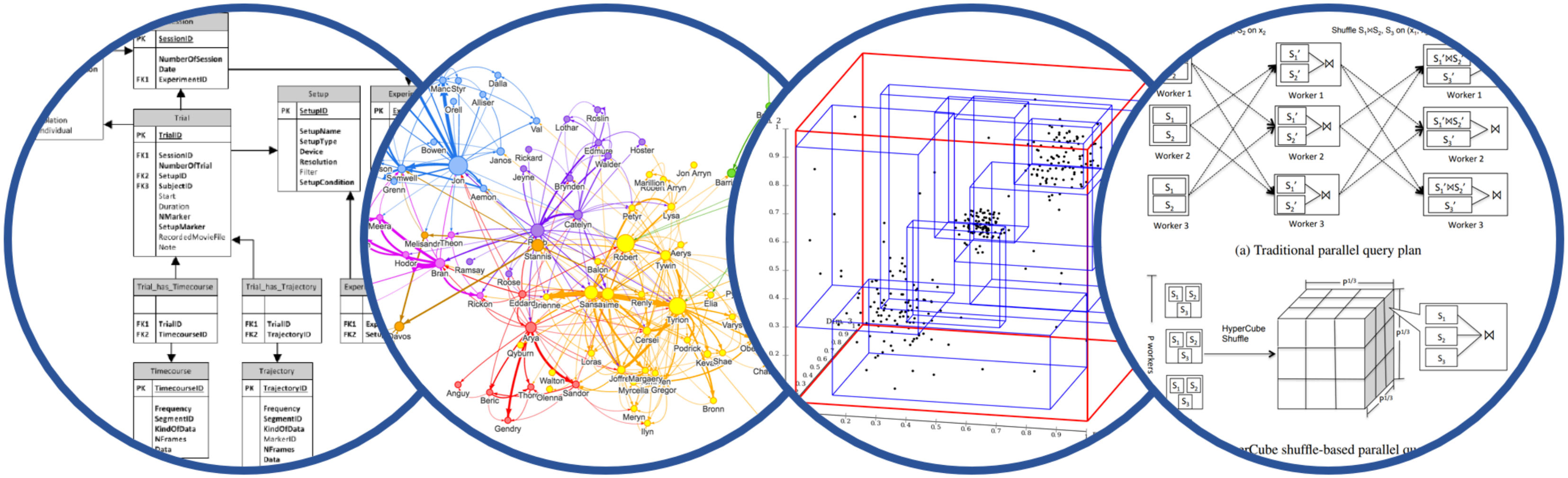


Database System Internals

Join Algorithms (cont.)

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Summary of External Join Algorithms

- Block Nested Loop: B(S) + B(R)*B(S)/(M-1)
- Index Join: B(R) + T(R)B(S)/V(S,a)
  (unclustered)
- Merge Join: 3B(R)+3B(S)
  - $B(R)+B(S) <= M^2$
- Partitioned Hash Join: (coming up next)

Partitioned Hash Algorithms

- Partition R it into k buckets:
  $R_1, R_2, R_3, \ldots, R_k$

Partitioned Hash Algorithms

- Partition R it into k buckets:
  $R_1, R_2, R_3, \ldots, R_k$

- Assuming $B(R_1)=B(R_2)=\ldots= B(R_k)$, we have
  $B(R_i) = B(R)/k$, for all i

Partitioned Hash Algorithms

- Partition R it into k buckets:
  $R_1, R_2, R_3, \ldots, R_k$
- Assuming $B(R_1)=B(R_2)=\ldots= B(R_k)$, we have
  $B(R_i) = B(R)/k$, for all i
- Goal: each R_i should fit in main memory:
  $B(R_i) \leq M$

Partitioned Hash Algorithms

- Partition R it into k buckets:
  $R_1, R_2, R_3, \ldots, R_k$

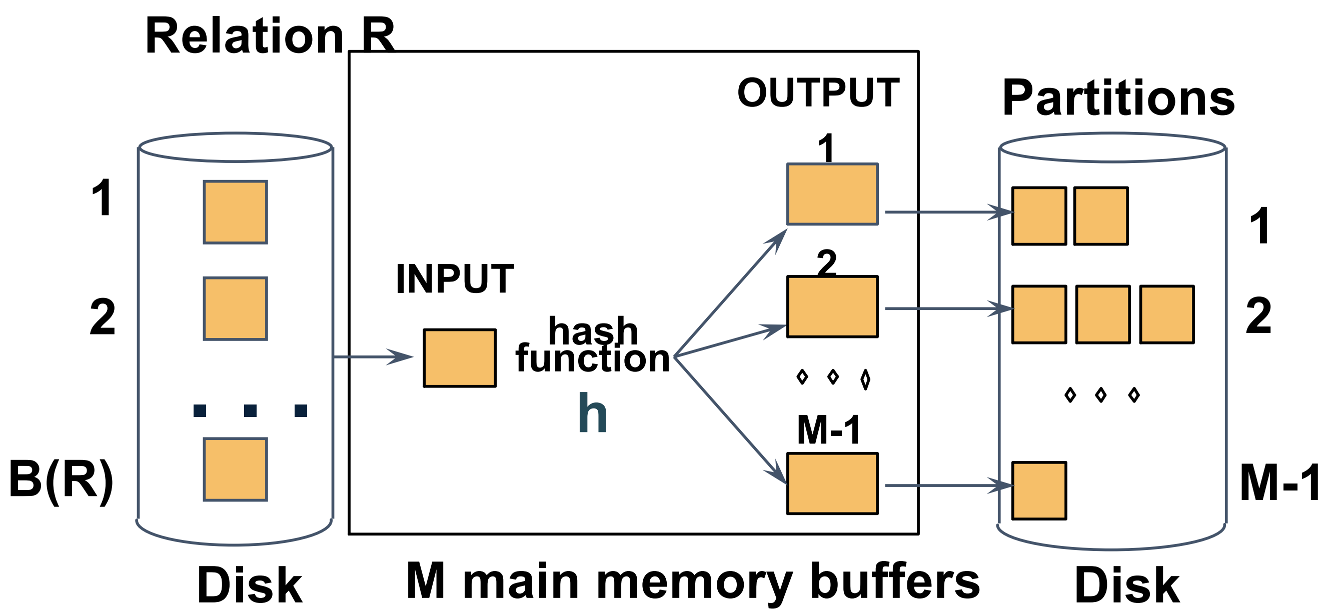
- Assuming $B(R_1)=B(R_2)=\ldots= B(R_k)$, we have
  $B(R_i) = B(R)/k$, for all i

- Goal: each R_i should fit in main memory:
  $B(R_i) \leq M$

How do we choose k?

Partitioned Hash Algorithms

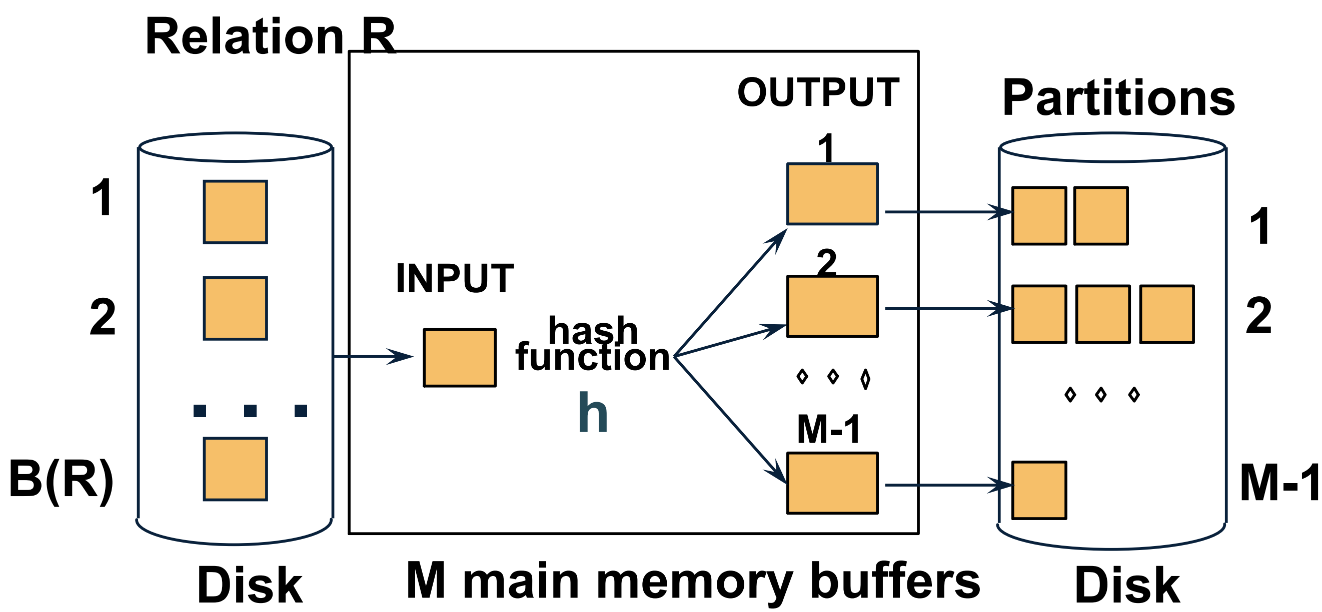
- We choose k = M-1 Each bucket has size approx. $B(R)/(M-1) \approx B(R)/M$



Assumption: $B(R)/M \leq M$, i.e. $B(R) \leq M^2$

Partitioned Hash Algorithms

- We choose k = M-1 Each bucket has size approx. B(R)/(M-1) ≈ B(R)/M



Assumption: $B(R)/M \leq M$, i.e. $B(R) \leq M^2$

Partitioned Hash Join (Grace-Join)

R ⋈ S

Note: partitioned hash-join is sometimes called <u>*grace-join*</u>

Partitioned Hash Join (Grace-Join)

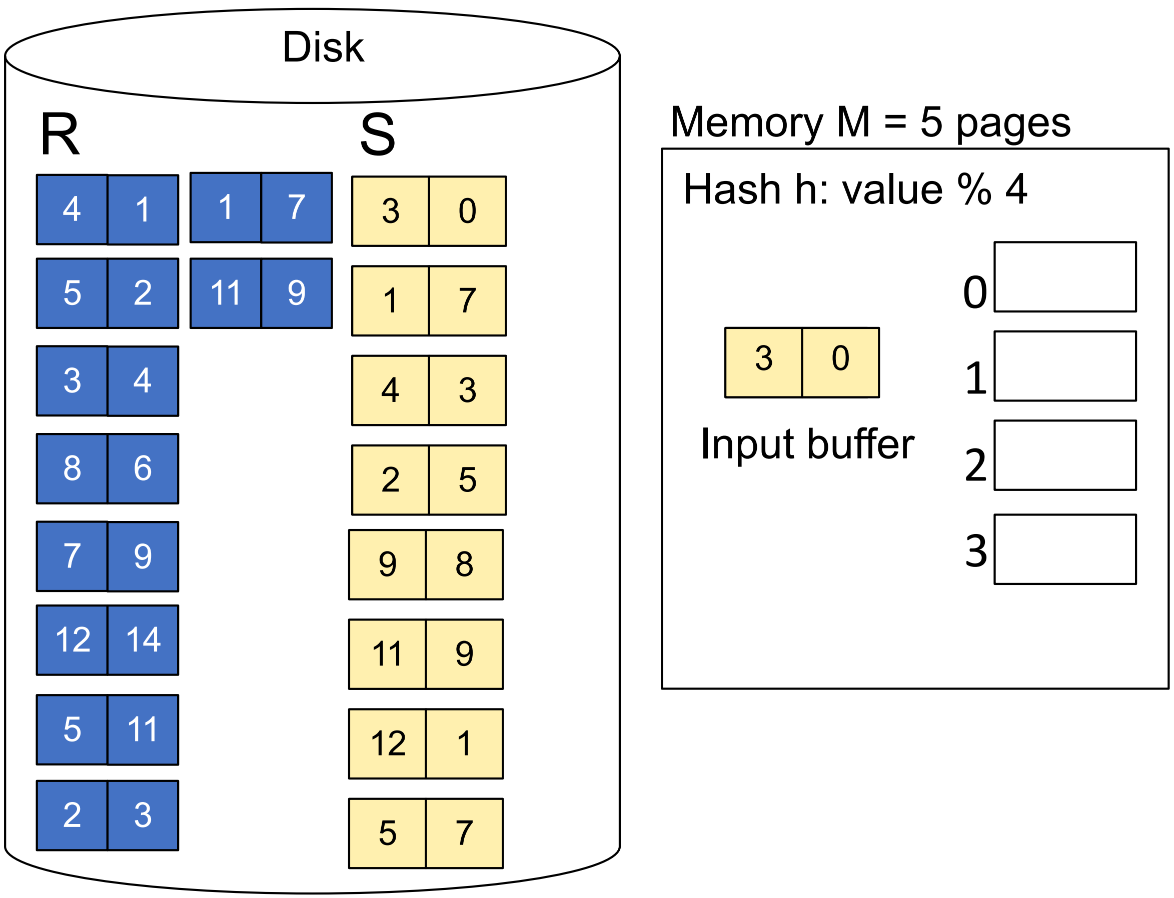
$R \bowtie S$

- Step 1:
  - Hash S into M-1 buckets
  - Send all buckets to disk
- Step 2
  - Hash R into M-1 buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Note: grace-join is also called *partitioned hash-join*

