# CS411
# Database Systems

**11: Query Execution**

**Kazuhiro Minami**

## Goals of Query Execution

- Given a RA operator such as Join, we want to design an algorithm to implement it
- What factor do we need to consider?
- What are the following cost parameters?
  - B(R)
  - T(R)
  - V(R, a)
  - M

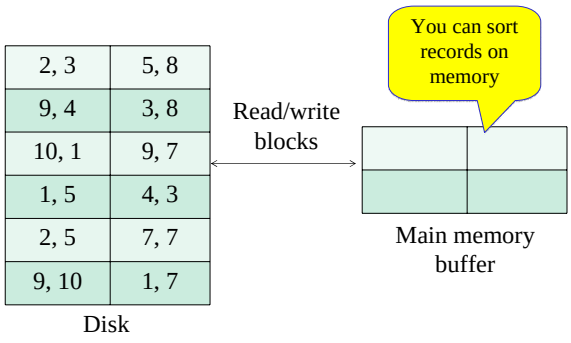


## Challenges in Query Execution

- Remember that only place we can modify data within a block is main memory
- However, we often don't have enough main memory buffer to keep all the records of a relation
- Q1. We want to sort records in a table, which is bigger than main memory. How can we do this?
- Q2. We want to join relations, which do not fit in main memory. How can do this?



## Q1: How to sort records in a large table?




- Each block or page contains 2 records

## Divide-and-conquer Approach

**MergeSort**

1. Divide the unsorted list into two sublists of about half the size
2. Sort each sublist recursively
3. Merge the two sublists back into one sorted list

Q: Can we apply this technique to our problem?

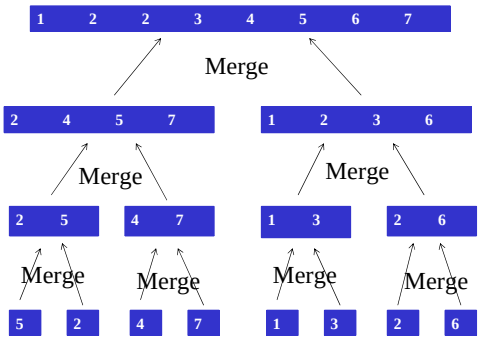


## (Human) Merge Sort Algorithm

1. Receive an unsorted list from your parent
2. If the list contains more than one,
   a) divide it into two unsorted sublist of the same size
   b) Find two children (i.e., your classmates) and pass them each of the sublists
   c) Receive your children's sorted sublists from the smallest elements of the two list
3. Return elements in the sorted list from the smallest one while coordinating with your sibling



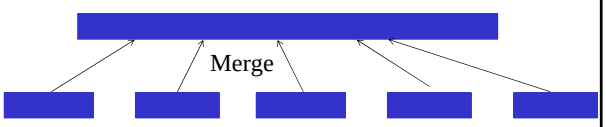
## Example Merge Process





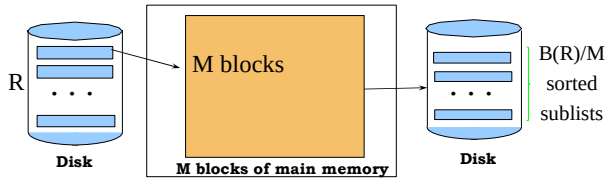

## Important differences from the standard Merge-sort

- Divide an unsorted list into sublist of size M
  - Q: why M?
- Combine multiple sorted sublists into a single sorted list
  - Q: how many sublists do we merge?



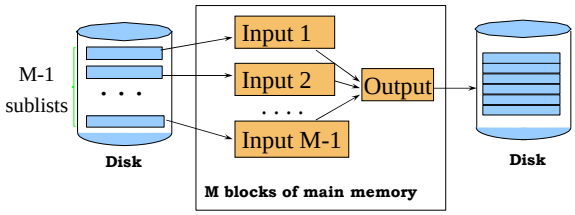
## External Merge-Sort

- Phase one: load M blocks in memory, sort
  - Result: B(R)/M sorted sublists of size M




## Phase Two

- Merge M – 1 runs into a new run
- Result: runs have now M (M – 1) blocks




## Cost of Two-Phase, Multiway Merge Sort

- Step 1: sort M-1 sublists of size M, write
  - Cost: 2B(R)
- Step 2: merge M-1 sublists, but include each tuple only once
  - Cost: B(R)
- Total cost: 3B(R), Assumption: B(R) <= $M^2$

## Update this figure: Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{M/B-1} \lceil NR/M \rceil \rceil$
- Think differently
  - Given B = 4KB, M = 64MB, R = 0.1KB
  - Pass 1: runs of length M/R = 640000
    - Have now sorted runs of 640000 records
  - Pass 2: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10,240,000,000 = $10^{10}$ records
  - Pass 3: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of $10^{14}$ records
    - Nobody has so much data !
- Can sort everything in 2 or 3 passes !

