Distributed Data Management

Distributed Query Optimization

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut

Distributed DBMSs

Overview

1. **Distributed Query Execution**
2. Distributed Join Execution
3. Bloom filter Optimized Joins
4. Multi-Relation Joins

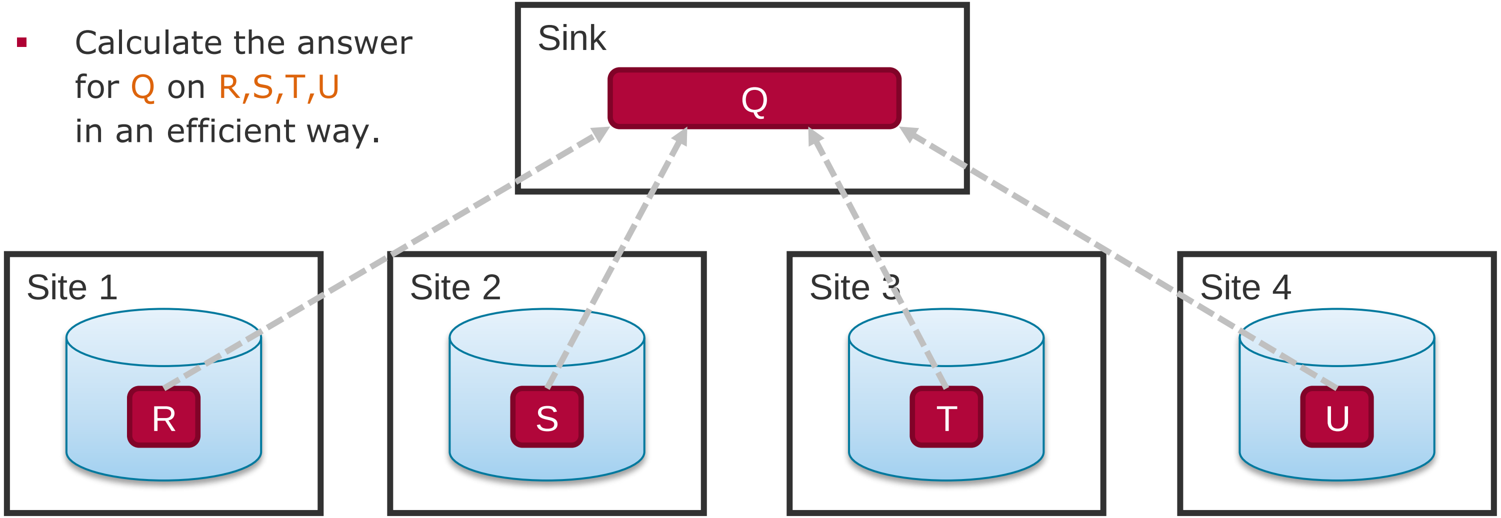
A Distributed Query

Given

- Relations R, S, T, U each on a different host (= site)
- Query Q issued by an arbitrary sink node

Task

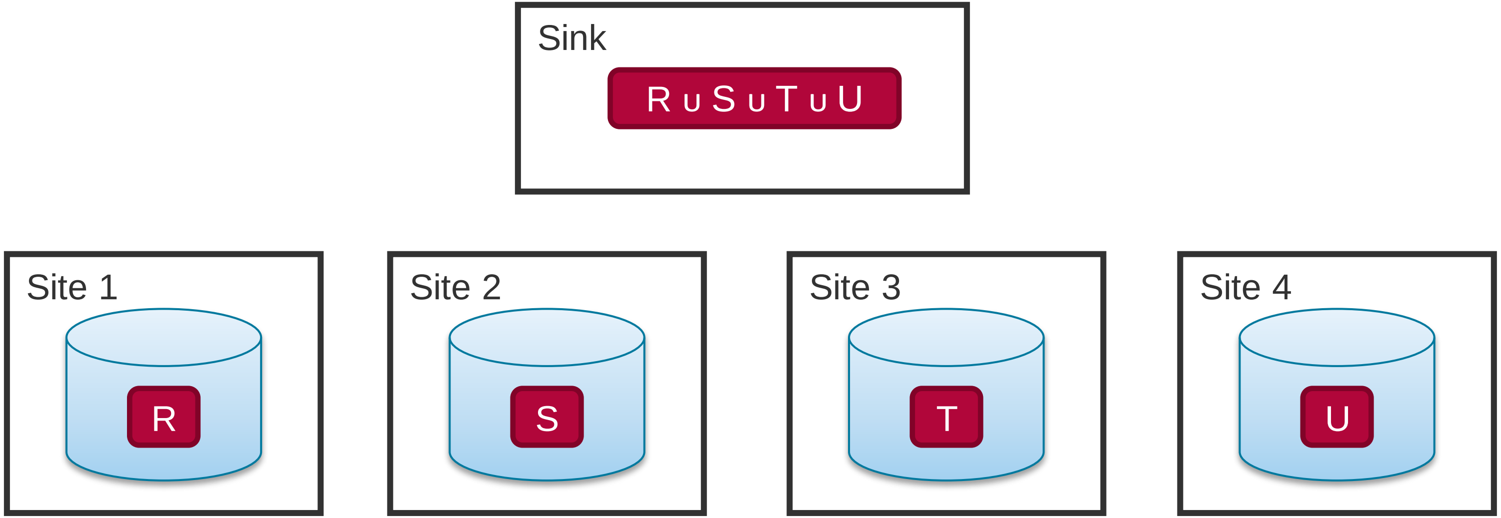
- Calculate the answer for Q on R,S,T,U in an efficient way.



Set Operations

Easy Operations

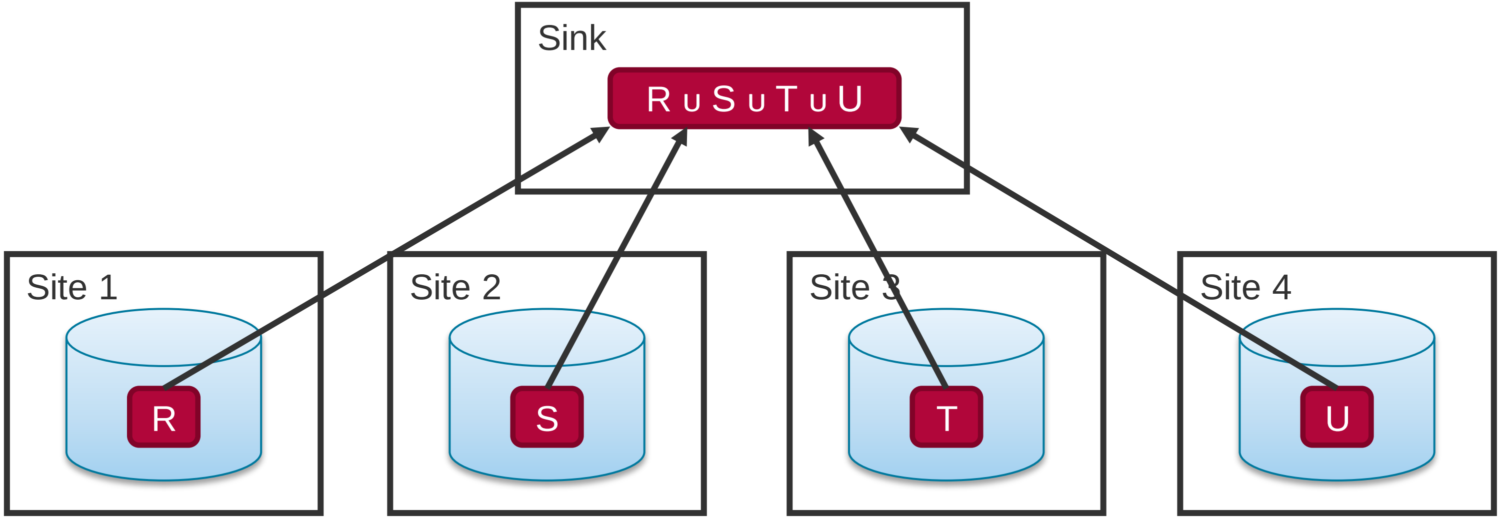
- Union:



Set Operations

Easy Operations

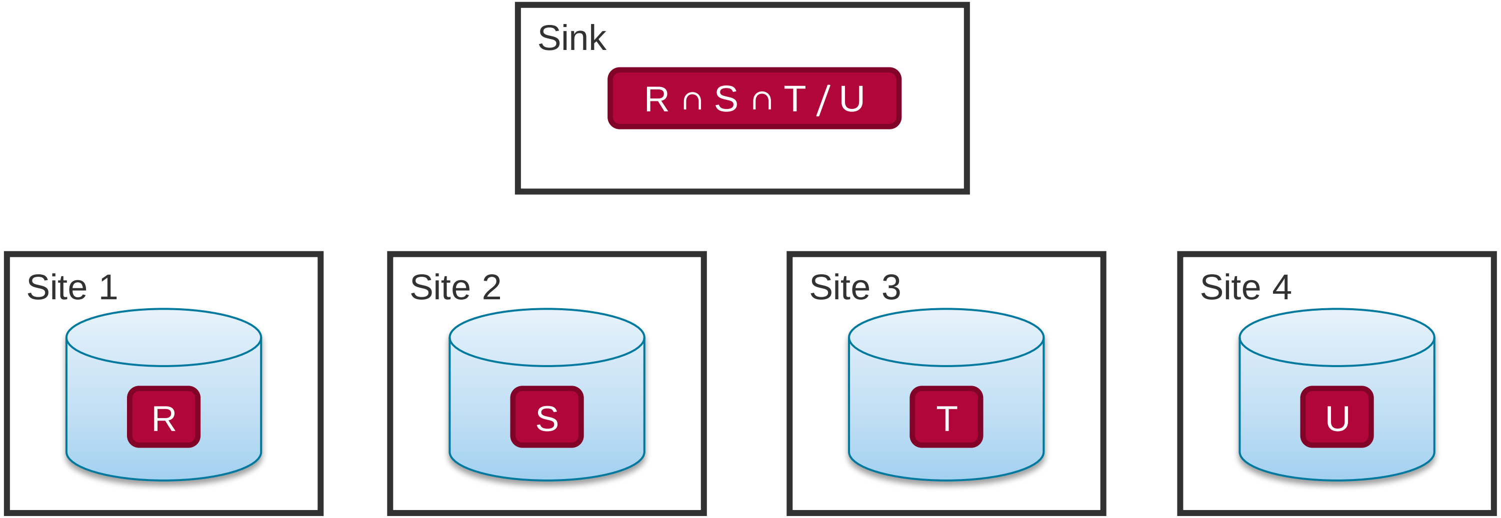
- Union: Send entire relations.



