Scaleable Online Statistical Processing

Chris Jermaine
Computer and Information Sciences and Engineering Department

University of Florida, Gainesville

Performance: Aren't DBs Fast Enough?

You decide:

- Quick check of latest TPC-H results
- Spend $0.75 million to store 300GB (17TB of 36GB disks)
- Q21 *still* takes 18 minutes to answer in throughput test
- Got 10TB?
- You can spend $6.7 million to wait almost four hours for Q18

My Conclusion:

- Current AP solutions are not there
- Result: difficult to treat the DBMS as a sandbox

How To Address This Problem?

Not easy:

- AP studied intensively for 15 years
- We’ve only managed two serious solutions
  1. Pre-computed cubes - “OLAP” - often too restrictive
  2. “Glue-on” solution; add bitmaps to your DBMS
- And one interesting “new” idea
  - “Column-oriented” DBs
  - Though this really just fixes the problems with 2. above
- Is there another way to go?

Use Randomization!

Three key observations:

1. AP almost always statistical
2. Not always clear that $1.3745m differs from $1.3757m
3. Most exploratory queries are “wrong”

So, imagine the following DBMS:

- You ask **any** AP-style query, it gives you an immediate guess
- Guess bracketed by error guarantees
- Confidence region shrinks throughout computation
- Zero-width at query completion
- Total execution time same as classic RDBMS

Hasn't This Been Done?

UC Berkeley Control project:

- Developed online aggregation, ripple joins
- Got second two bullets to work (shrinking conf. region)
- But not scaleability, generality

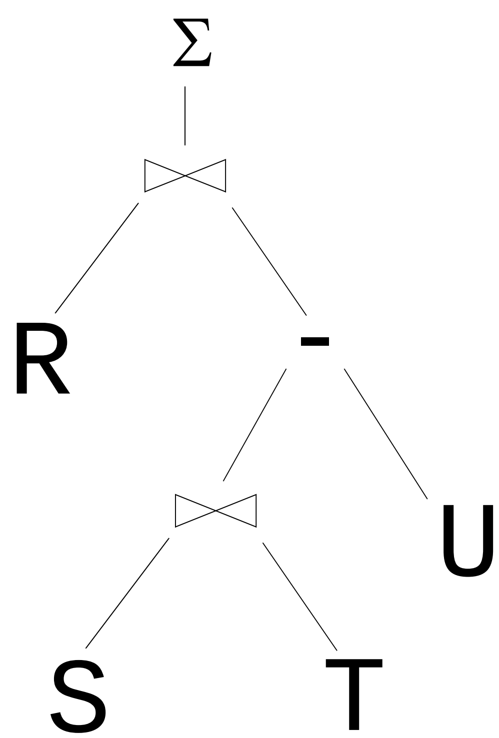
Aside: are synopsis structures relevant?

- Wavelets, histograms, sketches
- Great to study, but can they replace existing DBMS tech.?
  - Probably not; generally don't handle arbitrary queries
  - Bigger issue is that they are fixed precision

Our Goal

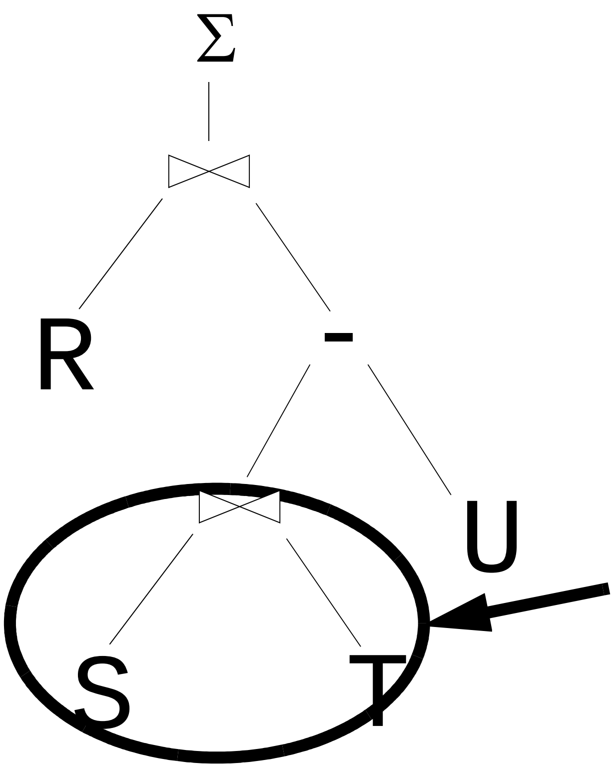
- Re-design database from ground up based on randomization
- Try to meet each of the five goals given previously:
  1. Any AP-style query $\rightarrow$ immediate guess
  2. Guess bracketed by error guarantees
  3. Shrinking confidence region
  4. Zero-width at completion
  5. Fast as classic RDBMS
- Resulting system is called *DB-Online*, or DBO for short

Seems Pretty Hard To Do...



Given an arbitrary plan like this one, how are you ever going to guess the answer from start to finish?

Seems Pretty Hard To Do...



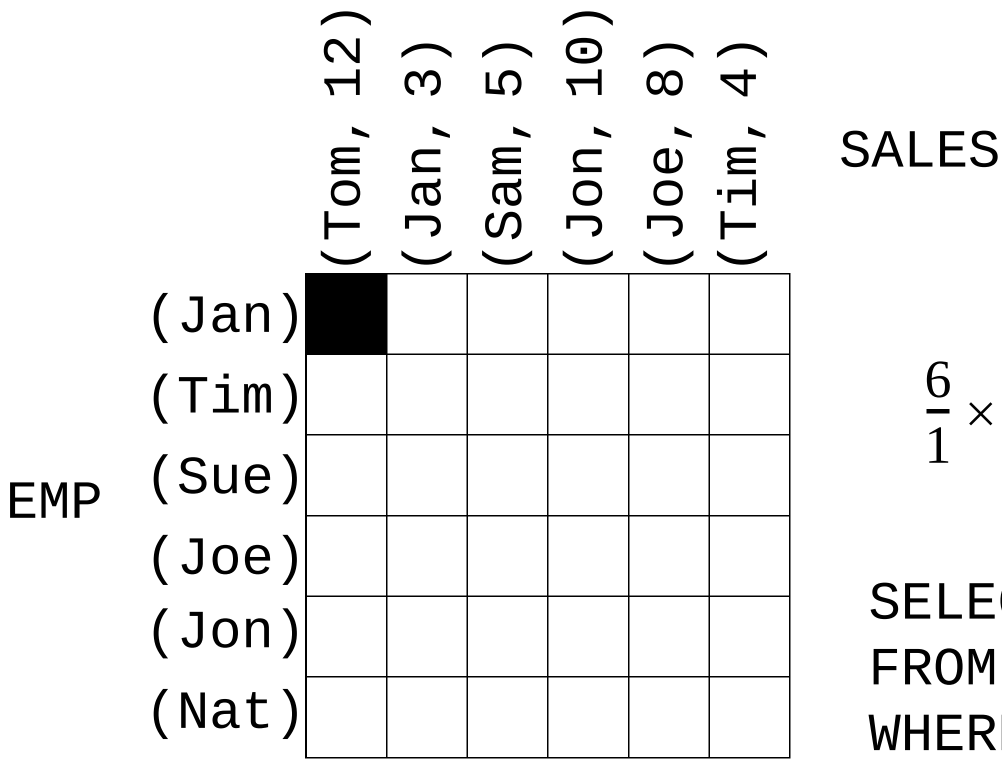
Given an arbitrary plan like this one, how are you ever going to guess the answer from start to finish?

Our first (imperfect) idea was to tackle a 2-table join, like this

Uses ripple join as basic building block...

The Ripple Join

- Scan randomly-permuted input relations in parallel

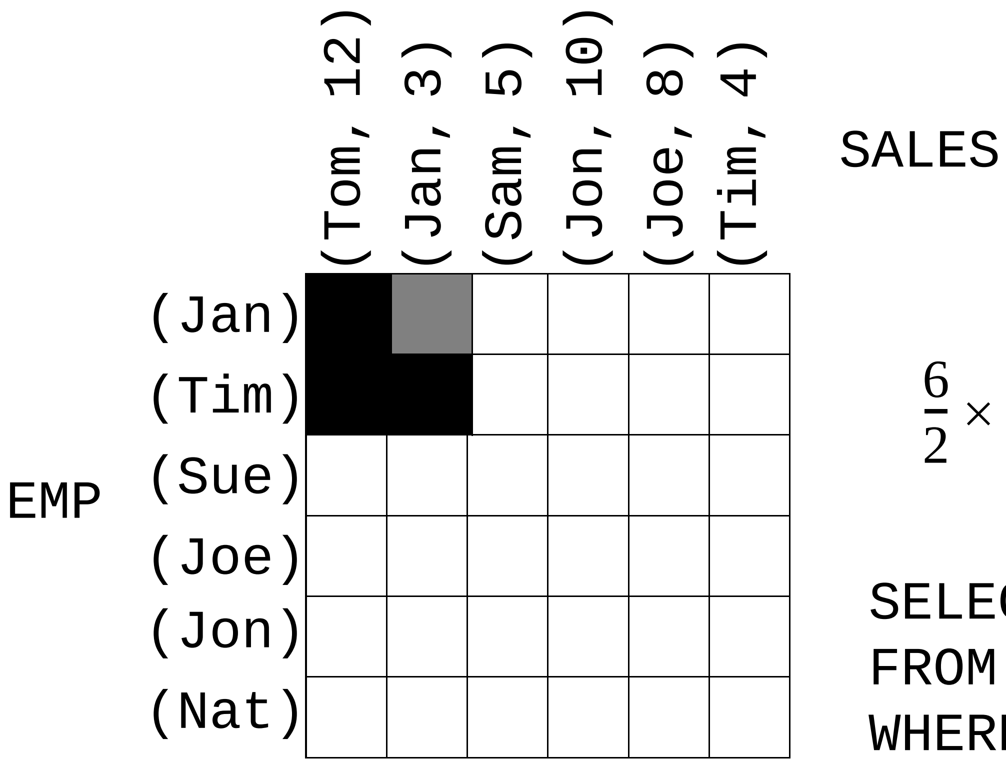


$$\frac{6}{1} \times \frac{6}{1} \times (0) = 0$$

```
SELECT SUM SALES.b
FROM EMP, SALES
WHERE EMP.a = SALES.a
```

The Ripple Join

- Scan randomly-permuted input relations in parallel

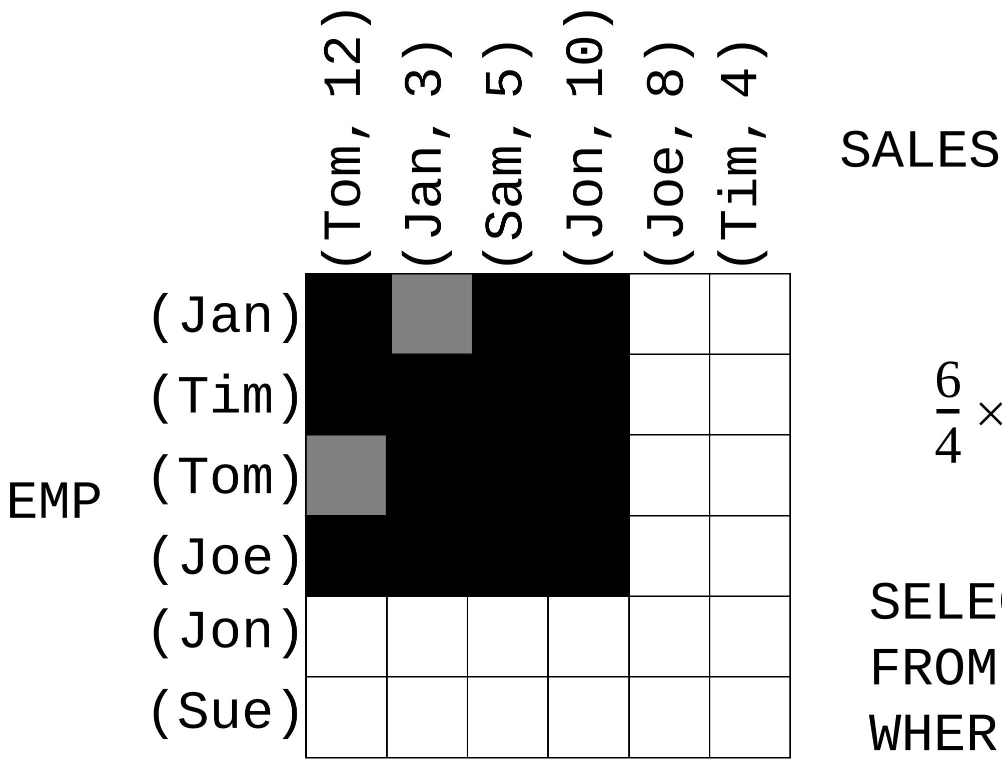


$$\frac{6}{2} \times \frac{6}{2} \times (3) = 27$$

```
SELECT SUM SALES.b
FROM EMP, SALES
WHERE EMP.a = SALES.a
```