## If input relations are sorted, we can do many other operations easily

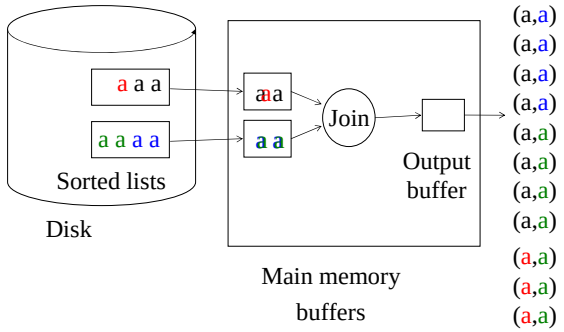
- Duplicate removal δ(R)
- Grouping and aggregation operations
- Binary operations: R ∩ S, R U S, R – S

Check total cost and assumption of each method with the textbook 15.4

## If two relations are sorted, we can perform Join (Simple Sort-based Join)

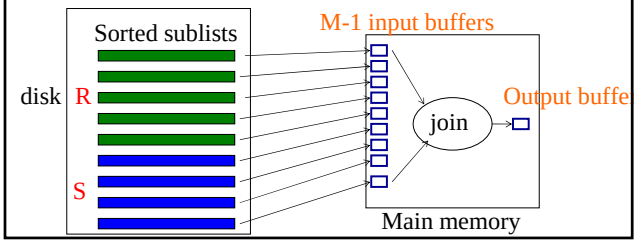
- Sort both R and S on the join attribute:
  - Cost: 4B(R)+4B(S) (because need to write to disk)
- Read both relations in sorted order, match tuples
  - Cost: B(R)+B(S)
- Difficulty: many tuples in R may match many in S
  - If at least one set of tuples fits in M, we are OK
  - Otherwise need nested loop, higher cost
- Total cost: 5B(R)+5B(S)
- Assumption: B(R) <= $M^2$, B(S) <= $M^2$, and the tuples with a common value for the join attributes fit in M.

## The Problem Regarding Too Many Tuples with the same join attributes

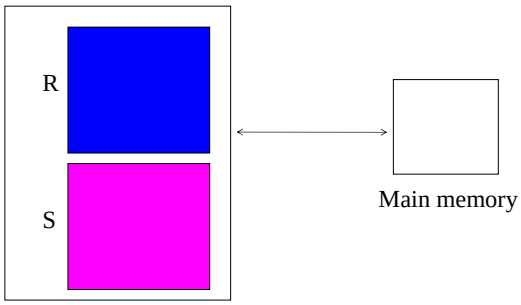



## We could do better if there are not many tuples with the same join attribute (Sort-Merge-Join)

- Idea: compute the join during the merge phase
- Total cost: 3B(R)+3B(S)
- Assumption: B(R) + B(S) <= $M^2$



## Q2: How to join two large tables without sorting them?






## Tuple-based Nested Loop Joins

- Join R ⋉ S

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

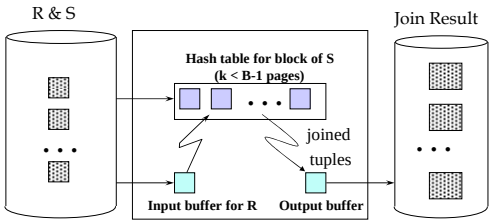
- Cost: T(R) T(S), or B(R) B(S) if R and S are clustered
- Q: How many memory buffers do we need?

