CompSci 516

Data Intensive Computing Systems

Lecture 21

Datalog

Instructor: Sudeepa Roy

Announcement

- HW3 due next Wednesday: 11/16

Today

- Datalog
  - for recursion in database queries
- A quick look at Incremental View Maintenance (IVM)

Reading Material: Datalog

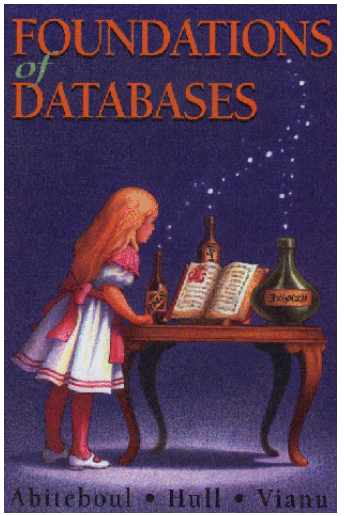
Optional:

1. The datalog chapters in the "Alice Book"

Foundations of Databases

Abiteboul-Hull-Vianu

Available online: http://webdam.inria.fr/Alice/



2. Datalog tutorial

SIGMOD 2011

"Datalog and Emerging Applications: An Interactive Tutorial"

Brief History of Datalog

- Motivated by Prolog – started back in 1970-80's – then quiet for a long time

- A long argument in the Database community whether recursion should be supported in query languages
  - *"No practical applications of recursive query theory ... have been found to date"*—Michael Stonebraker, 1998
    Readings in Database Systems, 3rd Edition Stonebraker and Hellerstein, eds.
  - Recent work by Hellerstein et al. on Datalog-extensions to build networking protocols and distributed systems. [Link]

Datalog is resurging!

- Number of papers and tutorials in DB conferences
- Applications in
  - data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing
- Systems supporting datalog in both academia and industry:
  - Lixto (information extraction)
  - LogicBlox (enterprise decision automation)
  - Semmle (program analysis)
  - BOOM/Dedalus (Berlekey)
  - Coral
  - LDL++

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Recall our drinker example in RC (Lecture 4)

Find drinkers that frequent some bar that serves some beer they like.

$$Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)$$

Drinker example is from slides by Profs. Balazinska and Suciu and the [GUW] book

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

RC:
$Q(x) = \exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)$

Datalog:
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

RC:
$Q(x) = \exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)$

Datalog:
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)

- Quick differences:
  - Uses ":-" not =
  - no need for $\exists$ (assumed by default)
  - Use "," on the right hand side (RHS)
  - Anything on RHS the of :- is assumed to be combined with $\wedge$ by default
  - $\forall$, $\Rightarrow$, not allowed – they need to use negation $\neg$
  - Standard "Datalog" does not allow negation
  - Negation allowed in datalog with negation
- How to specify disjunction (OR / $\vee$)?

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer "BestBeer"

RC:

$Q(x) = [\exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)] \quad \vee \quad [Likes(x, \text{“BestBeer”})]$

Datalog:

```
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer "BestBeer", (c) or, frequent bars that "Joe" frequents

RC:

$$Q(x) = [\exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)] \vee [Likes(x, \text{“BestBeer”})] \vee [\exists w\ Frequents(x, w) \wedge Frequents(\text{“Joe”}, w)]$$

Datalog:

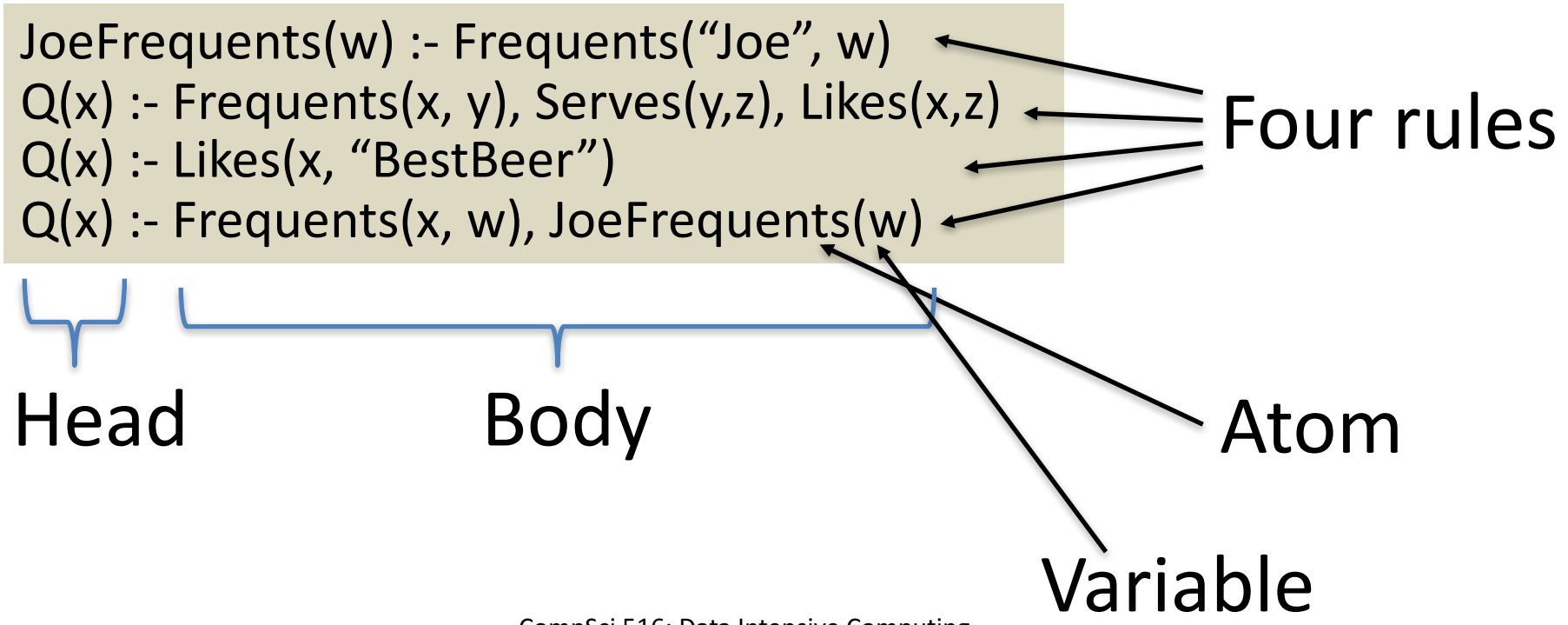
```
JoeFrequents(w) :- Frequents("Joe", w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
Q(x) :- Frequents(x, w), JoeFrequents(w)
```

- To specify "OR", write multiple rules with the same "Head"
- Next: terminology for Datalog

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Datalog Rules

- Each rule is of the form Head :- Body
- Each variable in the head of each rule must appear in the body of the rule



Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

EDBs and IDBs

- Extensional DataBases (EDBs)
  - Input relation names
  - e.g. Likes, Frequents, Serves
  - can only be on the RHS of a rule

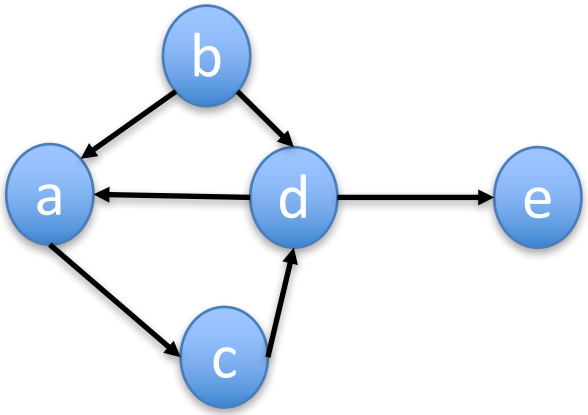
```
JoeFrequents(w) :- Frequents("Joe", w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
Q(x) :- Frequents(x, w), JoeFrequents(w)
```

- Intensional DataBases (IDBs)
  - Relations that are derived
  - Can be intermediate or final output tables
  - e.g. JoeFrequents, Q
  - Can be on the LHS or RHS (e.g. JoeFrequents)

Tuple in an EDB or an IDB: a FACT

either belongs to a given EDB relation, or is derived in an IDB relation

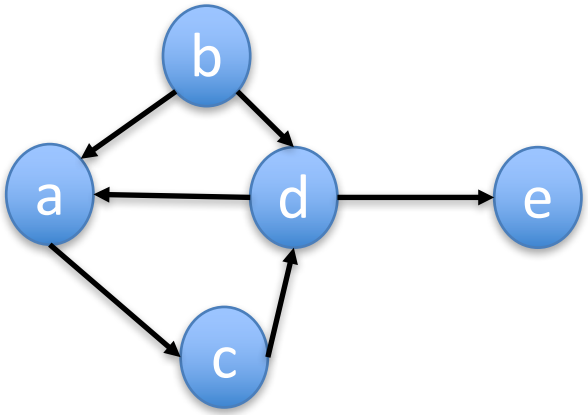
Graph Example



E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Example 1

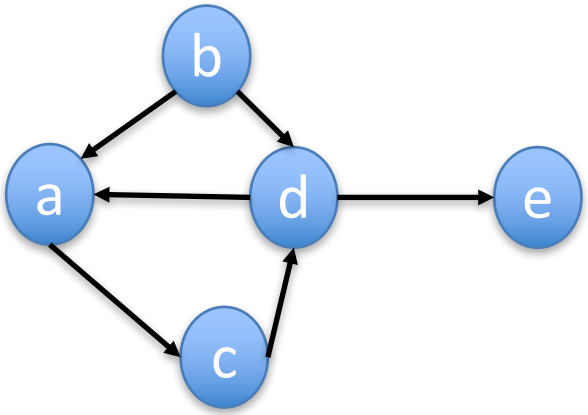


Write a Datalog program to find paths of length two (output start and finish vertices)

E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Example 1



Write a Datalog program to find paths of length two (output start and finish vertices)

E (edge relation)

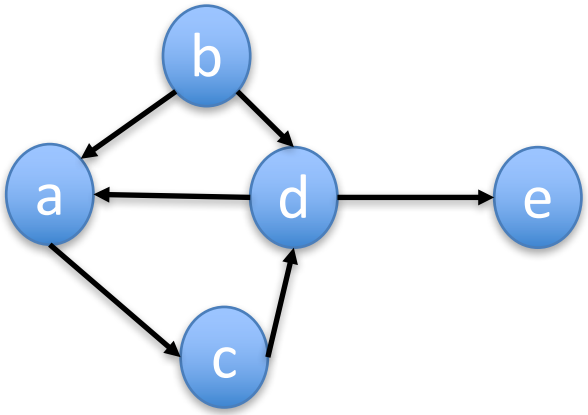
V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

