# Informationsintegration

## Optimierung mit Semi-Joins

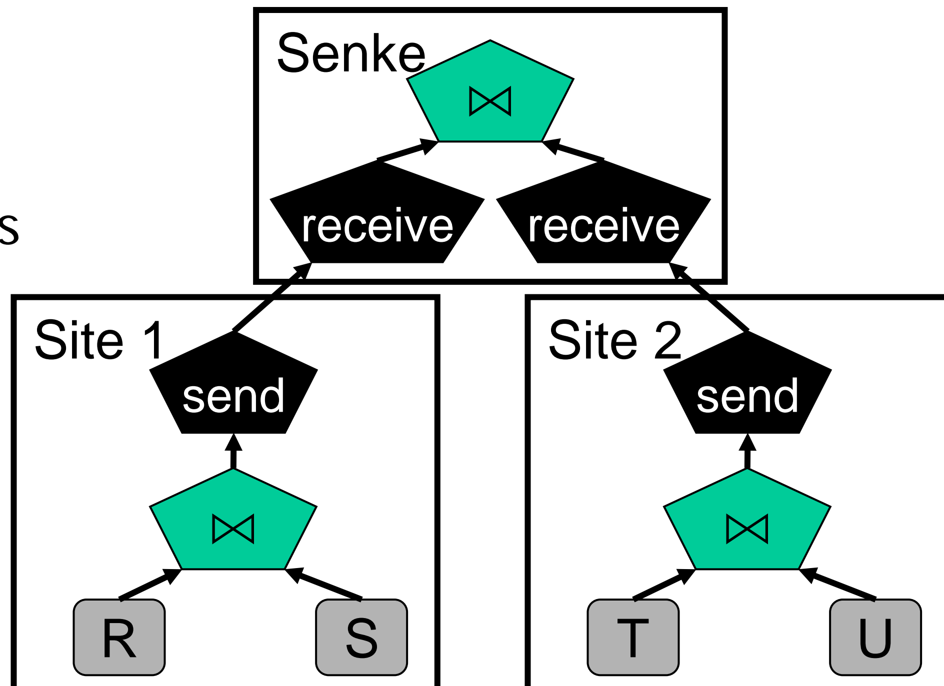
Ulf Leser

# Inhalt dieser Vorlesung

- Semi-Joins
- Bloomfilter: Semi-Join Optimierung
- Semi-Joins mit mehreren Relationen: Full Reducer

# Erinnerung: Auswertungsstrategien

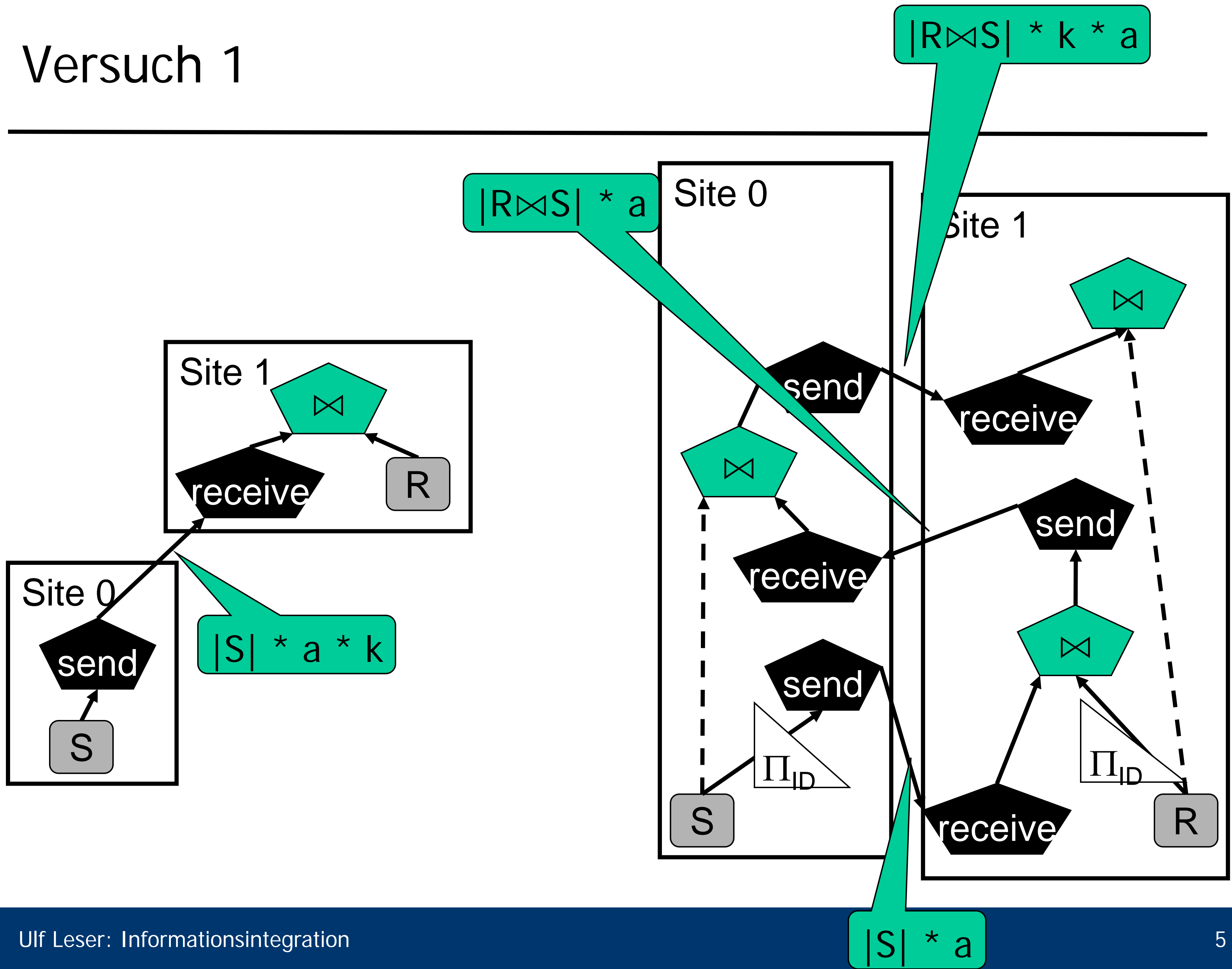
- Ship whole
  - Vollständige Relationen
  - Wenig Nachrichten, viele Byte
- Fetch rows as needed
  - Pushen von Selektionen/Joins
  - Bindings für Variable
  - Viele Nachrichten, wenig Bytes
- Fetch columns as needed
  - Semi-Join



# Ausgangslage

- Wir betrachten einen einzelnen Join $R \bowtie S$
  - Jedes Attribut hat Größe a, R/S mit k Attributen, keine Projektionen
- Zunächst drei mögliche Strategien
  - Daten von S nach R; Join in R ausführen
  - Daten von R nach S; Join in S ausführen
  - Daten von R und S zu drittem Knoten bewegen, Join dort ausrechnen
- Welche Daten bewegen wir?
  - Zwei Arten von Attributen: Joinattribute, andere Attribute
- Semi-Join: Konzentriert sich erst mal auf die Join(attribute)

