CSEP544
Data Management

Lectures 8
Advanced Topics

Announcements

- Today was the last review!
- Project milestone due this Sunday, 3/2
- HW4 deadline moved to Sunday, 3/9
- Sign up on google sheet (see Ed)

Outline

- Bloom Filters
- LSM Trees
- Cascades Extensible Optimizer
- Yannakakis' Algorithm

Terminology

Dictionary:

- Insert(k,v) = insert a key,value pair
- Find(k) = retrieve v
- Delete(k) = delete (k,v)
- Key k may be unique or not

Filter:

- Insert(k) = insert k (no value)
- Member(k) = check k; false positives OK!

Slides on Bloom Filters

Based in part on:

- Broder, Andrei; Mitzenmacher, Michael (2005), "Network Applications of Bloom Filters: A Survey", Internet Mathematics 1 (4): 485–509
- Bloom, Burton H. (1970), "Space/time trade-offs in hash coding with allowable errors", Communications of the ACM 13 (7): 422–42

Problem Setting

- Want a *very small*, and *very fast* dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test

Problem Setting

- Want a *very small*, and *very fast* dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test
- False positives are OK
  - find(k,H) = true: k may or may not be in H
  - find(k,H) = false: k is not in H

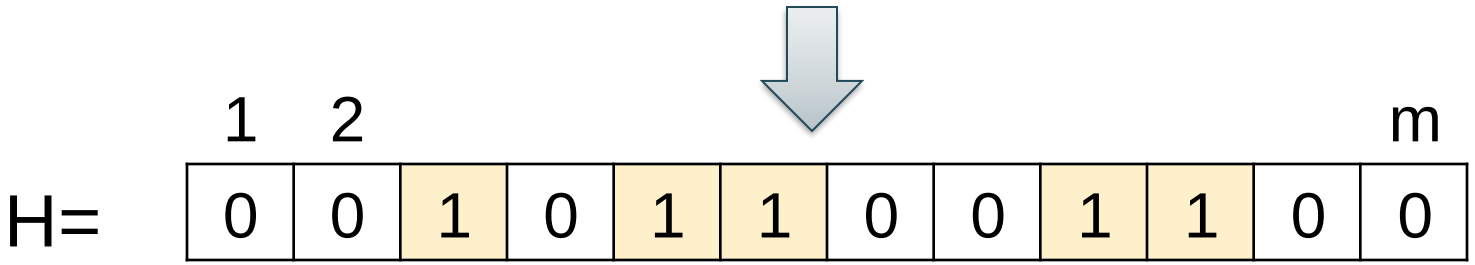
Problem Setting

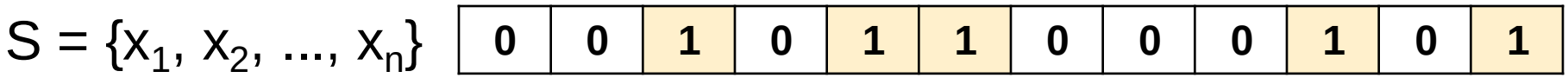
- Want a *very small*, and *very fast* dictionary H
  - Insert(k,H), member(k,H)
  - No values, just membership test
- False positives are OK
  - find(k,H) = true: k may or may not be in H
  - find(k,H) = false: k is not in H
- Goal: minimize false positive rate, FPR

Bit Map

- Let $S = \{x_1, x_2, \ldots, x_n\}$ be a data set
- Hash function h : S → {1, 2, …, m}
  - Typically, m=8n

$$S = \{x_1, x_2, \ldots, x_n\}$$





Bit Map = a Set

- Insert(x, H) = set bit h(x) to 1
  - Collisions are possible
- Member(y, H) = check if bit h(y) is 1
  - False positives are possible
- No deletions

$S = \{x_1, x_2, ..., x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Insert S into H

- Check membership of some y:
  - What is the probability member(y,H)=true?
  - This is the False Positive Rate, FPR

$S = \{x_1, x_2, ..., x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

Analysis

- Insert S into H

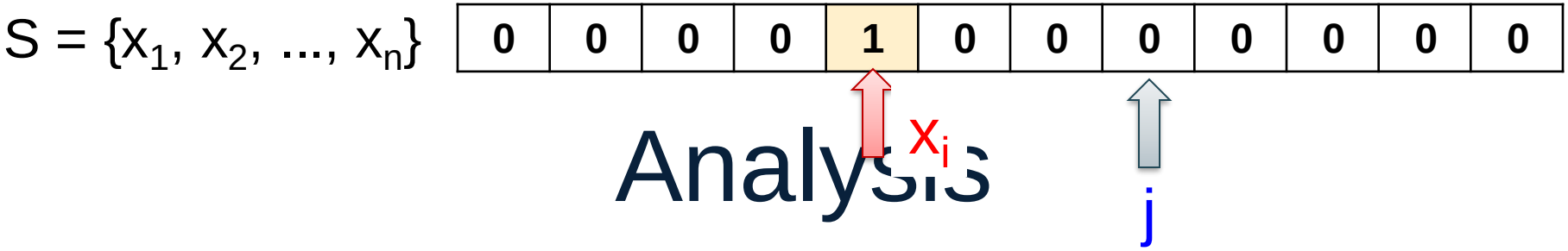
- Check membership of some y:
  - What is the probability member(y,H)=true?
  - This is the False Positive Rate, FPR
- Will compute in two steps
  - Will denote j=h(y)
  - FPR = Prob(bit(j)=1)

$S = \{x_1, x_2, ..., x_n\}$

Analysis

- Recall |H| = m

- What is the probability that bit j is 0 ?

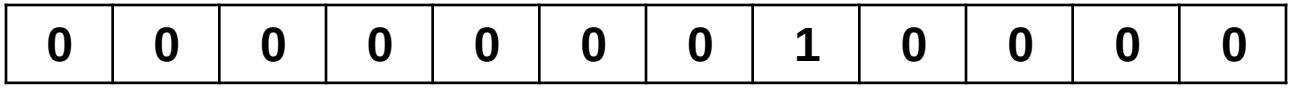


Analysis

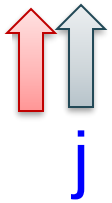
- Recall |H| = m
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, ..., x_n\}$

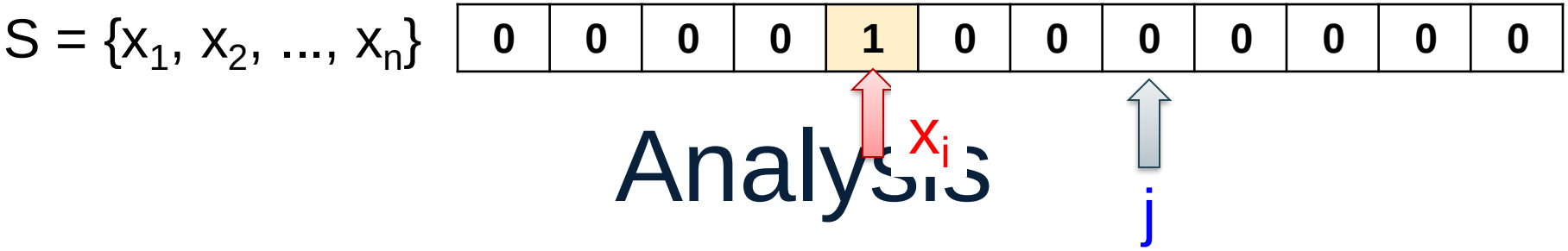


Analysis



- Recall |H| = m
- Let's insert only x_i into H

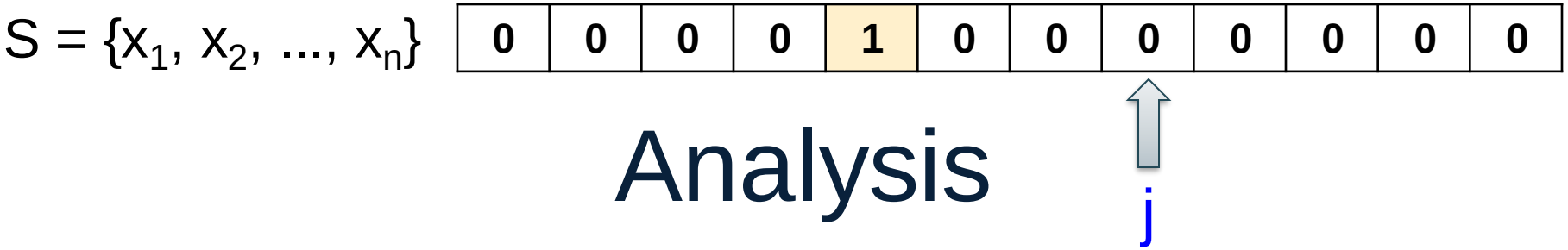
- What is the probability that bit j is 0 ?



Analysis

- Recall |H| = m
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

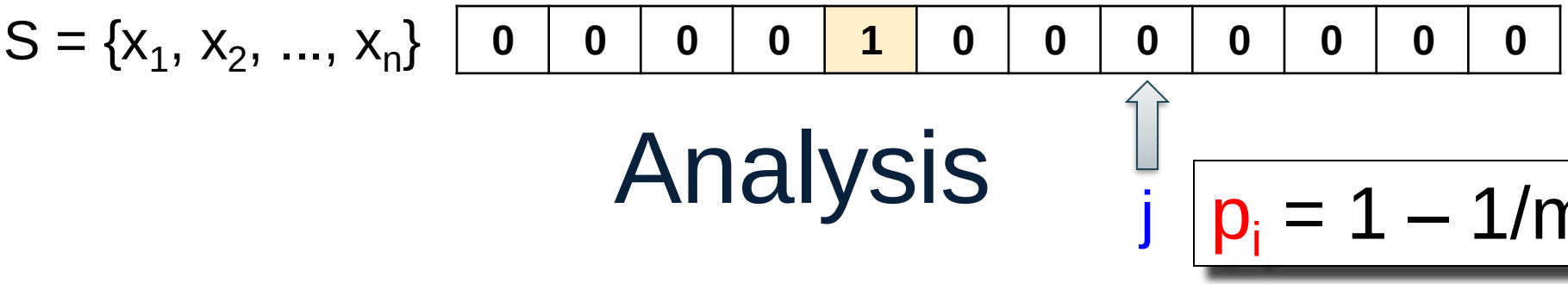


Analysis

- Recall |H| = m
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

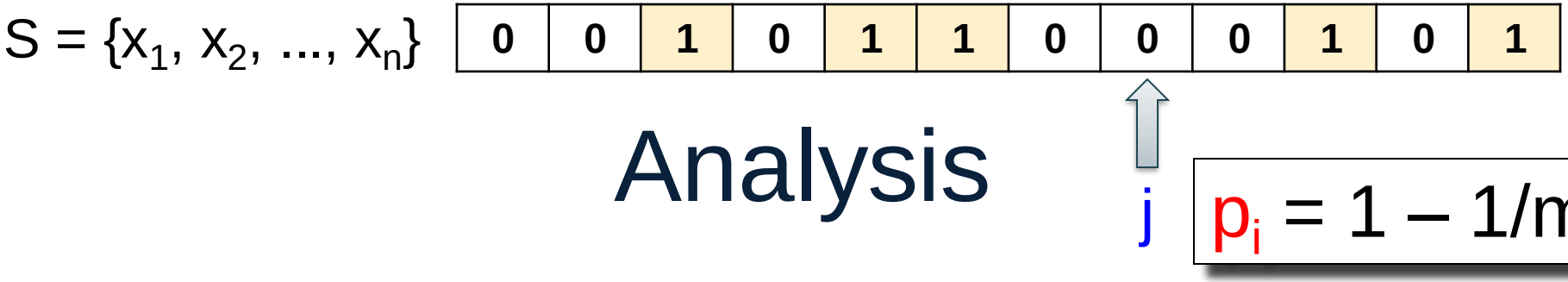
- Answer: $p_i = 1 - 1/m$



Analysis

- Recall |H| = m
- Let's insert only x_i into H

- What is the probability that bit j is 0 ?

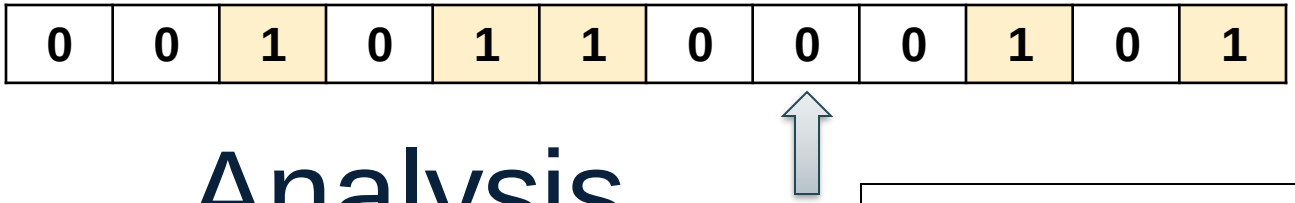


Analysis

- Recall |H| = m
- Let's insert $x_1, x_2, \ldots, x_n$ in H

- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, ..., x_n\}$

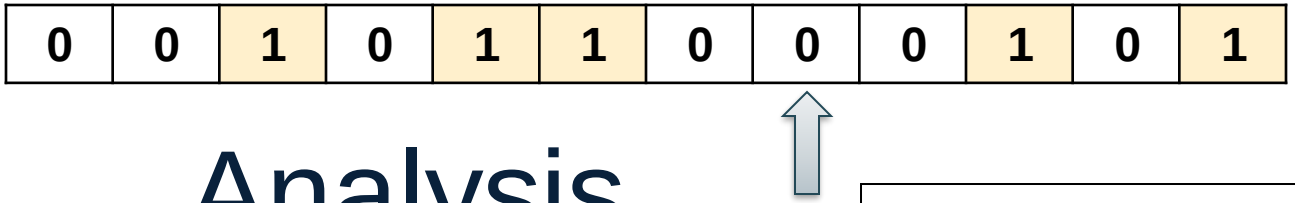


Analysis

$p_i = 1 - 1/m$

- Recall |H| = m
- Let's insert $x_1, x_2, \ldots, x_n$ in H
- What is the probability that bit j is 0 ?
- Answer: $P_j = p_1 p_2 \ldots p_n$

$S = \{x_1, x_2, ..., x_n\}$



Analysis

$p_i = 1 - 1/m$

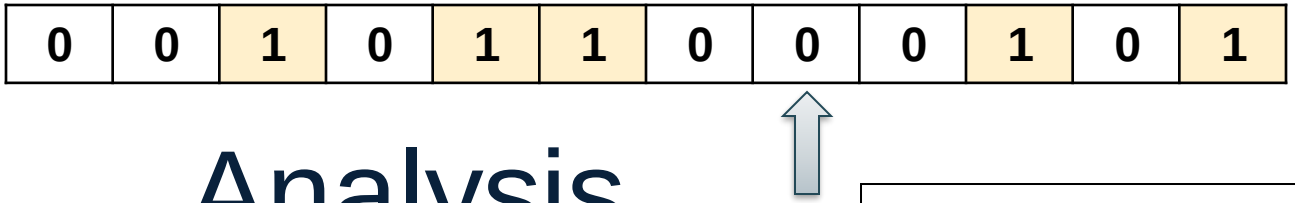
- Recall $|H| = m$
- Let's insert $x_1, x_2, \ldots, x_n$ in H

- What is the probability that bit j is 0 ?

- Answer: $P_j = p_1 p_2 \ldots p_n$

$$P_j = (1 - 1/m)^n$$

$S = \{x_1, x_2, \ldots, x_n\}$



Analysis

$p_i = 1 - 1/m$

- Recall $|H| = m$
- Let's insert $x_1, x_2, \ldots, x_n$ in H
- What is the probability that bit j is 0 ?
- Answer: $P_j = p_1 p_2 \ldots p_n$

1/m very small!

$$P_j = (1 - 1/m)^n \approx e^{-n/m}$$

$S = \{x_1, x_2, ..., x_n\}$

0	0	1	0	1	1	0	0	0	1	0	1

False Positive Rate

- FPR = Prob(member(y,H)=1) is:

$$FPR = 1 - (1 - 1/m)^n \approx 1 - e^{-n/m}$$

$S = \{x_1, x_2, ..., x_n\}$

False Positive Rate

- FPR = Prob(member(y,H)=1) is:

$$FPR = 1 - (1 - 1/m)^n \approx 1 - e^{-n/m}$$

- Example: m = 8n, then

  $FPR \approx 1 - e^{-n/m} = 1 - e^{-1/8} \approx 0.11$

  11% false positive rate

- Bloom filters improve that (next)

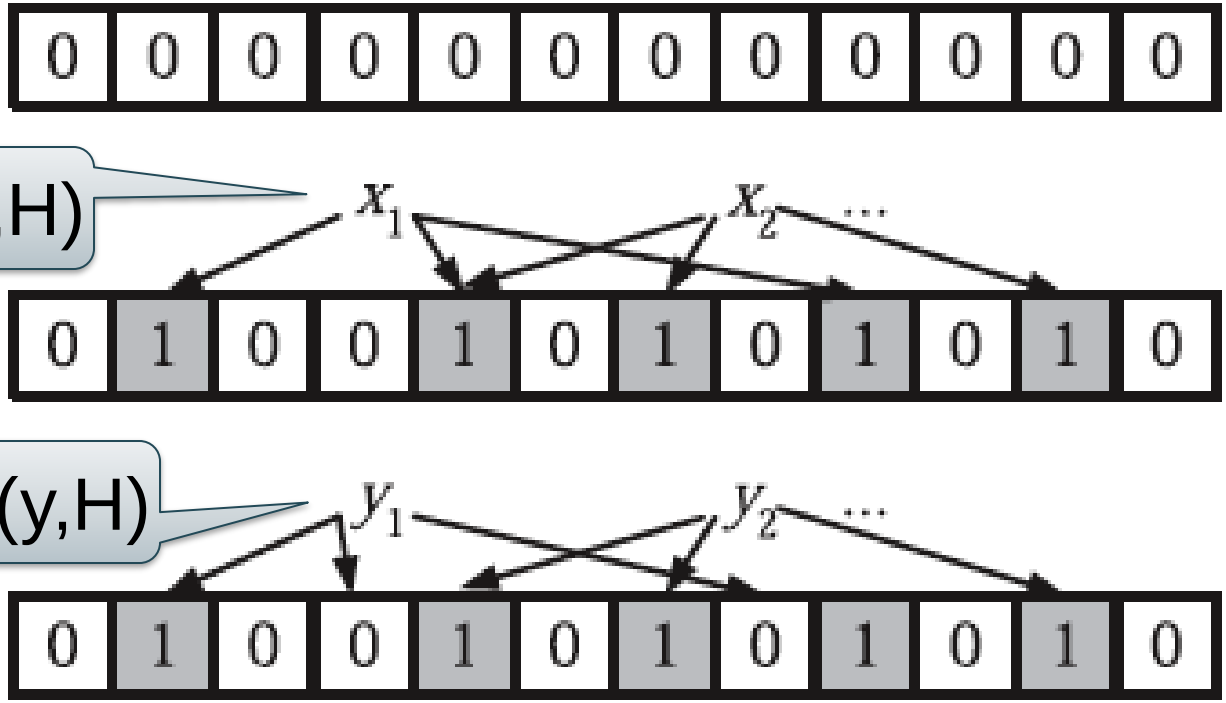
Bloom Filters

- Introduced by Burton Bloom in 1970
- Improve the false positive ratio
- Idea: use k independent hash functions

Bloom Filter = Dictionary

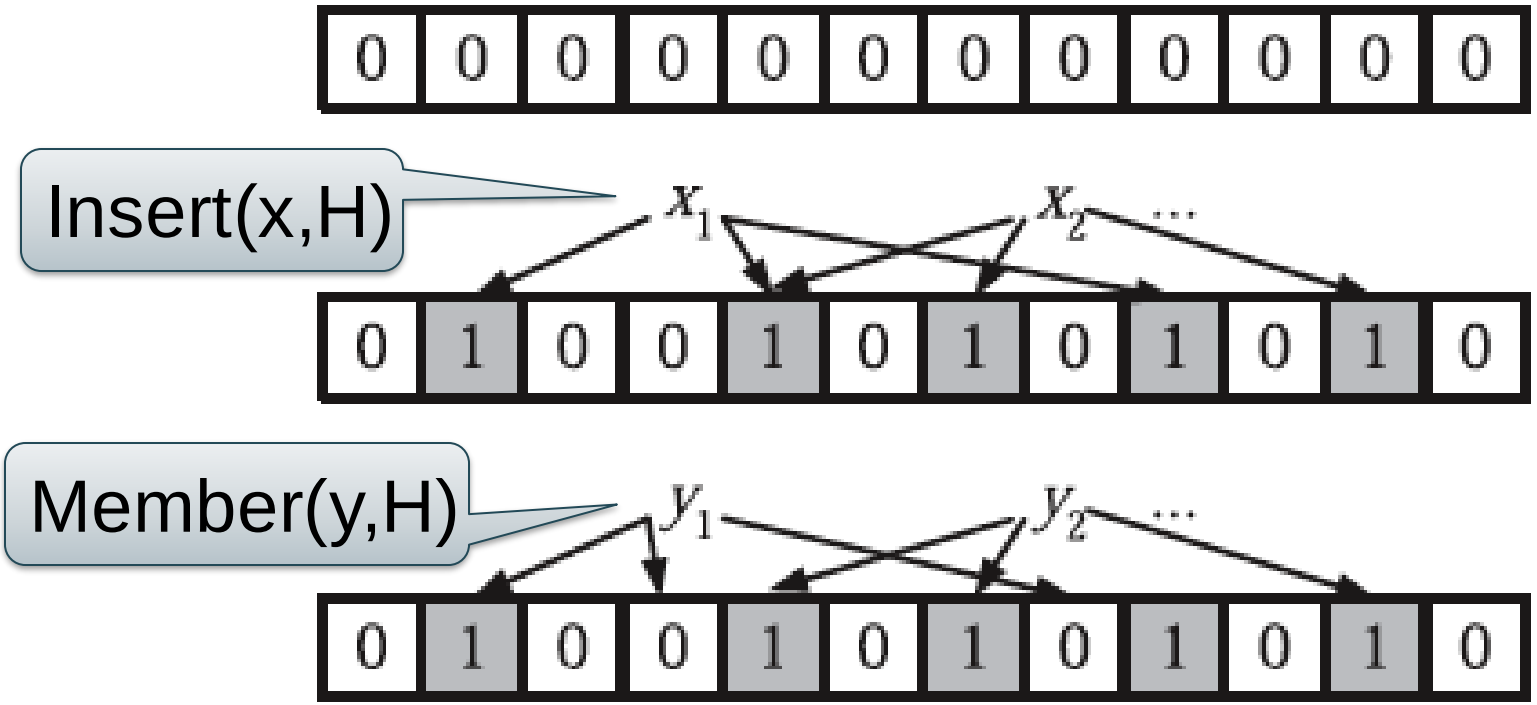
- Insert(x, H):
  - set bits $h_1(x), \ldots, h_k(x)$ to 1
- Member(y, H):
  - check if all bits $h_1(y), \ldots, h_k(y)$ are 1

Example Bloom Filter k=3



y_1 = is not in H (why ?);

Example Bloom Filter k=3



y_1 = is not in H (why ?); y_2 may be in H (why ?)

Choosing k (# hash functions)

Two competing forces:

- If k = large
  - Test more bits for member(y,H) ➔ low FPR
  - More bits in H are 1 ➔ high FPR

Choosing k (# hash functions)

Two competing forces:

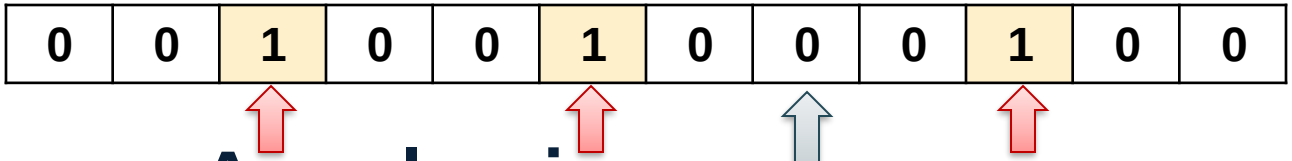
- If k = large
  - Test more bits for member(y,H) ➔ low FPR
  - More bits in H are 1 ➔ high FPR
- If k = small
  - More bits in H are 0 ➔ lower FPR
  - Test fewer bits for member(y,H) ➔ high FPR

Choosing k (# hash functions)

Two competing forces:

- If k = large
  - Test more bits for member(y,H) ➔ low FPR
  - More bits in H are 1 ➔ high FPR
- If k = small
  - More bits in H are 0 ➔ lower FPR
  - Test fewer bits for member(y,H) ➔ high FPR
- How do we choose k?
  Is FPR better than for a simple hash table?

$S = \{x_1, x_2, ..., x_n\}$



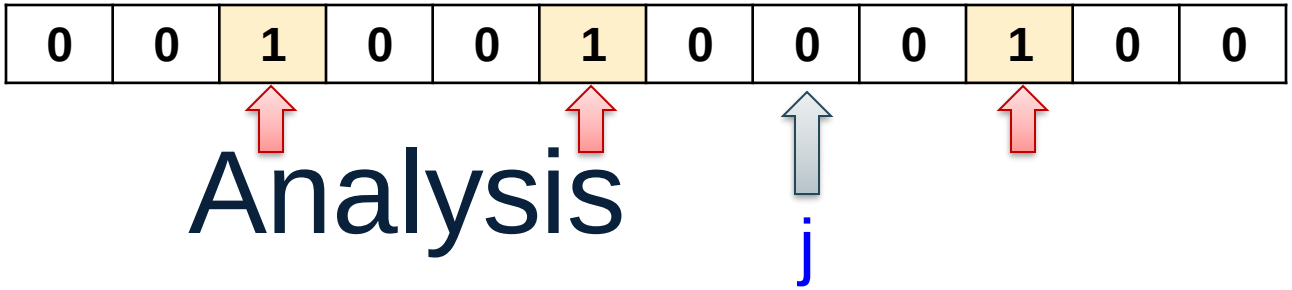
Analysis

j

- Recall |H| = m, #hash functions = k
- Let's insert only x_i into H

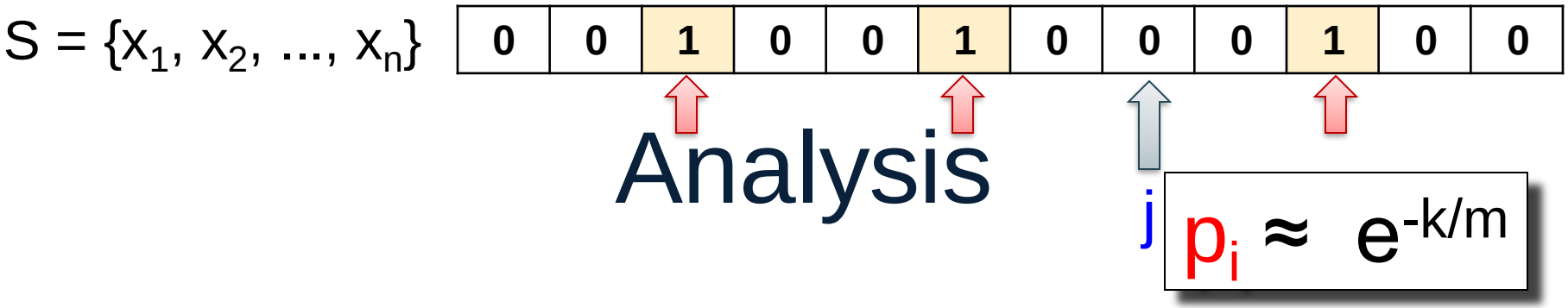
- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, ..., x_n\}$



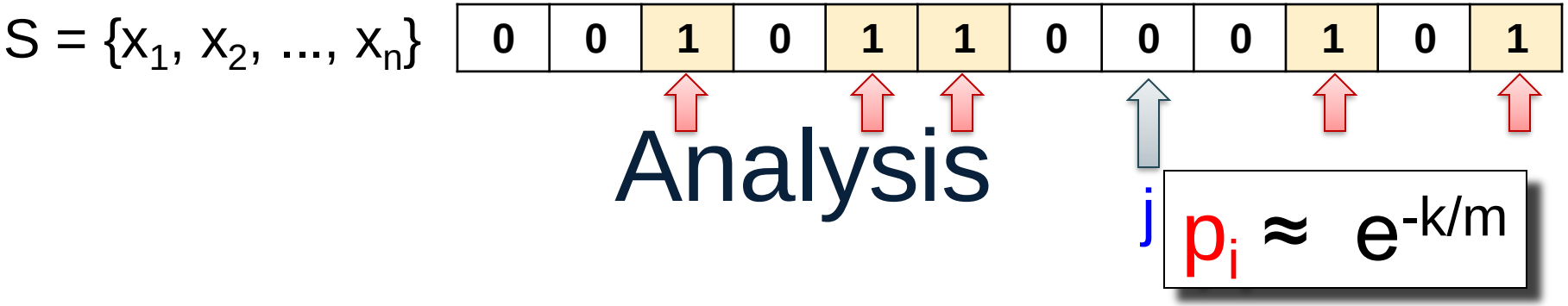
Analysis

- Recall |H| = m, #hash functions = k
- Let's insert only x_i into H
- What is the probability that bit j is 0 ?
- Answer: $p_i = (1 - 1/m)^k \approx e^{-k/m}$



- Recall |H| = m, #hash functions = k
- Let's insert only x_i into H

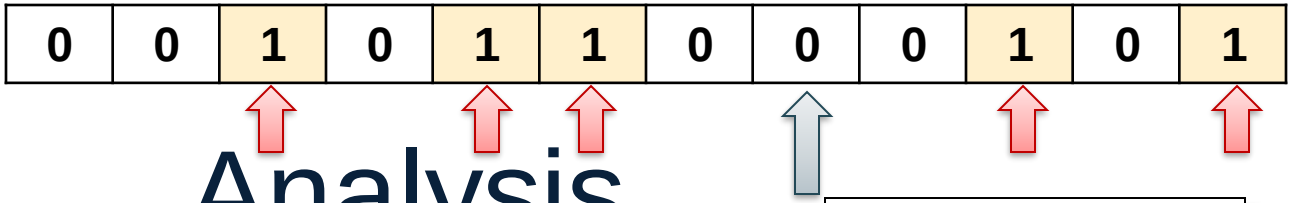
- What is the probability that bit j is 0 ?



- Recall |H| = m, #hash functions = k
- Let's insert $x_1, x_2, \ldots, x_n$ in H

- What is the probability that bit j is 0 ?