```
P2(x, y) :- E(x, z), E(z, y)
```

E (edge relation)

Example 1: Execution

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



Write a Datalog program to find paths of length two (output start and finish vertices)

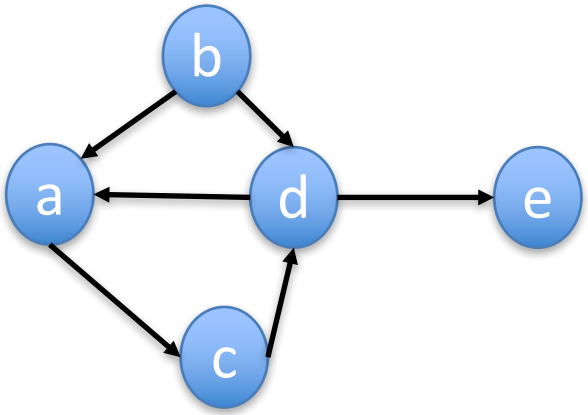
P2(x, y) :- E(x, z), E(z, y)

same as $E \bowtie_{E.V2=E.V1} E$

P2

V1	V2
a	d
b	c
b	e
c	a
c	e
d	c

Example 2



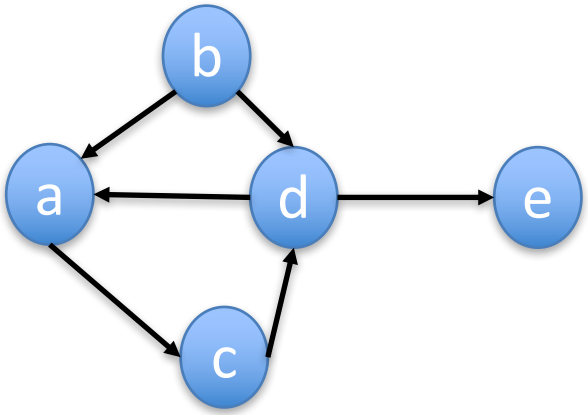
Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

- Can you write a SQL/RA/RC query for reachability?

Example 2



Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

E (edge relation)

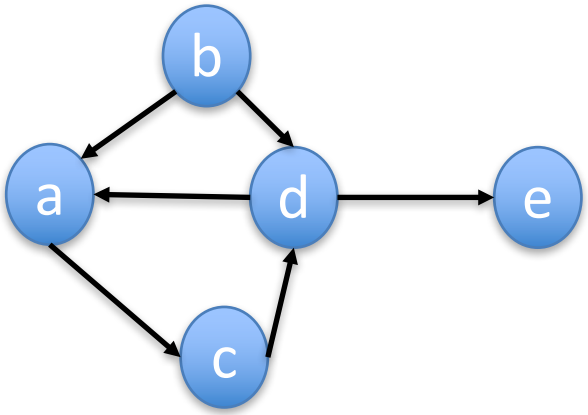
V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

- Can you write a SQL/RA/RC query for reachability?
- NO - SQL/RA/RC cannot express reachability

Example 2

E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

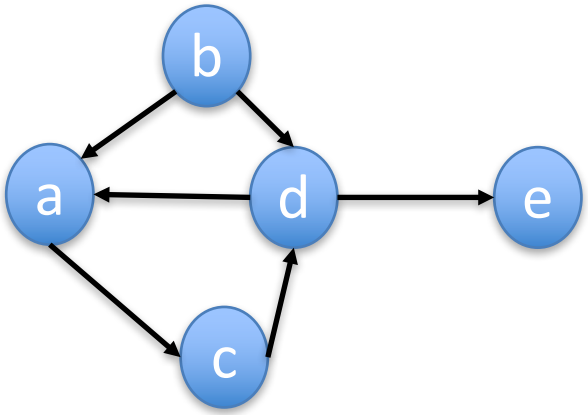
```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
```

Option 1

Example 2

E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
```

Option 1

non-linear

```
R(x, y) :- E(x, y)
R(x, y) :- R(x, z), R(z, y)
```

Option 3

linear

```
R(x, y) :- E(x, y)
R(x, y) :- R(x, z), E(z, y)
```

Option 2

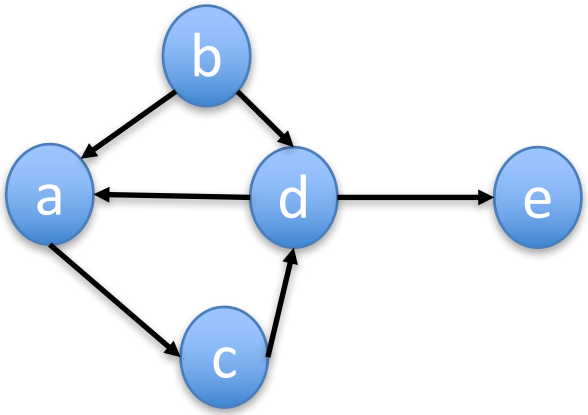
Linear Datalog

- Linear rule
  - at most one atom in the body that is recursive with the head of the rule
  - e.g. R(x, y) :- E(x, z), R(z, y)
- Linear datalog program
  - if all rules are linear
  - like linear recursion

