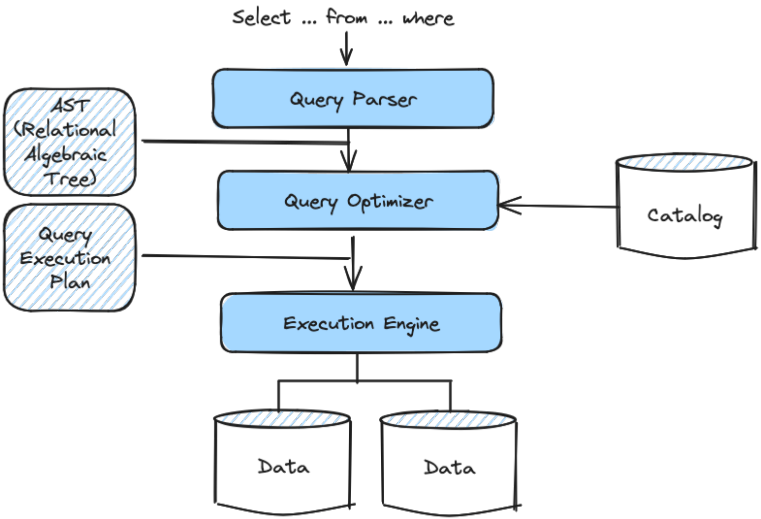
COL362/632 Introduction to Database Management Systems

Query Processing – Joins

Kaustubh Beedkar

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Query Processing (Overview)



Outline

1. Join Basics
2. Nested Loop Join
3. Sort-Merge Join
4. Hash Join

Joins

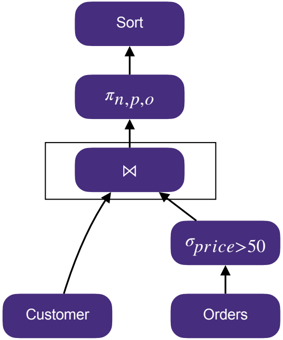
$R \bowtie_{\theta} S$

- We will assume equijoins, i.e., $\theta \equiv \{ R.A = S.A \}$
- Equijoin algorithms can be extented to support other joins
- R is outer relation
- S is inner relation

$R \bowtie S$ is one of the most common operation and needs to be carefully optimized

- Many algorithms
- No one size fits all

```
SELECT c.name, o.price, o.order_date
FROM customer c, orders o
WHERE c.customer_id = o.customer_id
AND o.price > 50
ORDER BY o.order_date DESC;
```



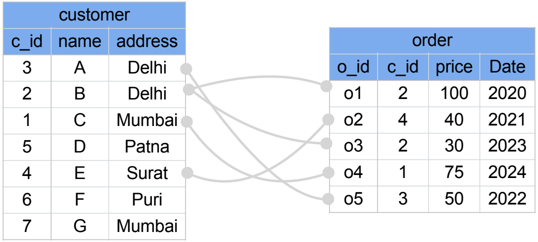
Logical Plan

Join Algorithms

1. Nested Loop Join (NLJ)
   - Naive NLJ
   - Block NLJ
   - Index NLJ

2. Sort-Merge Join

3. Hash Join
   - Simple hash join
   - Grace hash join
   - Hybrid hash join

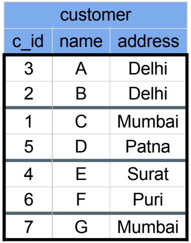


customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai

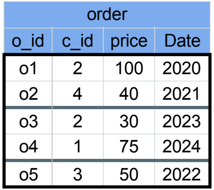
order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Notations & Cost Factors

- For relation R
  - r a tuple in R
  - b_R blocks
  - n_R tuples
- For Relation S
  - s a tuple in S
  - b_S blocks
  - n_S tuples
- Cost model based on # of I/Os
  - #blocks trasferred (recall: each block requires t_T time)
  - #blocks seeked (recall: each block requires t_S time)



customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai



order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Naive Nested Loop Join

Algorithm

```
for $r \in R$ do
    for $s \in S$ do
        if if $r$ and $s$ match then
            EMIT$(r \bowtie s)$
```

customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai

order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Cost

- #blocks transferred $= b_R + (n_R \times b_S)$
- #seeks $= b_R + n_R$
- cost $= [b_R + (n_R \times b_S)].t_T + (b_R + b_R).t_S$