The Ripple Join

- Scan randomly-permuted input relations in parallel

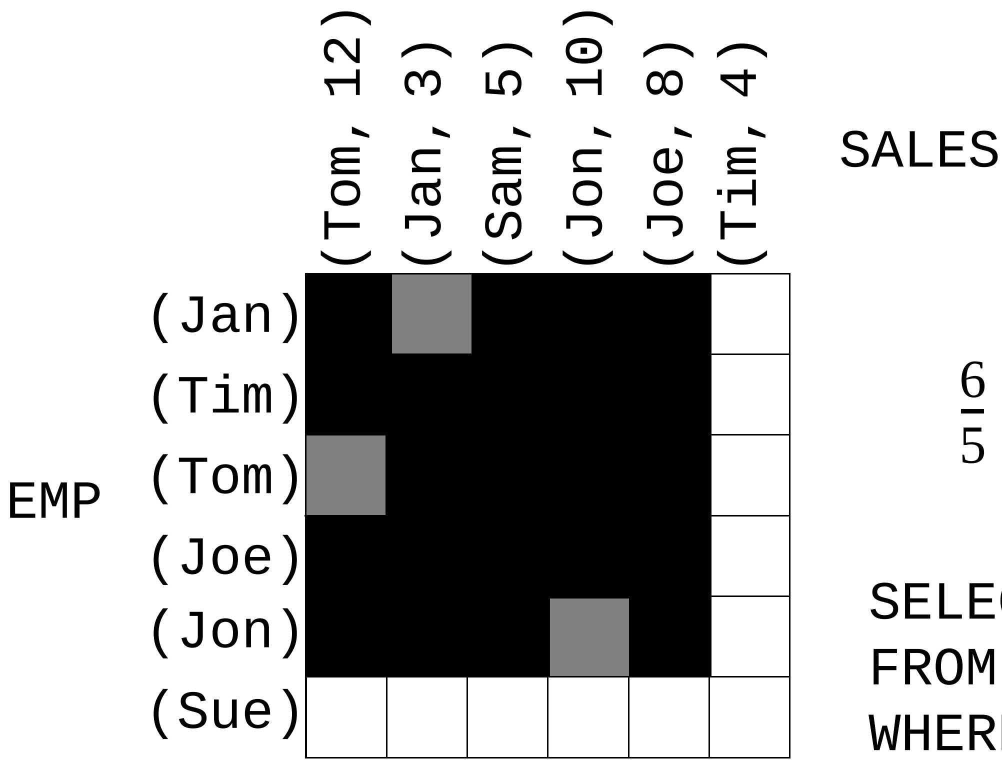


$$\frac{6}{4} \times \frac{6}{4} \times (3 + 12) = 33.75$$

```
SELECT SUM SALES.b
FROM EMP, SALES
WHERE EMP.a = SALES.a
```

The Ripple Join

- Scan randomly-permuted input relations in parallel

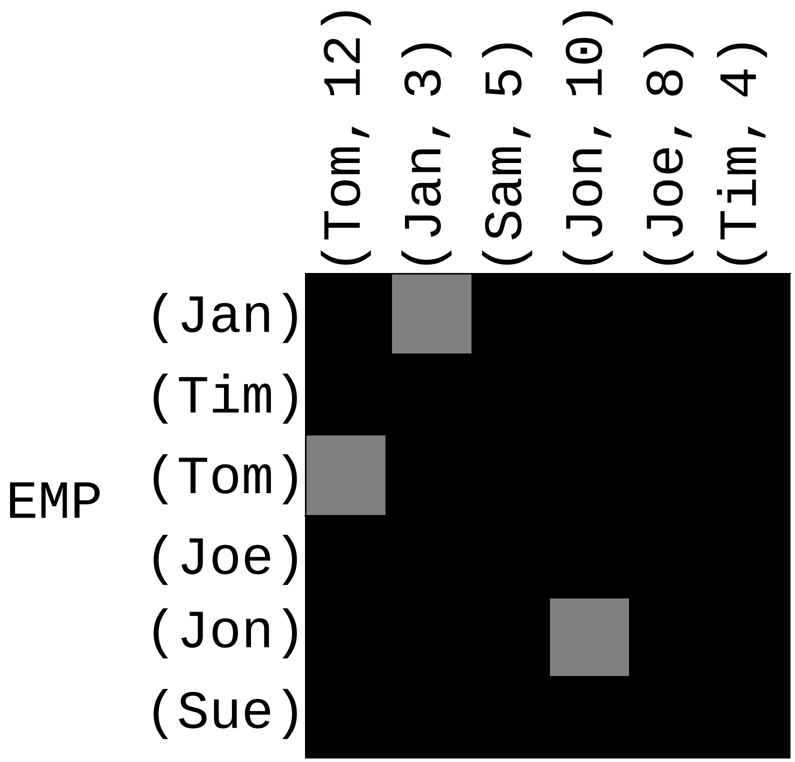


$$\frac{6}{5} \times \frac{6}{5} \times (3 + 12 + 10) = 36$$

```
SELECT SUM SALES.b
FROM EMP, SALES
WHERE EMP.a = SALES.a
```

The Ripple Join

- Scan randomly-permuted input relations in parallel



Final answer

$$\frac{6}{6} \times \frac{6}{6} \times (3 + 12 + 10) = 25$$

```
SELECT SUM SALES.b
FROM EMP, SALES
WHERE EMP.a = SALES.a
```

The Ripple Join

- Issue: only hashed RJ is practical
- But requires $e \in \mathtt{EMP}'$, $s \in \mathtt{SALES}'$ in memory

What do you do when you run out of memory?

Sort-Merge-Shrink Join

- 3-stage join algorithm

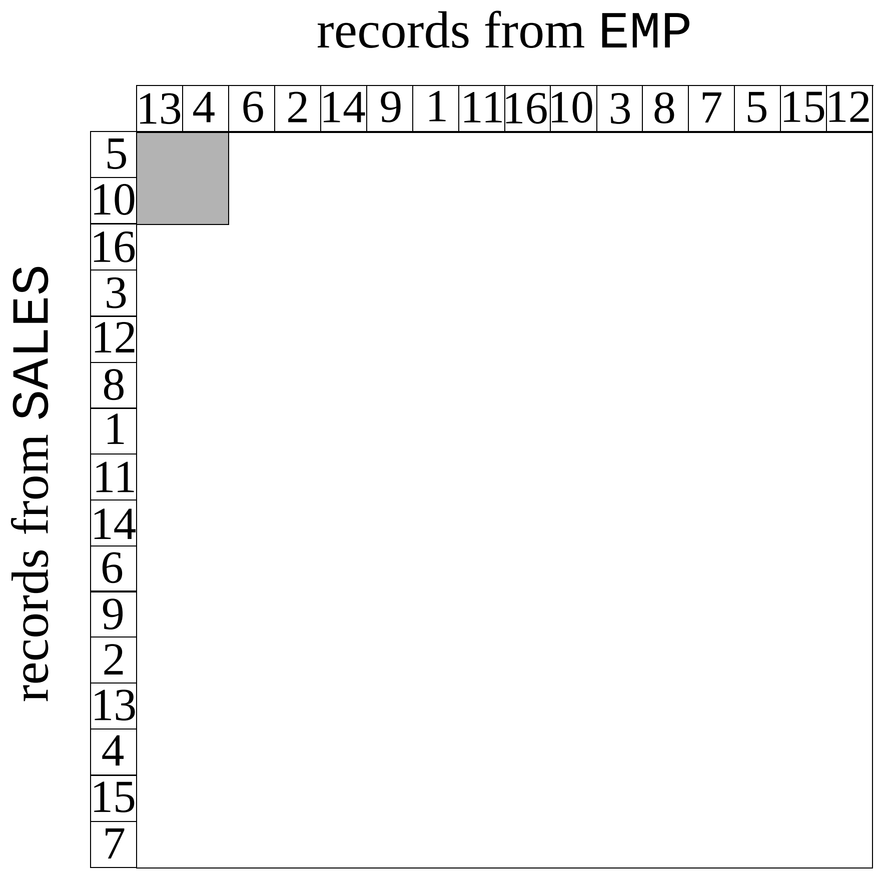
records from EMP

records from SALES

	13	4	6	2	14	9	1	11	16	10	3	8	7	5	15	12
5																
10																
16																
3																
12																
8																
1																
11																
14																
6																
9																
2																
13																
4																
15																
7																