- Top-down and bottom-up evaluation are possible
  - we will focus on bottom-up

Iteration 1

Example 2: Execution



E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

R = E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

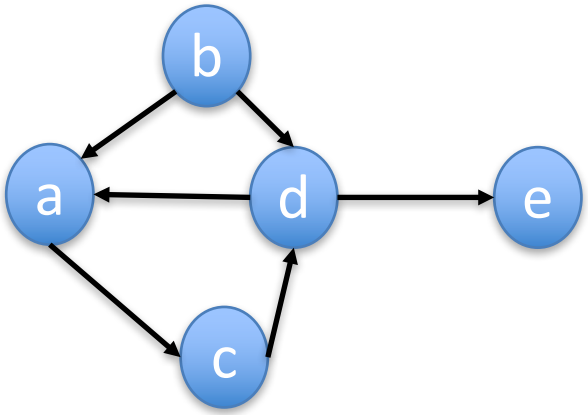
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

(vertices reachable in 1-hop by a direct edge)

Iteration 2

Example 2: Execution



E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

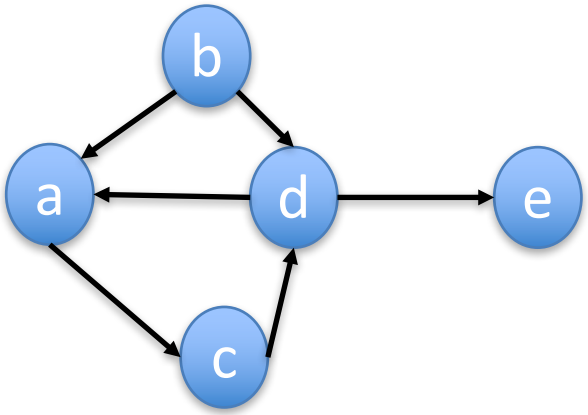
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

(vertices reachable in 2-hops)

Iteration 3

Example 2: Execution



E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

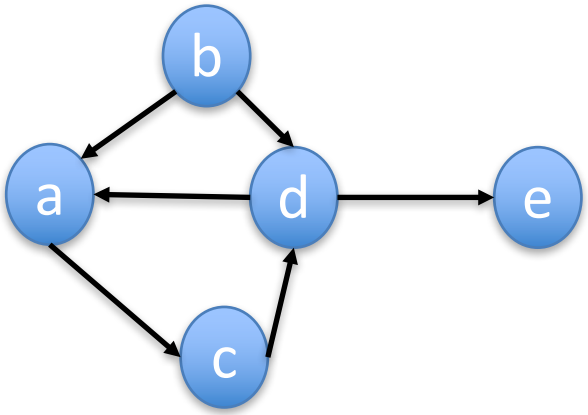
(vertices reachable in 3-hops)

R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

Iteration 4

Example 2: Execution



E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

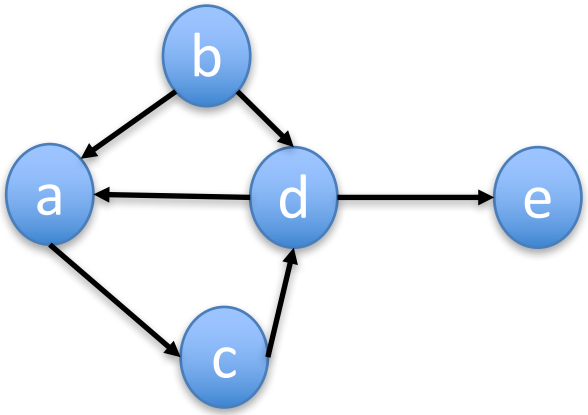
Option 1

R unchanged - stop

R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

Examples 3 and 4



E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Write a Datalog program to find all vertices reachable from b

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
QB(y) :- R(b, y)
```

Write a Datalog program to find all vertices u reachable from themselves R(u, u)

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
Q(x) :- R(x, x)
```

Termination of a Datalog Program

Q. A Datalog program always terminates – why?

Termination of a Datalog Program

Q. A Datalog program always terminates – why?

- Because the values of the variables are coming from the "active domain" in the input relations (EDBs)
- Active domain = (finite) values from the (possibly infinite) domain appearing in the instance of a database
  - e.g. age can be any integer (infinite), but active domain is only finitely many in R(id, name, age)
- Therefore the number of possible values in each of the IDBs is finite
- e.g. in the reachability example R(x, y), the values of x and y come from {a, b, c, d, e}
  - at most 5 x 5 = 25 tuples possible in the IDB R(x, y)
  - in any iteration, at least one new tuple is added in at least one IDB
  - Must stop after finite steps
  - e.g. the maximum number of iteration in the reachability example for any graph with five vertices is 25 (it was only 4 in our example)

Bottom-up Evaluation of a Datalog Program

- Naïve evaluation
- Semi-naïve evaluation

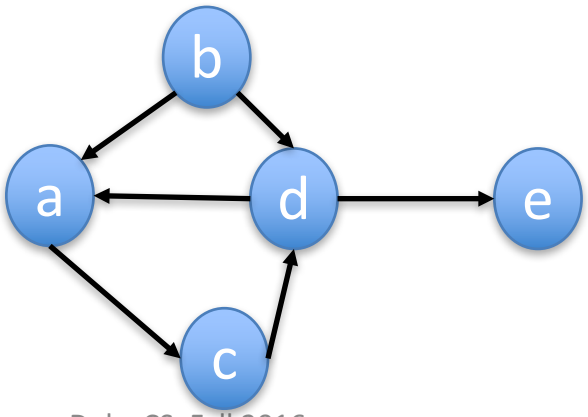
Naïve evaluation - 1

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)



In all subsequent iteration, check if any of the rules can be applied

Do union of all the rules with the same head IDB

Naïve evaluation - 2

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

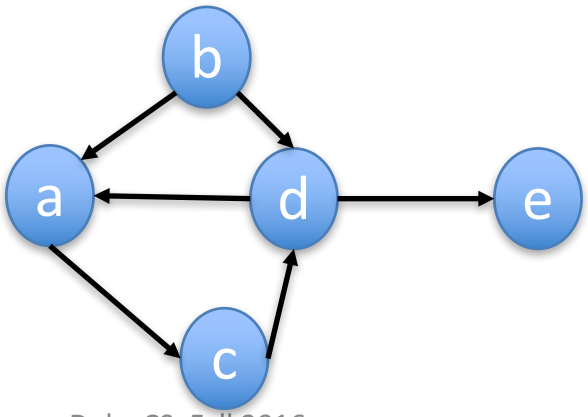
V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)

Iteration 2:
R = **E** ∪
E ⋈ R1
= R2 (say)

R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c



Naïve evaluation - 3

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)

Iteration 2:
R = **E** ∪
E ⋈ R1
= R2 (say)

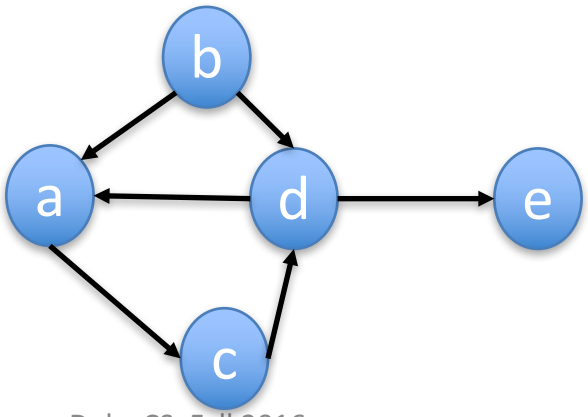
R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = **E** ∪
E ⋈ R2
= R3 (say)

R2 ≠ R3
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d



Naïve evaluation - 4

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)

Iteration 2:
R = **E** ∪
E ⋈ R1
= R2 (say)

