CSE544
Data Management
Lectures 8
Query Execution – Part 2
Announcements

• This Friday: Kyle’s office hour moved
  – 1:30-2:20, Gates 274
• Project proposals due on Monday, 2/5
• HW2 due on Wednesday 2/7
• R3 pushed to Wednesday, 2/14
Lifecycle of a Query

1. Parse & Rewrite Query
2. Select Logical Plan
3. Select Physical Plan
4. Query Execution

Disk

SQL query

Logical plan

Physical plan

Query optimization
Main Memory Operators
1. Nested Loop Join

Logical operator:
Supplier \bowtie_{sno=sno} Supply

```
for x in Supplier do
    for y in Supply do
        if x.sno = y.sno
            then output(x,y)
```

If |R|=|S|=n, what is the runtime?

O(n^2)
1. Index Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply} \]

```
for x in Supplier do
  for y in SupplyIndex(x.sno) do
    output(x,y)
```

If \(|R|=|S|=n\), what is the runtime?

O(n)

When data is on disk then big difference between clustered and unclustered
Clustered v.s. Unclustered

Logical operator:
$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

for x in Supplier do
  for y in SupplyIndex(x.sno) do
    output(x,y)

Rule of thumb:
Random reading 1-2% of Supply ≈ sequential scan entire file

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)
2. Hash Join

Logical operator: $\bowtie_{sno=sno}$

```latex
for x in Supplier do
    insert(x.sno, x)
for y in Supply do
    x = find(y.sno);
    output(x, y);
```

If $|R|=|S|=n$, what is the runtime?

$O(n)$
2. Hash Join

Logical operator:

\[ \text{Supply} \bowtie_{\text{sno}=\text{sno}} \text{Supplier} \]

For y in Supply do

insert(y.sno, y)

For x in Supplier do

for y in find(x.sno) do

output(x, y);

If \(|R| = |S| = n\), what is the runtime?

O(n)

But can be O(n^2) why?
3. Merge Join

Logical operator:

Supplier $\bowtie_{sno=sno}$ Supply

Sort(Supplier); Sort(Supply);

x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
      x.sno < y.sno: x = x.next()
      x.sno = y.sno: output(x,y); y = y.next()
      x.sno > y.sno: y = y.next()

If $|R| = |S| = n$, what is the runtime?

$O(n \log(n))$
Brief Review:
Hash Tables, Sorting
Hash Tables

• Array: map indices to memory locations
  – A[0], A[1], A[2], … sequential in memory

• How to map texts to memory locations?
  – A[“alice”], A[“bob”], A[“carl”]…???

• Hash function: maps strings to indices
Hash Tables

Separate chaining:

A (naïve) hash function:

\[ h(\text{"abc"}) = (\text{'a'} + \text{'b'} + \text{'c'}) \mod 10 \]
Separate chaining:

**Hash Tables**

A (naïve) hash function:

\[
h(\text{“abc”}) = (\text{‘a’+’b’+’c’}) \mod 10
\]

E.g. \( h(\text{“ker”}) = (\text{‘k’+’e’+’r’}) \mod 10 \)
\[
= (107+101+114) \mod 10
\]
\[
= 2
\]
Separate chaining:

Hash Tables

A (naïve) hash function:

\[
h(\text{"abc"}) = (a + b + c) \mod 10
\]

E.g. \( h(\text{"ker"}) = (k + e + r) \mod 10 \)
\[= (107 + 101 + 114) \mod 10 \]
\[= 2 \]
Separate chaining:

**Hash Tables**

A (naïve) hash function:

\[ h(“abc”) = (‘a’+’b’+’c’) \mod 10 \]

E.g. \[ h(“ker”) = (‘k’+’e’+’r’) \mod 10 \]
\[ = (107+101+114) \mod 10 \]
\[ = 2 \]

```
find(“yu”) = ??
insert(“alice”) = ??
```
Hash Table Takeaways

• Use good hash function, never your own. E.g. https://15445.courses.cs.cmu.edu/fall2023/slides/07-hashtables.pdf
  – Low probability of collision

• The vector needs to be pre-allocated:
  – Too big: waste space
  – To small: long chains
  – Extensible hash table: doubles the vector

• Skewed data leads to collisions: $\mathcal{O}(1) \rightarrow \mathcal{O}(N)$
Sorting

