

# Algorithms for Query Processing and Optimization

# Chapter Outline

- 0. Introduction to Query Processing
- 1. Translating SQL Queries into Relational Algebra
- 2. External Sorting
- 3. Algorithms for SELECT and JOIN Operations
- 4. Algorithms for PROJECT and SET Operations
- 5. Implementing Aggregate Operations and Outer Joins

# Different File Organizations

We need to understand the importance of appropriate file organization and index

Search key = <age, sal>

Consider following options:

- Heap files
  - random order; insert at end-of-file
- Sorted files
  - sorted on <age, sal>
- Clustered B+ tree file
  - search key <age, sal>
- Heap file with unclustered B+-tree index
  - on search key <age, sal>
- Heap file with unclustered hash index
  - on search key <age, sal>

# Possible Operations

eh el operation elly bn3mlha lama bnb3t query?

- **Scan** hat kol el records.
  - Fetch all records from disk to buffer pool
- **Equality search** eny akhtar keyam mo3ayna bs.
  - Find all employees with age = 23 and sal = 50
  - Fetch page from disk, then locate qualifying record in page
- **Range selection** bgeb range mo3yn, el B+ tree momken a2f 3nd 7eta mo3yna w ageb kol el akbur meno aw el asghr meno w keda.
  - Find all employees with age > 35
- **Insert a record** tb3n lazim a3rf lama agy a3ml insert, h5ud w2t ad a w keda.
  - identify the page, fetch that page from disk, insert record, write back to disk (possibly other pages as well)
- **Delete a record**
  - similar to insert

# Understanding the Workload

- A workload is a mix of queries and updates
- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected

# Choice of Indexes

wna gy akhtar el index, bshof eh aktur

index ghalbn hytdnh beh kter, w bast5dm el index bta3o 3la el  
asas da.

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

brdu lazmn brdu, ne5tar el option aw el data structure elly hnst5dmha.

# More on Choice of Indexes

- One approach:
  - Consider the most important queries
  - Consider the best plan using the current indexes
  - See if a better plan is possible with an additional index.
  - If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans
  - We will learn query execution and optimization later - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload

# Trade-offs for Indexes

- Indexes can make
  - queries go faster
  - updates slower
- Require disk space, too



# Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index
  - Range query suggests tree index
  - Clustering is especially useful for range queries
    - can also help on equality queries if there are many duplicates
- Try to choose indexes that benefit as many queries as possible
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - Order of attributes is important for range queries
- Note: clustered index should be used judiciously
  - expensive updates, although cheaper than sorted files

# Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples

What is a good indexing strategy?

- How selective is the condition?
  - everyone > 40, index not of much help, scan is as good
  - Suppose 10% > 40. Then?

```
SELECT E.dno  
FROM Emp E  
WHERE E.age>40
```

- Depends on if the index is clustered
  - otherwise can be more expensive than a linear scan
  - if clustered, 10% I/O (+ index pages)

Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

# Examples of Clustered Indexes

## Group-By query

What is a good indexing strategy?

- Use *E.age* as search key?
  - Bad If many tuples have *E.age* > 10 or if not clustered....
  - ...using *E.age* index and sorting the retrieved tuples by *E.dno* may be costly
- Clustered *E.dno* index may be better
  - First group by, then count tuples with age > 10
  - good when age > 10 is not too selective
- Note: the first option is good when the WHERE condition is highly selective (few tuples have age > 10), the second is good when not highly selective

```
SELECT E.dno, COUNT (*)  
FROM Emp E  
WHERE E.age>10  
GROUP BY E.dno
```

Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

# Examples of Clustered Indexes

What is a good indexing strategy?

## Equality queries and duplicates

- Clustering on *E.hobby* helps
  - hobby not a candidate key, several tuples possible
- Does clustering help now?
  - (eid = key)
  - Not much
  - at most one tuple satisfies the condition

```
SELECT E.dno  
FROM Emp E  
WHERE E.hobby='Stamps'
```

Which attribute(s)?  
Clustered/Unclustered?  
B+ tree/Hash?

```
SELECT E.dno  
FROM Emp E  
WHERE E.eid=50
```

# Indexes with Composite Search Keys

m3nah, eny momken a3ml indexing, bestkhdam composite key msh shrt ykon single key.

el composite key:  
\* bya5ud disk size aktur mn el single aked.

\* hwa by3ml retrieve asra3 mn el single key.

fanta bt5sr disk size w insertion time mokabl takhud sor3a fl retrieval.