R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

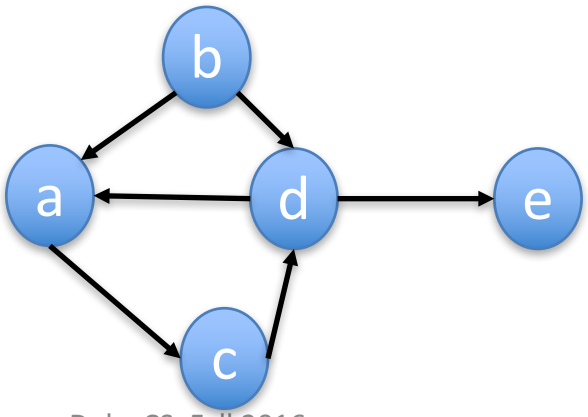
Iteration 3:
R = **E** ∪
E ⋈ R2
= R3 (say)

R2 ≠ R3
so continue

Iteration 4:
R = E ∪
E ⋈ R3
= R4 (say)

R3 = R4
so STOP

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d



Problem with Naïve Evaluation

- The same IDB facts are discovered again and again
  - e.g. in each iteration all edges in E are included in R
  - In the 2^{nd}-4^{th} iterations, the first six tuples in R are computed repeatedly

- Solution: Semi-Naïve Evaluation

- Work only with the new tuples generated in the previous iteration

Semi-Naïve evaluation - 1

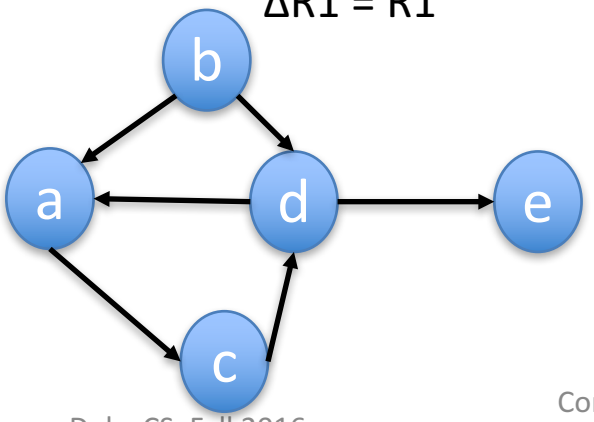
E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

Iteration 1:
R = E = R1 (say)
ΔR1 = R1



Semi-Naïve evaluation - 2

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

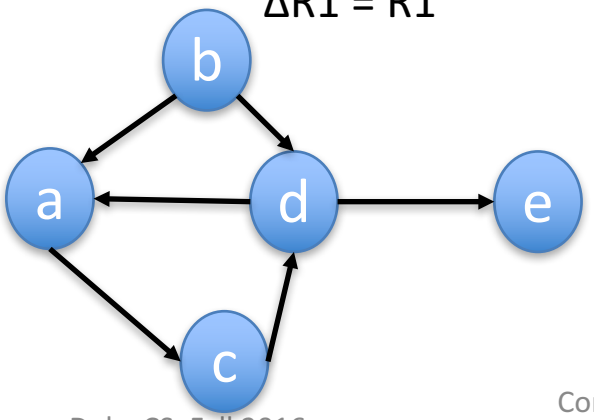
Iteration 1:
R = E = R1 (say)
ΔR1 = R1

Iteration 2:
R = R1 ∪
E ⋈ ΔR1
= R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c



Semi-Naïve evaluation - 3

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

Iteration 1:
R = E = R1 (say)
ΔR1 = R1

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 2:
R = R1 ∪ E ⋈ ΔR1
= R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ
so continue

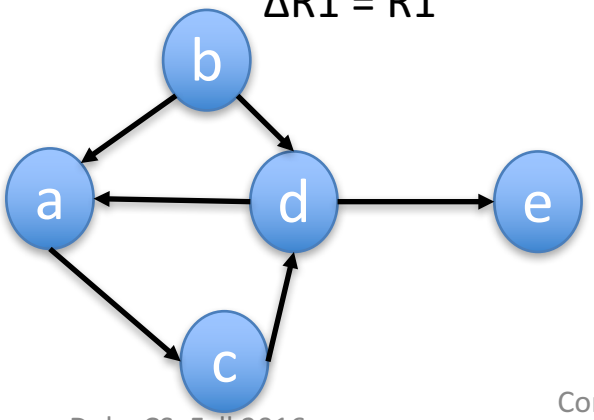
V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = R2 ∪ E ⋈ ΔR2
= R3 (say)

ΔR3 = R3 – R2

ΔR3 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d



Semi-Naïve evaluation - 4

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

Iteration 1:
R = E = R1 (say)
ΔR1 = R1

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 2:
R = R1 ∪ E ⋈ ΔR1
= R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = R2 ∪ E ⋈ ΔR2
= R3 (say)

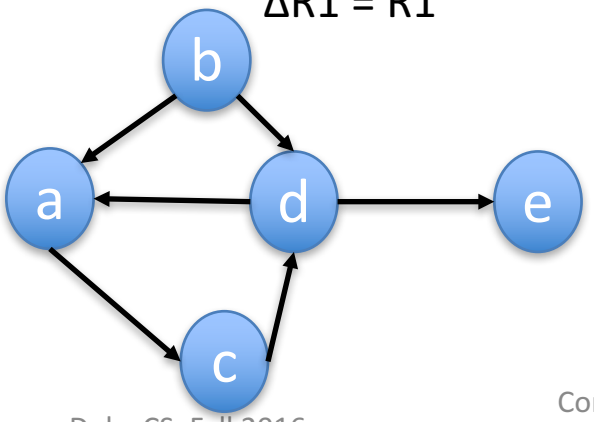
ΔR3 = R3 – R2

ΔR3 ≠ Φ
so continue

Iteration 4:
R = R3 ∪ E ⋈ ΔR3
= R4 (say)

ΔR4 = R4 – R3
ΔR = Φ
(CHECK ☺)
so STOP

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d



Incremental View Maintenance (IVM)

- Why did the semi-naïve algorithm work?
- Because of the generic technique of Incremental View Maintenance (IVM)

- Suppose you have
  - a database D = (R1, R2, R3)
  - a query Q that gives answer Q(D)
  - D = (R1, R2, R3) gets updated to D' = (R1', R2', R3')
  - e.g. R1' = R1 $\cup$ ΔR1 (insertion), R2' = R2 - ΔR1 (deletion) etc.