Naive Nested Loop Join

Example

- $R : b_R = 100; n_R = 5000$
- $S : b_S = 400; n_S = 10,000$

Cost

- $b_R + (n_R \times b_S) = 100 + (5000 \times 400) = 2,000,100$
- $b_R + n_R = 100 + 5000 = 5010$
- At 4 ms/block seek time and 0.1 ms/block transfer time, total time $\approx$ 3.5min

If we consider S as the outer relation

- $b_S + (n_S \times b_R) = 400 + (10,000 \times 100) = 1,000,400$
- $b_S + n_S = 100 + 5000 = 10,400$
- At 4 ms/block seek time and 0.1 ms/block transfer time, total time $\approx$ 2.3min

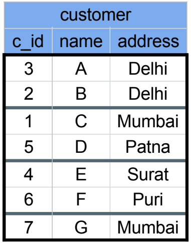
Block Nested Loop Join

Algorithm

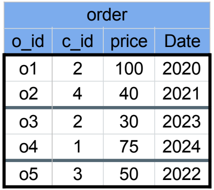
```
for each block $B_R \in R$ do
    for each block $B_S \in S$ do
        for each $r \in B_R$ do
            for each $s \in B_S$ do
                if $r$ and $s$ match then
                    EMIT($r \bowtie s$)
```

Cost

- ▶ block transfers: $b_R + (b_R \times b_S)$
- ▶ seeks: $2b_R$



customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai



order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Block Nested Loop Join

- Reduces to access to disk
- Smaller relation should be the outer table (why?)
  - Smaller based on number of pages!

Optimizations

- If join attribute of inner relation is a key, terminate outer loop as soon as a match is found
- Cache-conscious inner loop: scan alternatively forward and backward (how does this help?)
- Use biggest size of a relation as blocking unit (next slide)

Block Nested Loop Join

Assume that we have B buffer pages

- Use $B-2$ buffer pages for R
- Use one buffer page for S
- Use one buffer page for output

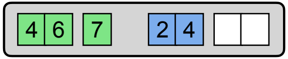
Algorithm

```
for each $B-2$ blocks $B'_R \in R$ do
    for each block $B_S \in S$ do
        for each $r \in B-2$ blocks do
            for each $s \in B_S$ do
                if $r$ and $s$ match then
                    $\text{EMIT}(r \bowtie s)$
```

- Improved cost $= b_R + (\left\lceil \frac{b_R}{B-2} \right\rceil \times b_S)$
- Improved seeks $= 2 \left\lceil \frac{b_R}{B-2} \right\rceil$

customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai

order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022



and so on...

Buffer pool (B=4)

Block Nested Loop Join

Example

- $R : b_R = 100; n_R = 5000$
- $S : b_S = 400; n_S = 10,000$

Cost

If $B = 12$

- $b_R + (\left\lceil \frac{b_R}{B-2} \right\rceil \times b_S) = 100 + 10 \times 400 = 4,100$
- $2 \left\lceil \frac{b_R}{B-2} \right\rceil = 2 \times 10 = 20$
- At 0.1ms/block transfer time and 4ms/block seek time, total time ≈ 0.4s

If $b_R < B - 2$ (outer relation will fit in memory)

- 500 block transfers
- 2 seeks
- At 0.1ms/block transfer time and 4ms/block seek time, total time ≈ 0.05s

Index Nested Loop Join

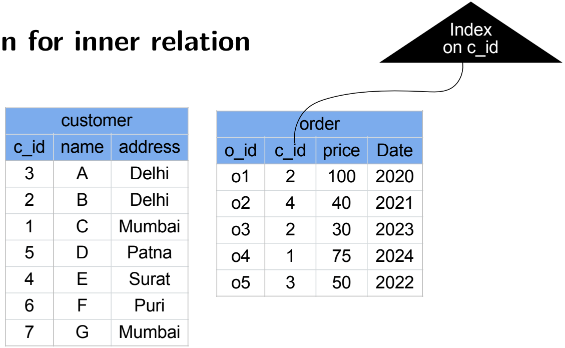
Use an index scan instead of sequential scan for inner relation

Algorithm

```
for each $r \in R$ do
    for each s $\in$ $\textsc{Index}(r_i = s_j)$ do
        if $r$ and $s$ match then
            $\textsc{emit}(r \bowtie s)$
```