$S = \{x_1, x_2, ..., x_n\}$



Analysis

$p_i \approx e^{-k/m}$

- Recall |H| = m, #hash functions = k
- Let's insert $x_1, x_2, \ldots, x_n$ in H
- What is the probability that bit j is 0 ?
- Answer: $P_j = p_1 p_2 \ldots p_n \approx e^{-kn/m}$

$P_j \approx e^{-kn/m}$

False Positive Rate

- FPR is the probability that member(y,H)=true

$P_j \approx e^{-kn/m}$

False Positive Rate

- FPR is the probability that member(y,H)=true

- Happens when bits $h_1(y), \ldots, h_k(y)$ are 1

  FPR= $(1-P_{j1}) * (1-P_{j2}) * \ldots * (1-P_{jk})$

$P_j \approx e^{-kn/m}$

False Positive Rate

- FPR is the probability that member(y,H)=true

- Happens when bits $h_1(y), \ldots, h_k(y)$ are 1

  FPR= $(1-P_{j1}) * (1-P_{j2}) * \ldots * (1-P_{jk})$

$$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items ⇧ FPR ?
- m = size of hash table(bits)⇧ FPR ?
- k = number of hash functions

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items ⇧ FPR⇧
- m = size of hash table(bits)⇧ FPR ?
- k = number of hash functions

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items ⇧ FPR⇧
- m = size of hash table(bits)⇧ FPR⇩
- k = number of hash functions

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items ⇧ FPR⇧
- m = size of hash table(bits)⇧ FPR⇩
- k = number of hash functions ??

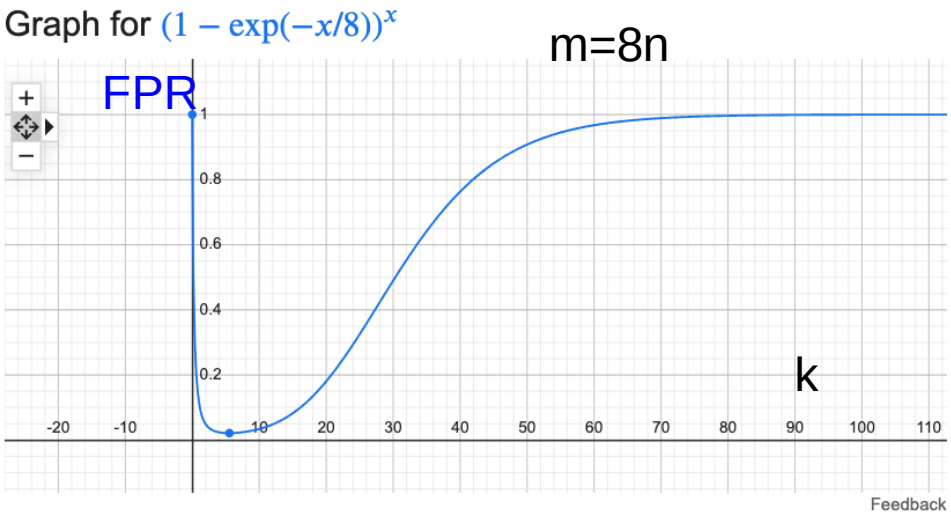
$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items FPR⇧
- m = size of hash table (bits) FPR⇩
- k = number of hash functions ??



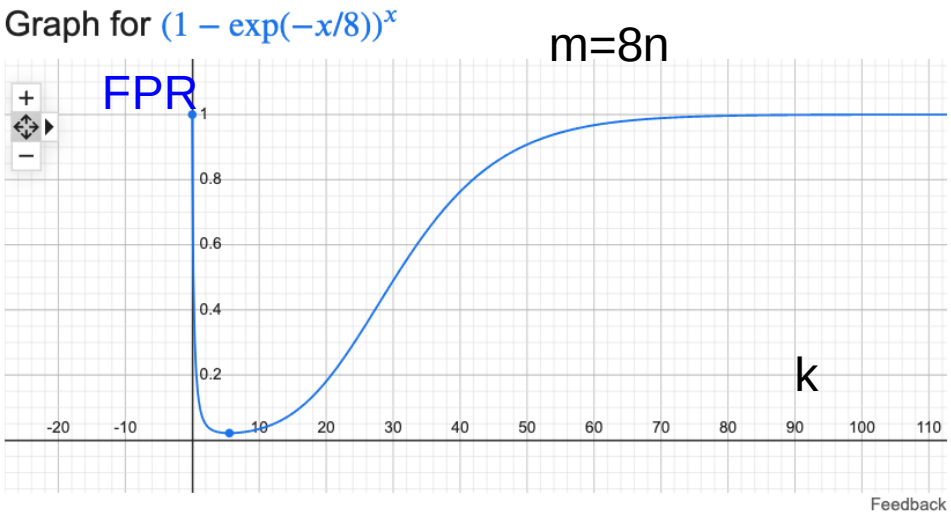
$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

How does FPR change if we increase:

- n = number of data items FPR⇧
- m = size of hash table (bits) FPR⇩
- k = number of hash functions ??



A little math leads to:

$k_{opt} = \ln 2 \times m/n$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Understanding the FPR

Proof of k_{opt}:

$ln(\text{FPR}) = k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right) = \frac{m}{n} \cdot \frac{kn}{m} \ln\left(1 - e^{-\frac{kn}{m}}\right) = -\frac{m}{n} \ln x \cdot \ln(1-x)$, where $x = e^{-\frac{kn}{m}}$.

We need to maximize the function $\text{g(x)} = \ln x \cdot \ln(1-x)$

$g'(x) = \frac{\ln(1-x)}{x} - \frac{\ln x}{1-x}$. Then notice:

1. $g'(x) = 0$ when $x = \frac{1}{2}$,
2. $g'(x)$ is decreasing

Hence, $g(x)$ takes the maximum value at $x = \frac{1}{2}$. Thus $x = e^{-\frac{kn}{m}} = \frac{1}{2}$ and $k = \frac{m}{n} \ln 2$

A little math leads to:

$k_{opt} = \ln 2 \times m/n$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m/n$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose k = ln 2 × m /n

Probability that bit j is 1

$1-P_j \approx e^{-kn/m} = ½$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m/n$

Probability that bit j is 1

$1-P_j \approx e^{-kn/m} = ½$

When k is optimal, then table is half full, and:

$FRP = (½)^{(\ln 2)m/n} \approx (0.6185)^{m/n}$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose $k = \ln 2 \times m/n$

Probability that bit j is 1

$1-P_j \approx e^{-kn/m} = ½$

When k is optimal, then table is half full, and:

$FRP = (½)^{(\ln 2)m/n} \approx (0.6185)^{m/n}$

A hack! $1-P_j \approx ½ \approx P_j$

$P_j \approx e^{-kn/m}$

$FPR = (1-P_j)^k \approx (1 - e^{-kn/m})^k$

Bloom Filter Summary

m, n are fixed → choose k = ln 2 × m /n

Probability that bit j is 1

$1-P_j \approx e^{-kn/m} = ½$

When k is optimal, then table is half full, and:

$FRP = (½)^{(\ln 2)m/n} \approx (0.6185)^{m/n}$

A hack! $1-P_j \approx ½ \approx P_j$

$$FPR = (1 - Pj)^k \approx P_j^k = e^{-\frac{k^2 n}{m}} = e^{-\frac{m}{n}\ln^2 2}$$

Discussion

- Always optimize k
- Must know data size n
- Hash table size: m = n*k / ln 2
- FPR = $(0.6185)^{m/n}$, or $e^{-\frac{m}{n}\ln^2 2}$
  - m/n=8 bits/item: FPR = $(0.6185)^8 \approx 0.02$
- Compare to hash table: $1 - e^{-n/m}$
  - m/n=8 then FPR ≈ 0.11

Outline

- Bloom Filters
- LSM Trees

Monkey: Optimal Navigable Key-Value Store, Dayan, Athanassoulis, Idreos, SIGMOD'2017

- Cascades Extensible Optimizer
- Yannakakis' Algorithm

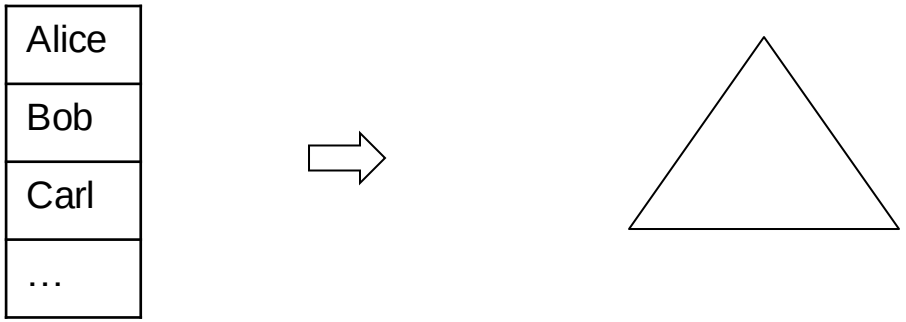
Motivation

- Sorted arrays = best for reads
- Unsorted log file = best for writes
- B+ trees = good for read, so-so for write

- LSM trees = optimize the writes
- Notice:
  - Primary (clustered) index only
  - Key/value stores, but also relational DBs

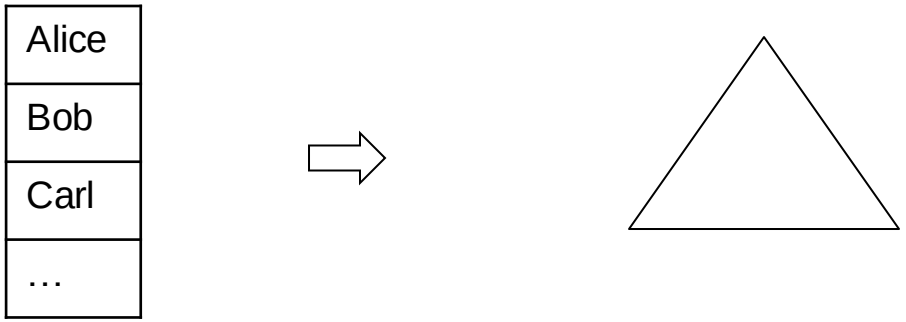
More Motivation

Index for one attribute:
Person.name

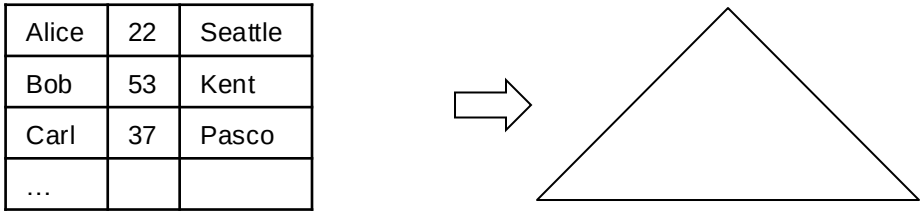


More Motivation

Index for one attribute:
Person.name

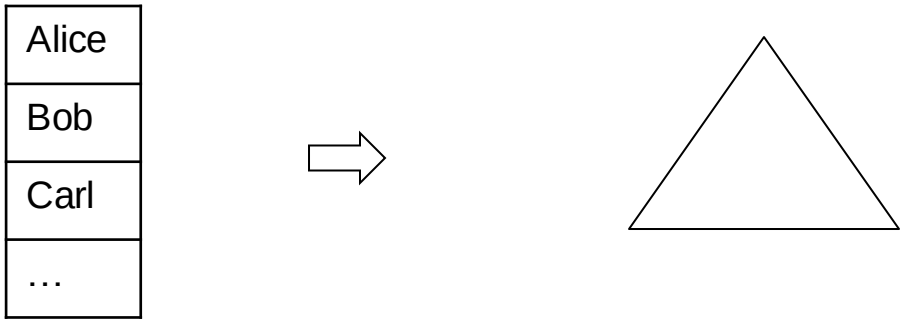


Index for entire table:
Person(name,age,city)

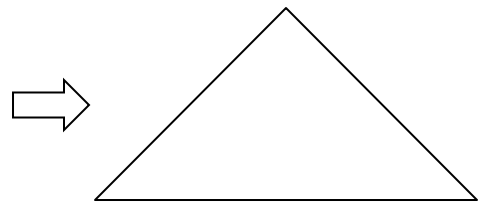


More Motivation

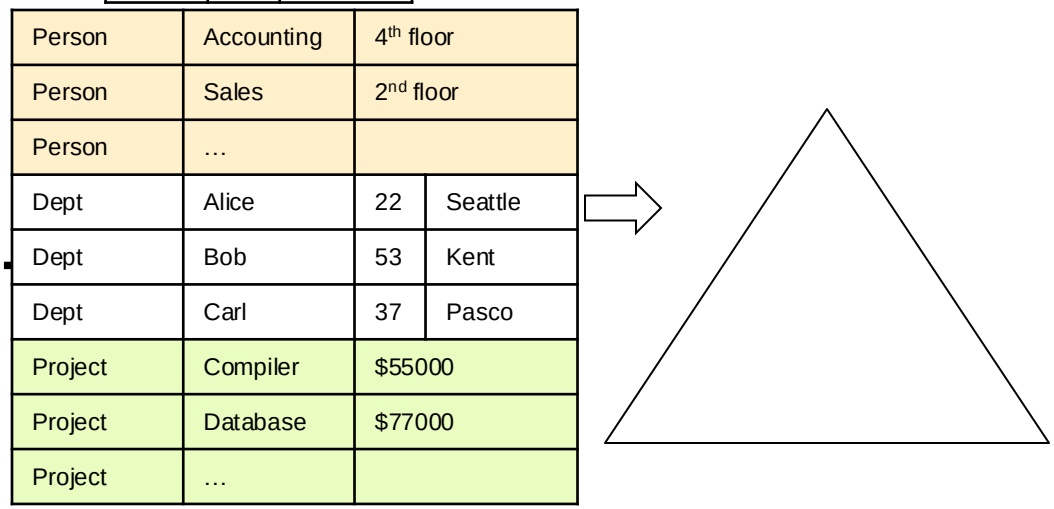
Index for one attribute:
Person.name



Index for entire table:
Person(name,age,city)

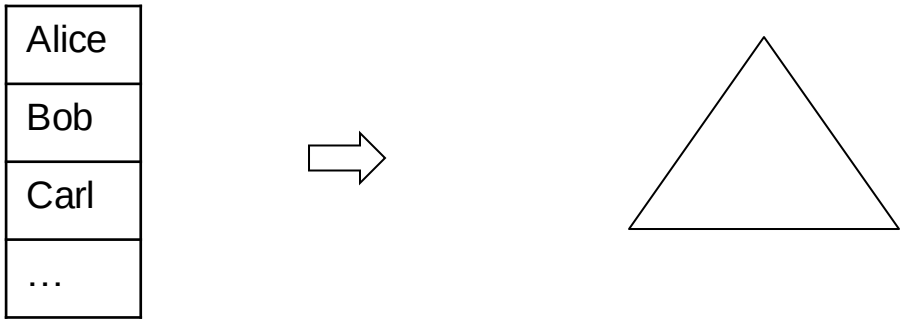


Index for entire db:
Person, Dept, Project,…

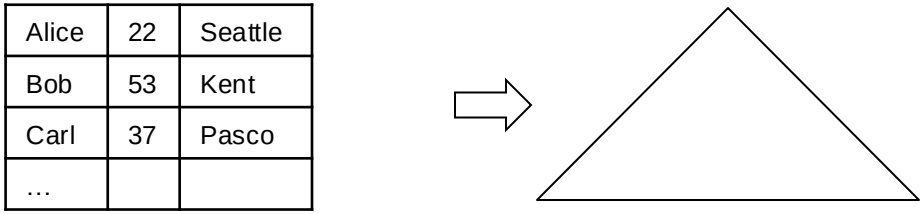


More Motivation

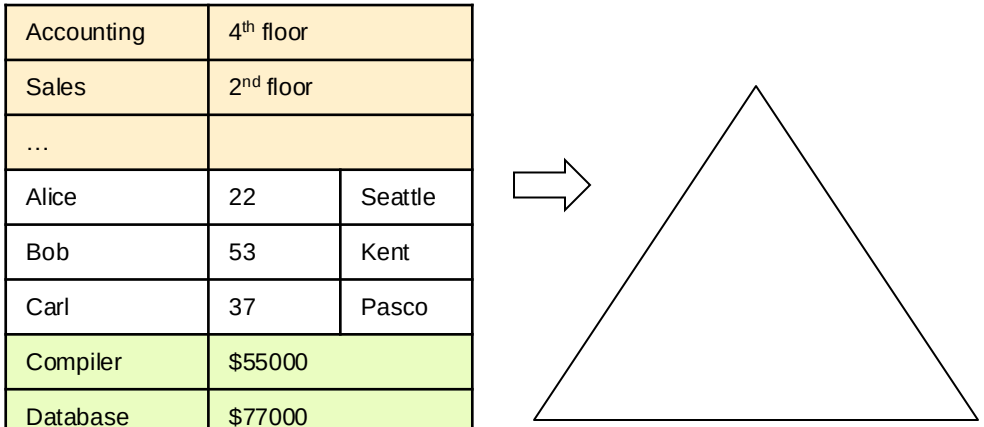
Index for one attribute:
Person.name



Index for entire table:
Person(name,age,city)



Index for entire db:
Person, Dept, Project,…



E.g. MySQL
on RocksDB
using MyRocks

Three Main Ideas for Writes

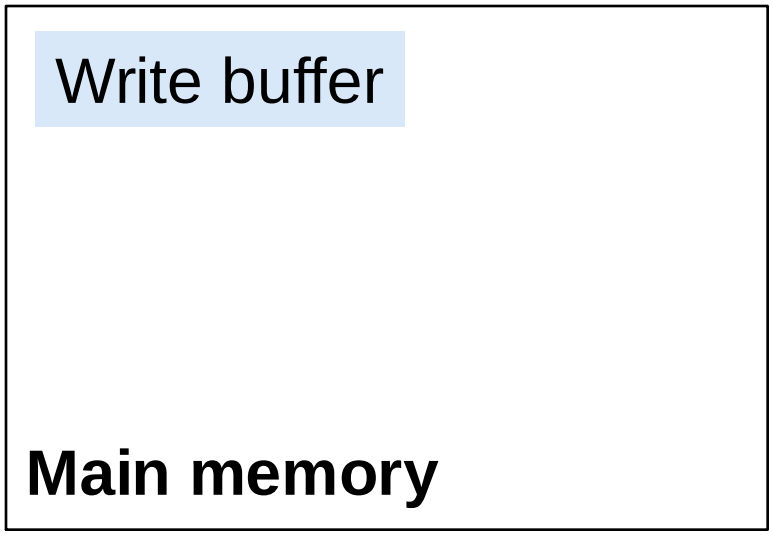
1. Store writes in a buffer in main memory
   When full: spill to disk

2. Spilling to disk (instead of a B+ tree):
   Sort and write to a sorted file.

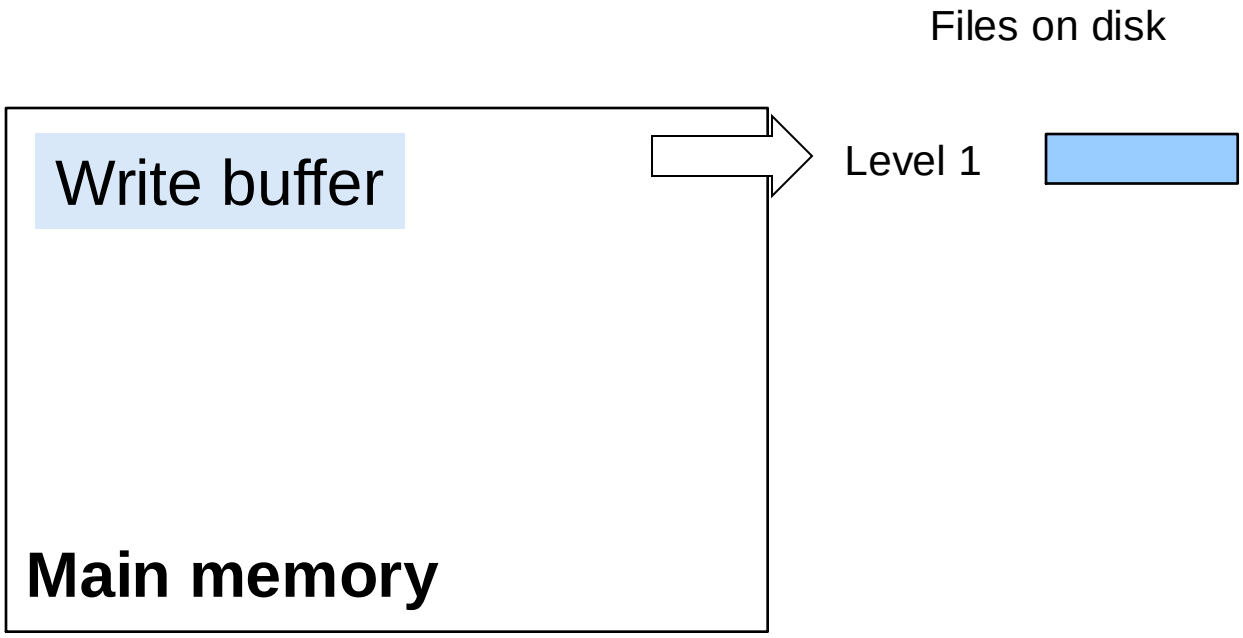
3. When too many sorted files:
   Merge them to a larger sorted file