**Composite Search Keys:** Search on a combination of fields

**Equality query:** Every field value is equal to a constant value. E.g. wrt **<sal,age> index:**

– age=20 and sal =75

**Range query:** Some field value is not a constant. E.g.:

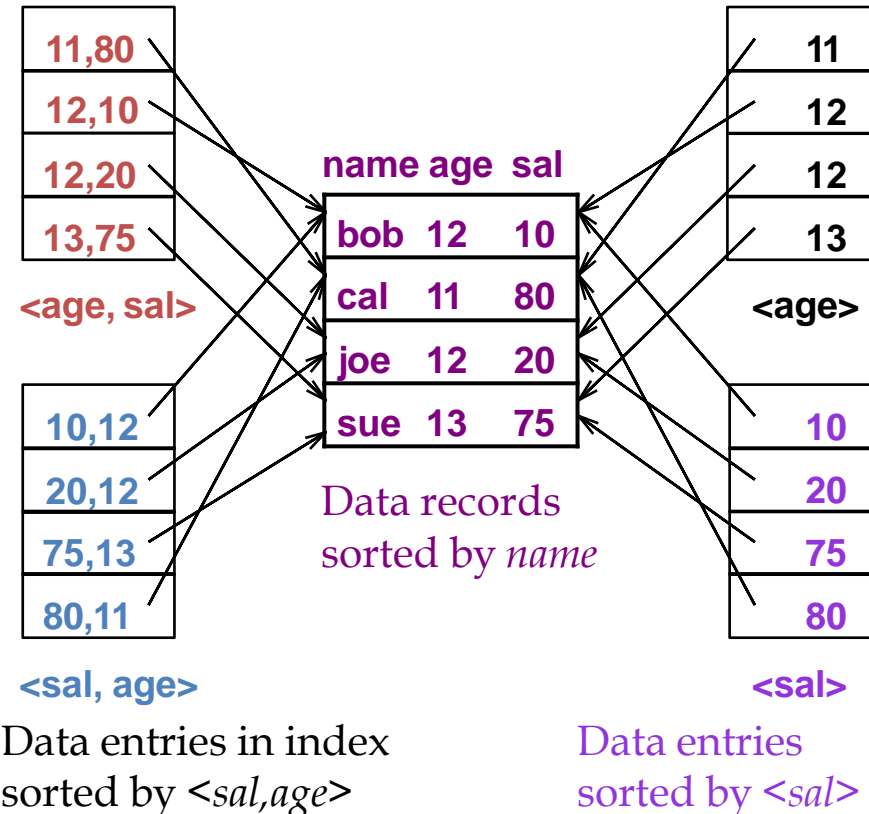
- sal > 10 – which combination(s) would help?
- <age, sal> does not help
- B+tree on <sal> or <sal, age> helps
- has to be a prefix

using one index.

hena el mfrod nshof eh el column elly hy3ml filtering asr3 w ast5dmo el awl, w b3den alg2 lel key elly b3do w hakaza.

akny b2olk masln hatly kol el employees elly senhom akbur mn 60 w el salary asghar mn 100000 msln. el zka2 hena eny awl 7aga a3ml filtration 3la el age el awl, b3d keda a3ml filtration 3la el salary l2n el nas elly senhom akbur mn 60 aked hayb2o olylen, fa sa3tha a2dr fe 3dd a2al mn el rows, eny adwr 3la el ana 3auzo, 3aks lw kan masln 2lk, hatly el employees elly senhom asghar mn 60, sa3tha aked mkonash hnbd2 bl age, l2n da t2rebn keda hyb2a el table bta3 el employees kolohom.

Examples of composite key indexes using lexicographic order.



# Composite Search Keys

- To retrieve Emp records with  $age=30$  AND  $sal=4000$ , an index on  $\langle age, sal \rangle$  would be better than an index on  $age$  or an index on  $sal$ 
  - first find  $age = 30$ , among them search  $sal = 4000$
- If condition is:  $20 < age < 30$  AND  $3000 < sal < 5000$ :
  - Clustered tree index on  $\langle age, sal \rangle$  or  $\langle sal, age \rangle$  is best.
- If condition is:  $age=30$  AND  $3000 < sal < 5000$ :
  - Clustered  $\langle age, sal \rangle$  index much better than  $\langle sal, age \rangle$  index
  - more index entries are retrieved for the latter
- Composite indexes are larger, updated more often

# Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available

```
SELECT E.dno, COUNT(*)  
FROM Emp E  
GROUP BY E.dno
```

$\langle E.dno \rangle$

mogrd htro7 bs lel index, w mno a2dr a3rf el egaba el ana 3auzha, mn gher ma a7tag aro7 lel table nafso.

- For index-only strategies, clustering is not important

```
SELECT E.dno, MIN(E.sal)  
FROM Emp E  
GROUP BY E.dno
```

$\langle E.dno, E.sal \rangle$

*Tree index!*

$\langle E.age, E.sal \rangle$

*Tree index!*

```
SELECT AVG(E.sal)  
FROM Emp E  
WHERE E.age=25 AND  
E.sal BETWEEN 3000 AND 5000
```

# External Sorting

lw 3ndk query msln select E.dno from Employee

hal de me7taga sorting?

l-awl wahla keda, ht2ol eno l2, lakn SQL Server msln w keda, hwa msh byrg3 duplicates, w 34an ye2dr eno yermy el duplicates, fa enta ashal haga te3mlha enk te3mlo sorting, w b3dha, takhud awl occurence bs, w tkml. fa hena by7tag ye3ml sorting ah