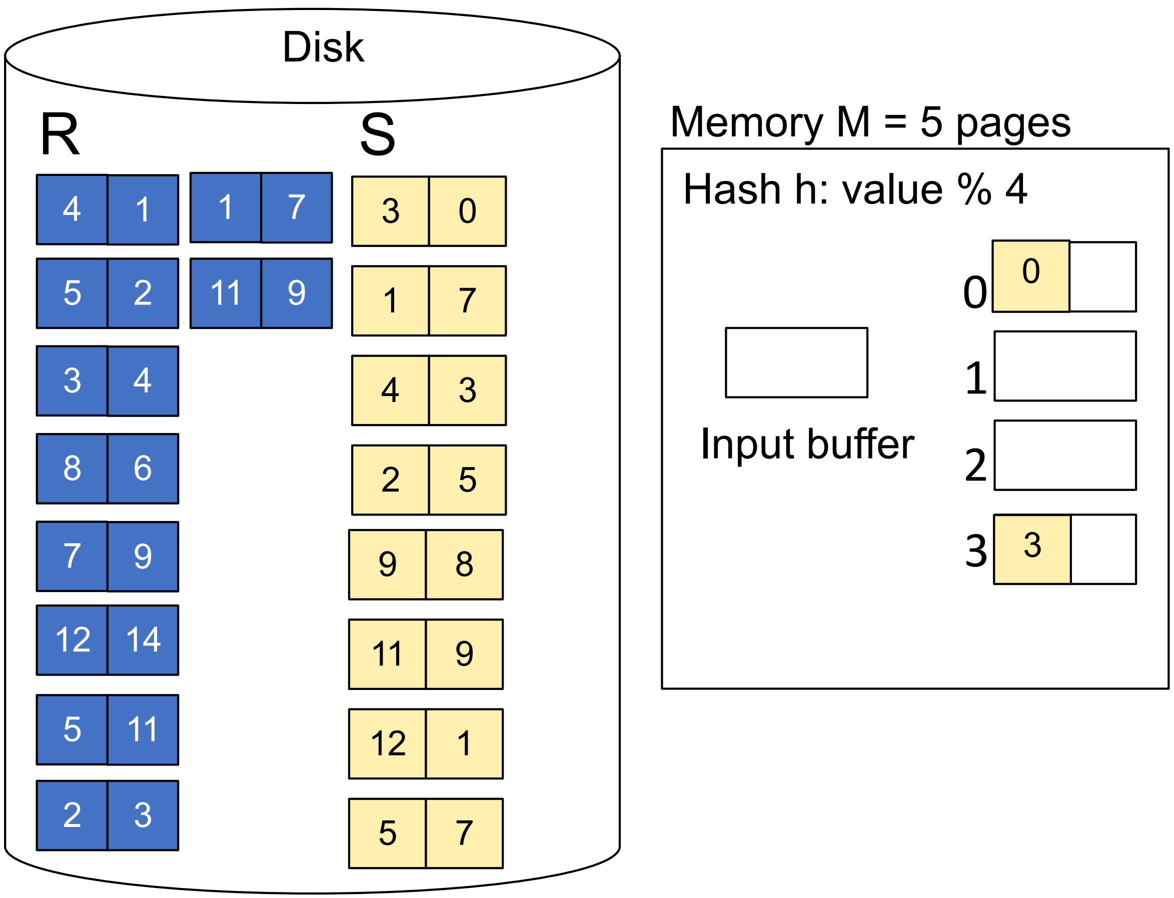
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into M-1 (=4 buckets)



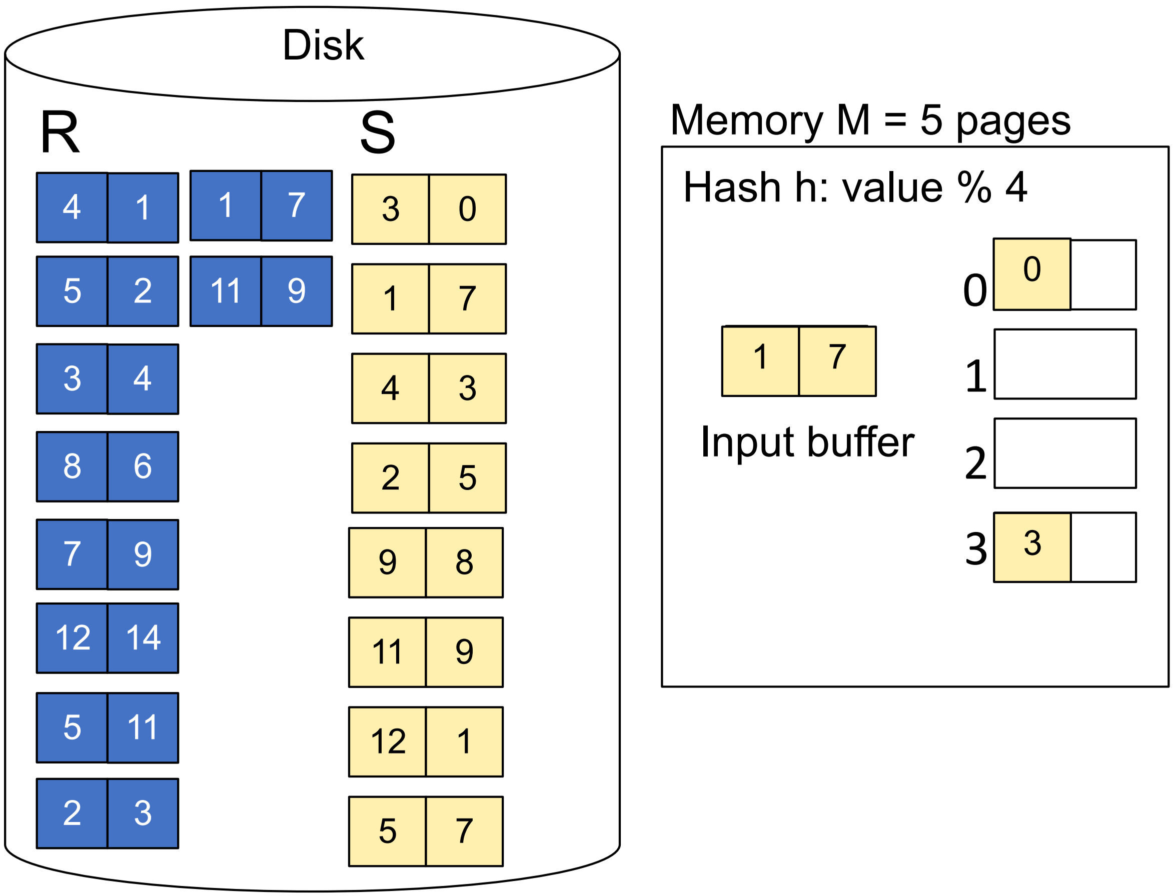
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets



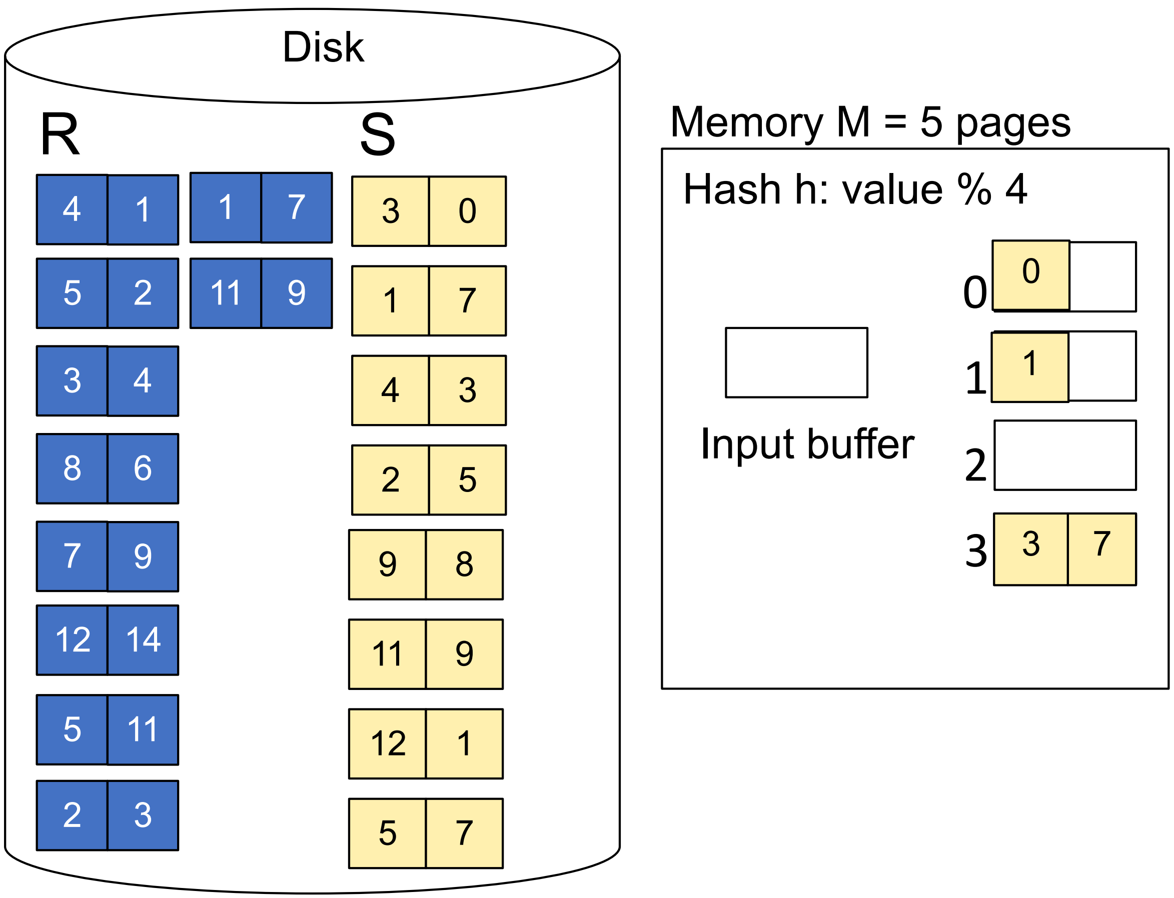
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets



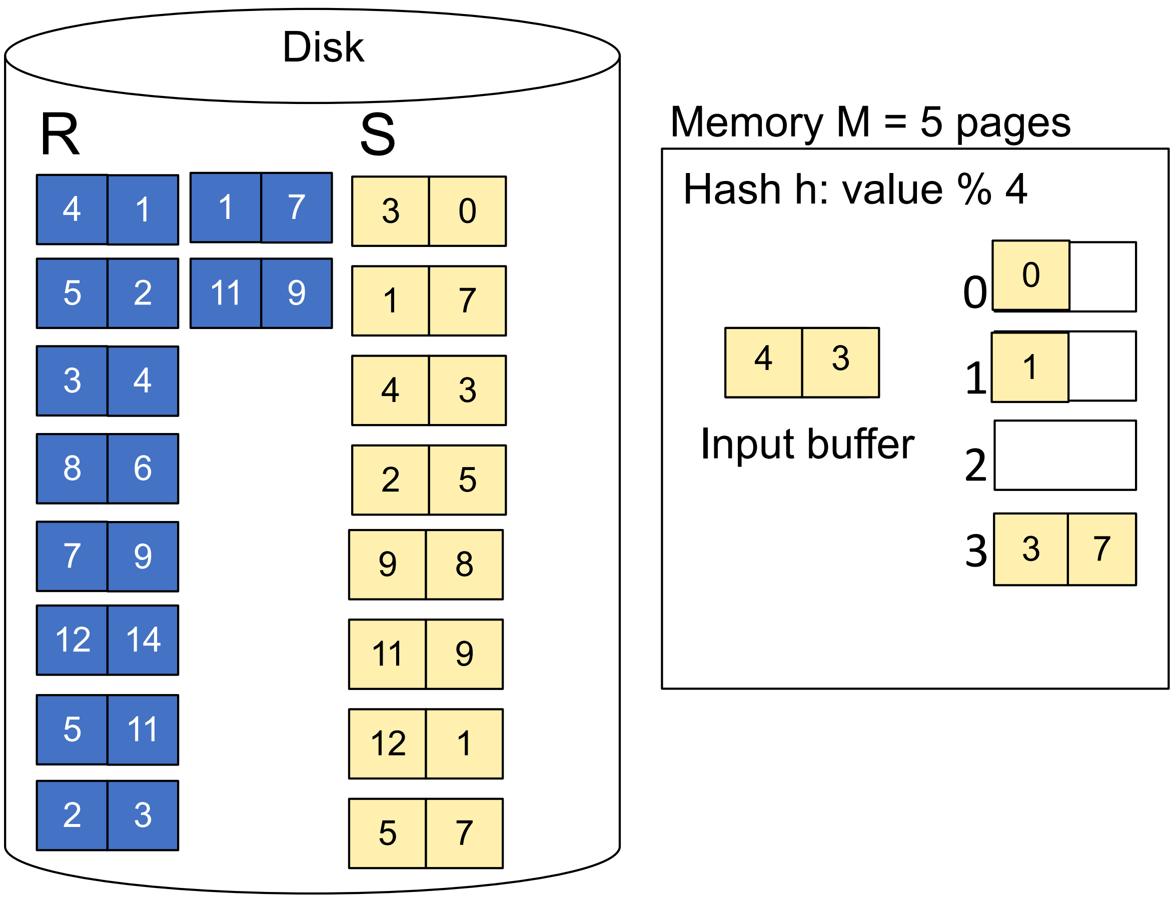
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets



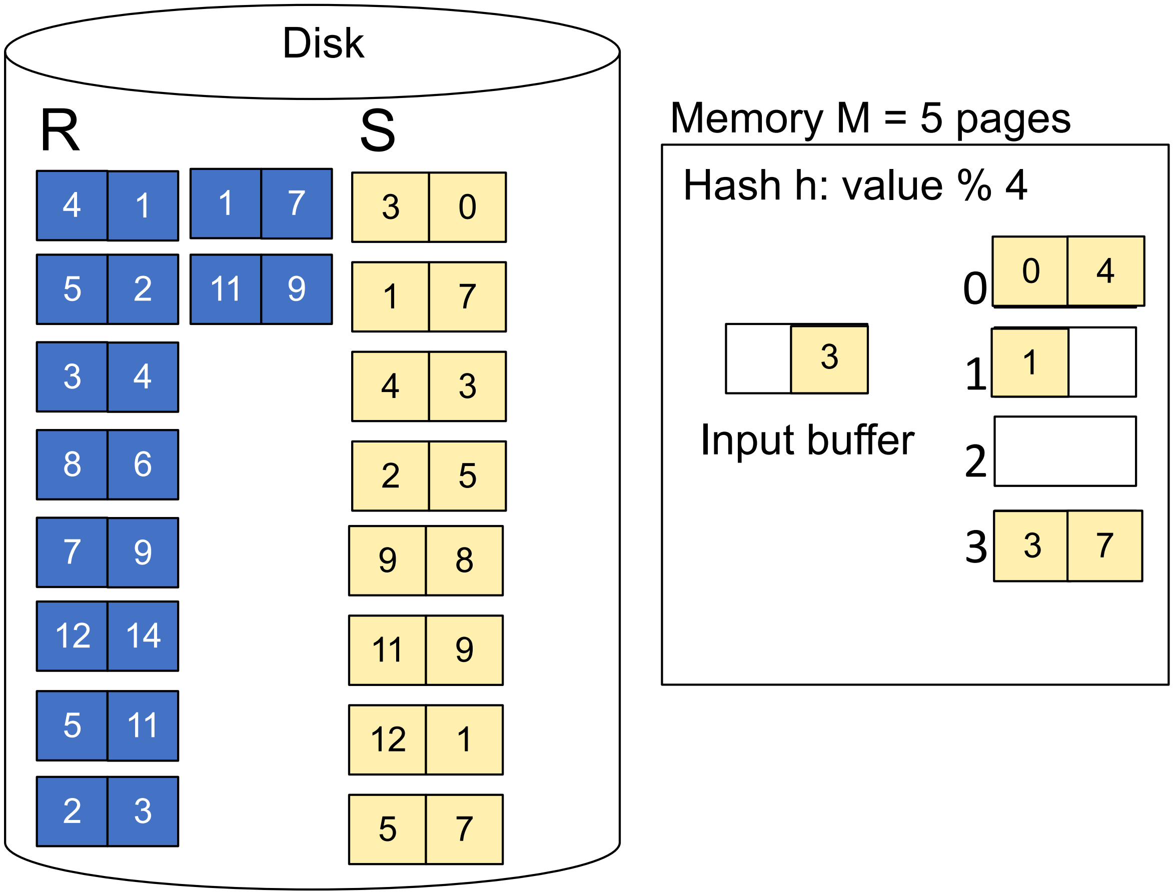
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets



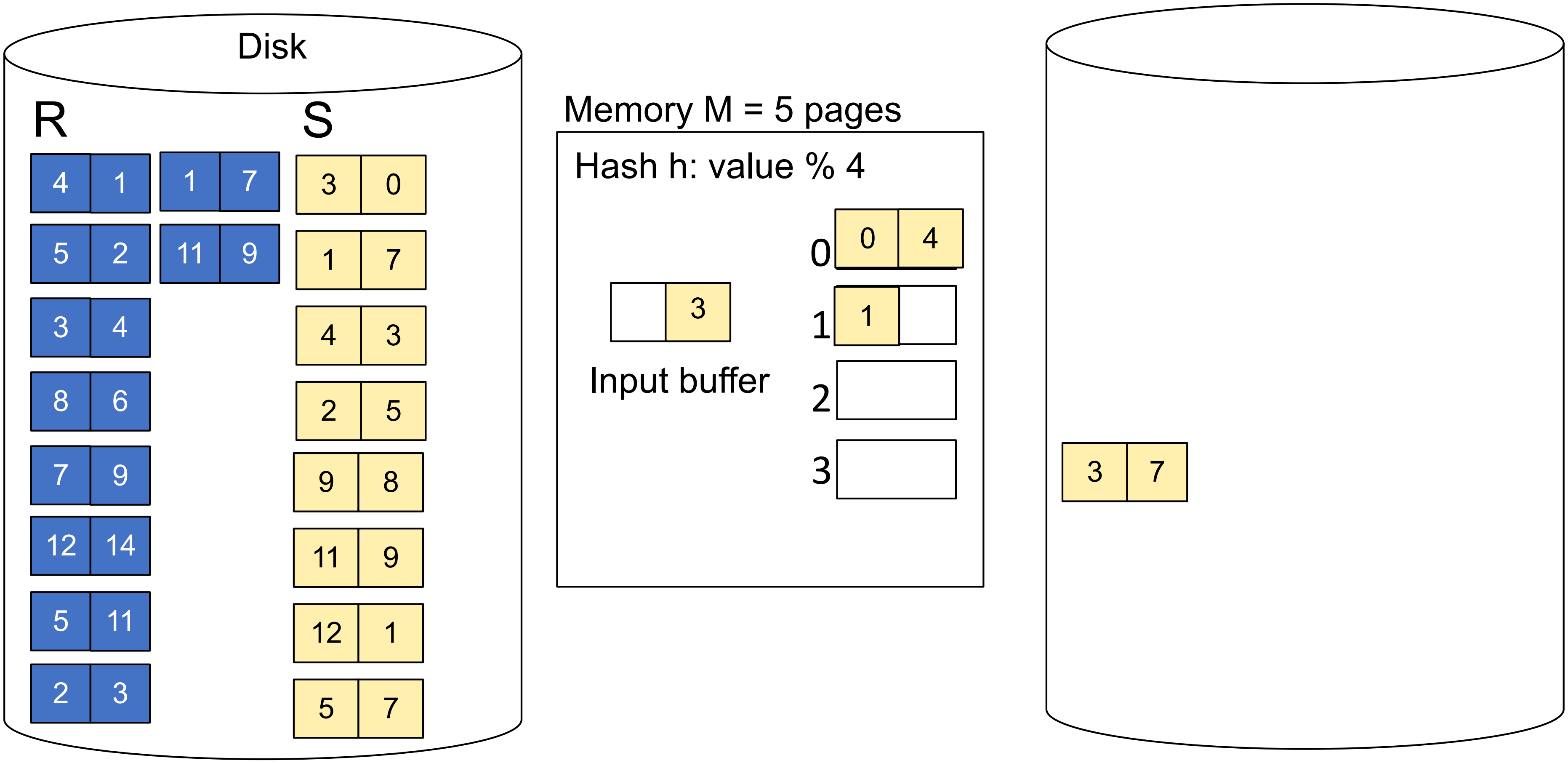
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets
When a bucket fills up, flush it to disk



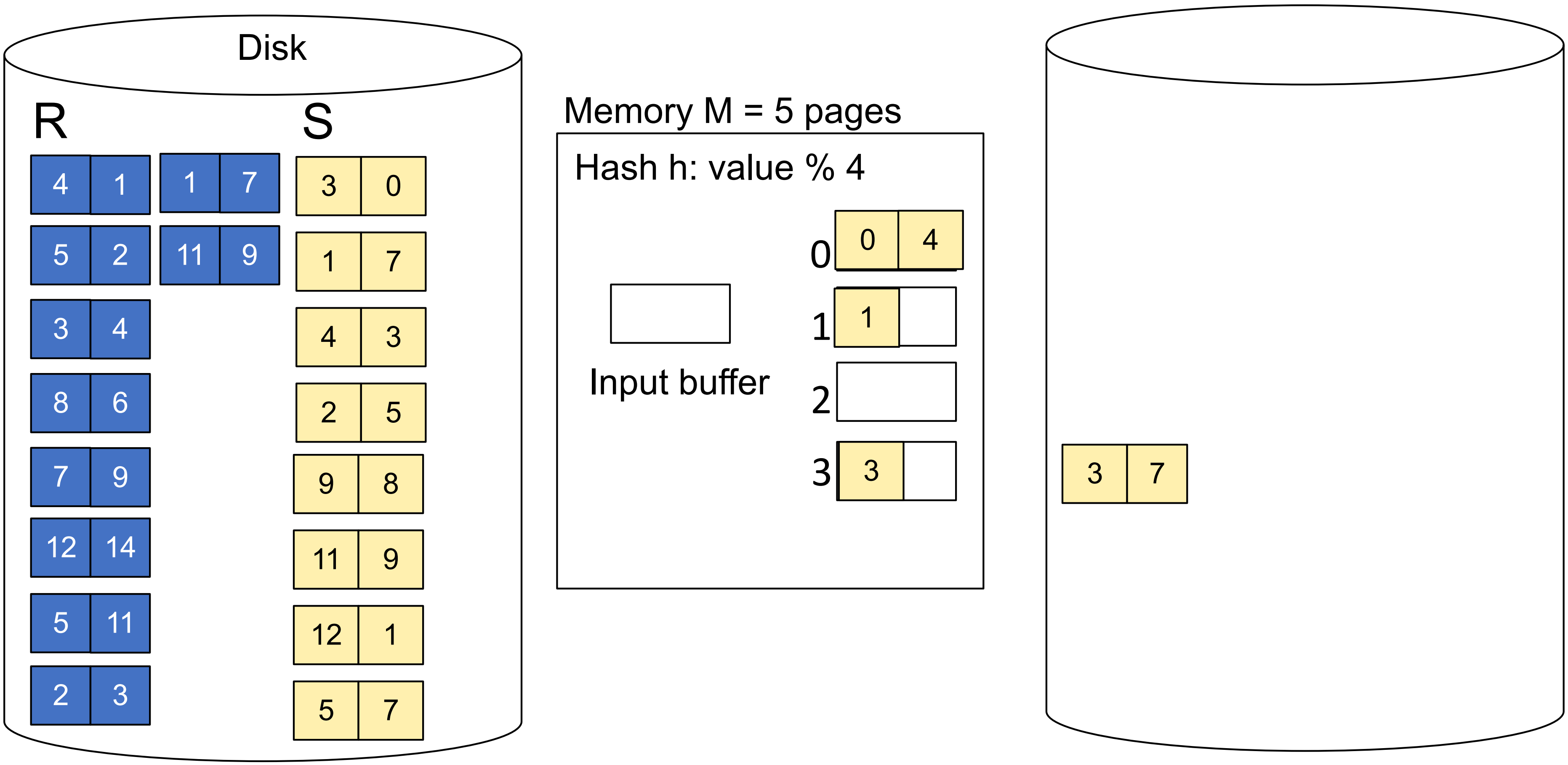
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets
When a bucket fills up, flush it to disk