LSM Trees

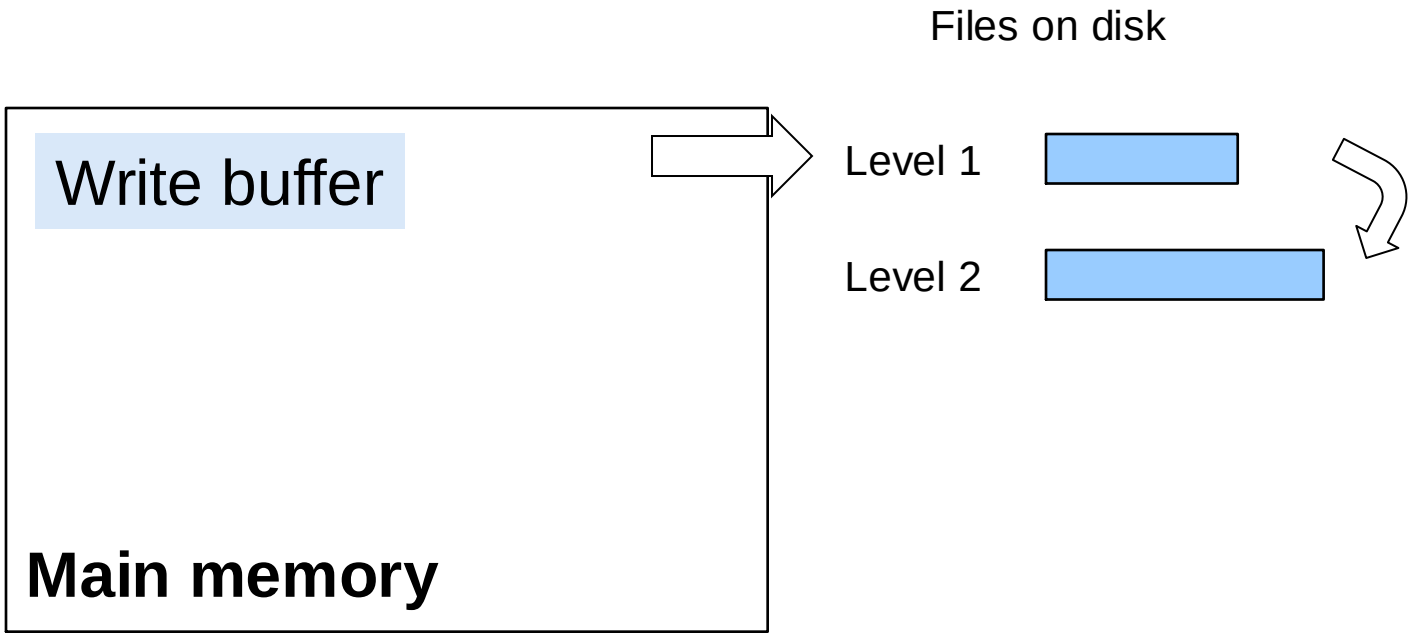
Files on disk



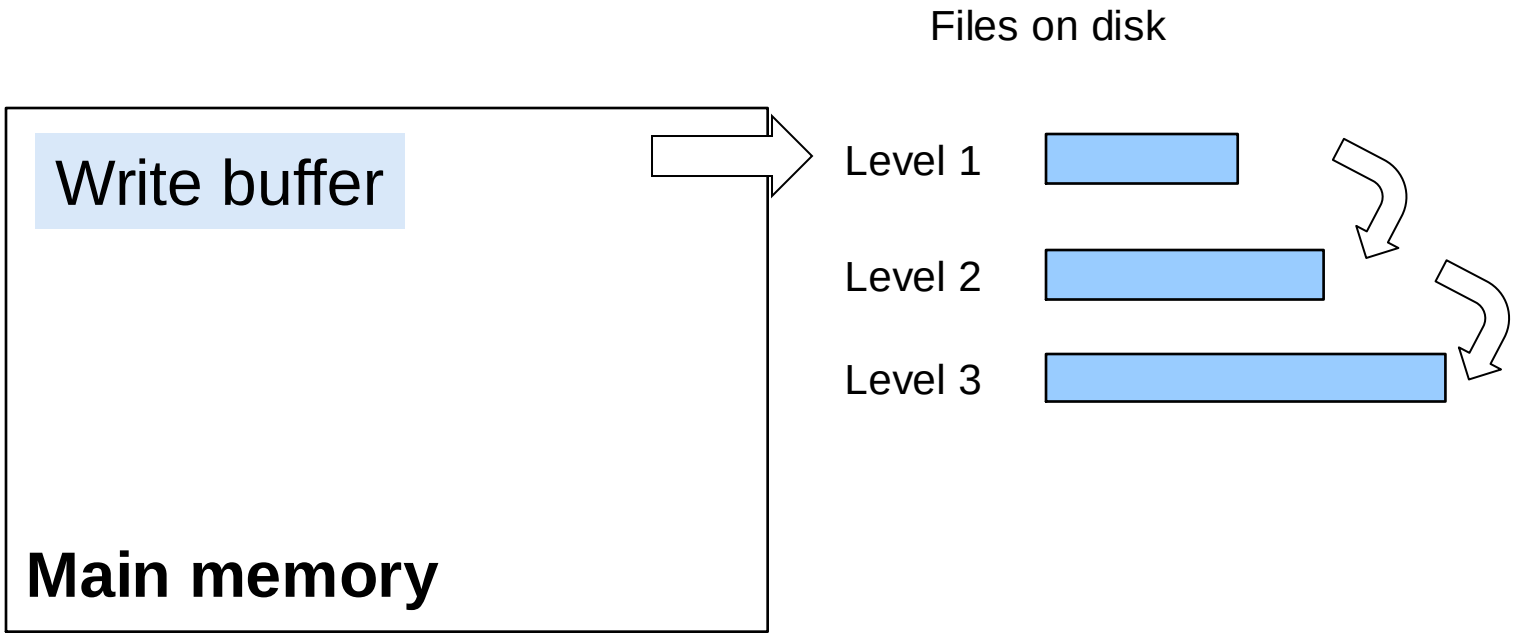
LSM Trees



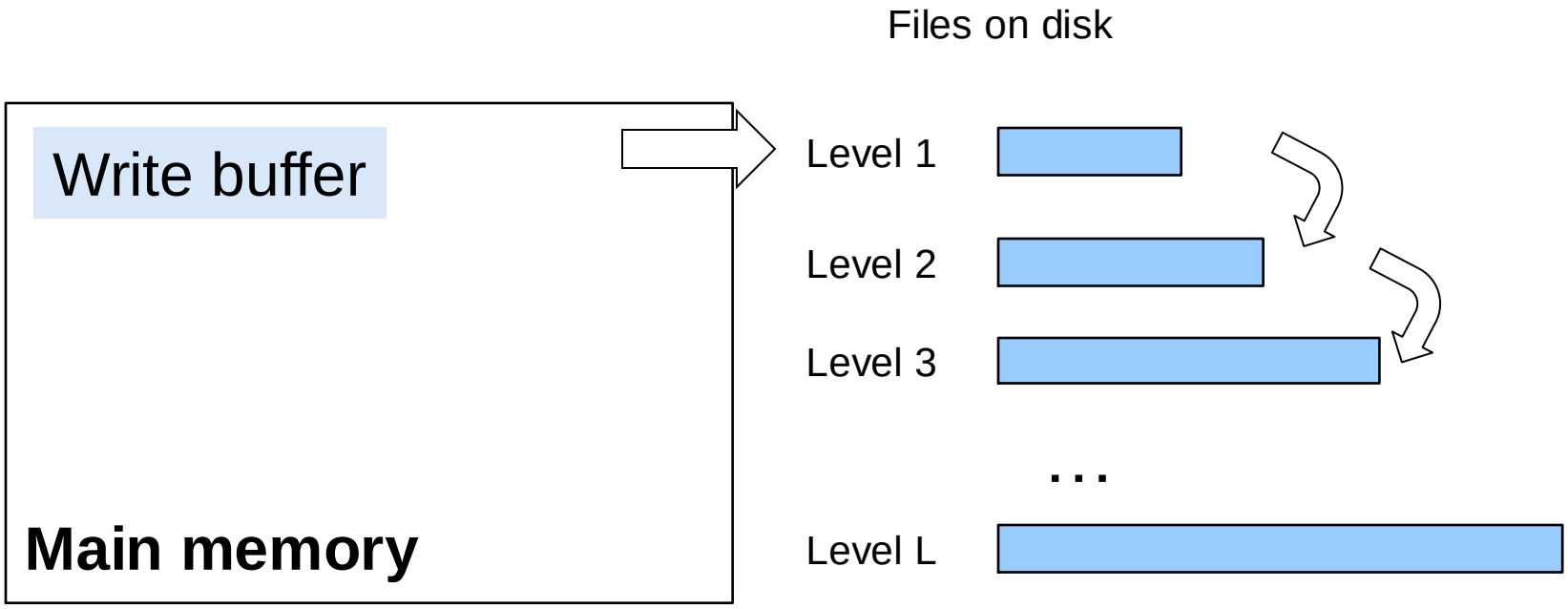
LSM Trees



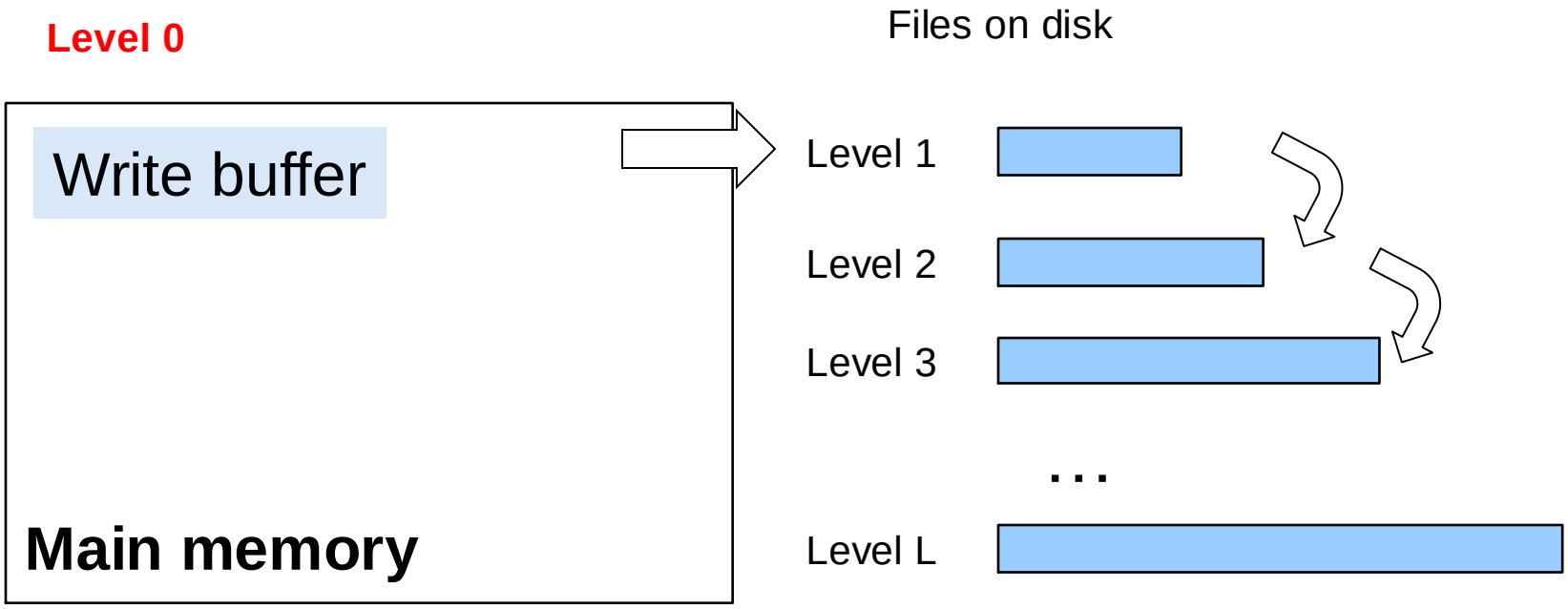
LSM Trees


Files on disk
Write buffer
Level 1
Level 2
Level 3
Main memory

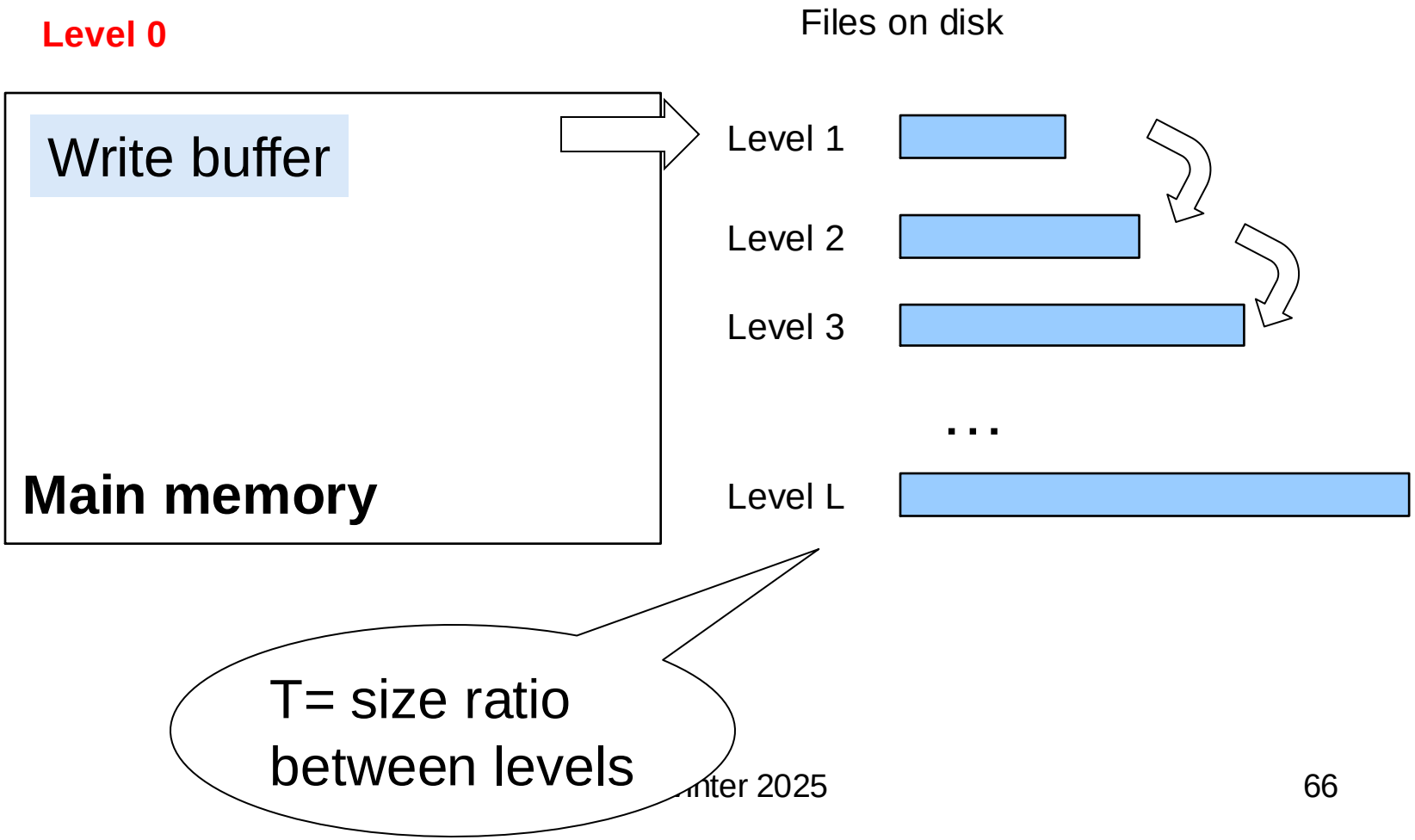
LSM Trees


Files on disk
Write buffer
Main memory
Level 1
Level 2
Level 3
...
Level L

LSM Trees


Level 0
Files on disk
Write buffer
Main memory
Level 1
Level 2
Level 3
...
Level L

LSM Trees


Level 0
Files on disk
Write buffer
Main memory
Level 1
Level 2
Level 3
...
Level L
T= size ratio
between levels

Discussion

- Spilling to next level is a bulk operation; inserts a large number of values
- Better amortized cost than inserting those values one by one into a B+ tree
- Typically done by offline process

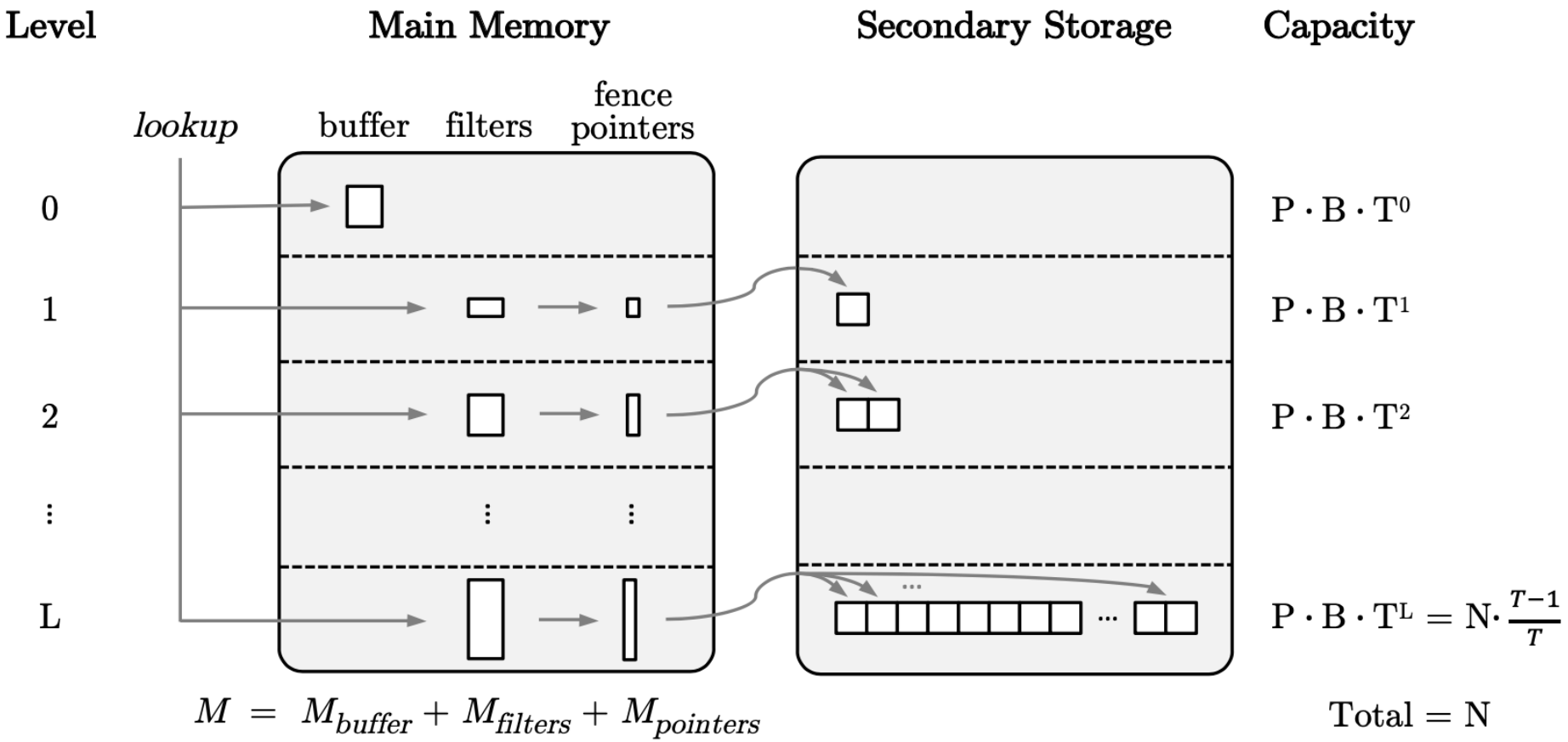
Read

- To read a key, we need to search it at all levels
- Cost is worse than B+ tree
- Three ideas to speedup reads (next)

Three Main Ideas for Reads

1. Bloom filter for each level
2. Fenceposts in main memory for each level
3. Read single block for each level, do binary search

Reading


Level
Main Memory
Secondary Storage
Capacity
lookup
buffer
filters
fence pointers
0
1
2
⋮
L
$P \cdot B \cdot T^0$
$P \cdot B \cdot T^1$
$P \cdot B \cdot T^2$
$P \cdot B \cdot T^L = N \cdot \frac{T-1}{T}$
$M = M_{buffer} + M_{filters} + M_{pointers}$
Total = N

Updates, Deletes

- Never!
- Instead, invalidate the record, and insert a new record if update

Next

- How do we optimize the main memory:
  - Write buffer
  - Bloom filters
  - Fence pointers
- Merge policy
  - Tiering or
  - Leveling

FPR= $e^{-\frac{m}{n}(\ln^2 2)}$, n= #items at given level

Optimizing Bloom Filters

Most memory used by Bloom filters

- Common practice:
  - Ensure the same FPR for all levels
  - FPR constant, space m increases by factor T

FPR= $e^{-\frac{m}{n}(\ln^2 2)}$, n= #items at given level

Optimizing Bloom Filters

Most memory used by Bloom filters

- Common practice:
  - Ensure the same FPR for all levels
  - FPR constant, space m increases by factor T
- Paper observes:
  - Cost per level is the same: reading 1 block
  - Space increases but benefit is constant!
  - Keep space constant, FPR increases by factor T

Merge Policy

- Tiering (write optimized)
  - Flush main memory buffer sorted to disk
  - Accumulate multiple sorted files/level
  - When more than T sorted files: merge them and add 1 file to the next level

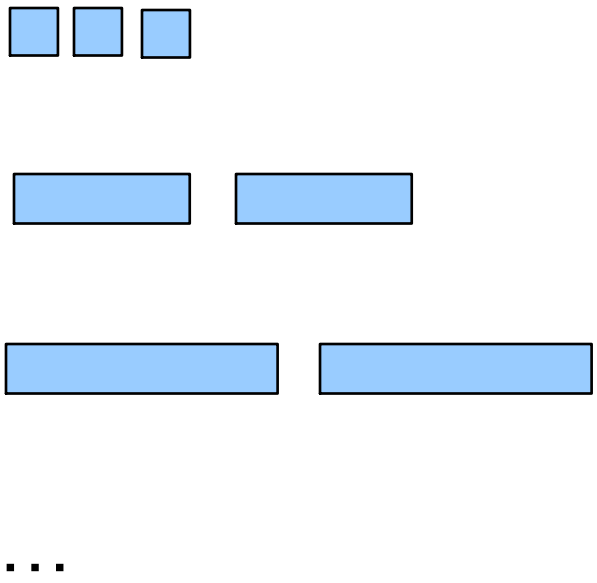
Merge Policy

- Tiering (write optimized)
  - Flush main memory buffer sorted to disk
  - Accumulate multiple sorted files/level
  - When more than T sorted files: merge them and add 1 file to the next level
- Leveling (read-optimized)
  - Merge-sort main memory with level 1
  - When a level becomes too large, move it to the next level by sorting

Size Ratio: T = 3

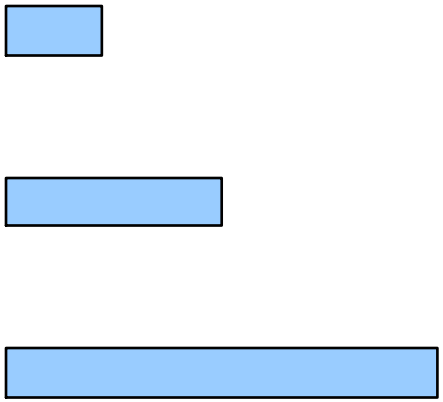
Merge Policies

Tiering



...

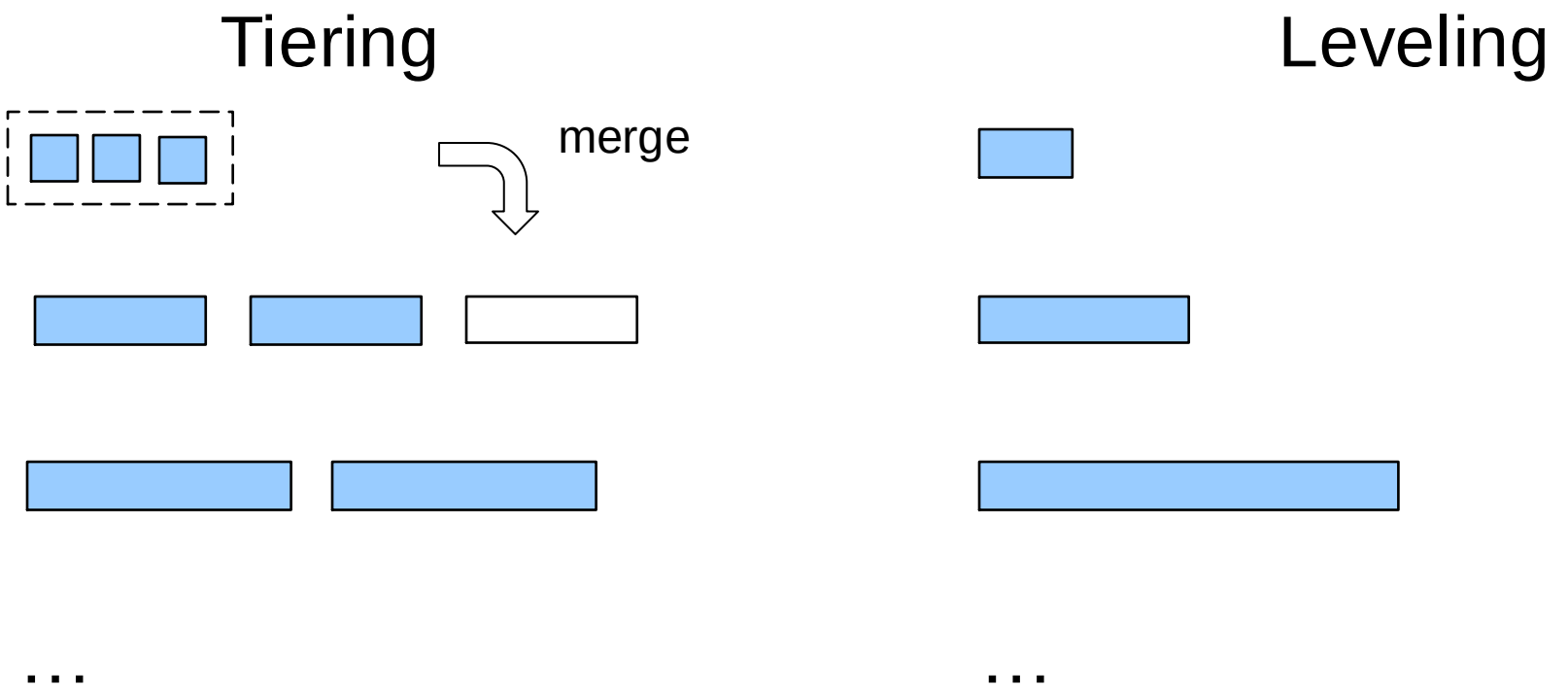
Leveling



...

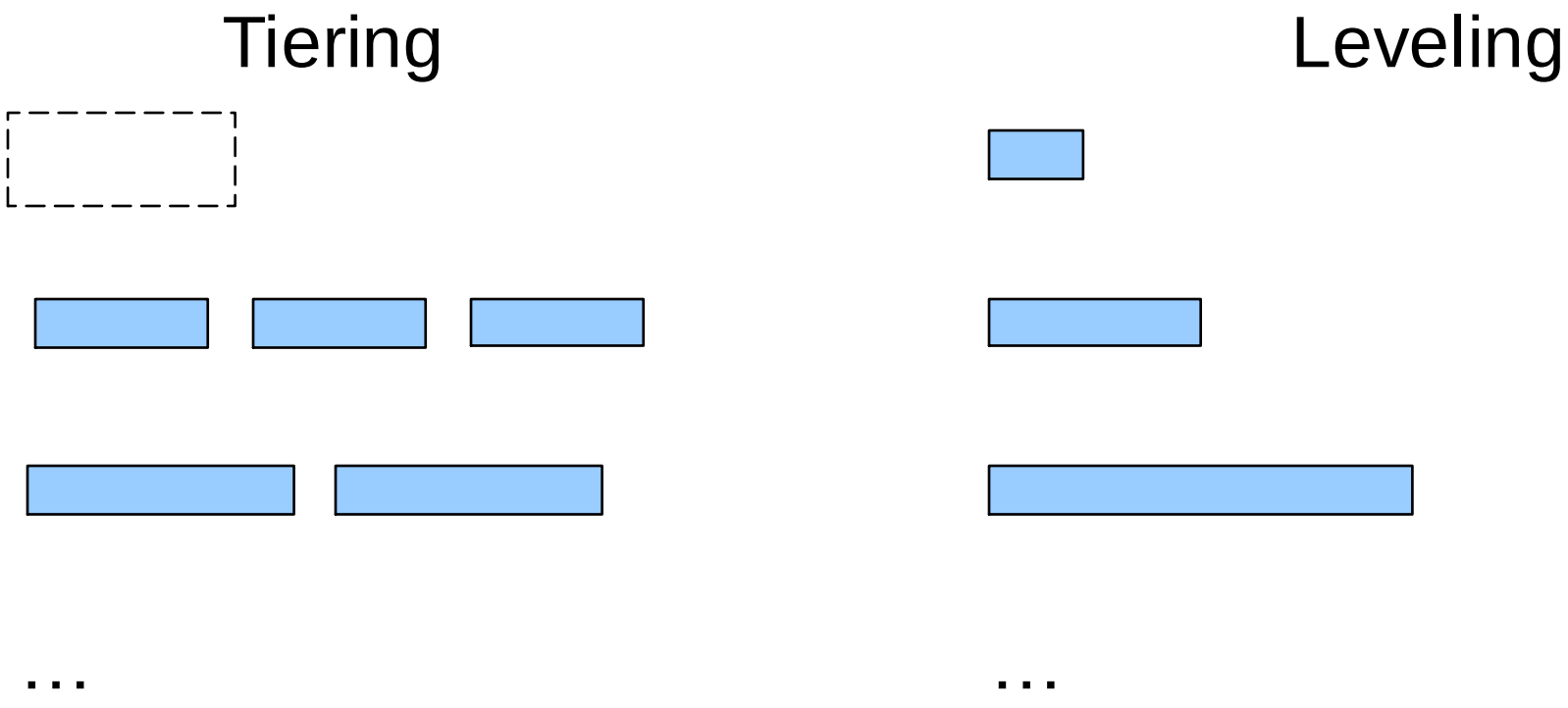
Size Ratio: T = 3

Merge Policies



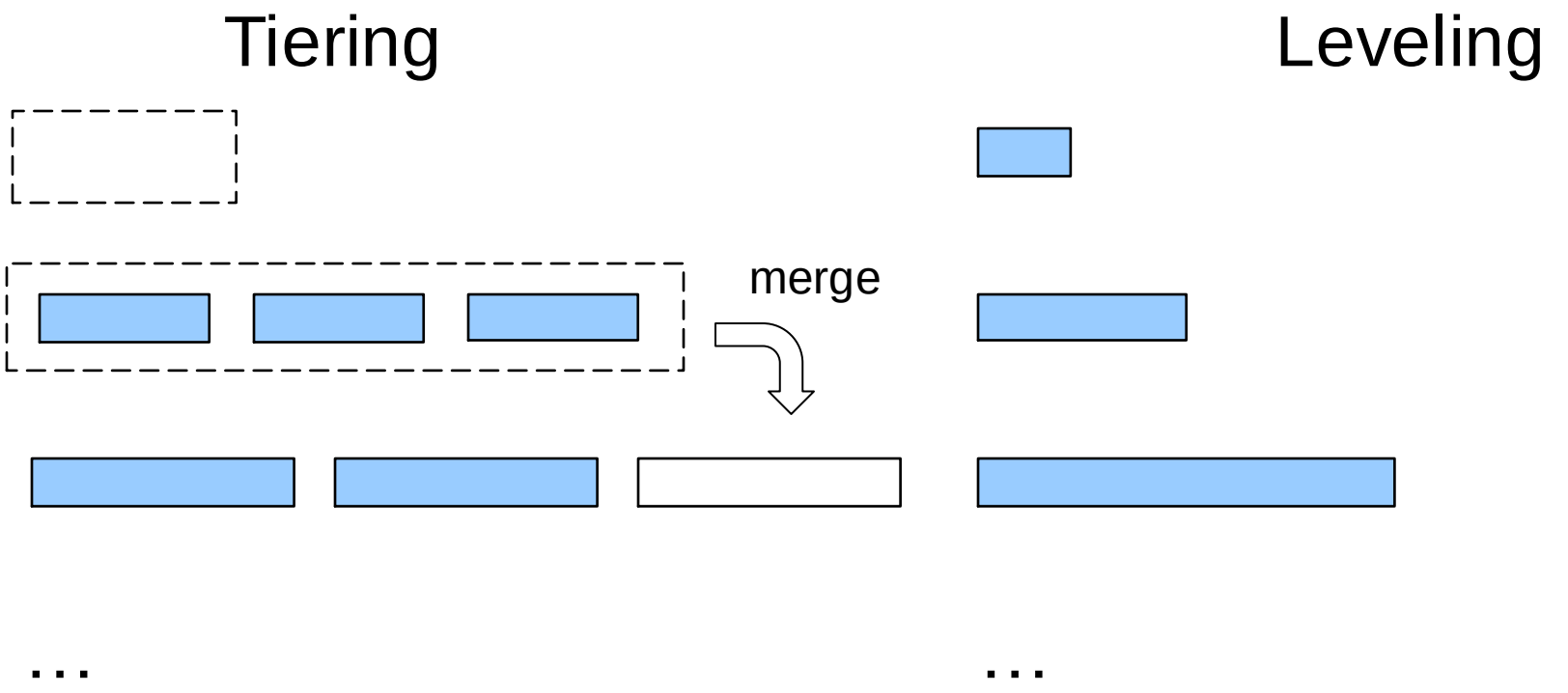
Size Ratio: T = 3

Merge Policies



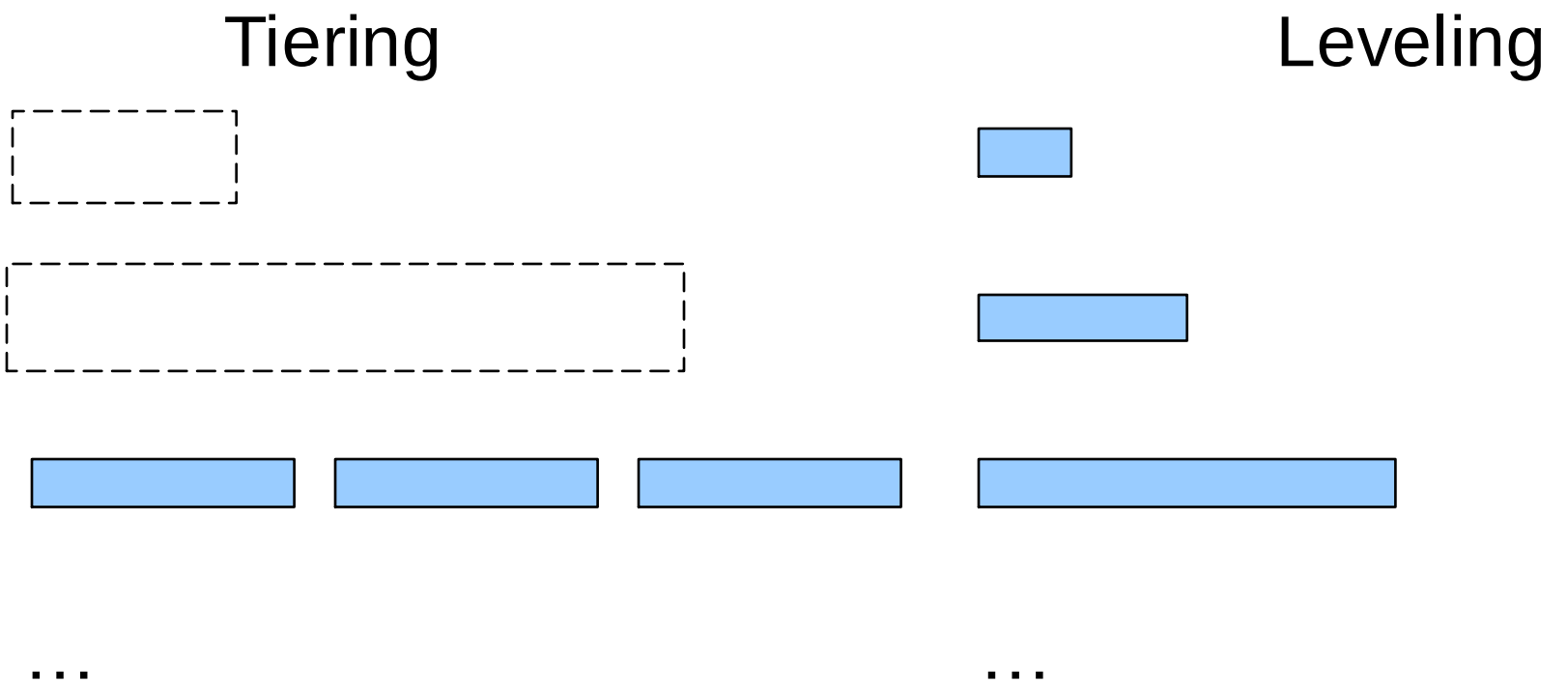
Size Ratio: T = 3

Merge Policies



Size Ratio: T = 3

Merge Policies

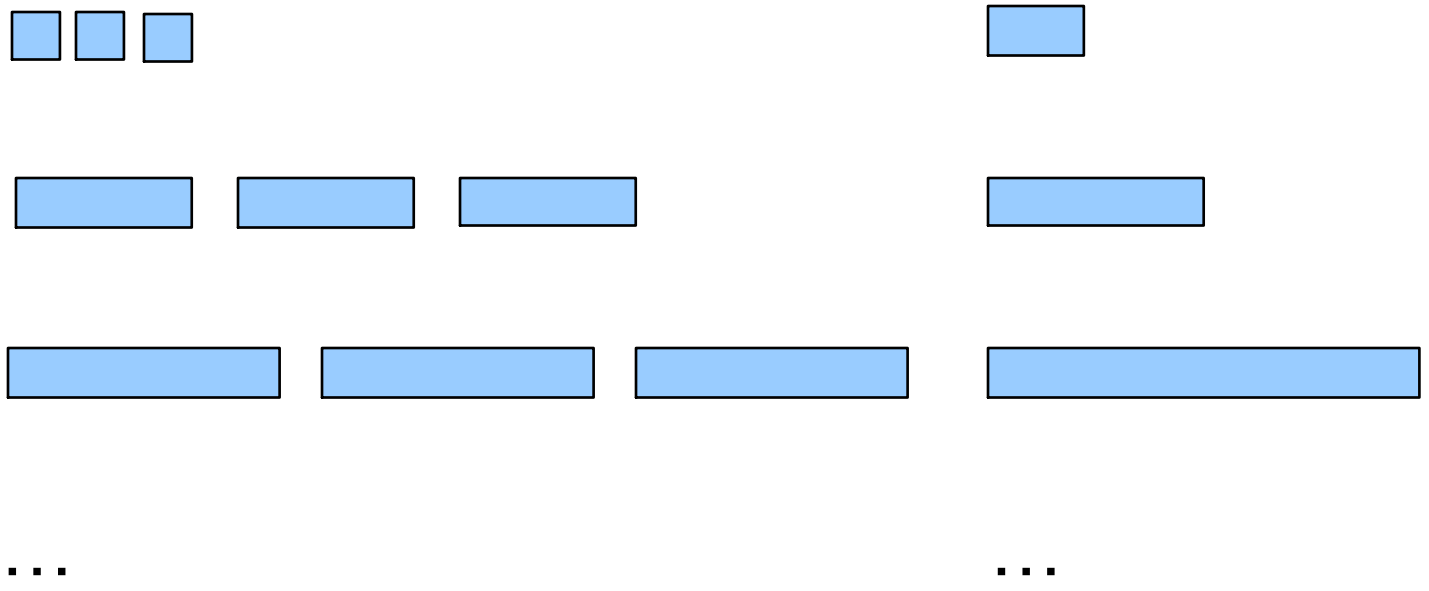


Size Ratio: T = 3

Merge Policies

Tiering

Leveling

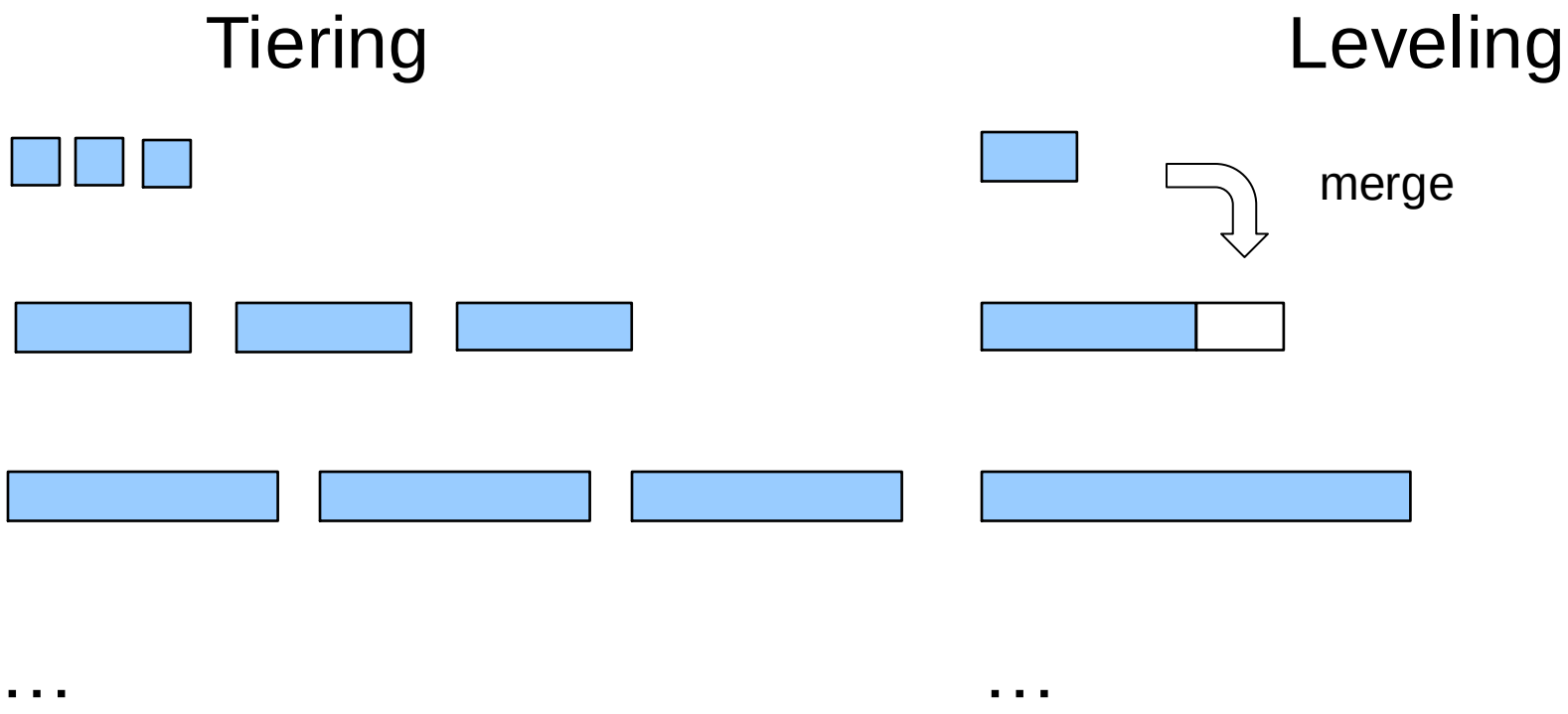


...