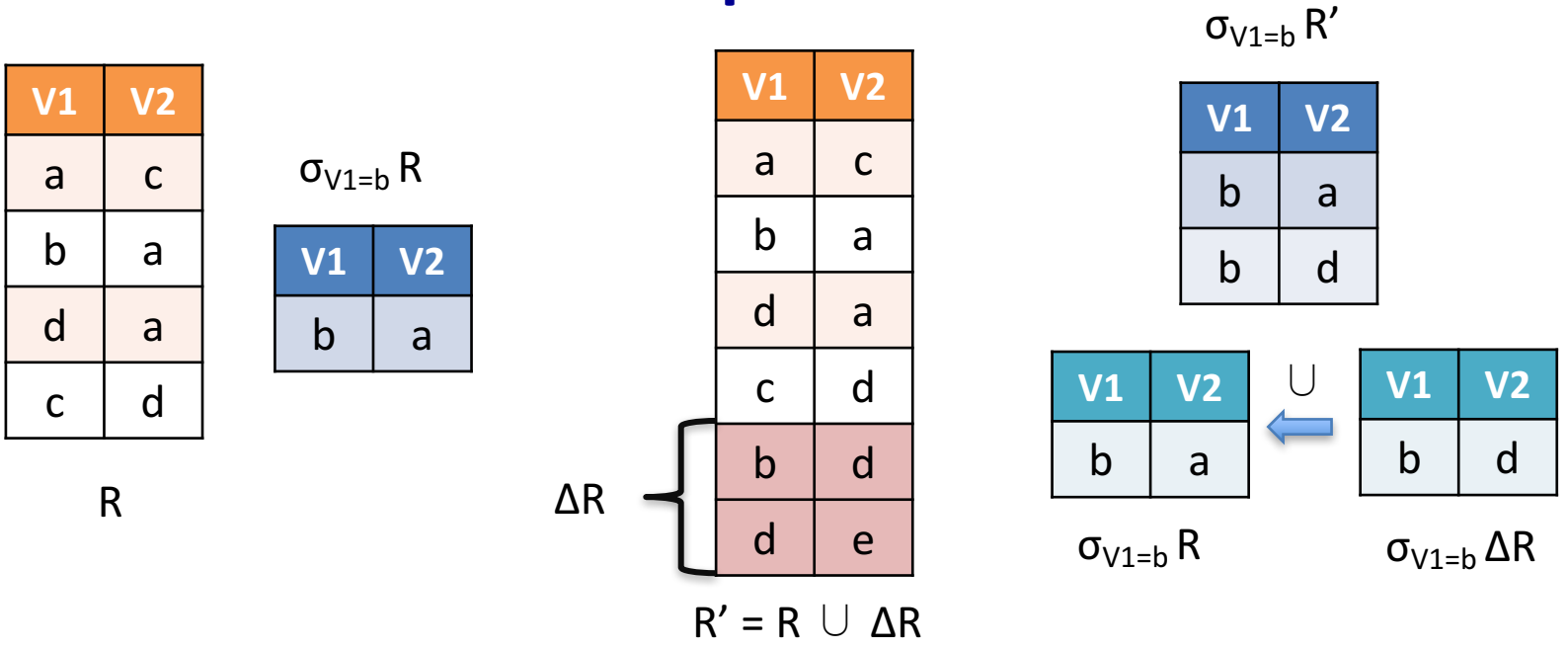
Incremental View Maintenance (IVM)

- Why did the semi-naïve algorithm work?
- Because of the generic technique of Incremental View Maintenance (IVM)

- Suppose you have
  - a database D = (R1, R2, R3)
  - a query Q that gives answer Q(D)
  - D = (R1, R2, R3) gets updated to D' = (R1', R2', R3')
  - e.g. R1' = R1 $\cup$ ΔR1 (insertion), R2' = R2 - ΔR1 (deletion) etc.

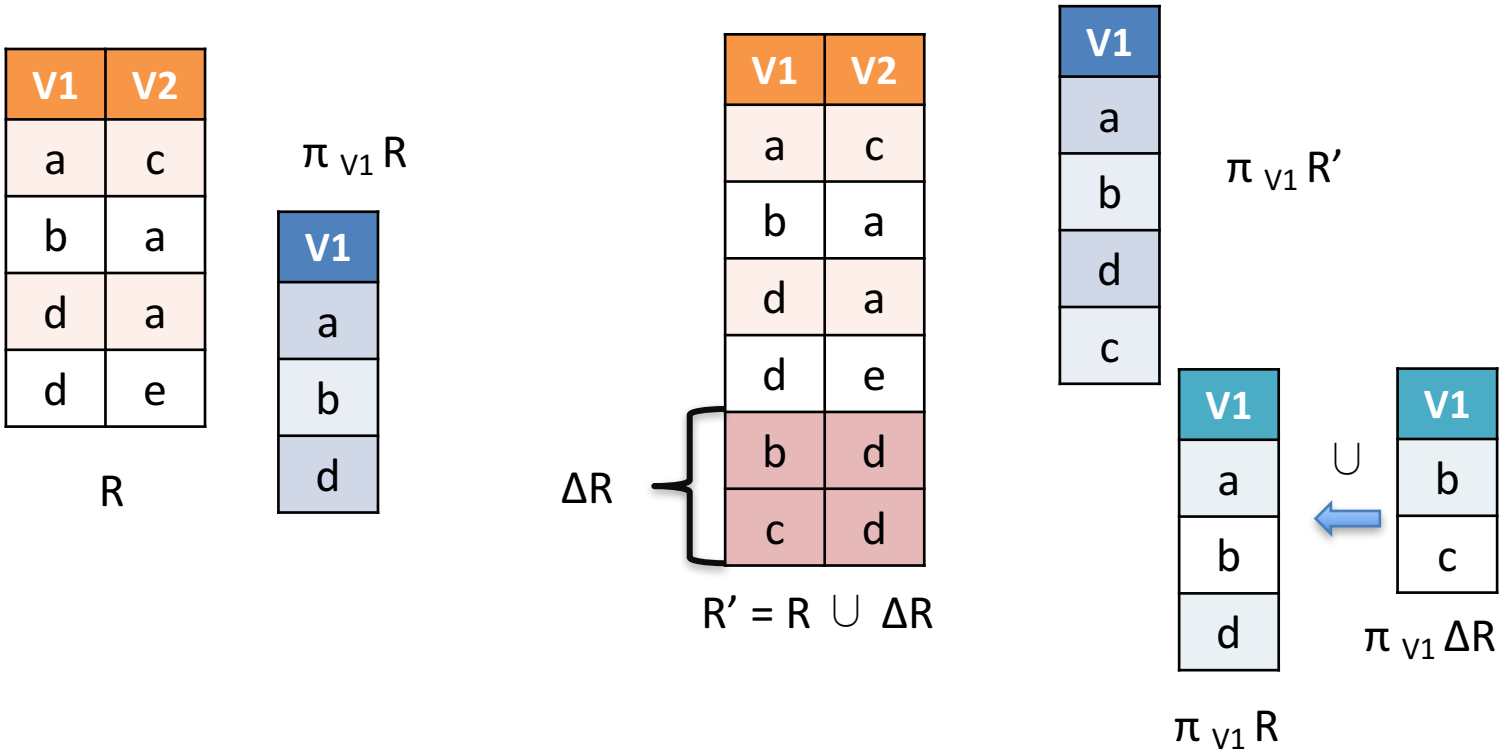
- IVM: Can you compute Q(D') using Q(D) and ΔR1, ΔR2, ΔR3 without computing it from scratch (i.e. do not rerun the query Q)?