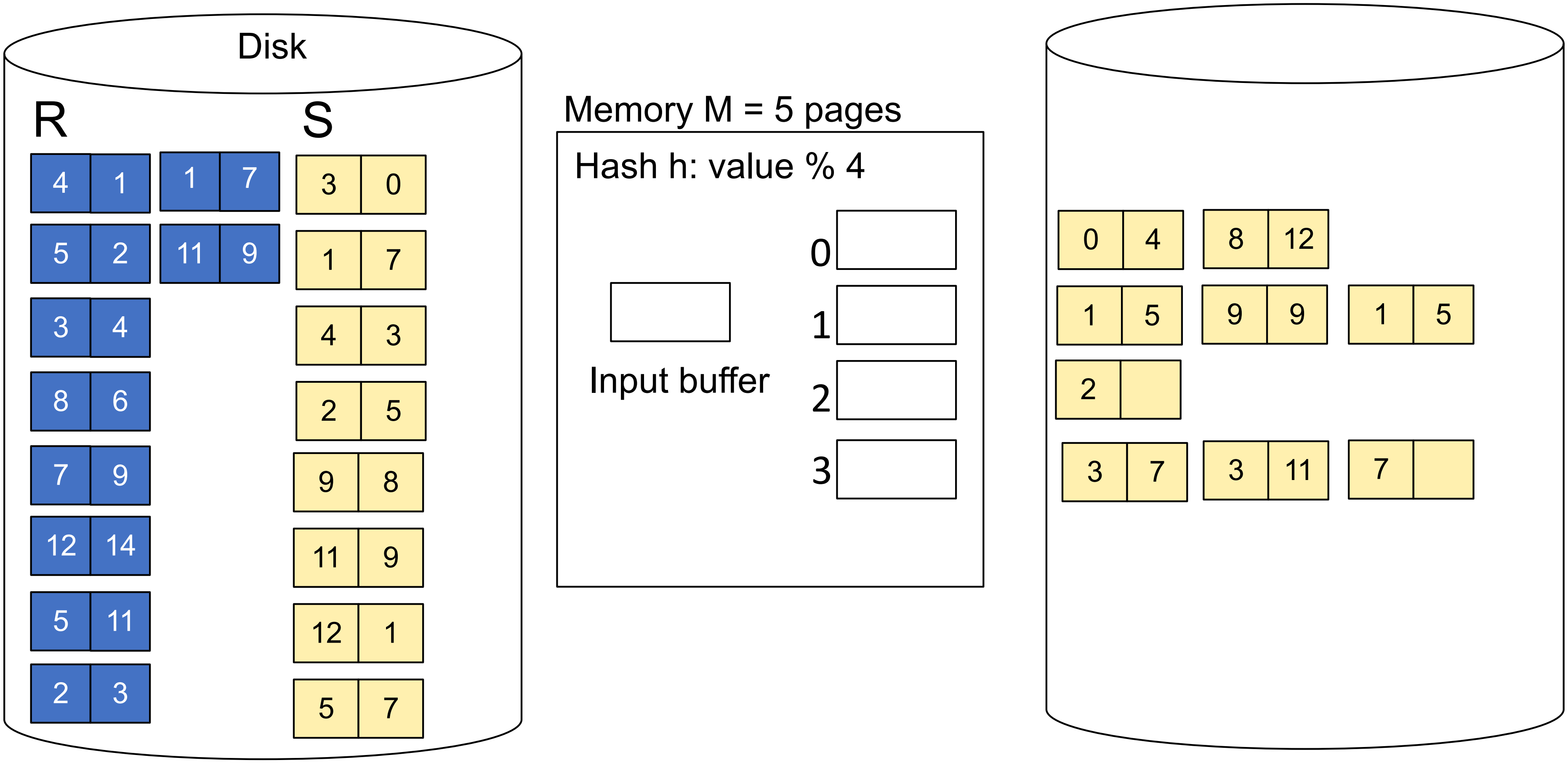
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets
When a bucket fills up, flush it to disk



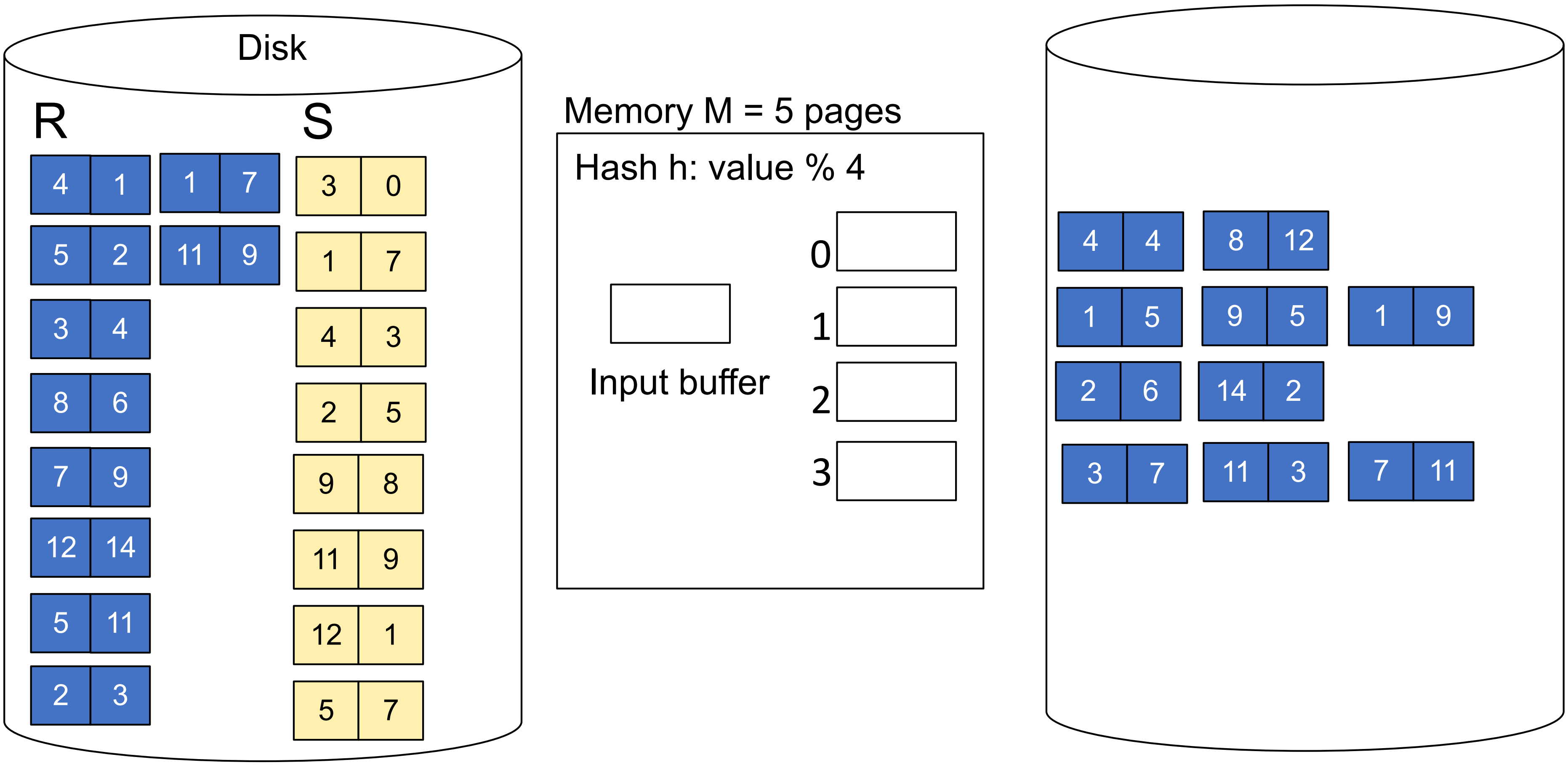
Partitioned Hash-Join Example

Step 1: Read relation S one page at a time and hash into the 4 buckets
At the end, we get relation S back on disk split into 4 buckets



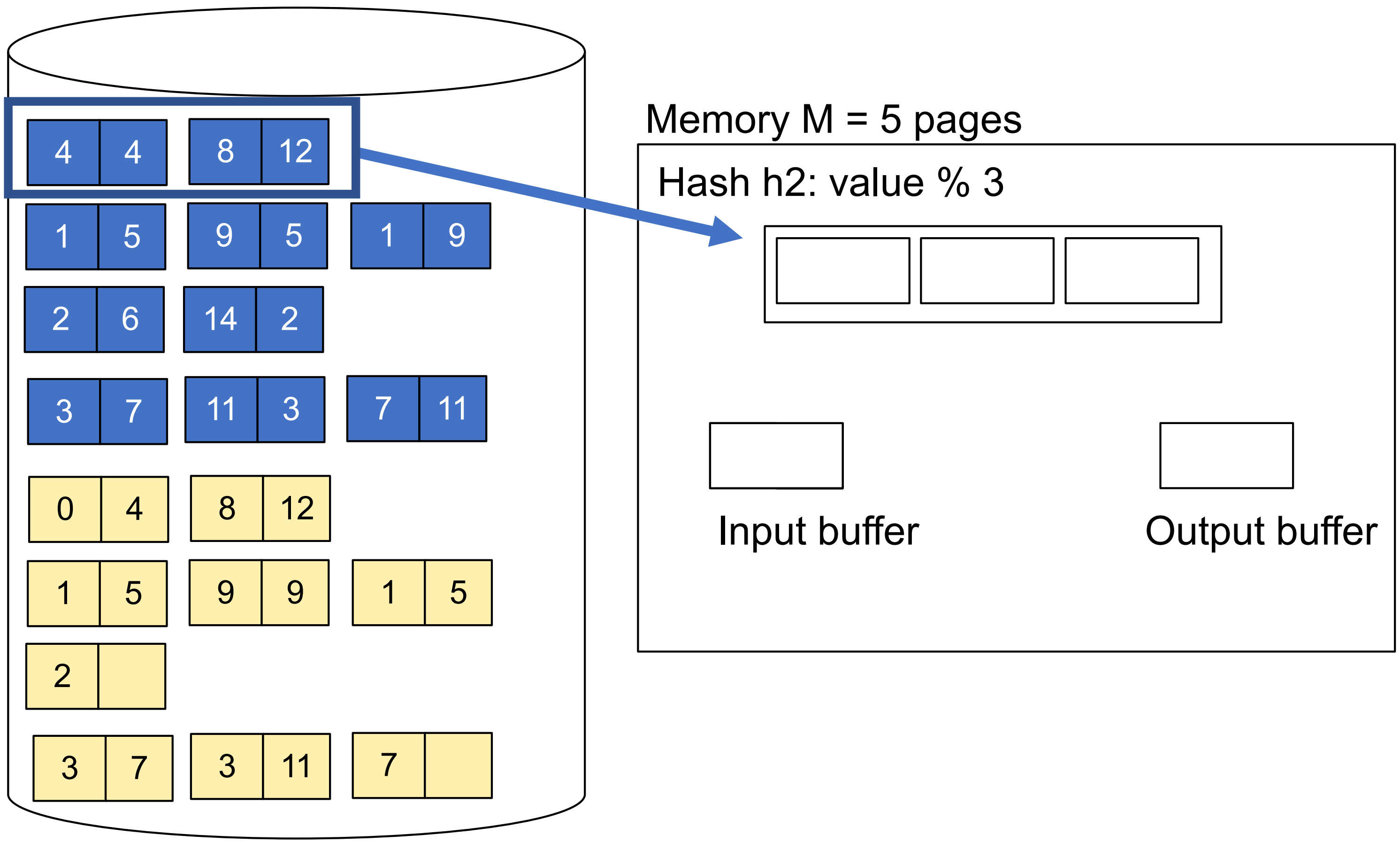
Partitioned Hash-Join Example

Step 2: Read relation R one page at a time and hash into same 4 buckets



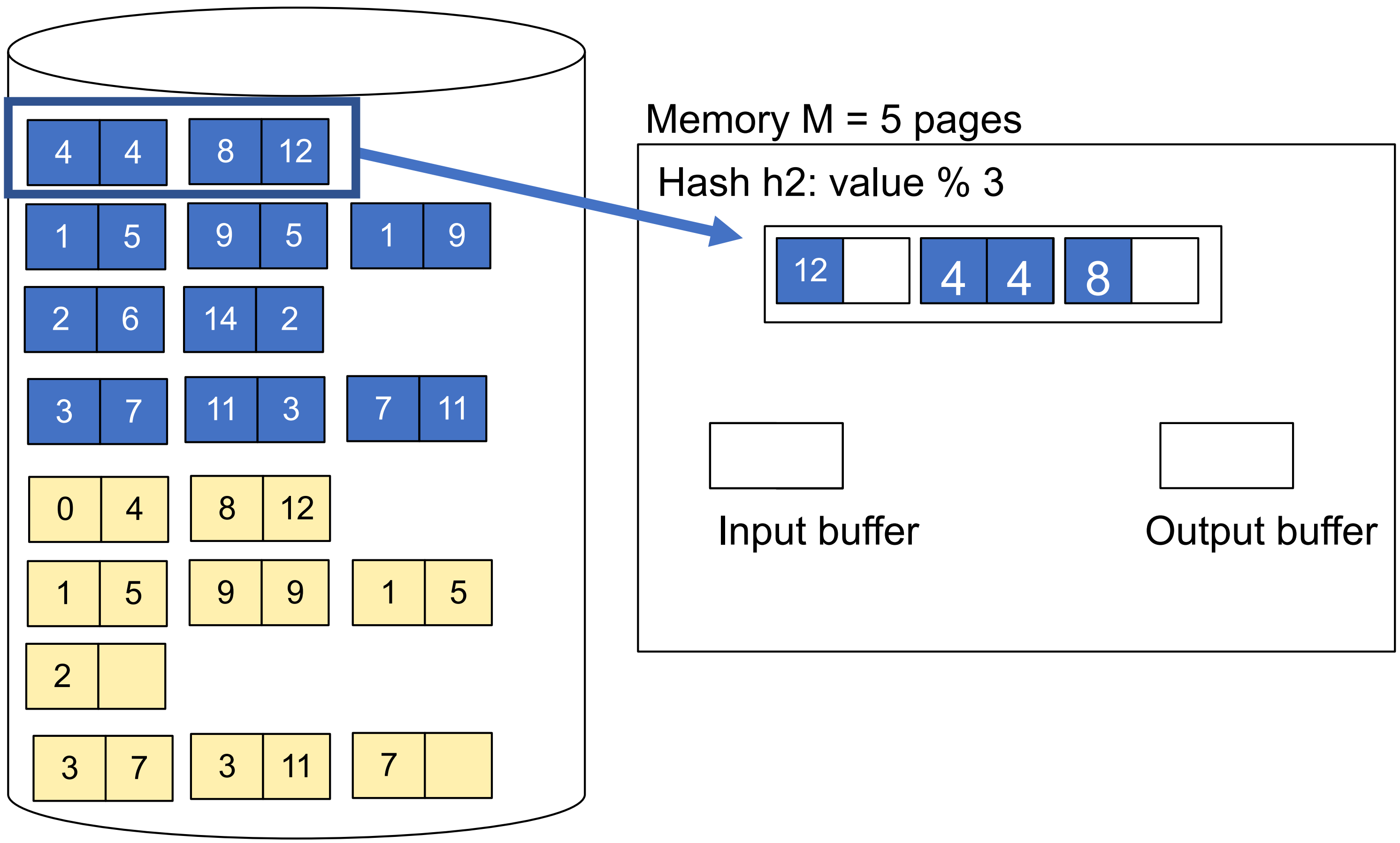
Partitioned Hash-Join Example

Step 3: Read one partition of R and create hash table in memory using a ***different*** hash function