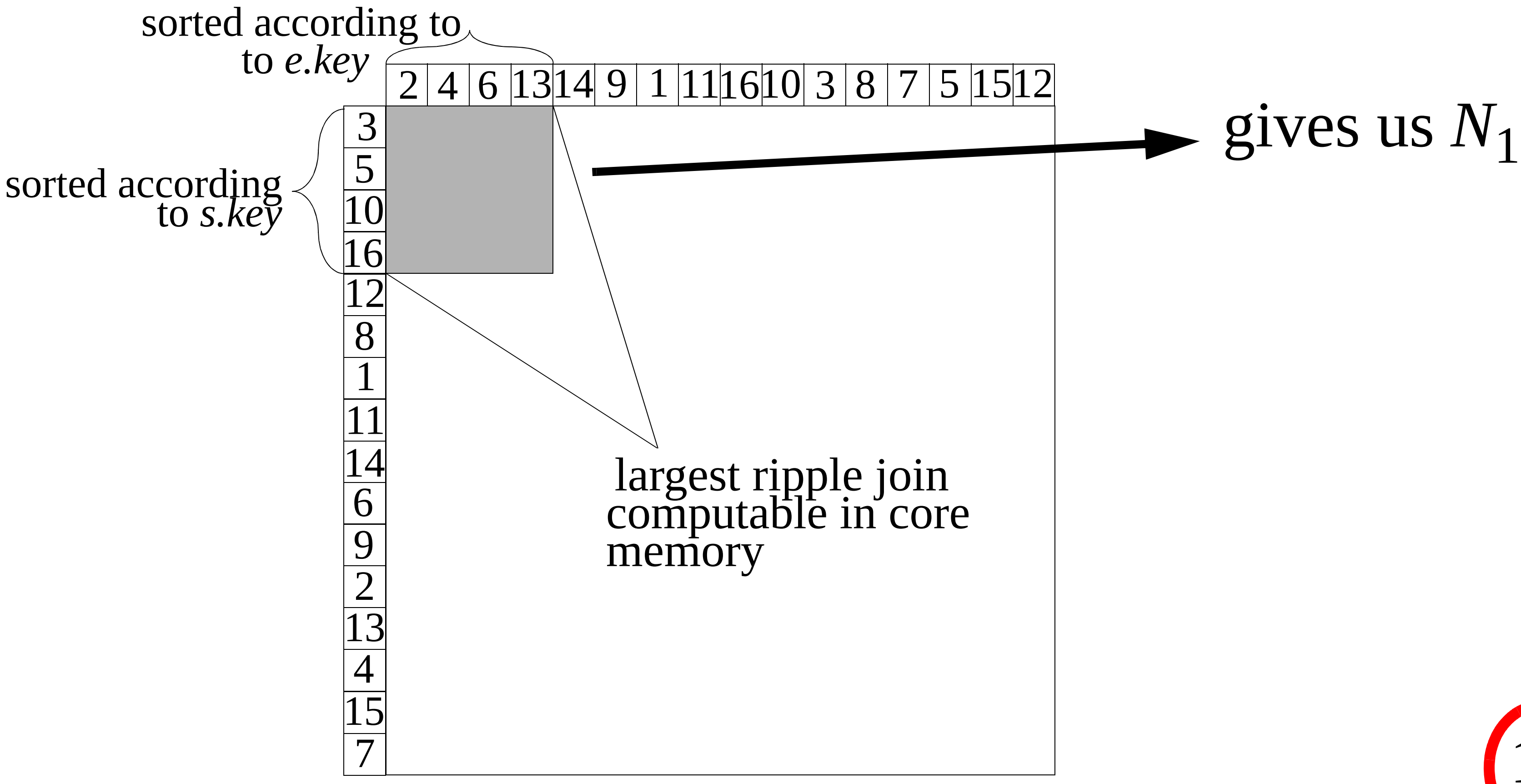
Sort-Merge-Shrink Join

- Sort phase: starts just like a ripple join



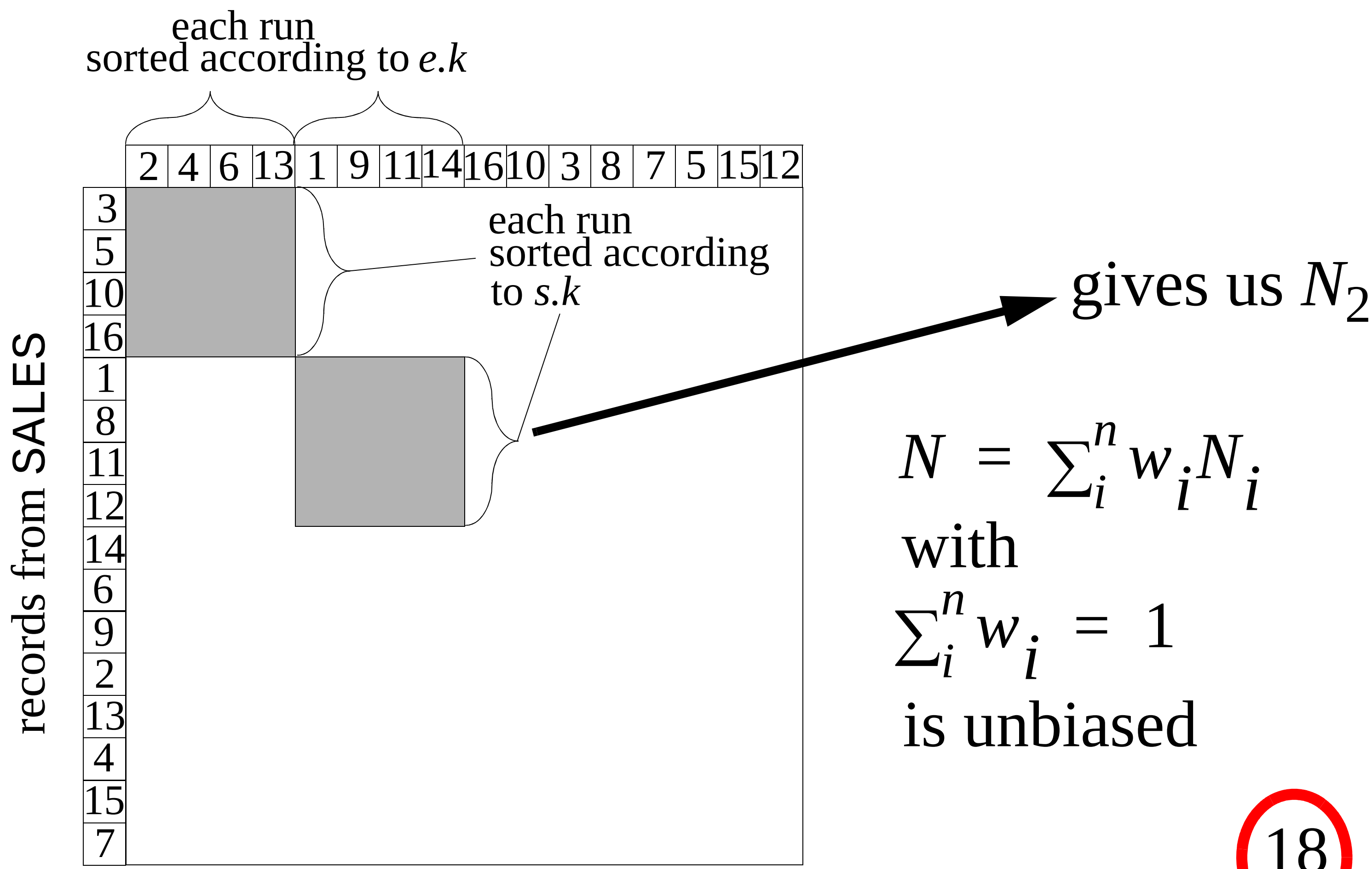
Sort-Merge-Shrink Join

- When memory fills, records from first RJ sorted and written back to disk



Sort-Merge-Shrink Join

- Process is then repeated



$N = \sum_i^n w_i N_i$

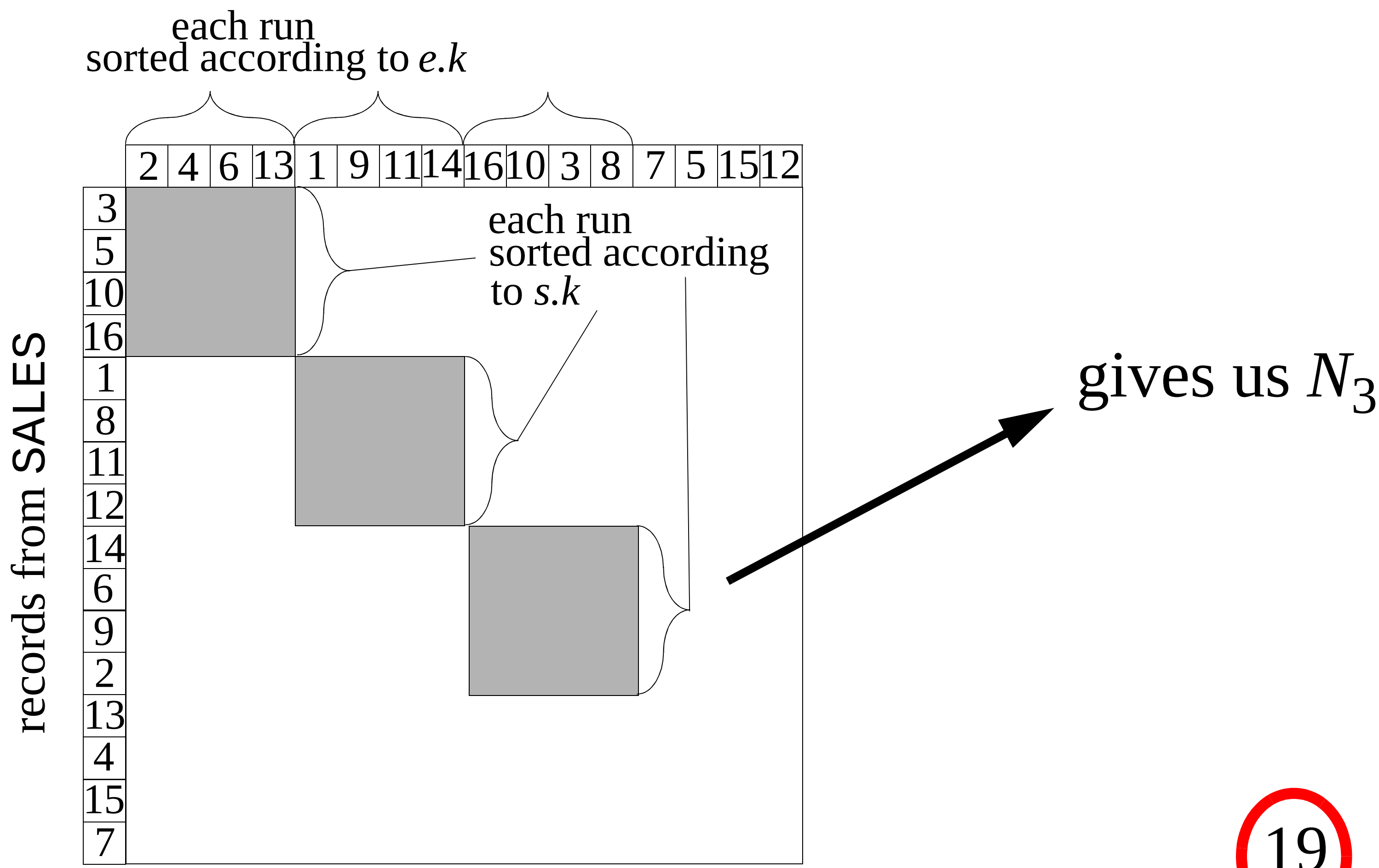
with

$\sum_i^n w_i = 1$

is unbiased

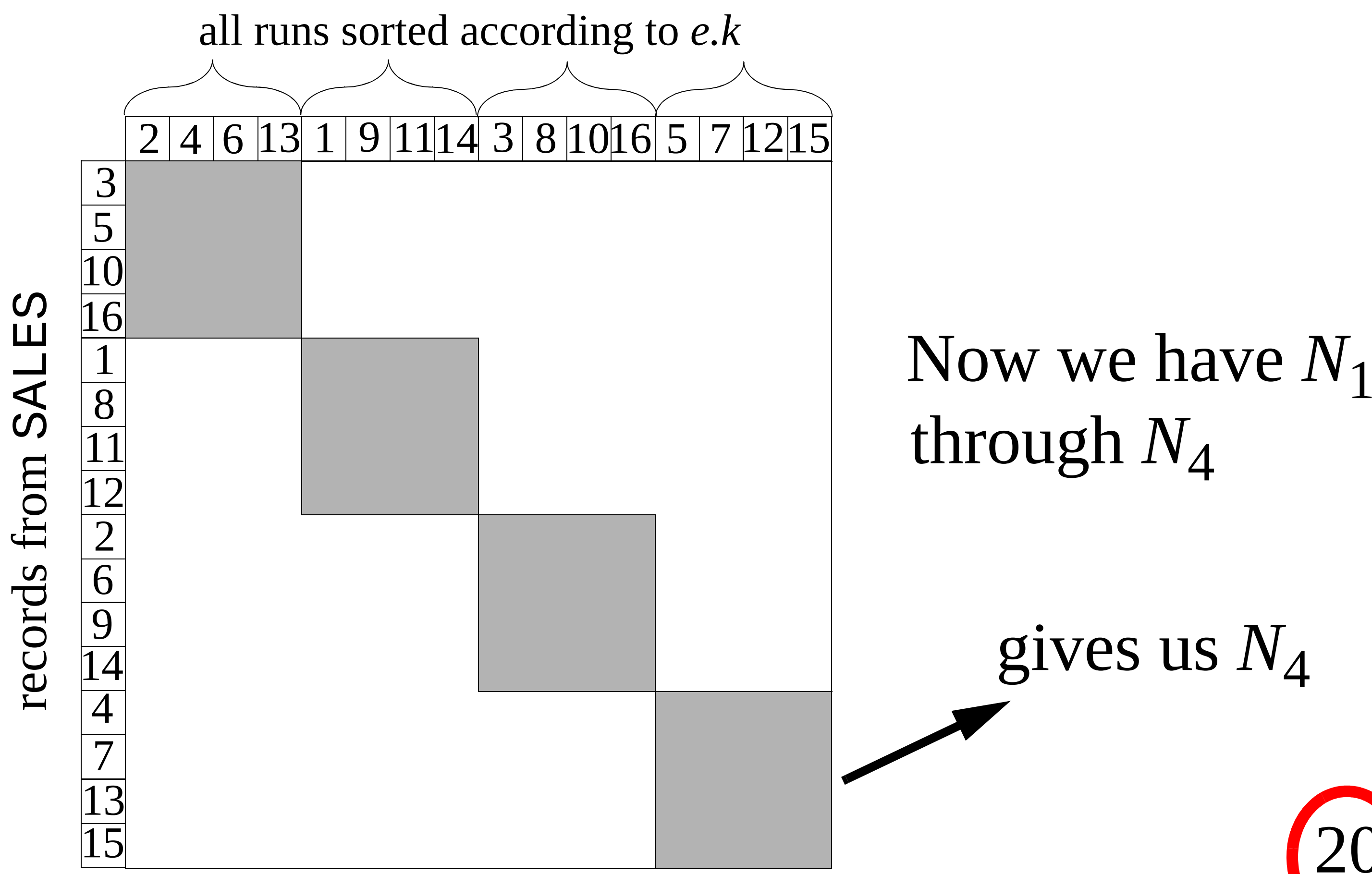
Sort-Merge-Shrink Join

- Process is then repeated



Sort-Merge-Shrink Join

- Until sort phase completes and all database records have participated in one RJ



Sort-Phase Statistical Considerations

- During sort phase, let N_i be estimate from ith ripple join - N_i unbiased so N is
- Know $Var(N)$ =

$$\sum_i^n w_i^2 Var(N_i) + \sum_j^n \sum_{i \neq j}^n w_i w_j Cov(N_i, N_j)$$

- Use this quantity to give bounds via CLT or other appropriate result

Computing the RJ Variance

$$\sum_i^n w_i^2 Var(N_i) + \sum_j^n \sum_{i \neq j}^n w_i w_j Cov(N_i, N_j)$$

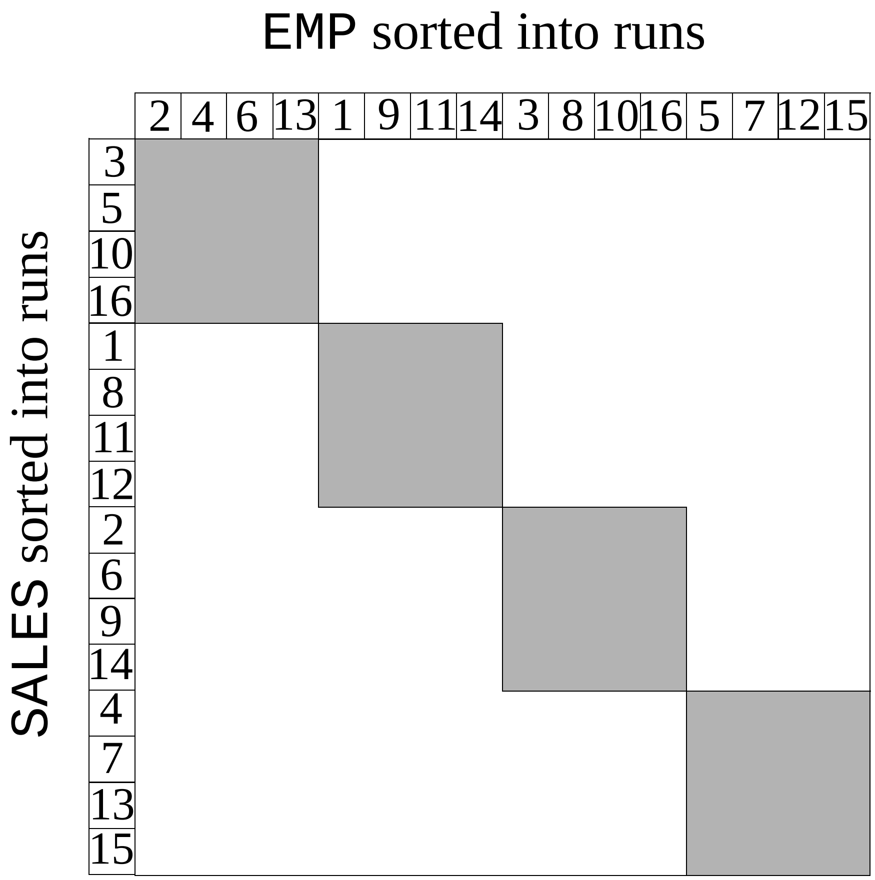
- $Var(N_i)$ can be estimated using formulas from HH99 (large sample w. replacement)
- Or can use the nasty (yet exact) permutational formula we derived
- w_i computed via a straightforward quadratic optimization problem