Set Operations

Easy Operations

- Union: Send entire relations.
- Except and Intersect:

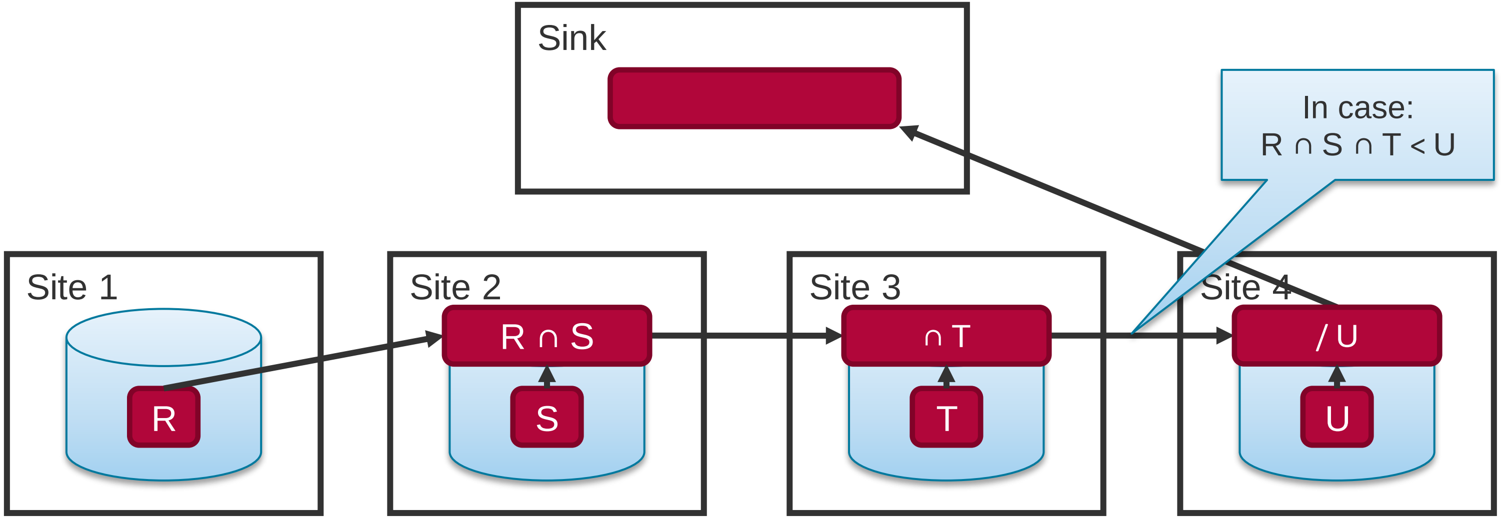


Distributed Query Execution

Set Operations

Easy Operations

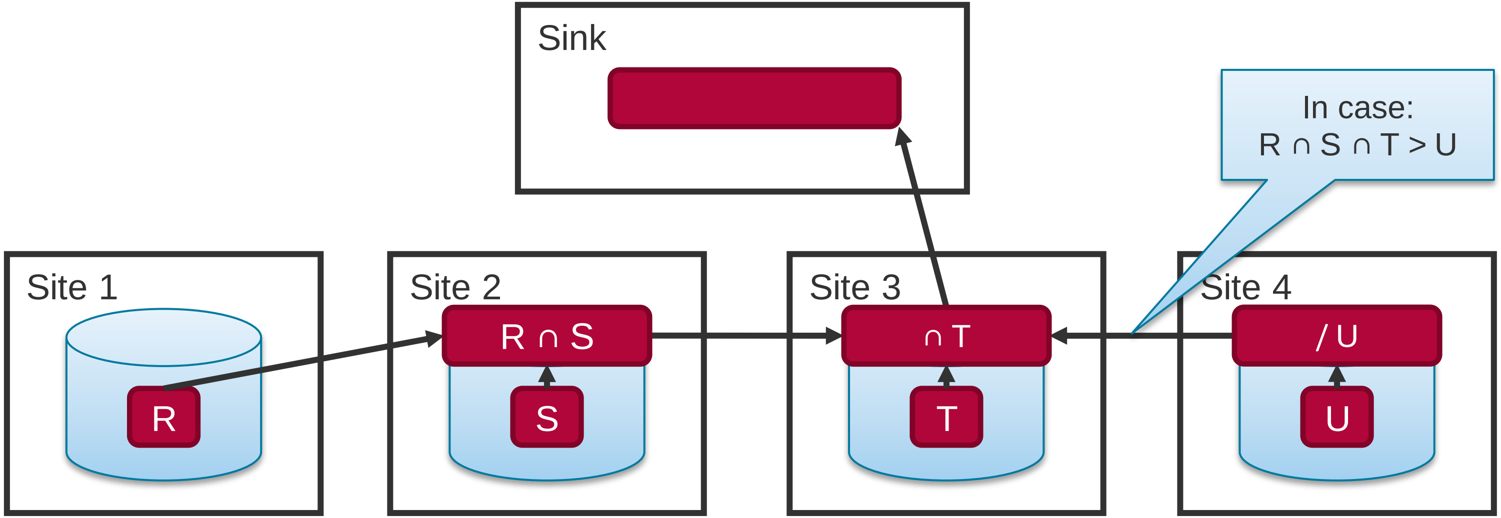
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.



Set Operations

Easy Operations

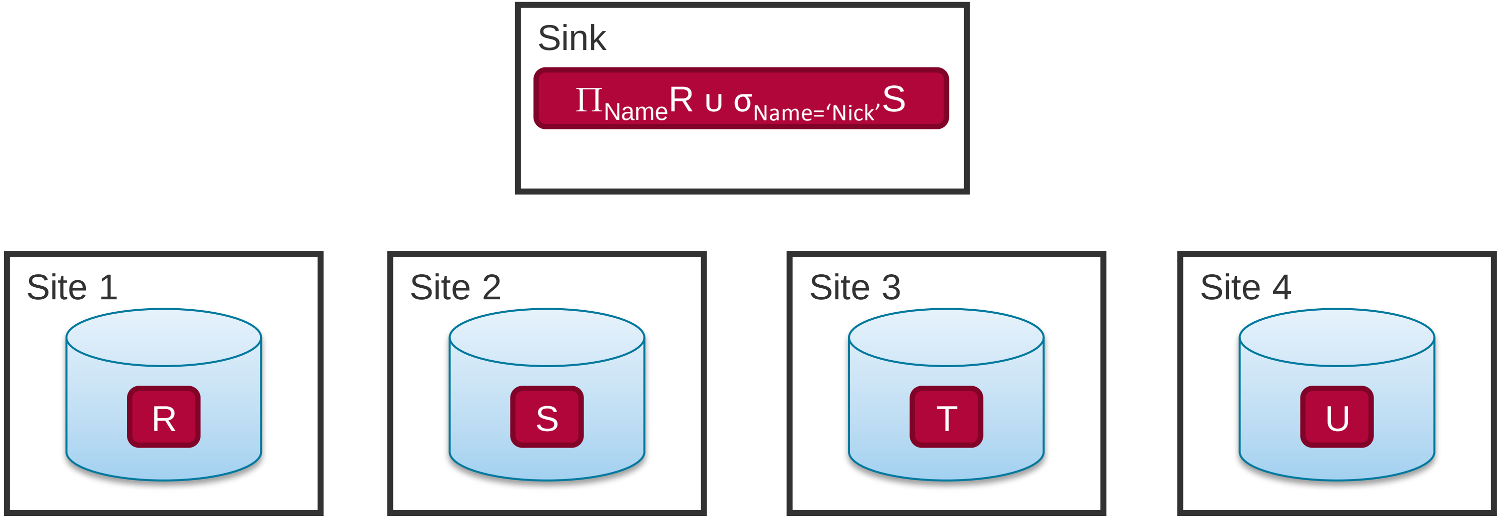
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.



Projections and Selections

Easy Operations

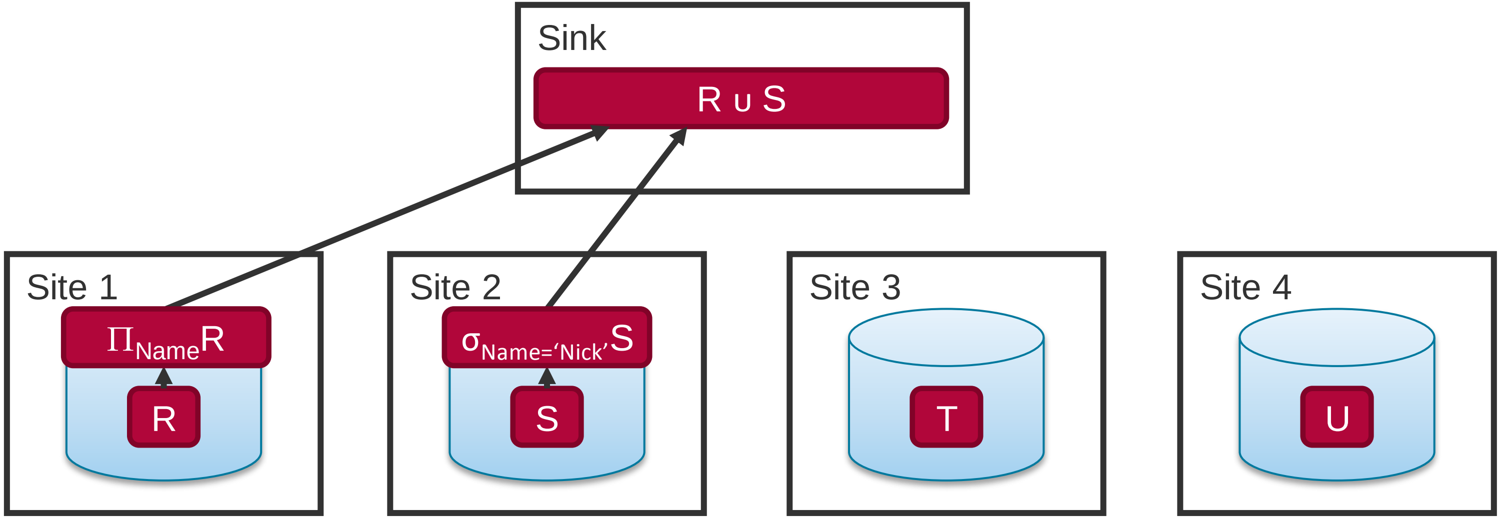
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections:



Projections and Selections

Easy Operations

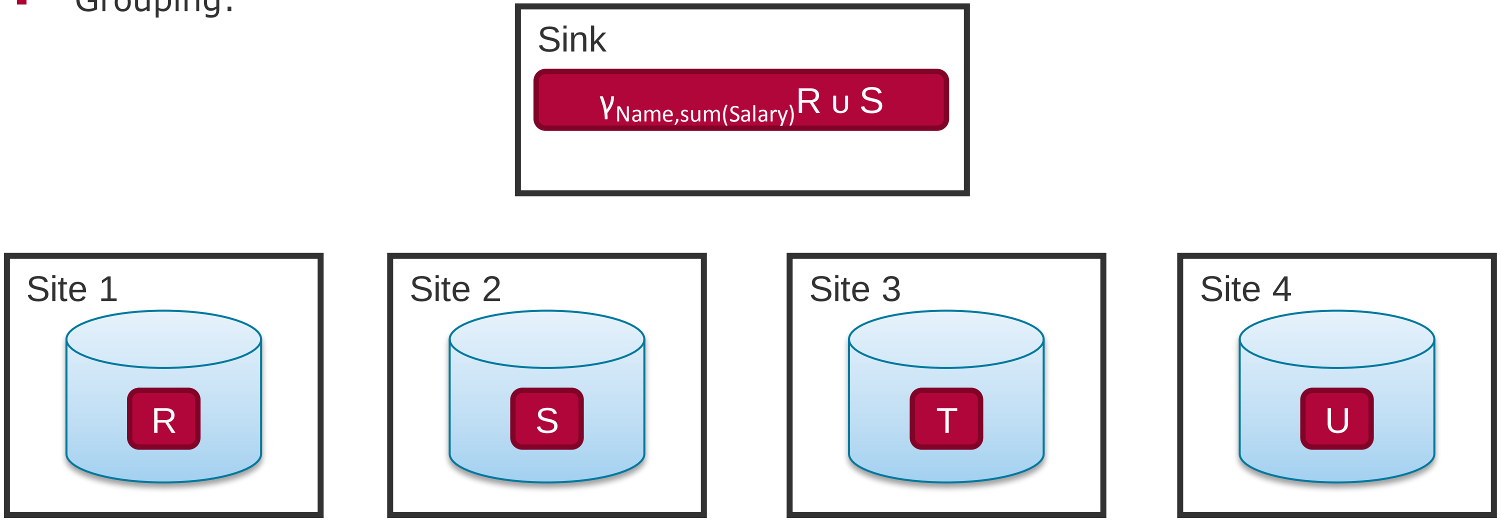
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections: Push operation down (if possible) and send the results to the sink.