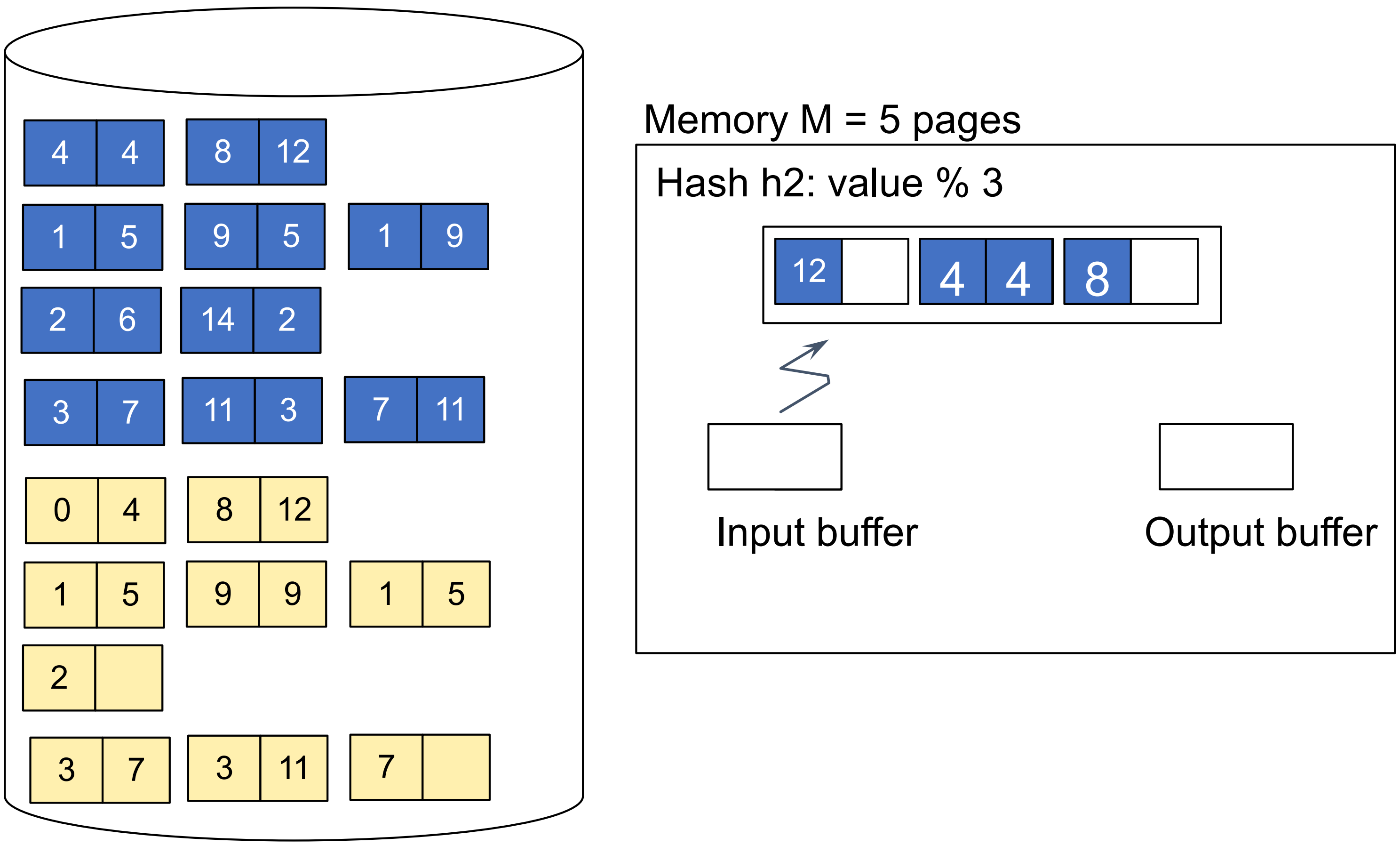
Partitioned Hash-Join Example

Step 3: Read one partition of R and create hash table in memory using a ***different*** hash function



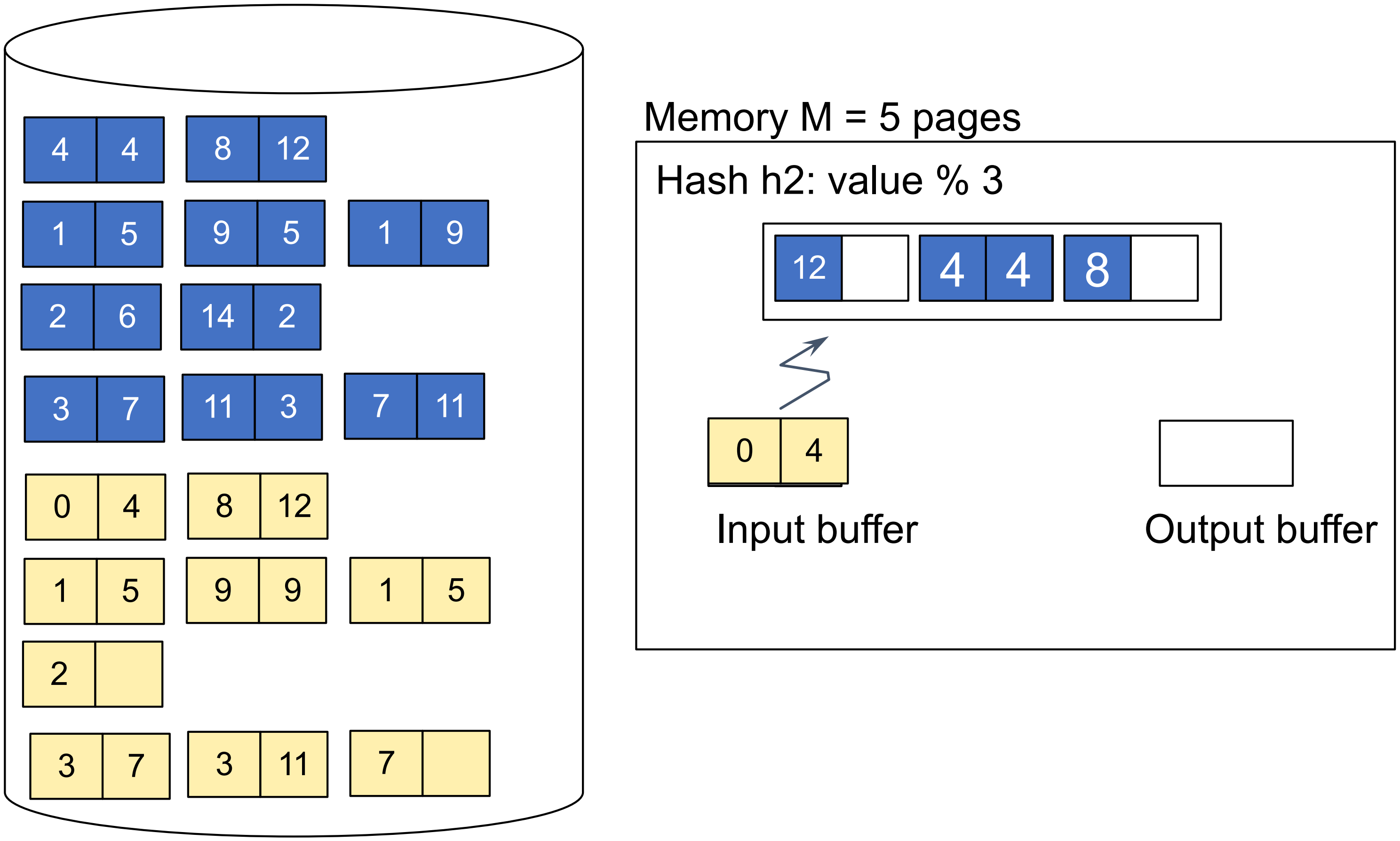
Partitioned Hash-Join Example

Step 3: Read one partition of R and create hash table in memory using a ***different*** hash function



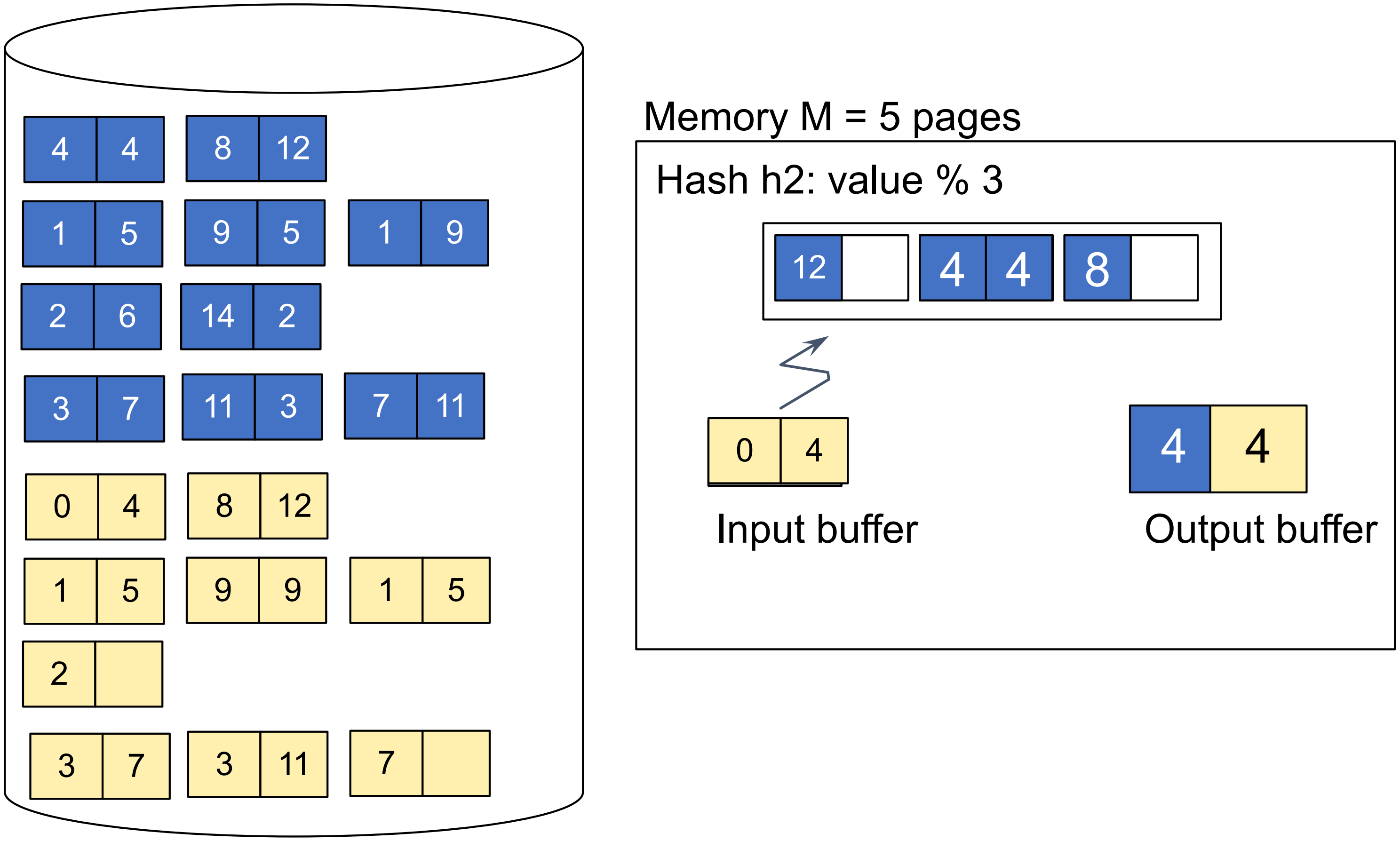
Partitioned Hash-Join Example

Step 4: Scan matching partition of S and probe the hash table
Step 5: Repeat for all the buckets
Total cost: 3B(R) + 3B(S)