...

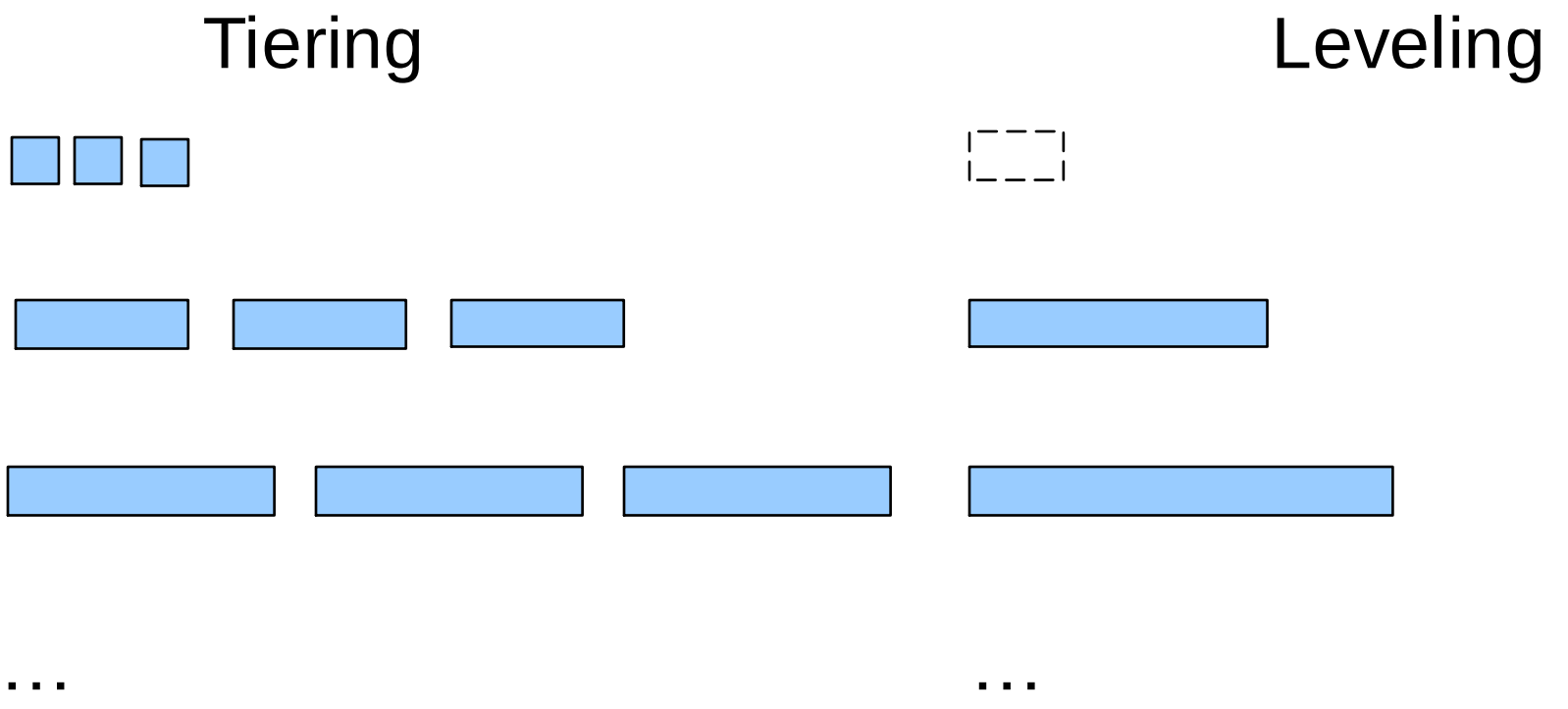
Size Ratio: T = 3

Merge Policies



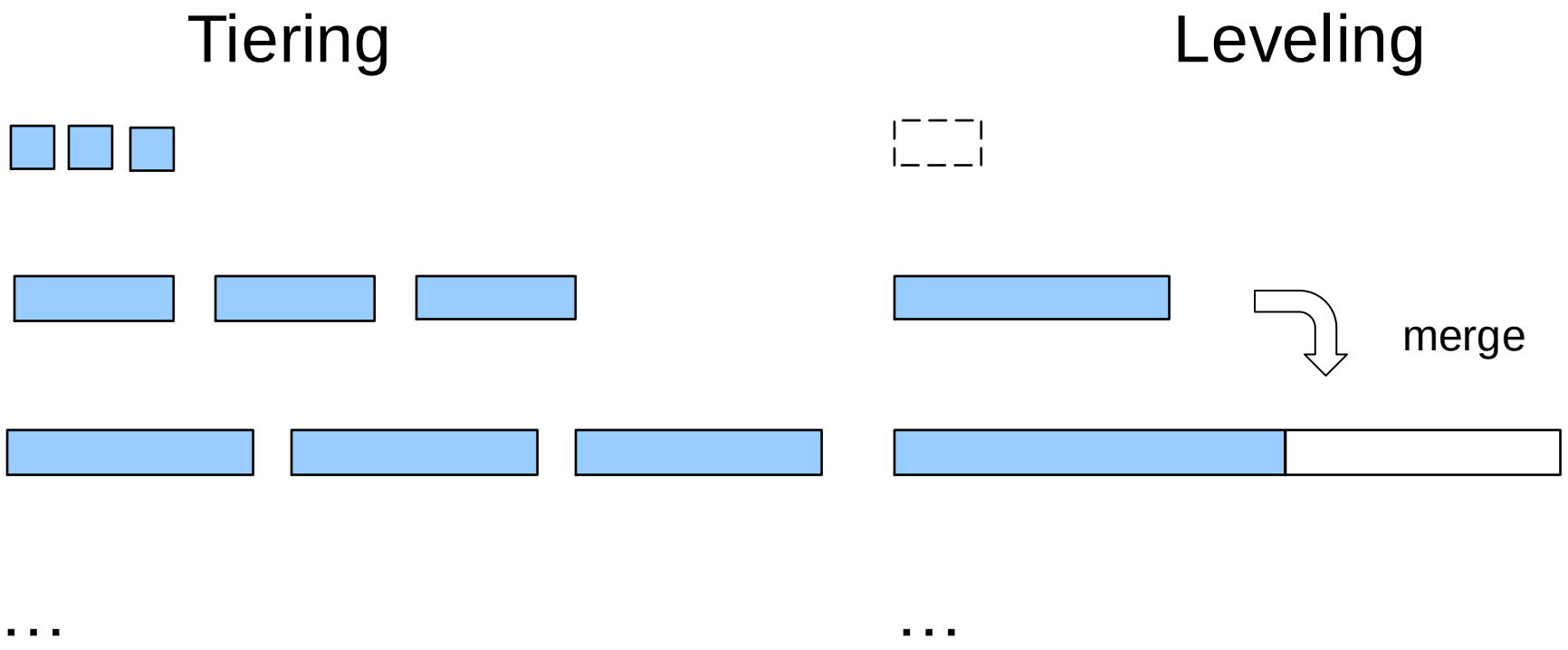
Size Ratio: T = 3

Merge Policies



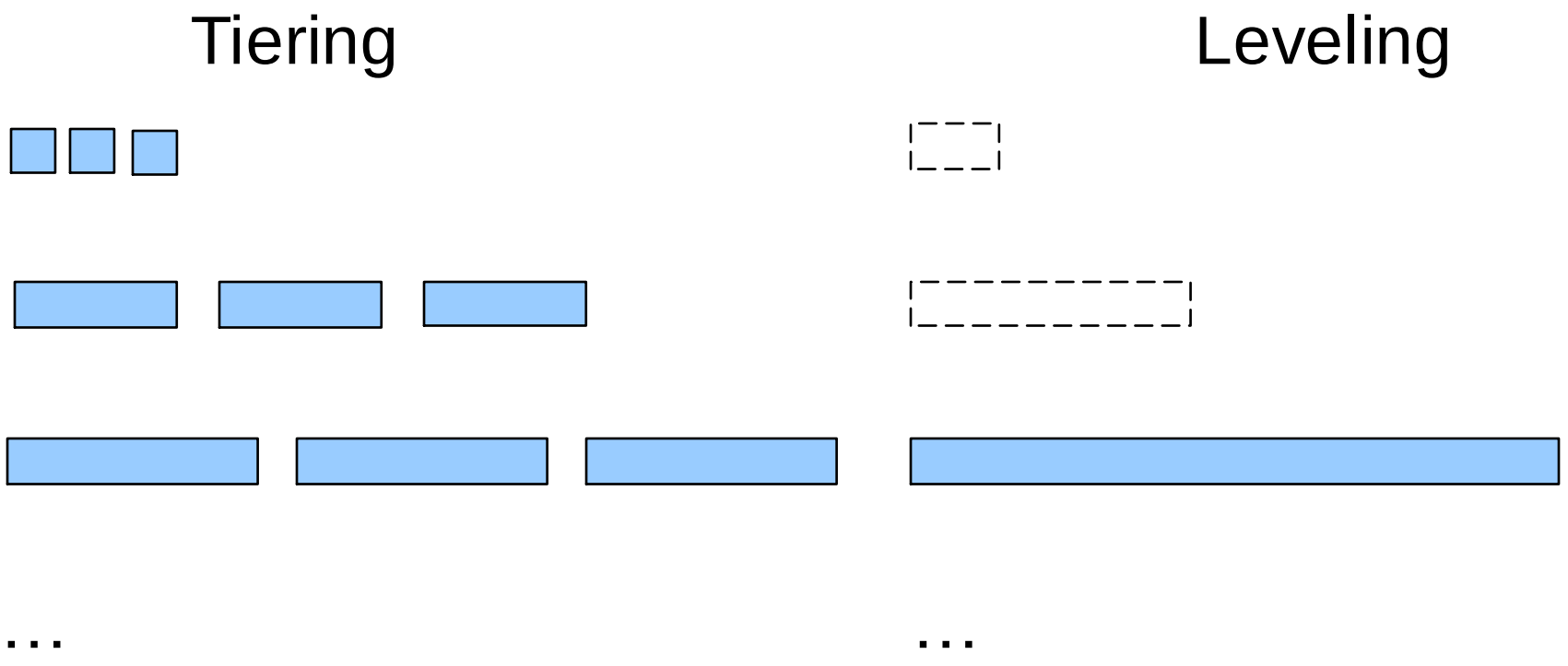
Size Ratio: T = 3

Merge Policies



Size Ratio: T = 3

Merge Policies

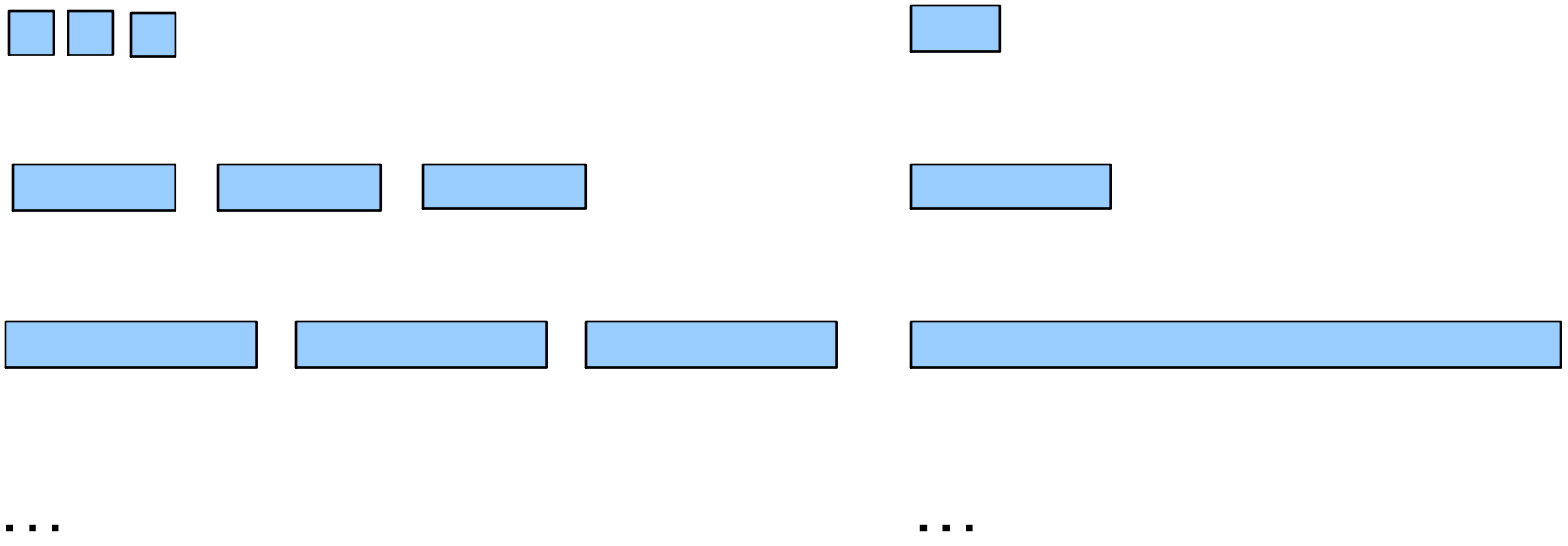


Size Ratio: T = 3

Merge Policies

Tiering

Leveling



…

…

What happens when $T \rightarrow \infty$?

Merge Policies

Tiering

Leveling

Then L = 1

What happens when $T \rightarrow \infty$?

Merge Policies

Tiering

Leveling

A log file!

Then L = 1

What happens when $T \rightarrow \infty$?

Merge Policies

Tiering

Leveling

A log file!

A sorted file!

Then L = 1

What happens when $T \rightarrow \infty$?

Outline

- Bloom Filters
- LSM Trees
- Cascades Extensible Optimizer
- Yannakakis' Algorithm

Cascades Optimizer

- Extends join ordering to full rewrite
- Supported by some of the most advanced DBMS today: SQL Server, Cocroach Lab; (not sure about DuckDB)
- Mostly “insider knowledge”

Cascades Optimizer

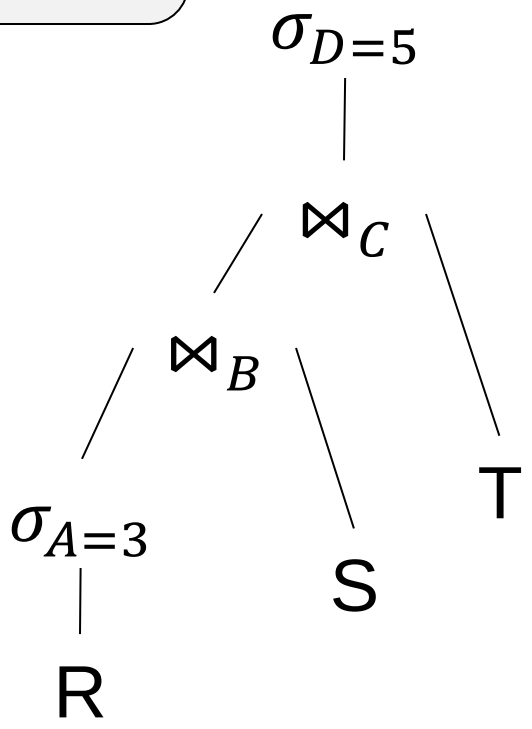
- Main idea: apply optimization rules:
  Q → Q'
- But keep both Q and Q'
- "Memo" data structure: reuses subplans

R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

Initialize Memo w/ one (naïve) plan



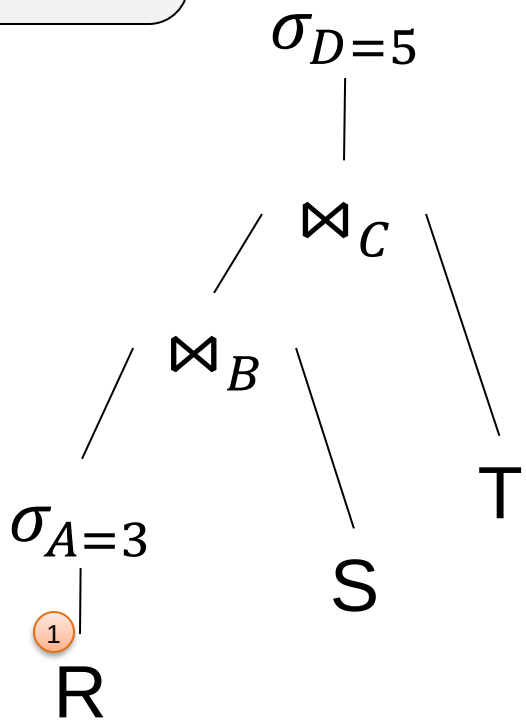
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1 Scan R

Initialize Memo w/ one (naïve) plan



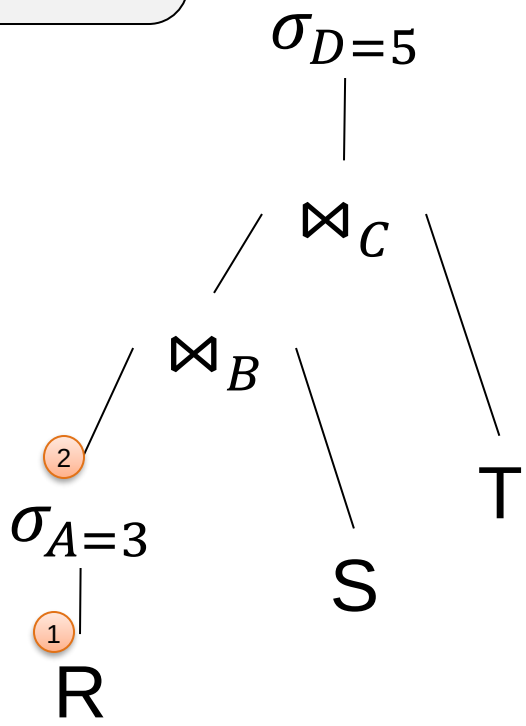
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1

Initialize Memo w/ one (naïve) plan



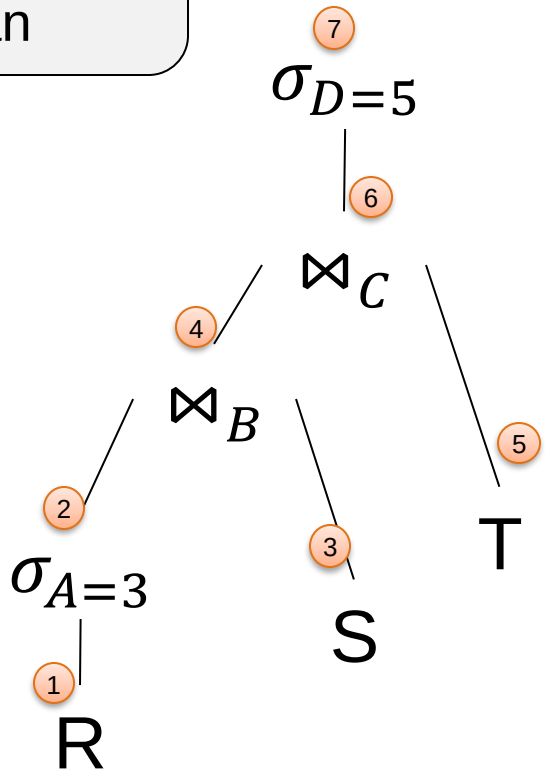
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6

Initialize Memo w/ one (naïve) plan

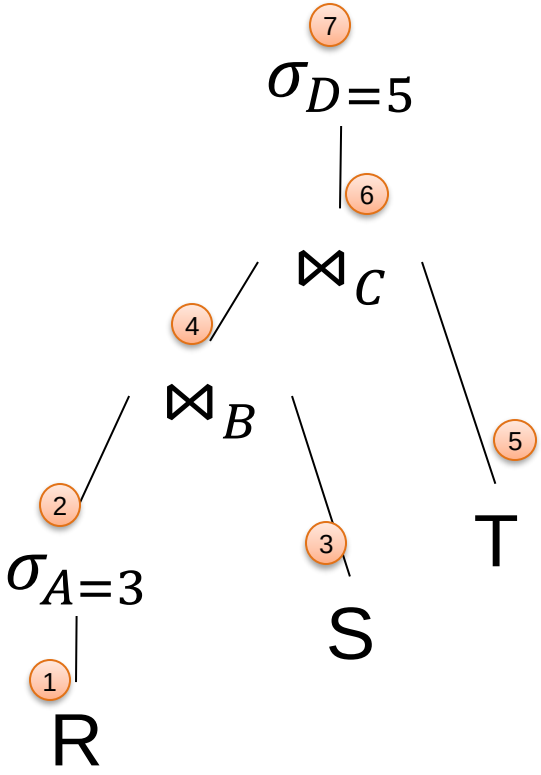


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6



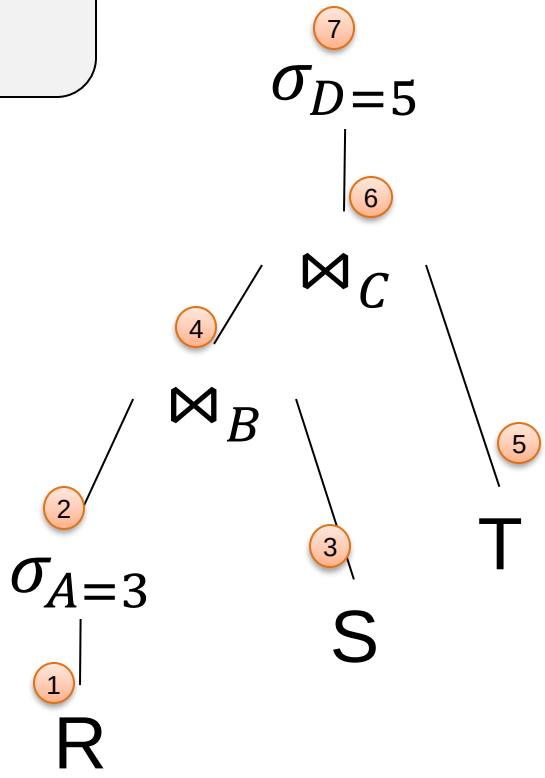
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6

Apply an optimization rule



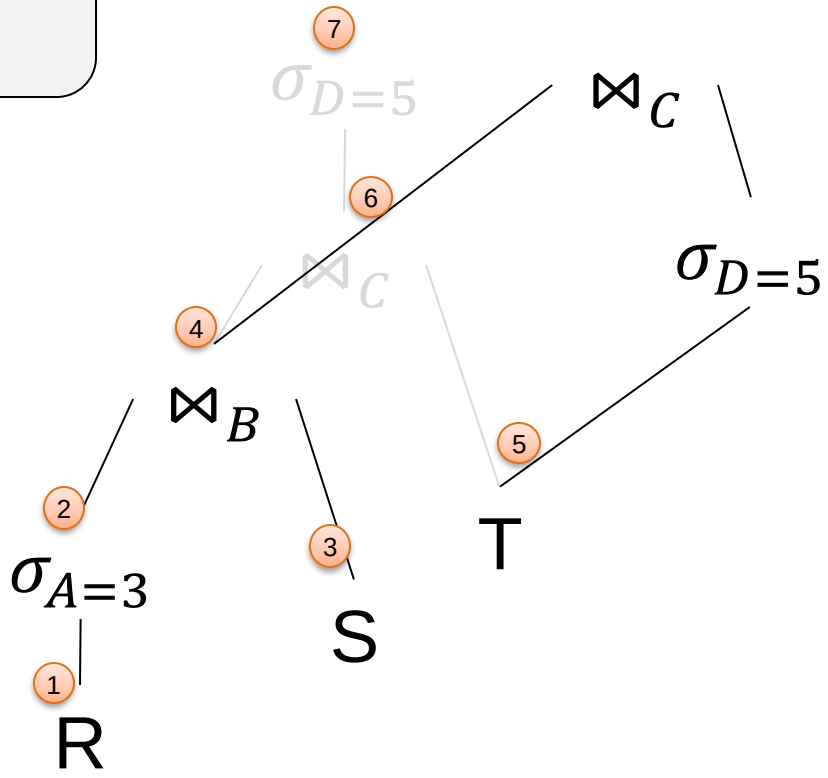
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6

Apply an optimization rule

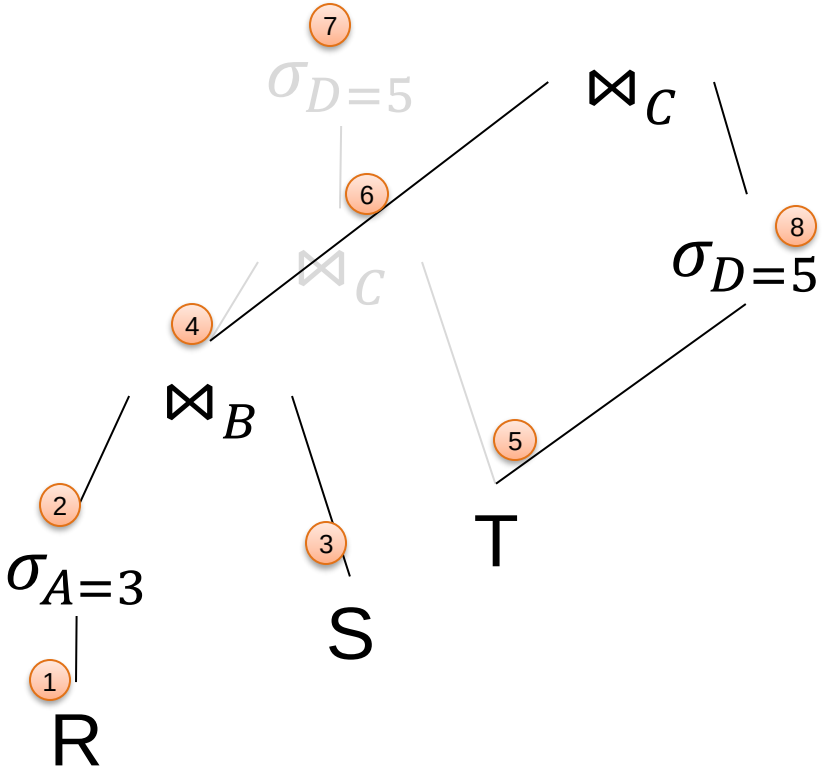


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6
8	Select[D=5] 5

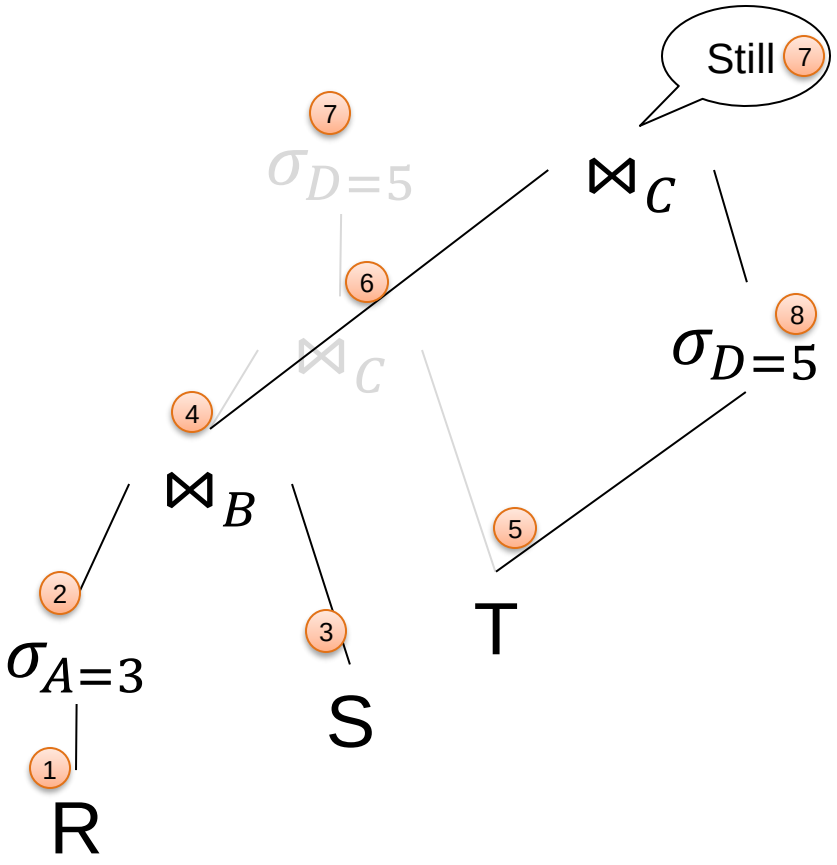


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R	
2	Select[A=3] 1	
3	Scan S	
4	Join[B=B] 2, 3	
5	Scan T	
6	Join[C=C] 4,5	
7	Select[D=5] 6	Join[C=C] 4,8
8	Select[D=5] 5	