Groupings

Easy Operations

- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections: Push operation down (if possible) and send the results to the sink.
- Grouping:

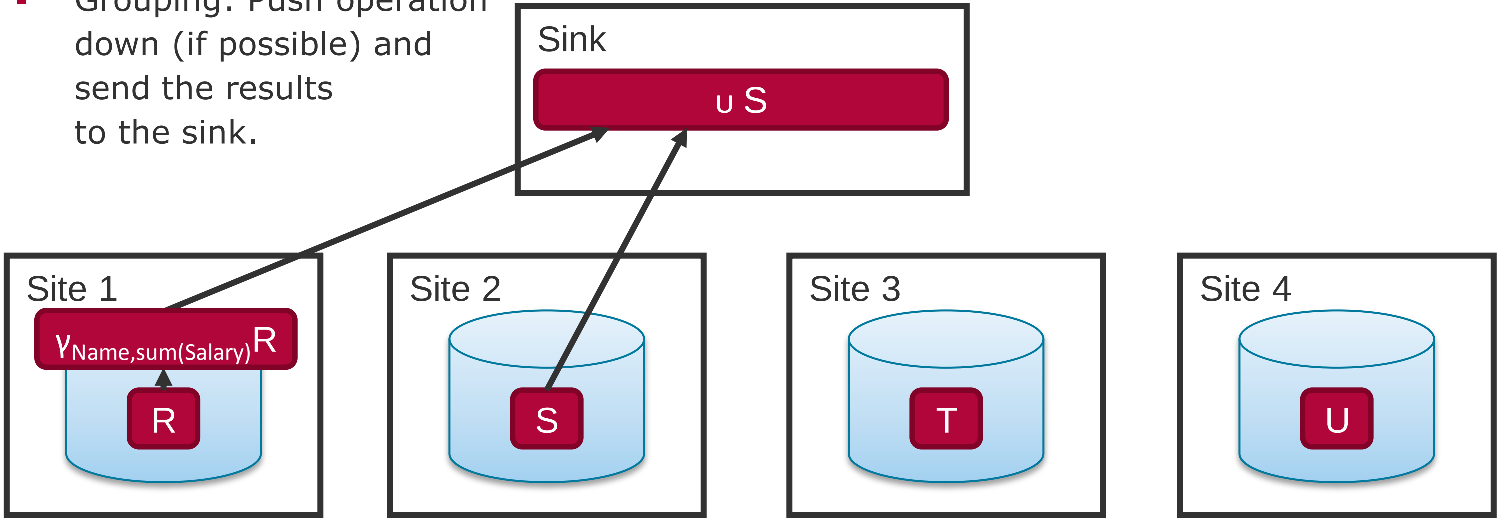


Distributed Query Execution

Groupings

Easy Operations

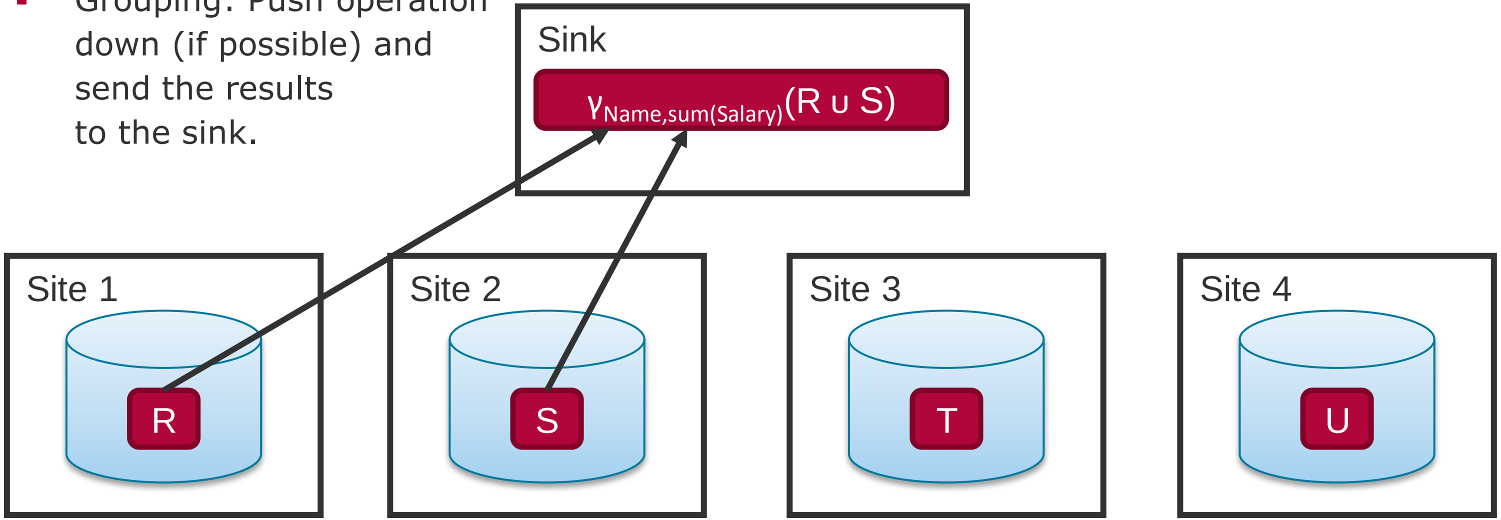
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections: Push operation down (if possible) and send the results to the sink.
- Grouping: Push operation down (if possible) and send the results to the sink.



Groupings

Easy Operations

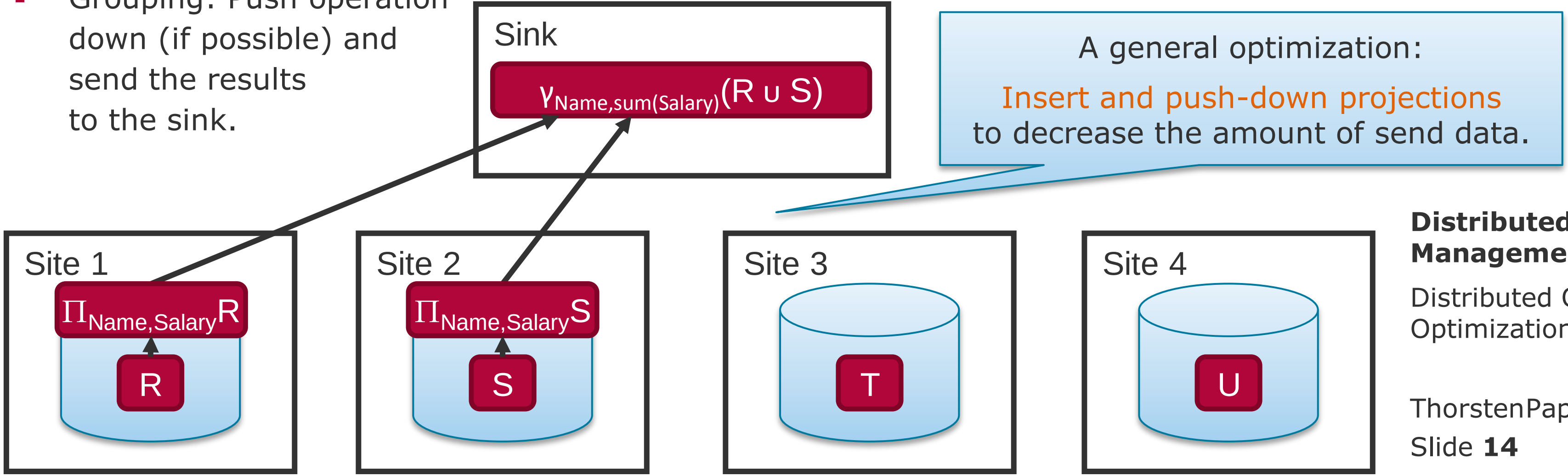
- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections: Push operation down (if possible) and send the results to the sink.
- Grouping: Push operation down (if possible) and send the results to the sink.



Groupings

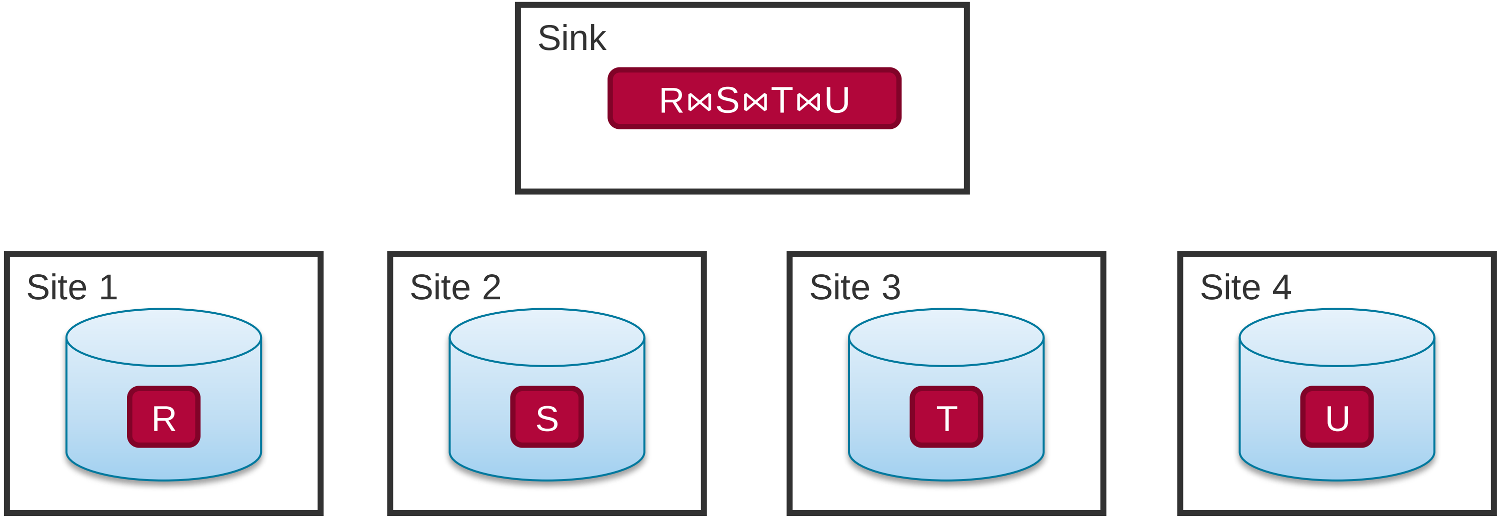
Easy Operations

- Union: Send entire relations.
- Except and Intersect: Send the smaller relation to the larger and the result to the sink.
- Projections and Selections: Push operation down (if possible) and send the results to the sink.
- Grouping: Push operation down (if possible) and send the results to the sink.