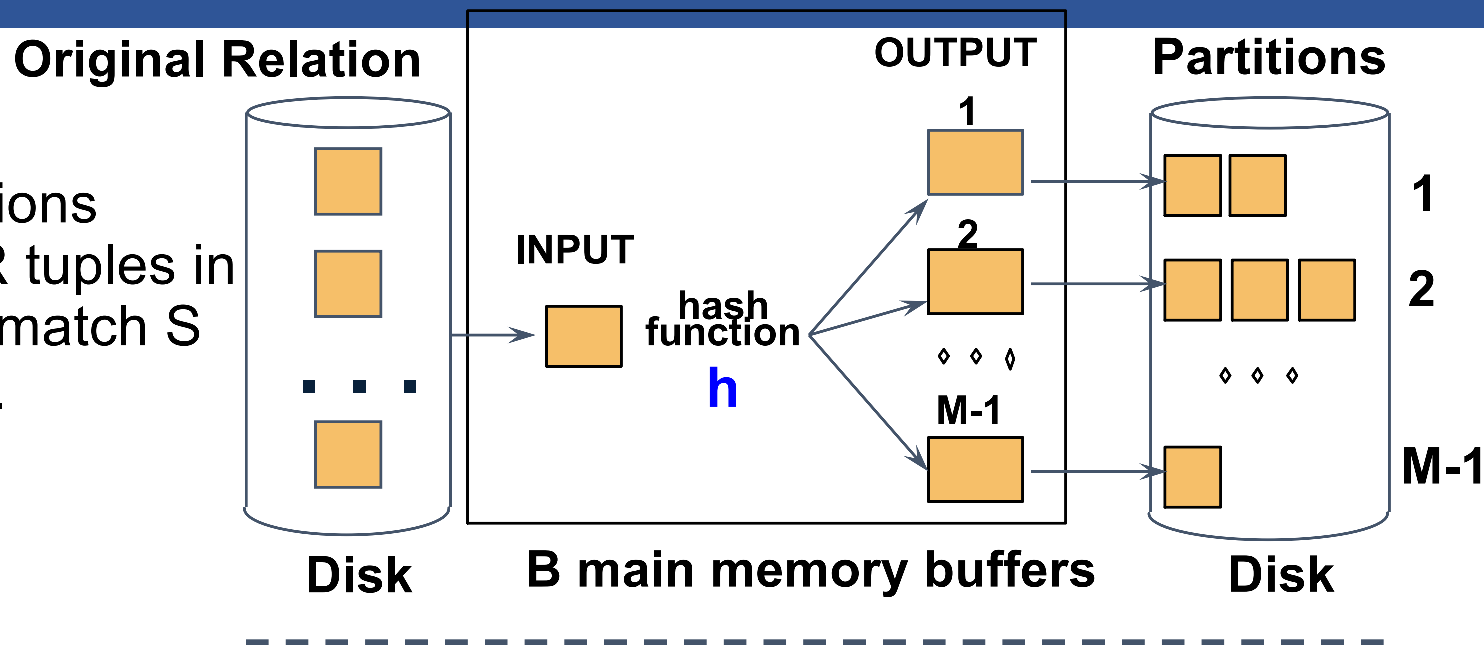
Partitioned Hash-Join Example

Step 4: Scan matching partition of S and probe the hash table
Step 5: Repeat for all the buckets
Total cost: 3B(R) + 3B(S)



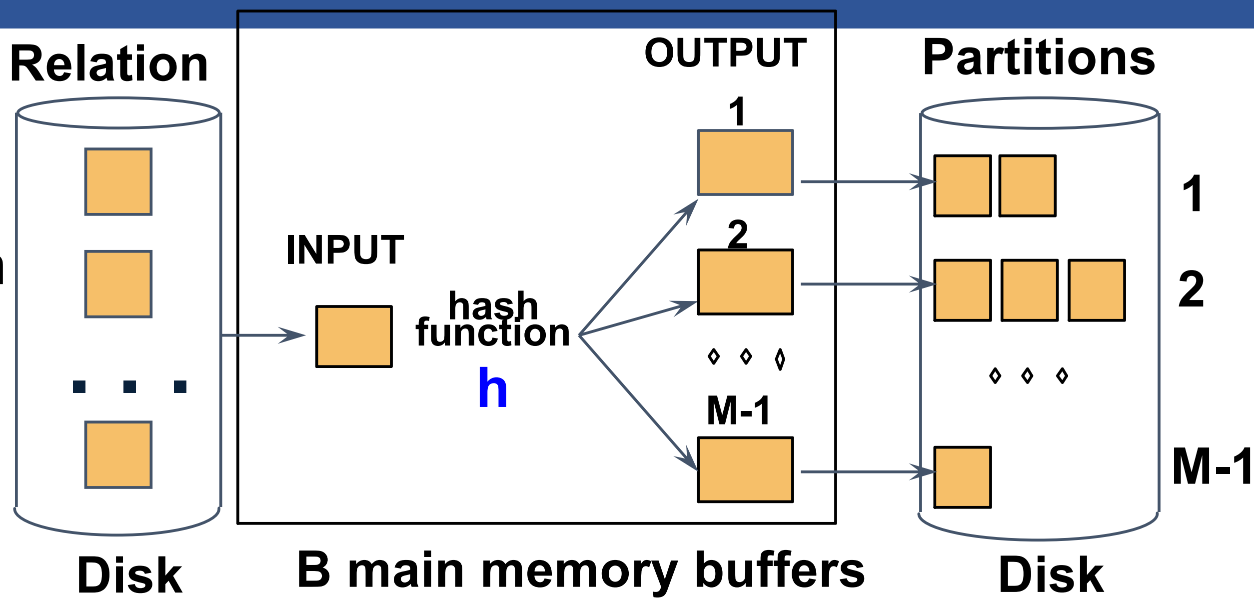
Partitioned Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.

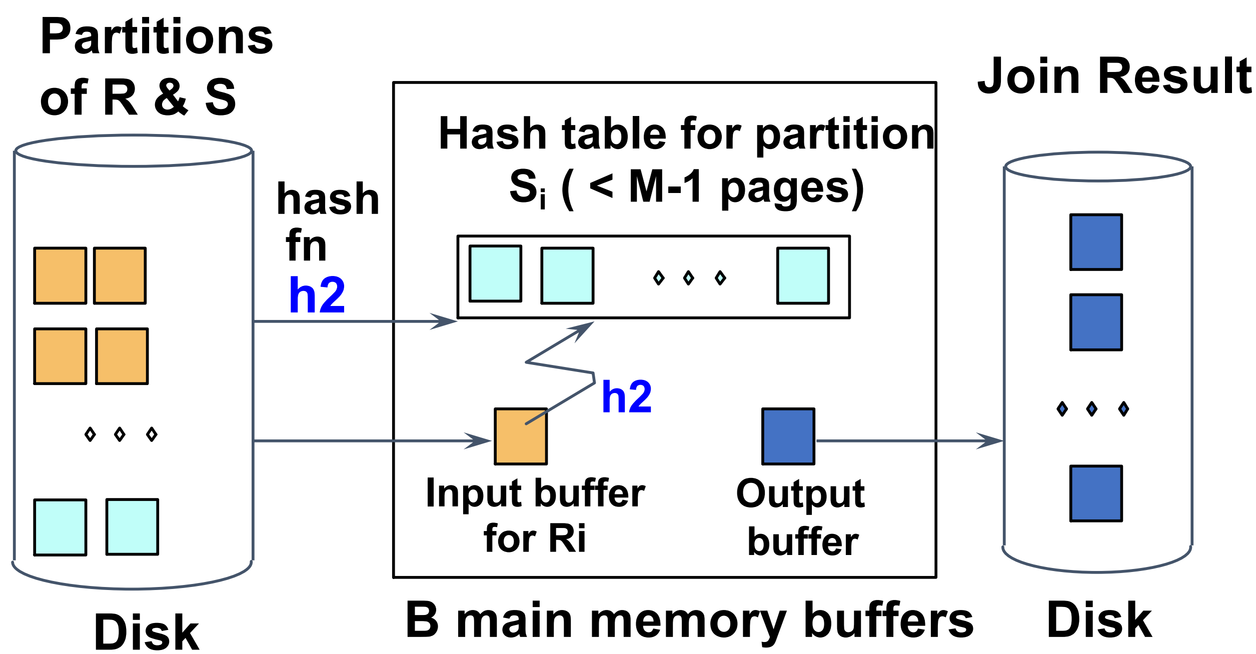


Partitioned Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Partitioned Hash-Join

- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M^2

Hybrid Hash Join Algorithm (see book)

- Partition S into k buckets

  t buckets S_1 , …, S_t stay in memory

  k-t buckets S_{t+1}, …, S_k to disk

- Partition R into k buckets
  - First t buckets join immediately with S
  - Rest k-t buckets go to disk

- Finally, join k-t pairs of buckets:

  (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Before We Go Into Query Plan Costs… How do Updates Work? (Insert/Delete)

Example Using Delete

```
delete from R where a=1;
```

Query plan

Delete

|

Filter ($\sigma_{a=1}$)

|

SeqScan

|

R

In SimpleDB, the Delete Operator calls BufferPool.deleteTuple()

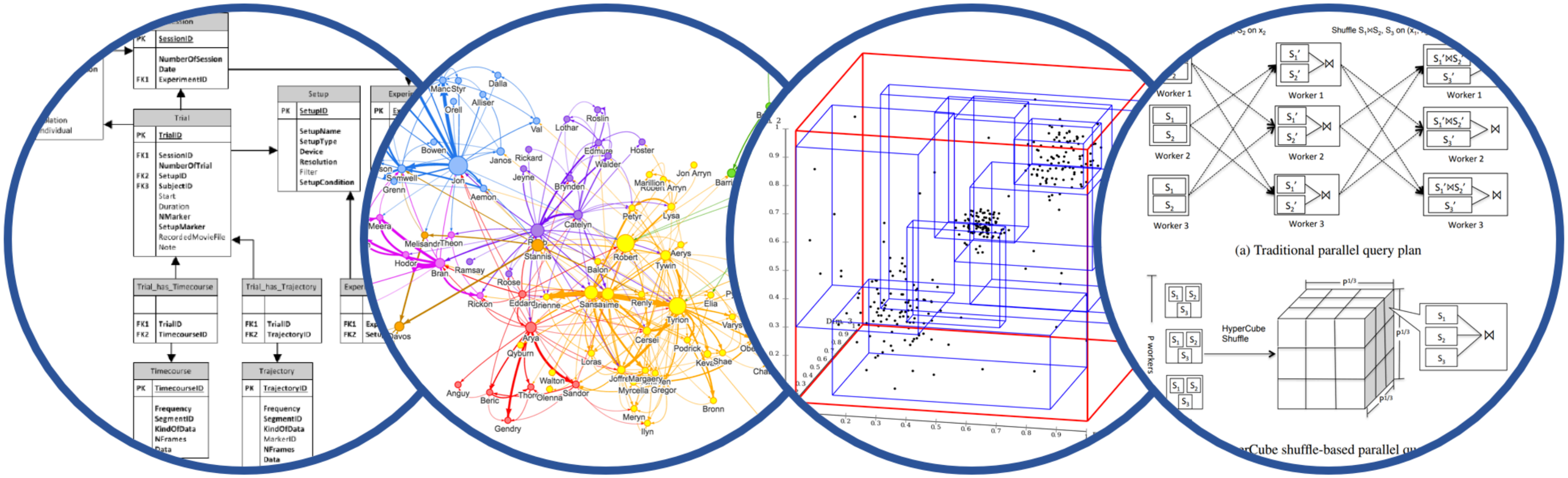
Why not call HeapFile.deleteTuple() directly?

Because there could also be indexes. Need some entity that will decide all the structures from where tuple needs to be deleted

BufferPool then calls HeapFile.deleteTuple()

Pushing Updates to Disk

- When inserting a tuple, HeapFile inserts it on a page but does not write the page to disk
- When deleting a tuple, HeapFile deletes tuple from a page but does not write the page to disk
- The buffer manager worries when to write pages to disk (and when to read them from disk)
- When need to add new page to file, HeapFile adds page to file on disk and then reads it through buffer manager



Database System Internals

Query Plan Costs

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Summary of External Join Algorithms

- Block Nested Loop: B(S) + B(R)*B(S)/(M-1)

- Index Join: B(R) + T(R)B(S)/V(S,a)
  (unclustered)

- Partitioned Hash: 3B(R)+3B(S);
  - min(B(R),B(S)) <= M^2

- Merge Join: 3B(R)+3B(S)
  - B(R)+B(S) <= M^2

Summary of Query Execution

- For each logical query plan
  - There exist many physical query plans
  - Each plan has a different cost
  - Cost depends on the data
- Additionally, for each query
  - There exist several logical plans
- Next lecture: query optimization
  - How to compute the cost of a complete plan?
  - How to pick a good query plan for a query?