# Versuch 1

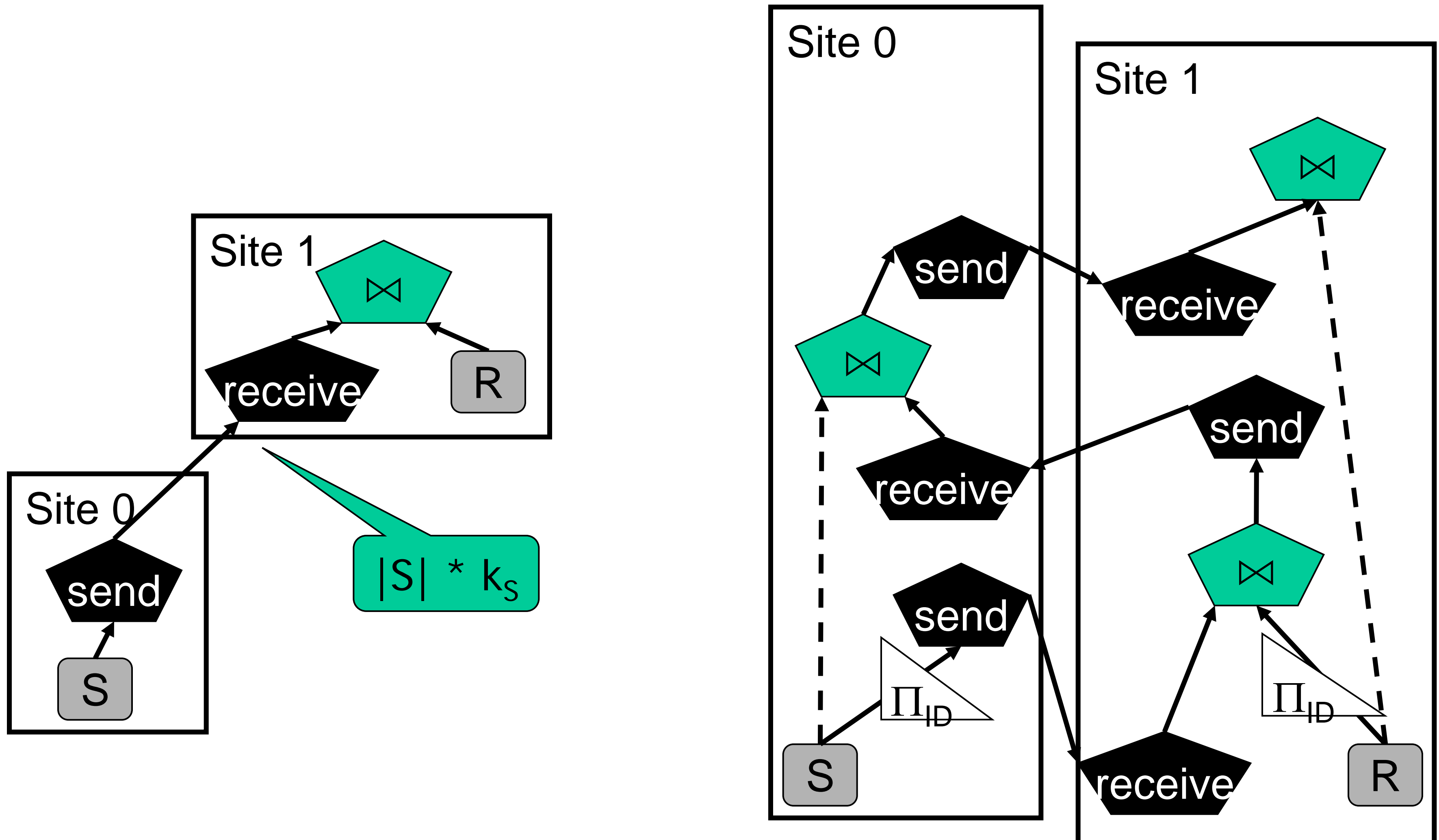


# Lohnt sich wann?

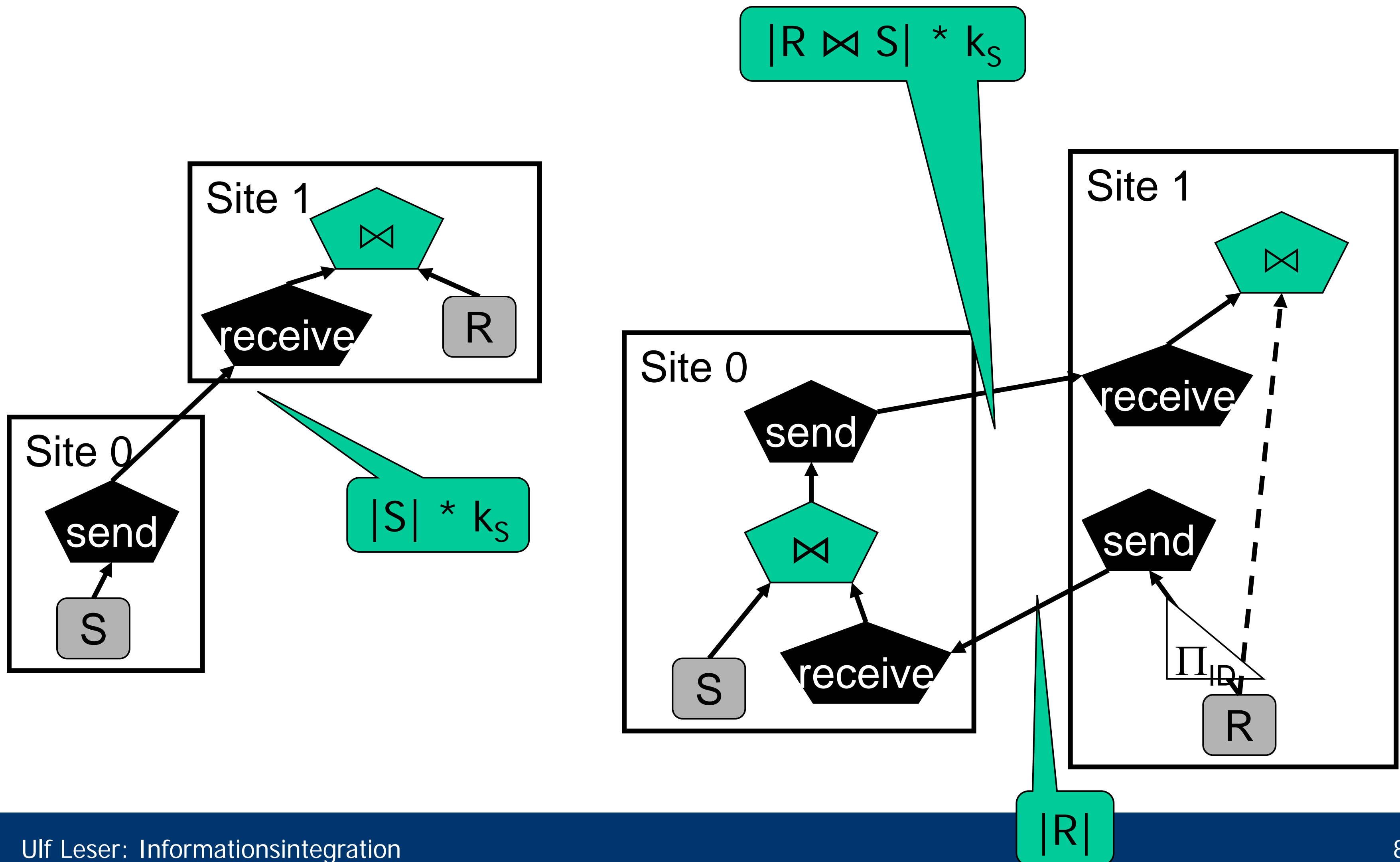
- Kosten Plan 1: $|S|*a*k$
- Kosten Plan 2: $|S|*a + |R \bowtie S|*a + |R \bowtie S|*a*k =$
  $|S|*a + |R \bowtie S|*a*(k+1)$

- Wann ist $|S|*a*k > |S|*a + |R \bowtie S|*a*(k+1)$
  - Unabhängig von Attributgröße: $|S|*k > |S|+|R \bowtie S|*(k+1)$
  - Wenn $|S| >> |R \bowtie S|$ ist
  - Wenn $|R \bowtie S|$ klein ist
  - Wenn k groß ist (viele Projektionen machen Semi-Join unattraktiv)
- Sprich: Wenn der Join hochselektiv ist und viele nicht-Joinattribute transportiert werden müssen

# Geht das nicht (vielleicht) besser?



# Semi-Join



# Lohnt sich wann?

- Kosten Plan 1: $|S|*k_S$
- Kosten Plan 2: $|S| + |R \bowtie S|*(k_S+1)$
- Kosten Plan 3: $|R| + |R \bowtie S|*k_S$

- 2 und 3 besser als 1, wenn kleines Joinergebnis und k groß
- 3 besser als 2, wenn $|R|<|S|$
  - Die kleinere Quelle initiiert und berechnet den Join
  - $|R \bowtie S|$ ist immer gleich groß, egal welche Quelle kleiner ist
  - Plan 4: $|S| + |R \bowtie S|*k_R$

# Weitere Variante

Site 1
⋈
receive
receive
Site 0
send
S
Site 2
send
R
|R| * k
|S| * k
|R ⋈ S| * k

Site 1
⋈
receive
receive
|R ⋈ S| * k
|R ⋈ S| * k
Site 0
send
⋉
receive
send
$\Pi_{ID}$
S
Site 2
send
|R ⋈ S|
send
$\Pi_{ID}$
⋊
receive
R
|S|

# Definition Semi-Join

- Definition

  *Gegeben Relationen R mit Attributmenge A und S mit Attributmenge B. Der Semi-Join $R \ltimes S$ ist definiert als*

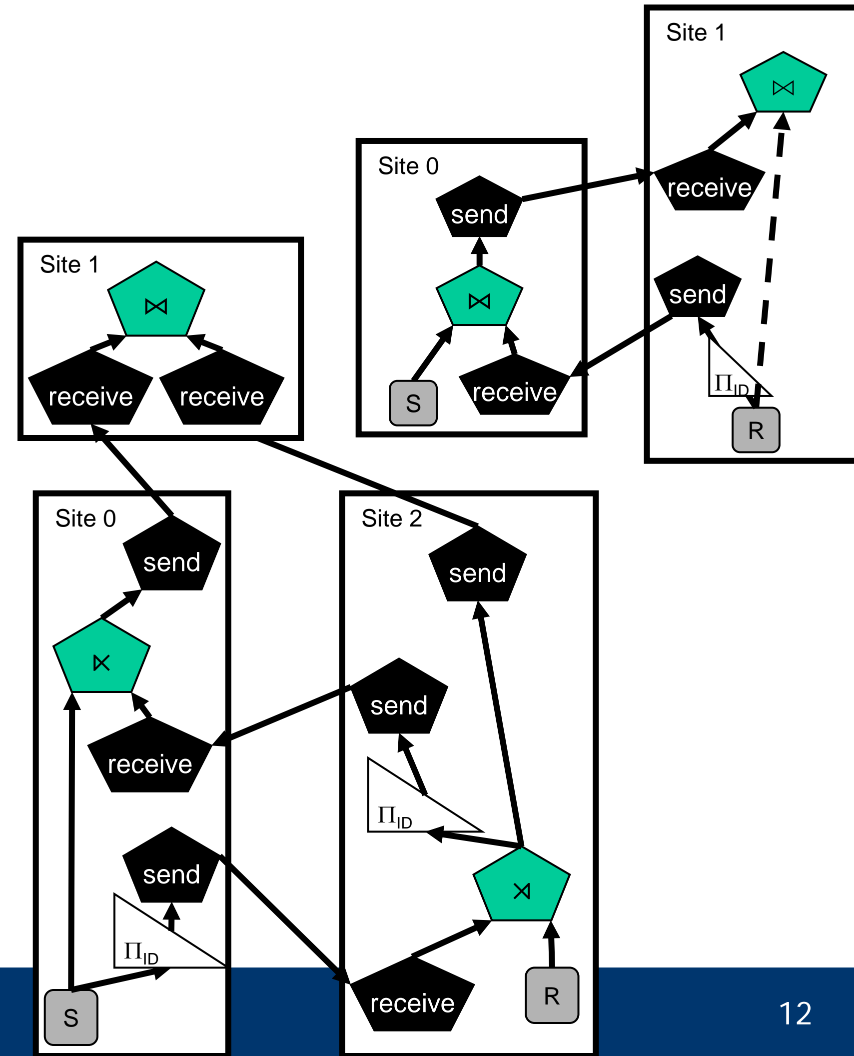
$$
\begin{aligned}
R \ltimes S := \; & \Pi_A(R \bowtie_{A \cap B} S) \\
= \; & \Pi_A(R) \bowtie_{A \cap B} \Pi_{A \cap B}(S) \\
= \; & R \bowtie_{A \cap B} \Pi_{A \cap B}(S)
\end{aligned}
$$

- Bemerkungen
  - Der Join sei ein Natural Join (über $A \cap B$)
  - Bei Join zwischen R.X und S.Y gilt: $R \ltimes S := R \bowtie_{X=Y} \Pi_Y(S)$
  - S wirkt als Filter auf den Tupeln von R
  - Semi-Join ist asymmetrisch

# Transformationsregeln

- Semi-Joins können auf verschiedene Arten zur Optimierung von Joins eingesetzt werden
- Äquivalenzumformungen

  $R \bowtie_F S =$
  - $(R \ltimes_F S) \bowtie_F S$
    - R verkleinern, dann Join mit S
  - $R \bowtie_F (S \ltimes_F R)$
    - S verkleinern, dann Join mit R
  - $(R \ltimes_F S) \bowtie_F (S \ltimes_F R)$
    - R und S verkleinern, dann Join
- Entspricht welchen verteilten Plänen?

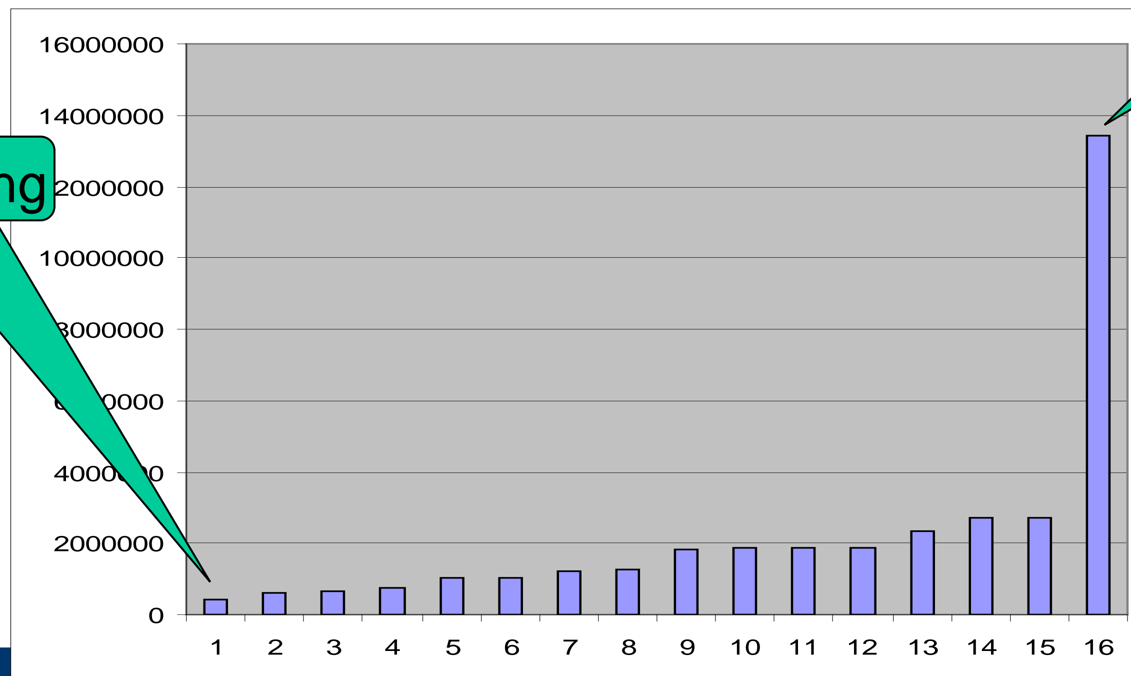


# Aus einer Übung

- Titel und Regisseur aller Filme, die jünger als 1980 sind

```
SELECT     F1.Titel, F2.Regie
FROM       Movie1.Filme1 F1, Movie2.Filme2 F2
WHERE      F1.Titel = F2.Titel
           AND F1.Jahr > 1980
```