Computing the RJ Variance

$$\sum_i^n w_i^2 Var(N_i) + \sum_j^n \sum_{i \neq j}^n w_i w_j Cov(N_i, N_j)$$

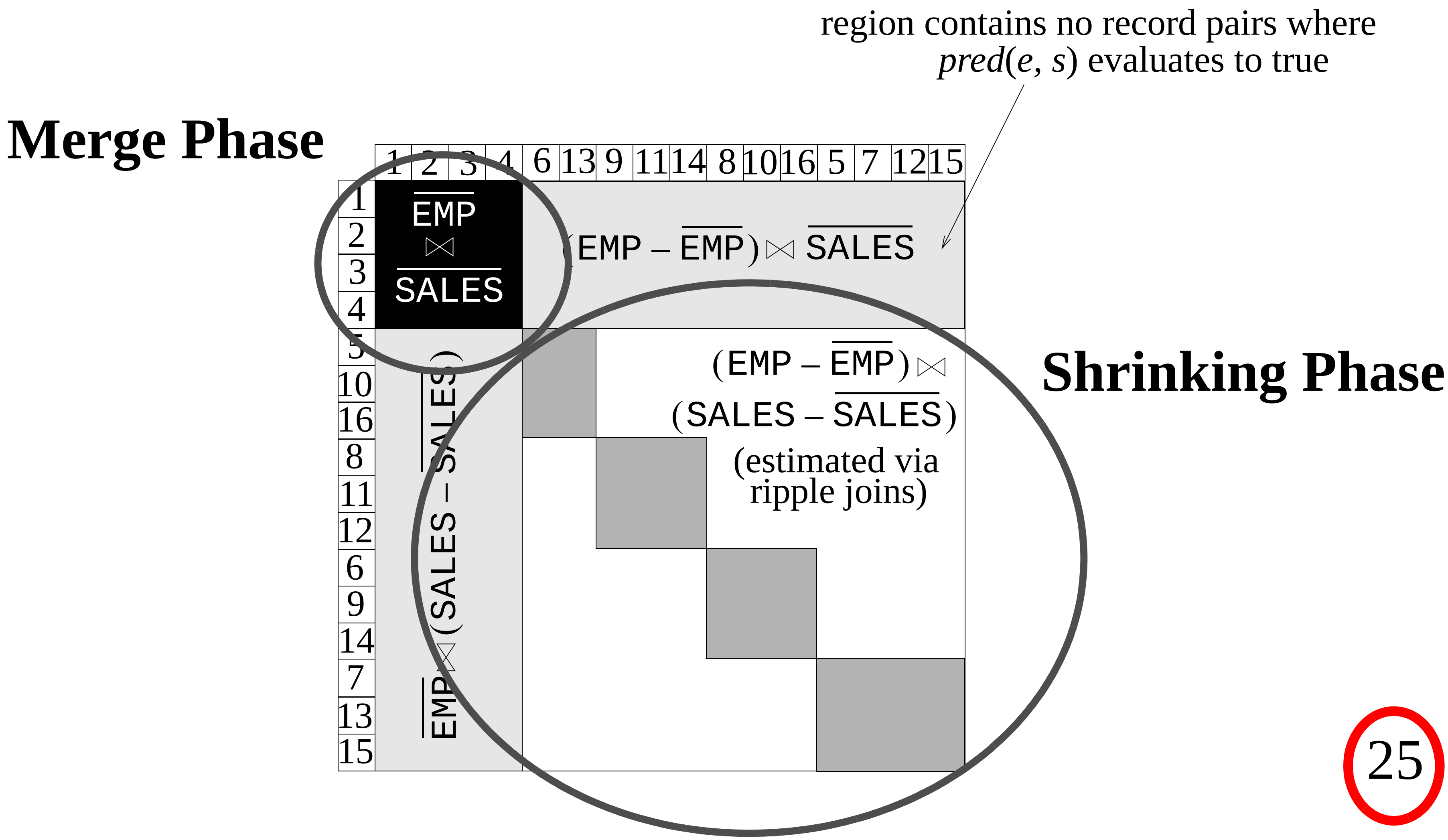
- How about the covariance?
- Can be estimated using the nasty formulas we derived
- Or can possibly ignore it... usually negative anyway

How To Finish the Join?



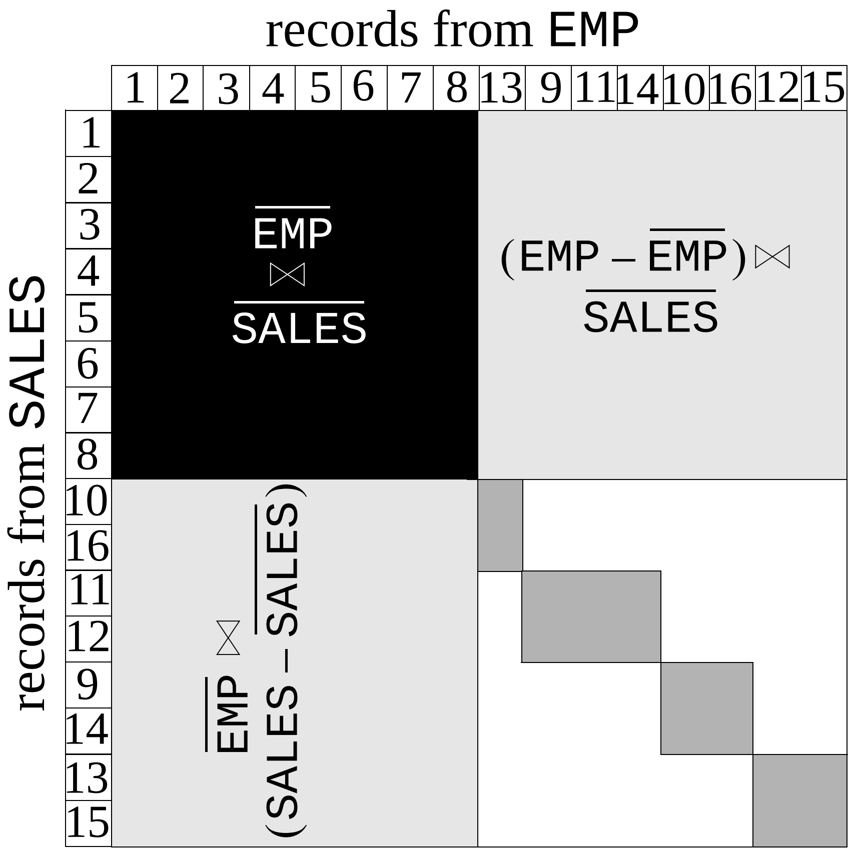
Sort-Merge-Shrink Join

- The merge/shrinking phases begin...