• Given an array, sort it in increasingly
Sorting

• Given an array, sort it in increasingly

\[
\begin{array}{cccccccccc}
74 & 7 & 45 & 99 & 2 & 90 & 19 & 87 & 61 & 82 \\
\hline
2 & 7 & 19 & 45 & 61 & 74 & 82 & 87 & 90 & 99 \\
\end{array}
\]
Sorting

- Given an array, sort it in increasingly

<table>
<thead>
<tr>
<th>74</th>
<th>7</th>
<th>45</th>
<th>99</th>
<th>2</th>
<th>90</th>
<th>19</th>
<th>87</th>
<th>61</th>
<th>82</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>19</td>
<td>45</td>
<td>61</td>
<td>74</td>
<td>82</td>
<td>87</td>
<td>90</td>
<td>99</td>
</tr>
</tbody>
</table>

- Simple algorithms: $O(N^2)$
- Quicksort: $O(N \log N)$
- Mergesort: $O(N \log N)$
Merge Sort: Overview

• A run is a subarray that is sorted

• Repeat:
  – Merge two runs
  – Store output in some array T
  – Switch A,T

• Stop when A is one single run
Merging two Arrays

```
Merge(A, B)  // A, B are sorted
```

Assume $\infty$ at the end
Merging two Arrays

Merge(A, B)  // A, B are sorted
i=0;  j=0;  k=0;
while i<n or j<n
    case:

Assume ∞ at the end
Merging two Arrays

Merge(A, B) // A, B are sorted
  i=0;  j=0;  k=0;
  while i<n or j<n
    case:

Assume ∞ at the end
Merging two Arrays

Merge(A, B)       // A, B are sorted
    i=0; j=0; k=0;
    while i<n or j<n
        case:

Time = O(|A|+|B|)

Assume ∞ at the end
Merge-Sort

| 74 | 7 | 45 | 99 | 2 | 90 | 19 | 87 | 61 | 82 |
Merge-Sort

Runs

74 7 45 99 2 90 19 87 61 82
Merge-Sort

Runs

74  7  45  99  2  90  19  87  61  82

7  74  45  99  2  90  19  87  61  82
Merge-Sort

Runs

74 7 45 99 2 90 19 87 61 82

7 74 45 99 2 90 19 87 61 82

7 45 74 99 2 19 87 90 61 82
Merge-Sort

Runs

74 7 45 99 2 90 19 87 61 82
7 74 45 99 2 90 19 87 61 82
7 45 74 99 2 19 87 90 61 82
2 7 19 45 74 87 90 99 61 82
Merge-Sort

Runs

74 7 45 99 2 90 19 87 61 82

7 74 45 99 2 90 19 87 61 82

7 45 74 99 2 19 87 90 61 82

2 7 19 45 74 87 90 99 61 82

2 7 19 45 61 74 82 87 90 99
Discussion

• Main memory algorithms use quicksort
  – $O(N \log N)$ expected runtime
  – Problem: random access, not good for disk
Discussion

• Main memory algorithms use quicksort
  – $O(N \log N)$ expected runtime
  – Problem: random access, not good for disk

• Merge-sort:
  – $O(N \log N)$ runtime
Discussion

• Main memory algorithms use quicksort
  – O(N log N) expected runtime
  – Problem: random access, not good for disk

• Merge-sort:
  – O(N log N) runtime
  – log N sequential reads will improve
Discussion

• Main memory algorithms use quicksort
  – $O(N \log N)$ expected runtime
  – Problem: random access, not good for disk

• Merge-sort:
  – $O(N \log N)$ runtime
  – $\log N$ sequential reads will improve
  – But: $O(N)$ memory overhead; Not used for main memory, except for python’s TimSort
External Memory Operators
Setup

Main cost = number of disk I/O’s
Always ignore the final write
Setup

Main cost = number of disk I/O’s
Always ignore the final write
• $B(R) =$ number of bloks used to store $R$
• $T(R) =$ number of records in $R$
Setup

Main cost = number of disk I/O’s
Always ignore the final write
• B(R) = number of blocks used to store R
• T(R) = number of records in R
• Sequential read of R: cost = B(R)
• Random read of R: cost = T(R)
Setup