IVM Example: Selection



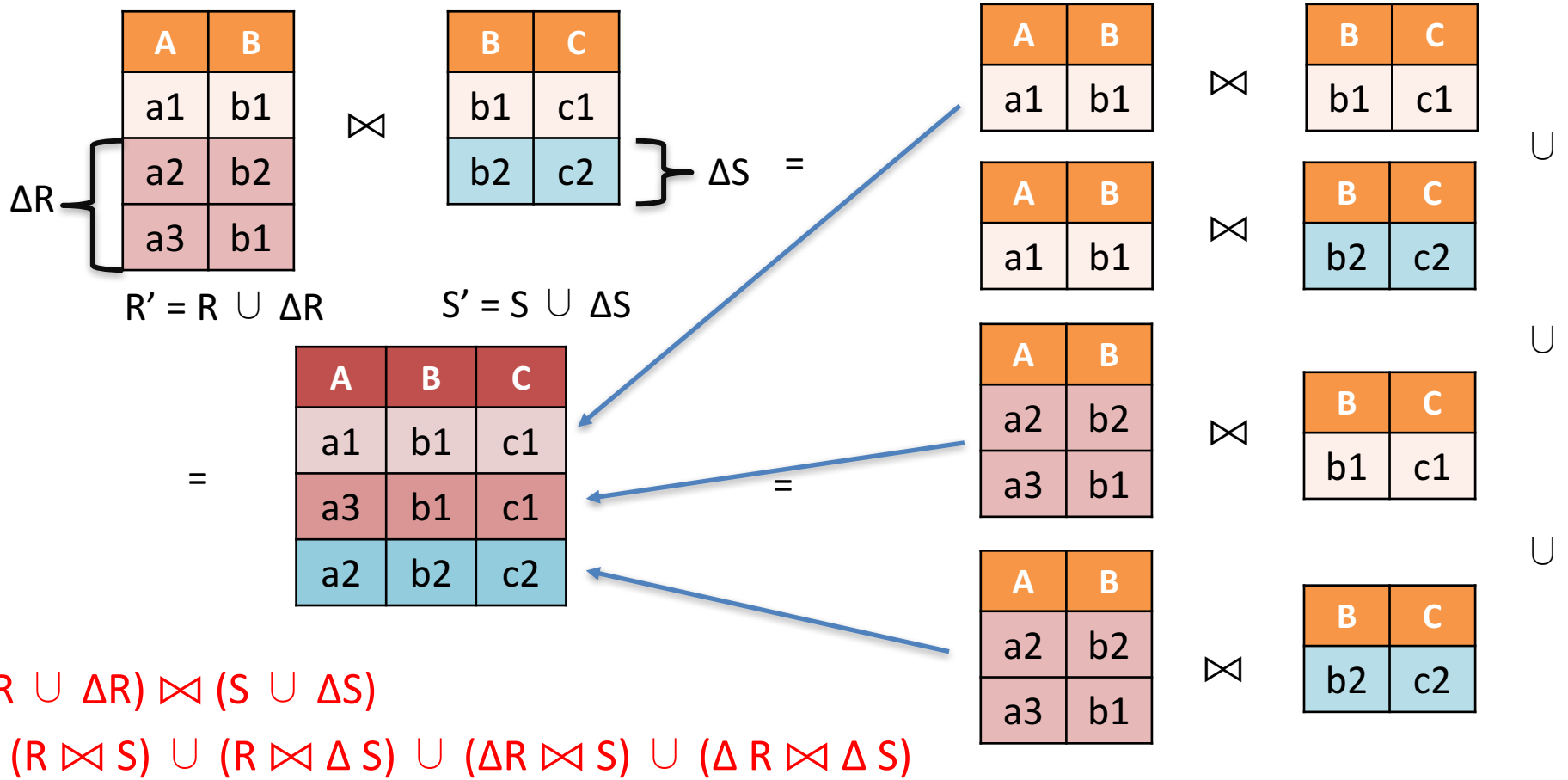
- $\sigma_{V1=b}(R \cup \Delta R) = \sigma_{V1=b} R \cup \sigma_{V1=b} \Delta R$
- It suffices to apply the selection condition only on ΔR
  - and include with the original solution

IVM Example: Projection



- $\pi_{V1}(R \cup \Delta R) = \pi_{V1} R \cup \pi_{V1} \Delta R$
- It suffices to apply the projection condition only on ΔR
  - and include with the original solution

IVM Example: Join



$(R \cup \Delta R) \bowtie (S \cup \Delta S)$

$= (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$

IVM for Linear Datalog Rule

A	B
a1	b1

⋈

B	C
b1	c1

=

A	B	C
a1	b1	c1

A	B
a1	b1
a2	b2
a3	b1

ΔR

R' = R ∪ ΔR

⋈

B	C
b1	c1

S' = S

=

A	B	C
a1	b1	c1
a3	b1	c1

- R(x, y) :- E(x, z), R(z, y)
  - i.e. $R_{new} = E \bowtie R$
- But E is EDB
  - $\Delta E = \Phi$
- Therefore,

$E \bowtie (R \cup \Delta R) = (E \bowtie R) \cup (E \bowtie \Delta R)$

- It suffices to join with the difference ΔR and include in the result in the previous round $E \bowtie R$
- Advantage of having "linear rule"

$(R \cup \Delta R) \bowtie (S \cup \Delta S)$

$= (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$

(Non-recursive) Datalog with Negation

- Recursion and negation together make Datalog execution complicated
  - there is a notion called "stratified semantic" for this purpose – compute IDB relations in strata/layers before taking a negation – not covered in this class

- We will only do negation for non-recursive Datalog

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Unsafe/Safe Datalog Rules

Find drinkers who like beer "BestBeer"

Q(x) :- Likes(x, "BestBeer")

Find drinkers who DO NOT like beer "BestBeer"

Q(x) :- ¬Likes(x, "BestBeer")

- What is the problem with this rule?
- What should this rule return?
  - names of all drinkers in the world?
  - names of all drinkers in the USA?
  - names of all drinkers in Durham?

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Problem with Negation in Datalog Rules

Find drinkers who like beer "BestBeer"

Q(x) :- Likes(x, "BestBeer")

Find drinkers who DO NOT like beer "BestBeer"

Q(x) :- ¬Likes(x, "BestBeer")

- What is the problem with this rule?
- Dependent on "domain" of drinkers
  - domain-dependent
  - infinite answers possible too..
    - keep generating "names"

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Problem with Negation in Datalog Rules

Find drinkers who like beer "BestBeer"

```
Q(x) :- Likes(x, "BestBeer")
```

Find drinkers who DO NOT like beer "BestBeer"

```
Q(x) :- ¬Likes(x, "BestBeer")
```

- Solution:
- Restrict to "active domain" of drinkers from the input *Likes* (or *Frequents*) relation
  - "domain-independence" – same finite answer always
- Becomes a "safe rule"

```
Q(x) :- Likes(x, y), ¬Likes(x, "BestBeer")
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (1/8)

Query: Find drinkers that like some beer (so much) that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (2/8)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

P => Q same as ¬P ∨ Q

$$\equiv \quad Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(\neg\ Serves(z,y)\ \vee\ Frequents(x,z))$$

Step 1: Replace ∀ with ∃ using de Morgan's Laws

∀x P(x) same as ¬∃x ¬P(x)

$$Q(x) = \exists y.\ Likes(x, y) \wedge\ \neg\exists z.(Serves(z,y)\ \wedge\ \neg Frequents(x,z))$$

¬(¬P ∨ Q) same as P ∧ ¬ Q

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (3/8)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ Likes(x, y) \wedge \forall z.(Serves(z,y) \Rightarrow Frequents(x,z))$$

Step 1: Replace $\forall$ with $\exists$ using de Morgan's Laws

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\exists z.(Serves(z,y) \wedge \neg Frequents(x,z))$$

(new) Step 2: Make all subqueries domain independent

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg\exists z.(Likes(x,y) \wedge Serves(z,y) \wedge \neg Frequents(x,z))$$

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (4/8)

$$Q(x) = \exists y.\ Likes(x, y) \wedge \neg \exists z.(Likes(x,y) \wedge Serves(z,y) \wedge \neg Frequents(x,z))$$

$H(x,y)$

(new) Step 3: Create a datalog rule for some subexpressions of the form
$\exists x\ \exists y \ldots.\ R(\ldots.) \wedge\ S(\ldots.) \wedge\ T(\ldots.) \wedge \ldots.$

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (5/8)

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE ……
```

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (6/8)

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE … …)
```

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Revisit example from Lecture 4

RC → Datalog with negation → SQL (7/8)

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE L2.drinker=L.drinker and L2.beer=L.beer
         and L2.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L2.drinker
                           and F.bar=S.bar))
```

Ack: slides by Profs. Balazinska and Suciu

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

RC → Datalog with negation → SQL (8/8)

Revisit example from Lecture 4

```
H(x,y)   :- Likes(x,y),Serves(z,y), not Frequents(x,z)
Q(x)     :- Likes(x,y), not H(x,y)
```

Unsafe rule

Sometimes can simplify the SQL query by using an unsafe datalog rule
Correctness ensured by safe outermost rule