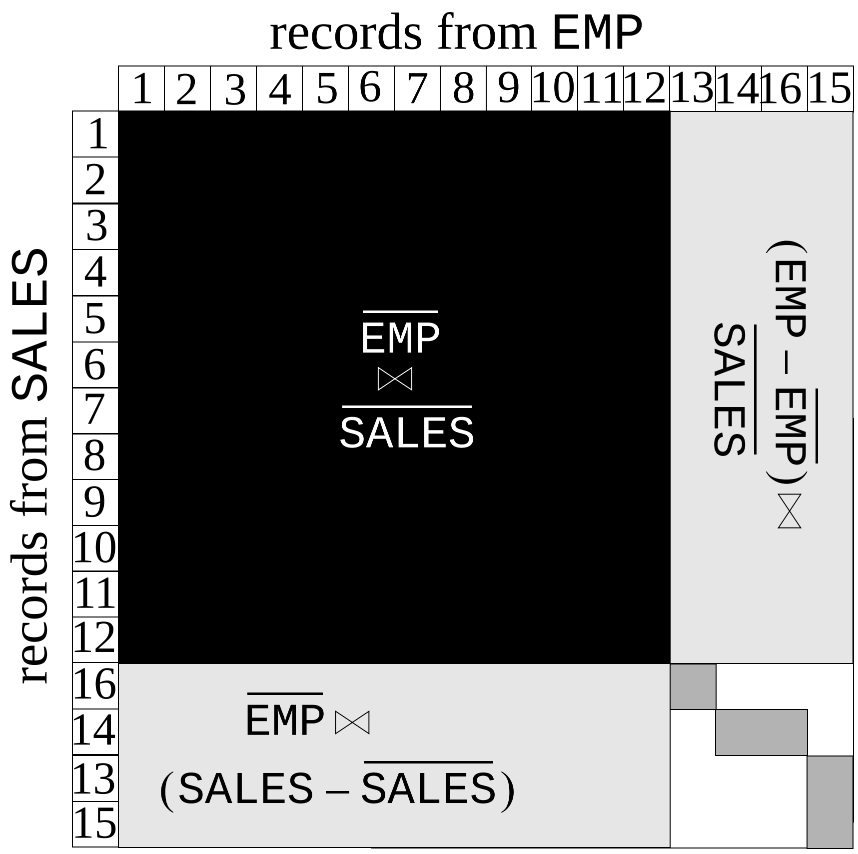
Sort-Merge-Shrink Join

- Merged portion of data space increases



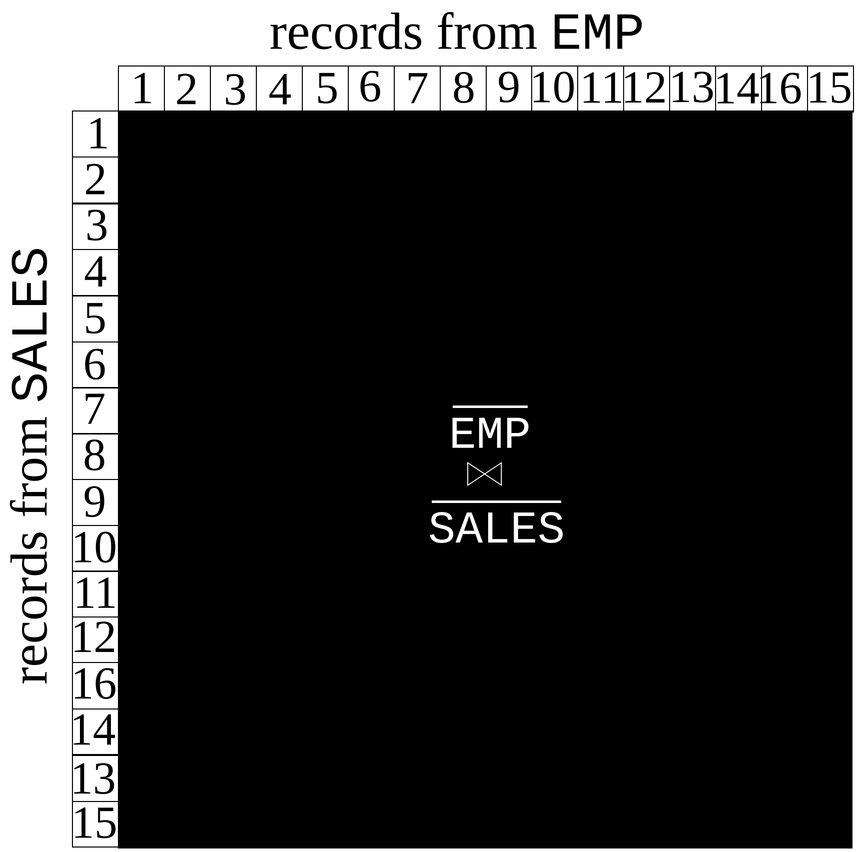
Sort-Merge-Shrink Join

- Until it dominates the data space



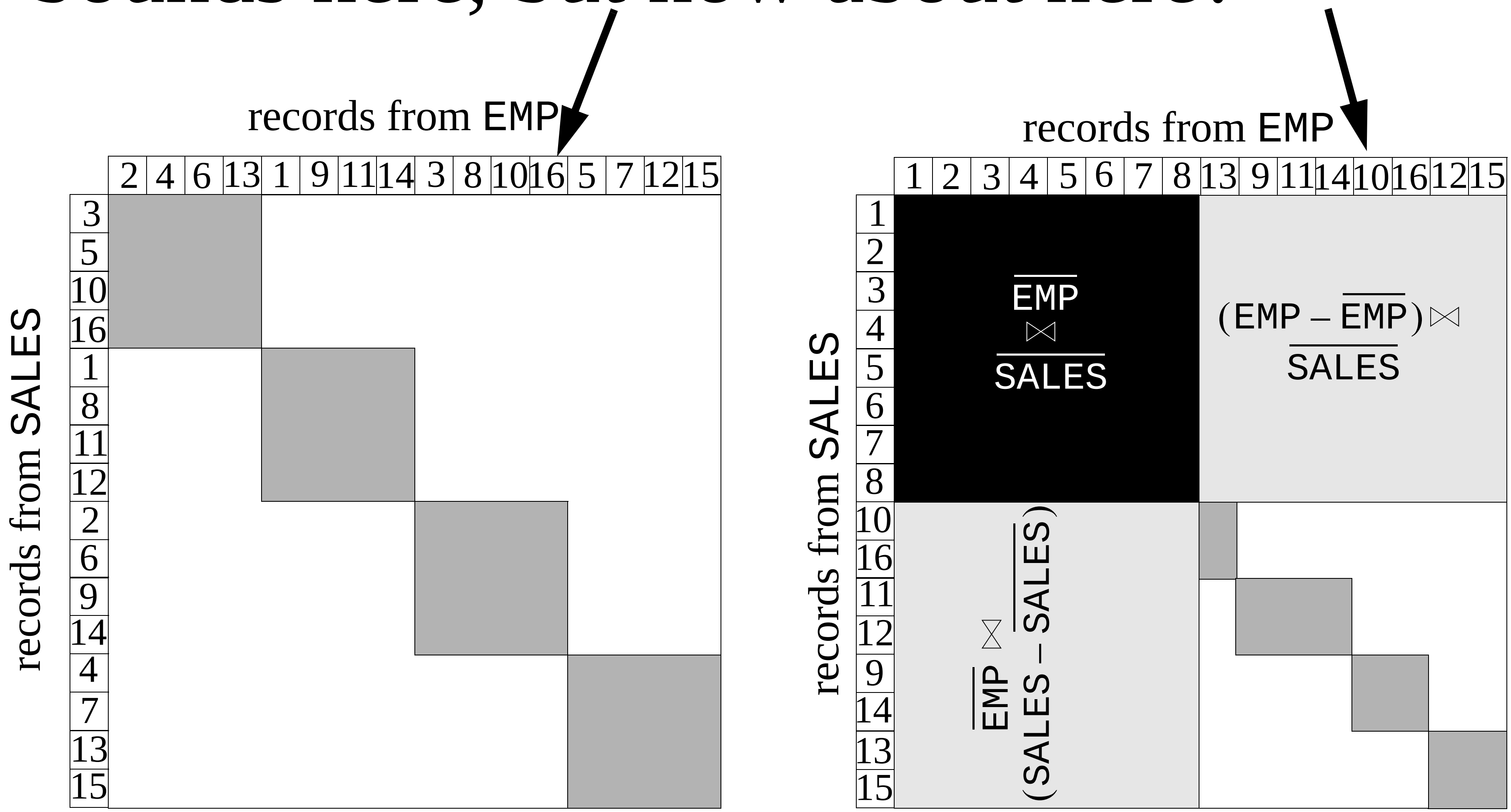
Sort-Merge-Shrink Join

- Then the join completes



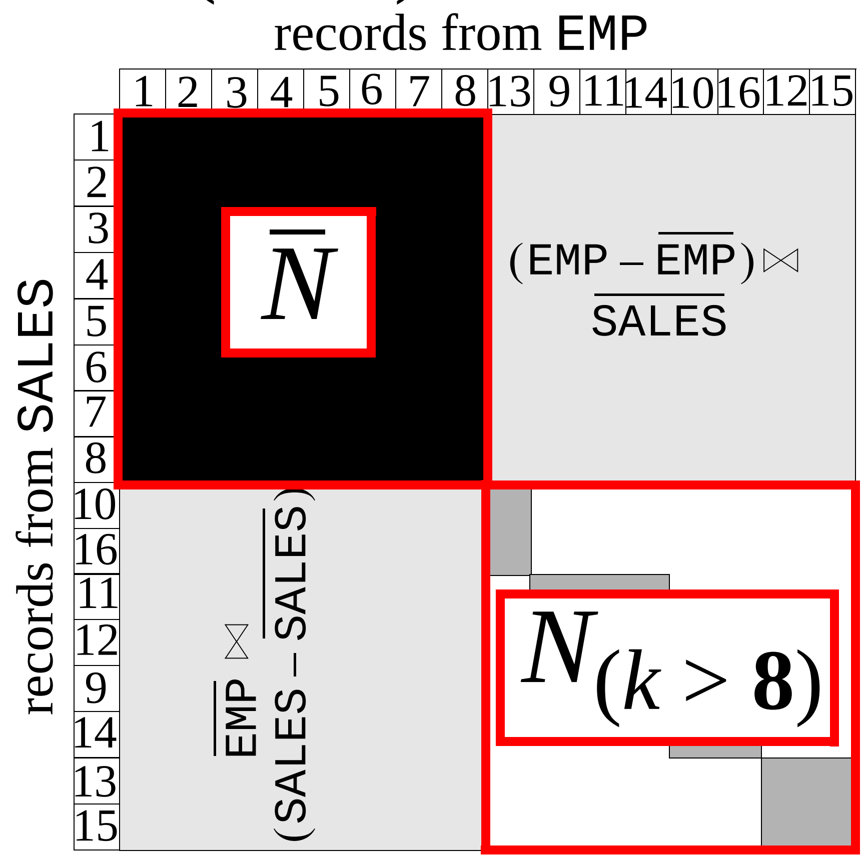
Merge/Shrink Statistics

- So we can get an estimate and confidence bounds here, but how about here?



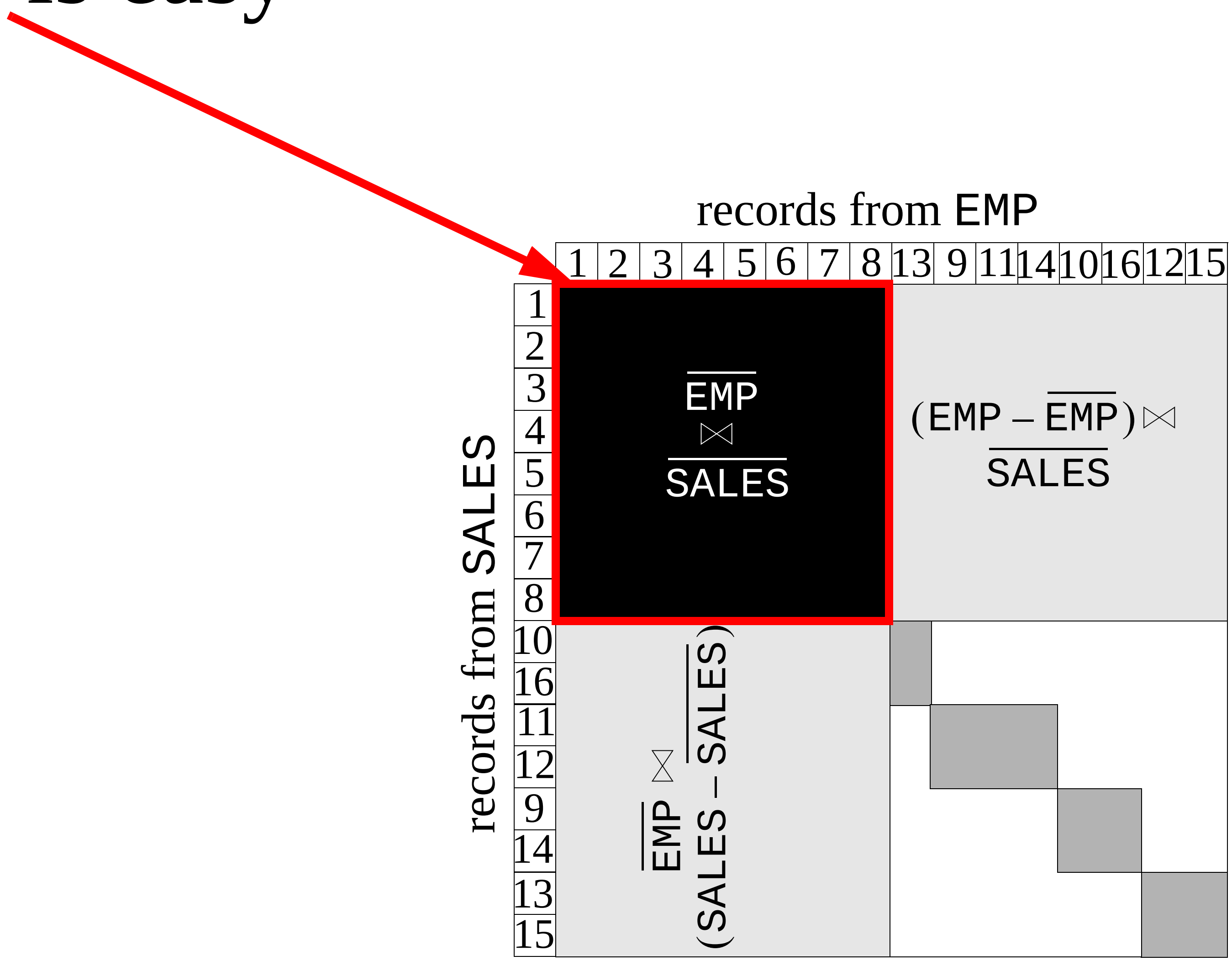
Merge/Shrink Statistics

- $\bar{N} + N_{(k > \mathbf{k})}$ is unbiased for query result
- $Var(\bar{N} + N_{(k > \mathbf{k})})$ is $Var(N_{(k > \mathbf{k})})$



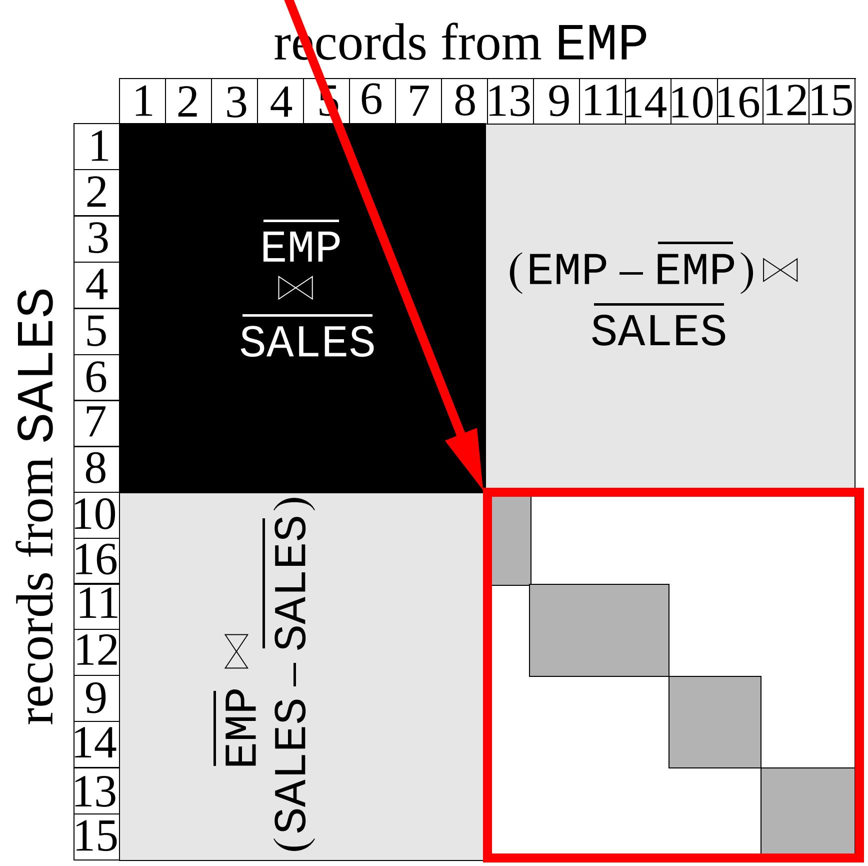
Merge/Shrink Statistics

- Getting $\overline{N}$ is easy



Merge/Shrink Statistics

- But $N_{(k > \mathbf{k})}$, $Var(N_{(k > \mathbf{k})})$ is harder. Why?
  - Each RJ has been written to disk, so no way to compute $N_{(k > \mathbf{k})}$ w/o re-reading data



Merge/Shrink Statistics

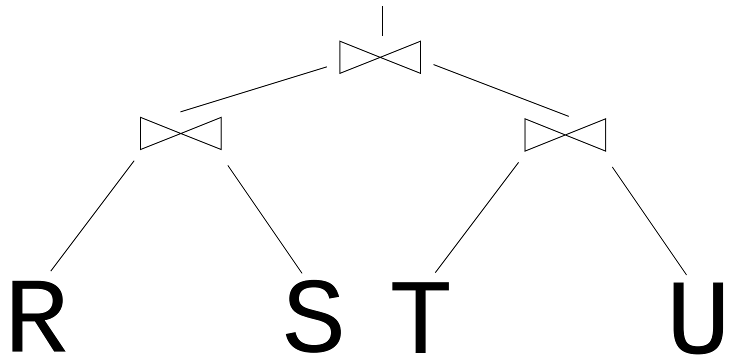
- Solution: precompute and store $N_{(k > \mathbf{k})}$ and $Var(N_{(k > \mathbf{k})})$ for "enough" **k** values
- Output new estimate every time you merge past one of those values
- Done via reverse ripple join during sort phase, before each run written back to disk
- Small storage: ~ 3600 × 100 × 12 bytes

What Do We Find When We Run This?

