Query Processing – Part 2: Physical Operators

Abdu Alawini
University of Illinois at Urbana-Champaign

CS411: Database Systems

Leaning Objectives

After this lecture, you should be able to:

- Develop one-pass operators and understand their cost and memory requirements
- Develop two-pass (sort/hash-based) operators and understand their cost and memory requirements
- Develop index-based physical operators and understand their cost and memory requirements.

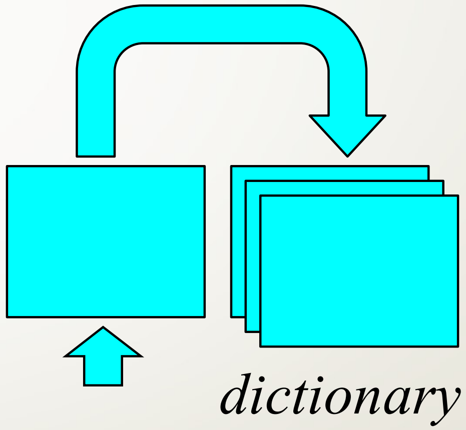
Today's Lecture

- One-Pass Algorithms
- Nested-Loop Join
- Two-Pass Algorithms
  - Sort-Based
  - Hash-based
- Index-Based Algorithms

One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep a "dictionary" in memory:
  - Unique tuples seen so far
  - balanced search tree, hash table, etc.
- Memory requirement: $M > B(\delta(R))$
  - [specifically, $M >= B(\delta(R)) + 1$]
  - We need to estimate $B(\delta(R))$ in advance, when planning whether to use this algorithm. Significant penalties if we underestimated!
- Cost: $B(R)$, again assuming clustered relation on disk, no index use.



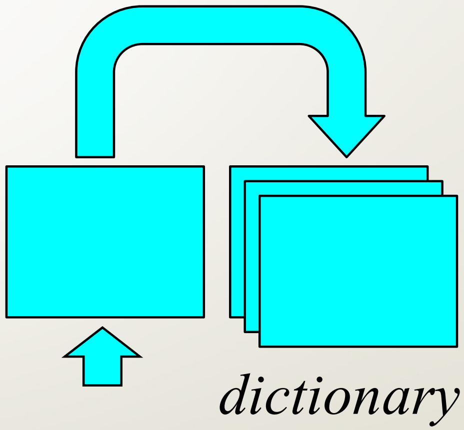
One-pass Algorithms

Grouping: $\gamma_{city,\ sum(price)}\ (R)$

SELECT city, SUM(price) FROM R GROUP BY city

- Need to keep a dictionary in memory
  - Each entry in the dictionary is: (city, sum(price))

Q: What if "SUM" was replaced with "AVG" ?



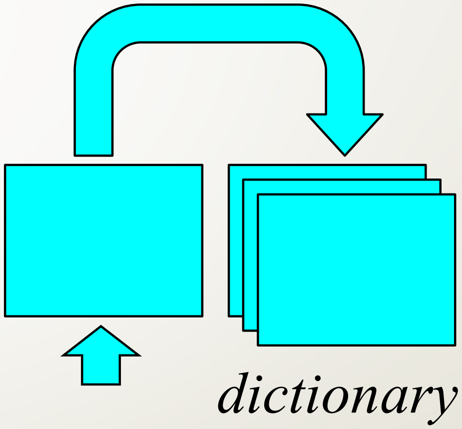
One-pass Algorithms

Grouping: $\gamma_{city,\ sum(price)}\ (R)$

SELECT city, SUM(price) FROM R GROUP BY city

- Need to keep a dictionary in memory
  - Each entry in the dictionary is: (city, sum(price))

Q: What if "SUM" was replaced with "AVG" ?