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Serves S
   WHERE L.beer=S.beer
         and not exists (SELECT * FROM Frequents F
                         WHERE F.drinker=L.drinker
                               and F.bar=S.bar))
```

Ack: slides by Profs. Balazinska and Suciu

Optional/additional slides

An Overview of Data Provenance with Annotations

Selected/adapted slides from the keynote by
Prof. Val Tannen, EDBT 2010
(optional material: full slide deck is available on Val's webpage)

optional slide

Lineage

- [Cui-Widom-Wiener'00]
- Lineage:
  - Given a data item in the view
  - Determine the source data that produced it
  - The process by which it was produced

optional slide

Applications

- OLAP/OLAM (mining)
  - origin of anomalous data to verify reliability
- Scientific Databases
  - how answer was produced from raw data
- Online Network monitoring and Diagnosis system
  - identify faulty sensor from network monitors

optional slide

Applications

- Cleansed data feedback
  - Clean raw data and send report to sources
- Materialized view schema evolution
  - if view schema is changed (new column added), recomputation may not be necessary
- View update
  - translate view updates to base data updates

optional slide

Data Provenance

provenance, n.

The fact of coming from some particular source or quarter; origin, derivation [Oxford English Dictionary]

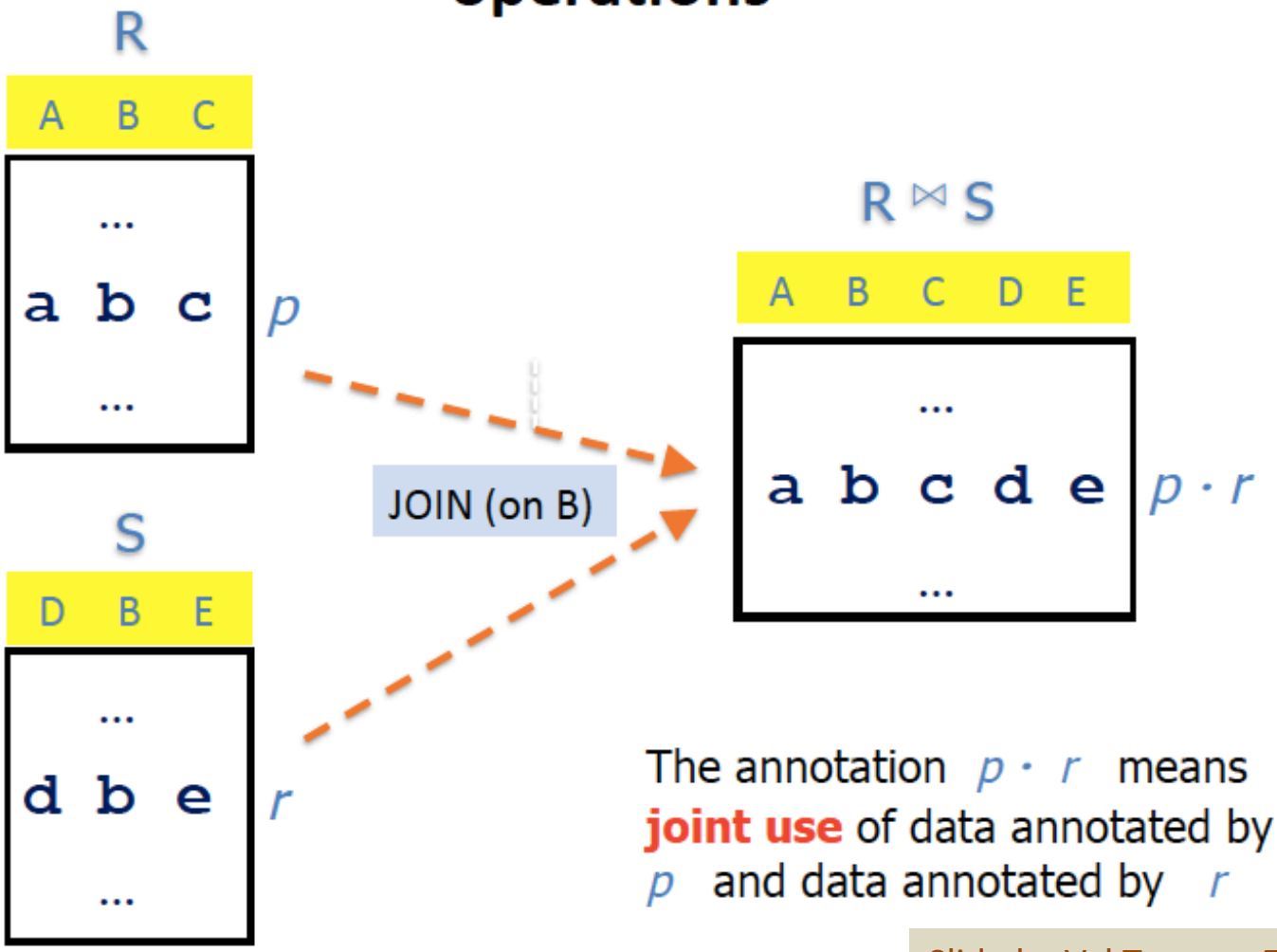
- **Data provenance** [BunemanKhannaTan 01]**:** aims to explain how a particular result (in an experiment, simulation, query, workflow, etc.) was derived.

- Most science today is **data-intensive**. Scientists, eg., biologists, astronomers, worry about data provenance all the time.

Slide by Val Tannen, EDBT 2010

optional slide

Propagating annotations through database operations

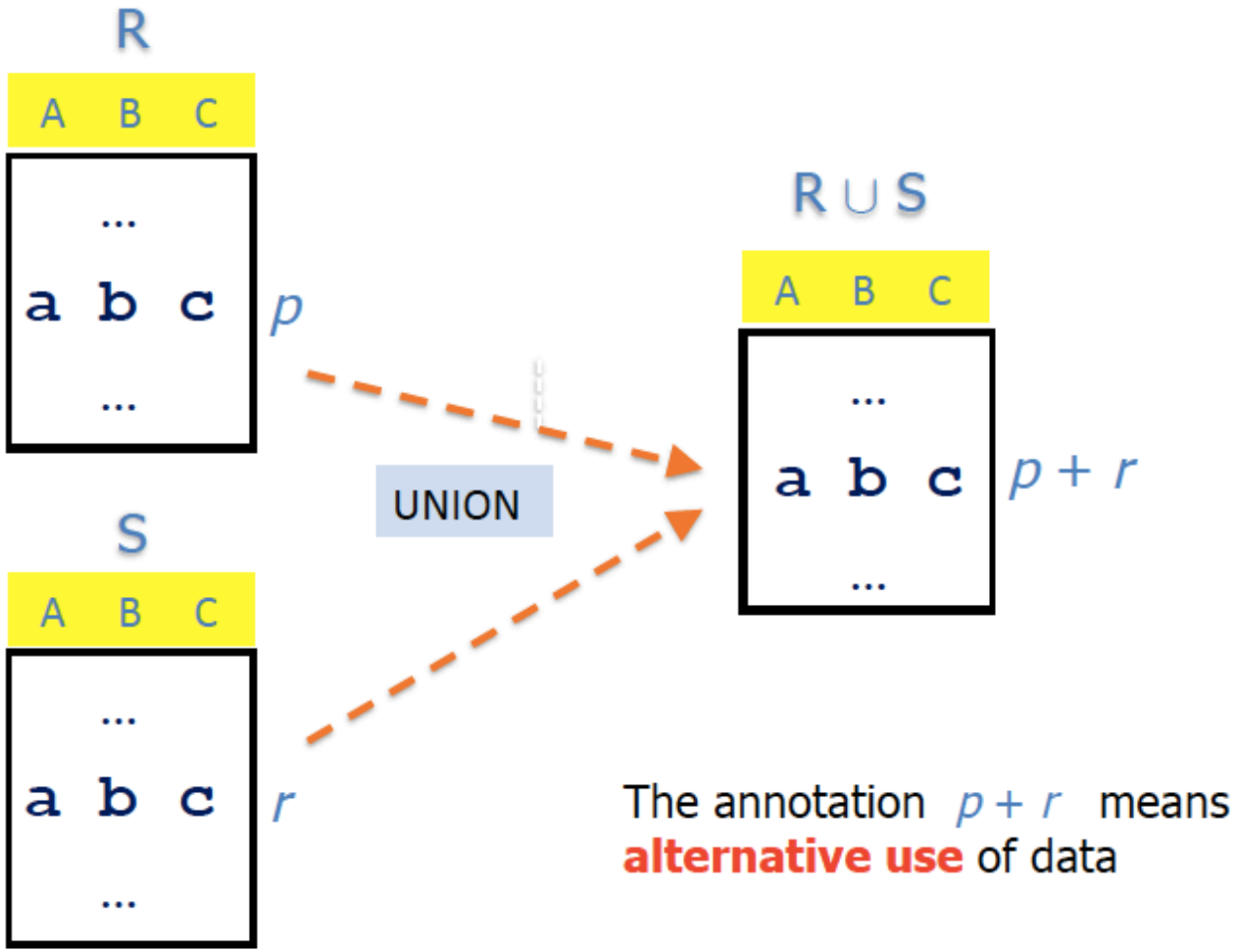


The annotation $p \cdot r$ means **joint use** of data annotated by p and data annotated by r

Slide by Val Tannen, EDBT 2010

optional slide

Another way to propagate annotations



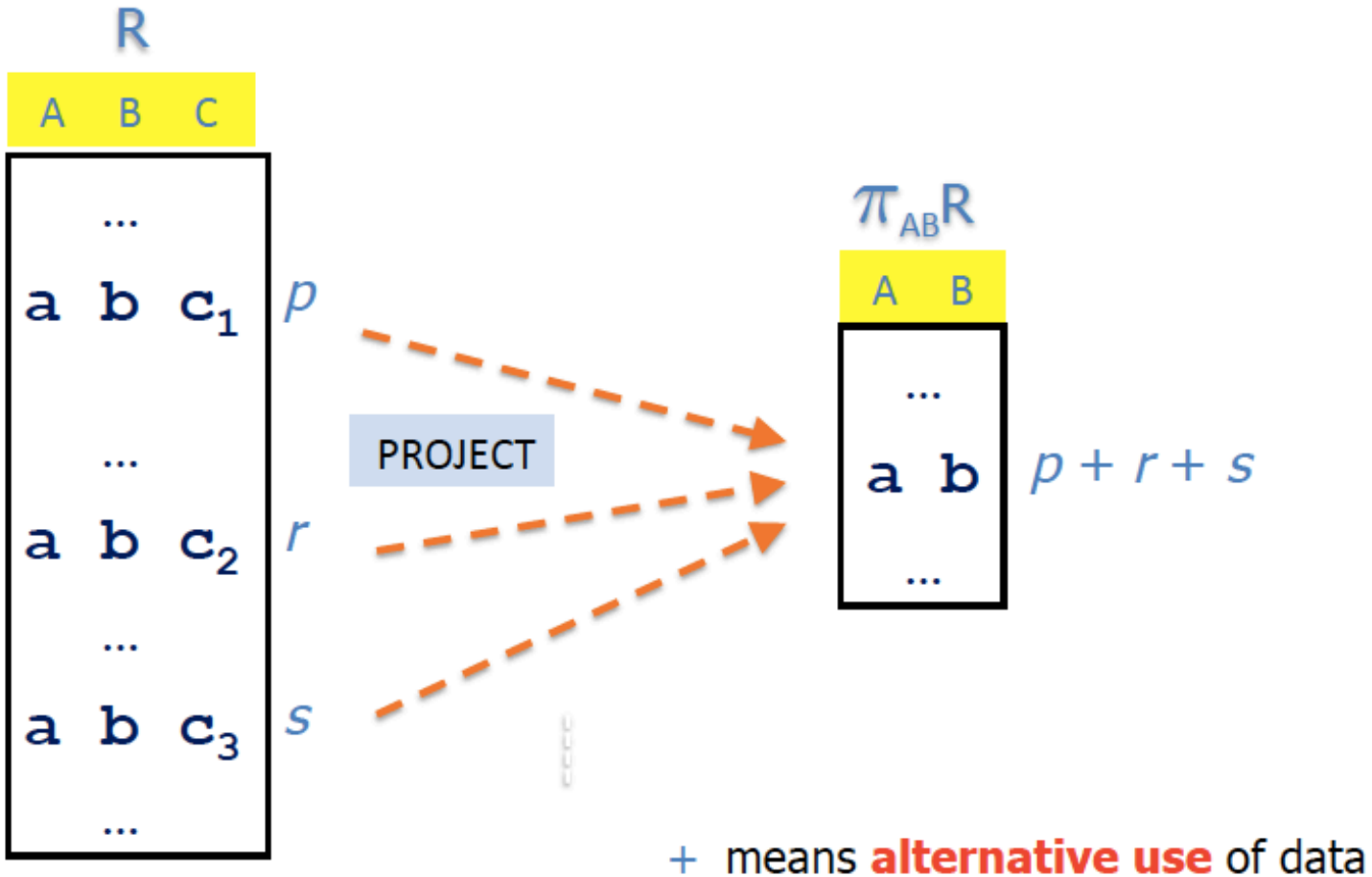