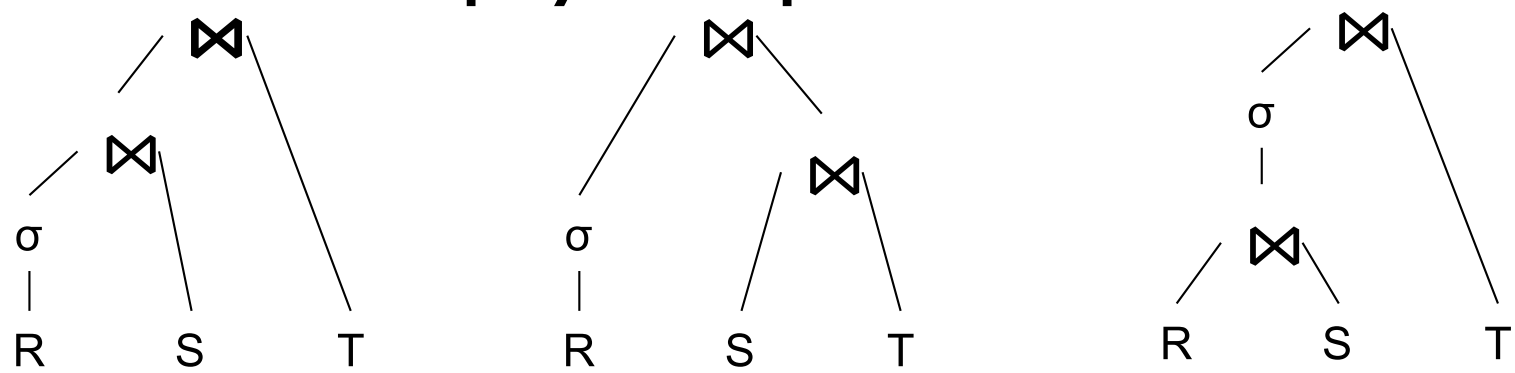
A Note About Skew

- Previously shown 2 pass join algorithms do not work for heavily skewed data

- For a sort-merge join, the maximum number of tuples with a particular join attribute should be the number of tuples per page:
  - This often isn't the case: would need multiple passes

Query Optimization Summary

Goal: find a physical plan that has minimal cost



What is the cost of a plan?

For each operator, cost is function of CPU, IO, network bw

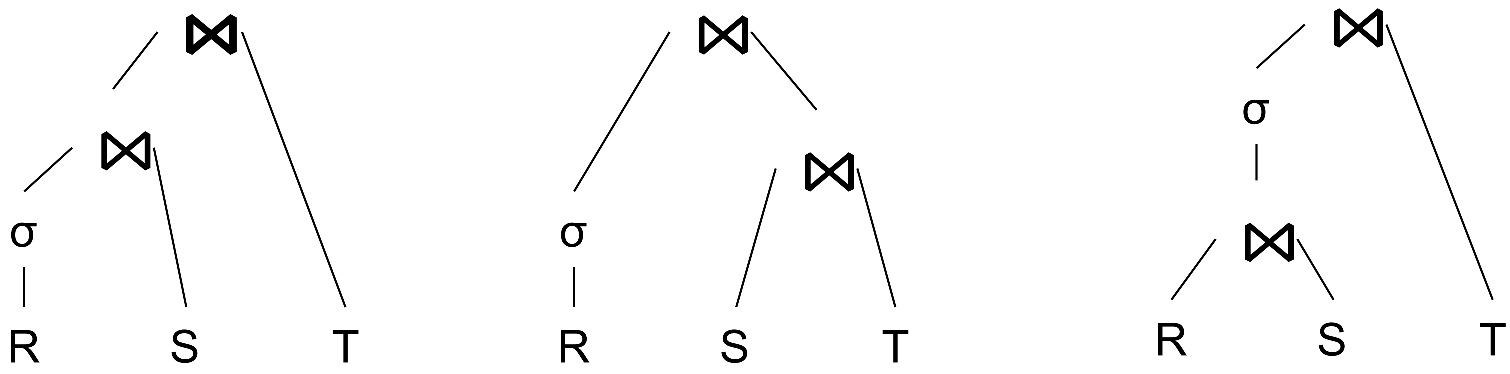
$$\text{Total\_Cost} = \text{CPUCost} + w_{IO}\,\text{IOCost} + w_{BW}\,\text{BWCost}$$

Cost of plan is total for all operators

In this class, we look only at IO

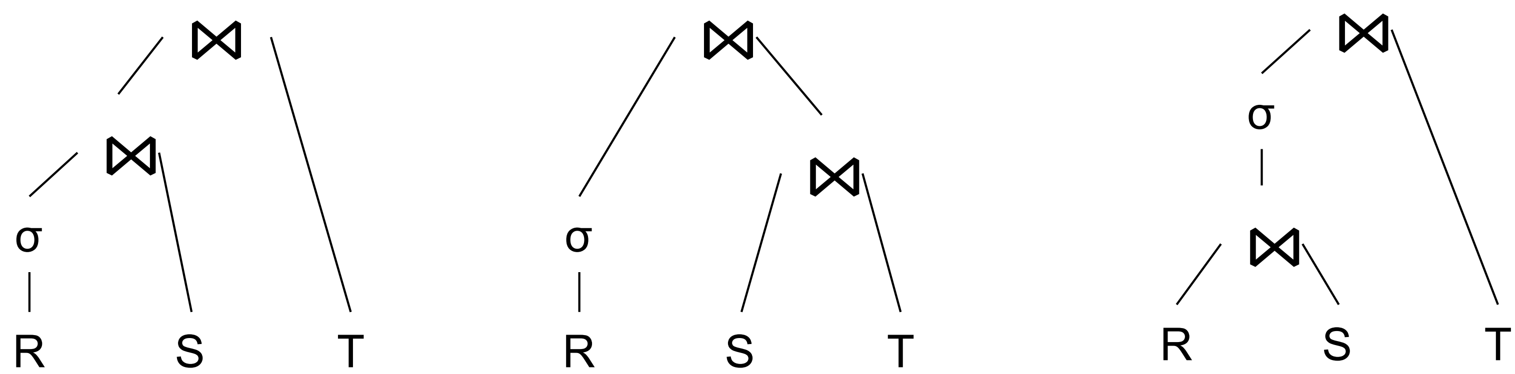
Query Optimization Summary

Goal: find a physical plan that has minimal cost



Query Optimization Summary

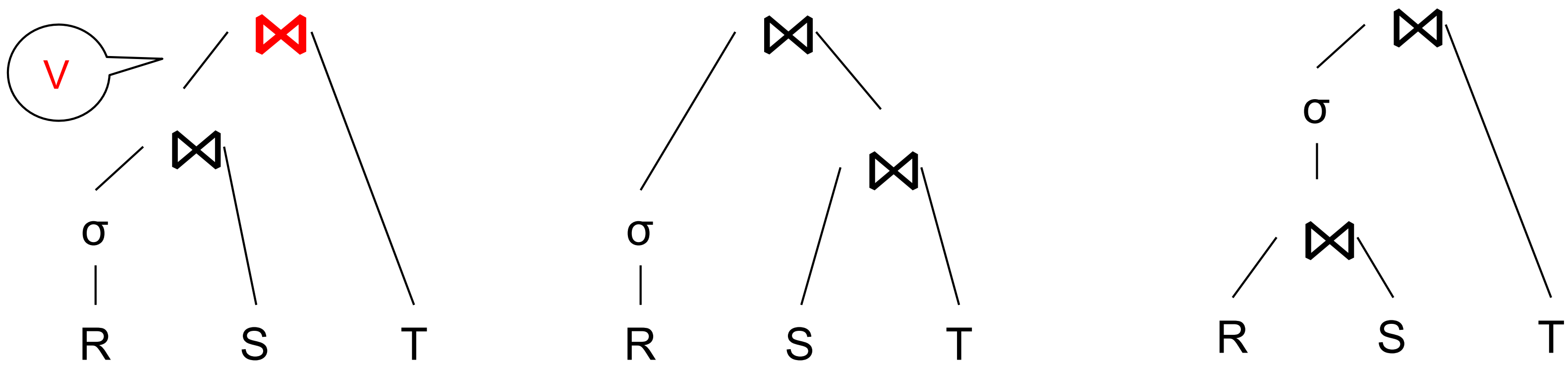
Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

Query Optimization Summary

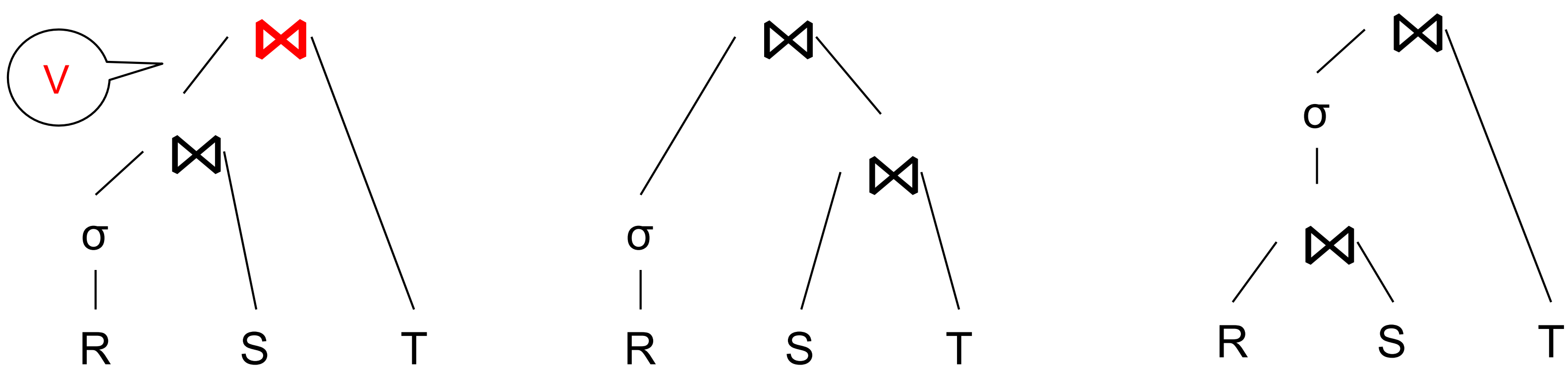
Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

Query Optimization Summary

Goal: find a physical plan that has minimal cost

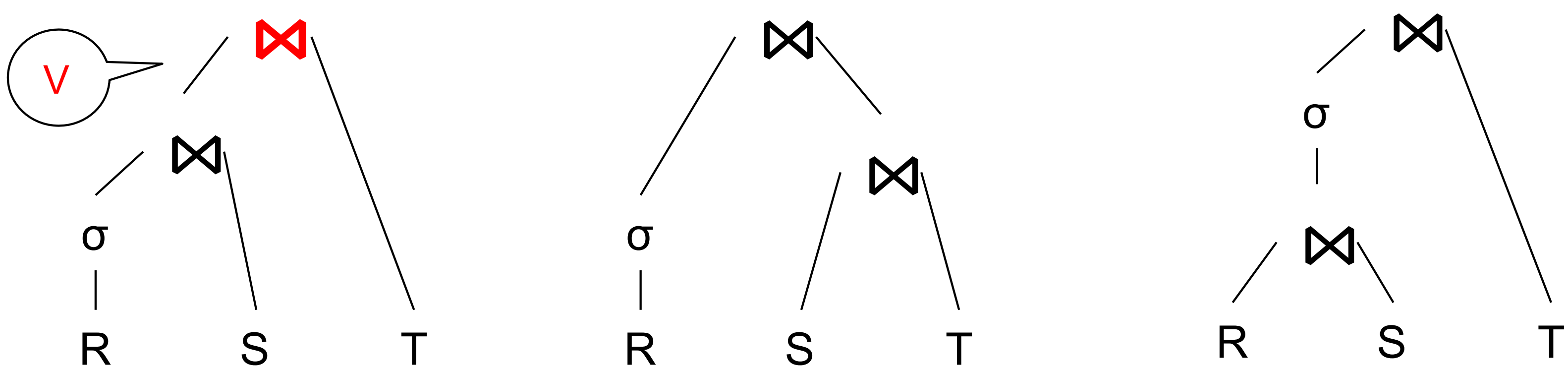


Know how to compute cost if know cardinalities

- Eg. Cost(V ⋈ T) = 3B(V) + 3B(T)
- B(V) = T(V) / PageSize
- T(V) = T(σ(R) ⋈ S)

Query Optimization Summary

Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

- Eg. Cost(V ⋈ T) = 3B(V) + 3B(T)
- B(V) = T(V) / PageSize
- T(V) = T(σ(R) ⋈ S)

Cardinality estimation problem: e.g. estimate T(σ(R) ⋈ S)

Database Statistics

- Collect statistical summaries of stored data

- Estimate size (=cardinality) in a bottom-up fashion
  - This is the most difficult part, and still inadequate in today's query optimizers
- Estimate cost by using the estimated size
  - Hand-written formulas, similar to those we used for computing the cost of each physical operator

Database Statistics

- Number of tuples (cardinality) T(R)
- Indexes, number of keys in the index V(R,a)
- Number of physical pages B(R)
- Statistical information on attributes
  - Min value, Max value, V(R,a)
- Histograms

- Collection approach: periodic, using sampling

Size Estimation Problem

```
Q = SELECT list
    FROM   R1, …, Rn
    WHERE  cond1 AND cond2 AND . . . AND condk
```

Given T(R1), T(R2), …, T(Rn)
Estimate T(Q)

How can we do this ? Note: doesn't have to be exact.

Size Estimation Problem

```
Q = SELECT list
    FROM   R1, …, Rn
    WHERE cond1 AND cond2 AND . . . AND condk
```

Remark: $T(Q) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Size Estimation Problem

```
Q = SELECT list
    FROM   R1, …, Rn
    WHERE  cond_1 AND cond_2 AND . . . AND cond_k
```

Remark: $T(Q) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Key idea: each condition reduces the size of T(Q) by some factor, called selectivity factor

Selectivity Factor

- Each condition cond reduces the size by some factor called selectivity factor

- Assuming independence, multiply the selectivity factors

Example

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

```
Q = SELECT *
      FROM R, S, T
      WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

Q: What is the estimated size of the query output T(Q) ?

Example

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

```
Q = SELECT *
      FROM R, S, T
      WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

Q: What is the estimated size of the query output T(Q) ?