- Example, joining two, 20GB tables:
  - Memory gone ~20 seconds, w. one disk at 50MB/sec
  - Confidence bounds halve after 100 secs
  - Confidence bounds halve again 1000 secs
  - 1/8 to 1/10 as wide after 3000 seconds

SMS Join Is Nice, Won't Work in DBO

In retrospect, a neat algorithm of questionable utility... say you have:

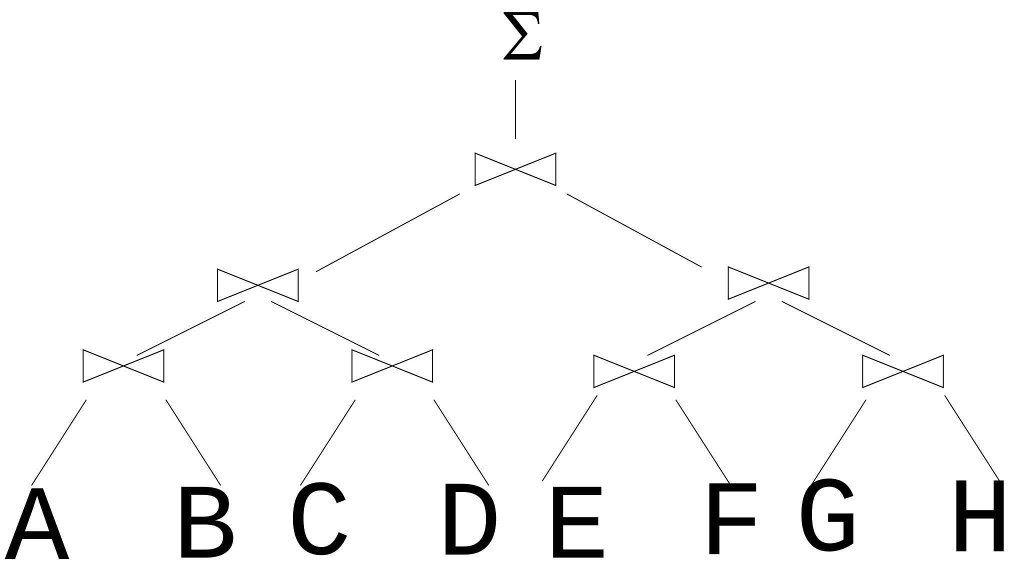


Can SMS join R and S, but rather useless for guessing the final query answer...

- Output tuples not produced randomly, so no subsequent use
- Can do 3+ tables in one SMS join, but only if can use the same sort order for each

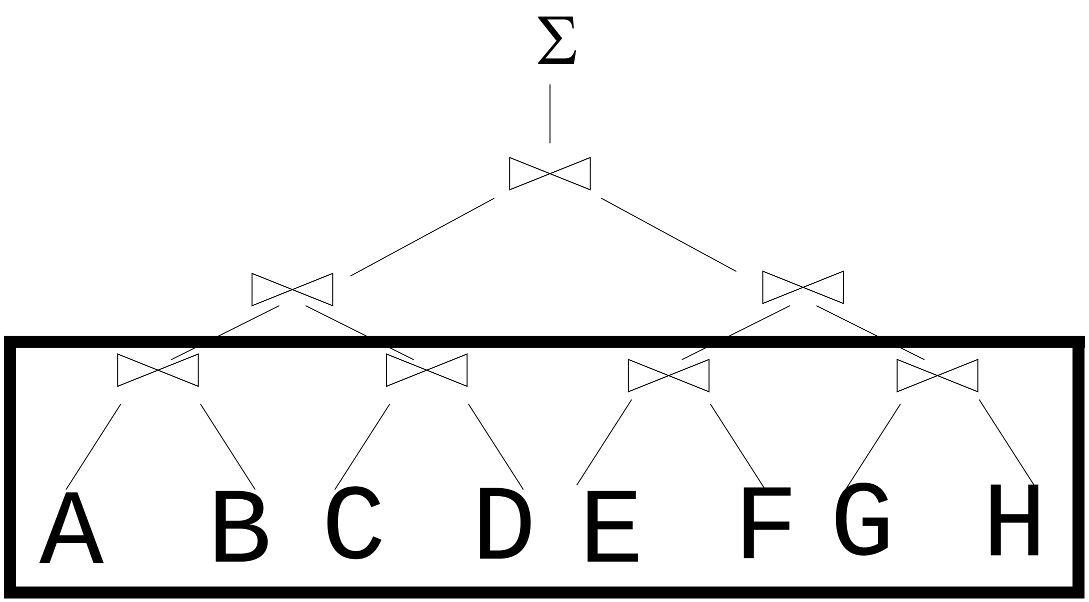
Basic Query Processing in DBO

Imagine a complex query plan:



Basic Query Processing in DBO

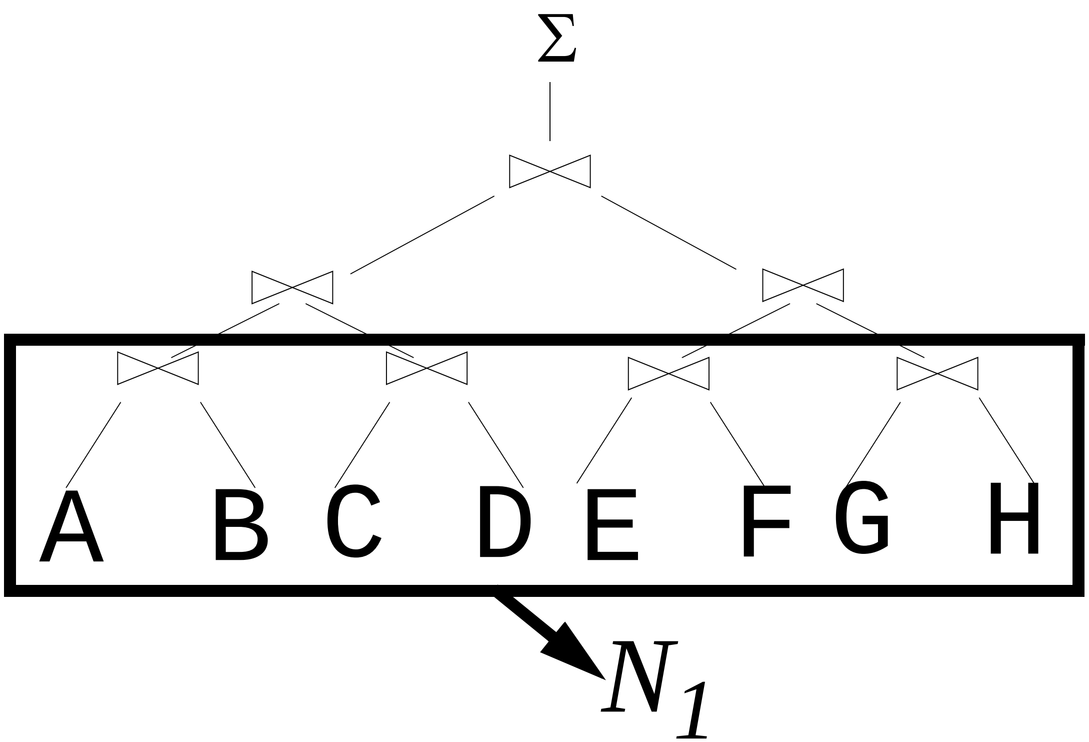
To process the plan:



First, all bottom-level joins are processed concurrently, kind of like one huge SMS sort phase; this is called a ***levelwise step***

Basic Query Processing in DBO

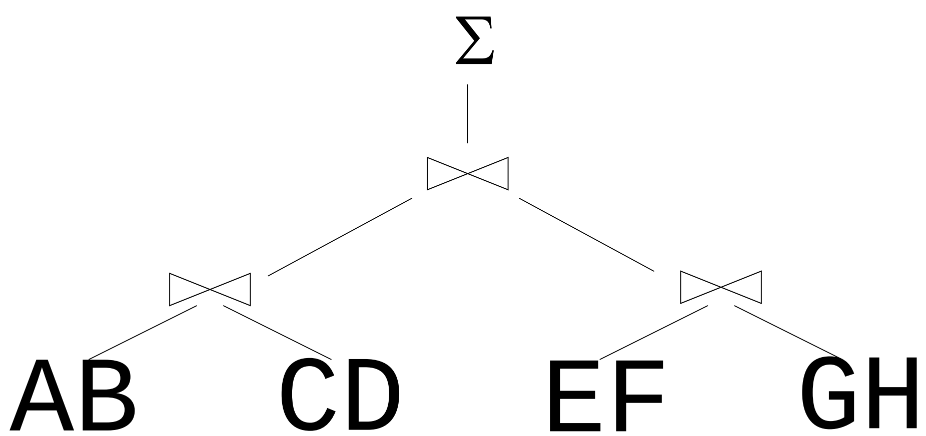
To process the plan:



The first levelwise step (at all times) checks for "lucky" answer tuples in order to maintain an online guess at the query result

Basic Query Processing in DBO

To process the plan:

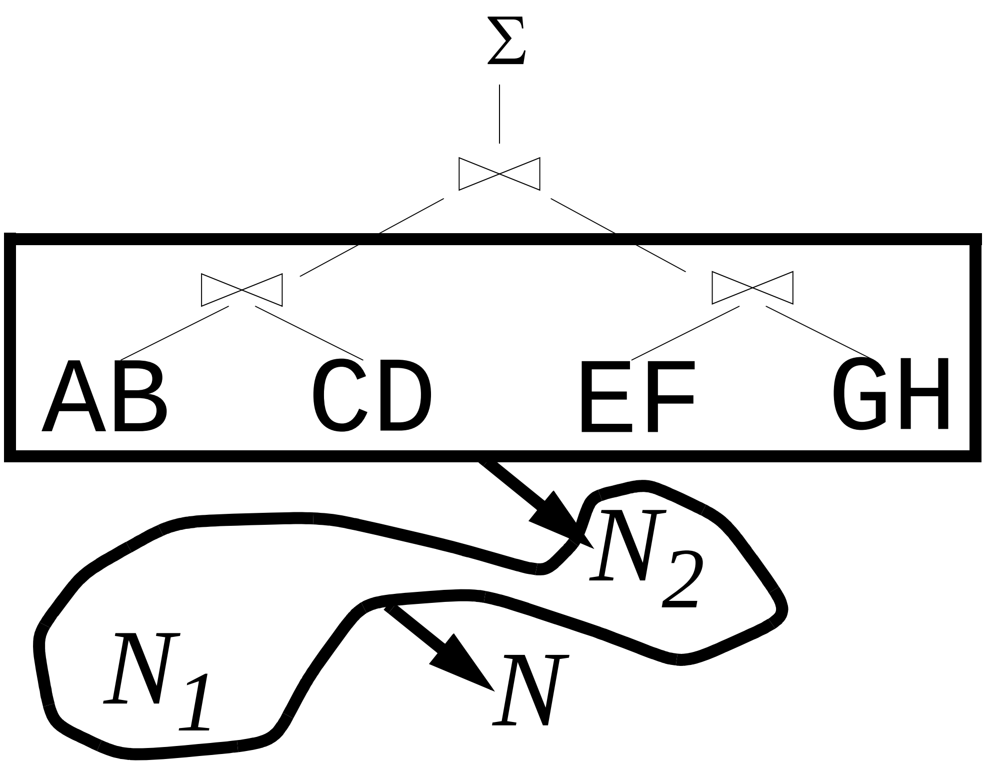


Eventually the first levelwise step completes all of the "lowest" joins, and N_1 is frozen

N_1

Basic Query Processing in DBO

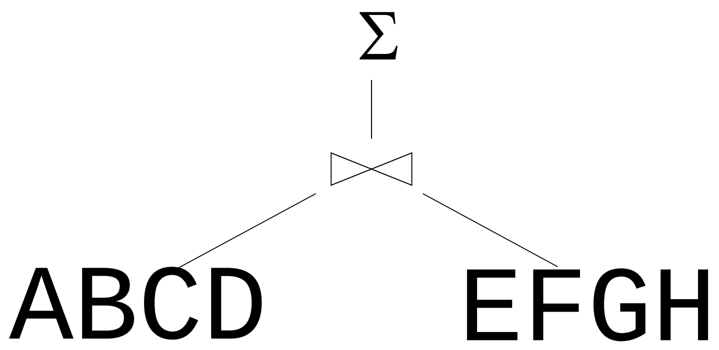
To process the plan:



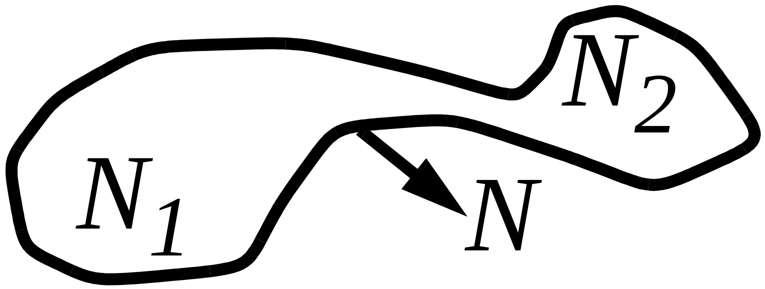
Then levelwise step two begins... N_2 will soon be far more accurate than N_1; they are combined to give the user an estimate

Basic Query Processing in DBO

To process the plan:

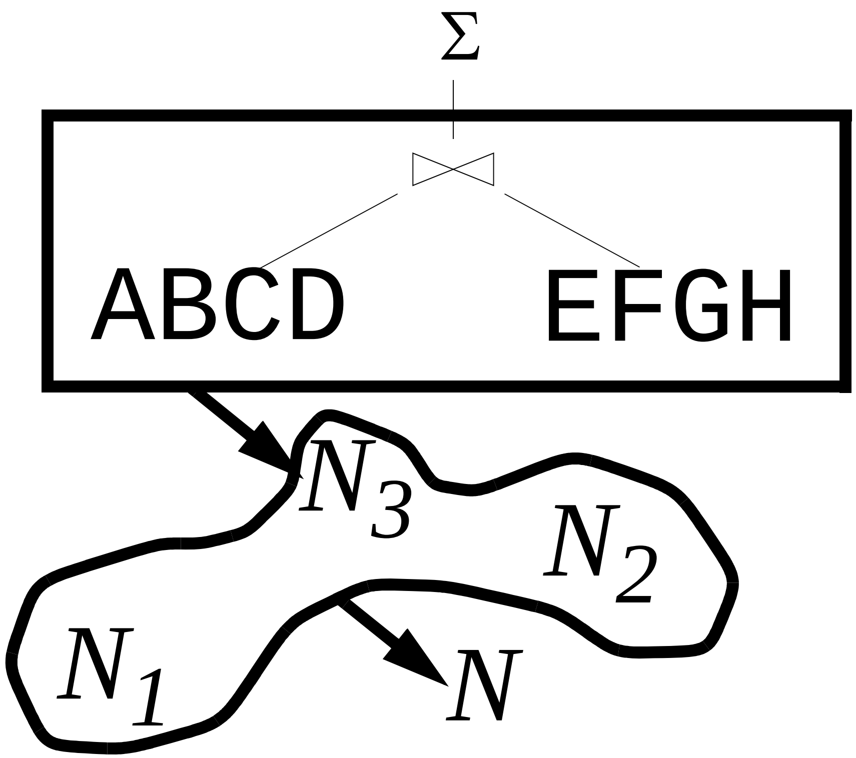


Eventually levelwise step two finishes and N_2 is frozen as well.