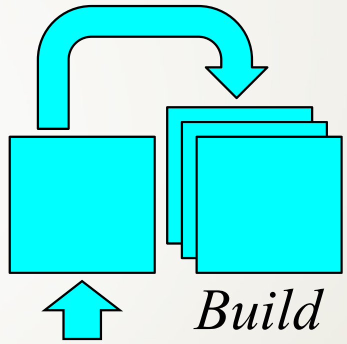


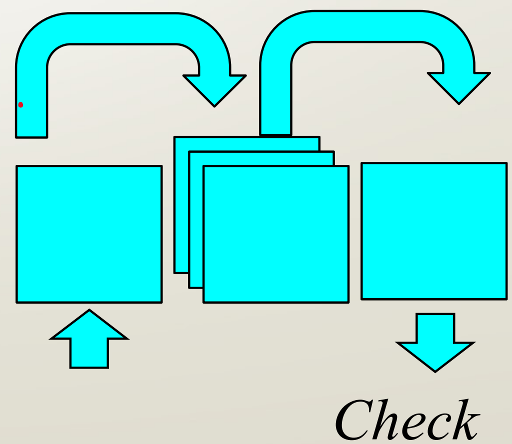
- Memory requirement: number of cities + aggregates fits in memory (plus one for input buffer)
- Cost: B(R), i.e., whatever it takes to read the blocks
- Note: Not ideally suited for the "iterator" model (pipelined production of output tuples). Why?

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.



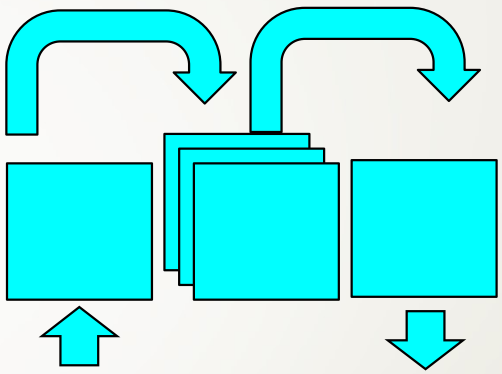


- Memory requirement: M > min(B(R), B(S))

  [more precisely M >= 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

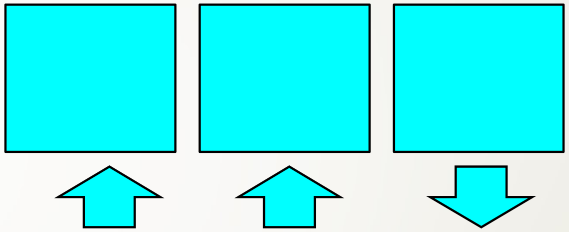
- Memory requirement:
  - M >= min(B(R), B(S)) + 2.
- Cost: B(R)+B(S)

Outline

- ✓ One-Pass Algorithms
- Nested-Loop Join
- Two-Pass Algorithms
  - Sort-Based
  - Hash-based
- Index-Based Algorithms

Nested Loop Joins

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



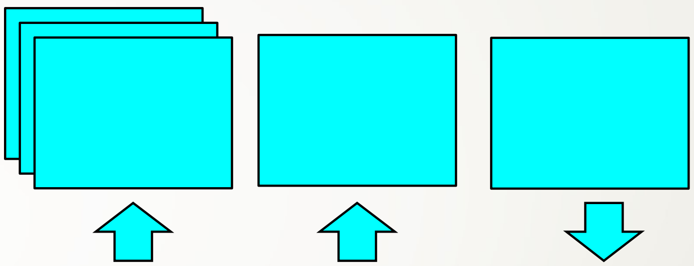
```
for each block r in R do
  for each block s in S do
     for each tuple r1 in r do
           for each tuple s1 in s do
              if r1 and s1 join then output (r,s)
```

- Memory: M >= 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, b)][(1, z), (1, c)][(1, v), (1, e)][(1, t), (1, g)]

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, b)][(1, z), (1, c)][(1, v), (1, e)][(1, t), (1, g)]

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

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

Think-Pair-Share

Why, in general, it is “slightly” better have the smaller relation is the outer relation?