Joins

Join Operations

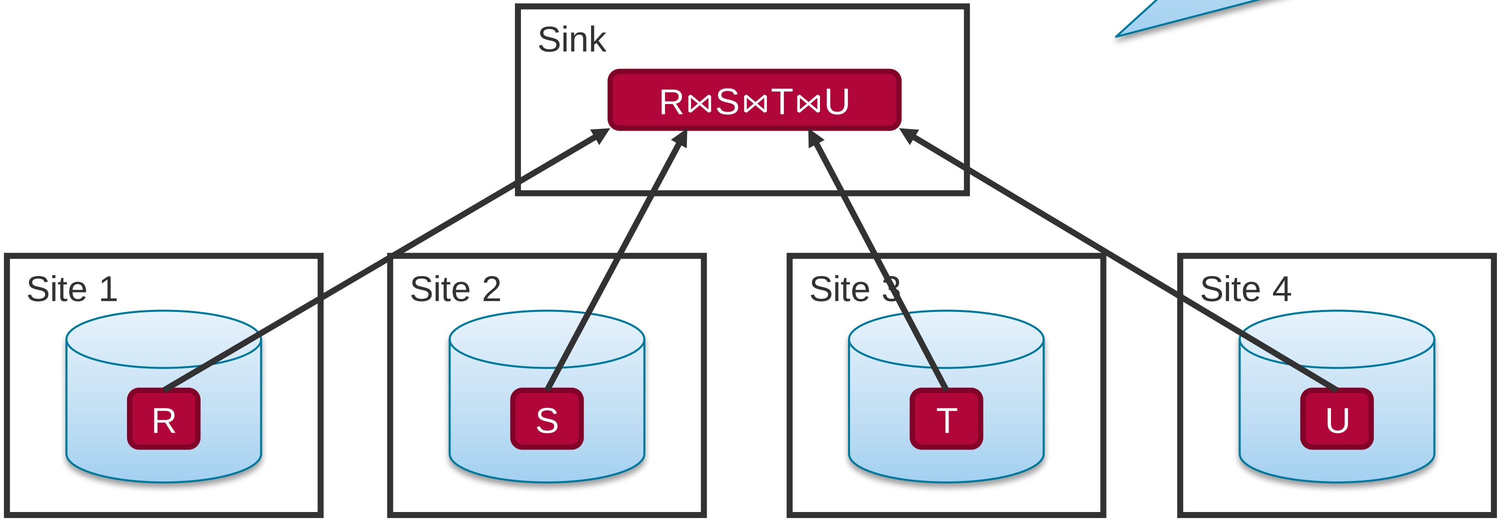


Joins

Join Operations

- Naïve approach: “Ship whole”
  - Send all relations
  - Few but large messages

We can do much better:
Calculate the join on only the join attributes and then fetch the data afterwards.



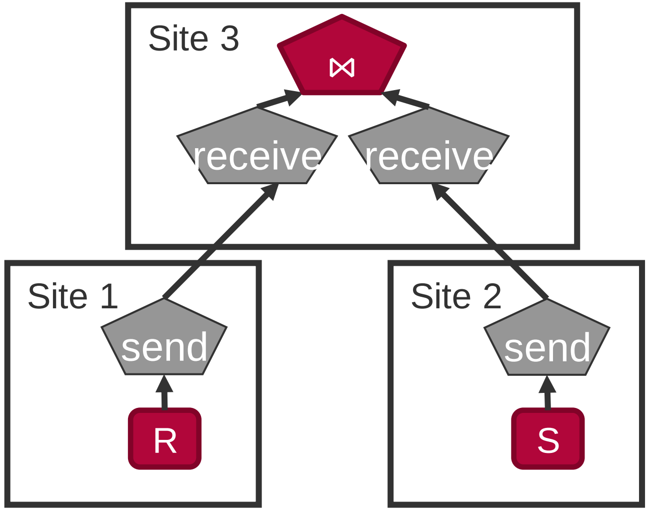
Distributed DBMSs

Overview

1. Distributed Query Execution
2. **Distributed Join Execution**
3. Bloom filter Optimized Joins
4. Multi-Relation Joins

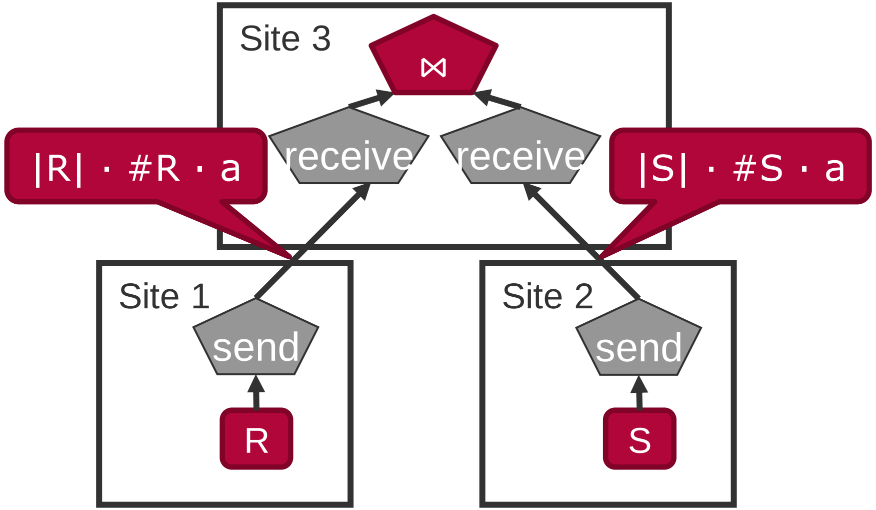
Naïve Join

- A join R⋈S over two relations R and S with
- There are |R| and |S| many attributes in R and S, respectively.
- There are #R and #S many values in R and S, respectively.
- Each attribute value in R and S has a size of a.
- Both R and S are stored on different hosts.
- Assume that one side can be the sink node.
- Two kinds of attributes:
  a. join-attributes
     (denoted as R.ID and S.ID
     but can have arbitrary names)
  b. data-attributes
     (denoted as R\ID and S\ID;
     = information that should be joined)
- Naïve join on third node
  - Costs: |R| · #R · a + |S| · #S · a

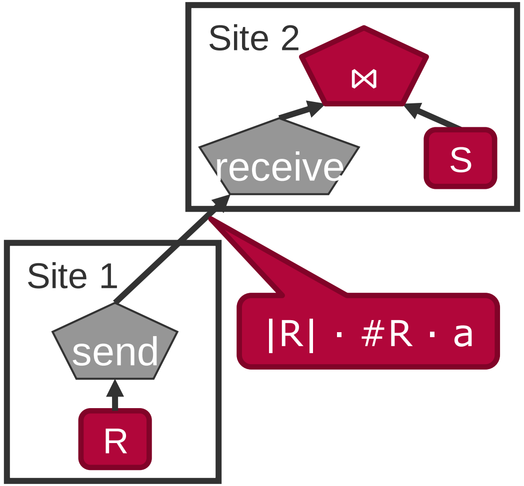


Site Join

- Naïve join on third node
  - Costs: |R| · #R · a + |S| · #S · a



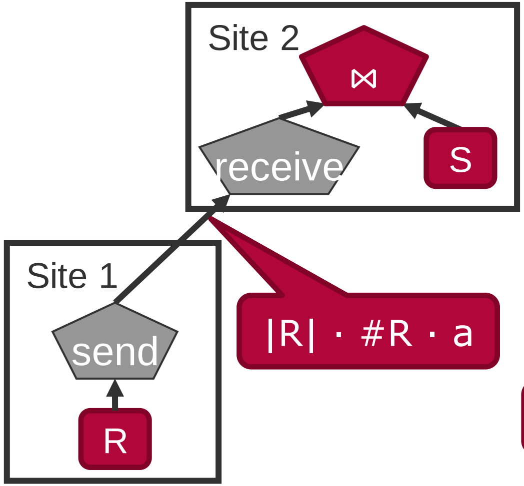
- Site join on one of the data nodes
  - Costs: |R| · #R · a



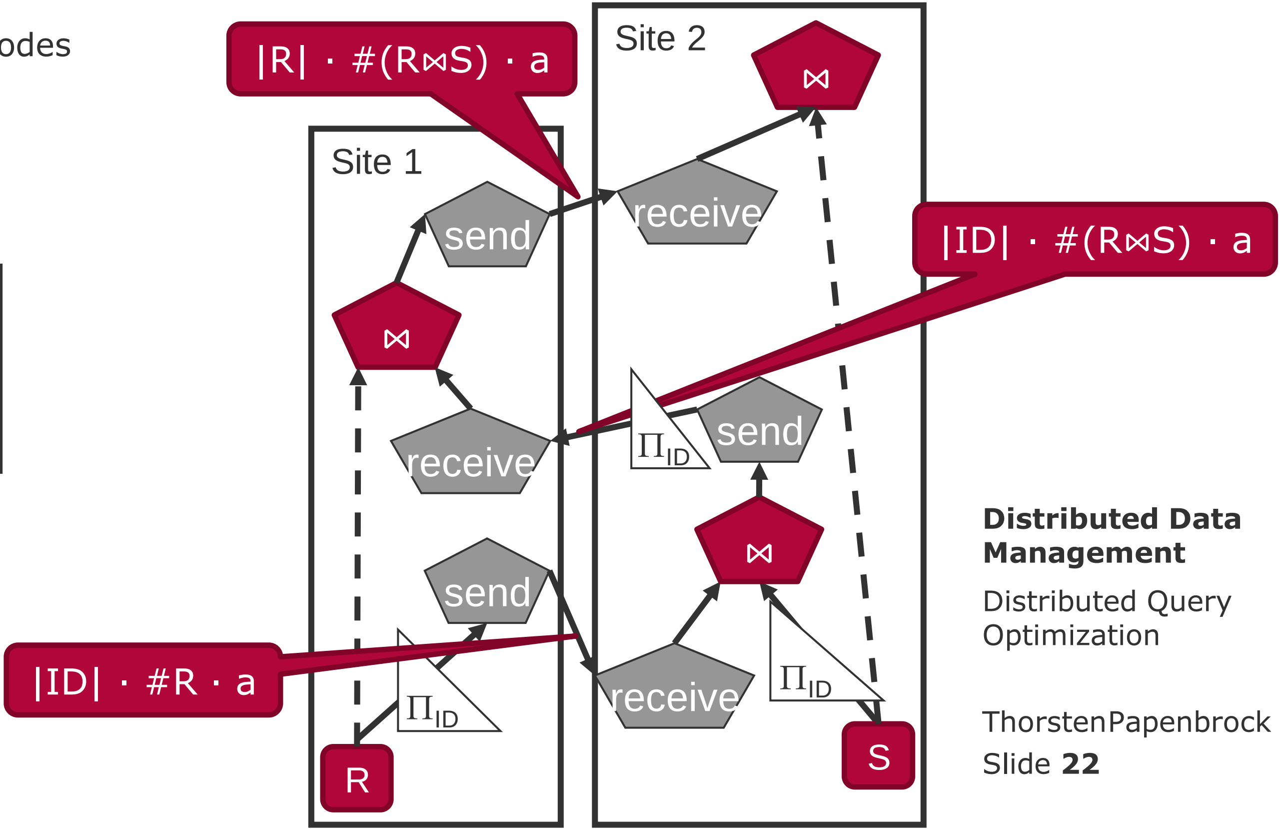
Distributed Join Execution

Projection Join

- Site join on one of the data nodes
  - Costs: $|R| \cdot \#R \cdot a$



- Projection join based on join attributes
  - Costs: $|ID| \cdot \#R \cdot a + |ID| \cdot \#(R \bowtie S) \cdot a + |R| \cdot \#(R \bowtie S) \cdot a$
    $= |ID| \cdot \#R \cdot a + (|ID| + |R|) \cdot \#(R \bowtie S) \cdot a$