Cost

- b_R block transfers $+$ b_R seeks
- index scan cost: $n_R \times C$
- cost $= b_R(t_T + t_S) + n_R \times C$
  - Assuming C is the cost of each index probe (recall cost of single selection)
- If index on both relations, use one with fewer tuples as outer relation



customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai

order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Sort Merge Join

1. Sorting
    - Sort relations on join key(s)
2. Merge
    - Scan two sorted relations and emit matching tuples
        - Note: This is different than merging of external sort

Sort Merge Join

Algorithm

```
$R' \leftarrow \textsc{Sort}(R)$, $S' \leftarrow \textsc{Sort}(S)$ on join keys
$p_R \leftarrow$ address of first tuple in $R'$
$p_S \leftarrow$ address of first tuple in $S'$
while $p_R$ and $p_S$ do
    if $p_R > p_S$ then
        $p_S \leftarrow$ next tuple of $S'$
    if $p_R < p_S$ then
        $p_R \leftarrow$ next tuple of $R'$
        backup if required
    if $p_R$ and $p_S$ match then
        $\textsc{Emit}(r \bowtie s)$
        $p_S \leftarrow$ next tuple of $S'$
```

	customer		
	c_id	name	address
	1	C	Mumbai
	2	B	Delhi
	3	A	Delhi
→	4	E	Surat
	5	D	Patna
	6	F	Puri
	7	G	Mumbai

order			
o_id	c_id	price	Date
o4	1	75	2024
o1	2	100	2020
o3	2	30	2023
o5	3	50	2022
o2	4	40	2021

→ EOF

Sort Merge Join

Cost

- Sorting R: $2b_R(1 + \left\lceil \log_{B-1} \left\lceil \frac{b_R}{B} \right\rceil \right\rceil)$
- Sorting S: $2b_S(1 + \left\lceil \log_{B-1} \left\lceil \frac{b_S}{B} \right\rceil \right\rceil)$
- Merging: $b_R + b_S$ (block transfers)
- Seeking: $\left\lceil \frac{b_R}{b_B} \right\rceil + \left\lceil \frac{b_S}{b_B} \right\rceil$
  - b_B buffer blocks allocated to each relation

Sort Merge Join

Example

- $R : b_R = 100; n_R = 5000$
- $S : b_S = 400; n_S = 10,000$

Cost

- if B=11 and $b_B = 1$
- sorting cost for $R : 2 \times 100(1 + \lceil \log_{10} \lceil 100/11 \rceil \rceil) = 400$
- sorting cost for $S : 2 \times 400(1 + \lceil \log_{10} \lceil 400/11 \rceil \rceil) = 2,400$
- merging cost $= 500$
- seeking: cost $= 500$
- At 0.1ms/transfer seek time and 4ms/block seek time, total time $\approx$ 2.3s

Sort Merge Join

Notes

- Worst case: when join attribute of all tuples contains the same value
- Useful when table(s) is already sorted in join key
- Useful when sorted output is required

Basic Idea

Key insight

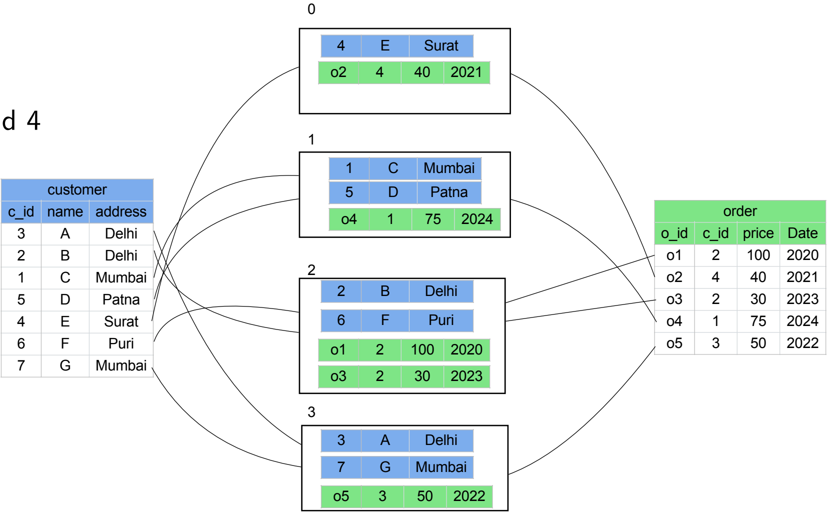
- if $r \in R$ matches $s \in S$, then both tuples when hashed on the join attribute using the same hash function should be in the same bucket i