A: because the smaller relation can fit in memory

B: By iterating over the smaller relation, the number of times to read the inner relation will be reduced

C: All of the above

D: None of the above

https://gerardnico.com/data/type/relation/operator/nested_loop/driving_table

Outline

- ✓ One-Pass Algorithms
- ✓ Nested-Loop Join
- Two-Pass Algorithms
  - Sort-Based
  - Hash-based
- Index-Based Algorithms

Two-Pass Algorithms Based on Sorting

Binary operations: R ∩ S, R U 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 ($x + y$ <= M-1)
  - *Why do we need all the sorted runs in memory at once?*
- Total cost: ??
  - 3B(R)+3B(S)
- Assumption: $B(R)+B(S) <= M(M-1) < M^2$

Two-Pass Algorithms Based on Sorting

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

(a) min(B(R), B(S)) <= M-2: Load smaller table to memory and load other table block by block. Cost: B(R)+B(S). This is the <u>one-pass algorithm</u>.

(b) Min(B(R), B(S)) > M-2: Load to memory M-2 blocks of S; go over every block of R; repeat. Cost: ~B(R)B(S)/M. This is the <u>nested-loop join algorithm</u>, can operate whenever M >= 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) <= 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.

Outline

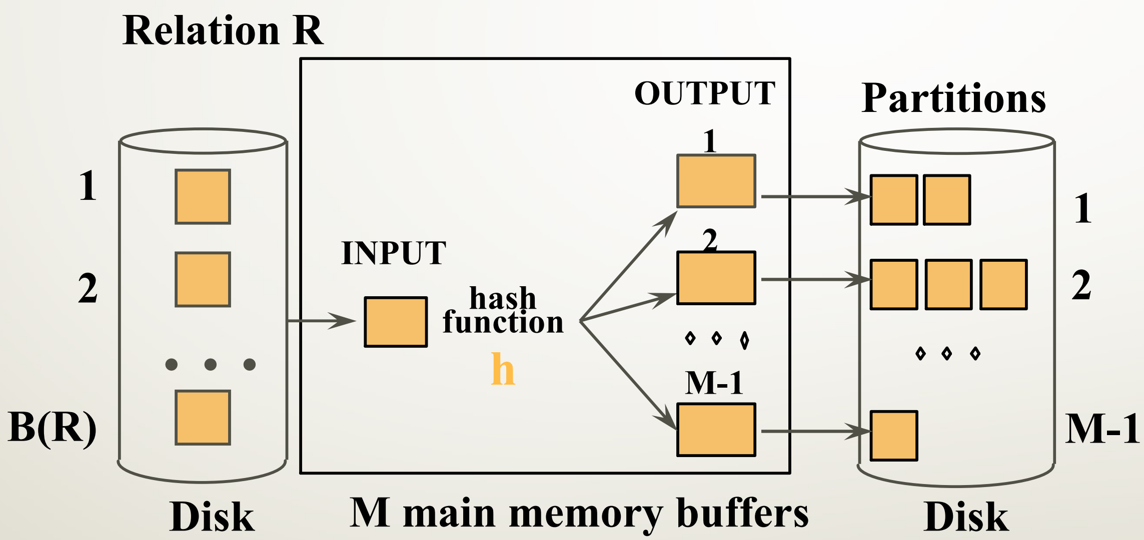
- ✓ One-Pass Algorithms
- ✓ Nested-Loop Join
- Two-Pass Algorithms
  - ✓ Sort-Based
  - Hash-based
- Index-Based Algorithms

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 $B(R)/(M-1) \le M$, 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.