Main cost = number of disk I/O’s
Always ignore the final write
• \( B(R) = \text{number of bloks used to store } R \)
• \( T(R) = \text{number of records in } R \)
• Sequential read of \( R \): \( \text{cost} = B(R) \)
• Random read of \( R \): \( \text{cost} = T(R) \)

\[ T(R) \gg B(R) \]
Sequential/Random Access

Sequential
Sequential/Random Access

Sequential Read
Sequential/Random Access

Sequential Read

Already in buffer pool
Sequential/Random Access

Sequential
Sequential/Random Access

Sequential: $B(R) = 2$ reads
Sequential/Random Access

Sequential: $B(R) = 2$ reads

Random

46
Sequential/Random Access

Sequential: $B(R) = 2$ reads

Random
Sequential/Random Access

Sequential: $B(R) = 2$ reads

Random: $T(R) = 10$ reads
External Memory Operators

- Block Nested Loop Join
- Merge Join
- Partitioned Hash Join
Block Nested Loop Join
Nested Loop Joins

\[ R \bowtie S \]

for x in R do
  for y in S do
    if join(x, y): output(x, y)

- Naïve nested loop join: \( B(R) + T(R) \times B(S) \)
- If \( T(R) = 1,000,000 \) then this is terrible…
Block Nested Loop Join

Idea: better use of the available memory

• $M = \# \text{ of blocks that fit in main memory}$
Block Nested Loop Join

Group of \((M-2)\) pages of \(R\), called a “block”

\[
\begin{align*}
\text{for each (M-2) pages PR of R do} \\
\phantom{\text{for each (M-2) pages PR of R do}} \text{for each page PS of S do} \\
\phantom{\text{for each (M-2) pages PR of R do}} \phantom{\text{for each page PS of S do}} \text{Main memory join: PR }\bowtie\text{ PS}
\end{align*}
\]
Block Nested Loop Join

Group of \((M-2)\) pages of \(R\), called a “block”

\[
\begin{align*}
&\text{for each (M-2) pages PR of } R \text{ do} \\
&\quad \text{for each page PS of } S \text{ do} \\
&\quad \text{Main memory join: } PR \bowtie PS
\end{align*}
\]

Why not use \(M-1\) pages?
Block Nested Loop Join

Group of \((M-2)\) pages of \(R\), called a “block”

\[
\text{for each (M-2) pages PR of } R \text{ do for each page PS of } S \text{ do }
\]

Main memory join: \(PR \bowtie PS\)

use the remaining page for output
Block Nested Loop Join

Group of \((M-2)\) pages of \(R\), called a “block”

for each \((M-2)\) pages \(PR\) of \(R\) do

for each page \(PS\) of \(S\) do

Main memory join: \(PR \bowtie PS\)

use the remaining page for output

\[ B(R) + \frac{B(R)B(S)}{(M-2)} \text{ disk I/Os. WHY?} \]
Block Nested Loop Join

M=8

R ⋈ S

1 block = 2 records
Block Nested Loop Join

R

S

M=8

1 block = 2 records

R \bowtie S
Block Nested Loop Join

R

M=8

S

PR

PS

OUT

R \bowtie S

1 block = 2 records
Block Nested Loop Join

M=8

R

S

PR
PS
OUT

1 block = 2 records

R \bowtie S
Block Nested Loop Join

M=8

1 block = 2 records

R \bowtie S
Block Nested Loop Join

M=8

R

S

1 block = 2 records

R \bowtie S
Block Nested Loop Join

M=8

Need to compute PR \Join PS. Will show a simple nested loop. A hash-join is likely better.
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

R \bowtie S
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

R \bowtie S
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

R \bowtie S

CSE 544 - Winter 2024
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

R \bowtie S
Block Nested Loop Join

M=8

R

S

PR

PS

OUT

R \bowtie S

1 block = 2 records
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

R \bowtie S

1 block = 2 records

CSE 544 - Winter 2024
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

R \bowtie S

1 block = 2 records
Block Nested Loop Join

\[ R \bowtie S \]

1 block = 2 records

M=8

\[
\begin{align*}
R & \quad PR \\
S & \quad PS \\
OUT & \quad \text{OUT}
\end{align*}
\]
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