Comparison

- Naïve join: $|R| \cdot \#R \cdot a + |S| \cdot \#S \cdot a$
- Site join: $|R| \cdot \#R \cdot a$
- Projection join: $|ID| \cdot \#R \cdot a + (|ID| + |R|) \cdot \#(R \bowtie S) \cdot a$

- When is the side join better than the projection join?

$$|R| \cdot \#R \cdot a > |ID| \cdot \#R \cdot a + (|ID| + |R|) \cdot \#(R \bowtie S) \cdot a \quad | \text{ Attribute size does not matter}$$

$$\Leftrightarrow \quad |R| \cdot \#R > |ID| \cdot \#R + (|ID| + |R|) \cdot \#(R \bowtie S)$$

- ➢ If $\#R >> \#(R \bowtie S)$ „If the join selectivity is high"
- ➢ If $|R| >> |ID|$ „If many data-attributes exist"

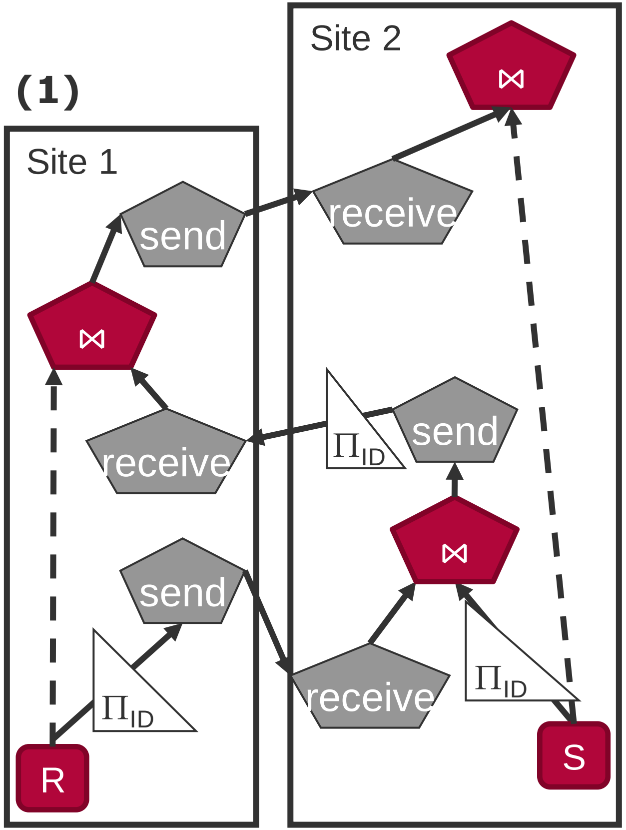
Distributed Join Execution

Projection Join (2)

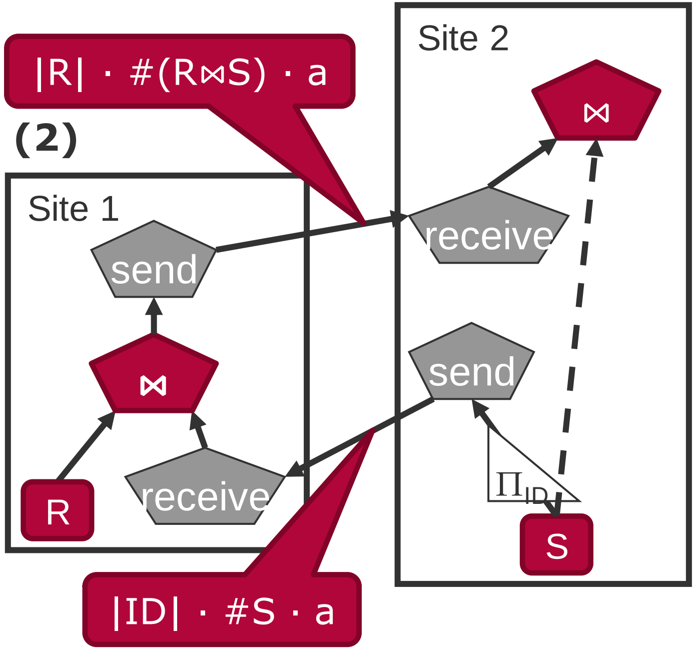
- $(1): |ID| \cdot \#R \cdot a + |ID| \cdot \#(R \bowtie S) \cdot a + |R| \cdot \#(R \bowtie S) \cdot a$
- $(2): |ID| \cdot \#S \cdot a + |R| \cdot \#(R \bowtie S) \cdot a$

  - ➢ If #R << #S, then (1) is likely better; otherwise (2) (with S being the relation on the site that should answer Q).
  - ➢ If we can choose the site for Q, then choose the smaller relation and strategy (2).

(1)



(2)

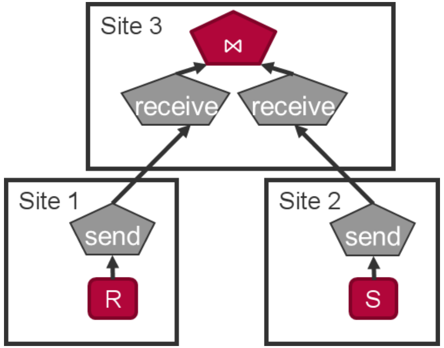


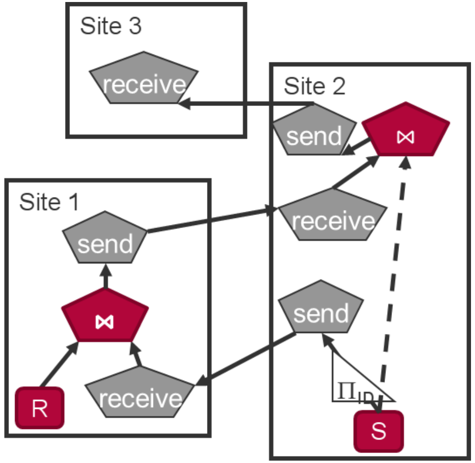
Distributed Join Execution

Three Sites

- Best solution so far:
  - Projection join (2): $|ID| \cdot \#S \cdot a + |R| \cdot \#(R \bowtie S) \cdot a$
- Costs if the result is needed on some third site:

  $|ID| \cdot \#S \cdot a + |R| \cdot \#(R \bowtie S) \cdot a + (|R| + |S|) \cdot \#(R \bowtie S) \cdot a$
- Which can be worse than the Naïve join on a third node if $\#(R \bowtie S)$ is large (i.e. if the join selectivity is small).
  - Naïve join: $|R| \cdot \#R \cdot a + |S| \cdot \#S \cdot a$

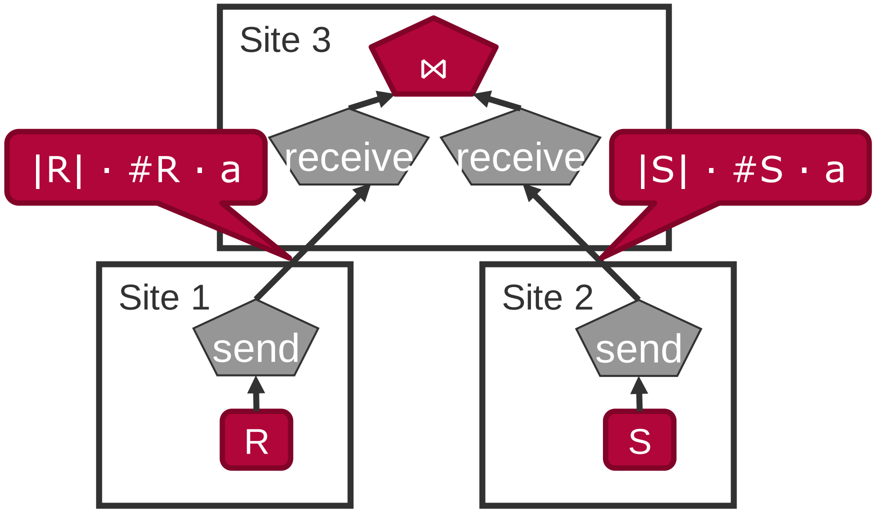


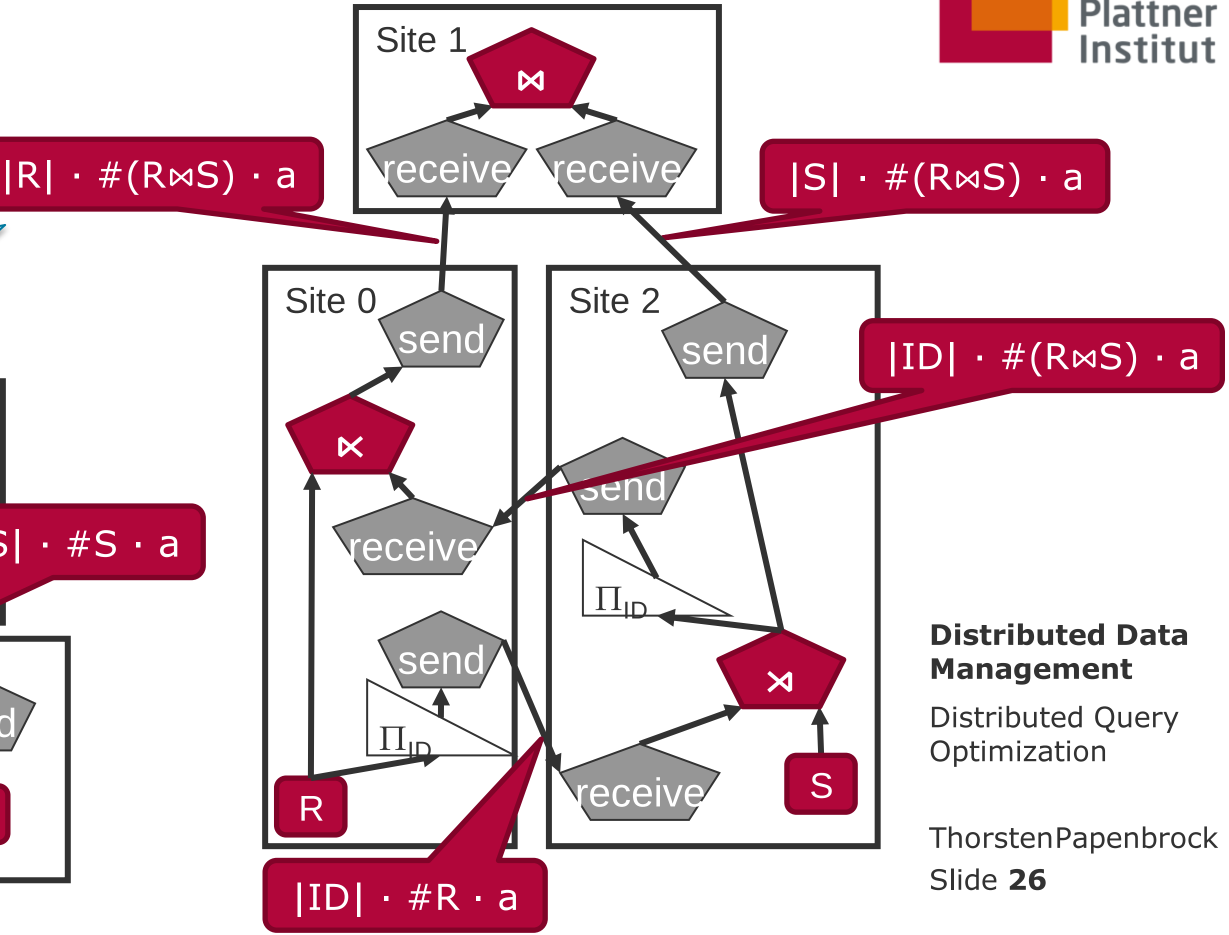


Distributed Join Execution

Three Site Join

Again: If #(R⋈S) is much smaller than #R and #S, the pre-filtering improves the query performance.





