



Query Processing: Physical Operators

Abdu Alawini

University of Illinois at Urbana-Champaign

CS411: Database Systems

Scanning Tables

- Read the entire contents of a relation R
 - or partial contents: all tuples that satisfy a criterion
- **Table-scan:** R is stored in some area of secondary storage (disk), in blocks.
 - These blocks are known to the system.
 - Can get all blocks one by one.
- **Index-scan:** if we have index to find the blocks.
 - “Scan the index, grab the relevant blocks”
 - This can be particularly useful for getting tuples that satisfy a predicate

Sorting while scanning tables

- May want to sort the tuples as we read them: “Sort-scan”
- Why?
 - ORDER BY in query
 - Some RA operations are implemented using sort
 - Union, Intersection, ...
 - Future operations may be cheaper (e.g., GROUP BY)
- How?
 - If indexed, then trivial
 - If fits in main memory, then table-scan or index-scan and sort in memory
 - If too large, “multiway merge sort” (see later)

Sorting with Scanning

First cut: relation is small

- Have M main memory blocks available for use
- Assumption: $B(R) \leq M$
- Method:
 - Read all $B(R)$ blocks, sort, write
- Cost: $B(R)$

Sorting with Scanning

Second cut: Relation is large

- Have M main memory blocks available for use
- How would you do this?

Sorting with Scanning

Second cut: Two pass “multi-way merge sort”

- Have M main memory blocks available for use
- Step 1:
 - Read M blocks at a time, sort, write
 - Result: have runs of length M on disk
- Step 2:
 - Merge $M-1$ runs at a time, construct output, write to disk
 - Result: have runs of length $M(M-1) \approx M^2$

Example, cont.

$M = 3$, each block holds two values

Data on disk: [3,9] [8,1] [7,6] [9,5] [2,0] [3,8]

Step 1: Sort Runs

[3,9] [8,1] [7,6] => [1,3] [6,7] [8,9]

[9,5] [2,0] [3,8] => [0,2] [3,5] [8,9]

Step 2: Merge Sorted Runs

[1,3][0,2][_,_] => [1,3][0,2][0,1] => output [0,1]

[1,3][0,2][_,_] => [1,3][0,2][2,3] => output [2,3]

[6,7][3,5][_,_] => [6,7][3,5][3,5] => output [3,5]

[6,7][8,9][_,_] => [6,7][8,9][6,7] => output [6,7]

[8,9][8,9][_,_] => [8,9][8,9][8,8] => output [8,8]

Multi-way merge sort cost

Cost: $3B(R)$,

Assumption: $B(R) < M^2$; *why?*

$$B(R) / M \leq M-1 \Leftrightarrow B(R) \leq M(M-1) < M^2$$

Cost of the Scan Operator

- Table scan: $B(R)$; Sort-scan: $3B(R)$
[assuming $B(R) \leq M(M-1)$]
- Index scan: $B(R) + \# \text{blocks of the index}$
 $\approx B(R)$; Sort-scan: $B(R)$
(assuming index on sorting key)

Cost of the Scan Operator (Cont.)

- Unclustered relation:
 - we have assumed so far that all tuples of R are “clustered”, i.e., stored in $\sim B$ blocks.
 - If tuples of R are interspersed with tuples of other relations, then cost:
 - scan: $T(R)$; sort: $T(R) + 2B(R)$

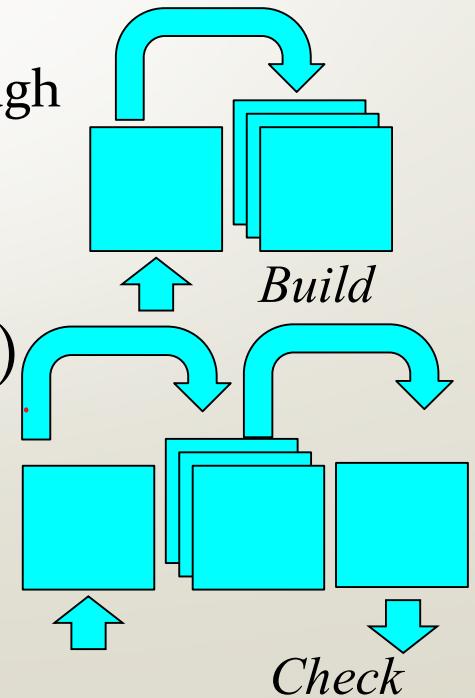
Now Generalizing...