- Partition R into $R_0, R_1, \ldots R_n$
- Partition S into $S_0, S_1, \ldots S_n$
- Using a hash function $h : \texttt{join\_attribute} \mapsto \{0, 1, \ldots, n\}$

Basic Idea

Example

- $h : x \bmod 4$



customer		
c_id	name	address
3	A	Delhi
2	B	Delhi
1	C	Mumbai
5	D	Patna
4	E	Surat
6	F	Puri
7	G	Mumbai

0

4	E	Surat	
o2	4	40	2021

1

1	C	Mumbai	
5	D	Patna	
o4	1	75	2024

2

2	B	Delhi	
6	F	Puri	
o1	2	100	2020
o3	2	30	2023

3

3	A	Delhi	
7	G	Mumbai	
o5	3	50	2022

order			
o_id	c_id	price	Date
o1	2	100	2020
o2	4	40	2021
o3	2	30	2023
o4	1	75	2024
o5	3	50	2022

Naive In-memory Hash Join

1. **Build:** Scan the outer relation and build a hash table
2. **Probe:** Scan the inner relation and find the matching tuple

Algorithm

```
build a hash table $H$ for $R$
for each $s \in S$ do
    if $h(s) \in H$ then
        find matching tuple(s) and EMIT
```

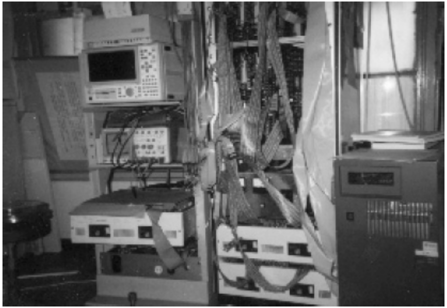
- Does not work on large relations
- Buffer pool manager may swap out pages of hash table!

Grace Hash Join

- Also known as partitioned hash join
- Adopts a divide and conquer approach
- **Partitioning phase:** Partition R and S using the same hash function
- **Build & Probe Phase:** Compare tuples in each partition

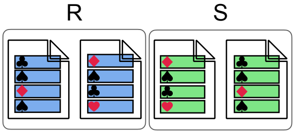
GRACE Parallel Relational Database Machine

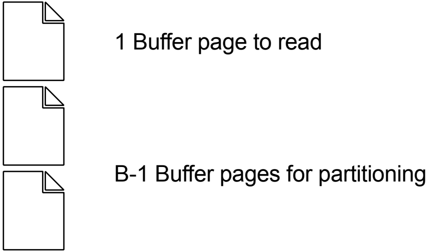
https://museum.ipsj.or.jp/en/computer/other/0014.html



Grace Hash Join (Example)