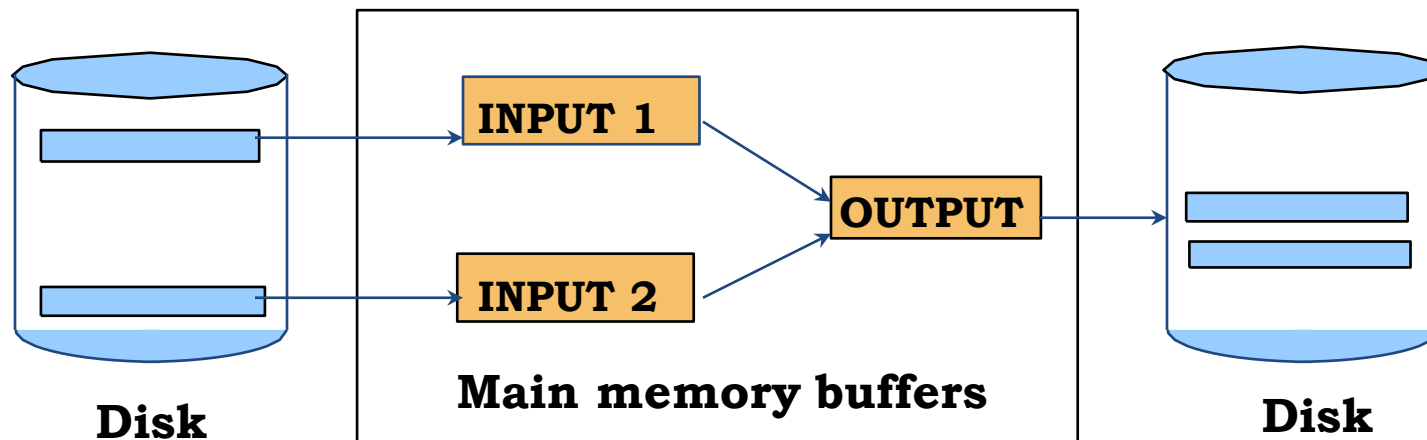


# Why Sort?

- A classic problem in computer science
- Data requested in sorted order
  - e.g., find students in increasing gpa order
- Sorting is first step in bulk loading B+ tree index
- Sorting useful for eliminating duplicate copies in a collection of records
- Sort-merge join algorithm involves sorting
- Problem: sort 1Gb of data with 1Mb of RAM
  - need to minimize the cost of disk access