- Done with costing of scan operators
- Now we will generalize to other operators
 - A lot more complex than scan
 - Many different classifications/tradeoffs

One-pass Algorithms

Binary operations:

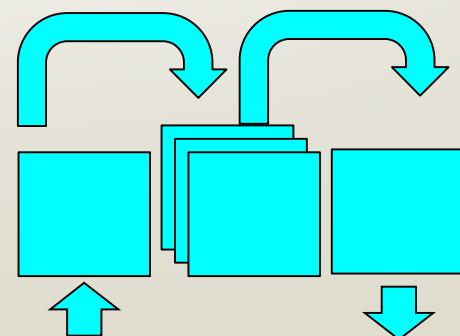
- Read the smaller relation, store in memory.
 - Build some data structure so that tuples can be accessed and inserted efficiently. (Hash table or B-tree)
 - Read the other relation, block by block, go through each tuple and decide whether to output or not.
- **Memory requirement:** $M > \min(B(R), B(S))$
[more precisely $M \geq \min(B(R), B(S)) + 2$]
- **Cost:** $B(R) + B(S)$

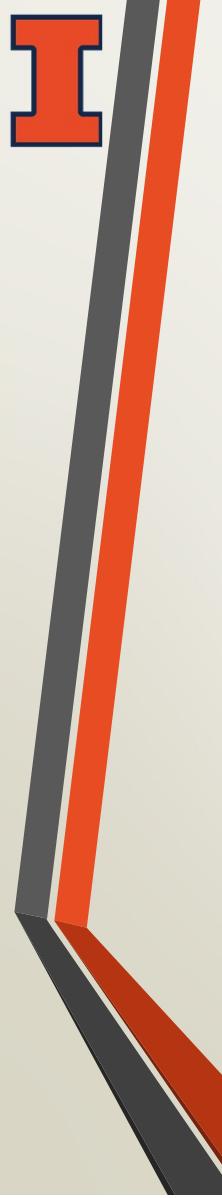


One-pass Algorithms

Set Union:

- Read the smaller relation (say S) into memory .
 - Build a search structure whose search key is entire tuple. (*up to $M-2$ blocks can be used*)
 - Read R, block by block (*1 block*). Once a block is loaded, for each tuple t in that block, see if t is in S; if not, copy t to the output block (*1 block*).
 - Output distinct tuples of S from dictionary structure
- **Memory requirement:**
- $M \geq \min(B(R), B(S)) + 2$.
- **Cost:** $B(R)+B(S)$





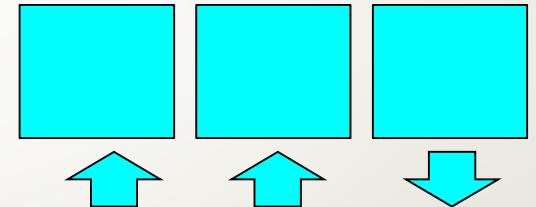
Nested-Loop JOIN

1.5-pass algorithm

Nested Loop Joins

- Simple block-based nested loop $R \bowtie S$
- R=outer relation, S=inner relation

```
for each block r in R do
    for each block s in S do
        for each tuple r1 in r do
            for each tuple s1 in s do
                if r1 and s1 join then output (r,s)
```



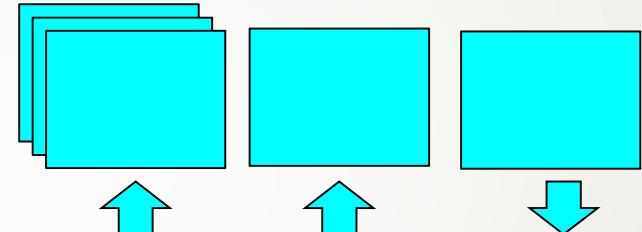
- Memory: $M \geq 3$; Cost: $B(R) B(S)$

Can we do even better?

Hint: lots of memory that can be used...