- Anzahl übertragener Bytes



# Tricks

- Nur notwendige Bytes übertragen: Rtrim()
- Richtige Reihenfolge: Filme2 ist kleiner
- Projektionen wo immer möglich
- Kompression: Duplikate nicht übertragen
  - Semi-Join gemischt mit DISTINCT
- Bloomfilter

# Inhalt dieser Vorlesung

- Grundidee des Semi-Joins
- Bloomfilter: Semi-Join Optimierung
- Semi-Joins mit mehreren Relationen: Full Reducer

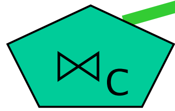
# Bloomfilter

- Effiziente Implementierung von (Semi-)Joins
- Beobachtung
  - Semi-Joins lohnen sich, wenn man die Join-Selektivität als sehr hoch einschätzt
  - Also werden nur sehr wenige Werte einen Joinpartner finden
  - Warum dann alle übertragen?
- Idee von Bloomfiltern (Hashfilter) für $R \ltimes_F S$
  - Bloom, B. H. (1970). "Space/Time Trade-offs in Hash Coding with Allowable Errors." Communications of the ACM
  - Hashe alle Werte R.F mit Hashfunktion h in (kleine) Hashtabelle H
  - Übertrage H nach S
  - $\forall$ $f \in S.F$ mit $H(h(f))=0$ gilt: f hat keinen Join-Partner in R
  - $\forall$ $f \in S.F$ mit $H(h(f))=1$ gilt: f hat vielleicht einen Join-Partner in R

# Beispiel

| $R \bowtie S$ | | | | |
|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ |
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ |
| $a_3$ | $b_3$ | $c_1$ | $d_1$ | $e_1$ |
| $a_5$ | $b_5$ | $c_3$ | $d_2$ | $e_2$ |

15 Werte




12 Werte (4 Tupel, davon 2 falsch positiv)

**Hashfunktion: mod 6**

| $R$ | | |
|---|---|---|
| $A$ | $B$ | $C$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_1$ |
| $a_4$ | $b_4$ | $c_2$ |
| $a_5$ | $b_5$ | $c_3$ |
| $a_6$ | $b_6$ | $c_2$ |
| $a_7$ | $b_7$ | $c_6$ |

6 Bit

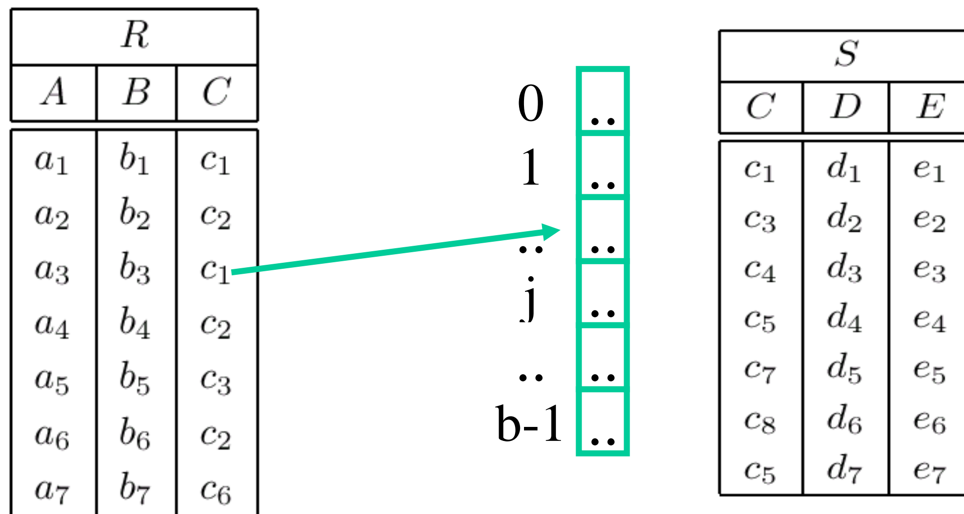
| $S$ | | |
|---|---|---|
| $C$ | $D$ | $E$ |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |
| ~~$c_4$~~ | ~~$d_3$~~ | ~~$e_3$~~ |
| ~~$c_5$~~ | ~~$d_4$~~ | ~~$e_4$~~ |
| $c_7$ | $d_5$ | $e_5$ |
| $c_8$ | $d_6$ | $e_6$ |
| ~~$c_5$~~ | ~~$d_7$~~ | ~~$e_7$~~ |

Falsch Positiv

Quelle: VL-Folien, Alfons Kemper, TU München

# Wsk für eine 1

- Wsk, dass ein bestimmtes Bit gesetzt ist
  - Sei b=|H|; wir nehmen Gleichverteilung von h an
  - Wsk, dass ein bestimmtes f∈R das Bit setzt: 1/b
  - Wsk, dass kein f∈R das Bit setzt: $(1-1/b)^{|R|}$
  - Wsk, dass irgendein f∈R ein bestimmtes Bit setzt: $1-(1-1/b)^{|R|}$

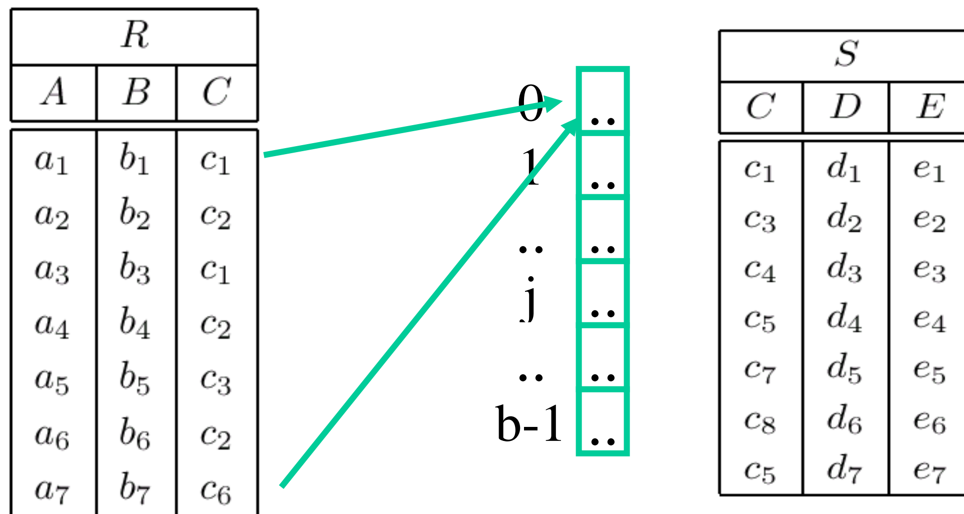



| R | | |
|---|---|---|
| $A$ | $B$ | $C$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_1$ |
| $a_4$ | $b_4$ | $c_2$ |
| $a_5$ | $b_5$ | $c_3$ |
| $a_6$ | $b_6$ | $c_2$ |
| $a_7$ | $b_7$ | $c_6$ |

| S | | |
|---|---|---|
| $C$ | $D$ | $E$ |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |
| $c_4$ | $d_3$ | $e_3$ |
| $c_5$ | $d_4$ | $e_4$ |
| $c_7$ | $d_5$ | $e_5$ |
| $c_8$ | $d_6$ | $e_6$ |
| $c_5$ | $d_7$ | $e_7$ |

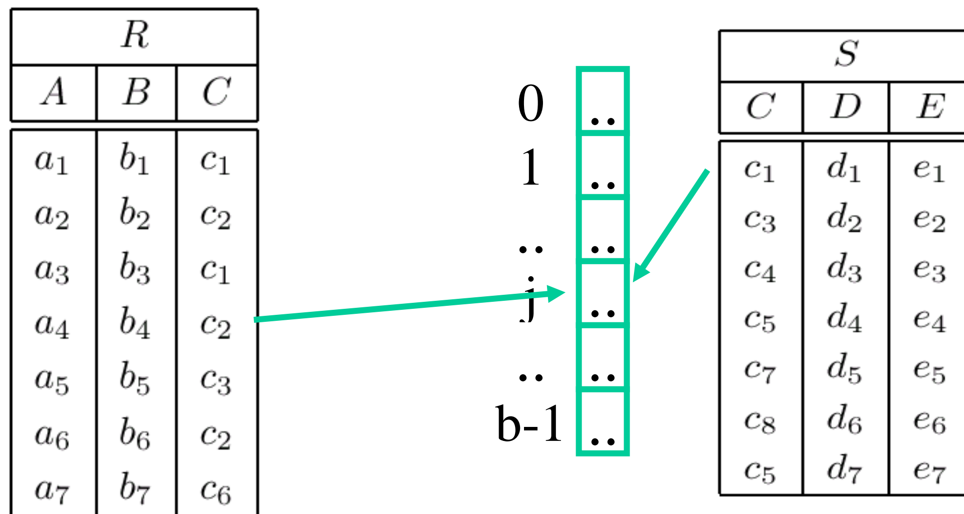
# Anzahl von 1'ern

- Wie viele Bits erwarten wir als gesetzt? $b*[1-(1-1/b)^{|R|}]$
  - Dabei beachten wir, dass mehrere f∈R dasselbe Bit setzen können
- Wenn b >> |R| ist das zu kompliziert
  - Dann nimmt man einfach an: Alle f∈R setzen unterschiedliche Bits
  - Nicht unrealistisch: Man braucht ja nur b bits
  - Damit: Wsk, dass ein bestimmtes Bit j gesetzt ist: |R|/b



# Auswirkung auf S

- Wsk, dass eine gegebene Position j im Hasharray irgendein $f \in S$ trifft: $1-(1-1/b)^{|R|}$
- Wie viele $f \in S$ werden ausgewählt? $|S|*(1- (1-1/b)^{|R|})$
- Alternative Berechnung (alle f setzen unterschiedliche Bits)
  - Wsk, dass ein bestimmtes Bit j gesetzt ist: |R|/b
  - |S|*(|R|/b) Elemente aus S werden ausgewählt



# Beispiel

- $|R|=|S|=1000$, $b=4000$, $a=15$
  - Übertragung aller Schlüssel: 15.000 Byte
  - Übertragung von H: 4000/8 = 500 Byte
- Erwartete Anzahl ausgewählter Werte in S
  - $|S| * [1- (1-1/b)^{|R|}] = 1000*[1- (1-1/4000)^{1000}] \sim 221$
  - Approximation: $|S|*(|R|/b) = 250$
- Nur in ca. 250 Bits erwartet man in beiden Arrays eine „1"
  - Wenn Werte durch Hashfunktion gleichverteilt werden und R und S unabhängig voneinander sind
- Wahrscheinlich sind nur wenige der „1" sind falsch Positive
  - Kann man genau ausrechnen

# Trade-Off

- Je größer b
  - Desto breiter wird R über H gestreut
  - Desto mehr Bit müssen im Filterschritt übertragen werden
  - Desto weniger Tupel aus S finden eine 1 in H
  - Desto weniger Tupel aus S finden fälschlicherweise eine 1 in H
  - Desto weniger Tupel müssen an R zurückgeschickt werden

# Bloom-Filter: Universeller Trick

- Signatur-Files (~Indexe für die Filterung von Daten)
- Beim „normalen“ Hash-Join
- Für Star-Joins in Data Warehouses
- Bloomfilter: Immer, wenn
  - … Mengen verglichen werden und
  - … man erwartet, dass nur wenige Elemente Treffer sind und
  - … Datenübertragung teuer ist

# Inhalt dieser Vorlesung

- Grundidee des Semi-Joins
- Bloomfilter: Semi-Join Optimierung
- Semi-Joins mit mehreren Relationen: Full Reducer

# Semi-Joins mit mehr als einem Join

$$R \bowtie_F S \bowtie_G T =$$
$$(R \ltimes_F S) \bowtie_F (S \ltimes_G T) \bowtie_G T =$$
$$(R \ltimes_F (S \ltimes_G T)) \bowtie_F (S \ltimes_G T) \bowtie_G T =$$

...