- Compute $R \bowtie S$



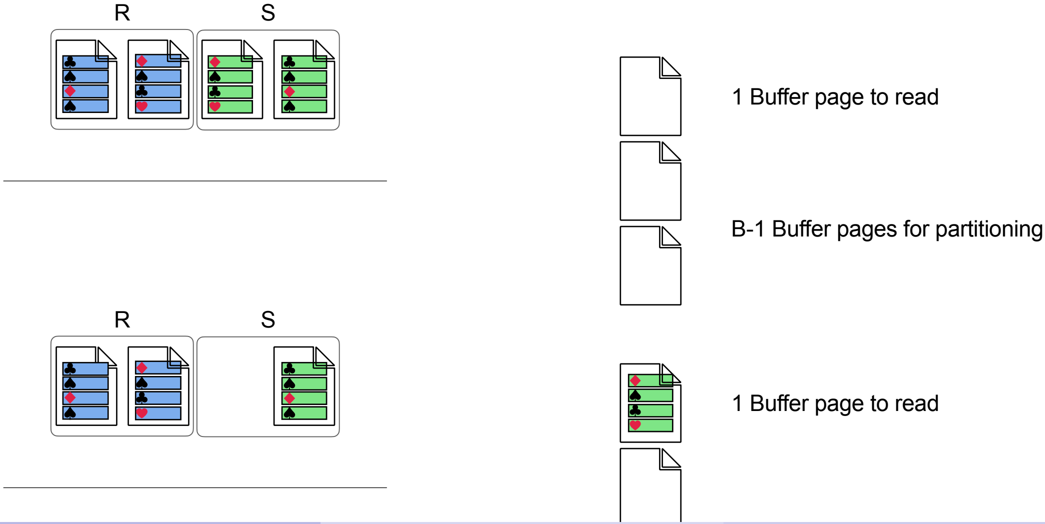


1 Buffer page to read

B-1 Buffer pages for partitioning

Grace Hash Join (Example)– Partitioning Phase

- Partition *S*



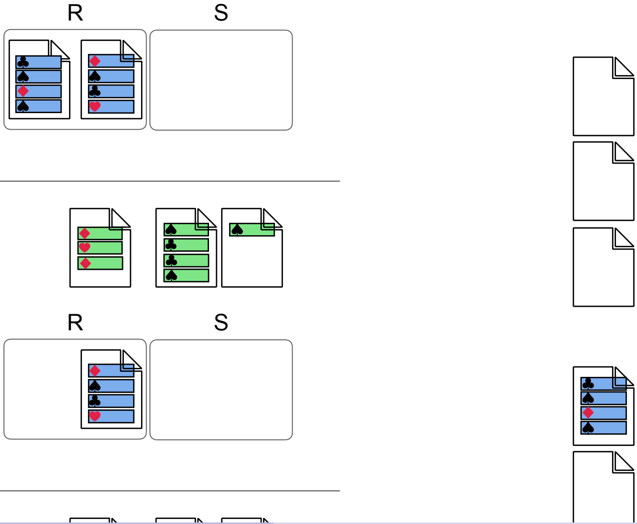
1 Buffer page to read

B-1 Buffer pages for partitioning

1 Buffer page to read

Grace Hash Join (Example)– Partitioning Phase

- Partition *R*



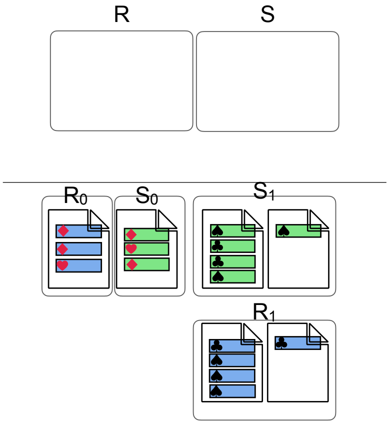
1 Buffer page to read

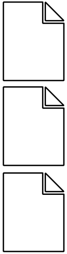
B-1 Buffer pages for partitioning

1 Buffer page to read

Grace Hash Join (Example)