Nested Loop Joins

- Block-based Nested Loop Join



```
for each (M-2) blocks r of R do
    for each block s of S do
        for each tuple r1 in r do
            for each tuple s1 in s do
                if r1 and s1 join then output(r,s)
```

one block reserved for the output buffer

Nested Loop Joins

$M = 4;$

$R = [(1, a), (1, b)][(1, c), (1, d)][(1, e), (1, f)][(1, g)(1, h)]$

$S = [(1, x), (1, y)][(1, z), (1, w)][(1, v), (1, u)][(1, t), (1, s)]$

Nested Loop Joins

$M = 4;$

$R = [(1, a), (1, b)][(1, c), (1, d)][(1, e), (1, f)][(1, g)(1, h)]$

$S = [(1, x), (1, y)][(1, z), (1, w)][(1, v), (1, u)][(1, t), (1, s)]$

R	S	Output
$[(1, a), (1, b)]$	$[(1, c), (1, d)]$	$[(1, x), (1, y)]$
$[(1, a), (1, b)]$	$[(1, c), (1, d)]$	$[(1, z), (1, w)]$
$[(1, a), (1, b)]$	$[(1, c), (1, d)]$	$[(1, v), (1, u)]$
$[(1, a), (1, b)]$	$[(1, c), (1, d)]$	$[(1, t), (1, s)]$
$[(1, e), (1, f)]$	$[(1, g)(1, h)]$	$[(1, x), (1, y)]$
$[(1, e), (1, f)]$	$[(1, g)(1, h)]$	$[(1, z), (1, w)]$
$[(1, e), (1, f)]$	$[(1, g)(1, h)]$	$[(1, v), (1, u)]$
$[(1, e), (1, f)]$	$[(1, g)(1, h)]$	$[(1, t), (1, s)]$

Nested Loop Joins

- Block-based Nested Loop Join
- Cost:
 - Read R once: cost $B(R)$
 - Outer loop runs $B(R)/(M-2)$ times, and each time need to read S: costs $B(S)B(R)/(M-2)$
 - Total cost: $B(R) + B(S)B(R)/(M-2)$
[Approximately $B(R) B(S)/M$]
- Notice: it is better to iterate over the smaller relation first- i.e., R



Two-pass Algorithms

Two-Pass Algorithms Based on Sorting

Binary operations: $R \cap S$, $R \cup S$, $R - S$

- Idea: sort R, sort S, then do the right thing
 - *What do we sort on?*
- A closer look:
 - Step 1: split R into sorted runs of size M, then split S into sorted runs of size M. Cost: $2B(R) + 2B(S)$
 - Step 2: merge *all* x runs from R; merge all y runs from S; output a tuple on a case by case basis ($x + y \leq M-1$)
 - *Why do we need all the sorted runs in memory at once?*
- Total cost: ??
 - $3B(R) + 3B(S)$
- Assumption: $B(R) + B(S) \leq M(M-1) < M^2$

Two-Pass Algorithms Based on Sorting

Join $R \bowtie S$. Let's recap what we've seen so far -- extremes

- (a) $\min(B(R), B(S)) \leq M-2$: Load smaller table to memory and load other table block by block. **Cost:** $B(R)+B(S)$. This is the one-pass algorithm.
- (b) $\min(B(R), B(S)) > M-2$: Load to memory $M-2$ blocks of S ; go over every block of R ; repeat. **Cost:** $\sim B(R)B(S)/M$. This is the nested-loop join algorithm, can operate whenever $M \geq 3$

Nested loop join is the only option

if $\min(B(R), B(S)) > M-2$,

but is too expensive, quadratic ($B(R)B(S)$).

Two-Pass Algorithms Based on Sorting

Join $R \bowtie S$

- Start by writing out runs of R and S on the join attribute:
 - Cost: $2B(R)+2B(S)$ (because need to write to disk)
- “Merge” runs of both relations in sorted order, match tuples
 - Cost: $B(R)+B(S)$
- **Total cost:** $3B(R)+3B(S)$
- **Assumption:** $B(R) + B(S) \leq M(M-1)$

Two-Pass Algorithms Based on Sorting

- One difficulty: many tuples in R may match many in S
 - If at least one set of tuples fits in M, we are OK
 - Otherwise need nested loop, higher cost
 - But let's assume that this is not the case; we are in a good situation – can we do even better?
- See Section 15.4.6.