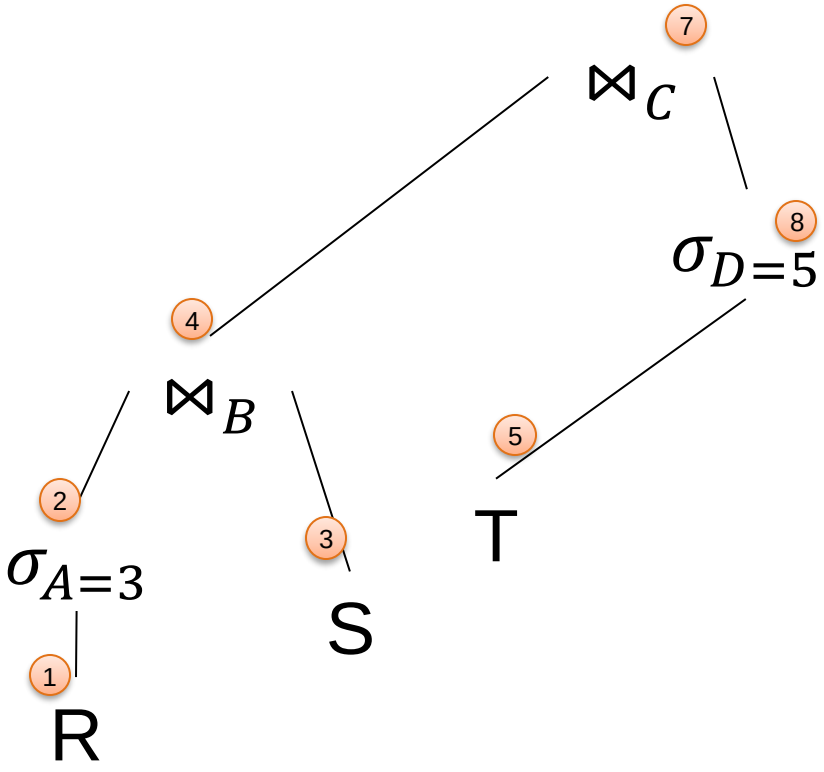


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R	
2	Select[A=3] 1	
3	Scan S	
4	Join[B=B] 2, 3	
5	Scan T	
6	Join[C=C] 4,5	
7	Select[D=5] 6	Join[C=C] 4,8
8	Select[D=5] 5	

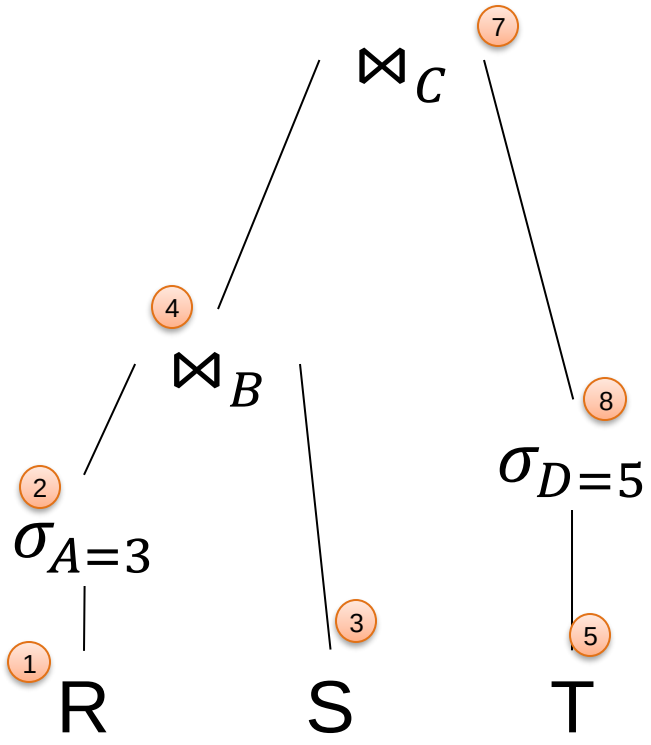


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6 Join[C=C] 4,8
8	Select[D=5] 5



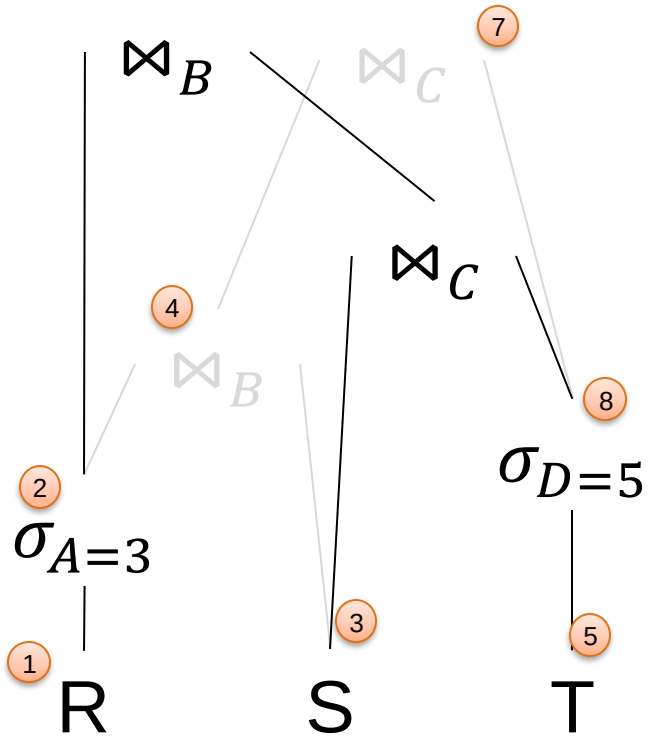
R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R	
2	Select[A=3] 1	
3	Scan S	
4	Join[B=B] 2, 3	
5	Scan T	
6	Join[C=C] 4,5	
7	Select[D=5] 6	Join[C=C] 4,8
8	Select[D=5] 5	

Apply another rule

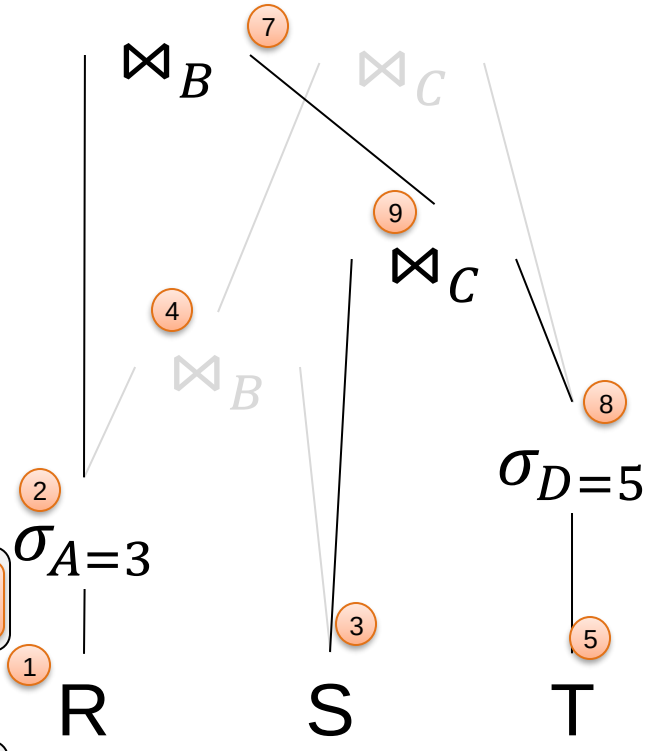


R(A,B), S(B,C), T(C,D)

```
select *
from R, S, T
where R.B=S.B
    and S.C=T.C
    and R.A = 3
    and T.D = 5
```

The Memo

1	Scan R
2	Select[A=3] 1
3	Scan S
4	Join[B=B] 2, 3
5	Scan T
6	Join[C=C] 4,5
7	Select[D=5] 6 Join[C=C] 4,8 Join[B=B] 2,9
8	Select[D=5] 5
9	Join[C=C] 3, 8



Outline

- Bloom Filters
- LSM Trees
- Cascades Extensible Optimizer
- Yannakakis' Algorithm

Motivation

- What is the worst time complexity to compute these queries?
  - Q(X,Y,Z) = R(X,Y), S(Y,Z)
  - Q(X,Y,Z,U,V)=R(X,Y),S(Y,Z),T(Z,U),K(U,V)

- If |R| = |S| = … = N, then the time for the queries above is $O(N^2)$, and $O(N^4)$

Motivation

- Yannakakis' algorithm computes any acyclic query in time O(N+OUT), where OUT is the size of the output

- There are two key ideas:
  - It works only for acyclic queries
  - It performs a semi-join reduction before computing the joins

Semi-Join Reduction

Semi-join definition:

$$R \ltimes S = \Pi_{attr(R)}(R \bowtie S)$$

Semi-Join Reduction

Semi-join definition:

$$R \ltimes S = \Pi_{attr(R)}(R \bowtie S)$$

Basic law:

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

Example 1

- Example:

$$Q(A,B,C) = R(A,B) \bowtie S(B,C)$$

Example 1

- Example:

$$Q(A,B,C) = R(A,B) \bowtie S(B,C)$$

- A semijoin reducer is:

$$R_1(A,B) = R(A,B) \ltimes S(B,C)$$

Example 1

- Example:

$$Q(A,B,C) = R(A,B) \bowtie S(B,C)$$

- A semijoin reducer is:

$$R_1(A,B) = R(A,B) \ltimes S(B,C)$$

- The rewritten query is:

$$Q(A,B,C) = R_1(A,B) \bowtie S(B,C)$$

Example 2

$$Q(y,z,u) = R(\text{‘a’}, y), S(y,z), T(z,u), K(u,\text{’b’})$$

Semi-join reducer:

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)

Example 2

$Q(y,z,u) = R(\text{‘a’}, y), S(y,z), T(z,u), K(u,\text{’b’})$

Semi-join reducer:

$S'(y,z) :- S(y,z) \ltimes R(\text{‘a’}, y)$
$T'(z,u) :- T(z,u) \ltimes S'(y,z)$
$K'(u) :- K(u,\text{’b’}) \ltimes T'(z,u)$

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)
K’(u) :- K(u,’b’) ⋉ T’(z,u)
T’’(z,u) :- T’(z,u) ⋉ K’(u)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)
K’(u) :- K(u,’b’) ⋉ T’(z,u)
T’’(z,u) :- T’(z,u) ⋉ K’(u)
S’’(y,z) :- S’(y,z) ⋉ T’’(z,u)
R’’(y) :- R(‘a’,y) ⋉ S’’(y,z)

Example 2

$Q(y,z,u) = R(\text{‘a’}, y), S(y,z), T(z,u), K(u,\text{’b’})$

Semi-join reducer:

$S'(y,z) :- S(y,z) \ltimes R(\text{‘a’}, y)$
$T'(z,u) :- T(z,u) \ltimes S'(y,z)$
$K'(u) :- K(u,\text{’b’}) \ltimes T'(z,u)$
$T''(z,u) :- T'(z,u) \ltimes K'(u)$
$S''(y,z) :- S'(y,z) \ltimes T''(z,u)$
$R''(y) :- R(\text{‘a’},y) \ltimes S''(y,z)$

Reduced query:

$Q(y,z,u) = R''(y), S''(y,z), T''(z,u), K''(u)$

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

T is called *join tree*

Acyclic Queries

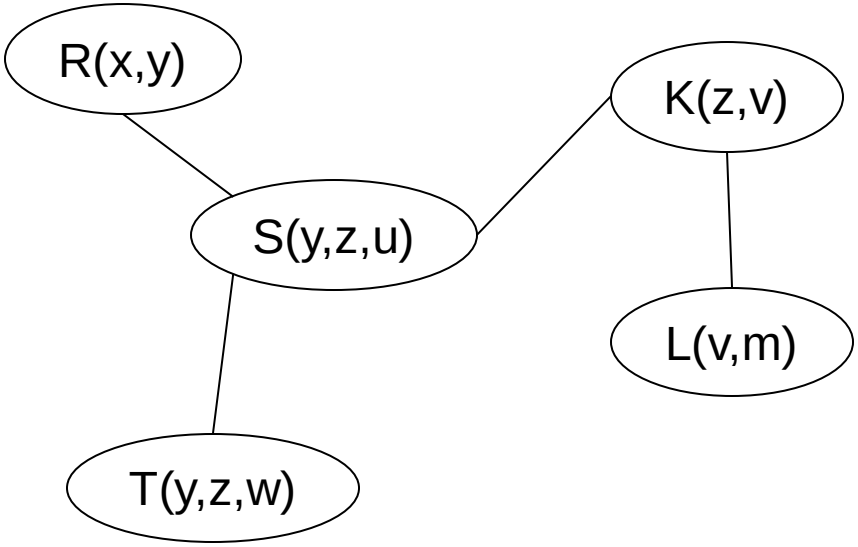
Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

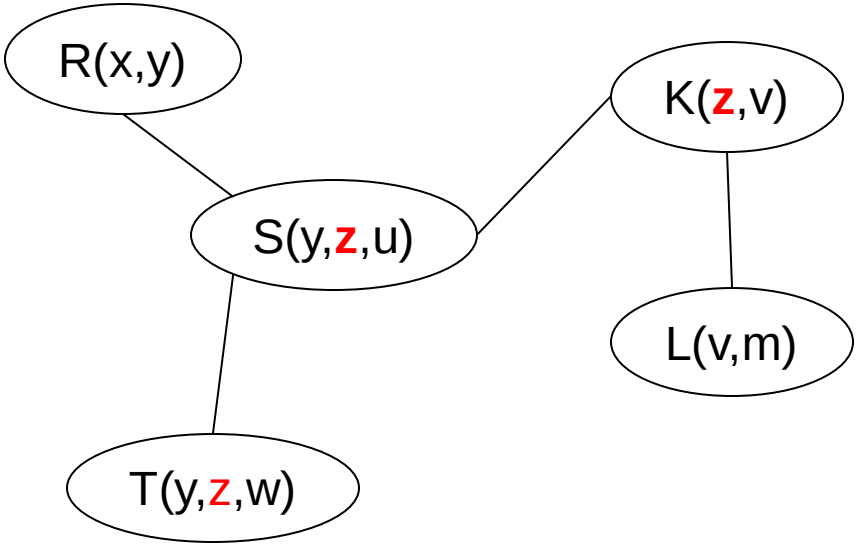


R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



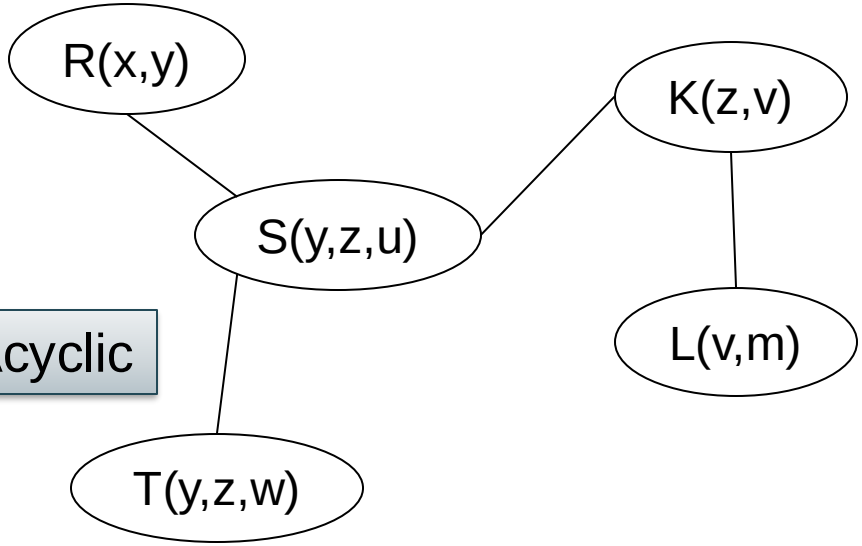
E.g. **z** forms a connected component

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



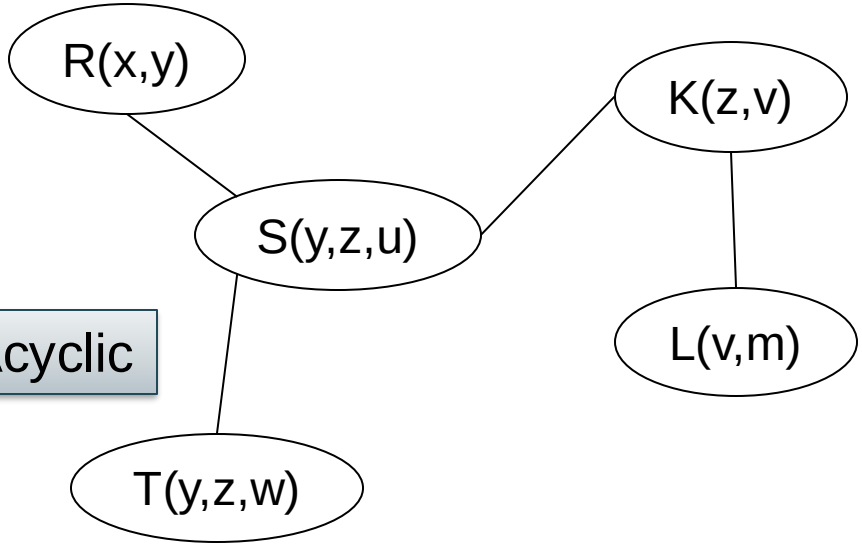
Acyclic

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



Acyclic

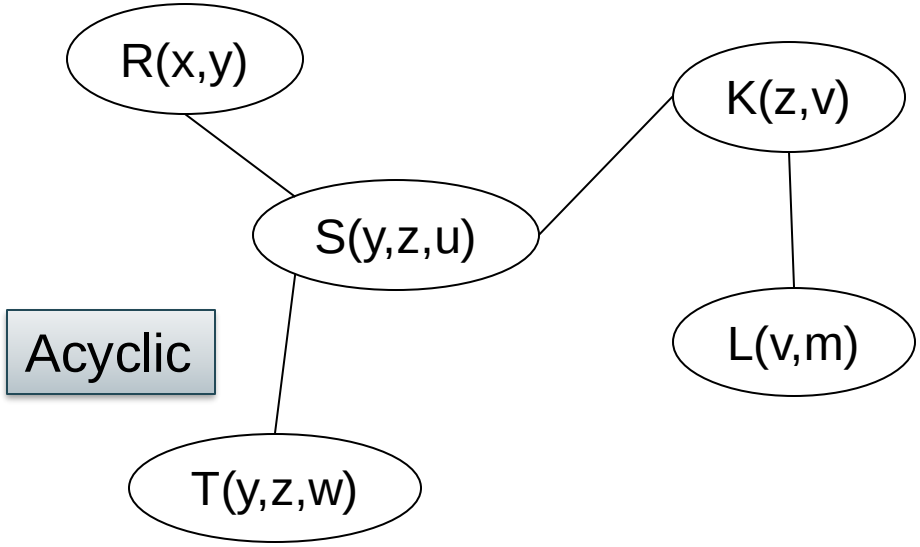
R(x,y),S(y,z),T(z,x)

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

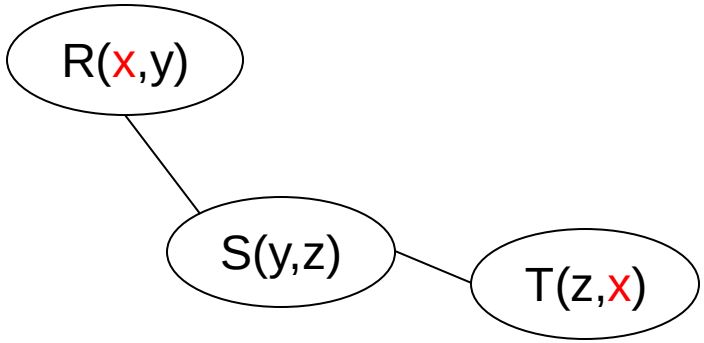
Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



Acyclic

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

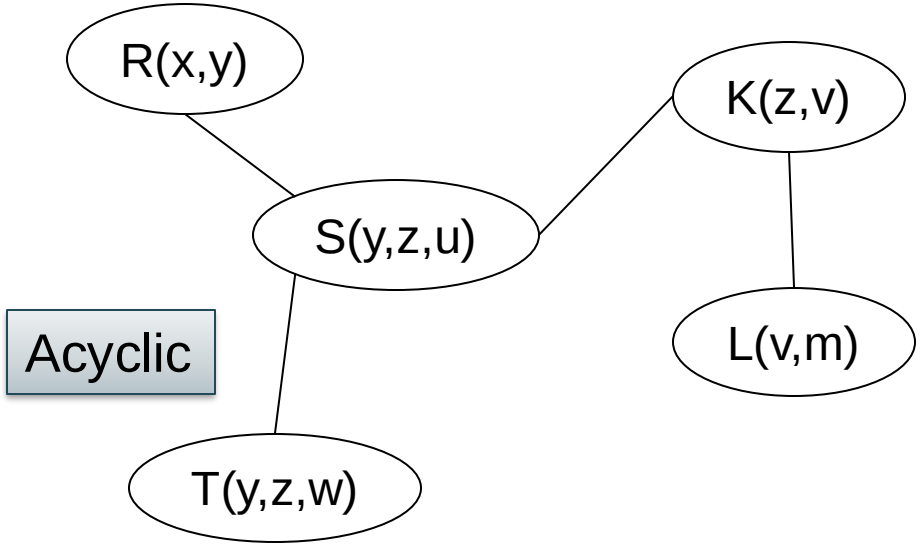


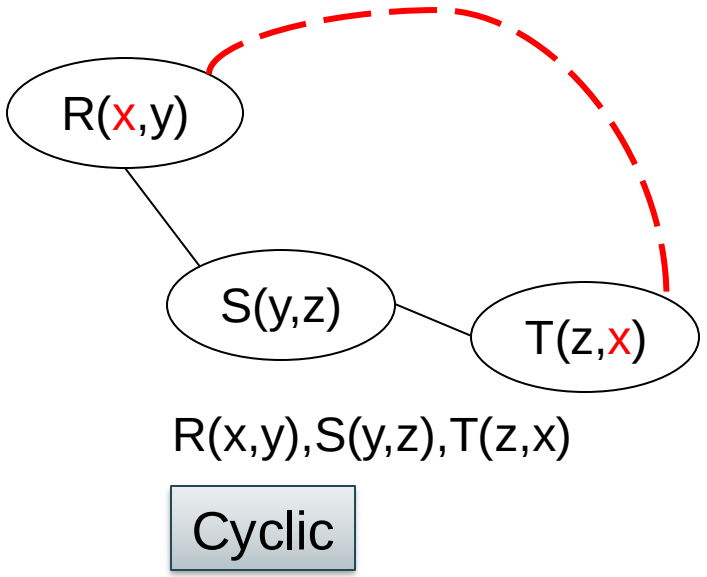
R(x,y),S(y,z),T(z,x)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component





A Theorem

Q = an acyclic query Q that is:

- Boolean, or
- Full, or
- Aggregate with $\leq$1 group-by variable

Theorem Q can be computed in time*:

$$\tilde{O}(|Input| + |Output|)$$

* $\tilde{O}$ means *plus a logarithmic factor (for sorting)*

Yannakakis Algorithm

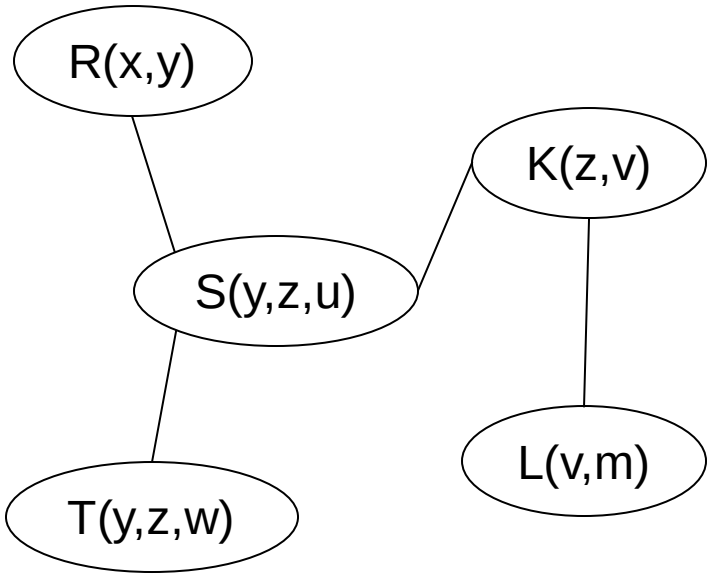
- Step 1: semi-join reduction
  - Pick any root node in the join tree of Q
  - Semi-join reduction from leaves to root
  - Semi-join reduction from root to leaves
- Step 2:
  - Compute the joins bottom up,
  - Push group-by down

Example: Full CQ

$$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$$

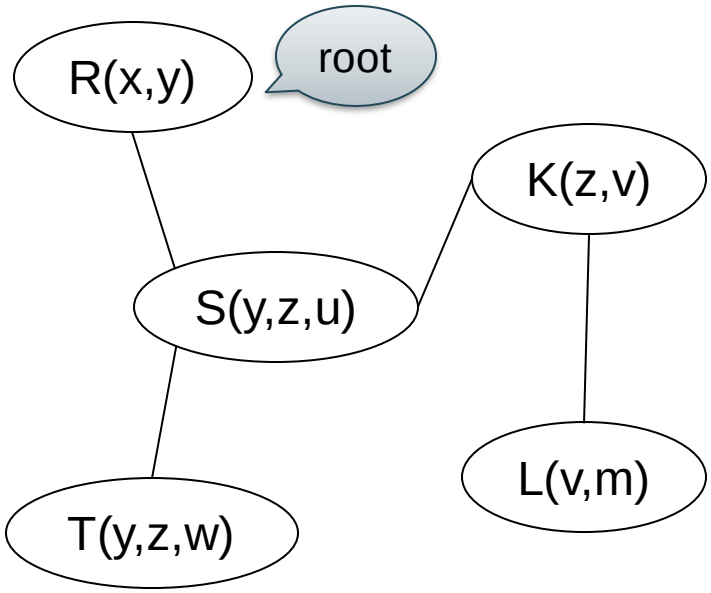
Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



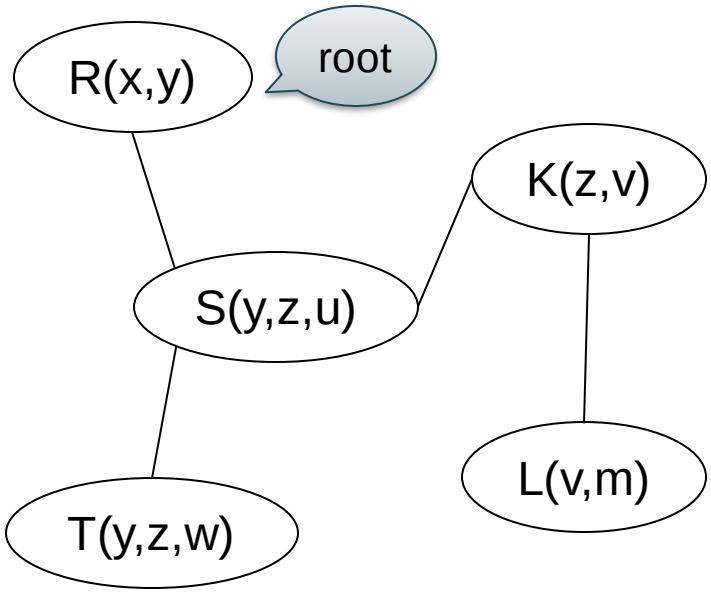
Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

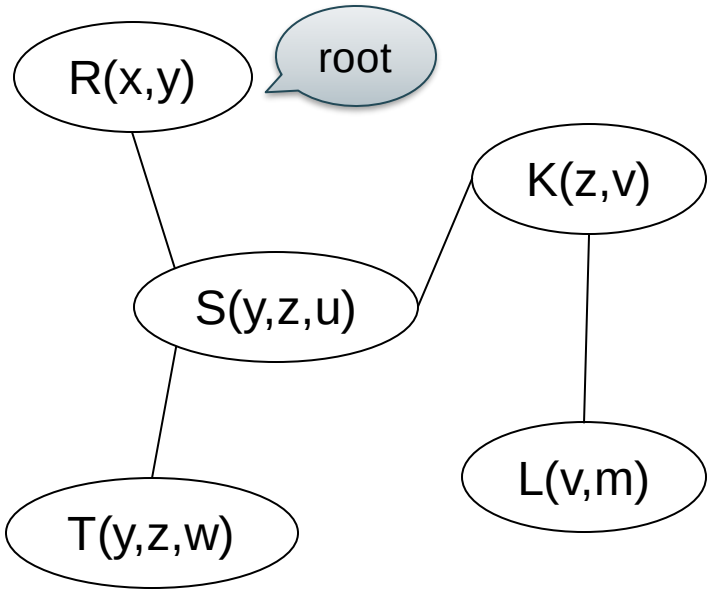


-- Leaves to root:

$K :- K \ltimes L$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



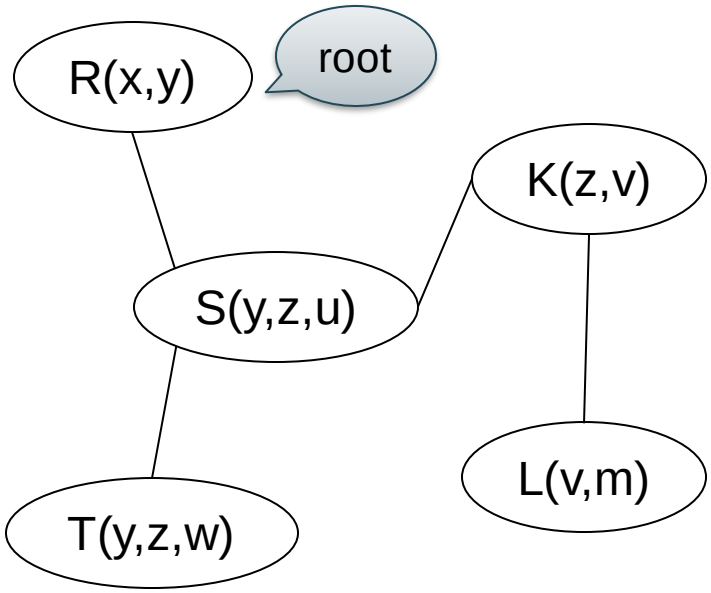
-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

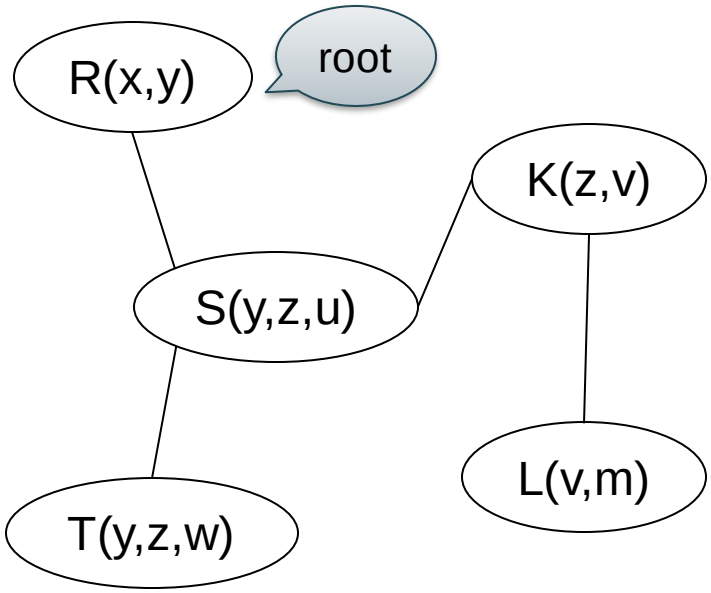
$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

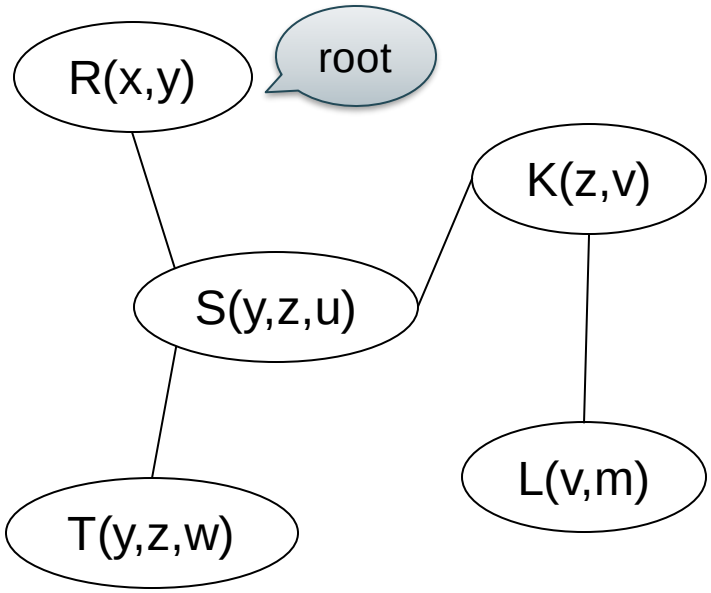
$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

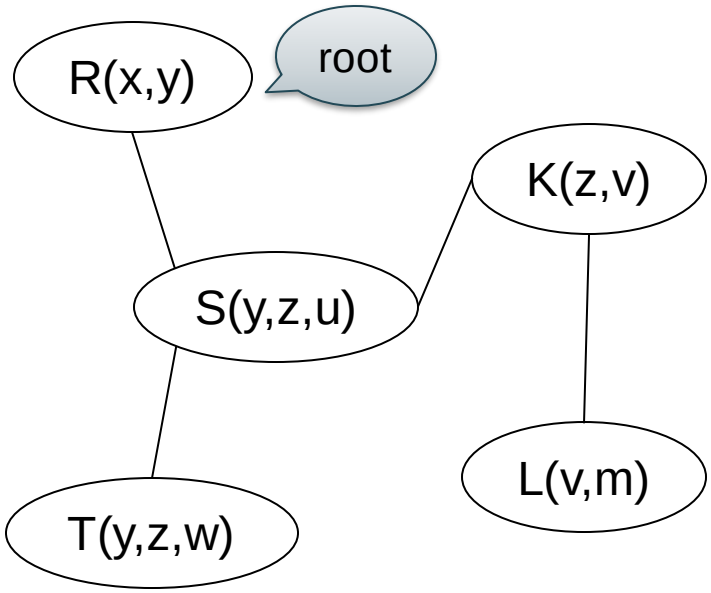
$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

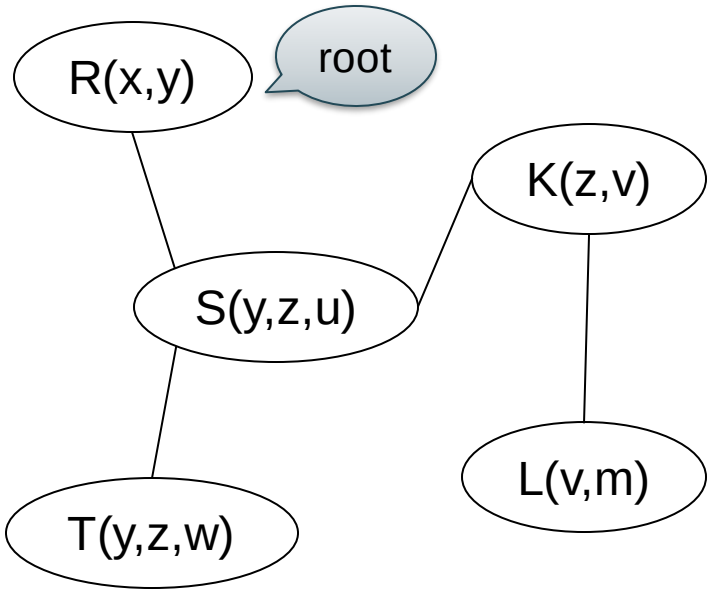
$R :- R \ltimes S$

-- Root to leaves:

$S :- S \ltimes R$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

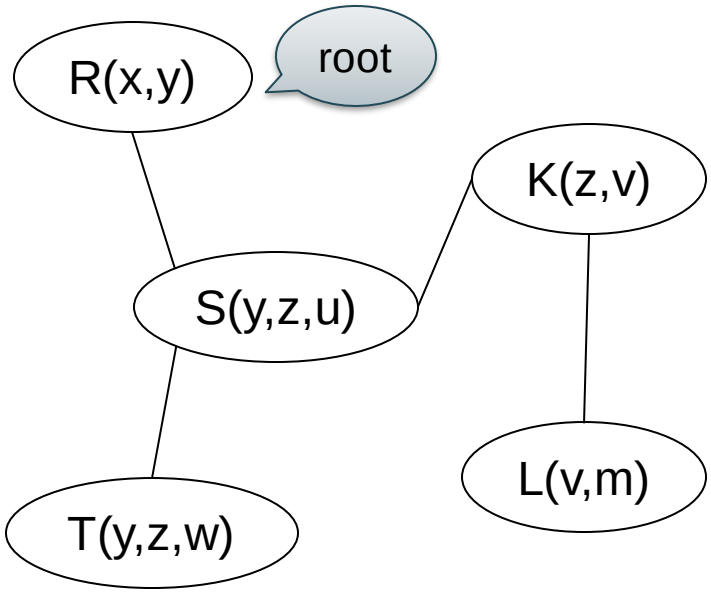
-- Root to leaves:

$S :- S \ltimes R$

$T :- T \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

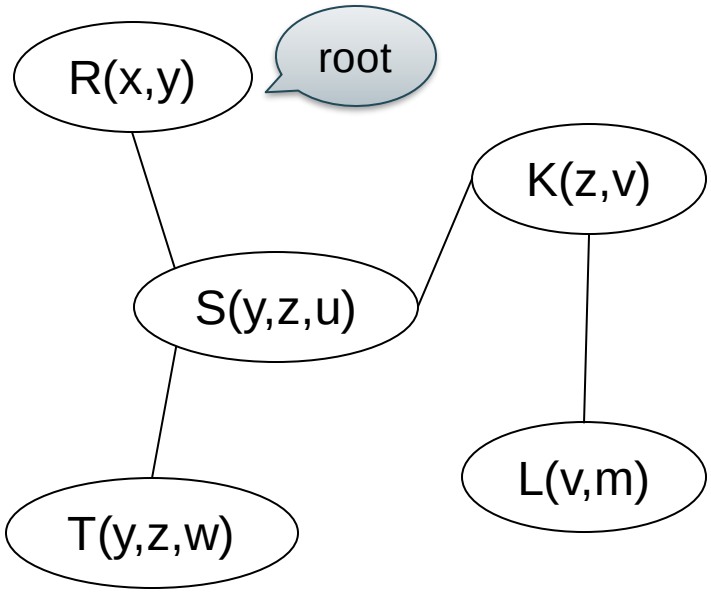
$S :- S \ltimes R$

$T :- T \ltimes S$

$K :- K \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

$S :- S \ltimes R$

$T :- T \ltimes S$

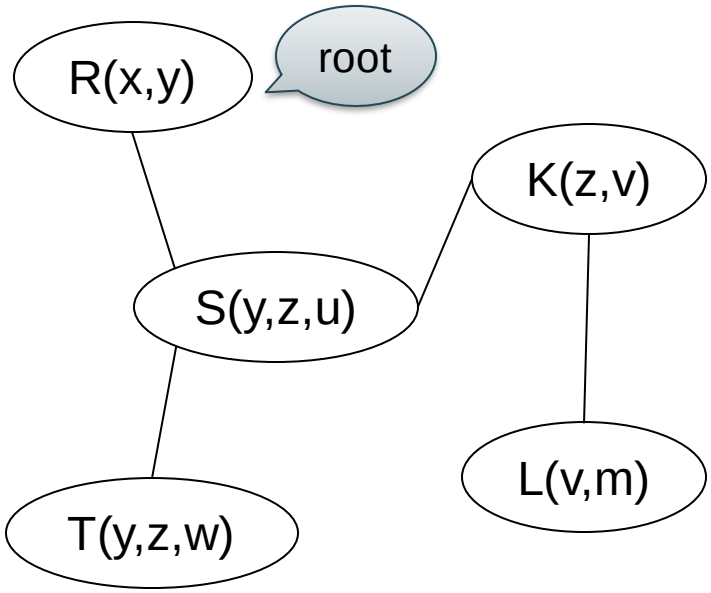
$K :- K \ltimes S$

$L :- L \ltimes K$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

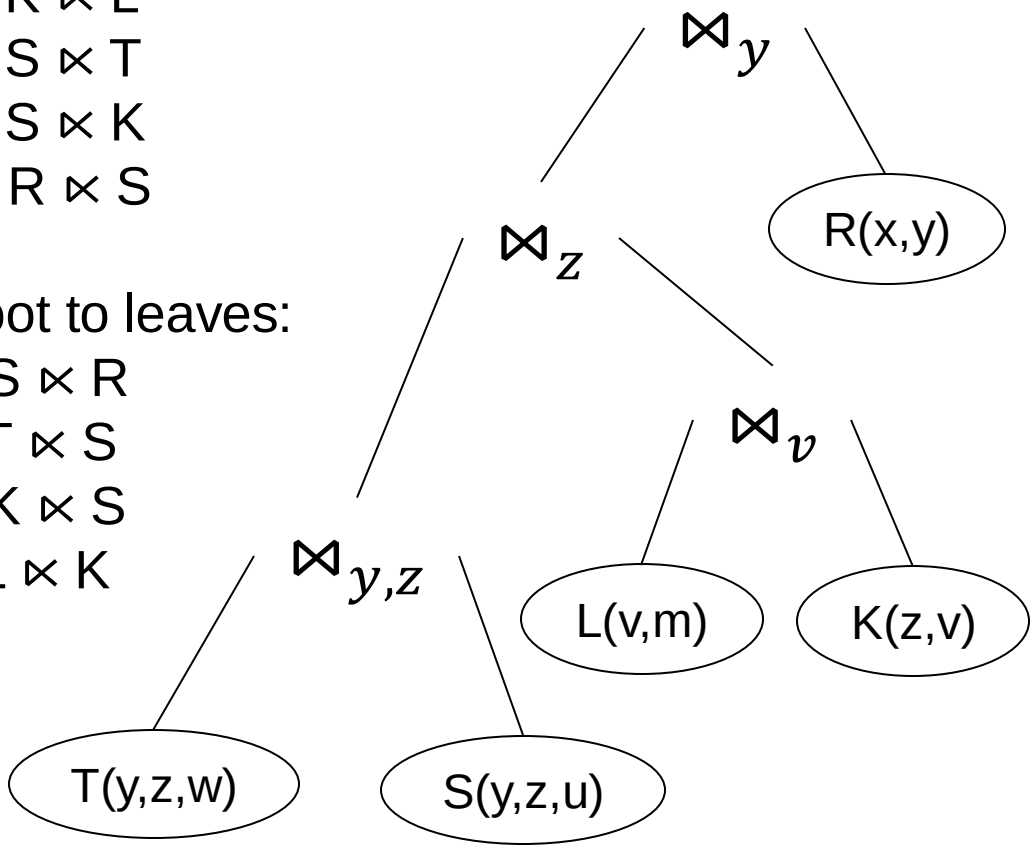
-- Root to leaves:

S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

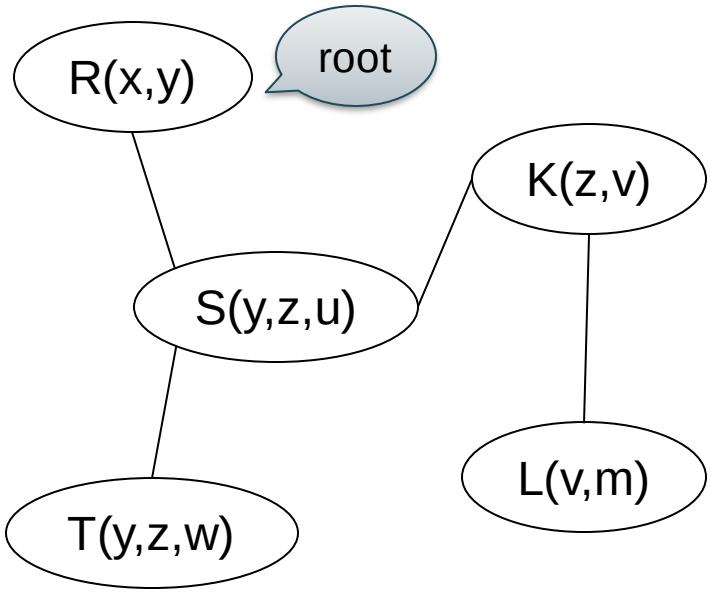
L :- L ⋉ K



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

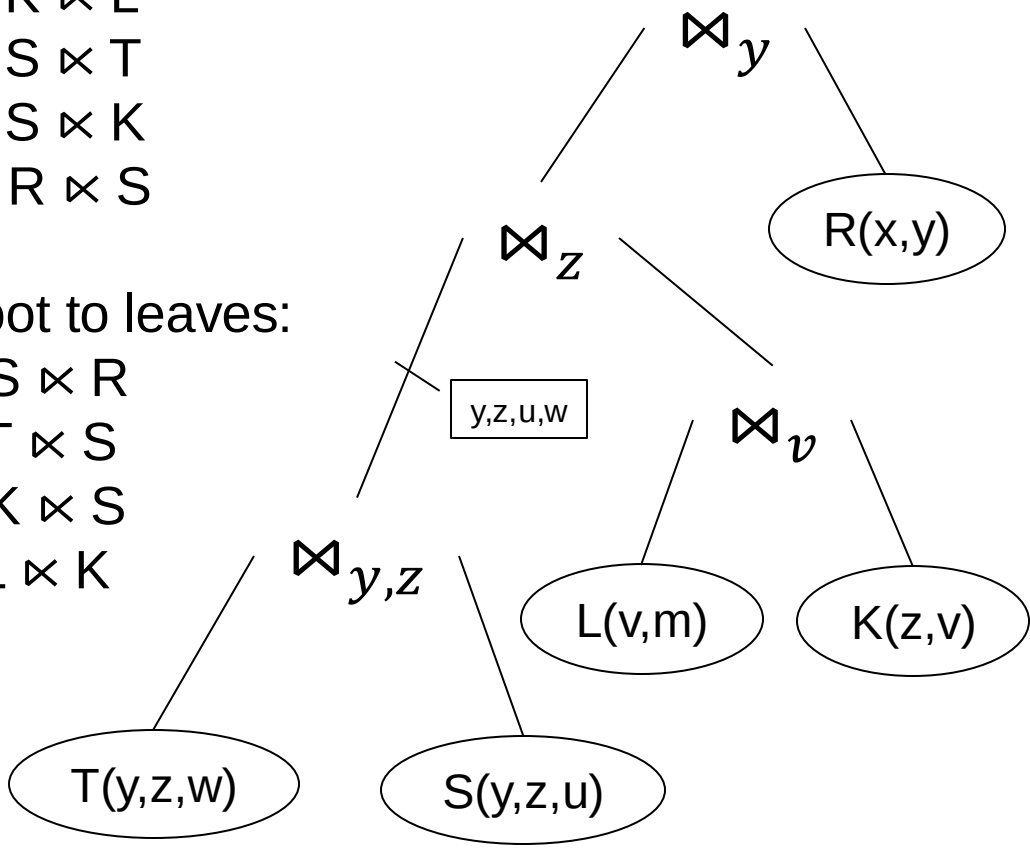
-- Root to leaves:

S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

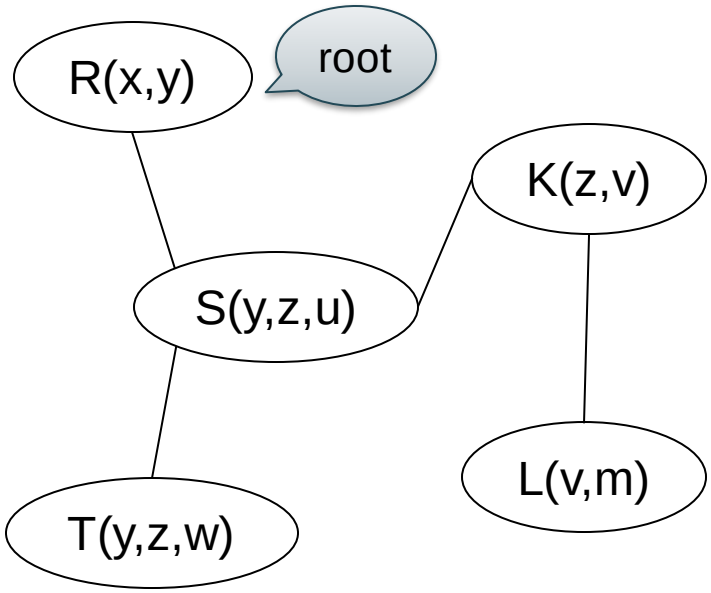
L :- L ⋉ K



Example: Full CQ

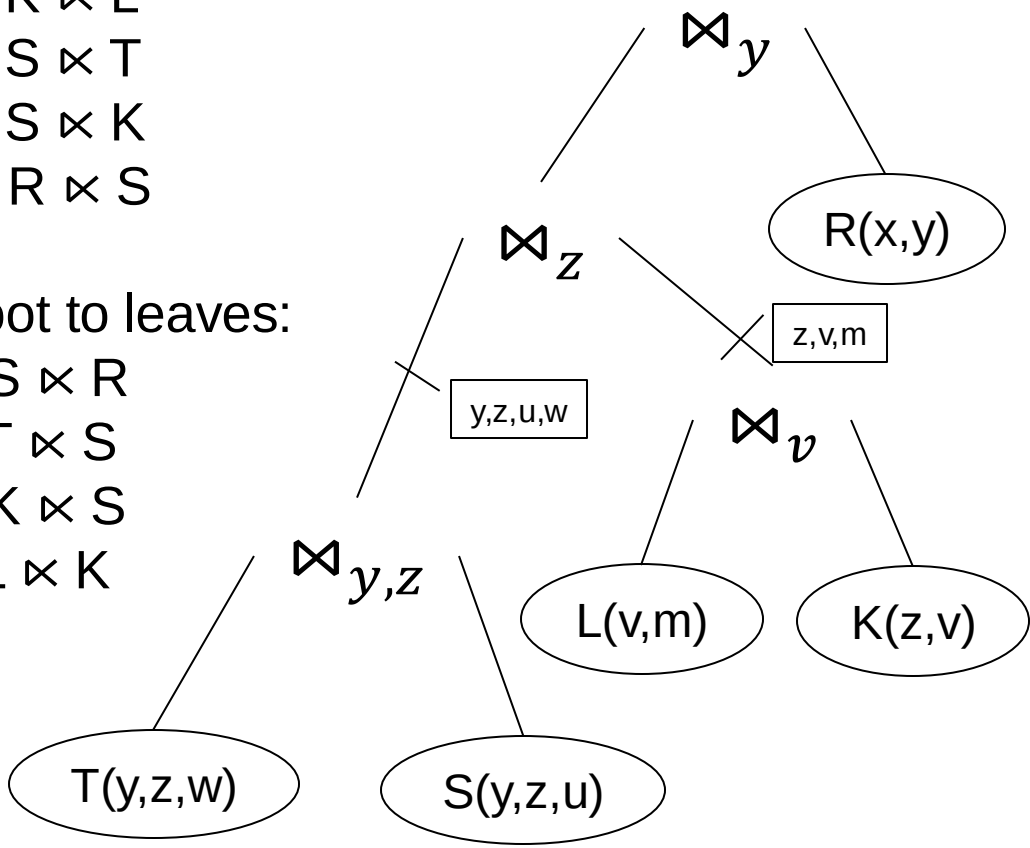
Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

Join (any order in the tree)



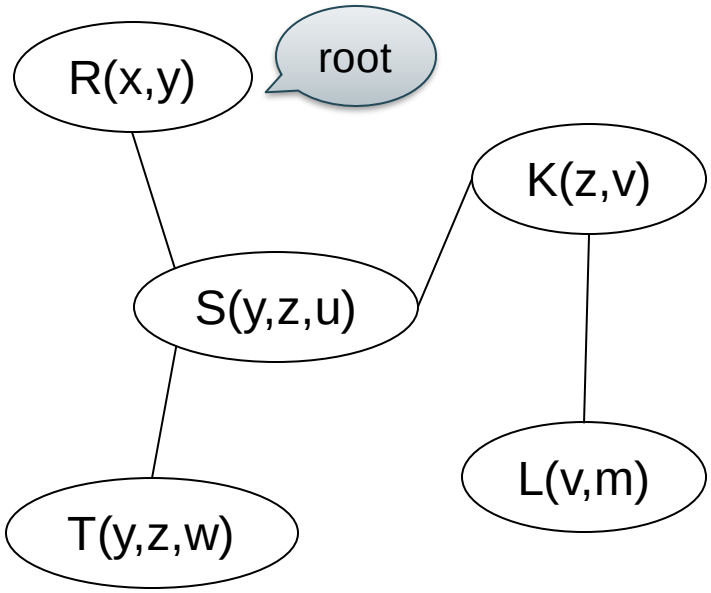
-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K



Example: Full CQ

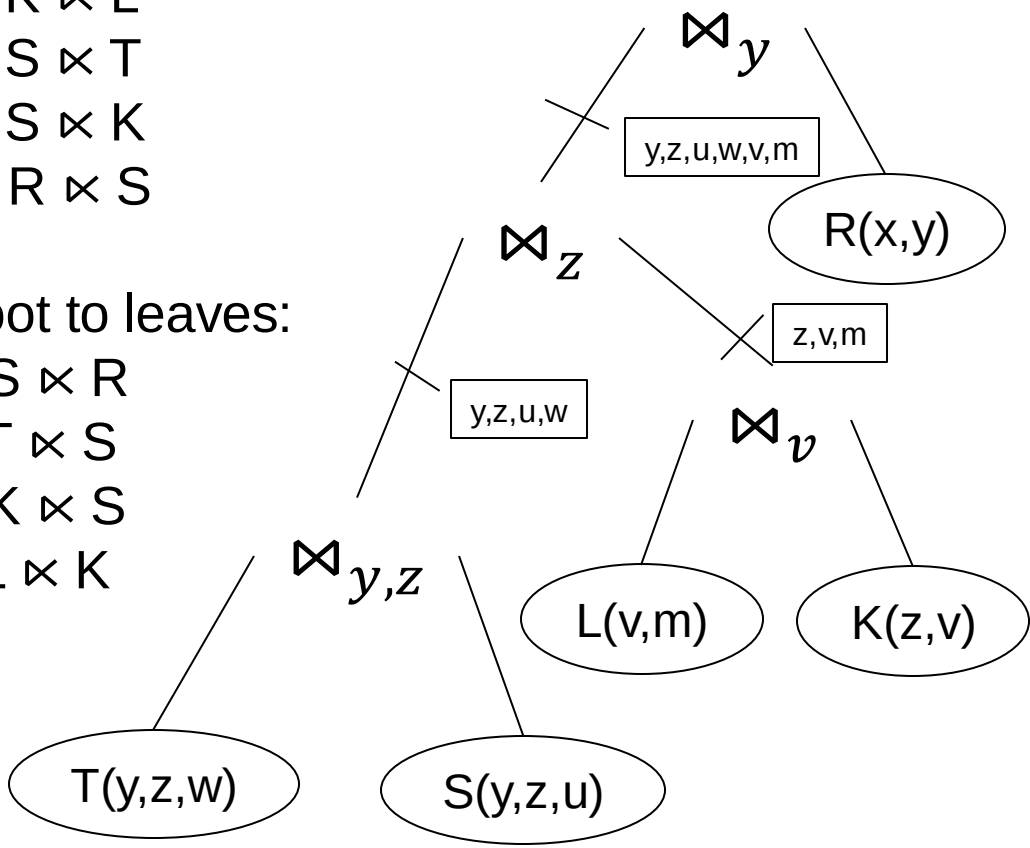
Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

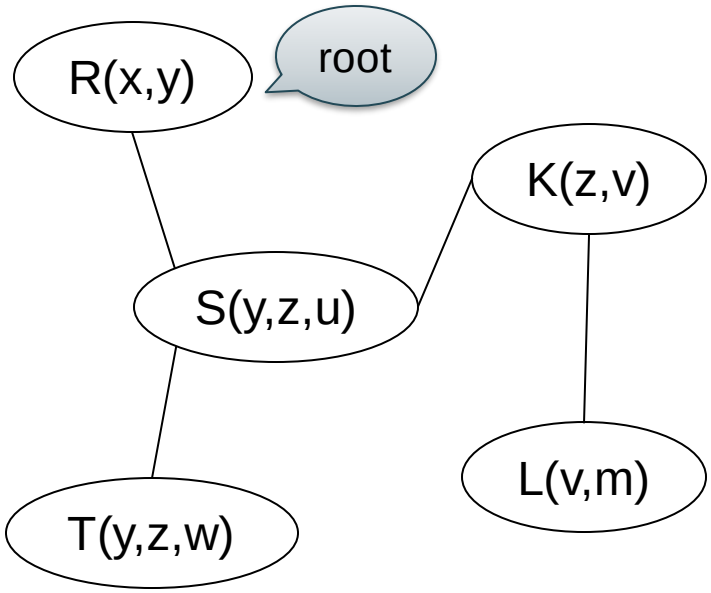
-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K

Join (any order in the tree)