- After partitioning *R* and *S*, we have



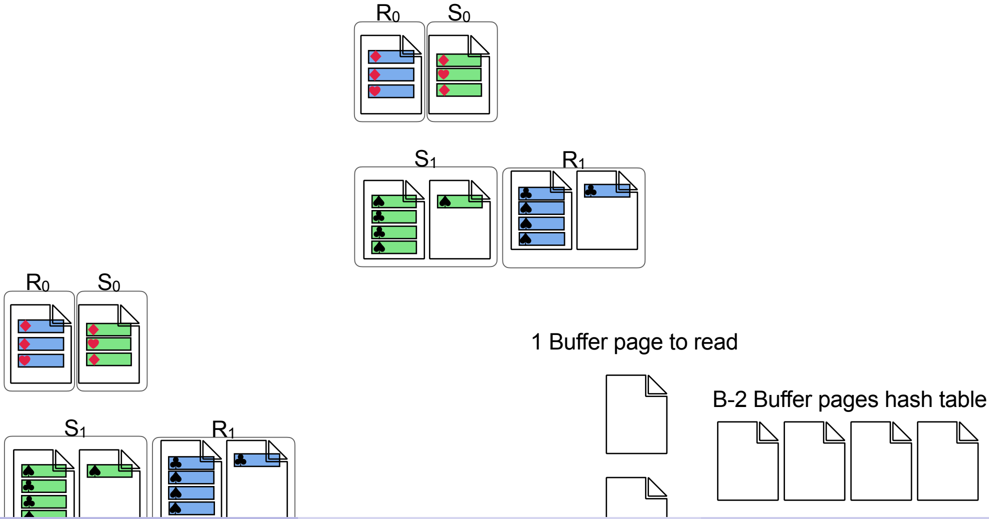


1 Buffer page to read

B-1 Buffer pages for partitioning

Grace Hash Join (Example) – Build & Probe Phase

- Build in-memory hash table on *R*

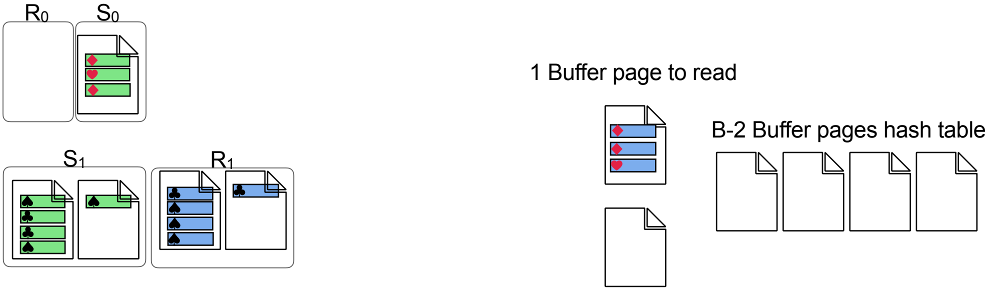


1 Buffer page to read

B-2 Buffer pages hash table

Grace Hash Join (Example) – Build & Probe Phase

- Build in-memory hash table on *R*



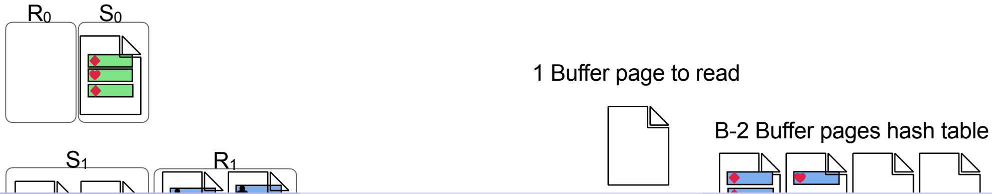
R_0 S_0

S_1 R_1

1 Buffer page to read

B-2 Buffer pages hash table

1 Buffer page for output



R_0 S_0

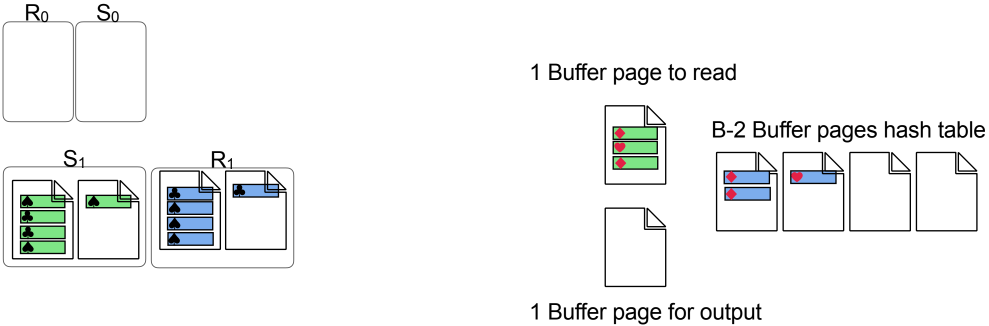
S_1 R_1

1 Buffer page to read

B-2 Buffer pages hash table

Grace Hash Join (Example) – Build & Probe Phase

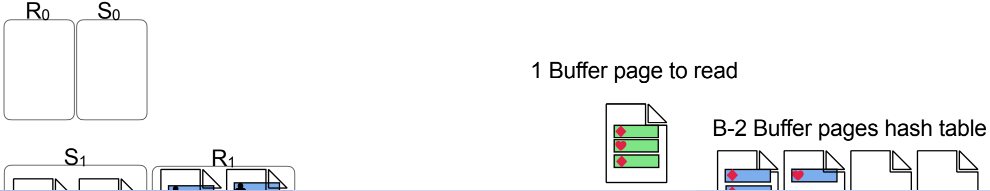
- Probe *S* and compute matching tuples



1 Buffer page to read

B-2 Buffer pages hash table

1 Buffer page for output

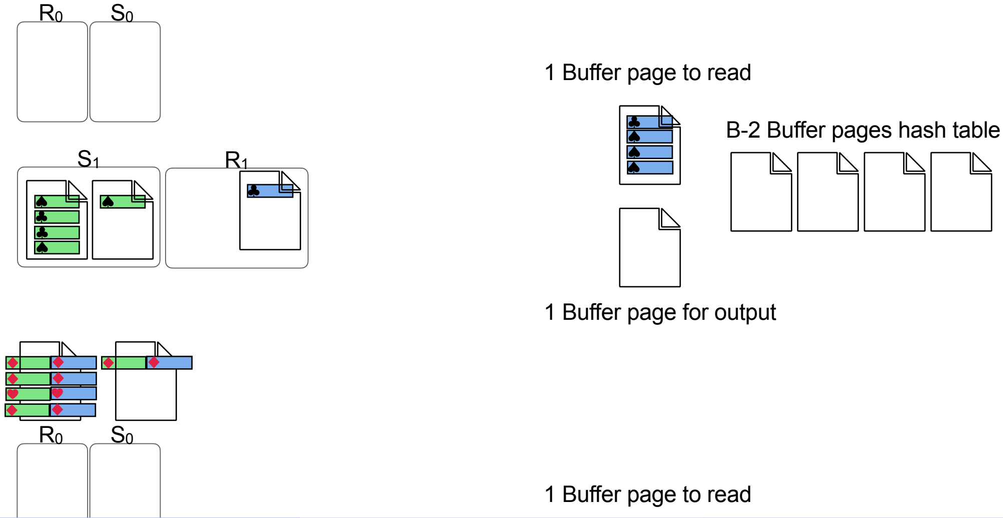

R0
S0
S1
R1

1 Buffer page to read

B-2 Buffer pages hash table

Grace Hash Join (Example) – Build & Probe Phase

- Same process for other partitions



R_0 S_0

S_1 R_1

R_0 S_0

1 Buffer page to read

B-2 Buffer pages hash table

1 Buffer page for output

1 Buffer page to read

Corner Cases

If $n \geq \#$ blocks of memory

- Relations cannot be partitioned in one pass
- in one pass, a relation can at most be partitioned into $B - 1$ partitions (or number of buffers available as output buffers)
- if a partition does not fit in memory, then **recursively partition** using different hash functions

If partitioning is **skewed**

- Increase the number of partitions by a **fudge factor** (usually 20%)