Basic Query Processing in DBO

To process the plan:



Then levelwise step three begins... as it proceeds, the variance of N_3 goes to zero

Basic Query Processing in DBO

To process the plan:

Σ

|

ABCDEFGH

When levelwise step three finishes, the query has completed execution.

Details, Details, Details...

Each levelwise step has two phases: the *scan phase* and the *merge phase*

The scan phase is analogous to the sort phase of a SMJ, except:

1. There is one scan phase for all joins at a levelwise step
2. Any "answer tuples" discovered are used to update N_i
3. Round-robin processing of runs
4. Makes use of a randomized sort order (provided by H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

WHERE $R_1.B = R_2.C$

AND $R_2.E = R_3.F$

AND $R_3.G = R_4.H$

R_1

A	B	
8	7	4
1	0	5
9	7	4
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
7	1	5	4
0	5	4	5
4	3	1	0
6	4	9	3
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Read one run from *each relation* in the level-wise step into memory

R_1

A	B	
8	7	4
1	0	5
9	7	4
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
7	1	5	4
0	5	4	5
4	3	1	0
6	4	9	3
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

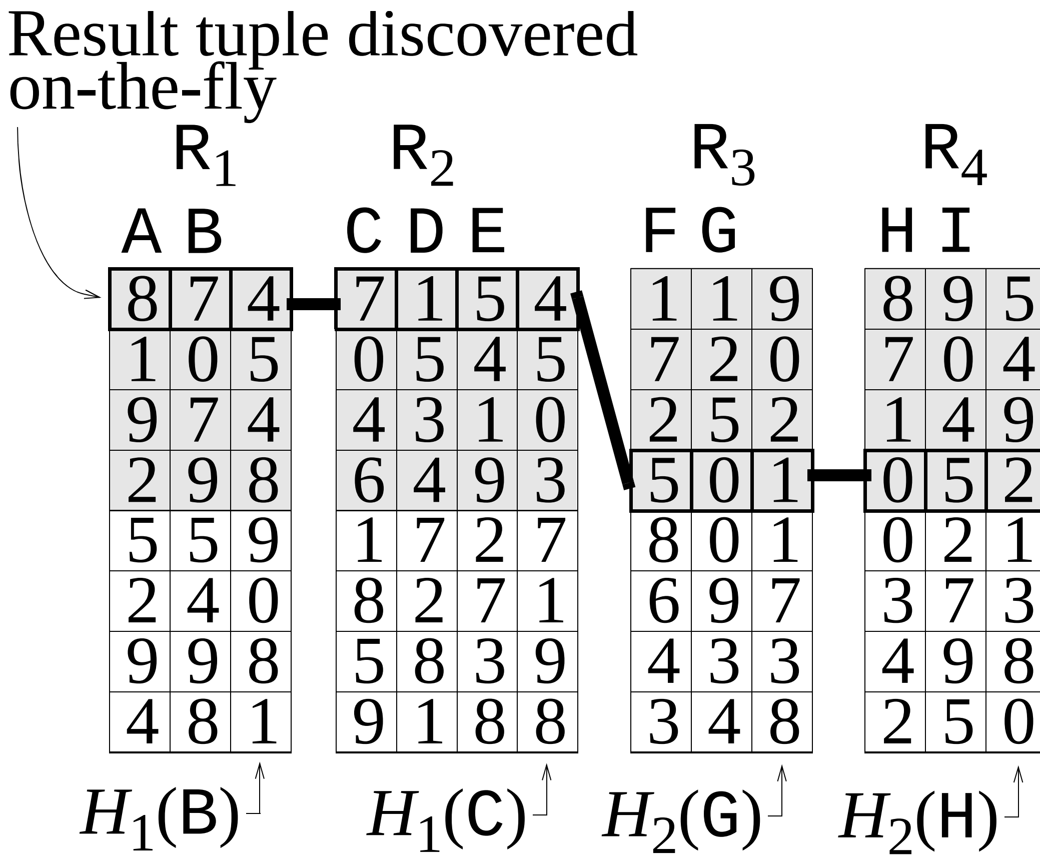
H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Any output tuples are immediately discovered; used to produce N_i (unbiased!)



Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Sort run from first relation on H_1(B); write back to disk

Sorted on H_1(B), written out to disk

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
7	1	5	4
0	5	4	5
4	3	1	0
6	4	9	3
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Load second run
from first relation

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
7	1	5	4
0	5	4	5
4	3	1	0
6	4	9	3
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Use any discovered result tuples to update estimate (still unbiased!)

R_1

A B

8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C D E

7	1	5	4
0	5	4	5
4	3	1	0
6	4	9	3
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F G

1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H I

8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Sort run on H_1(C) and write it back to disk

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Load next run from second relation into memory

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
1	1	9
7	2	0
2	5	2
5	0	1
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

No output tuples discovered, so update estimate and write next run back to disk

R_1

A B

8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C D E

4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F G

7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H I

8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Load next run
into memory

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Immediately discover any output tuples and update the estimate

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

$H_1(B)$

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

$H_1(C)$

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
6	9	7
4	3	3
3	4	8

$H_2(G)$

R_4

H	I	
8	9	5
7	0	4
1	4	9
0	5	2
0	2	1
3	7	3
4	9	8
2	5	0

$H_2(H)$

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Sort and write back to disk

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
5	5	9
2	4	0
9	9	8
4	8	1

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
1	7	2	7
8	2	7	1
5	8	3	9
9	1	8	8

H_1(C)

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
6	9	7
4	3	3
3	4	8

H_2(G)

R_4

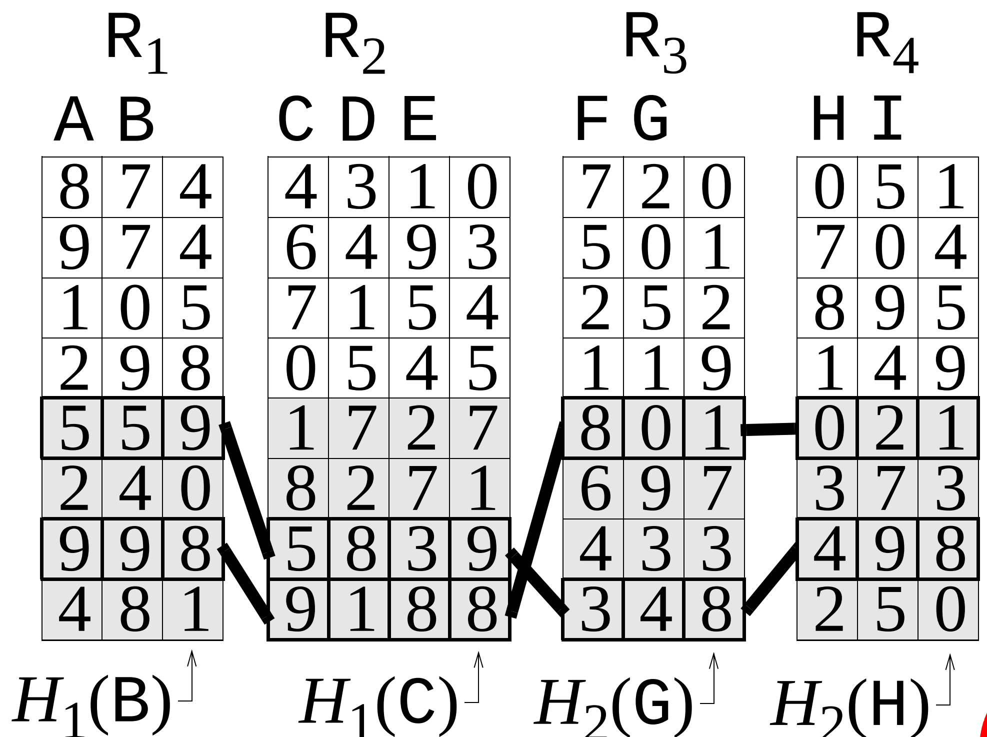
H	I	
0	5	1
7	0	4
8	9	5
1	4	9
0	2	1
3	7	3
4	9	8
2	5	0

H_2(H)

Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

Do the same for the fourth relation



Scan Phase in Detail

To concurrent join (R_1 and R_2), (R_3 and R_4)

All tuples have been read, so write back all in-memory tuples, and we're done!

R_1

	A	B	
Run 1	8	7	4
	9	7	4
	1	0	5
	2	9	8
Run 2	2	4	0
	4	8	1
	9	9	8
	5	5	9

H_1(B)

R_2

	C	D	E	
Run 1	4	3	1	0
	6	4	9	3
	7	1	5	4
	0	5	4	5
Run 2	8	2	7	1
	1	7	2	7
	9	1	8	8
	5	8	3	9

H_1(C)

R_3

	F	G	
Run 1	7	2	0
	5	0	1
	2	5	2
	1	1	9
Run 2	8	0	1
	4	3	3
	6	9	7
	3	4	8

H_2(G)

R_4

	H	I	
Run 1	0	5	1
	7	0	4
	8	9	5
	1	4	9
Run 2	2	5	0
	0	2	1
	3	7	3
	4	9	8

H_2(H)

Key Points

- (In this example) Five different updates to N_i
- One update every time a run is processed
- N_i is unbiased assuming random input order
- Can characterize variance (very challenging!)
- Ready for merge phase...

Merge Phase

- Separate merge for each join in *i*th levelwise step
- Each merge a lot like merge phase of SMJ
- Recall that we use a random sort order
- So result tuples come out in (semi-) random order
- Output tuples pipelined directly into scan phase of *next* level-wise step
- Since tuples produced randomly, (i + 1)th levelwise step valid

Merge Phase in Detail

R_1: A B

Run			
Run 1	8	7	4
Run 1	9	7	4
Run 1	1	0	5
Run 1	2	9	8
Run 2	2	4	0
Run 2	4	8	1
Run 2	9	9	8
Run 2	5	5	9

H_1(B)

R_2: C D E

Run				
Run 1	4	3	1	0
Run 1	6	4	9	3
Run 1	7	1	5	4
Run 1	0	5	4	5
Run 2	8	2	7	1
Run 2	1	7	2	7
Run 2	9	1	8	8
Run 2	5	8	3	9

H_1(C)

R_3: F G

Run			
Run 1	7	2	0
Run 1	5	0	1
Run 1	2	5	2
Run 1	1	1	9
Run 2	8	0	1
Run 2	4	3	3
Run 2	6	9	7
Run 2	3	4	8

H_2(G)

R_4: H I

Run			
Run 1	0	5	1
Run 1	7	0	4
Run 1	8	9	5
Run 1	1	4	9
Run 2	2	5	0
Run 2	0	2	1
Run 2	3	7	3
Run 2	4	9	8

H_2(H)

Output so far:

R_{12} R_{34}

Merge Phase in Detail

Head of each run of each relation is read into memory...

R_1

Run	A	B	
Run 1	8	7	4
	9	7	4
	1	0	5
	2	9	8
Run 2	2	4	0
	4	8	1
	9	9	8
	5	5	9

H_1(B) ↑

R_2

Run	C	D	E	
Run 1	4	3	1	0
	6	4	9	3
	7	1	5	4
	0	5	4	5
Run 2	8	2	7	1
	1	7	2	7
	9	1	8	8
	5	8	3	9

H_1(C) ↑

R_3

Run	F	G	
Run 1	7	2	0
	5	0	1
	2	5	2
	1	1	9
Run 2	8	0	1
	4	3	3
	6	9	7
	3	4	8

H_2(G) ↑

R_4

Run	H	I	
Run 1	0	5	1
	7	0	4
	8	9	5
	1	4	9
Run 2	2	5	0
	0	2	1
	3	7	3
	4	9	8

H_2(H) ↑

Output so far:

R_{12} R_{34}

Merge Phase in Detail

Search for output tuples in first join...

R_1

A B

8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

H_1(B)

R_2

C D E

4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

H_1(C)

R_3

F G

7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

H_2(G)

R_4

H I

0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

H_2(H)

Run 1

Run 2

Joined through key 1

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7

R_{34}

pipelined directly into next levelwise step

Merge Phase in Detail

Search for output tuples in second join...

R_1

Run	A	B	
Run 1	8	7	4
	9	7	4
	1	0	5
	2	9	8
Run 2	2	4	0
	4	8	1
	9	9	8
	5	5	9

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

H_1(C)

Joined through key 1

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

H_2(G)

R_4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

H_2(H)

Joined through key 1

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2

Merge Phase in Detail

Exhausted run-heads are replaced in memory...

