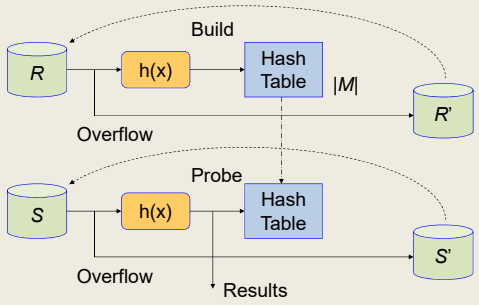
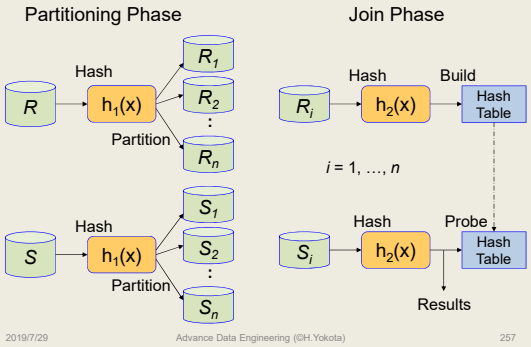
Illustration of Simple Hash Join



Parallelize Simple Hash Join

- Where can we parallelize simple hash join?
- Loop ?
  - There are dependencies in iteration
    - (i-1)-th results are used for i-th iteration
- A step of the iteration ?
  - That IS Join
    - The Original Problem
- It is hard to parallelize simple hash join

Illustration of GRACE Hash Join



Parallelize GRACE Hash Join

- Parallelize Bucket Decomposition in Phase 1
  - Each relation is partitioned in advance
    - The average number of tuples in each disk {R}/N and {S}/N
    - Selection operations are also executed in advance in parallel
  - Virtually divide connected disk into Read Disk and Write Disk
  - Assign each bucket to each processor
    - Send each tuple by its hash value
- Parallelize Join Operation in Phase 2
  - There is no communication within Phase 2

Pseudo Code for Phase1

- Each PE has two threads:

Thread 1:

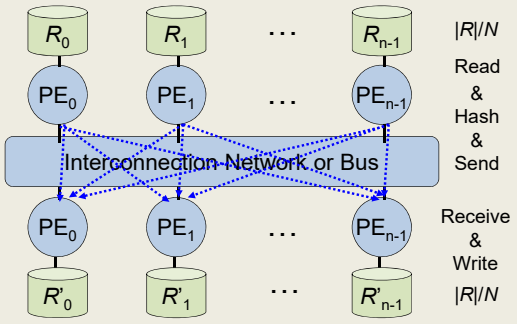
```
for (j = 1; j ≤ {R/N}; j++) {
  read j-th tuple t and attribute value v in t;
  x = h0(v);    /* e.g. h(v) = mod(v, N) */
  send t to PE_x
}
```

Thread 2

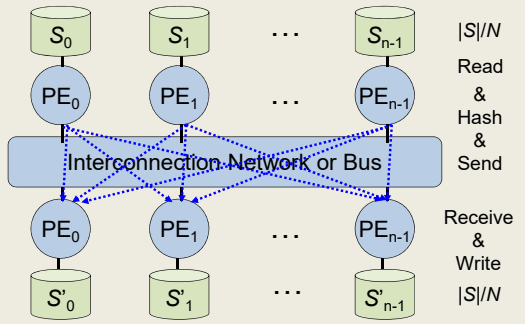
```
for (;;) {
  receive t and attribute value v in t;
  y = h1(v);
  write t into a file for y
}
```

- It should be combined with the phase switch of all-to-all communication

Parallel GRACE Hash Join (1)

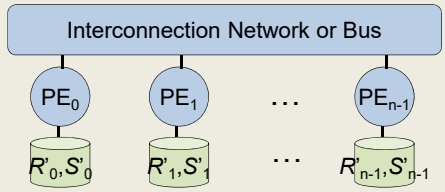


Parallel GRACE Hash Join (2)



Parallel GRACE Hash Join (3)

Phase 2



Read & Local Join

(|*R*| + |*S*|) / *N*

Cost of Parallel GRACE Hash Join

- Phase 1
  - (|*R*| + |*S*|) / *N* read and (|*R*| + |*S*|) / *N* write in each PE
  - With all-to-all communication cost: α
- Phase 2
  - (|*R*| + |*S*|) / *N* read in each PE
  - No communication
- Total I/O in each PE
  - 3 x (|*R*| + |*S*|) / *N*
  - It means (1/*N* + α), if there is no skew