- **Overflow resolution** by re-partitioning
- **Overflow avoidance** by first create many smaller partitions, then combine them to fit in memory
- if resolution and avoidance fail, then use NLJ for keys with “many” values

Partitioned Hash Join

Cost (if no recursive partitioning is required)

- partitioning phase : $2 \times (b_R + b_S)$ block transfers
- build & probe phase : $b_R + b_S + 4n$

  – $4n$ is the overhead for partially filled blocks (usually ignored in cost calculations)
- Total transfer cost $= 3(b_R + b_S)$

- $2(\left\lceil \frac{b_R}{b_B} \right\rceil + \left\lceil \frac{b_S}{b_B} \right\rceil)$ seeks in partitioning phase
- $2n$ seeks in build and probe phase
- Total seek cost $= 2(\left\lceil \frac{b_R}{b_B} \right\rceil + \left\lceil \frac{b_S}{b_B} \right\rceil) + 2n$

Partitioned Hash Join

Example

- $R : b_R = 100; n_R = 5000$
- $S : b_S = 400; n_S = 10,000$

Cost

- assuming $b_B = 3$
- $3(100 + 400) = 1500$ block transfers
- $2(\left\lceil \frac{100}{3} \right\rceil + \left\lceil \frac{400}{3} \right\rceil) = 336$ seeks
- At 0.1ms/block transfer time and 4ms/block seek time, total time $\approx$ 0.6s

Note

- If memory is large enough, simple hash join requires only $b_R + b_S$ block transfers and 2 seeks!

Hash Join

Notes

- We don't care about size of inner table, only outer table needs to fit in memory
- Use static hashing, if size of outer table is known

  otherwise, use dynamic hashing (Recall extendible hashing)

Homework exercise

Join Algorithm	I/O cost (block transferred)	Example
NLJ		
BNLJ		
SMJ		
GHJ		

Other Operators

Self study: aggregation and set operators (B1: Ch 15 (15.6))