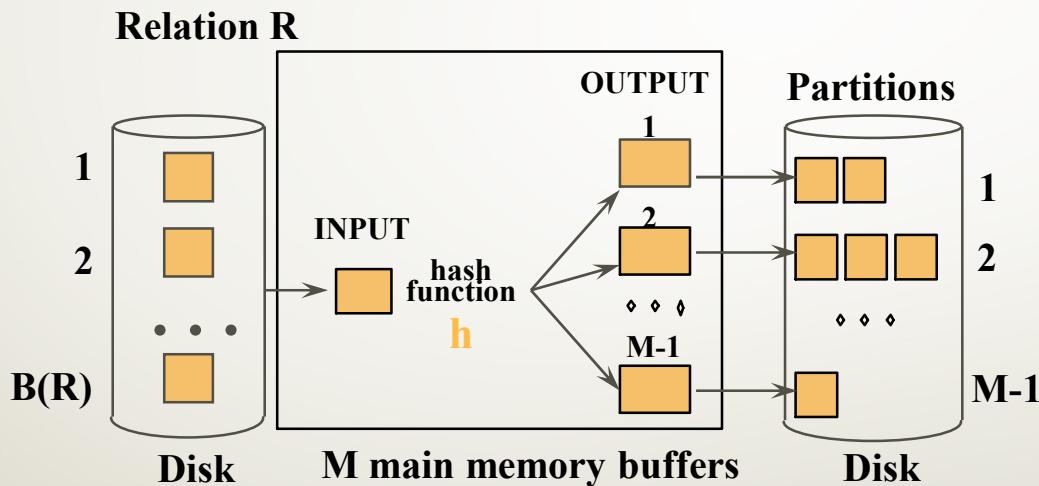
Two-Pass Hash-Based Algorithms

Two Pass Algorithms Based on Hashing

- Idea: **partition** a relation R into M roughly equal sized buckets, on disk
 - Hashing is crucial to affect this partitioning
- These buckets can then be examined “**in one shot**”, independent of other buckets
 - Everything that needs to be considered together is in a small number of buckets (one or two)
- Can therefore do the operation by only looking at one or two buckets at a time.

Two Pass Algorithms Based on Hashing

- How to partition a relation R into $(M-1)$ roughly equal sized buckets (on disk)?
- Each bucket has size approx. $B(R)/(M-1)$



- Does each bucket fit in main memory ?
 - Yes if $B(R)/(M-1) \leq M$, roughly $B(R) < M^2$

Recap: One pass Hashing-based Join

- $R \bowtie S$
- Scan S into memory, build buckets in main memory
- Then scan R , hash the tuples of R , output those that match
- Assuming that the smaller table is smaller than the memory available.

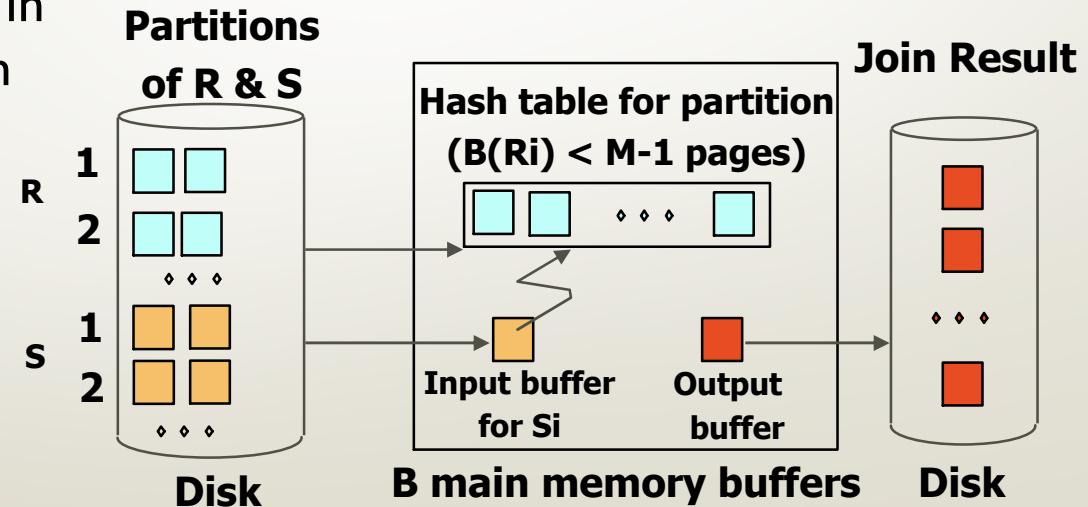
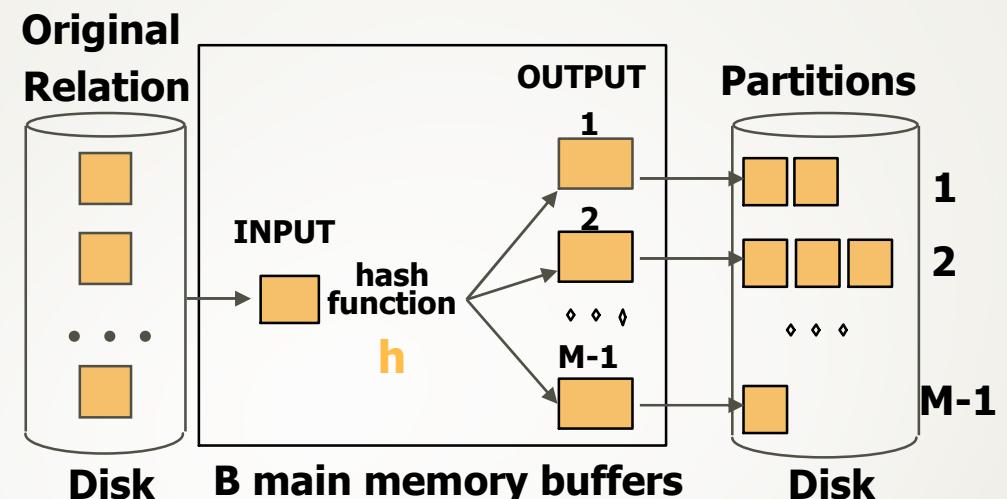
Two pass Hashing-based Join