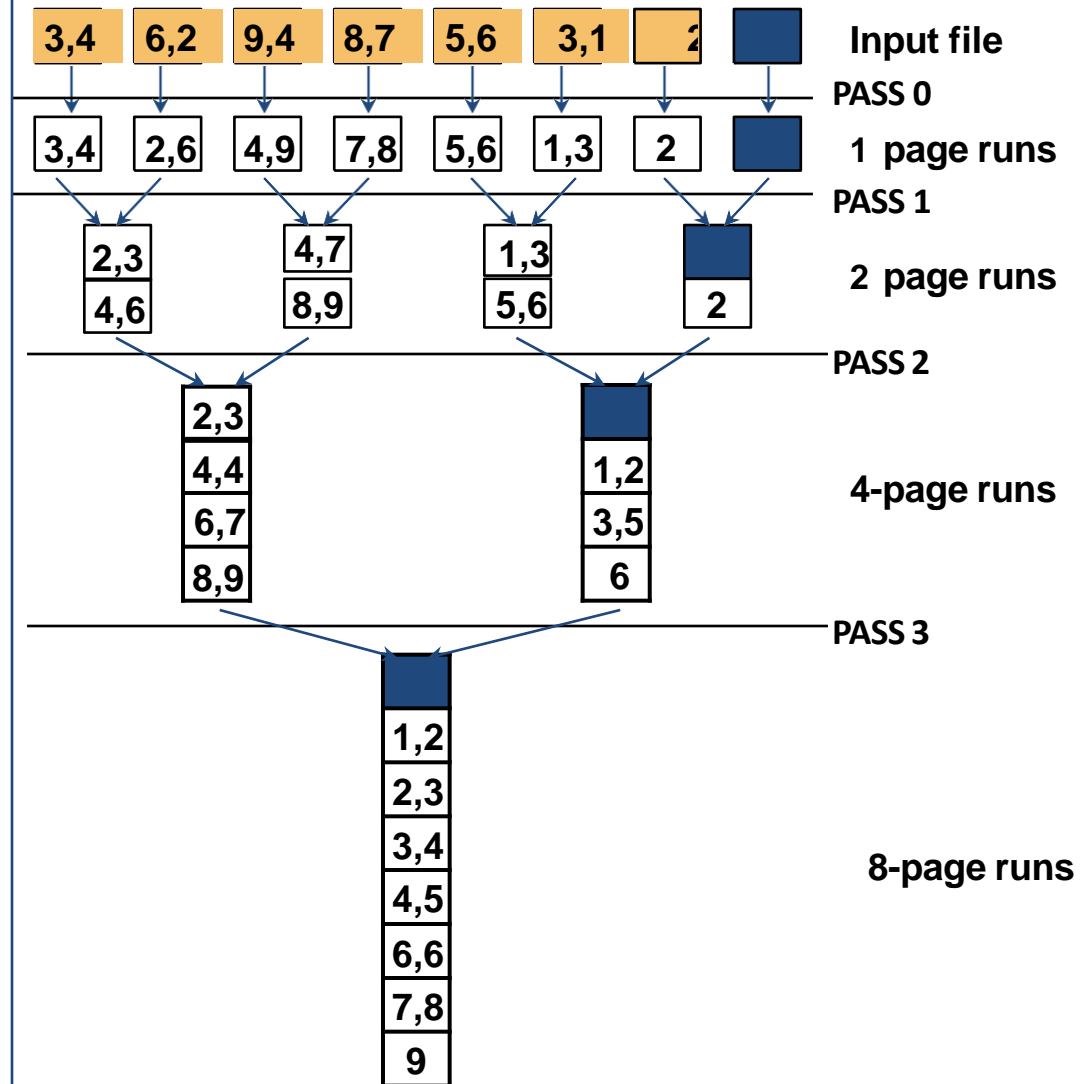
# 2-Way Sort: Requires 3 Buffers

- Suppose  $N = 2^k$  pages in the file
- Pass 0: Read a page, sort it, write it.
  - repeat for all  $2^k$  pages
  - only one buffer page is used
- Pass 1:
  - Read two pages, sort (merge) them using one output page, write them to disk
  - repeat  $2^{k-1}$  times
  - three buffer pages used
- Pass 2, 3, 4, ..... continue



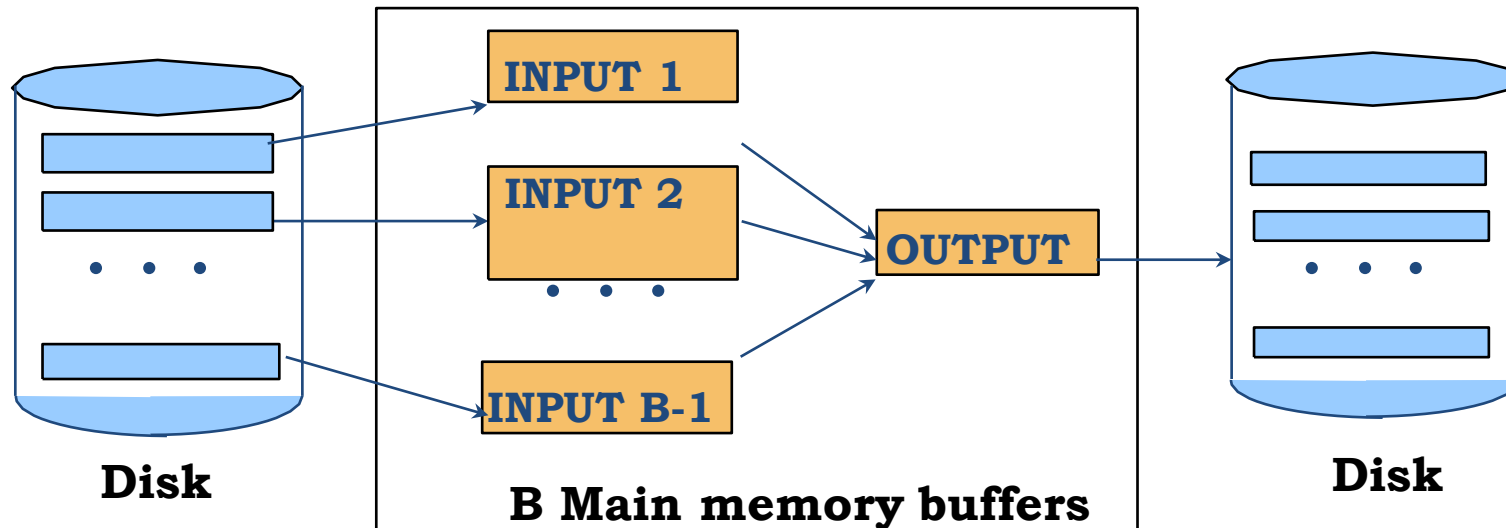
# Two-Way External Merge Sort

- Each sorted sub-file is called a **run**
  - each run can contain multiple pages
- Each pass we read + write each page in file.
- N pages in the file,
- => the number of passes  $= \lceil \log_2 N \rceil + 1$
- So total cost is:  $2N (\lceil \log_2 N \rceil + 1)$
- Not too practical, but useful to learn basic concepts for external sorting



# General External Merge Sort

- Suppose we have more than 3 buffer pages.
- How can we utilize them?
- To sort a file with  $N$  pages using  $B$  buffer pages:
  - Pass 0: use  $B$  buffer pages:
    - Produce  $\lceil N/B \rceil$  sorted runs of  $B$  pages each.
  - Pass 1, 2, ..., etc.: merge  $B-1$  runs to one output page
    - keep writing to disk once the output page is full