Run	R_1: A	B		R_2: C	D	E		R_3: F	G		R_4: H	I	
Run 1	8	7	4	4	3	1	0	7	2	0	0	5	1
	9	7	4	6	4	9	3	5	0	1	7	0	4
	1	0	5	7	1	5	4	2	5	2	8	9	5
	2	9	8	0	5	4	5	1	1	9	1	4	9
Run 2	2	4	0	8	2	7	1	8	0	1	2	5	0
	4	8	1	1	7	2	7	4	3	3	0	2	1
	9	9	8	9	1	8	8	6	9	7	3	7	3
	5	5	9	5	8	3	9	3	4	8	4	9	8

H_1(B) H_1(C) H_2(G) H_2(H)

Joined through key 1 Joined through key 1

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2

Merge Phase in Detail

New output tuples are written to output...

R_1

Run	A	B	
Run 1	8	7	4
	9	7	4
	1	0	5
	2	9	8
Run 2	2	4	0
	4	8	1
	9	9	8
	5	5	9

R_2

Run	C	D	E	
Run 1	4	3	1	0
	6	4	9	3
	7	1	5	4
	0	5	4	5
Run 2	8	2	7	1
	1	7	2	7
	9	1	8	8
	5	8	3	9

R_3

Run	F	G	
Run 1	7	2	0
	5	0	1
	2	5	2
	1	1	9
Run 2	8	0	1
	4	3	3
	6	9	7
	3	4	8

R_4

Run	H	I	
Run 1	0	5	1
	7	0	4
	8	9	5
	1	4	9
Run 2	2	5	0
	0	2	1
	3	7	3
	4	9	8

H_1(B) H_1(C) H_2(G) H_2(H)

Joined through key 3 Joined through key 3

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7

Merge Phase in Detail

Exhausted runs are replenished, new output tuples discovered...

R_1

Run	A	B	H_1(B)
Run 1	8	7	4
Run 1	9	7	4
Run 1	1	0	5
Run 1	2	9	8
Run 2	2	4	0
Run 2	4	8	1
Run 2	9	9	8
Run 2	5	5	9

R_2

Run	C	D	E	H_1(C)
Run 1	4	3	1	0
Run 1	6	4	9	3
Run 1	7	1	5	4
Run 1	0	5	4	5
Run 2	8	2	7	1
Run 2	1	7	2	7
Run 2	9	1	8	8
Run 2	5	8	3	9

Joined through key 4

R_3

Run	F	G	H_2(G)
Run 1	7	2	0
Run 1	5	0	1
Run 1	2	5	2
Run 1	1	1	9
Run 2	8	0	1
Run 2	4	3	3
Run 2	6	9	7
Run 2	3	4	8

R_4

Run	H	I	H_2(H)
Run 1	0	5	1
Run 1	7	0	4
Run 1	8	9	5
Run 1	1	4	9
Run 2	2	5	0
Run 2	0	2	1
Run 2	3	7	3
Run 2	4	9	8

Joined through key 3

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7

Merge Phase in Detail

Again replace processed tuples in memory (perhaps skipping some steps!)...

Run 1 / Run 2

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

H_1(B)

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

H_1(C)

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

H_2(G)

R_4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

H_2(H)

Joined through key 4

Joined through key 3

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7

Merge Phase in Detail

New output tuples from first relation are written to output...

Run 1 / Run 2 (runs separated after the 4th row)

R_1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

H_1(B) — Done!

R_2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

H_1(C)

R_3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

H_2(G)

R_4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

H_2(H)

Joined through key 3

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5
1	0	0	5	4
2	9	9	1	8
9	9	9	1	8
5	5	5	8	3

R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7

Merge Phase in Detail

Same for second relation, and we are done!

R_1

Run	A	B	H_1(B)
Run 1	8	7	4
Run 1	9	7	4
Run 1	1	0	5
Run 1	2	9	8
Run 2	2	4	0
Run 2	4	8	1
Run 2	9	9	8
Run 2	5	5	9

Done!

R_2

Run	C	D	E	H_1(C)
Run 1	4	3	1	0
Run 1	6	4	9	3
Run 1	7	1	5	4
Run 1	0	5	4	5
Run 2	8	2	7	1
Run 2	1	7	2	7
Run 2	9	1	8	8
Run 2	5	8	3	9

R_3

Run	F	G	H_2(G)
Run 1	7	2	0
Run 1	5	0	1
Run 1	2	5	2
Run 1	1	1	9
Run 2	8	0	1
Run 2	4	3	3
Run 2	6	9	7
Run 2	3	4	8

Done!

R_4

Run	H	I	H_2(H)
Run 1	0	5	1
Run 1	7	0	4
Run 1	8	9	5
Run 1	1	4	9
Run 2	2	5	0
Run 2	0	2	1
Run 2	3	7	3
Run 2	4	9	8

Output so far:

R_{12}

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5
1	0	0	5	4
2	9	9	1	8
9	9	9	1	8
5	5	5	8	3

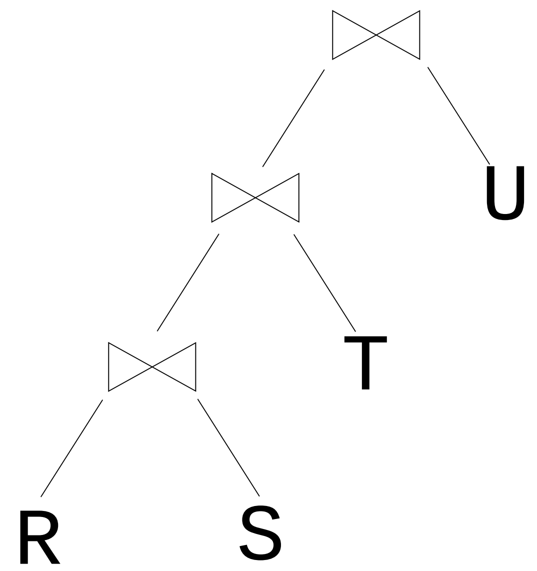
R_{34}

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7
3	4	4	9
1	1	1	4

A Few More Details...

Handling "inconvenient" queries

```
SELECT SUM (R.A)
FROM R, S, T, U
WHERE R.A = S.A AND
     R.B = T.B AND
     R.C = U.C
```

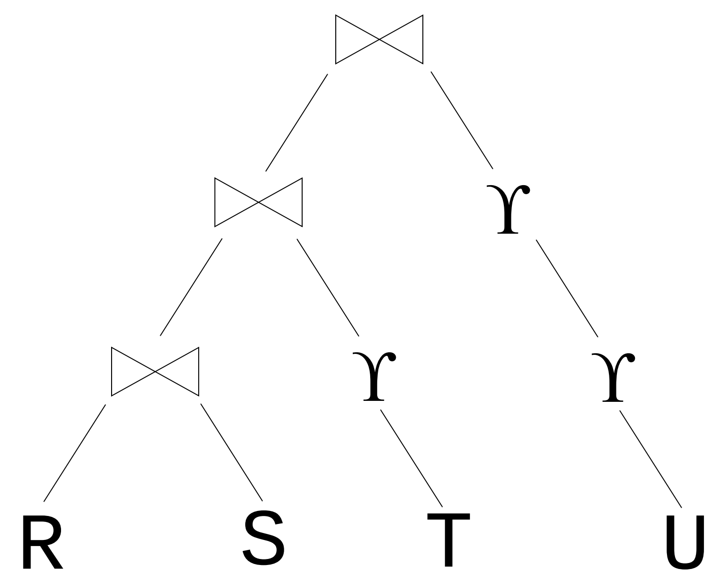


Issue: which relations are in the first levelwise step?

A Few More Details...

Handling "inconvenient" queries

```
SELECT SUM (R.A)
FROM R, S, T, U
WHERE R.A = S.A AND
     R.B = T.B AND
     R.C = U.C
```



Solution: introduce a "scan and reorder" operator

A Few More Details...

Computing the current estimate

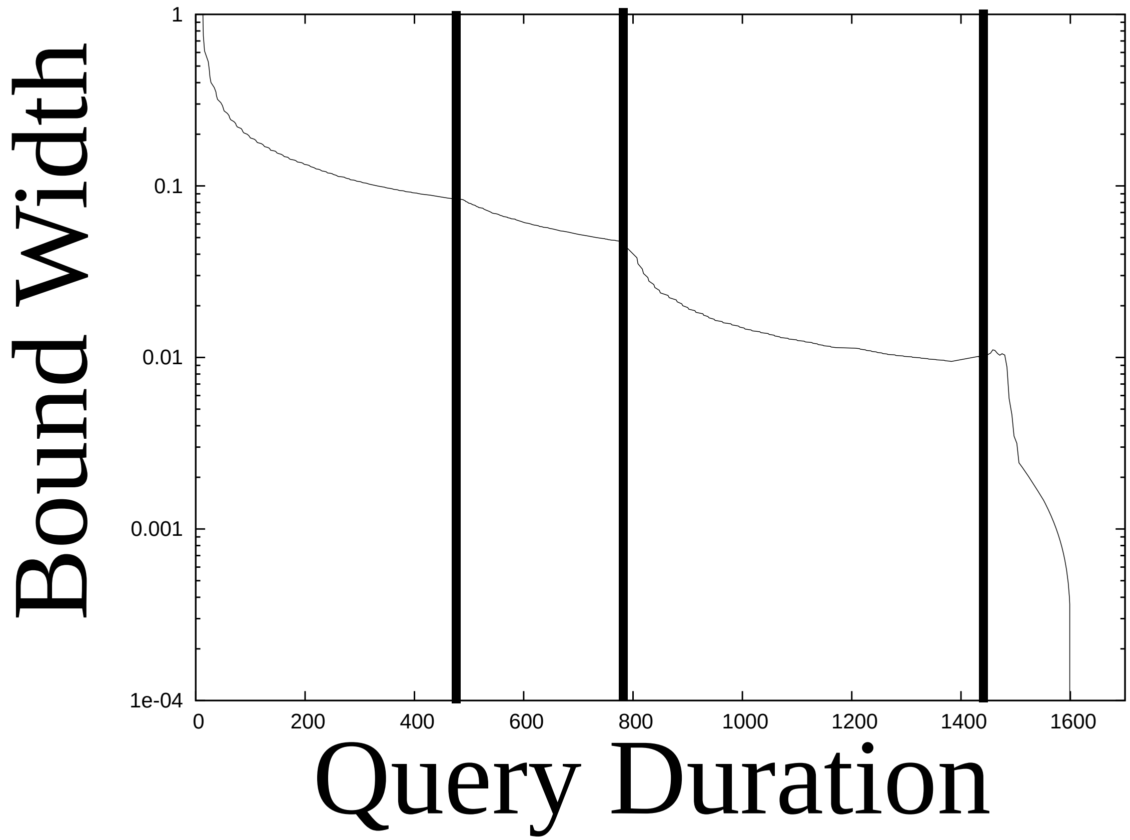
- Use $N = \sum w_i N_i$; since independent, getting w_i is an easy optimization problem

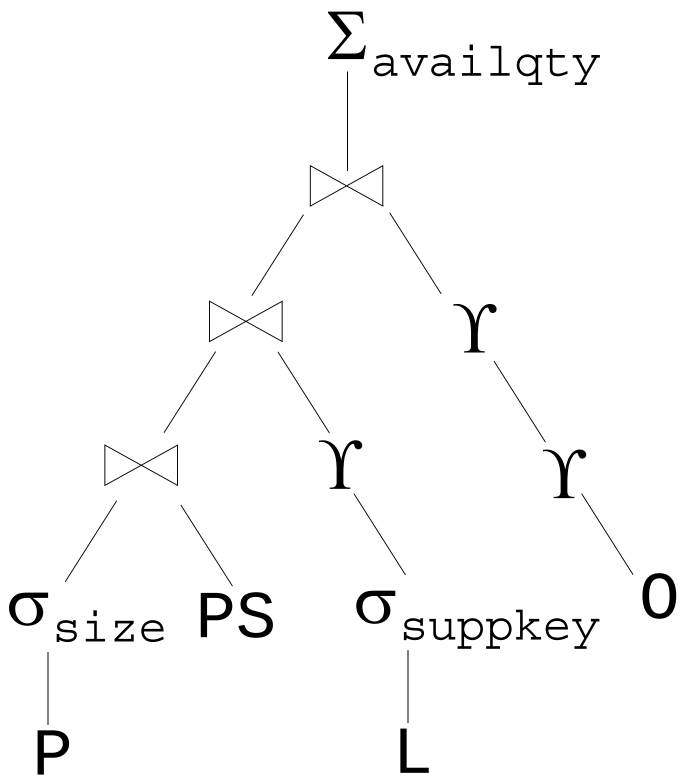
Computing accuracy guarantees

- Not easy!
    1. Two kinds of randomization
    2. “Chunks” of tuples from merge phase
    3. LOTS of algebra, recursive variance formulas
    4. Covariance among estimates making up each N_i

How Well Does This Work?

Ex: TPC-H query: DBO 26m42s
Postgres 43m47s





Are We Done Yet?

Of Course Not!

- Subtraction and related operators are a big problem
- Indexing (how to provide randomness?)
- Query optimization: what is the goal?
- Many more...

Thank You!

- Special thanks to my UF colleague Alin Dobra, and my students Abhijit Pol, Subramanian Arumugam, Shantanu Joshi
- Some papers on this stuff:
  1. "A Disk-Based Join with Probabilistic Guarantees," Jermaine, Dobra, Arumugam, Joshi, Pol, SIGMOD 2005
  2. "The Sort-Merge-Shrink Join," Jermaine, Dobra, Arumugam, Joshi, Pol, TODS 31(4), December 2006