- Jeder Semi-Join reduziert (potentiell) die Zahl von Tupeln einer Relation, die übertragen werden müssen
  - Man nennt Semi-Joins daher auch „Reducer“
- Eine Relation heißt „reduced“, wenn sie keine Tupel mehr enthält, die nicht im Gesamtergebnis gebraucht werden
  - Globale Eigenschaft – auch weit entfernte Joins beeinflussen die „notwendigen“ Tupel einer Relation

# Formaler

- Definition

  *Seien $R_1,\ldots,R_n$ Relationen. Ein Semi-Join Programm ist eine Folge von Semi-Joins der Art*

  $$R_i := R_i \ltimes R_j$$

- Bemerkung
  - Joinattribute geben wir nicht mit an (ergeben sich aus Query)
  - Die Wirkung jedes Semi-Joins ist prinzipiell eine Reduzierung der Tupel in $R_i$
  - Gemeint ist nur eine temporäre Änderung von $R_i$
    - Es wird also eigentlich ein $R_i'$ produziert

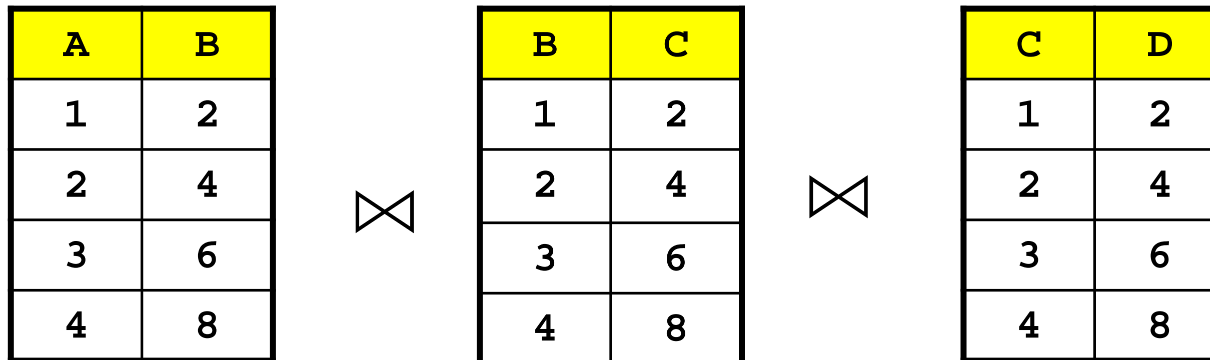
# Full Reducer

- Definition
  *Sei $Q=R_1\bowtie\ldots\bowtie R_n$ eine relationale Anfrage:*
  - *Ein Reducer für eine Relation $R_i$ in Q ist ein Semi-Join Programm, das aus $R_i$ alle Tupel entfernt, die nicht zur Berechnung von result(Q) benötigt werden*
  - *Ein Full Reducer für Q ist ein Semi-Join Programm, das ein Reducer für alle $R_i$ in Q ist*
- Bemerkung
  - Die $R_i$ müssen nicht unterschiedlich sein
  - Reducer für eine Relation – Full Reducer für eine Query

# Full Reducer und verteile Anfragen

- Ein Full Reducer entspricht einem Plan zur Abarbeitung einer verteilten Anfrage
- Der überträgt wenig (am wenigsten) Tupel für Joins, die „hoch“ im Ausführungsplan sind (also spät berechnet werden)
- Dafür werden Joins öfters ausgeführt – viel größerer Suchraum
- Ist nicht unbedingt optimal im Sinne der insgesamt kleinsten übertragenen Datenmenge
  - Denn zur Reduktion müssen ja Tupel übertragen werden
- Wie schwer ist es, einen Full Reducer zu finden?

# Beispiel



| A | B |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

⋈

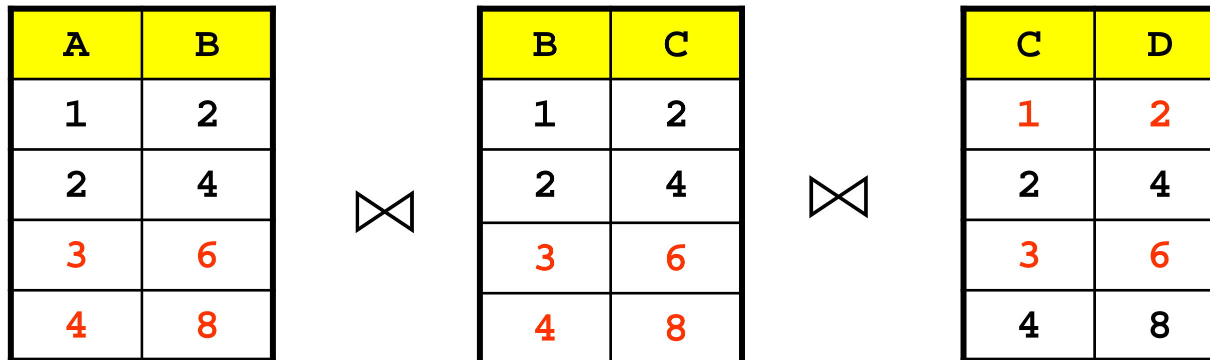
| B | C |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

⋈

| C | D |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

- Viele Tupel sind „dangling“
  - Z.B. 2-4, 4-8, 8-? oder ?-?, ?-1, 1-2
- Beispiel für ein Semi-Join Programm:
  - `AB := AB ⋉ BC` entfernt aus AB: (3,6) und (4,8)
  - `BC := BC ⋉ CD` entfernt aus BC: (3,6) und (4,8)
  - `CD := CD ⋉ BC` entfernt aus CD: (1,2) und (3,6)

# Beispiel



| A | B |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

$\bowtie$

| B | C |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

$\bowtie$

| C | D |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 6 |
| 4 | 8 |

- Ist das ein Full Reducer?
  - Nein: in AB ist (2,4) überflüssig; ebenso BC: (1,2) und CD: (2,4)
  - Um das zu sehen, muss man reduzierte Relationen zur weiter Reduktion verwenden
- Full Reducer?
  - **BC := BC $\ltimes$ AB** entfernt aus BC: (1,2) und (3,6)
  - **CD := CD $\ltimes$ BC** entfernt aus CD: (1,2) und (2,4) und (3,6)
  - **BC := BC $\ltimes$ CD** entfernt aus BC: (4,8)
  - **AB := AB $\ltimes$ BC** entfernt aus AB: (2,4) und (3,6) und (4,8)

# Etwas Theorie

- Die Schwere des Problems „Finde einen Full Reducer für eine gegebene Query Q" hängt von der Art der Query ab
- Definition
  *Sei $Q=R_1\bowtie\ldots\bowtie R_n$. Der Hypergraph von Q wird wie folgt konstruiert*
  - *Jedes Attribut in Q wird ein Knoten*
  - *Verschmelze (transitiv) alle Knoten, die über einen Equi-Join verbunden sind*
  - *Für jede $R_i$ füge eine Hyperkante ein, die alle Attribute von $R_i$ verbindet*
- Bemerkung
  - Wir zeigen Hyperkanten als Mengen

# Beispiel

- Schema
  - `Books( title, author, publisher, ISBN)`
  - `Publisher( publisher, paddr, pcity)`
  - `Borrower( name, baddr, bcity, ID)`
  - `Loan( ID, ISBN, date)`
- Query (berechnet was?)
  - 
```
SELECT   *
FROM     books, publisher, borrower, load
WHERE    books.publisher = publisher.publisher AND
         books.ISBN = loan.ISBN AND
         borrower.ID = loan.ID AND
         borrower.bcity = publisher.pcity
```
- Hypergraph …
  - Verschmelzung von Joinattributen ist durch die Namensgleichheit der Attribute schon fast erledigt

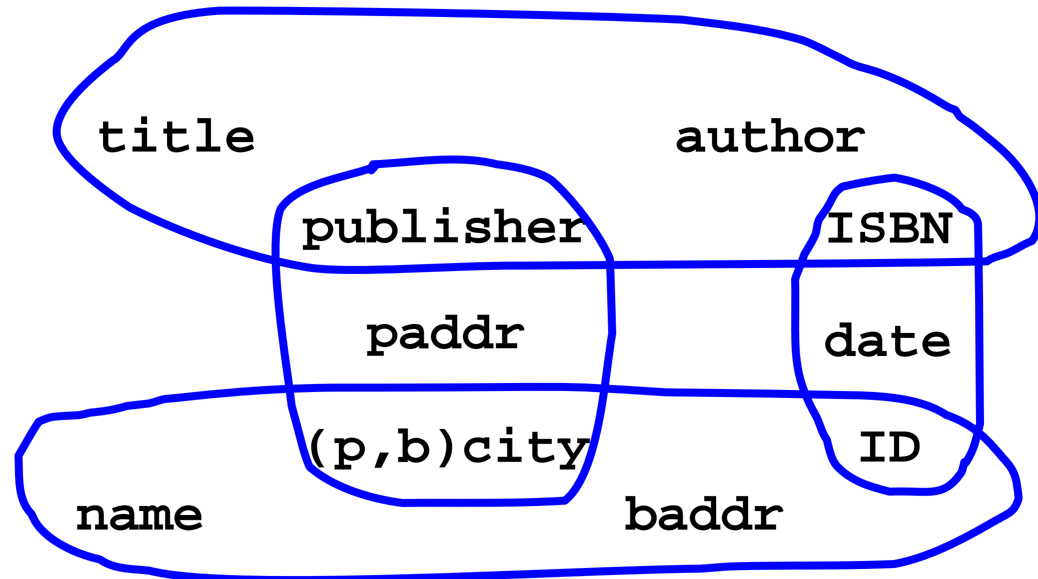
# Hypergraph

Title, publisher, author, ISBN

Publisher, paddr, pcity

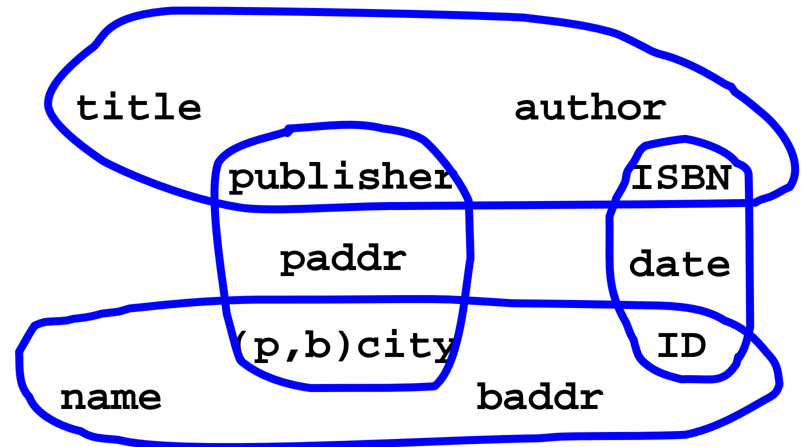
Name, baddr, bcity, ID

ID, ISBN, city



# GYO-Reduktion (Graham, Yu and Ozsoyoglu, 1979)

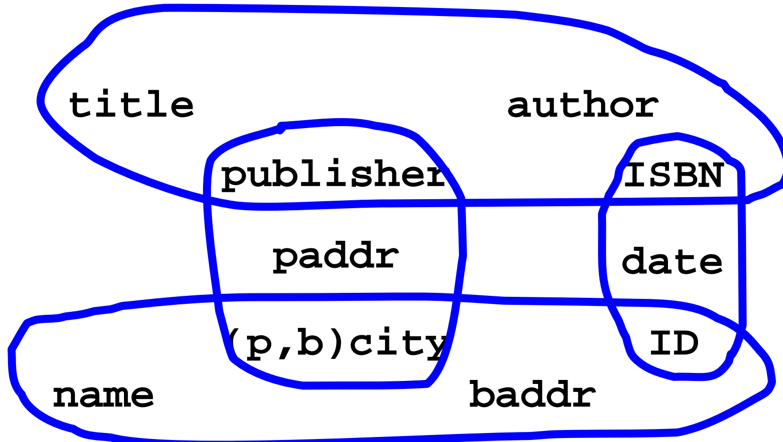
- GYO Reduktion eines Hypergraphen H
  - Sei E eine Hyperkante aus H. Wir nennen E ein Ohr, wenn
    - es kein Attribut mit einer anderen Hyperkante gemeinsam hat oder
    - es eine Hyperkante F gibt, so dass die Attribute in E-F in keiner anderen Hyperkante als E enthalten sind
- Ohren kann man abschneiden
  - Denn sie werden nur an einer Stelle gehalten
- Abschneiden heißt
  - Entfernen von E
  - Entfernen aller Knoten aus E-F
- Keine Ohren:

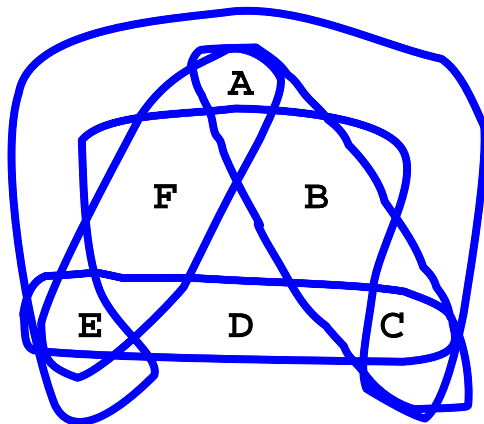


# Azyklische Hypergraphen

- Definition
  *Ein Hypergraph H ist azyklisch, wenn ein wiederholtes Entfernen aller Ohren so lange, bis es kein Ohr mehr gibt, den leeren Graphen erzeugt.*
- Bemerkung
  - Wir können uns beim Entfernen von Ohren nicht verlaufen – durch Abschneiden eines Ohrs werden keine anderen Abschneidungen verhindert (höchstens ermöglicht)
  - Das Ergebnis der GYO Reduktion ist eindeutig
    - Aber es gibt i.d.R. verschiedene Reihenfolgen dahin

# Beispiel

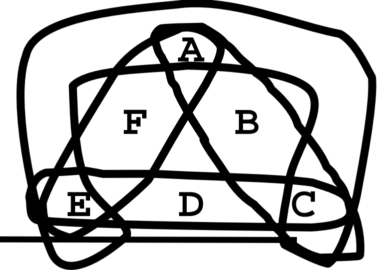


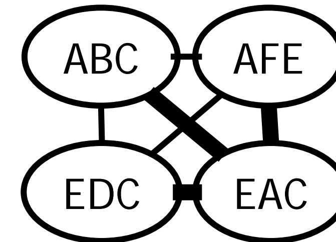

- Ist zyklisch

- Ist?
  - (ABC)-(ACE)=(B) und B ist nur in (ABC) – also (ABC) entfernen
  - (AEF)-(ACE)=(F) … also (AEF) entfernen
  - (EDC)-(ACE)=(D) … also (EDC) entfernen
  - Nun auch (ACE) entfernen

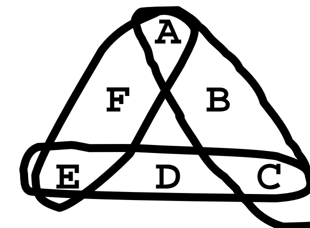
# Join-Graphen



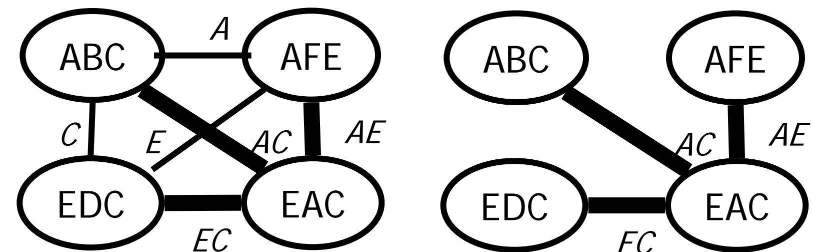

- Die Definition von azyklisch ist intuitiv nicht leicht zu erfassen
  - Der Join-Graph des letzten Beispiels ist zyklisch




  - Der azyklische Hypergraph hat einen zyklischen Subgraphen




  - Die Query kann so (äquivalent) umgeformt werden, dass der Join-Graph azyklisch ist




- Theorem
  *Eine Query Q ist azyklisch, wenn und nur wenn es eine äquivalente Umformung von Q gibt, deren Join-Graph azyklisch ist*

# Komplexität

- Theoreme
  - *Eine Query hat einen Full Reducer gdw. ihr Hypergraph azyklisch ist*
  - *Sei Q eine azyklische Anfrage und E eine Hyperkante ihres Hypergraphen. Dann gibt es eine Sequenz von Ohr-Entfernungen, die E als letztes Element entfernt*
    - *Wird noch wichtig – später*
  - *Für eine lineare Query ist das Finden eines Full Reducers linear*
    - *Eine Query ist linear, wenn man ihre Relationen so anordnen kann, dass jede Relation nur einen Join mit ihrem Vorgänger und einen Join mit ihrem Nachfolger hat*
- Beweise: Literatur

# Full Reducer für lineare Anfragen

- Wir benötigen zwei Phasen
  - Vorwärts
  - Rückwärts
- Ablauf
  - $R1 \bowtie_A R2 \bowtie_B \ldots \bowtie_Y R(n\text{-}1) \bowtie_Z Rn$
  - Vorwärts
    - $R2' = R2 \ltimes R1$
    - $R3' = R3 \ltimes R2' = R3 \ltimes (R2 \ltimes R1)$
    - …
    - $Rn' = Rn \ltimes R(n\text{-}1)' = \ldots$
  - Rückwärts
    - $R(n\text{-}1)'' = R(n\text{-}1)' \ltimes Rn'$
    - $R(n\text{-}2)'' = R(n\text{-}2)' \ltimes R(n\text{-}1)''$
    - …
    - $R1'' = R1 \ltimes R2''$

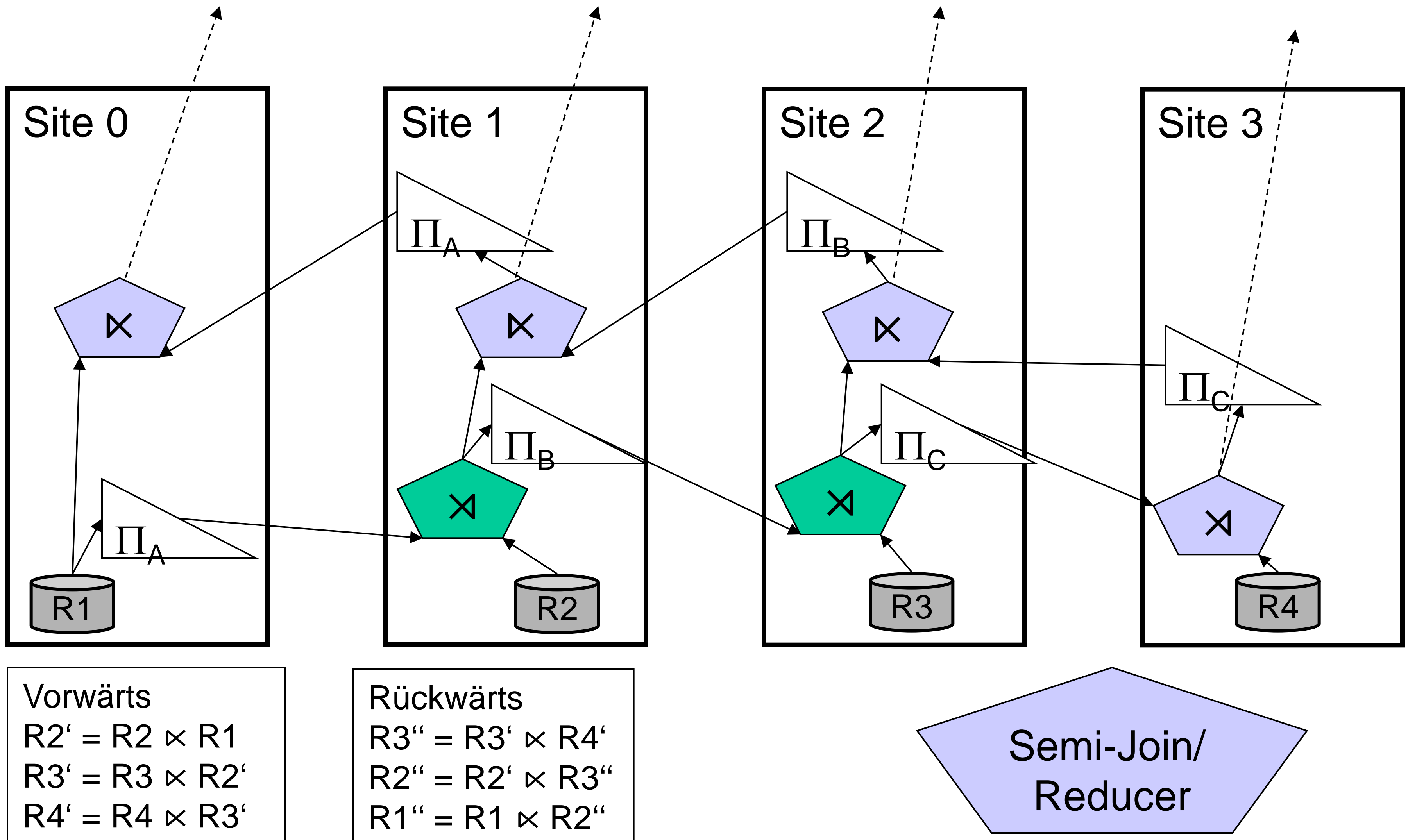
Reducer für Rn

Reducer für R(n-1)