## Block-based Nested Loop Joins

- Organize access to both argument relations by blocks
- Use as much main memory as we can to store tuples belonging to relation S, the relation of the outer loop

for each (M-1) blocks bs of S do
  for each block br of R do
    for each tuple s in bs do
      for each tuple r in br do
        if r and s join then output(r,s)

## Block-based Nested Loop Joins



## Block-based Nested Loop Joins

- Cost:
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-1) times, and each time need to read R: costs B(S)B(R)/(M-1)
  - Total cost: B(S) + B(S)B(R)/(M-1)
- Notice: it is better to iterate over the smaller relation first
- S ⋉ R: S=outer relation, R=inner relation

## Divide-and-conquer Approach Again

- If one of input relations fit into main memory, our job is easy
- So, we want to divide relations R and S into sub-relations $R_1,\ldots,R_n$ and $S_1,\ldots,S_n$ such that $R \bowtie S = (R_1 \bowtie S_1) \cup \ldots \cup (R_n \bowtie S_n)$
- Q: How can we divide R and S?



## Hashing-Based Algorithms

- Hash all the tuples of input relations using an appropriate hash key such that:
  - All the tuples that need to be considered together to perform an operation goes to the same bucket
- Perform the operation by working on a bucket (a pair of buckets) at a time
  - Apply a one-pass algorithm for the operation
- Reduce the size of input relations by a factor of M

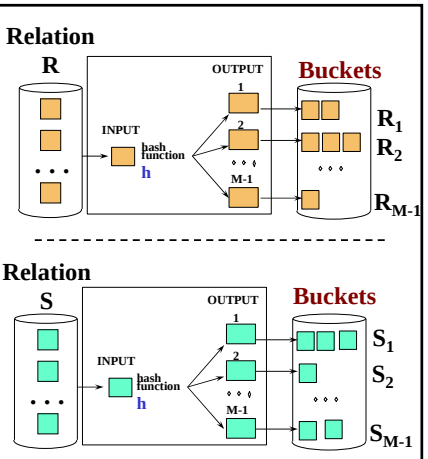
## Partitioned Hash Join

$R \bowtie S$

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

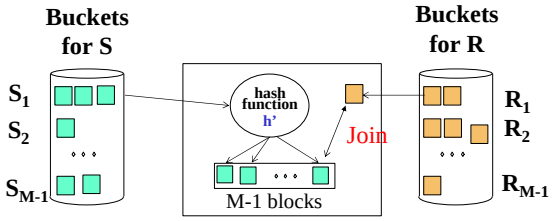
## Partitioned Hash-Join

- Partition tuples in R and S using join attributes as key hash
- Tuples in partition $R_i$ only match tuples in partition $S_i$.




## Partitioned Hash-Join: Second Pass

- Read in a partition of $R_i$, hash it using another hash function h'
- Scan matching partition of S, search for matches.




Main Memory Buffers

## Partitioned Hash Join

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

## Sort-based vs. Hash-based Algorithms

- Hash-based algorithms for binary operations have a size requirement only on the smaller of two input relations
- Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later
- Hash-based algorithm depends on the buckets being of equal size, which may not be true if the number of different hash keys is small

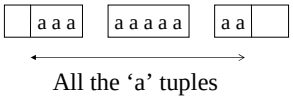
## Two-Pass Algorithms Based on Index

## Index-based Algorithms

- The existence of an index on one ore more attributes of a relations makes available some algorithm that would not be feasible without the index
- Useful for selection operations
- Also, algorithms for join and other binary operations use indexes to good advantage

## Clustering indexes

- In a clustered index all tuples with the same value of the key are clustered on as few blocks as possible




All the 'a' tuples

Q: how many blocks do we need to read?

## Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index on a: cost T(R)/V(R,a)

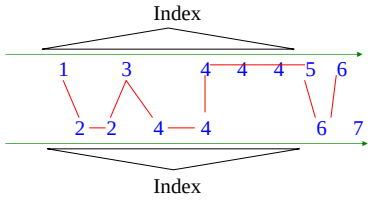
We here ignore the cost of reading index blocks

## Index Based Join

- R ⋉ 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 is clustered: B(R) + T(R)B(S)/V(S,a)
  - If index is unclustered: B(R) + T(R)T(S)/V(S,a)

## Index Based Join

- Assume both R and S have a sorted index (B-tree) on the join attribute
- Then perform a merge join (called zig-zag join)
- Cost: B(R) + B(S)




## Summary

- One-pass algorithms (Read the textbook 15.2)
  - Read the data only once from disk
  - Usually, require at least one of the input relations fit in main memory

✓Nested-Loop Join algorithms
  - Read one relation only once, while the other will be read repeatedly from disk

✓Two-pass algorithms
  - First pass: read data from disk, process it, write it to the disk
  - Second pass: read the data for further processing