R \bowtie S

1 block = 2 records

CSE 544 - Winter 2024
Block Nested Loop Join

\[ R \bowtie S \]

\( M = 8 \)

1 block = 2 records

CSE 544 - Winter 2024
Block Nested Loop Join

M=8

1 block = 2 records

R \bowtie S
Block Nested Loop Join

M=8

PR

PS

OUT

R ⋈ S

1 block = 2 records

Done with 1st record of PS

R

S

CSE 544 - Winter 2024
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

Now 2\textsuperscript{nd} record

R \bowtie S

CSE 544 - Winter 2024
Block Nested Loop Join

M=8

1 block = 2 records

R

S

R \bowtie S
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

Done with this block (for now)

R \bowtie S

1 block = 2 records
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

1 block = 2 records

R \bowtie S

Read next block

R

S

CSE 544 - Winter 2024
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

R \bowtie S

...and next

1 block = 2 records

CSE 544 - Winter 2024
Block Nested Loop Join

M = 8

R

S

PR

PS

OUT

1 block = 2 records

Done first pass on S

R \bowtie S
Block Nested Loop Join

M = 8

More efficient to organize PR as a hash table, and use for each block of S

Done first pass on S

1 block = 2 records

R \Join S
Block Nested Loop Join

M = 8

PR
PS
OUT

1 block = 2 records

R \Join S

Done first pass on S

Done with this block of R

R

S

CSE 544 - Winter 2024
Block Nested Loop Join

M=8

1 block = 2 records

Read next block of R

Restart S

R \bowtie S
Block Nested Loop Join

R

S

M=8

PR

PS

OUT

R \bowtie S

1 block = 2 records

End
Block Nested Loop Join

R

\[ M = 8 \]

PR

PS

OUT

S

End

R \bowtie S

End

1 block = 2 records

CSE 544 - Winter 2024
Block Nested Loop Join

M = 8

Read R once: \( B(R) \)
Read S \( B(R)/(M-2) \) times
Total cost: \( B(R) + B(R)B(S)/(M-2) \)
We ignore the final write
Merge Sort, Merge Join
Merge-Sort

Merge-sort reads/writes sequentially

- Run lengths: 2, 4, 8, 16, ...
- Need $\log(N)$ sequential reads and writes

$$\text{Cost} = 2 \log(N) \cdot B(R)$$
Multi-Way Merge-Sort

N-Way Merge Sort: use entire memory M

• Merge M-1 runs at once; run lengths: 
  \((M-1), (M-1)^2, (M-2)^3, \ldots\)

• Need \(\log_{M-1} N\) reads/writes; \(\log_{M-1} N \approx 2\)

\[\text{Cost} = 3 \ B(R)\]
Merging n Arrays

\[ \text{Merge}(A_1, A_2, \ldots, A_n) \quad // A_1, \ldots, A_n \text{ are sorted} \]
Merging n Arrays

\[
\text{Merge}(A_1, A_2, \ldots, A_n) \quad // A_1, \ldots, A_n \text{ are sorted}
\]
\[
i_1 = 0; \quad \ldots, \quad i_n = 0; \quad j = 0
\]
\[
\text{while } i_1 \leq N \text{ or } \ldots \text{ or } i_1 \leq N
\]
\[
\text{let } A_k[i_k] = \min(A_1[i_1], \ldots, A_n[i_n])
\]
Merging n Arrays

```
Merge(A_1, A_2, ..., A_n)  // A_1, ..., A_n are sorted
i_1 = 0;  ..., i_n = 0;  j=0
while i_1 ≤ N or ... or i_1 ≤ N
    let A_k[i_k] = min(A_1[i_1], ..., A_n[i_n])
    T[k++] = A_k[i_k++]
```
Merging n Arrays

Merge(A_1, A_2, ..., A_n)   // A_1, ..., A_n are sorted
    i_1 = 0;  ..., i_n = 0;  j = 0
    while i_1 \leq N or ... or i_1 \leq N
        let A_k[i_k] = \min(A_1[i_1], ..., A_n[i_n])
        T[k++] = A_k[i_k++]

Time = O(|A_1| + |A_2| + ... |A_n|)
Merging n Arrays

Merge(A_1, A_2, ..., A_n)  // A_1, ..., A_n are sorted
  i_1 =0;  ..., i_n = 0;  j=0
  while i_1 ≤ N or ... or i_1 ≤ N
    let A_k[i_k] = min(A_1[i_1], ..., A_n[i_n])
    T[k++] = A_k[i_k++]