Example: Full CQ

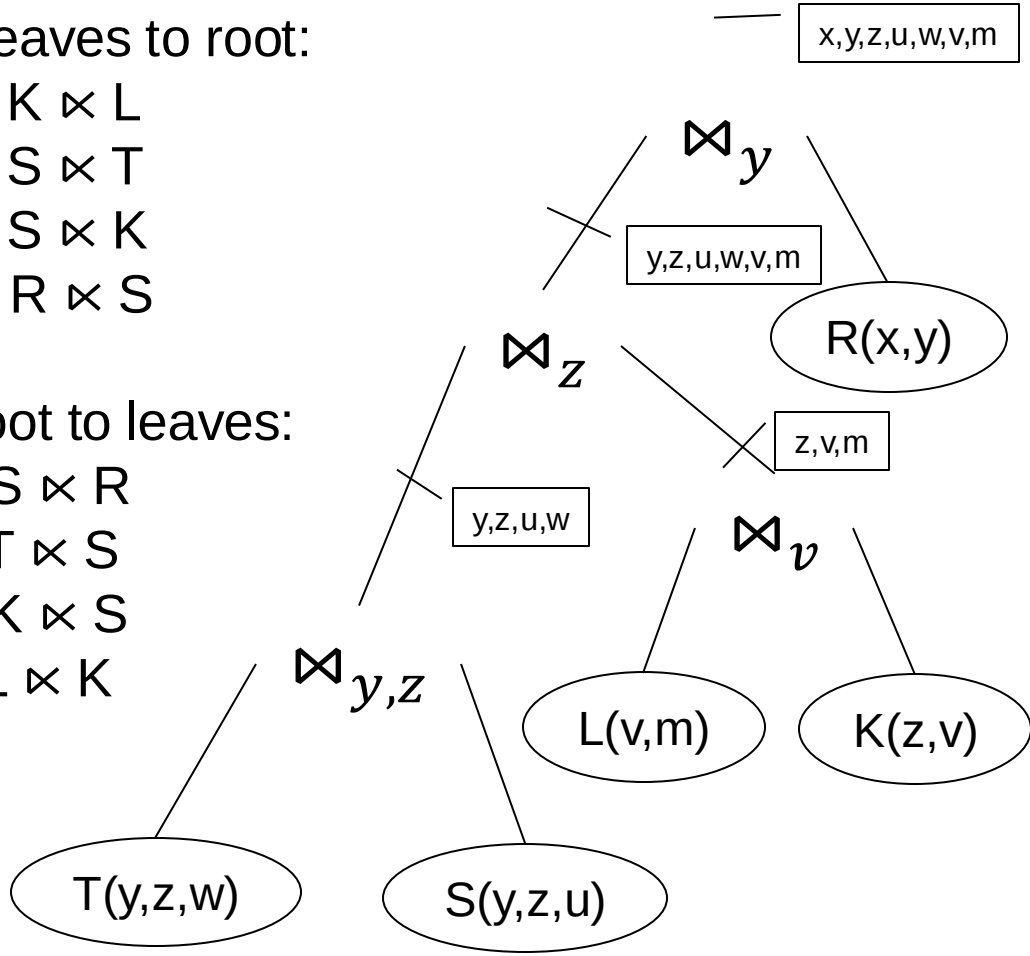
Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

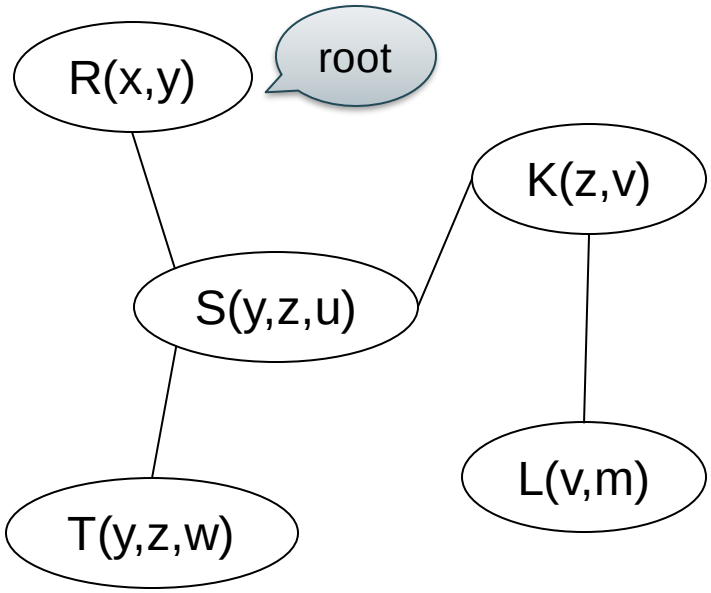
-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K

Join (any order in the tree)



Example: Full CQ

Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

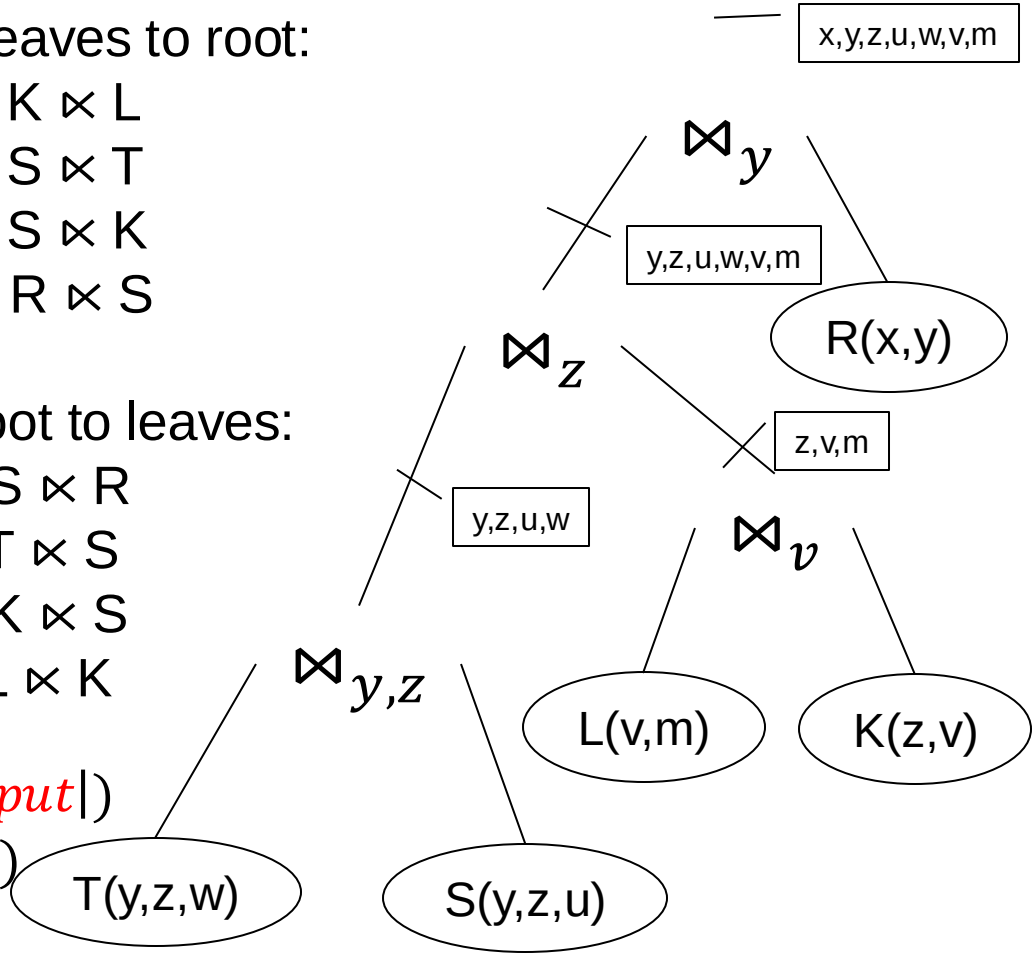


-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K

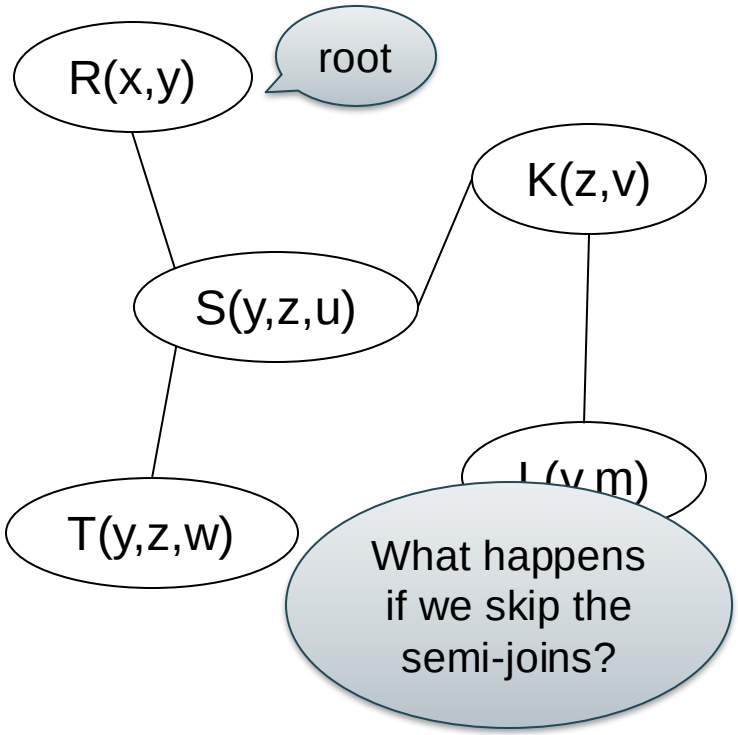
Runtime:
- Every semi-join takes time $\tilde{O}(|Input|)$
- Every join takes time $\tilde{O}(|Output|)$

Join (any order in the tree)



Example: Full CQ

Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

-- Root to leaves:

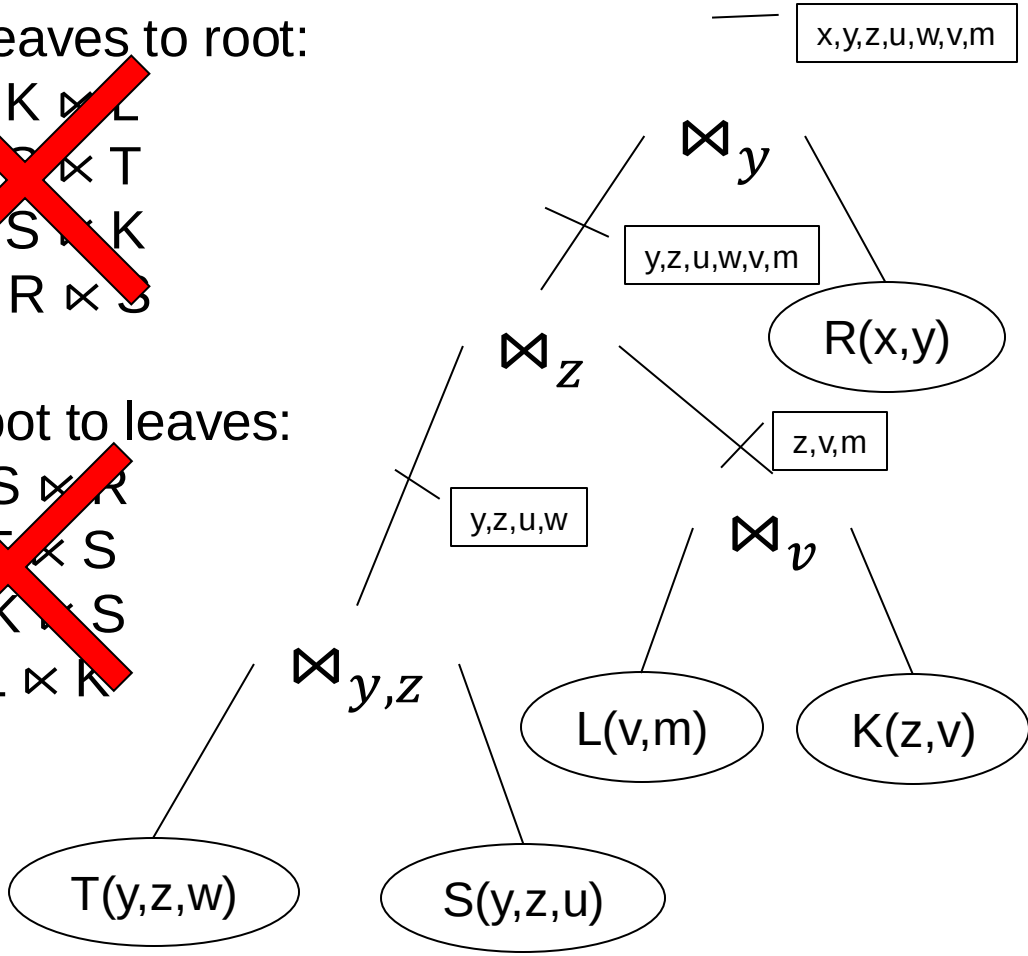
S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

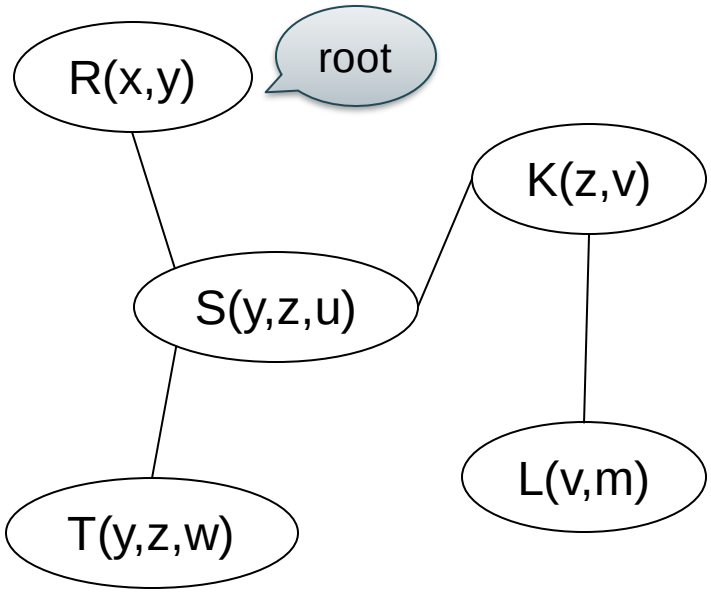
L :- L ⋉ K

Join (any order in the tree)