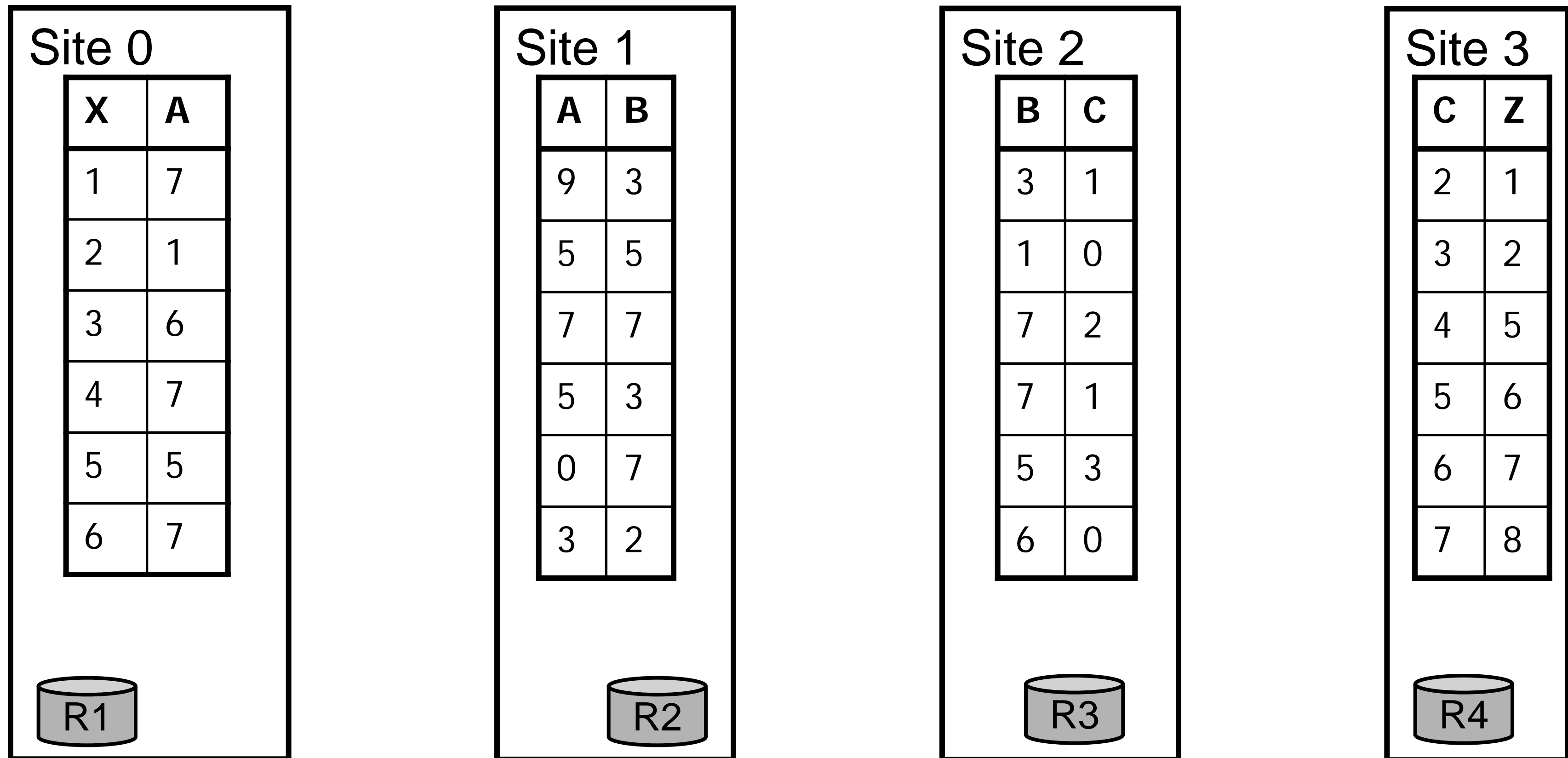
Reducer für R1

Full Reducer für Q

# Als Ausführungsplan



# Beispiel




Site 0:

| X | A |
|---|---|
| 1 | 7 |
| 2 | 1 |
| 3 | 6 |
| 4 | 7 |
| 5 | 5 |
| 6 | 7 |

Site 1:

| A | B |
|---|---|
| 9 | 3 |
| 5 | 5 |
| 7 | 7 |
| 5 | 3 |
| 0 | 7 |
| 3 | 2 |

Site 2:

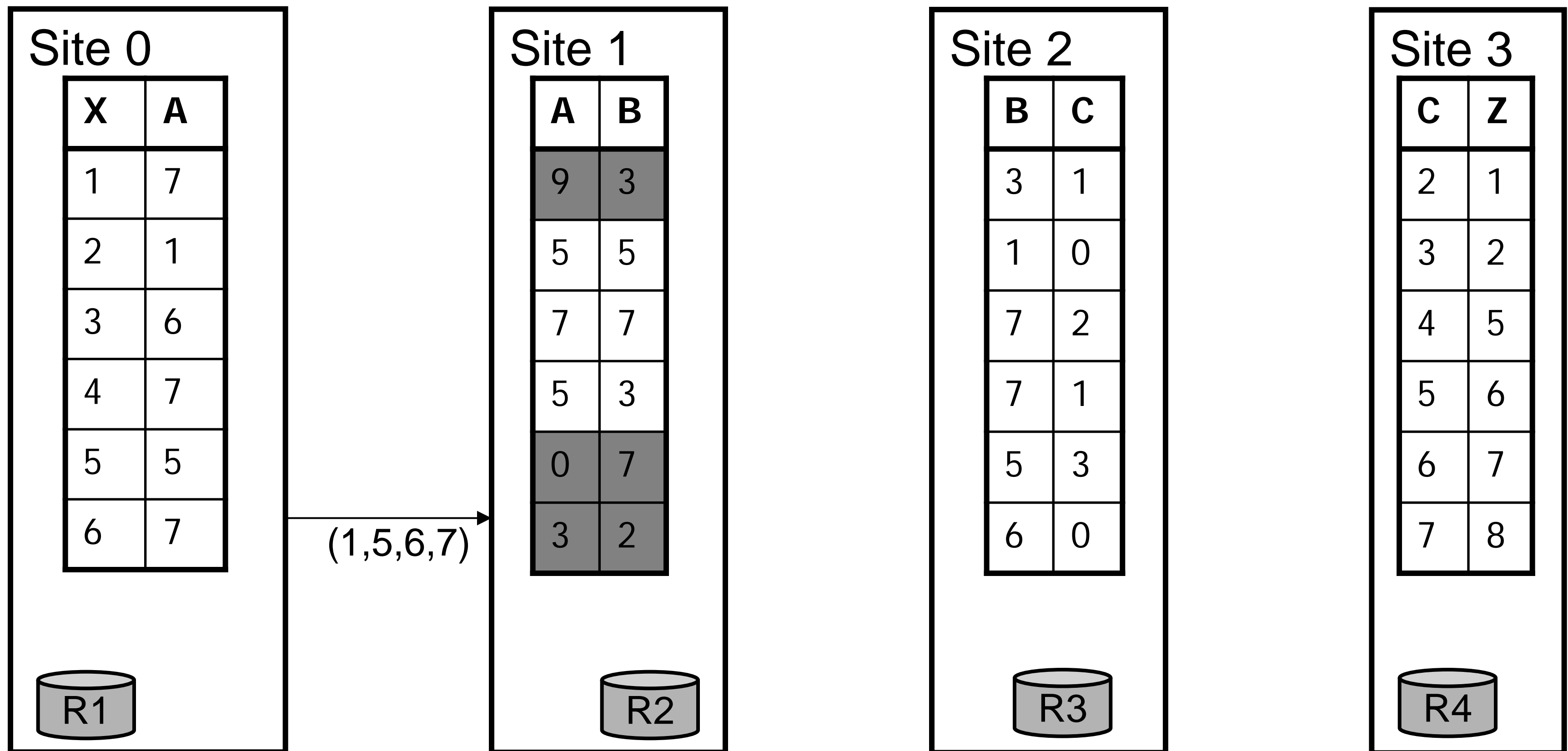
| B | C |
|---|---|
| 3 | 1 |
| 1 | 0 |
| 7 | 2 |
| 7 | 1 |
| 5 | 3 |
| 6 | 0 |

Site 3:

| C | Z |
|---|---|
| 2 | 1 |
| 3 | 2 |
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 8 |

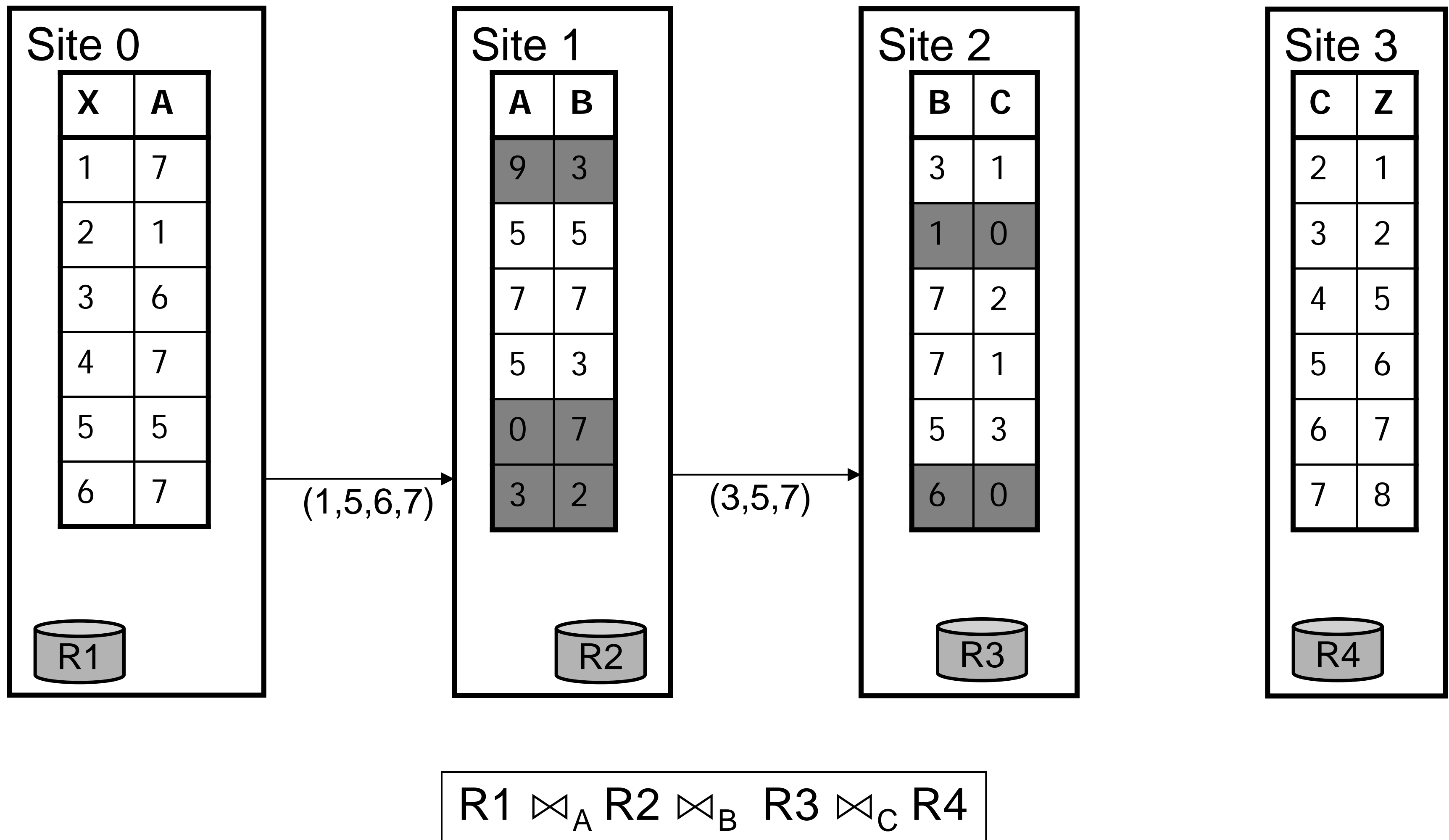
$$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$$