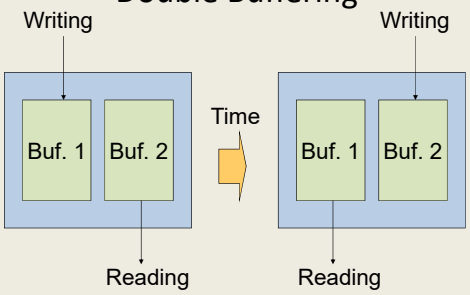
Estimate α (1/3)

- Bandwidth of each connection of network: 10MB/s
- Network setup time for each connection: 50 μs
- |R| and |S|: 64MB each
- The number of processors (N): 8
- Consider the cost for communication, assuming each processing element has enough large buffer space to keep |R|/N or |S|/N
- Also consider the cost when each processing element has memory for two pages (8KB)

Estimate α (2/3)

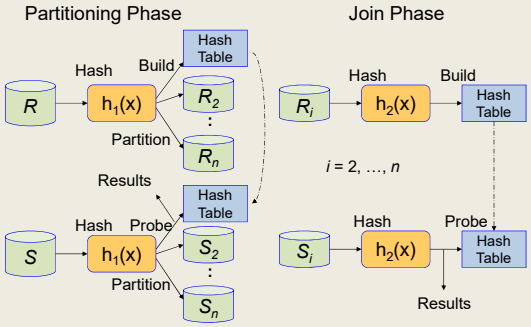
Estimate α (3/3)

Double Buffering



- It enables simultaneous reading and writing

Illustration of Hybrid Hash Join



Parallelize Hybrid Hash Join

- Consider how to parallelize the Hybrid hash join
- Pseudo Code

Thread 2

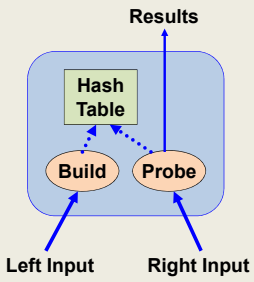
```
for (;;) {
  receive t and attribute value v in t;
  y = h1(v);
  if y is 0 then build (or probe) a hash table
  else write t into a file for y
}
```

Multiple Joins

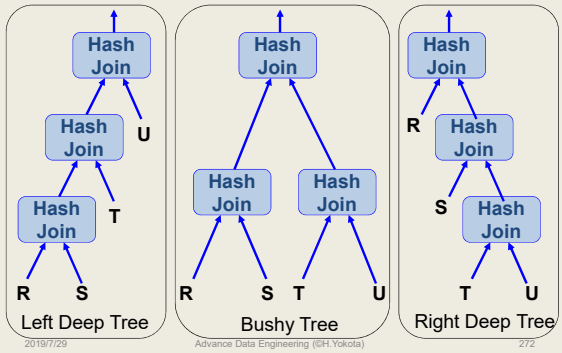
- There tend to be a number of join operations in a query
  - The query construct a query tree
  - The configuration of the tree deeply influence the performance of query processing
    - especially under parallel environment
- Parallel executions in a query tree
  - Independent executions
  - Pipeline executions
- Here, we assume hash join algorithm

Notation of a Hash Join Node

- Let left input is for **Build** and right for **Probe**
  - Hash Join cannot start Probe until the Build Phase is finished



Query Tree Configurations



Parallel Multiple Join

- Parallel execution of multiple join is depend on the structure of the query tree
  - Left Deep Tree:
    - Sequential
  - Right Deep Tree:
    - Pipeline Execution (Parallel Build)
  - Bushy Tree:
    - Child node can be executed in parallel
    - Pipeline Execution can also be done

Parallel Aggregation

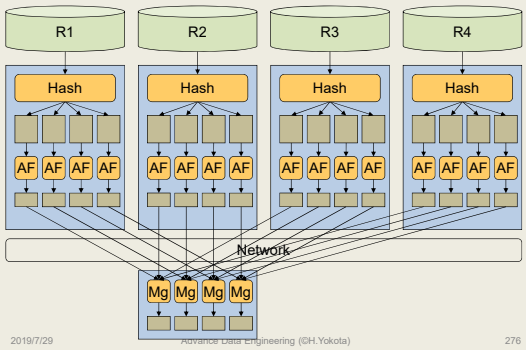
- Based on sequential aggregation algorithm using hash
  - Hashing for a Group-By operation
  - Applying aggregation functions to each group
    - Count, Sum, Average, Max, Min
- Three algorithms for parallel execution
  - Centralized Two Phase Algorithm (C-2P)
  - Two-Phase Algorithm (2P)
  - Repartitioning Algorithm (Rep)