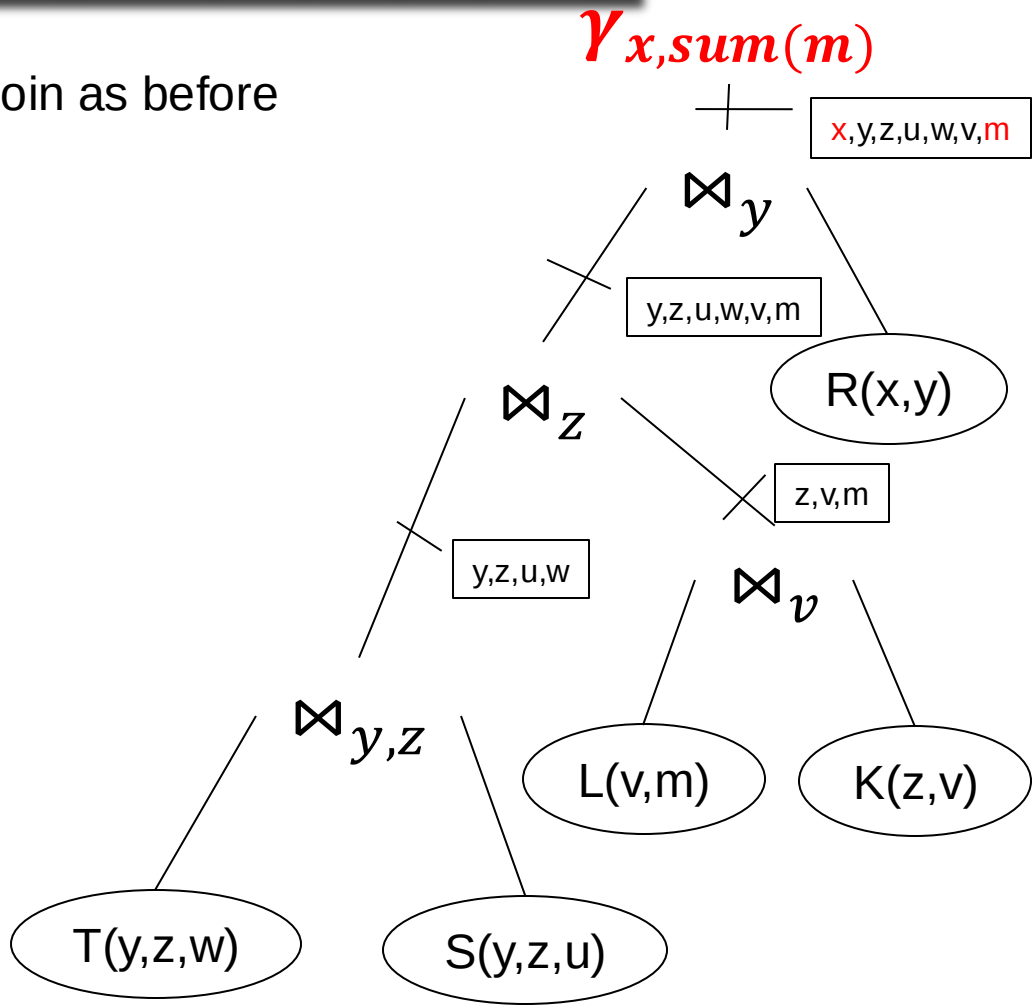


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

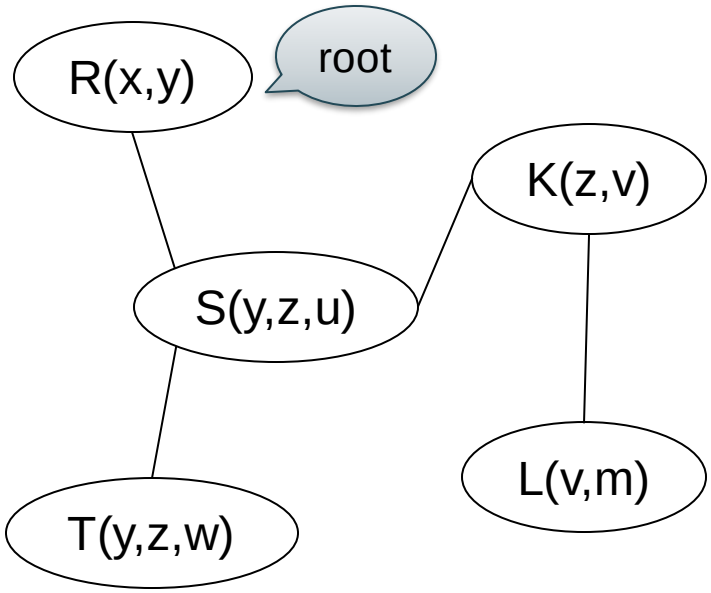


Semi-join as before

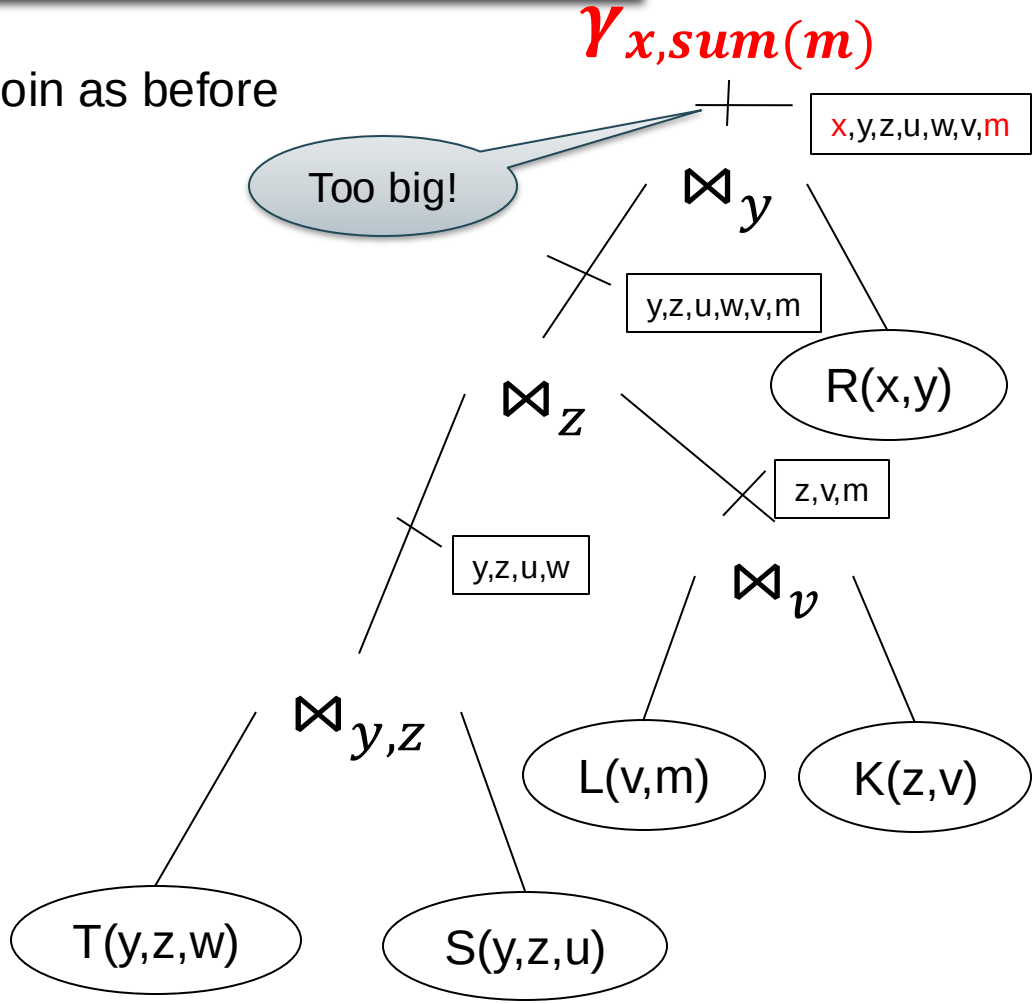


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

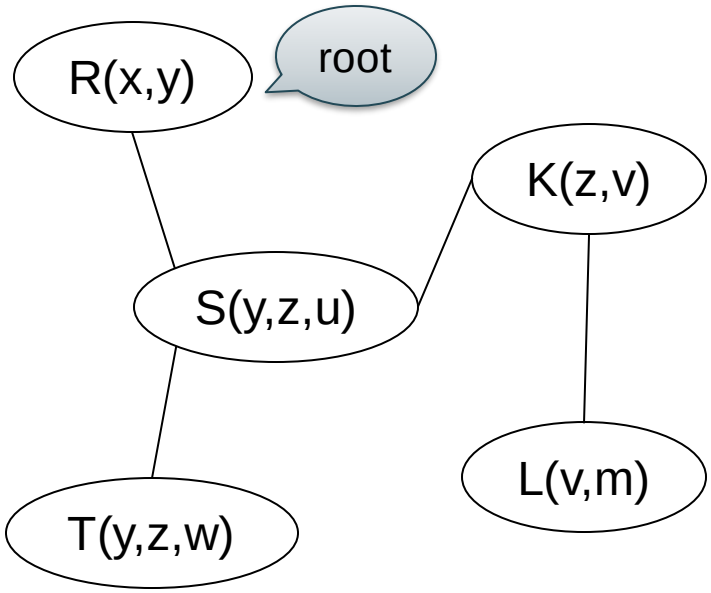


Semi-join as before

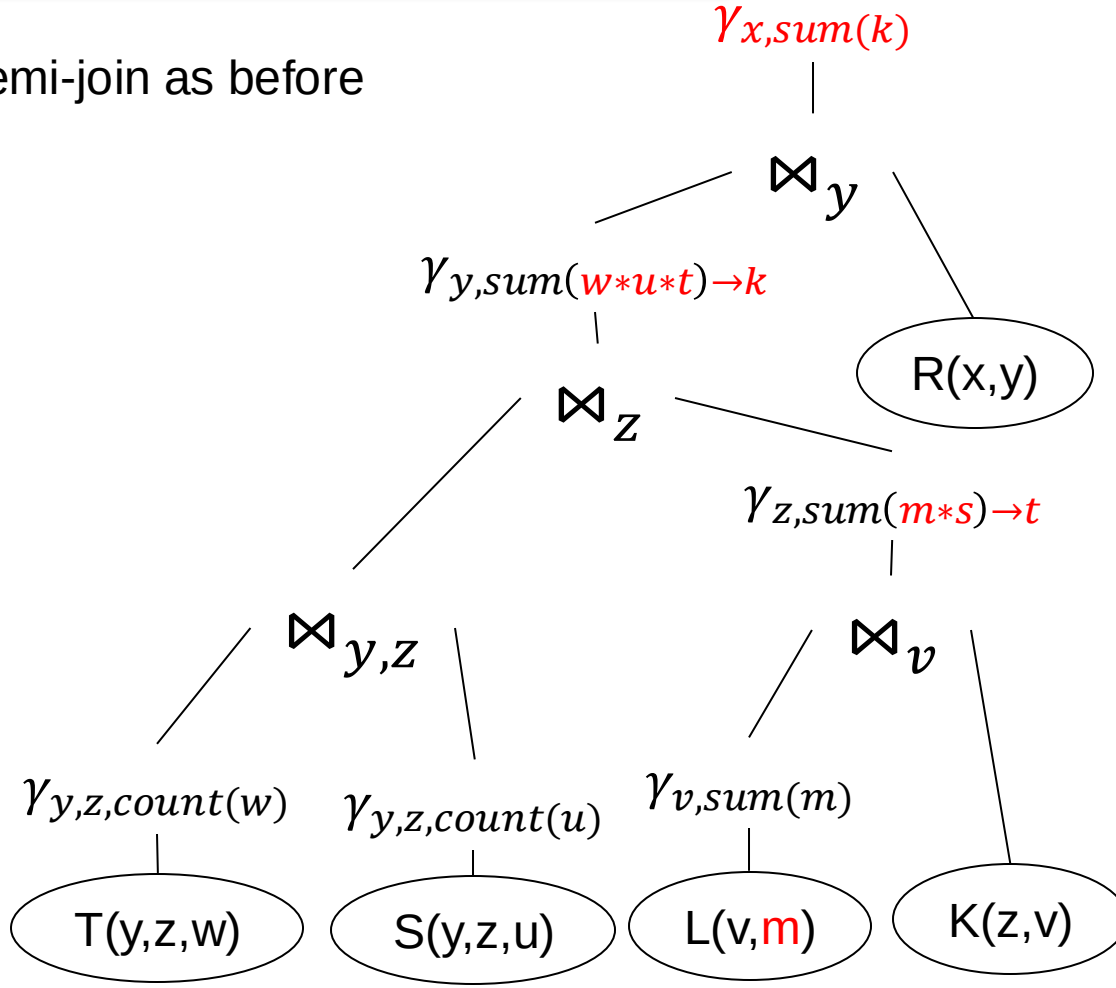


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

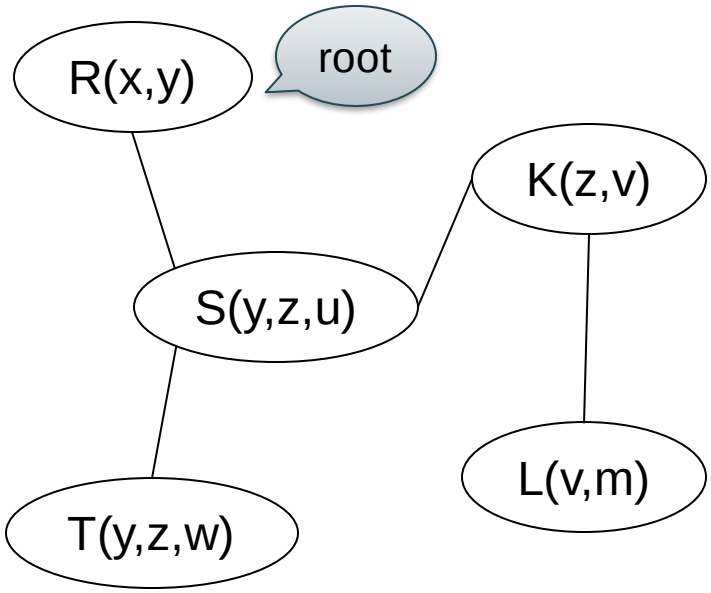


Semi-join as before

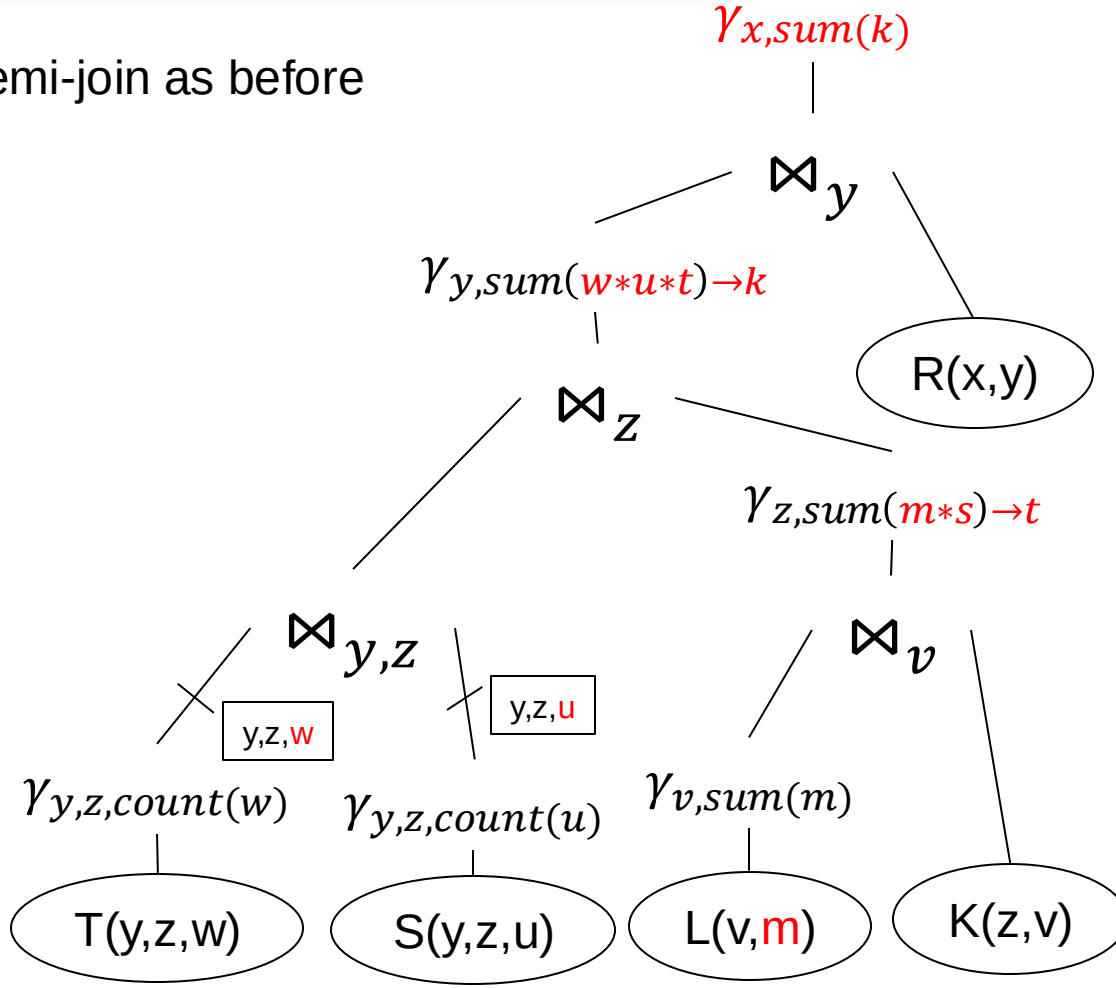


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

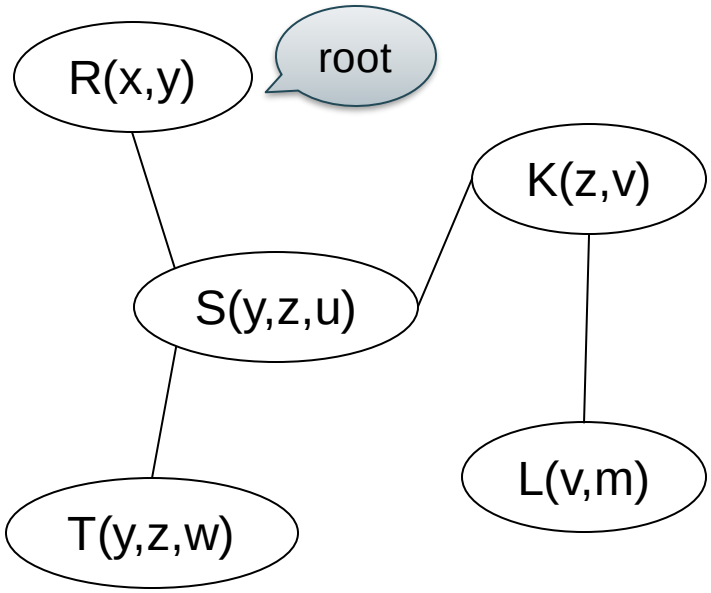


Semi-join as before

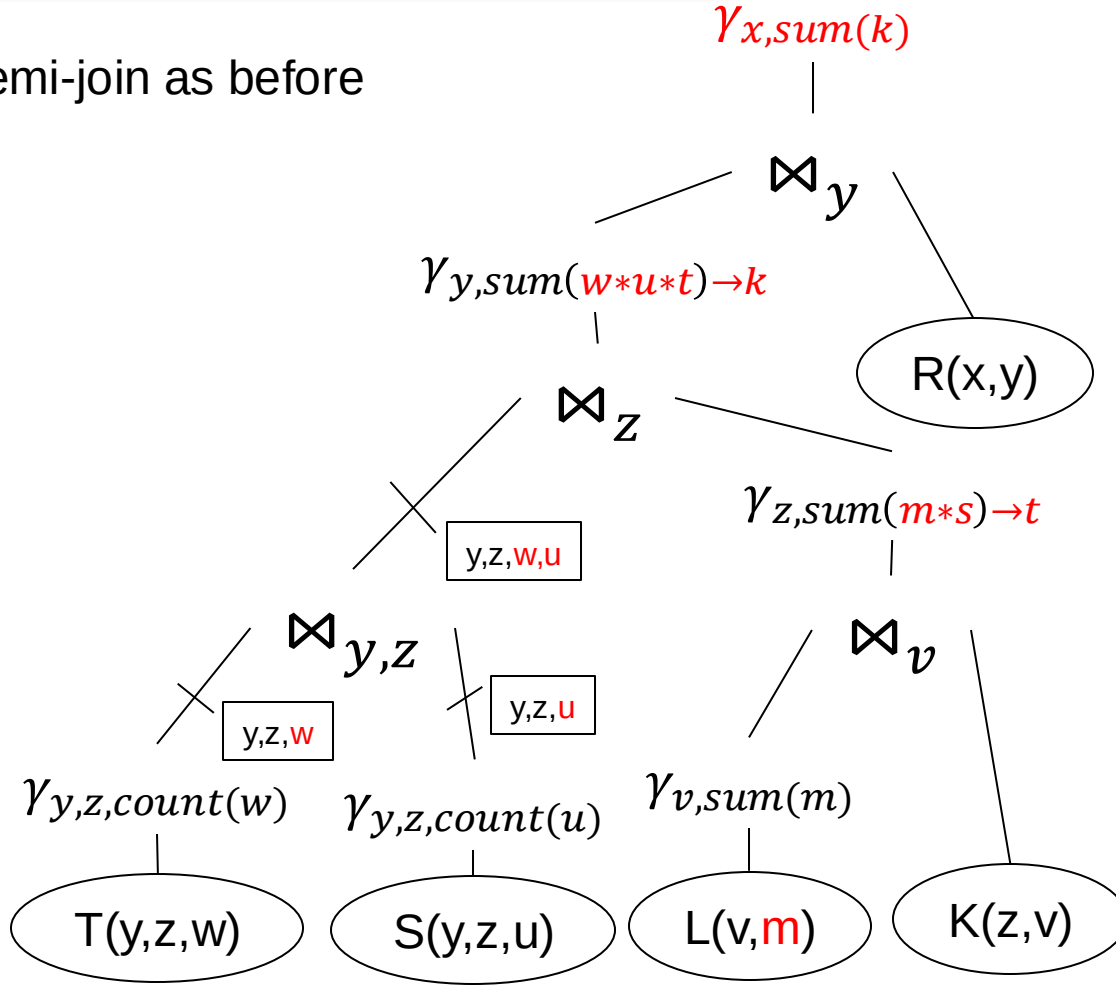


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

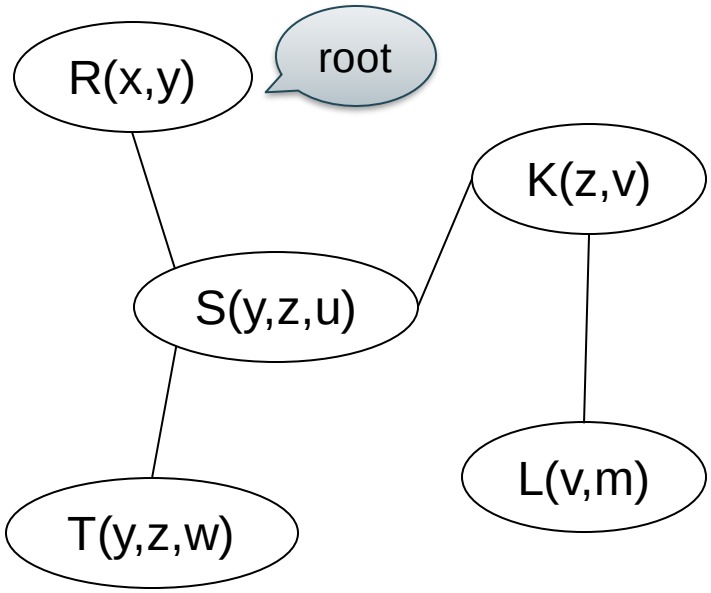


Semi-join as before

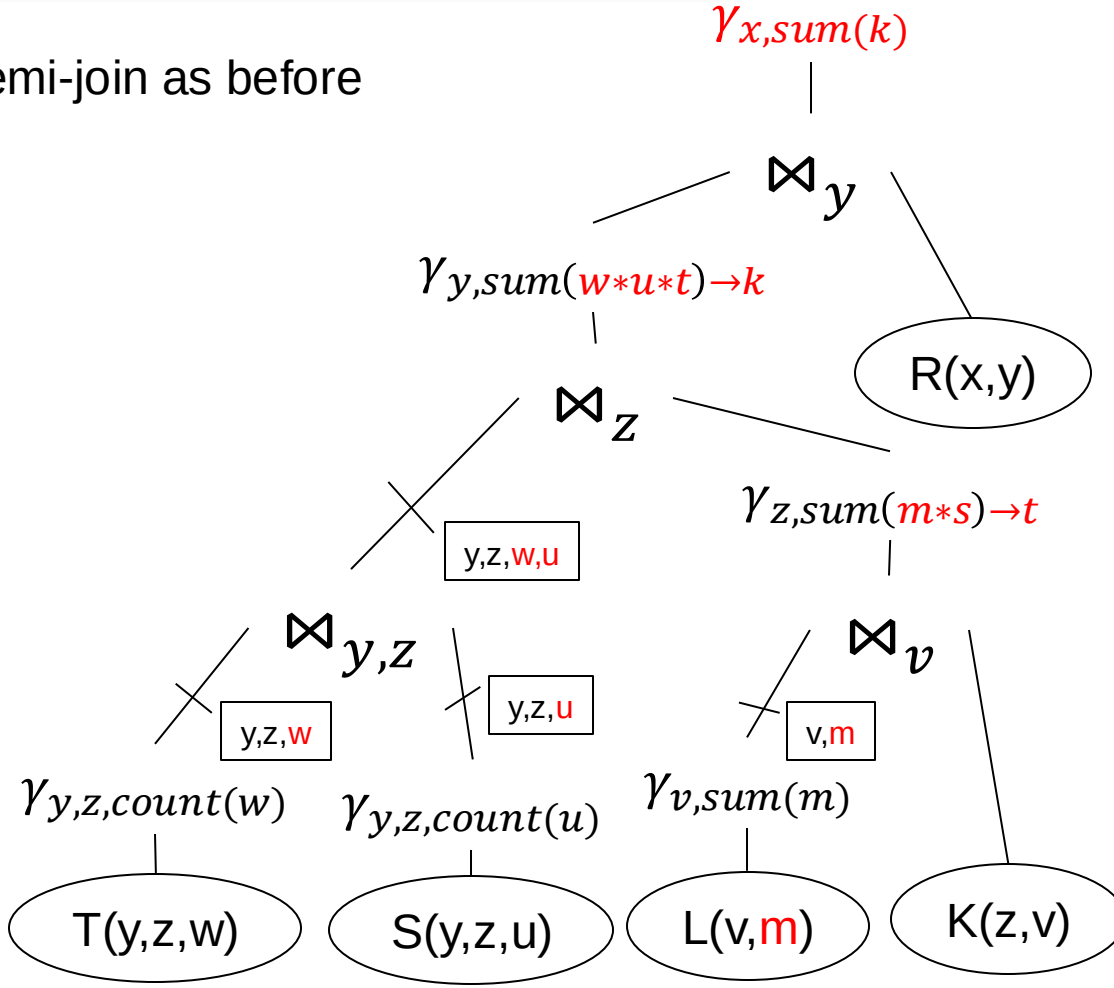


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

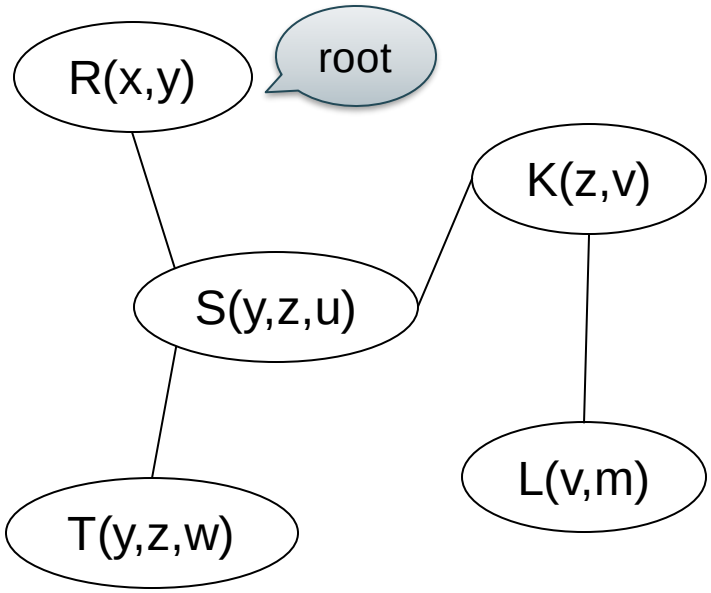


Semi-join as before

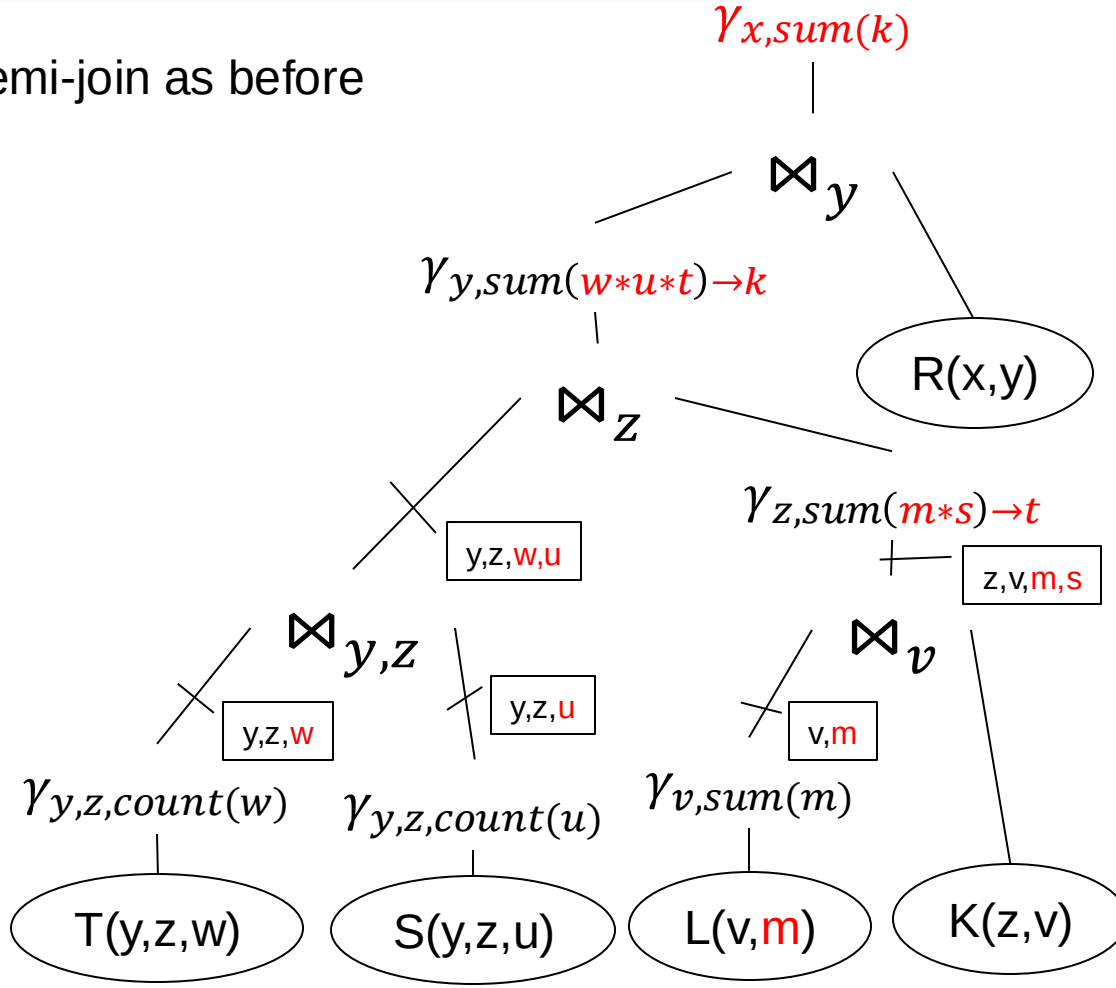


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

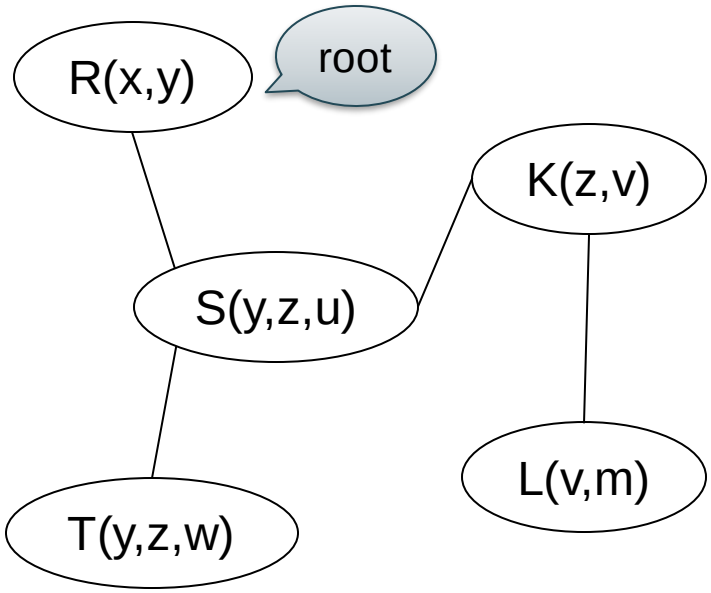


Semi-join as before

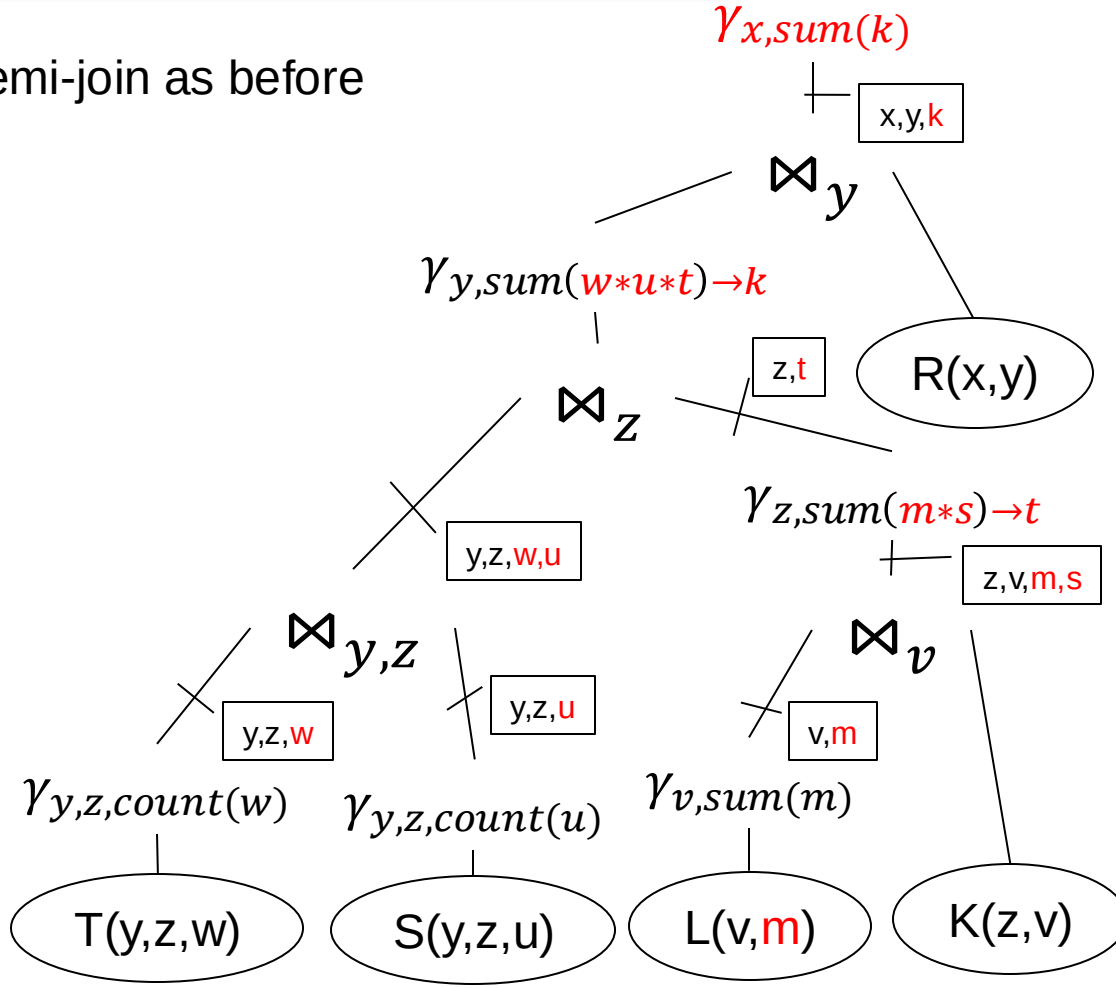


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

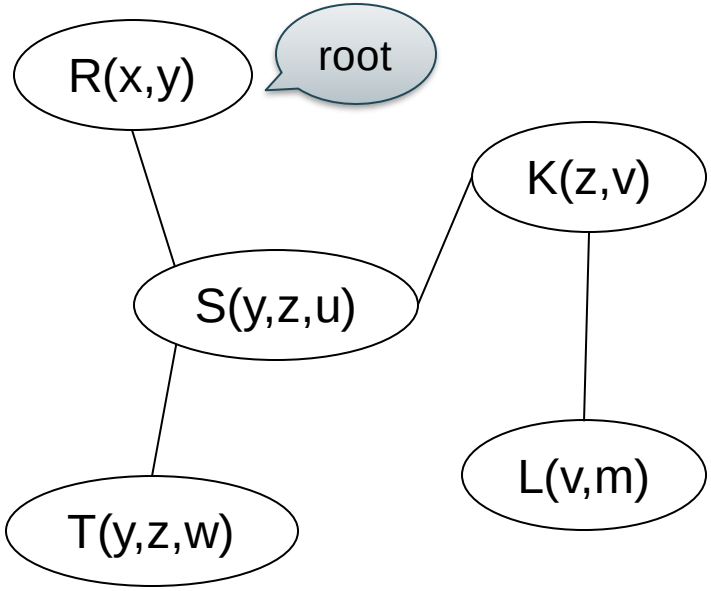


Semi-join as before

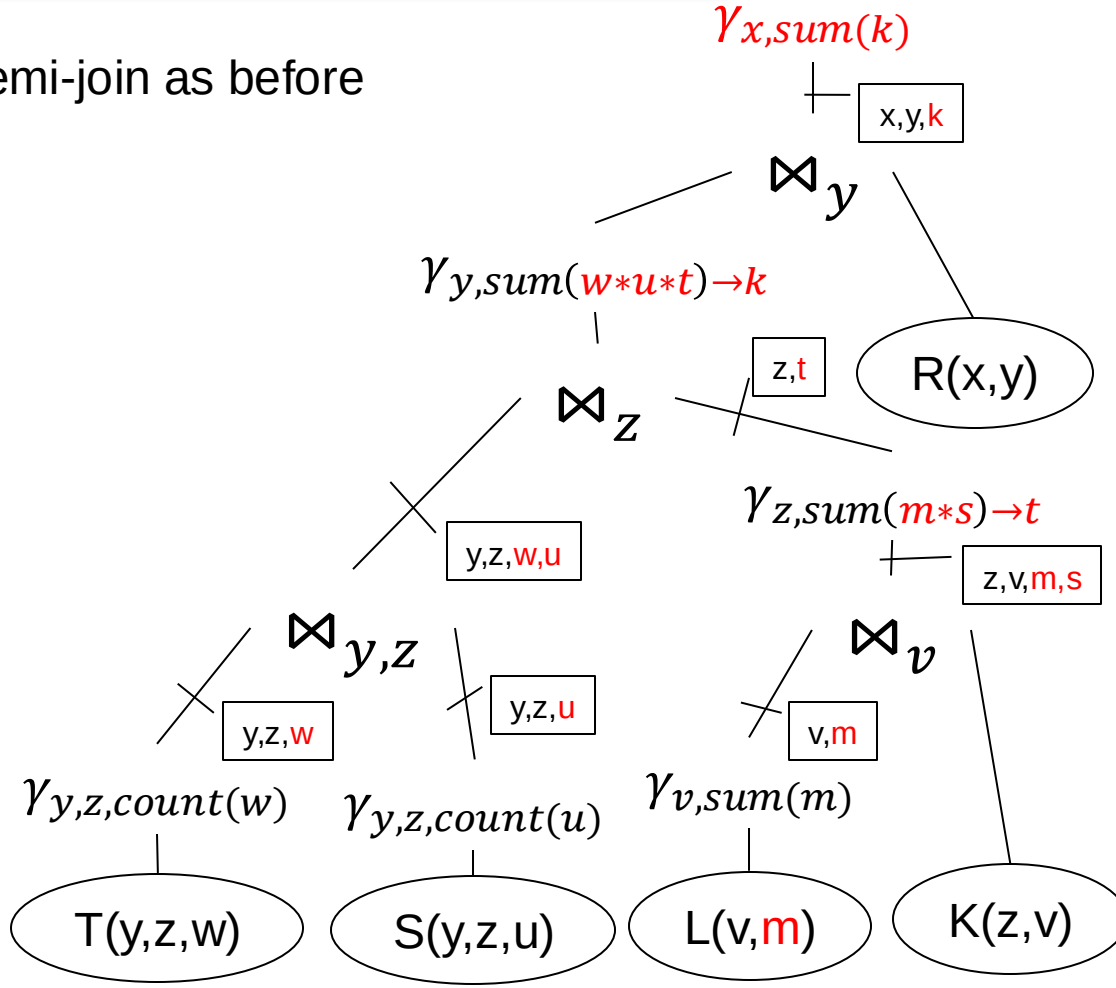


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



Semi-join as before

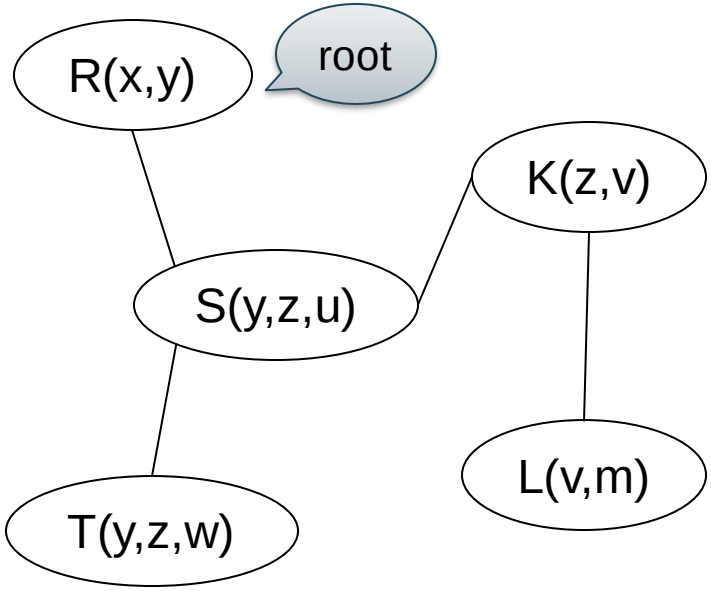


Runtime:

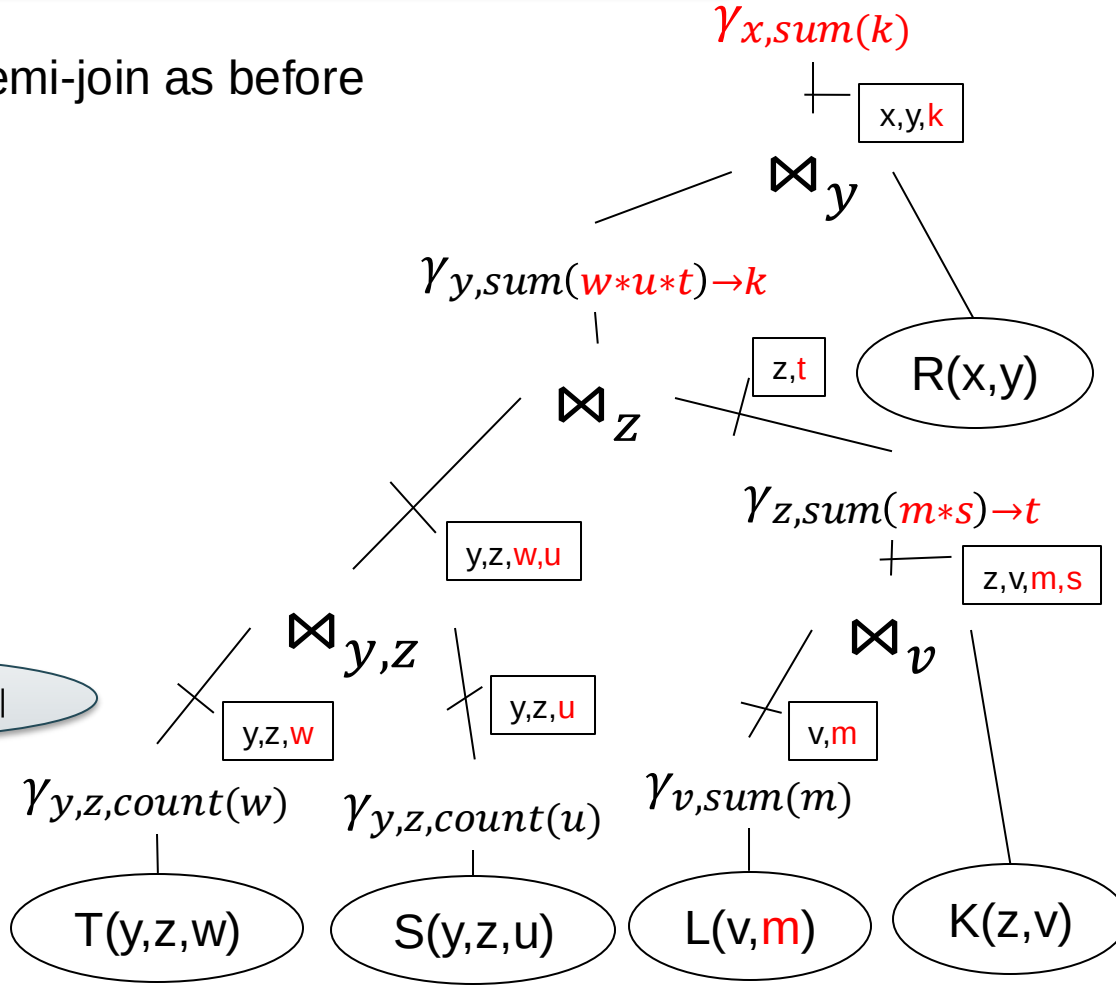
- Semi-joins: $\tilde{O}(|Input|)$
- Join/group-by: $\tilde{O}(|Input|)$

Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



Semi-join as before



$|Output| \leq |Input|$

Runtime:

- Semi-joins: $\tilde{O}(|Input|)$
- Join/group-by: $\tilde{O}(|Input|)$

Discussion

- Database optimizers rarely do semi-join reduction
  - When they do, they sometimes call it a *magic set optimization* (we'll explain next)
- Reason: when semi-join is ineffective, then it increases cost by a factor of 3

Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated

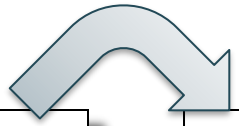
Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated

```
T(x,y) :- Parent(x,y)
T(x,y) :- T(x,z),Parent(z,y)
Q(y)   :- T('Alice',y)
```

Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated



```
T(x,y) :- Parent(x,y)
T(x,y) :- T(x,z),Parent(z,y)
Q(y)   :- T('Alice',y)
```

```
Q(y) :- Parent('Alice',y)
Q(y) :- Q(x),Parent(x,y)
```

Discussion

- Recent research aims at adapting Yannakakis' algorithm to avoid the factor 3 amplification