# Cost of External Merge Sort

- Number of passes:  $1 + \lceil \log_{B-1} [N/B] \rceil$
- Cost =  $2N * (\text{\# of passes})$  – why 2 times?
- E.g., with 5 buffer pages, to sort 108 page file:
- Pass 0: sorting 5 pages at a time
  - $\lceil 108/5 \rceil = 22$  sorted runs of 5 pages each (last run is only 3 pages)
- Pass 1: 4-way merge
  - $\lceil 22/4 \rceil = 6$  sorted runs of 20 pages each (last run is only 8 pages)
- Pass 2: 4-way merge
  - (but 2-way for the last two runs)
  - $\lceil 6/4 \rceil = 2$  sorted runs, 80 pages and 28 pages
- Pass 3: 2-way merge (only 2 runs remaining)
  - Sorted file of 108 pages

# Number of Passes of External Sort

High B is good, although CPU cost increases

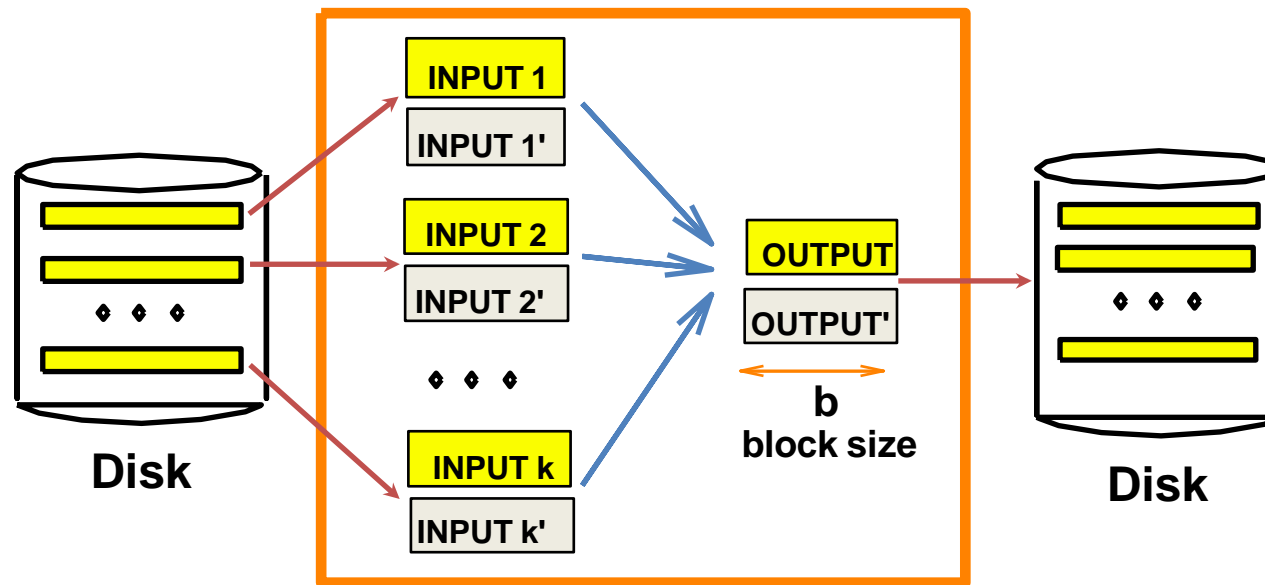
N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

# I/O for External Merge Sort

- If 10 buffer pages
  - either merge 9 runs at a time with one output buffer
  - or 8 runs with two output buffers
- If #page I/O is the metric
  - goal is minimize the #passes
  - each page is read and written in each pass
- If we decide to read a block of  $b$  pages sequentially
  - Suggests we should make each buffer (input/output) be a block of pages
  - But this will reduce fan-out during merge passes
    - i.e. not as many runs can be merged again any more
  - In practice, most files still sorted in 2-3 passes

# Double Buffering

- To reduce CPU wait time for I/O request to complete, can prefetch into 'shadow block'.





# Overview of Query Evaluation

# Overview of Query Evaluation

- How queries are evaluated in a DBMS
  - How DBMS describes data (tables and indexes)
- Relational Algebra Tree/Plan = Logical Query Plan
- Now Algorithms will be attached to each operator = Physical Query Plan
- Plan = Tree of RA ops, with choice of algorithm for each op.
  - Each operator typically implemented using a “pull” interface
  - when an operator is “pulled” for the next output tuples, it “pulls” on its inputs and computes them

# Overview of Query Evaluation

- Two main issues in query optimization:
  1. For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan
  2. How is the cost of a plan estimated?
- Ideally: Want to find best plan
- Practically: Avoid worst plans!

e7na ka bashar, bn7awl newsl le a7sn plan, lakn el computer fl 7a2e2a, by7awl yeb3d 3n aw7sh plan.

Assumption: ignore final write

- i.e. assume that your final results can be left in memory
  - and does not be written back to disk
  - unless mentioned otherwise
- Why such an assumption?