Centralized Two Phase Algorithm (C-2P)

1. Read tuples from each local disk, apply a hash function, and execute aggregate function for hash buckets in each PE
2. Send the results of aggregation to **a node** to merge them

- The centralized node will be a bottleneck when the number of PE increases

Centralized Two Phase Algorithm (C-2P)



Pseudo Code for C-2P

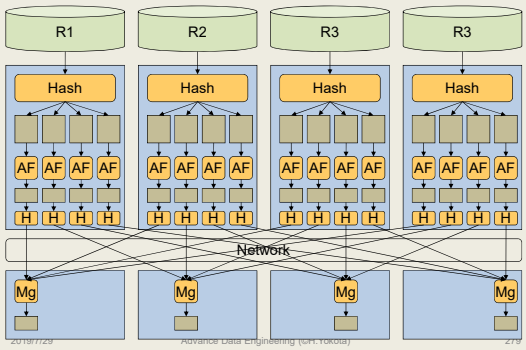
```
In each PE (indicated by x)
 for(i=1; i<={Rx}; i++) {
   get i-th tuple t from Rx;
   derive target attribute value v in t;
   y = h(v);
   keep t in buffer[y] }
  for(i=1; i <= group#; i++) {
   apply aggregate functions for buffer[i]
   send the result to a PE}
In the PE(0)
  merge the results for each group
```


Two-Phase Algorithm (2P)

1. Read tuples from each local disk, apply a hash function, and execute aggregate function for hash buckets in each PE
2. Send the results of aggregation to **nodes** corresponded with hash partition in **parallel**

- The merge operations are also executed in parallel

Two-Phase Algorithm (2P)



Pseudo Code for 2P

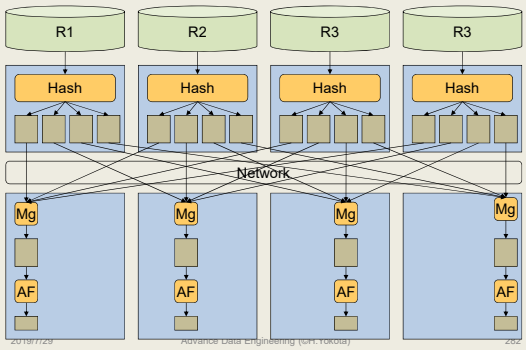
```
In each PE (indicated by x)
  for(i=1; i<={Rx}; i++) {
    get i-th tuple t from Rx;
    derive target attribute value v in t;
    y = h(v);
    keep t in buffer[y] }
  for(i=1; i <= group#; i++) {
    apply aggregate functions for buffer[i]
    z = h2(i);
    send the result to PE(z)}
In each PE
  merge the results for corresponding groups
```


Repartitioning Algorithm (Rep)

1. Read tuples from each local disk, and apply a hash function
2. Send the result to nodes depend on the hash result
3. A node receive the result execute aggregate function in **parallel**

- The number of invocations of aggregate functions can be reduced

Repartitioning Algorithm (Rep)



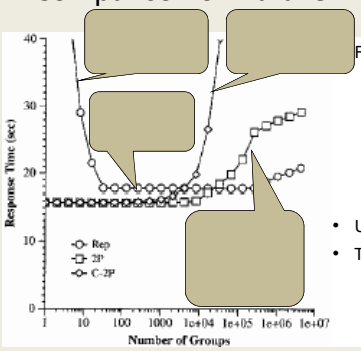
Pseudo Code for Rep (1)

```
In each PE (indicated by x)
for(i=1; i<={Rx}; i++) {
   get i-th tuple t from Rx;
   derive target attribute value v in t;
   y = h2(v);
   send t to PE(y) }
```


Pseudo Code for Rep (2)

```
In each PE
  merge tuples;
  n = count tuples;
  for(i = 1, i < n ; i++) {
    get i-th tuple from buffer and derive target attribute
    value v in t;
    z = h(v);
    store t in buffer[z];
   }
  for( j= 1; j < number of buffer in the PE; j++)  {
    apply aggregate functions for buffer[j]
  }
```

Comparison on Parallel Aggregation



From:

- Adaptive Parallel Aggregation Algorithm
- A. Shatdal and J.F. Naufhton,
- Proc. of Int'l Conf. SIGMOD '95, pp.104--114, (fig1)

- Using 32 PEs
- The graph indicates:
  - When there are small groups, 2p is better
  - Otherwise, Rep is better

2019/7/29 Advance Data Engineering (©H.Yokota) 285