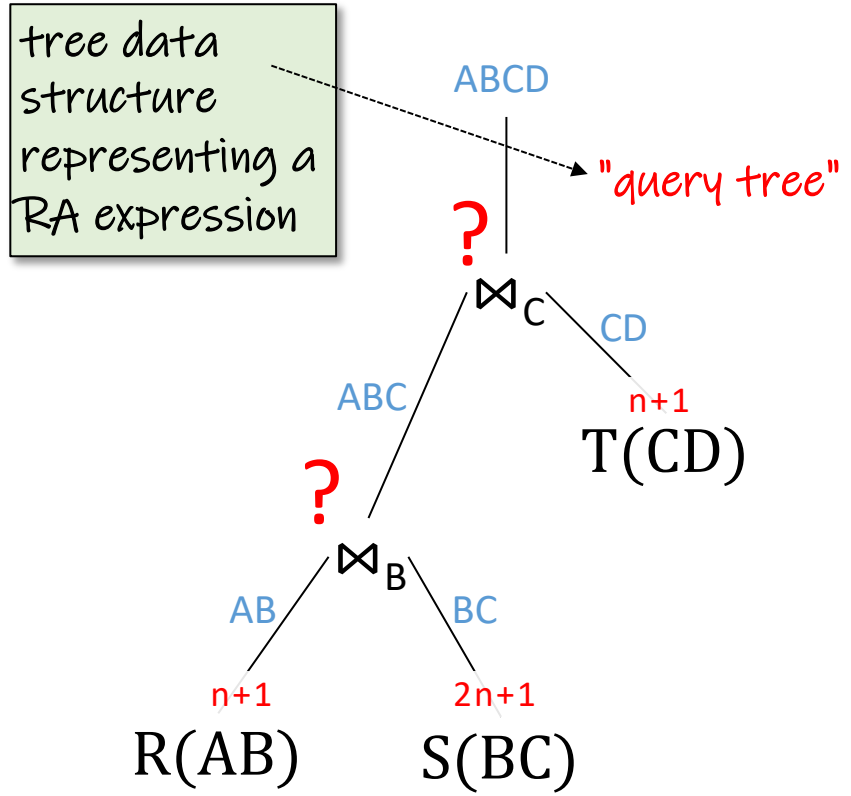
- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes $O(n^2)$ despite constant output ☹

What is a typical query plan?

$$Q = \left((R \bowtie_B S) \bowtie_C T\right)$$

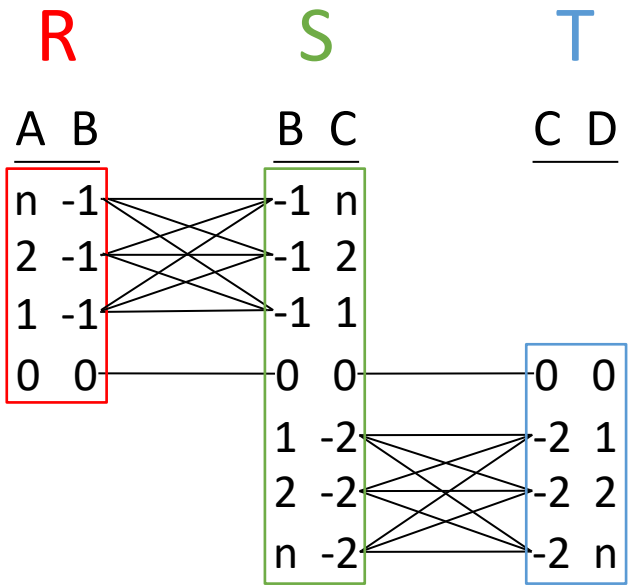
tree data structure representing a RA expression



What are the cardinalities at each stage?

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Semi-joins can also help if data are local



$$Q = R \bowtie_B S \bowtie_C T$$

What is the problem?