# Fully Reduce – Beispiel

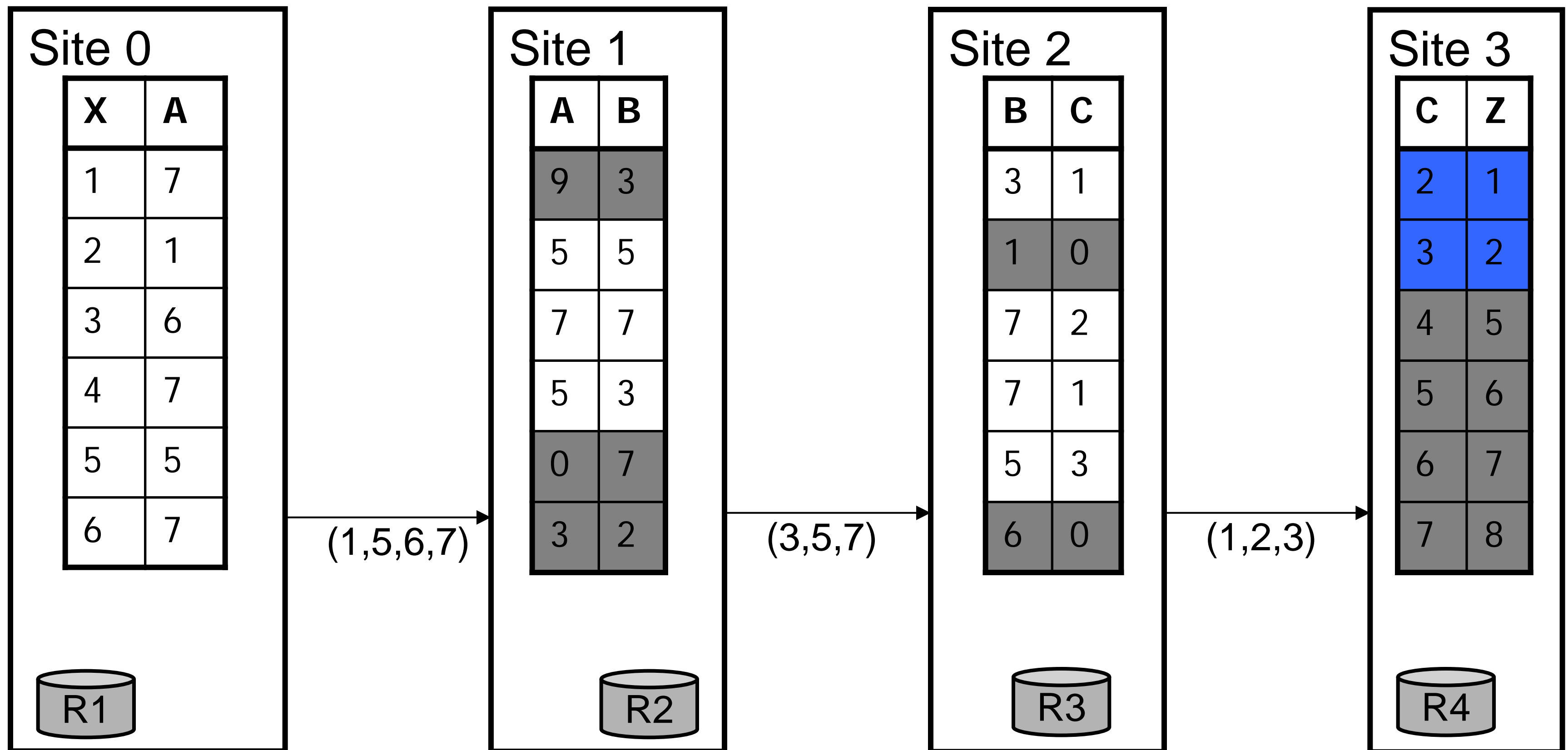



$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

# Fully Reduce – Beispiel

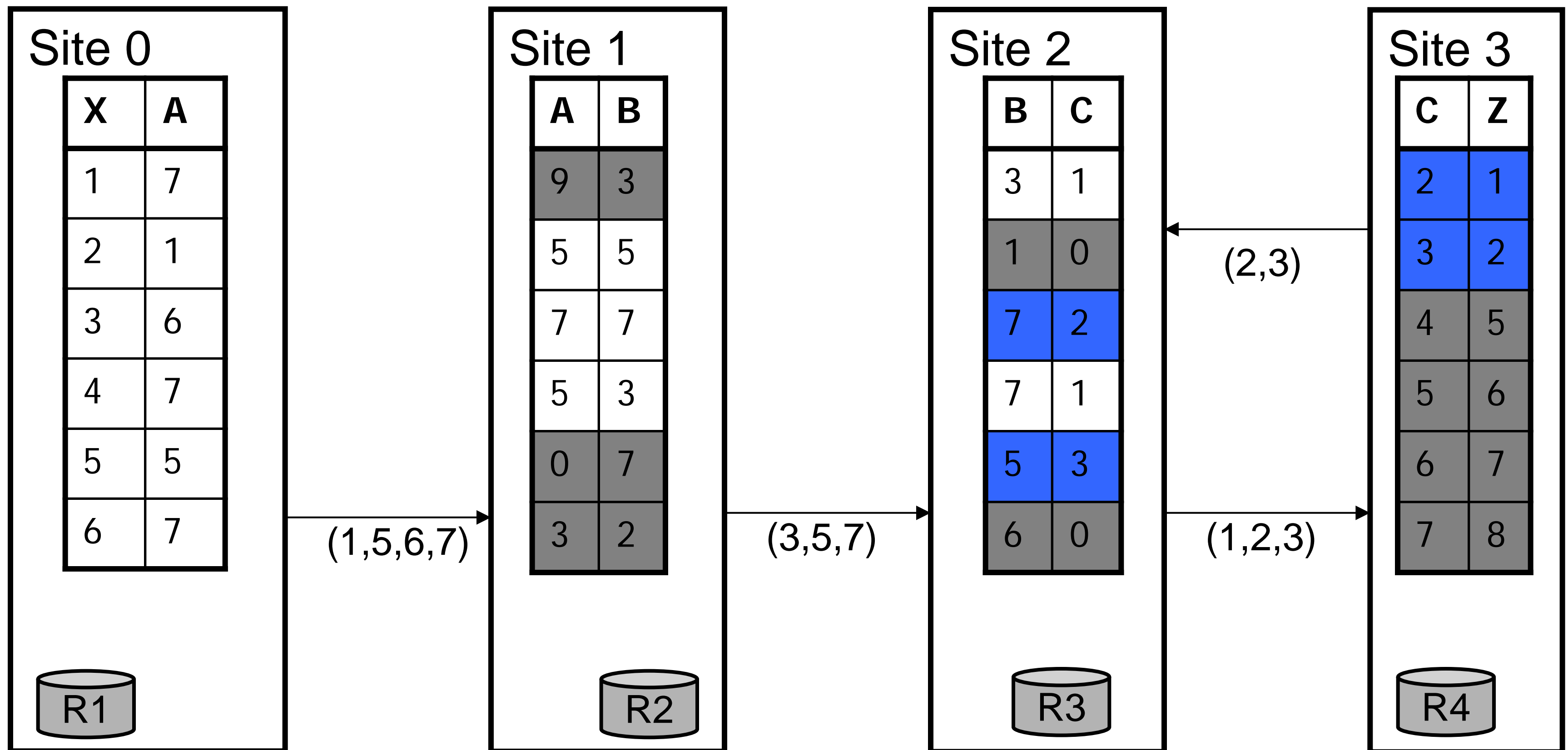


# Fully Reduce – Beispiel



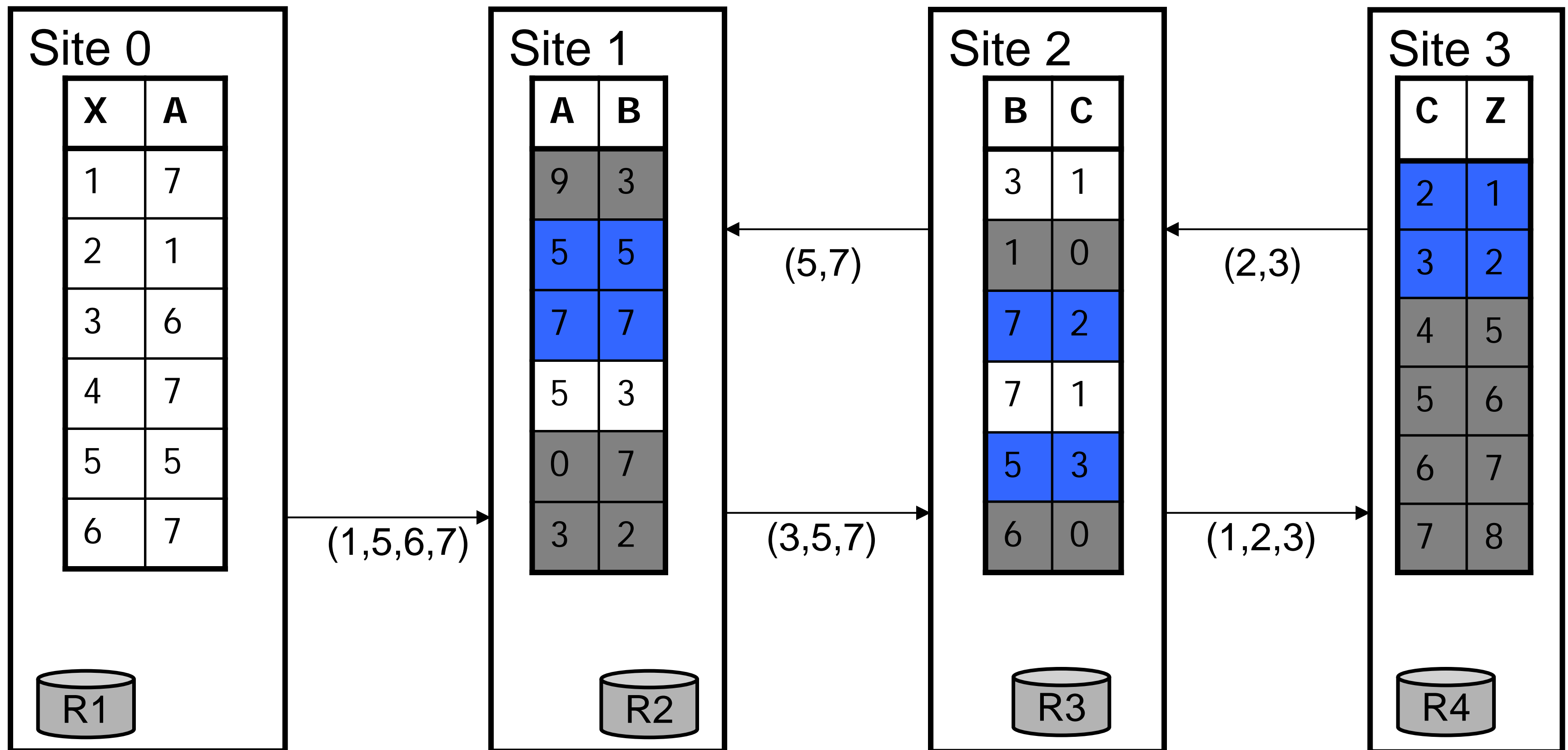

$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

# Fully Reduce – Beispiel



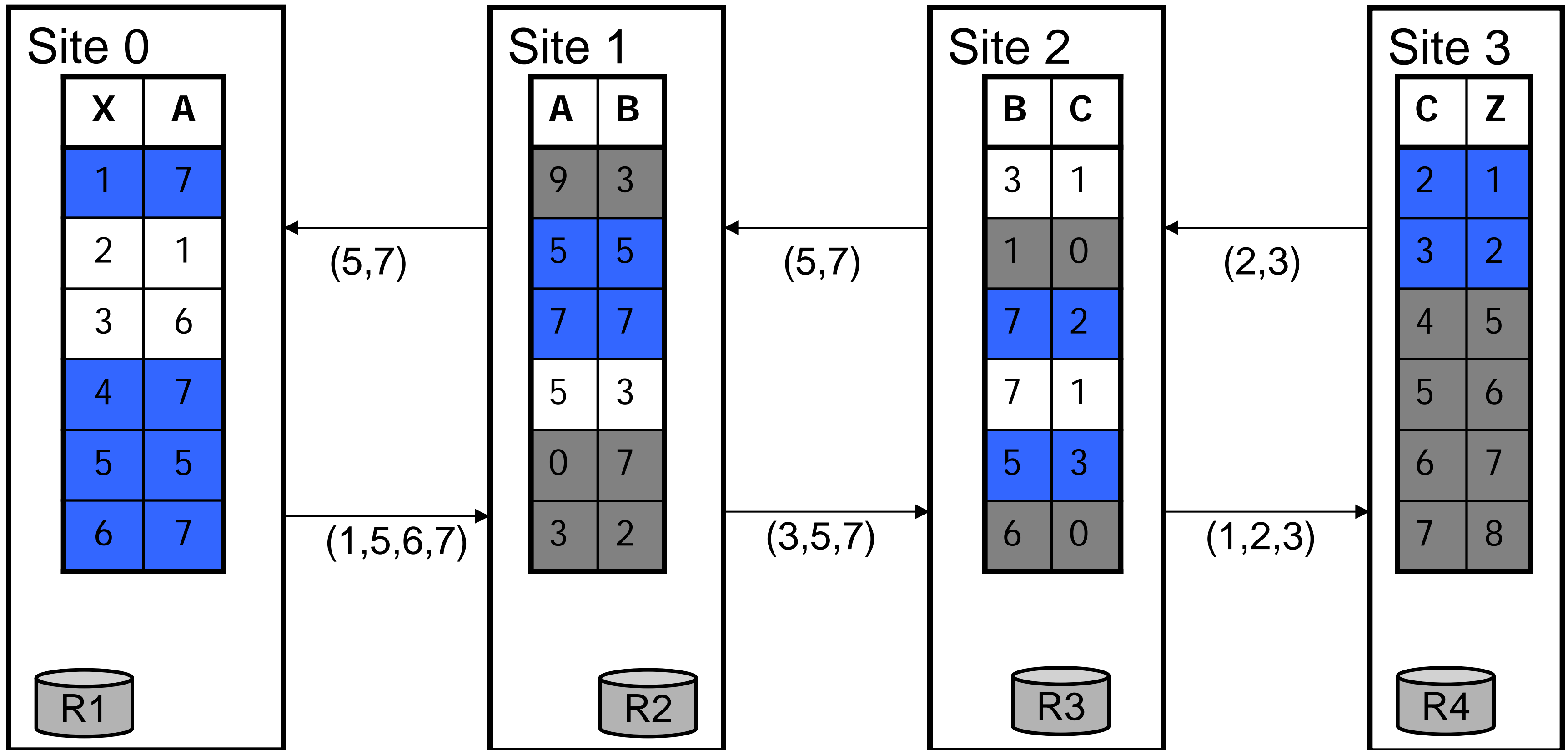

$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

# Fully Reduce – Beispiel




$$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$$

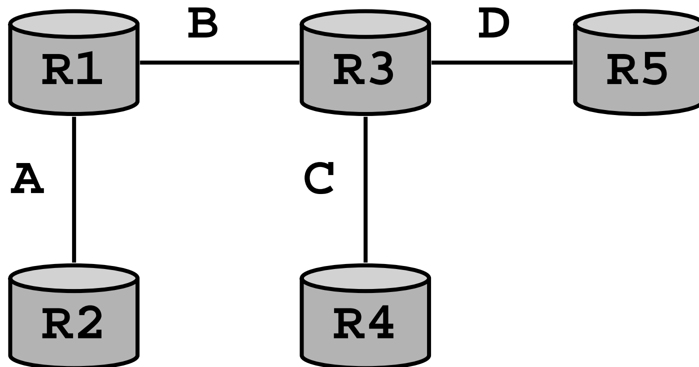
# Fully Reduce – Beispiel

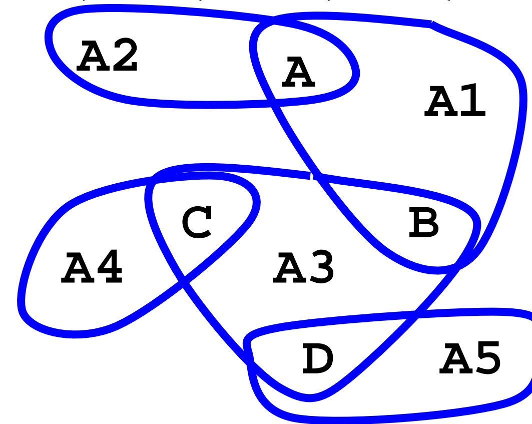



$$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$$

# Azyklische Anfragen

- Für azyklische Anfragen können wir in linearer Zeit einen Reducer für eine beliebig gewählte Relation finden
- Gegeben eine azyklische, nicht-lineare Anfrage Q
  - R1(A1,A,B), R2(A2,A), R3(A3,B,C,D), R4(A4,C), R5(A5,D)

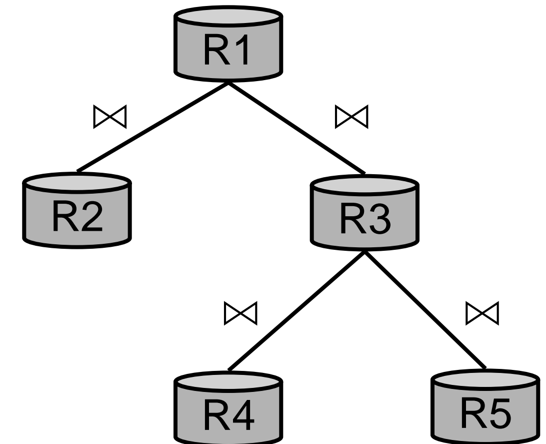



- Offensichtlich azyklisch
- Wähle eine Relation (R1) als letztes Element

# Reducer für azyklische Anfragen

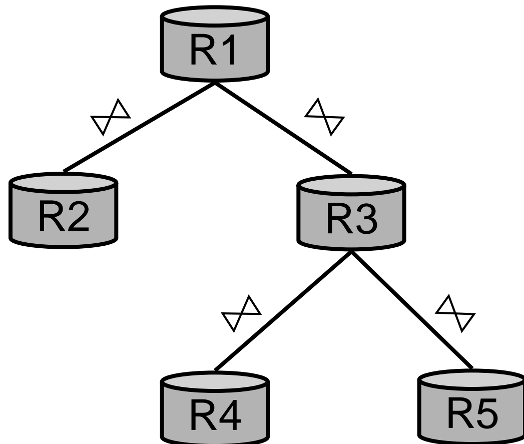
- Setze die gewählte Relation als Wurzel eines Baumes
- Baue den Baum von den Blätter her auf
  - Füge sukzessive abgeschnittene Ohren O zu dem Baum
  - Kinder sind die Ohren, die vorher abgeschnitten werden mussten, um O „abschneidbar“ zu machen
  - Wurzel muss bleiben
- Reducer für die Wurzel
  - Von unten nach oben
  - Einführung von Semi-Joins von Knoten zu ihren Eltern

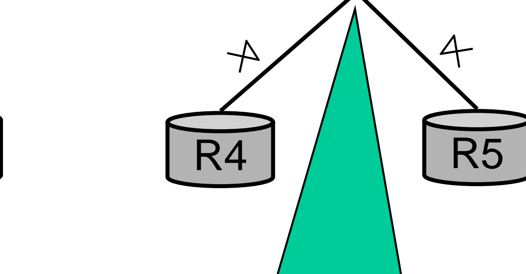


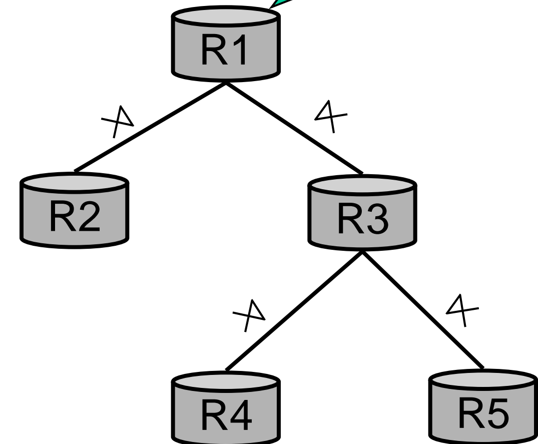
# Reducer für R1

R1(A1,A,B), R2(A2,A), R3(A3,B,C,D), R4(A4,C), R5(A5,D)

Erfüllte Bedingungen:
- R3.C = R4.C
- R3.D = R5.D
- R1.A = R2.A
- R1.B = R3.B




Erfüllte Bedingungen:
- R3.C = R4.C
- R3.D = R5.D

Ergebnis in R1 erfüllt alle Bedingungen, ist also reduced.

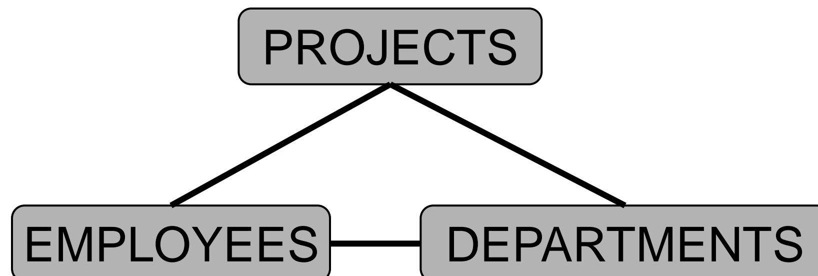
# Zyklische Anfragen

- Zyklische Anfragen
  - Alle Mitarbeiter, die an Projekten der eigenen Abteilung arbeiten