Semi-Join

Definition:

Given relation R with attribute set A and relation S with attribute set B.
The Semi-Join $R \ltimes S$ is defined as

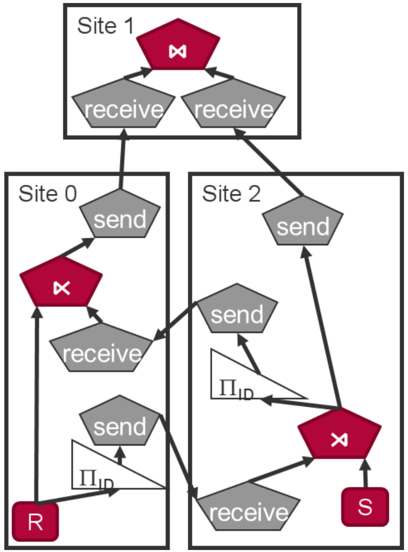
$$
\begin{aligned}
R \ltimes S \quad &:= \quad \Pi_A(R \bowtie_{A \cap B} S) \\
&= \quad \Pi_A(R) \bowtie_{A \cap B} \Pi_{A \cap B}(S) \\
&= \quad R \bowtie_{A \cap B} \Pi_{A \cap B}(S)
\end{aligned}
$$

- Remarks
  - The join is a natural join (over common attributes $A \cap B$).
  - For theta joins between R.X and S.Y it is: $R \ltimes S := R \bowtie_{X=Y} \Pi_Y(S)$
  - S functions as a filter on R‘s tuples.
  - The semi-join is asymmetric.

Distributed Join Execution
Semi-Join

- Semi-joins function as filters.
  - They can be used like selections and projections to minimize intermediate results before these are send to other sites.
- Rules:

  $R \bowtie_F S$ =
  - $(R \ltimes_F S) \bowtie_F S$
    - Filter R, then join with S
  - $R \bowtie_F (S \ltimes_F R)$
    - Filter S, then join with R
  - $(R \ltimes_F S) \bowtie_F (S \ltimes_F R)$
    - Filter R and S, then join both results



Distributed DBMSs

Overview

1. Distributed Query Execution
2. Distributed Join Execution
3. **Bloom filter Optimized Joins**
4. Multi-Relation Joins

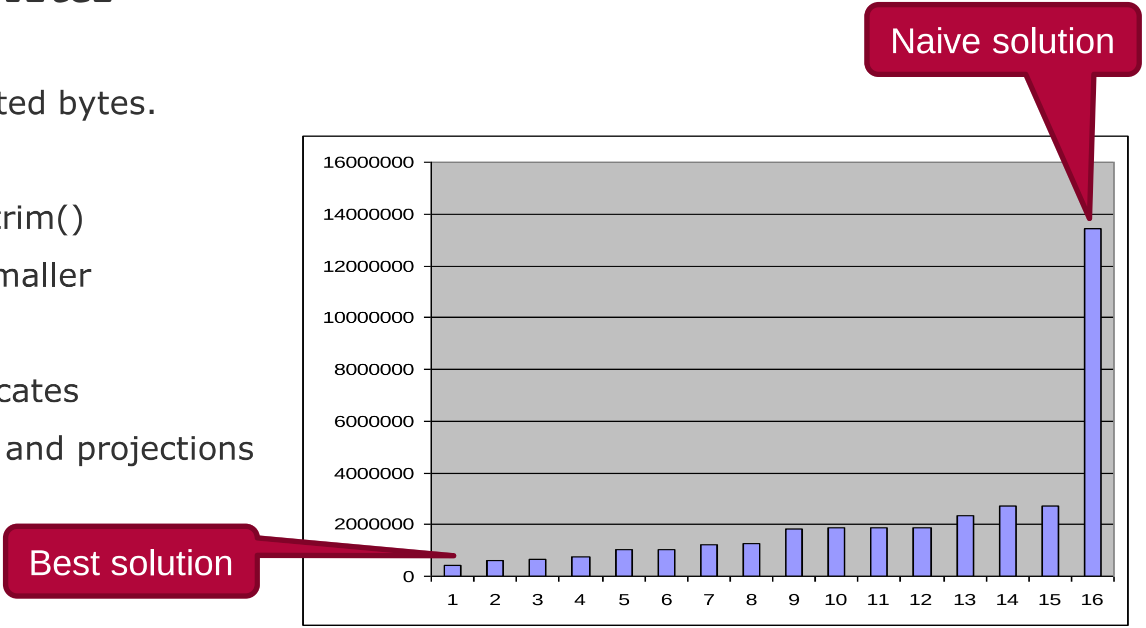
Bloom filter Optimized Joins

From a Database Exercise

- "Find all titles and directors of films that are younger than 1980."

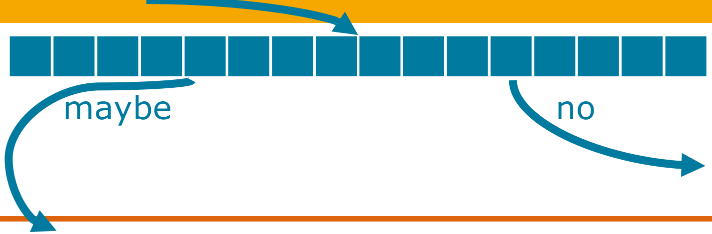
```
SELECT   F1.Titel, F2.Regie
FROM     Movie1.Filme1 F1, Movie2.Filme2 F2
WHERE    F1.Titel = F2.Titel
AND      F1.Jahr > 1980
```

- Task: Minimize the number of transmitted bytes.
- Tricks:
  - Transfer only necessary bytes: Rtrim()
  - Use better join order: Filme2 is smaller
  - Insert projections where possible
  - Use compression: Eliminate duplicates
    - DISTINCTs after semi-joins and projections
  - Bloom filter



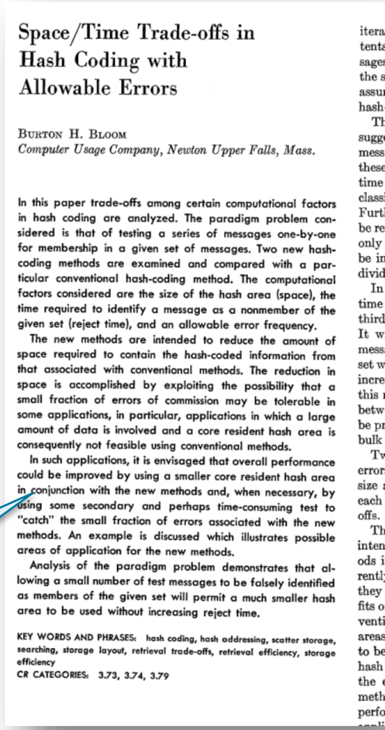
Fast Storage and Retrieval

Bloom filter (Recap)



A Bloom filter is a probabilistic data structure that answers set containment questions in constant time and with constant memory consumption.

- "Does element X appear in the set?"
- Answer "no" is guaranteed to be correct.
- Answer "yes" has a certain probability to be wrong (hence, "maybe").
  - ➢ But then the concrete look-up will just fail.
  - ➢ Very nice property that allows the use of Bloom filters in exact systems.
- Structure
  - Bitset of fixed size (typically a long array)
  - One (or more) hash functions



Space/Time Trade-offs in Hash Coding with Allowable Errors

Burton H. Bloom
Computer Usage Company, Newton Upper Falls, Mass.

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications, in particular, applications in which a large amount of data is involved and a core resident hash area is consequently not feasible using conventional methods.

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

Analysis of the paradigm problem demonstrates that allowing a small number of test messages to be falsely identified as members of the given set will permit a much smaller hash area to be used without increasing reject time.

KEY WORDS AND PHRASES: hash coding, hash addressing, scatter storage, searching, storage layout, retrieval trade-offs, retrieval efficiency, storage efficiency
CR CATEGORIES: 3.73, 3.74, 3.79

> 6,000 citations

Burton H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, volume 13, number 7, pages 422-426, 1970

Bloom filter

- Problem: Is f an element of column A?
  - Column A is large – must be stored on disk
- Idea: Store small representation of A in main memory
  - ➢ Bloom filter H
- Use H to probabilistically mark whether an element f is in A.
- Test can fail, but only in one direction:
  - If $k \in T$, we cannot be sure whether $k \in A$.
  - If $k \notin T$, we know that $k \notin A$.

➢ Use H for a probabilistic semi-join implementation!