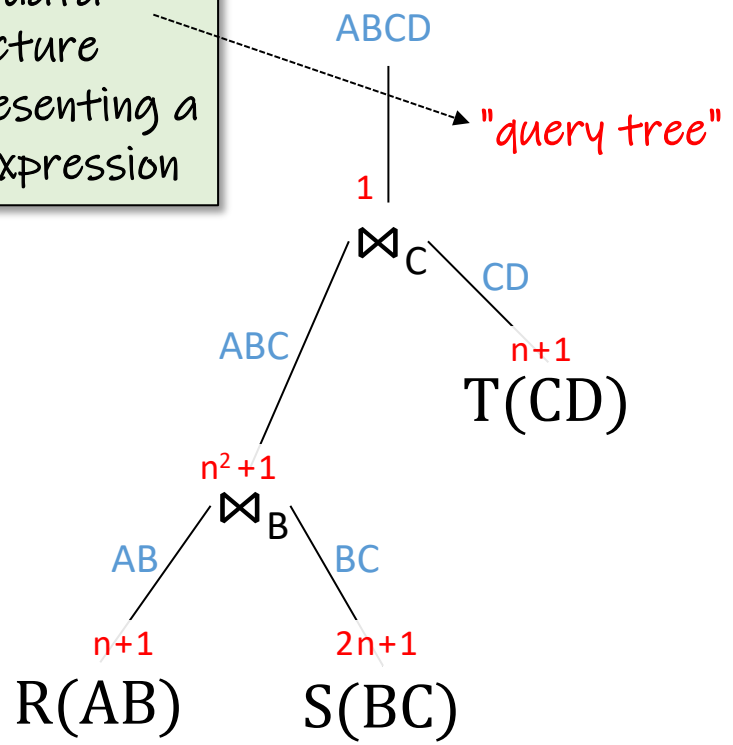
- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes $O(n^2)$ despite constant output ☹

What is a typical query plan?

$$Q = \left((R \bowtie_B S) \bowtie_C T\right)$$

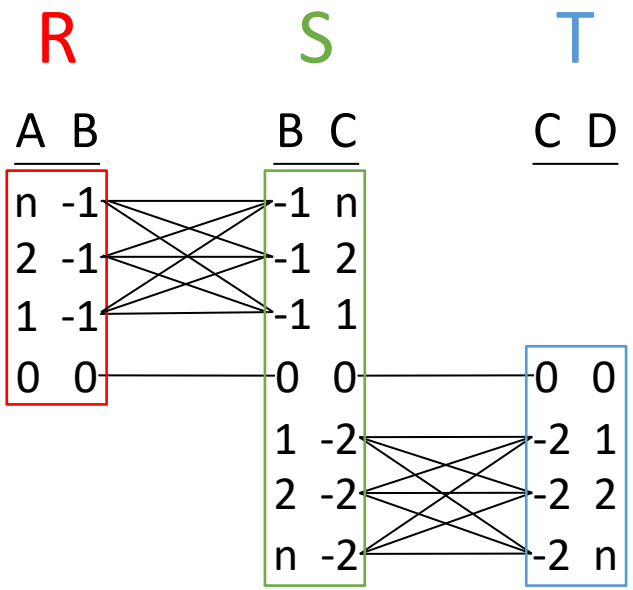
tree data structure representing a RA expression



What are the cardinalities at each stage?

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Semi-joins can also help if data are local



?

Can you think of a faster evaluation plan?

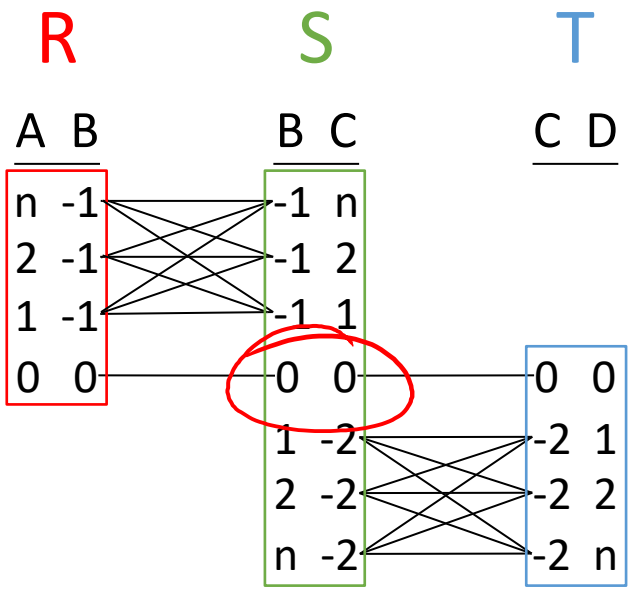
```
---------------------------
-- Query 1
---------------------------
select *
into record1
from R natural join S natural join T;
```

n=1000: t_{Q1}=1451 msec t_{Q2}=5 msec

n=2000: t_{Q1}=6104 msec t_{Q2}=8 msec

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Semi-joins can also help if data are local