$R \bowtie S$

- Step 1:
 - Hash S into $(M-1)$ buckets, using join attribute(s) as hash key
 - Send all buckets to disk
- Step 2
 - Hash R into $(M-1)$ buckets, using join attribute(s) as hash key
 - Send all buckets to disk
- Step 3
 - Join every pair of buckets with the same bucket number. Use the one pass algorithm for this.
 - Works when for each bucket no. i , either R_i or S_i fits in memory.

Hash-Join

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



Two pass Hashing-based Join

$R \bowtie S$

- Step 1:
 - Hash S into $(M-1)$ buckets, using join attribute(s) as hash key
 - Send all buckets to disk
- Step 2
 - Hash R into $(M-1)$ buckets, using join attribute(s) as hash key
 - Send all buckets to disk
- Step 3
 - Join every pair of buckets with the same bucket number. Use the one pass algorithm for this.
 - Works when for each bucket no. i , either R_i or S_i fits in memory.
 - $\min(B(R), B(S))/(M-1) \leq (M-2) \Leftrightarrow \min(B(R), B(S)) \leq (M-1)(M-2) < M^2$
- Cost = $3(B(R)+B(S))$

Sort-based vs Hash-based (for binary ops)

- For sorting-based implementations of binary operations, size requirement was $B(R)+B(S) \leq M(M-1) < M^2$. For hashing-based implementation, requirement is $\min(B(R), B(S)) \leq (M-1)(M-2) < M^2$.
 - Hashing wins!
- Output of sorting-based algorithms are in sorted order, which may be useful for subsequent operations.
 - Sorting wins!
- Hashing-based algorithms rely on buckets being of roughly equal size. This may be a problem.
 - Sorting wins!
- Other differences too. Read 15.5.7.



Index-based Algorithms

Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- Clustered index on a: $\text{cost} = B(R)/V(R,a)$
 - $V(R, a)$ was defined as the number of distinct values of the attribute a.
- Unclustered index on a: $\text{cost} = T(R)/V(R,a)$

Index Based Selection

- Example: $B(R) = 2000$, $T(R) = 100,000$, $V(R, a) = 20$, compute the cost of $\sigma_{a=v}(R)$.
- Cost of un-indexed selection:
 - If R is clustered on some key: $B(R) = 2000$ I/Os
 - If R is unclustered for all keys: $T(R) = 100,000$ I/Os
- Cost of index-based selection:
 - If index on a is clustered: $B(R)/V(R,a) = 100$
 - If index on a is unclustered: $T(R)/V(R,a) = 5000$
- Note: when $V(R,a)$ is small, then unclustered index is useless

Index Based (Nested-Loop) Join

- $R \bowtie S$
- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S
- Assume R is clustered. Cost:
 - If index (on S) is clustered: $B(R) + T(R)B(S)/V(S,a)$
 - If index (on S) is unclustered: $B(R) + T(R)T(S)/V(S,a)$
- Looks useless (Example 15.12), but see text (paragraph following the example).

Index Based (Sort-Merge or Zig-zag) Join

- Assume both R and S have a sorted index (B+ tree) on the join attribute. (A clustering index.)
- Then perform a merge join (called zig-zag join)
 - This is only the last step of the “two pass sorting-based join” algorithm we saw previously.
 - “bring all relevant tuples into memory”
- Cost: $B(R) + B(S)$