# Algorithms for Joins

# Equality Joins With One Join Column

```
SELECT *  
FROM   Reserves R, Sailors S  
WHERE  R.sid=S.sid
```

- In algebra:  $R \bowtie S$ 
  - Common! Must be carefully optimized
  - $R \times S$  is large; so,  $R \times S$  followed by a selection is inefficient
- Cost metric: # of I/Os
  - Remember, we will ignore output costs (always)
    - = the cost to write the final result tuples back to the disk

# Common Join Algorithms

## 1. Nested Loops Joins (NLJ)

- Simple nested loop join
- Block nested loop join

## 2. Sort Merge Join Very similar to external sort

## 3. Hash Join

# Algorithms for Joins

## 1. NESTED LOOP JOINS



# Simple Nested Loops Join

$R \bowtie S$

$O(M * p_r)$  foreach tuple  $r$  in  $R$  do  
 $O(N * p_s)$      foreach tuple  $s$  in  $S$  where  $r_i == s_j$  do  
                   add  $\langle r, s \rangle$  to result

$M = 1000$  pages in  $R$   
 $p_R = 100$  tuples per page

$N = 500$  pages in  $S$   
 $p_S = 80$  tuples per page

- For each tuple in the **outer** relation  $R$ , we scan the entire **inner** relation  $S$ .

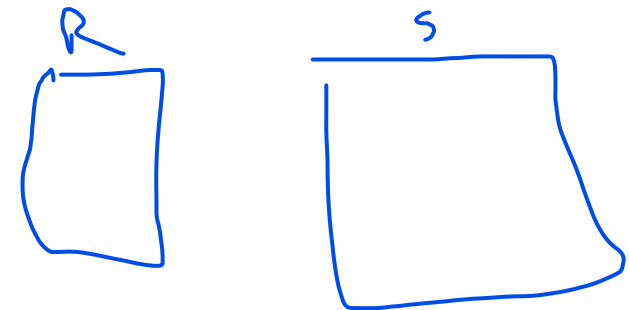
– Cost:  $M + (p_R * M) * N = 1000 + 100 * 1000 * 500$  I/Os.

- Page-oriented Nested Loops join:**

- For each *page* of  $R$ , get each *page* of  $S$
- and write out matching pairs of tuples  $\langle r, s \rangle$
- where  $r$  is in  $R$ -page and  $S$  is in  $S$ -page.
- Cost:  $M + M * N = 1000 + 1000 * 500$

- If smaller relation ( $S$ ) is outer

– Cost:  $N + M * N = 500 + 500 * 1000$



e7na hn2ra el R -> M readings keda keda.  
 el S -> hn2raha  $M * N * p_r$  -> 34an de nested loop

akn el R byzed lw7do, w l el S byzed lw7do fa lw 7sbtha

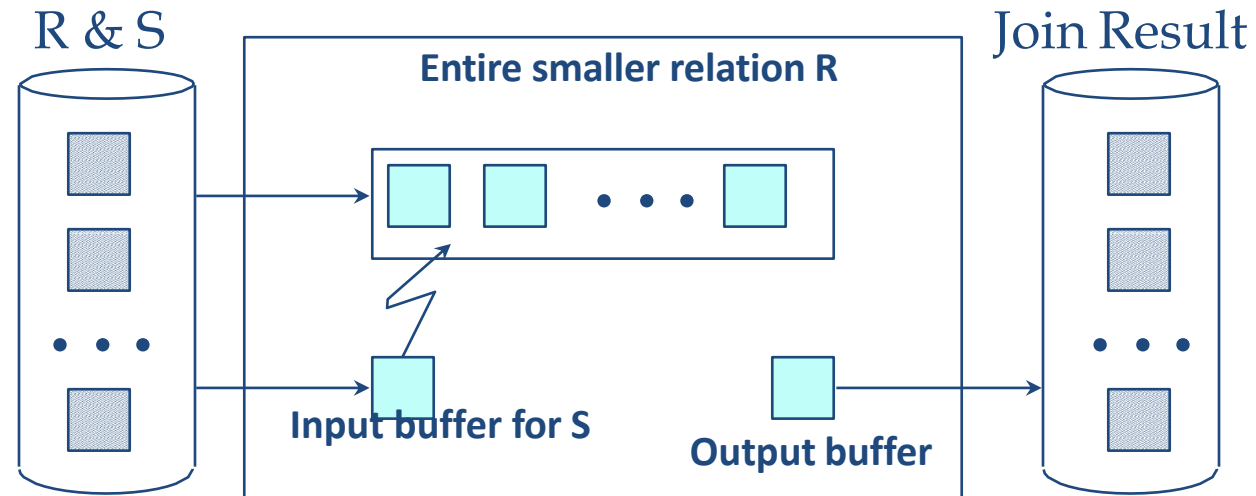
**How many buffer pages  
do you need?**

1 1  
 2 2  
 3 3  
 4 4  
 5 5  
 2 1  
 2 2  
 3 3  
 .. htl2y el R arenaha M  
 wl S arenaha  $M * N * p_s$

lw hnst5dm el algo da, dayman khly el R hya el soghyra.