Bloom filter for Semi-Joins

Bloom filter-based (semi-)joins

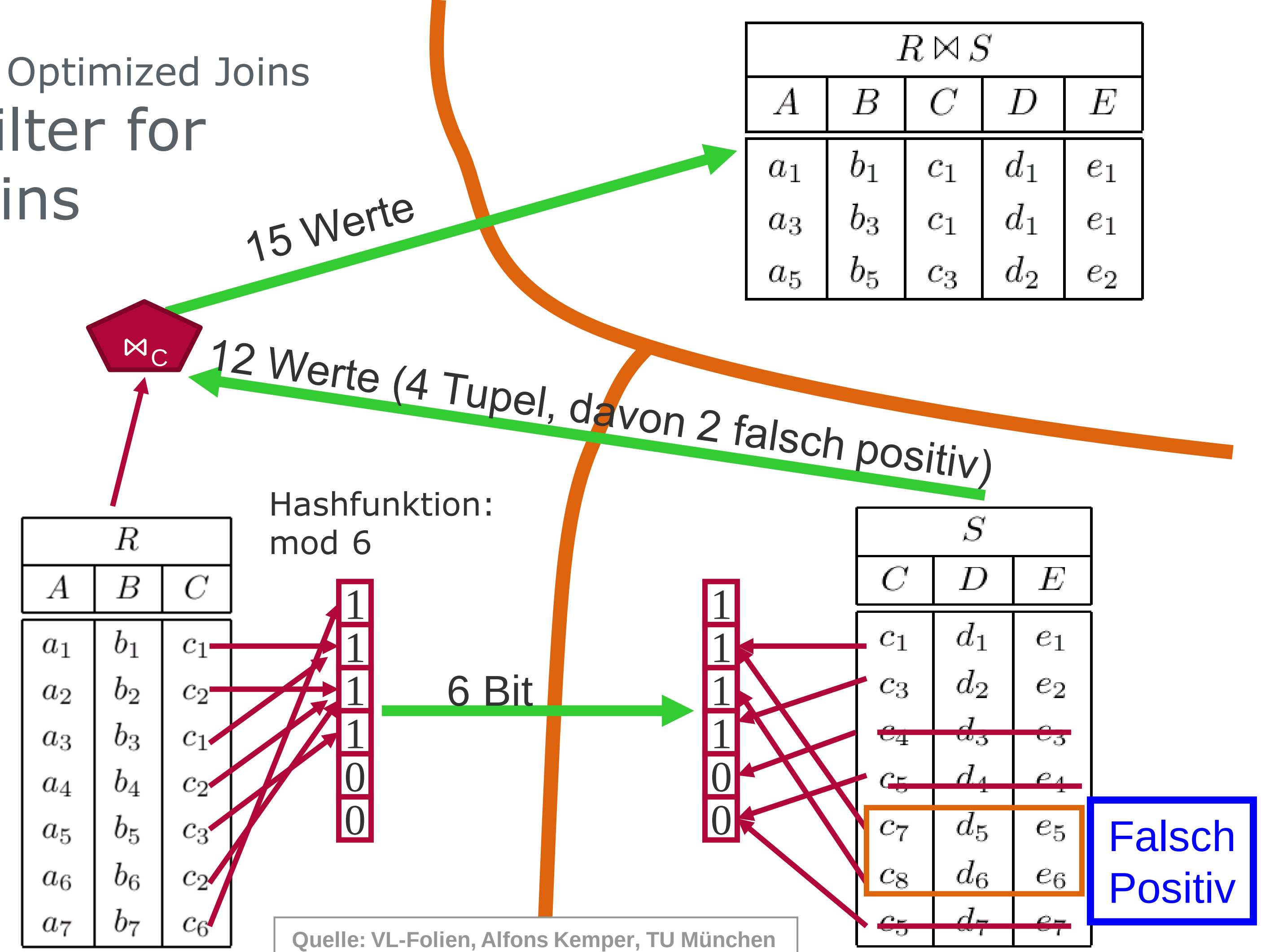
- Also called hash-filter-joins
- Use Bloom filter to calculate $R \ltimes_F S$:
  1. Hash all values in R.F with funktion h into (small) hash tabelle H
  2. Transmit only H to S
  3. $\forall$ f$\in$S.F with H(h(f))=0: f does not have a join partner in R; ignore local record.
  4. $\forall$ f$\in$S.F with H(h(f))=1: f does probably have a join-partner in R; send local record.

- The higher the join selectivity…
  - the lower the risk of false positives.
  - the smaller we can make H.

Bloom filter Optimized Joins

Bloom filter for Semi-Joins

Example



Quelle: VL-Folien, Alfons Kemper, TU München

Bloom filters as universal Trick

- Always use Bloom filters if …
  - sets of values need to be compared and
  - only a few hits are expected and
  - data transfer is expensive.

- Examples:
  - „Normal“ hash joins
  - Star joins in data warehouses
  - Intersect and minus set operations

Distributed DBMSs

Overview

1. Distributed Query Execution
2. Distributed Join Execution
3. Bloom filter Optimized Joins
4. **Multi-Relation Joins**

The Semi-Join Trick for Multi-Relation Joins

Task

- Calculate the join across arbitrary many relations.
- Example with three relations:

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

…

Approach

- Use semi-joins on any relation that needs to be transmitted.
- Semi-joins reduce the relations to only necessary tuples.
  - ➢ Hence, they are called "reducer".
- A relation is called "reduced" if it does not contain any tuple that is not needed for the final result.
  - ➢ Global property, because also remote relations reduce needed tuples.

Full Reducer

Semi-Join Program

- *Given the realtions $R_1,\ldots,R_n$, a semi-join program is a sequence of semi-joins*

$$R_i := R_i \ltimes R_j$$

- Comments:
  - We omit join attributes, because they result from the join query.
  - The effect of the semi-join is a reduction of the tuples in R_i.

Full Reducer

- *Given a query $Q=R_1 \bowtie \ldots \bowtie R_n$, a reducer for R_i in Q is a semi-join programm that removes all tuples from R_i that are not needed to calculate result(Q).*
- *A full reducer for Q is a semi-join programm that is a reducer for all R_i in Q.*
- Comments:
  - The R_i do not need t be different (self-joins)
  - Intuition: reducer for relations – full reducer for queries

Multi-Relation Joins

Full Reducer

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

- Is this a full reducer?
  - No, because S and T are not „reduced".
- But it is enough to minimize network traffic, i.e., R is minimized before sending to S and S is minimized after its join with T before sending to T, right?
  - Yes, but only if the join $\bowtie_F$ is calculated on S's node and $\bowtie_G$ on T's node.
  - If the join is evaluated elsewhere, we transmit $(S \ltimes_G T)$ and T.
    - ➢ Calculate full reducer first!

Full Reducer for linear Joins

- Given: $Q = R1 \bowtie_A R2 \bowtie_B \ldots \bowtie_Y R(n\text{-}1) \bowtie_Z Rn$
- Task: Find a full reducer for Q that reduces all Ri.

- Two-Phase Approach:
  - Forward:
    - $R2` = R2 \ltimes R1$
    - $R3` = R3 \ltimes R2` = R3 \ltimes (R2 \ltimes R1)$
    - …
    - $Rn` = Rn \ltimes R(n\text{-}1)` = \ldots$
  - Backward:
    - $R(n\text{-}1)`` = R(n\text{-}1)` \ltimes Rn`$
    - $R(n\text{-}2)`` = R(n\text{-}2)` \ltimes R(n\text{-}1)`$
    - …
    - $R1`` = R1 \ltimes R2``$

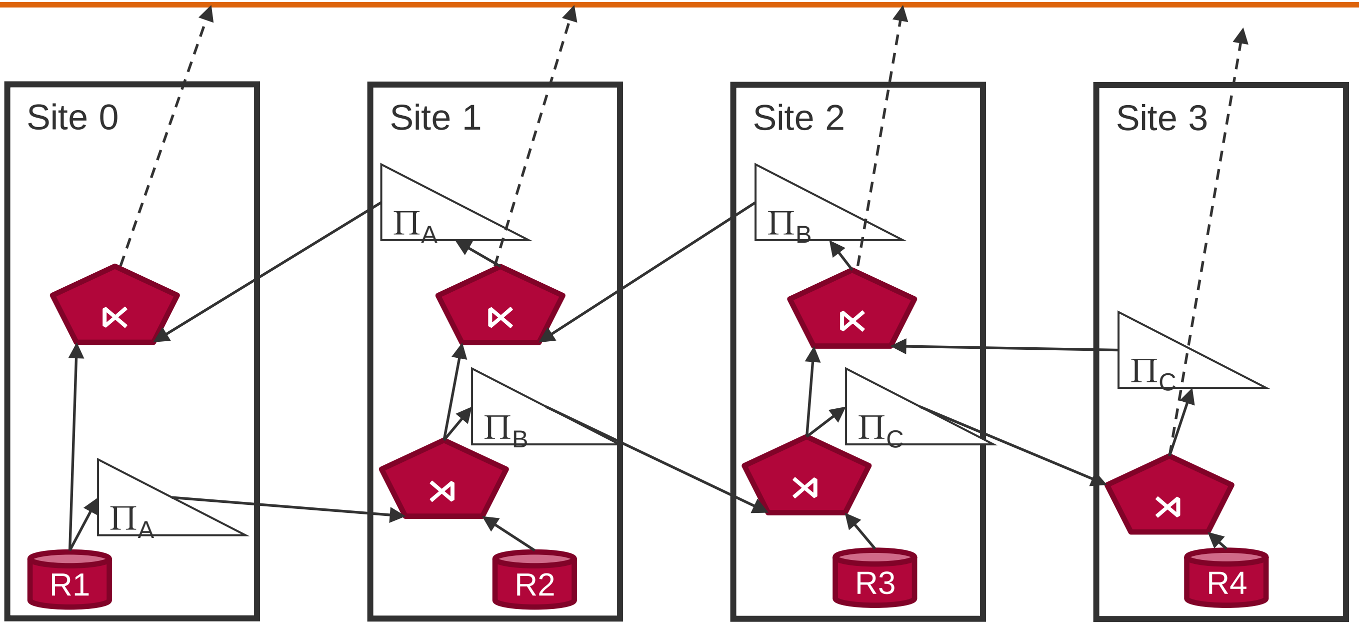
Reducer für Rn

Reducer für R(n-1)

Reducer für R1

Full Reducer for Q

Full Reducer for linear Joins



Forward
R2‘ = R2 ⋉ R1
R3‘ = R3 ⋉ R2‘
R4‘ = R4 ⋉ R3‘

Backward
R3“ = R3‘ ⋉ R4‘
R2“ = R2‘ ⋉ R3“
R1“ = R1 ⋉ R2“

Semi-Join/ Reducer

Full Reducer for linear Joins – Example

Site 0

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

R1

Site 1

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

R2

Site 2

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

R3

Site 3

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

R4

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

Full Reducer for linear Joins – Example

Site 0

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

R1

(1,5,6,7)

Site 1

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

R2

Site 2

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

R3

Site 3

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

R4

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

Full Reducer for linear Joins – Example

Site 0

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

R1

(1,5,6,7)

Site 1

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

R2

(3,5,7)