A: T(Q) = 30k * 200k * 10k * 1/3 * 1/10 * ½ = 10^{12}

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- A < c /* $\sigma_{A<c}(R)$ */
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))

Selectivity Factors for Conditions

- A = c $\quad$ /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- A < c $\quad$ /* $\sigma_{A<c}(R)$ */
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))
- A = B $\quad$ /* $R \bowtie_{A=B} S$ */
  - Selectivity = 1 / max(V(R,A),V(S,A))
  - (will explain next)

Assumptions

- *Containment of values*: if V(R,A) <= V(S,B), then all values R.A occur in S.B
  - Note: this indeed holds when A is a foreign key in R, and B is a key in S

- *Preservation of values*: for any other attribute C, $V(R \bowtie_{A=B} S, C) = V(R, C)$ (or V(S, C))
  - Note: we don't need this to estimate the size of the join, but we need it in estimating the next operator

Selectivity of $R \bowtie_{A=B} S$

Assume $V(R,A) \le V(S,B)$

- A tuple t in R joins with $T(S)/V(S,B)$ tuple(s) in S
- Hence $T(R \bowtie_{A=B} S) = T(R)\,T(S) / V(S,B)$

$$T(R \bowtie_{A=B} S) = T(R)\,T(S) / \max(V(R,A),V(S,B))$$

Complete Example

Supplier(<u>sno</u>, sname, scity, sstate)
Supply(<u>sno, pno</u>, quantity)

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sno = y.sno
   and  y.pno = 2
   and x.scity = ‘Seattle’
   and x.sstate = ‘WA’
```

Suppy.sno references Supplier.sno

- Some statistics
  - T(Supplier) = 1000 records
  - T(Supply) = 10,000 records
  - B(Supplier) = 100 pages
  - B(Supply) = 100 pages
  - V(Supplier,scity) = 20, V(Suppliers,state) = 10
  - V(Supply,pno) = 2,500
  - Both relations are clustered
- M = 11

Physical Query Plan 1

T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno

(On the fly) π_{sname}

(On the fly) $\sigma_{scity='Seattle' \wedge sstate='WA' \wedge pno=2}$

(Nested loop memory optimized) $\bowtie_{sno = sno}$

Supplier (File scan)

Supply (File scan)

Selection and project on-the-fly
-> No additional cost.

Total cost of plan is thus cost of join:
= B(Supplier)+B(Supplier)*B(Supply)
= 100 + 100 * 100 / (11-1)
= 1,100 I/Os

Physical Query Plan 2

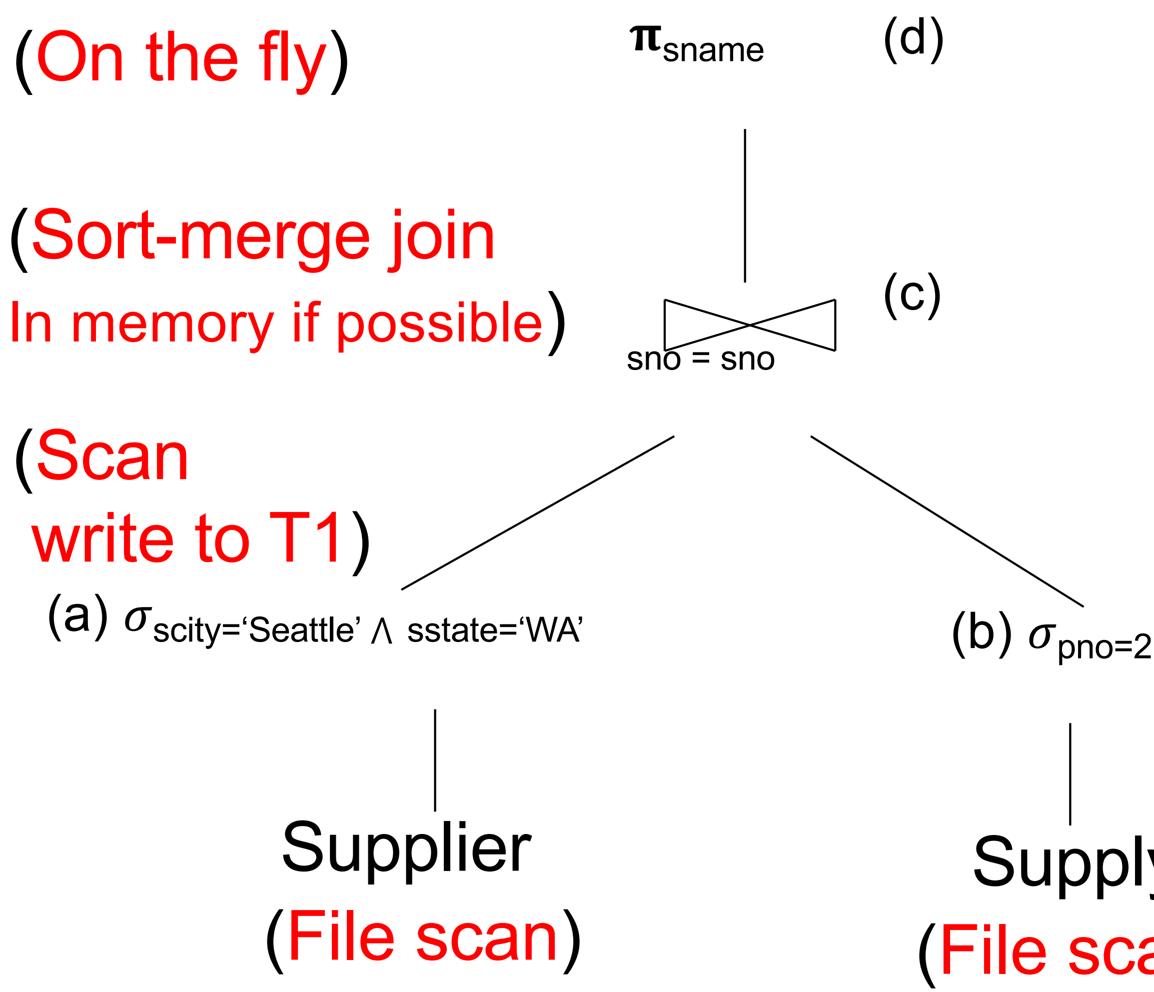
T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno



Total cost
= 100 + 100 * 1/20 * 1/10 (a)
+ 100 + 100 * 1/2500 (b)
+ 1 + 1 (c)
+ 0 (d)

Total cost ≈ **204 I/Os**

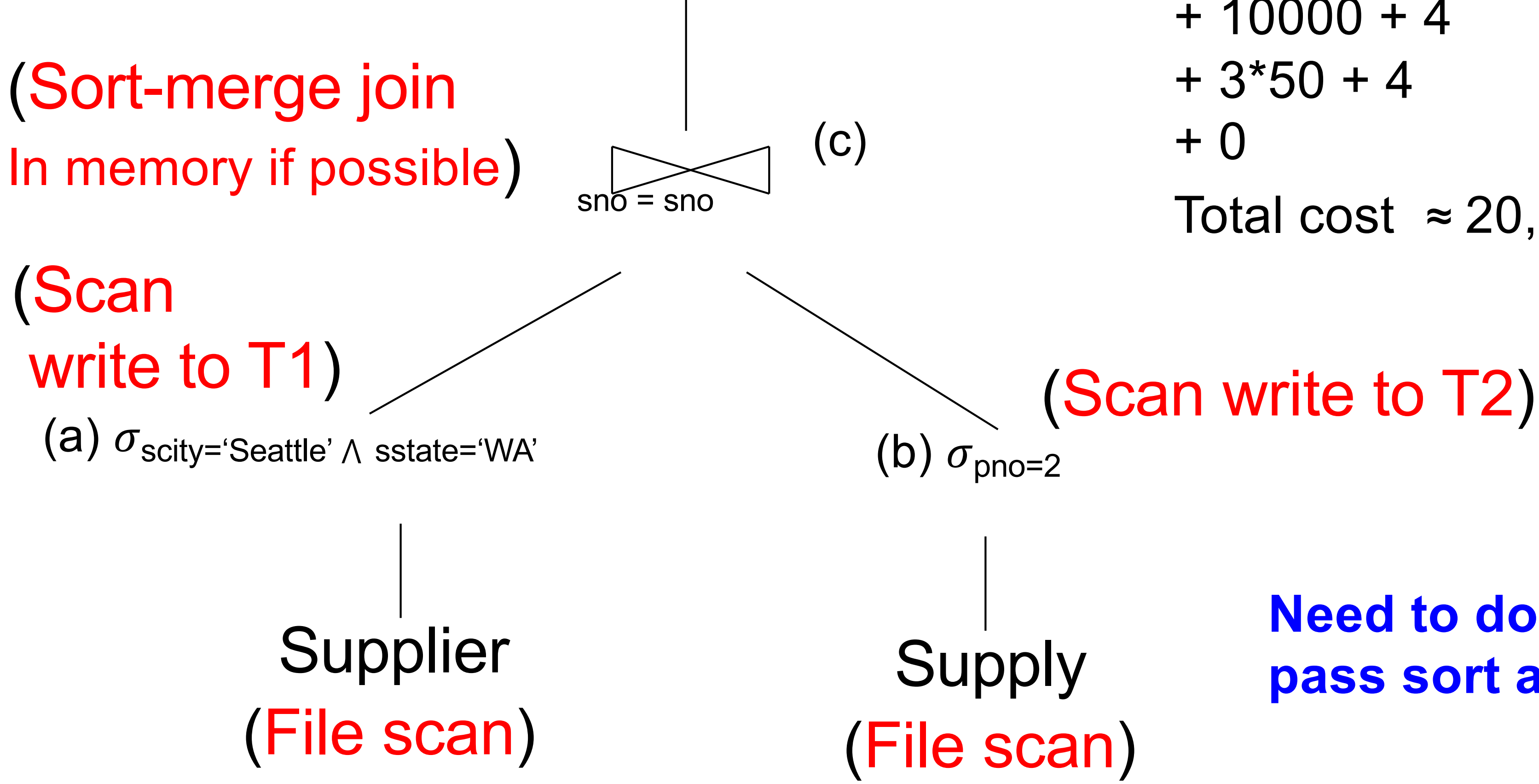
Plan 2 with Different Numbers

V(Supplier,scity) = 20   V(Supplier,state) = 10   V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno

What if we had:
10K pages of Supplier
10K pages of Supply



Total cost
= 10000 + 50 (a)
+ 10000 + 4 (b)
+ 3*50 + 4 (c)
+ 0 (d)

Total cost ≈ 20,208 I/Os

Need to do a two-pass sort algorithm

Physical Query Plan 3

T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno

(On the fly) (d) π_{sname}

(On the fly) (c) $\sigma_{scity=\text{'Seattle'} \wedge sstate=\text{'WA'}}$

(b) $\bowtie_{sno = sno}$ (Index nested loop)

Remember: Suppy.sno references Supplier.sno

4 tuples

(Use hash index) (a) $\sigma_{pno=2}$

Supply
(Hash index on pno)
Assume: clustered

Supplier
(Hash index on sno)
Clustering does not matter

Total cost
= 1 (a)
+ 4 (b)
+ 0 (c)
+ 0 (d)

Total cost ≈ **5 I/Os**

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(<u>ssn</u>, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Histograms

Employee(<u>ssn</u>, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Estimate = 25000 / 50 = 500

Estimate = 25000 * 6 / 50 = 3000

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}(\text{Empolyee}) = ?$ $\sigma_{age>28 \text{ and } age<35}(\text{Empolyee}) = ?$

Age:	0-20	20-29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Histograms

Employee(<u>ssn</u>, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}(\text{Empolyee}) = ?$ $\sigma_{age>28 \text{ and } age<35}(\text{Empolyee}) = ?$

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Estimate = 1200

Estimate = 1*80 + 5*500 = 2580

Types of Histograms

- How should we determine the bucket boundaries in a histogram?

Types of Histograms

- How should we determine the bucket boundaries in a histogram?

- Eq-Width
- Eq-Depth
- Compressed
- V-Optimal histograms

Histograms

Employee(ssn, name, age)

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Eq-depth:

Age:	0-33	33-38	38-43	43-45	45-54	> 54
Tuples	1800	2000	2100	2200	1900	1800

Compressed: store separately highly frequent values: (48,1900)

V-Optimal Histograms

- Defines bucket boundaries in an optimal way, to minimize the error over all point queries
- Computed rather expensively, using dynamic programming
- Modern databases systems use V-optimal histograms or some variations

Difficult Questions on Histograms

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY ?
- *Not* updated during database update, but recomputed periodically
  - WHY ?
- Multidimensional histograms rarely used
  - WHY ?

Difficult Questions on Histograms

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY? All histograms are kept in main memory during query optimization; plus need fast access
- *Not* updated during database update, but recomputed periodically
  - WHY? Histogram update creates a write conflict; would dramatically slow down transaction throughput
- Multidimensional histograms rarely used
  - WHY? Too many possible multidimensional histograms, unclear which ones to choose