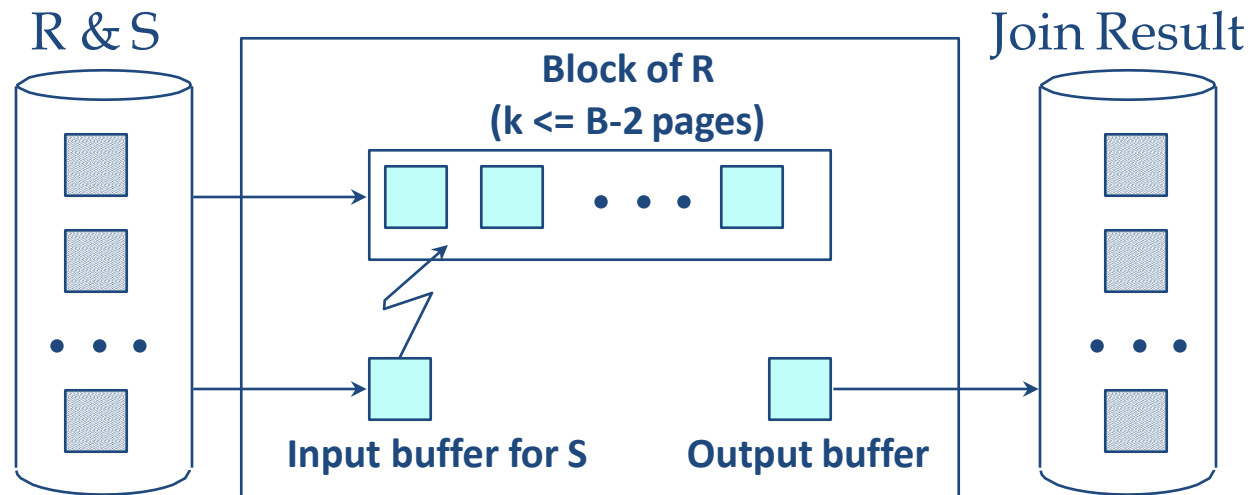
# Block Nested Loops Join

- Simple-Nested does not properly utilize buffer pages (uses 3 pages)
- Suppose have enough memory to hold the smaller relation  $R$  + at least two other pages
  - e.g. in the example on previous slide ( $S$  is smaller), and we need  $500 + 2 = 502$  pages in the buffer
- Then use one page as an input buffer for scanning the inner
  - one page as the output buffer
  - For each matching tuple  $r$  in  $R$ -block,  $s$  in  $S$ -page, add  $\langle r, s \rangle$  to result
- Total I/O =  $M+N$
- What if the entire smaller relation does not fit?



# Block Nested Loops Join

- If R does not fit in memory,
  - Use one page as an input buffer for scanning the inner S
  - one page as the output buffer
  - and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add  $\langle r, s \rangle$  to result
  - Then read next R-block, scan S, etc.



# Cost of Block Nested Loops

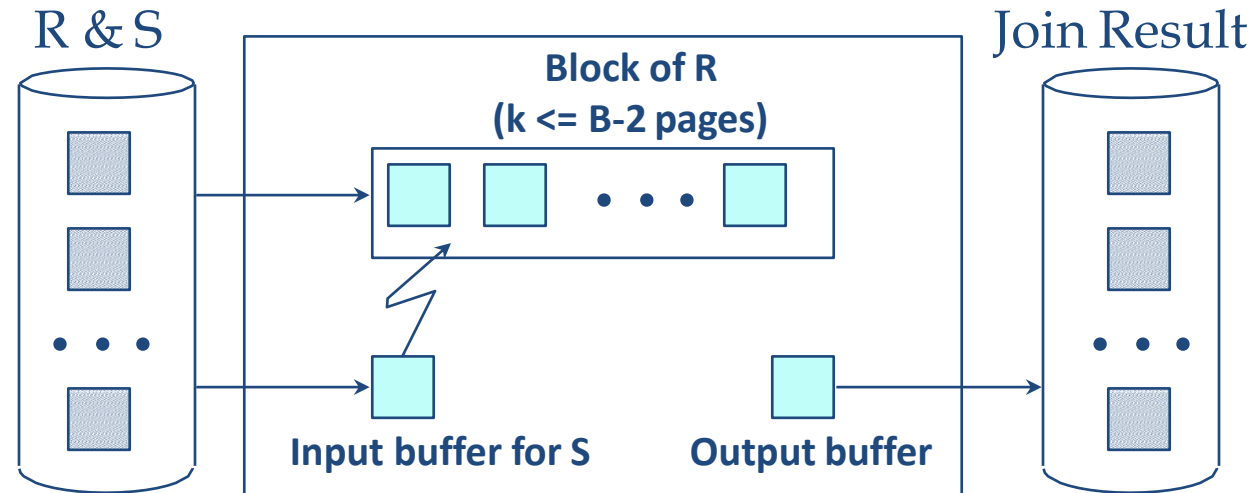
$M = 1000$  pages in  $R$   
 $p_R = 100$  tuples per page

$N = 500$  pages in  $S$   
 $p_S = 80$  tuples per page

in class

- $R$  is outer
- $B-2 = 100$ -page blocks
- How many blocks of  $R$ ?
- Cost to scan  $R$ ?
- Cost to scan  $S$ ?
- Total Cost?

```
foreach block of B-2 pages of R do
  foreach page of S do {
    for all matching in-memory tuples r in R-
      block and s in S-page
      add <r, s> to result
```