Time = O(|A_1| + |A_2| + ... |A_n|)

Additional log n factor to find min using priority queue
Merging n Arrays

Merge($A_1, A_2, \ldots, A_n$)  // $A_1, \ldots, A_n$ are sorted
    $i_1 = 0; \ldots, i_n = 0; j = 0$
    while $i_1 \leq N$ or $\ldots$ or $i_1 \leq N$
        let $A_k[i_k] = \min(A_1[i_1], \ldots, A_n[i_n])$
        $T[k++] = A_k[i_k++]$

Time = $O(|A_1| + |A_2| + \ldots |A_n|)$

Additional log n factor to find min using priority queue

Important: Need to read only one element from each array
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort

Sort using in place algorithm (no extra space)
Merge-Sort: Step 1

• Phase 1: load M-1 blocks in memory, sort

Sort using in place algorithm (no extra space)
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort

Sort using in place algorithm (no extra space)
Merge-Sort: Step 1

- Phase 1: load $M-1$ blocks in memory, sort
Merge-Sort: Step 1

- Phase 1: load M-1 blocks in memory, sort

![Diagram showing the process of merging sorted blocks from disk to main memory, then to disk again.](image-url)
Merge-Sort: Step 1

• Phase 1: load M-1 blocks in memory, sort
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …

Diagram:

- Main memory
  - run 1
  - run 2
  - run $M-1$

- Disk
  - Out

Legend:

- Read only one block from each run
Merge-Sort: Steps 2, 3, …

- Merge \( M - 1 \) runs into a new run
- New runs of length \((M - 1)^2, (M - 1)^3, \ldots\)
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …

When done with these records, read another block
Merge-Sort: Steps 2, 3, …

• Merge $M - 1$ runs into a new run
• New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, ...

- Merge M – 1 runs into a new run
- New runs of length $(M – 1)^2$, $(M – 1)^3$, ...

![Diagram of Merge-Sort process]

Main memory

Disk

run 1

run 2

run M-1

Out
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run.
- New runs of length $(M - 1)^2$, $(M - 1)^3$, …
Merge-Sort: Steps 2, 3, …

- Merge $M - 1$ runs into a new run
- New runs of length $(M - 1)^2, (M - 1)^3, …$
Merge-Sort: Discussion

Size of the runs increases fast
• Example: B=10001

Each block is 16Kb
Merge-Sort: Discussion

Size of the runs increases fast

- Example: \( B = 10001 \)
- Step 1: run length 10000 = 160Mb

Each block is 16Kb
Merge-Sort: Discussion

Size of the runs increases fast
- Example: B=10001
- Step 1: run length 10000 = 160Mb
- Step 2: run length 100000000 = 1.6Tb
- ...

Each block is 16Kb
Merge-Sort: Discussion

Size of the runs increases fast

- Example: \( B = 10001 \)
- Step 1: run length \( 10000 = 160\text{Mb} \)
- Step 2: run length \( 100000000 = 1.6\text{Tb} \)
- …

Usually, 2 steps suffice

\[ \text{Cost} = 3B(R) \]
Finally: Merge-Join

\[ R \bowtie S \]
Finally: Merge-Join

$R \bowtie S$

- Create runs of $R \Rightarrow B(R)/(M-1)$ runs
- Create runs of $S \Rightarrow B(S)/(M-1)$ runs
Finally: Merge-Join

\[ R \bowtie S \]

- Create runs of \( R \)  \( \rightarrow \) \( B(R)/(M-1) \) runs
- Create runs of \( S \)  \( \rightarrow \) \( B(S)/(M-1) \) runs
- Use N-way merge to compute the join
Finally: Merge-Join

\[
R \bowtie S
\]

- Create runs of \( R \rightarrow B(R)/(M-1) \) runs
- Create runs of \( S \rightarrow B(S)/(M-1) \) runs
- Use N-way merge to compute the join

- Need \( B(R)+B(S) \leq (M-1)^2 \) why?
  Cost = \( 3(B(R)+B(S)) \)
Partitioned Hash-Join
Hash-Partitioned Join