```
SELECT  EMP.name, DEPT.name
FROM    EMP, DEPT, PROJ
WHERE   EMP.d_ID = DEPT.ID
        AND EMP.p_ID = PROJ.ID
        AND PROJ.d_ID = DEPT.ID
```

| DEPT | |
|---|---|
| DName | ID |
| A | 1 |
| b | 2 |

| PROJ | | |
|---|---|---|
| d_ID | ID | PName |
| 1 | 6 | Clio |
| 2 | 7 | HumMer |




| EMP | | |
|---|---|---|
| EName | d_ID | p_ID |
| x | 1 | 7 |
| y | 2 | 6 |

Was passiert?

# Zyklische Anfragen

- Zyklische Anfragen
  - Alle Mitarbeiter, die an Projekten der eigenen Abteilung arbeiten

```
SELECT  EMP.name, DEPT.name
FROM    EMP, DEPT, PROJ
WHERE   EMP.d_ID = DEPT.ID
        AND EMP.p_ID = PROJ.ID
        AND PROJ.d_ID = DEPT.ID
```

| DEPT | |
|---|---|
| DName | ID |
| A | 1 |
| b | 2 |

| PROJ | | |
|---|---|---|
| d_ID | ID | PName |
| 1 | 6 | Clio |
| 2 | 7 | HumMer |

| EMP | | |
|---|---|---|
| EName | d_ID | p_ID |
| x | 1 | 7 |
| y | 2 | 6 |

- Semi-Join betrachtet nur zwei Relationen
- Kein paarweiser Join ist leer
- Es gibt keinen Full Reducer
- Aber das Gesamtergebnis ist leer

# Warnung

- Ein Full Reducer entfernt alle überflüssigen Tupel
- Um einen Full Reducer auszurechnen, müssen aber schon Tupel bewegt werden
- Wann die minimale Menge von Zwischenergebnissen bewegt wird, haben wir noch nicht betrachtet
  - Freiheitsgrade, z.B. Welche Relation wähle ich als Wurzel?
- Ein optimaler Plan unter Verwendung eines (Full-)Reducers muss auch andere Optimierungstechniken benutzen
  - Welches ist der beste (Full-)Reducer?
  - Wo und in welcher Reihenfolge werden Semi-Joins ausgeführt?
  - Minimaler Gesamttransport von Daten

# Literatur


- Philip A. Bernstein, Dah-Ming W. Chiu: Using Semi-Joins to Solve Relational Queries. Journal of the ACM 28(1): 25-40 (1981)
- W. Meng and C. Yu, "Query Processing in Multidatabase Systems," in Modern Database Systems, W. Kim, Ed. New York: ACM Press, Addison-Wesley, 1995, pp. 551-572.
- J. D. Ullman, „Principles of Database Systems and Knowledge-Based Systems. Volume II: The New Technologies“. Computer Science Press, Rockville, 1989.