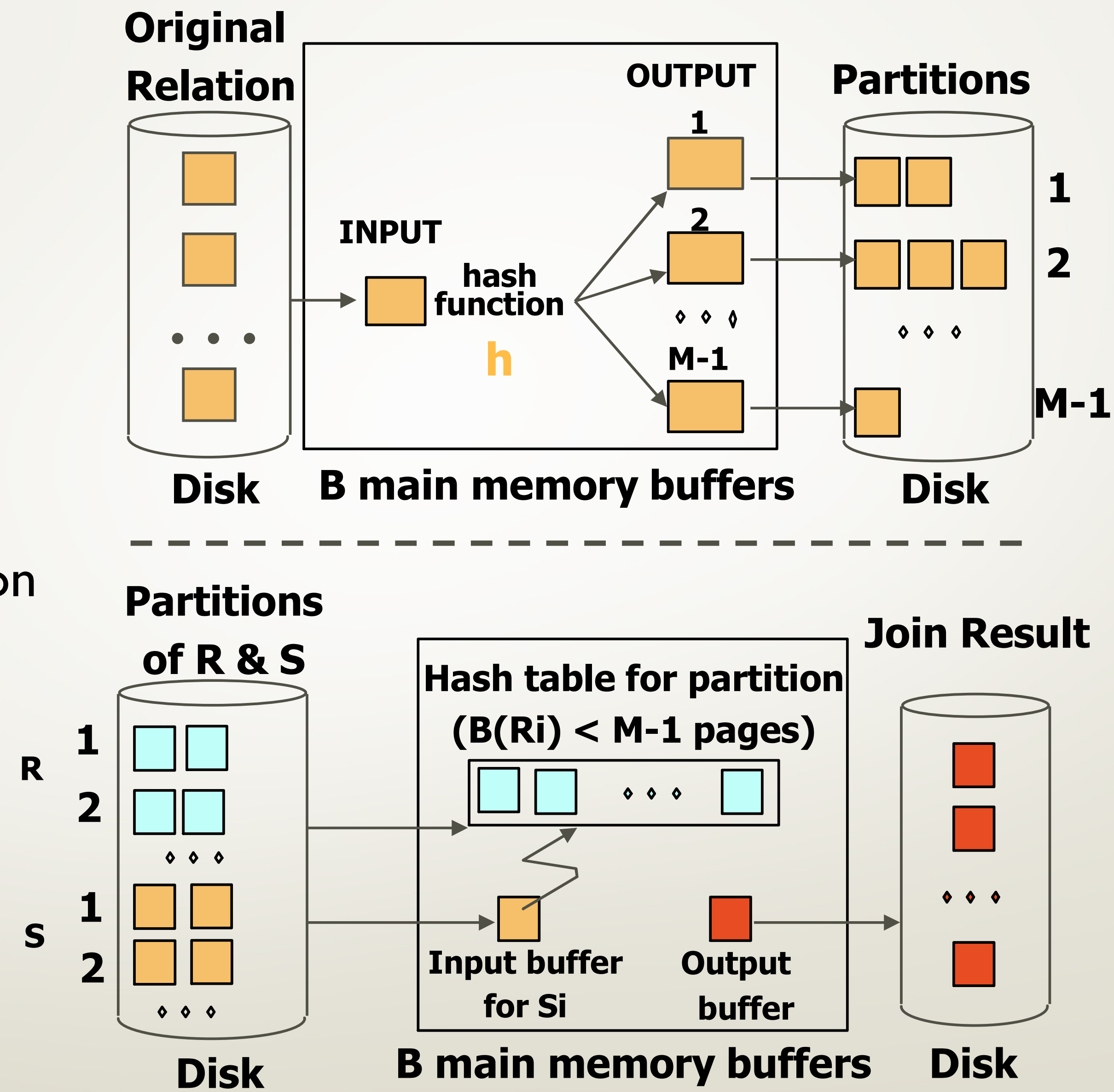
Two pass Hashing-based Join

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.
  - $\min(B(R), B(S))/(M-1) <= (M-2) \Leftrightarrow$
    $\min(B(R), B(S)) <= (M-1)(M-2) < M^2$
- Cost = $3(B(R)+B(S))$

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

- For sorting-based implementations of binary operations, size requirement was $B(R)+B(S)<=M(M-1)<M^2$. For hashing-based implementation, requirement is $\min(B(R),B(S))<=(M-1)(M-2)<M^2$.
  - 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.

Outline

- ✓ One-Pass Algorithms
- ✓ Nested-Loop Join
- Two-Pass Algorithms
  - ✓ Sort-Based
  - ✓ Hash-based
- Index-Based Algorithms

Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- Clustered index on a: 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: 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: $B(R) + B(S)$

Outline

- ✓ One-Pass Algorithms
- ✓ Nested-Loop Join
- ✓ Two-Pass Algorithms
  - ✓ Sort-Based
  - ✓ Hash-based
- ✓ Index-Based Algorithms