Site 2

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

R3

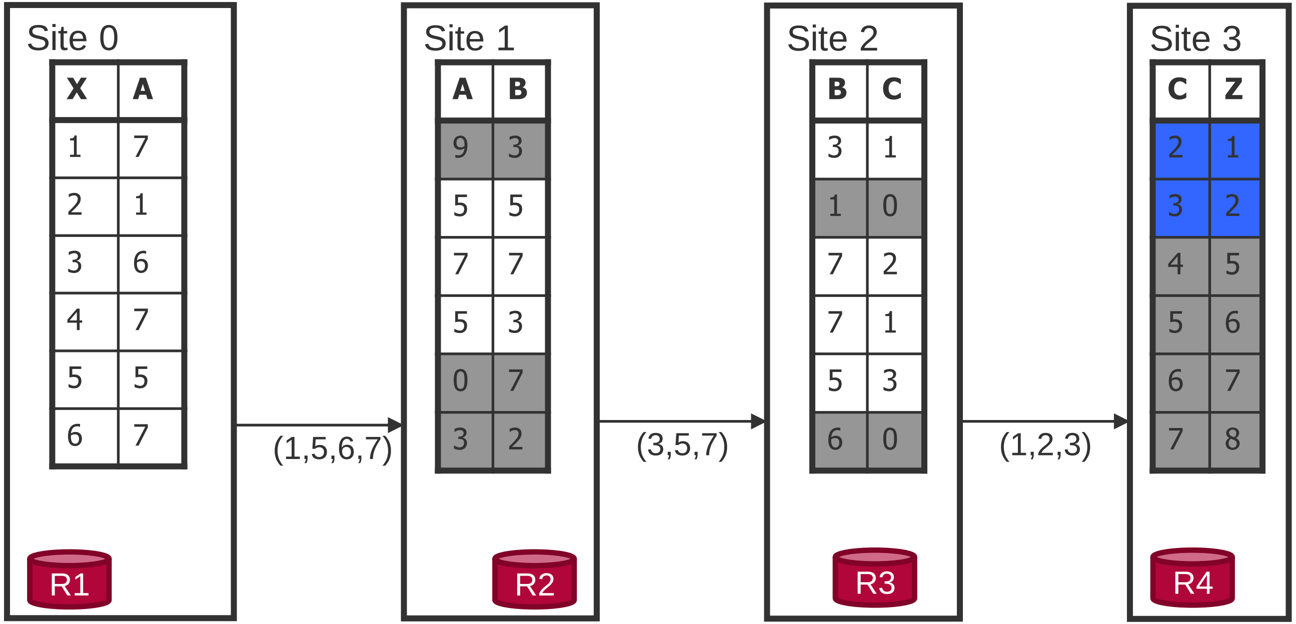
Site 3

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

R4

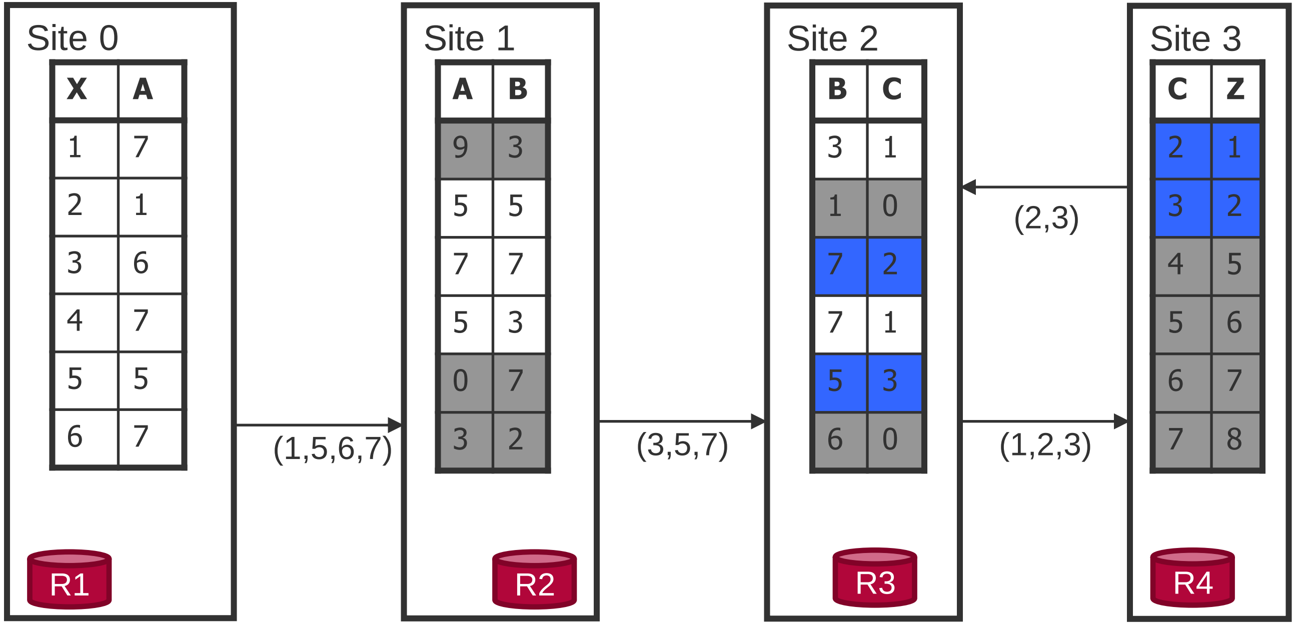
$$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$$

Full Reducer for linear Joins – Example



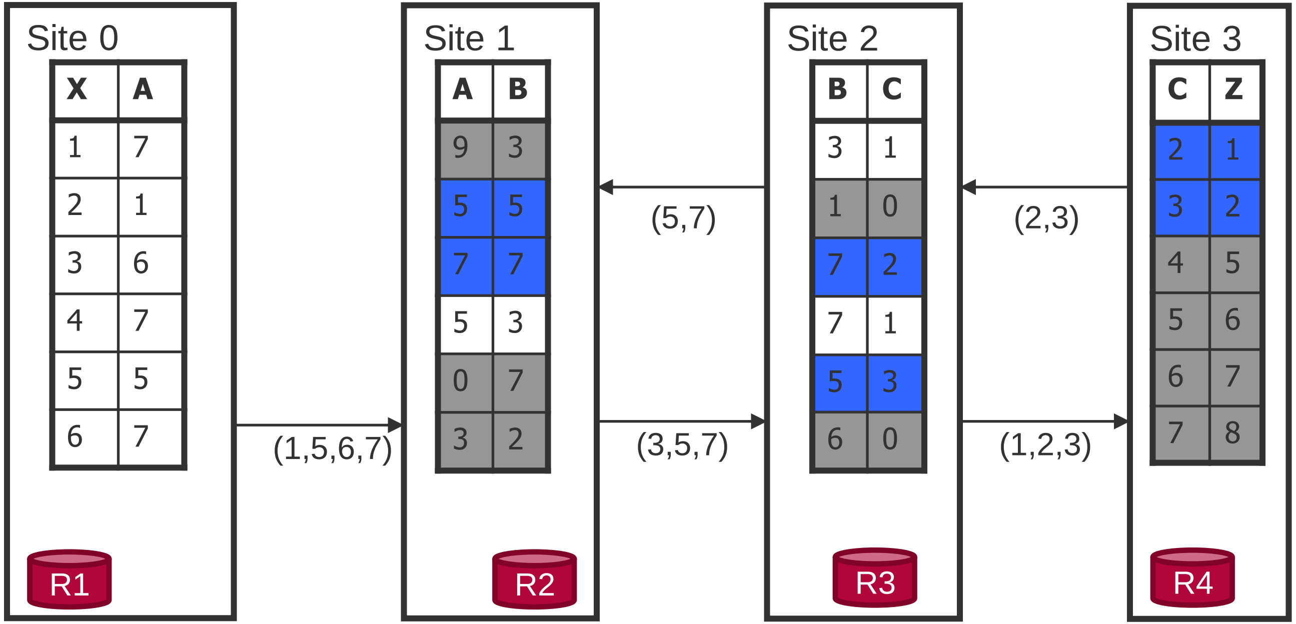
$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

Full Reducer for linear Joins – Example



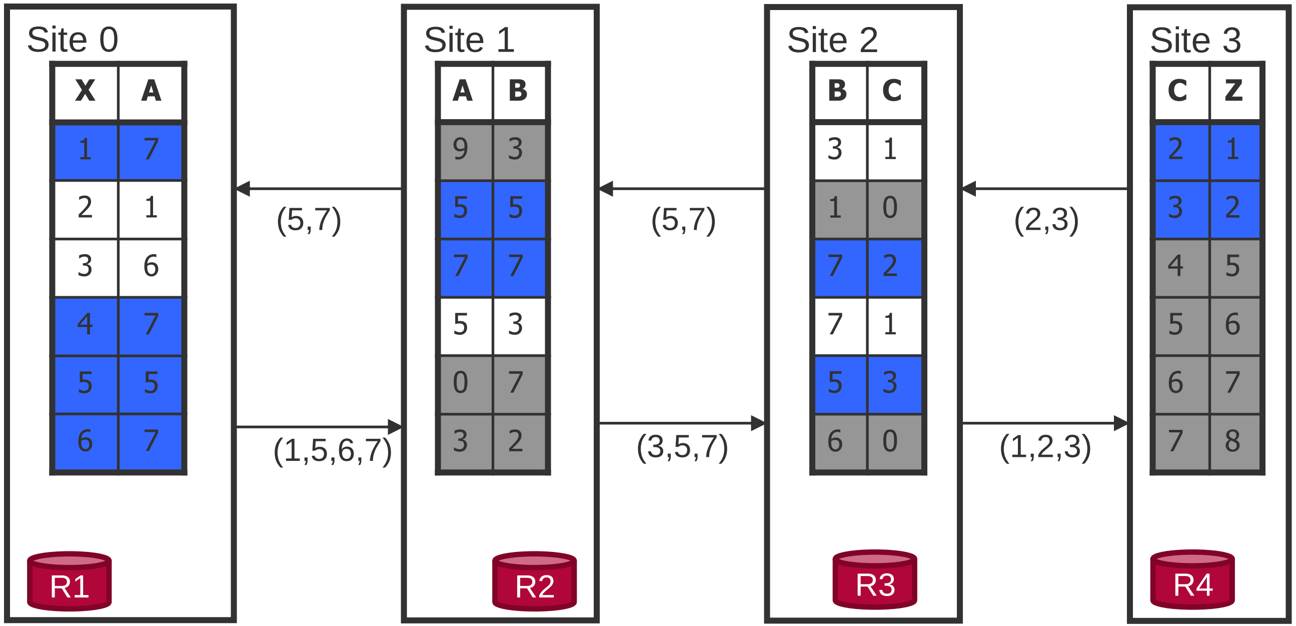
$R1 \bowtie_A R2 \bowtie_B R3 \bowtie_C R4$

Full Reducer for linear Joins – Example



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

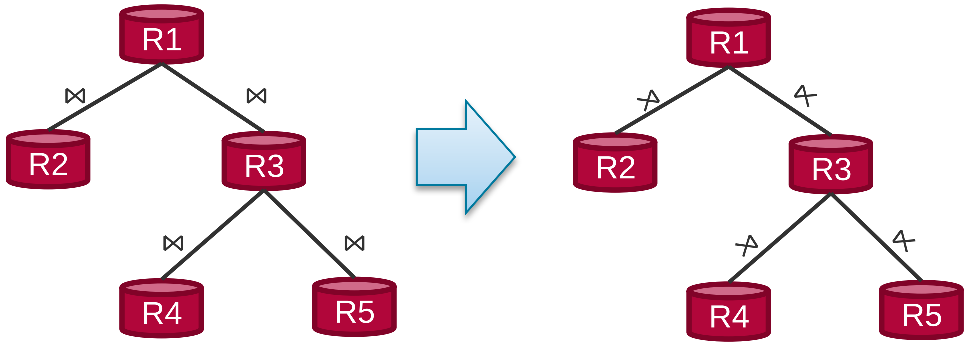
Full Reducer for linear Joins – Example



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

Reducer for non-linear Joins

- Given: A non-linear but acyclic join
- Task: Find a reducer for each relation.



- Approach:
  - Select the relation that needs to be reduced as root node.
  - Reduce the relations bottom-up level-wise to the root node.
    - ➢ Add semi-joins from nodes to their parent.

Reducer – Final Notes

- Finding a full reducer for cyclic joins is a problem.
  - In many cases, this full reducer simply does not exist.

- Optimizing reducer calculation in practice is challenging:
  - Semi-joins also need to send around data.
    - Does the minimization even pay off?
  - Minimizing intermediate results is challenging.
    - Which relation is the best root node?
  - Not all nodes may be able to perform query calculation.
    - What is the best reduce order?
    - Where do we calculate the semi-joins?
    - Do we need to calculate a *full* reducer?