WITH clauses (also **CTE** = Common Table Expression) act like temporary views

```
-------------------------
-- Query 1
-------------------------
select *
into record1
from R natural join S natural join T;
```

n=1000: t_{Q1}=1451 msec

n=2000: t_{Q1}=6104 msec

```
-------------------------
-- Query 2
-------------------------
With S2 as
     (SELECT *
     FROM S
     WHERE S.B in
          (SELECT R.B
          FROM R)),
S3 as
     (SELECT *
     FROM S2
     WHERE S2.C in
          (SELECT T.C
          FROM T))
select a, b, c, d
into record2
from R natural join S3 natural join T;
```

t_{Q2}=5 msec

t_{Q2}=8 msec

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

Diversion into CTE's (Common Table Expressions)


```
select *
into record3
from R natural join
    (SELECT *
    FROM
        (SELECT *
        FROM S
        WHERE S.B in
            (SELECT R.B
            FROM R)) S2
    WHERE S2.C in
        (SELECT T.C
        FROM T)) S3
natural join T;
```

WITH clauses (also **CTE** = Common Table Expression) act like temporary views

- Allow to deconstruct more complex queries into simple blocks to be used and reused if necessary.
- Can increase readability by emphasizing a more procedural interpretation of a query in a workflow
- Especially useful if you need to reference a derived table multiple times in a single query or you perform the same calculation multiple times across multiple query components (= memoization)

```
-------------------------
-- Query 2
-------------------------
With S2 as
    (SELECT *
    FROM S
    WHERE S.B in
        (SELECT R.B
        FROM R)),
S3 as
    (SELECT *
    FROM S2
    WHERE S2.C in
        (SELECT T.C
        FROM T))
select a, b, c, d
into record2
from R natural join S3 natural join T;
```

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

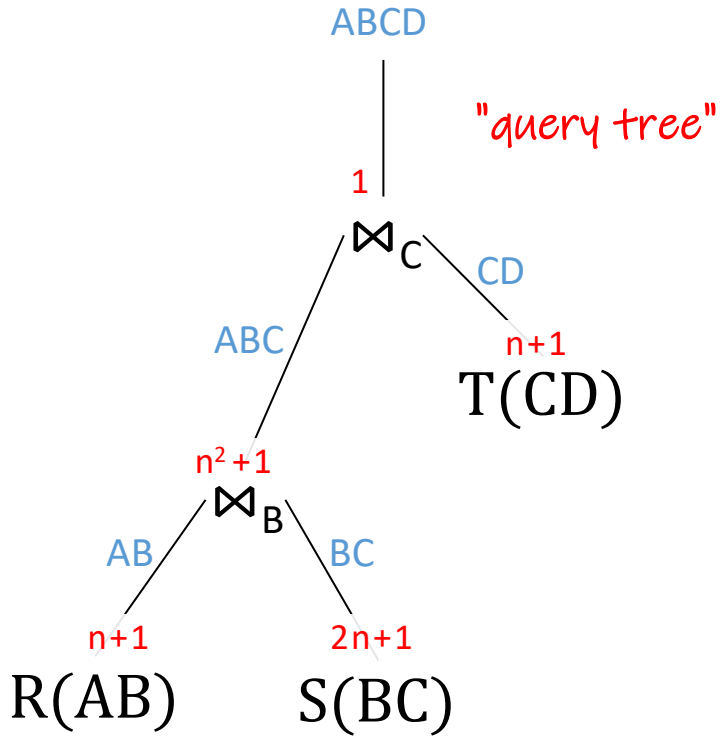
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

The more general idea: "Sideways information passing"

Sideways information passing:

- "sending information from one subexpression not simply to its parent expression, but also to some other correlated portion of the query computation, in order to prune irrelevant results" [Ives, Taylor 08]
- includes techniques like two-way semijoins [Bernstein, Goodman 81] and magic sets [Beeri, Ramakrishnan 91]

$$Q = \left((R \bowtie_B S) \bowtie_C T\right)$$



[Bernstein, Goodman 81]. "Using Semi-Joins to Solve Relational Queries", JACM 1981. https://doi.org/10.1145/322234.322238
[Beeri, Ramakrishnan 91]: "On the power of magic", Journal of Logic Programming, 1991. https://doi.org/10.1016/0743-1066(91)90038-Q
Definition from: [Ives, Taylor 08]. "Sideways Information Passing for Push-Style Query Processing", ICDE 2008. https://doi.org/10.1109/ICDE.2008.4497486
Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/