The annotation $p + r$ means **alternative use** of data

03/24/10 EDBT Keynote, Lausanne

Slide by Val Tannen, EDBT 2010

optional slide

Another use of +



\+ means **alternative use** of data

Slide by Val Tannen, EDBT 2010

optional slide

An example in positive relational algebra (SPJU)

$$Q = \sigma_{C=e}\pi_{AC}(\pi_{AC}R \bowtie \pi_{BC}R \cup \pi_{AB}R \bowtie \pi_{BC}R)$$

R

A	B	C	
a	b	c	p
d	b	e	r
f	g	e	s

A	C	
a	c	$(p \cdot p + p \cdot p) \cdot 0$
a	e	$p \cdot r \cdot 1$
d	c	$r \cdot p \cdot 0$
d	e	$(r \cdot r + r \cdot s + r \cdot r) \cdot 1$
f	e	$(s \cdot s + s \cdot r + s \cdot s) \cdot 1$

For selection we multiply
with two special annotations, 0 and 1

03/24/10 EDBT Keynote, Lausanne

Slide by Val Tannen, EDBT 2010

optional slide

Provenance Example

Boolean query **Q() :- HasAsthma (x), Friend (x, y), Smoker(y)**

	HasAsthma
x_1	~~Ann~~
x_2	Bob

	Friend	
y_1	Ann	Joe
y_2	Ann	Tom
y_3	Bob	Tom

	Smoker
z_1	Joe
z_2	Tom

Provenance $F_{Q,D} = x_1y_1z_1 + x_1y_2z_2 + x_2y_3z_2$

- $x, y, z \in \{0, 1\}$
- 1 = present
- 0 = absent
- There are three alternative ways to derive the “true” answer

Application to "Deletion Propagation"

optional slide

Boolean query **Q:** $\exists x \exists y$ **HasAsthma (x)** $\wedge$ **Friend (x, y)** $\wedge$ **Smoker(y)**

	HasAsthma
x_1	~~Ann~~
x_2	Bob

	Friend	
y_1	Ann	Joe
y_2	Ann	Tom
y_3	Bob	Tom

	Smoker
z_1	Joe
z_2	Tom

Provenance $F_{Q,D} = x_1y_1z_1 + x_1y_2z_2 + x_2y_3z_2$ [Green et al. '07]

- $x, y, z \in \{0, 1\}$
- What happens if Ann is deleted?
  - Does the answer change to false from true?

Application to "Deletion Propagation"

optional slide

Boolean query **Q:** $\exists x \exists y$ **HasAsthma (x)** $\wedge$ **Friend (x, y)** $\wedge$ **Smoker(y)**

	HasAsthma
x_1	~~Ann~~
x_2	Bob

	Friend	
y_1	Ann	Joe
y_2	Ann	Tom
y_3	Bob	Tom

	Smoker
z_1	Joe
z_2	Tom

Provenance $F_{Q,D} = x_1y_1z_1 + x_1y_2z_2 + x_2y_3z_2$ [Green et al. '07]

- $x, y, z \in \{0, 1\}$

0

- What happens if Ann is deleted?
  - Does the answer change to false from true?
- No need to re-evaluate the query
  - just plug in x1 = 0 and evaluate