• The outer relation $R$ needs to fit in main memory
• The inner relation $S$ doesn’t need to fit

```plaintext
for x in R do
    insert(x.key, x)
for y in S do
    x = find(y.key);
    output(x, y);
```
Partitioned Hash-Join

• $R \bowtie S$, both bigger than main memory
Partitioned Hash-Join

• $R \bowtie S$, both bigger than main memory
• Step 1:
  – Hash partition both $R$ and $S$
  – Store buckets on disk
Partitioned Hash-Join

• $R \bowtie S$, both bigger than main memory

• Step 1:
  – Hash partition both $R$ and $S$
  – Store buckets on disk

• Step 2:
  – Read one $R$-bucket in main memory
  – Hash-Join with corresponding $S$-bucket
  – Repeat for all buckets
Step 1: Hash-partition

- Partition R into buckets, on disk

![Diagram showing the partitioning of a relation R into buckets and its interaction with main memory buffers and disk.]
Step 1: Hash-partition

- Partition R into buckets, on disk

![Diagram showing hash partition process]

Relation R

INPUT

hash function

OUTPUT

M main memory buffers

Disk

Partitions

1 2 M-1
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk.
Step 1: Hash-partition

- Partition R into buckets, on disk

Diagram:
- Relation R
- Disk
- B(R)
- M main memory buffers
- Hash function $h$
- OUTPUT
- Partitions
- Disk
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk
Step 1: Hash-partition

- Partition R into buckets, on disk
- Partition S

Relation S

OUTPUT

INPUT

hash function h

M main memory buffers

Disk

Partitions

bucket

B(S)

1

2

\ldots

Disk

M-1

1

2
Step 2: Join Buckets

\[ R \bowtie S \]

- Read one R-bucket; hash-partition it using \( h_2 \neq h \)
Step 2: Join Buckets

$R \bowtie S$

- Read one R-bucket; hash-partition it using $h2 \neq h$
Step 2: Join Buckets

$R \bowtie S$

- Read one $R$-bucket; hash-partition it using $h_2 (\neq h)$
Step 2: Join Buckets

R \bowtie S

- Read one R-bucket; hash-partition it using $h_2 (\neq h)$
Step 2: Join Buckets

R \bowtie S

- Read one R-bucket; hash-partition it using $h_2 (\neq h)$
Step 2: Join Buckets

$R \bowtie S$

- Read one $R$-bucket; hash-partition it using $h_2 (\neq h)$
- Scan corresponding $S$ bucket and join
Step 2: Join Buckets

$R \bowtie S$

- Read one R-bucket; hash-partition it using $h_2 \neq h$
- Scan corresponding S bucket and join
Step 2: Join Buckets

\( R \bowtie S \)

- Read one R-bucket; hash-partition it using \( h2 \) (\( \neq h \))
- Scan corresponding S bucket and join
Step 2: Join Buckets

$R \bowtie S$

- Read one $R$-bucket; hash-partition it using $h_2 (\neq h)$
- Scan corresponding $S$ bucket and join
Step 2: Join Buckets

$R \bowtie S$

- Read one $R$-bucket; hash-partition it using $h_2 (\neq h)$
- Scan corresponding $S$ bucket and join
**Step 2: Join Buckets**

$$R \bowtie S$$

- Read one R-bucket; hash-partition it using $$h_2 \neq h$$
- Scan corresponding S bucket and join

Diagram:
- Buckets of R & S
- Hash table for partition $$R_i \ (< M-1 \text{ pages})$$
- Input buffer for $$S_i$$
- Output buffer
- B main memory buffers
- Join Result

CSE 544 - Winter 2024
Step 2: Join Buckets

\( R \bowtie S \)

- Read the next R bucket
Step 2: Join Buckets

$R \bowtie S$

- Read the next $R$ bucket
Step 2: Join Buckets

R \bowtie S

- Read the next R bucket
- Scan the matching S bucket
Partitioned Hash Join

- Cost: $3B(R) + 3B(S)$
- Assumption: $\min(B(R), B(S)) \leq (M-1)^2$
Summary

• Basic algorithms:
  – Nested loop
  – Hash-based
  – Sort-based

• If larger than main memory, partition data by using temporary files on disk

• Usually one partitioning step suffices: create runs, or hash-partition