# Cost of Block Nested Loops

$M = 1000$  pages in  $R$   
 $p_R = 100$  tuples per page

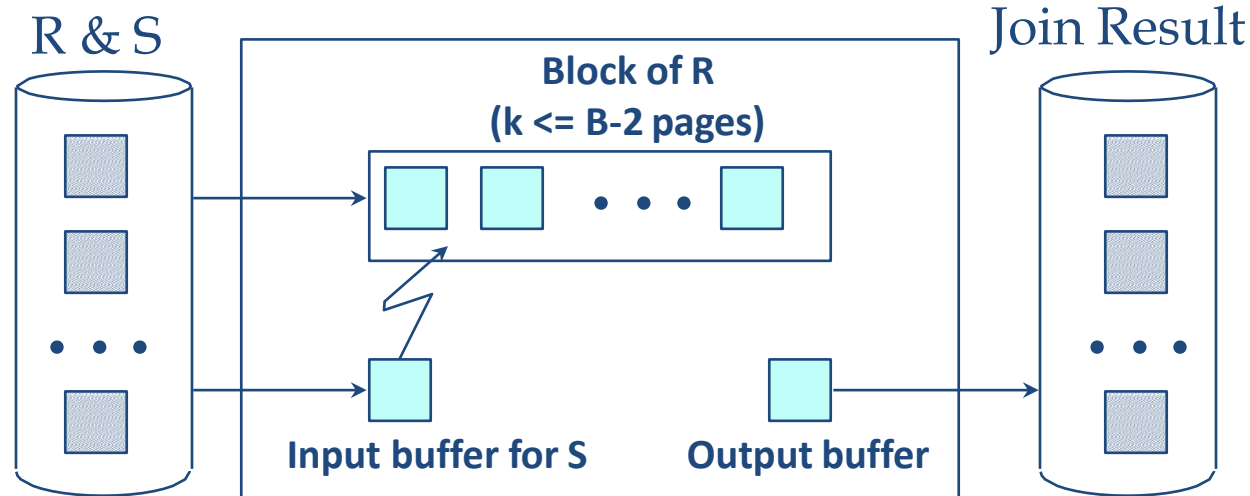
$N = 500$  pages in  $S$   
 $p_S = 80$  tuples per page

- $R$  is outer
- $B-2 = 100$ -page blocks
- How many blocks of  $R$ ? **10**
- Cost to scan  $R$ ? **1000**
- Cost to scan  $S$ ?  **$10 * 500$**
- Total Cost?  **$1000 + 5000 = 6000$**
- (check yourself)
  - If space for just 90 pages of  $R$ , we would scan  $S$  12 times, cost = 7000

```
foreach block of B-2 pages of R do
  foreach page of S do {
    for all matching in-memory tuples r in R-
      block and s in S-page
      add <r, s> to result
```

- Cost: Scan of outer + #outer blocks \* scan of inner
  - #outer blocks =  $\lceil \text{\#pages of outer relation} / \text{blocksize} \rceil$

for blocked access, it might be good to equally divide buffer pages among  $R$  and  $S$  ("seek time" less)



# Algorithms for Joins

## 2. SORT-MERGE JOINS


# Sort-Merge Join

- Sort R and S on the join column
- Then scan them to do a ``merge'' (on join col.)
- Output result tuples.

# Sort-Merge Join: 1/3

- Advance scan of R until current R-tuple  $\geq$  current S tuple
  - then advance scan of S until current S-tuple  $\geq$  current R tuple
  - do this as long as current R tuple = current S tuple


Sailors



**S**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Reserves



**R**

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin



## Sort-Merge Join: 2/3

- At this point, all R tuples with same value in  $R_i$  (*current R group*) and all S tuples with same value in  $S_j$  (*current S group*)
  - match
  - find all the equal tuples
  - output  $\langle r, s \rangle$  for all pairs of such tuples

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

**S** →

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

**R** →

WRITE TWO OUTPUT TUPLES

# Sort-Merge Join: 3/3

- Then resume scanning R and S

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

**S** →

WRITE THREE OUTPUT TUPLES

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

→ **R**

# Sort-Merge Join: 3/3

- ... and proceed till end

S	<u>sid</u>	sname	rating	age
	22	dustin	7	45.0
	28	yuppy	9	35.0
	31	lubber	8	55.5
→	44	guppy	5	35.0
	58	rusty	10	35.0

NO MATCH, CONTINUE SCANNING S →


R	<u>sid</u>	<u>bid</u>	<u>day</u>	rname
	28	103	12/4/96	guppy
	28	103	11/3/96	yuppy
	31	101	10/10/96	dustin
	31	102	10/12/96	lubber
	31	101	10/11/96	lubber
→	58	103	11/12/96	dustin

# Sort-Merge Join: 3/3

- ... and proceed till end

**S**

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0



WRITE ONE OUTPUT TUPLE

**R**

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin



# Example of Sort-Merge Join

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Typical Cost:  $O(M \log M) + O(N \log N) + (M+N)$ 
  - ignoring B (as the base of log)
  - cost of sorting R + sorting S + merging R, S
  - The cost of scanning in merge-sort,  $M+N$ , could be  $M*N$ !
    - assume the same single value of join attribute in both R and S
    - but it is extremely unlikely

# Cost of Sort-Merge Join

$M = 1000$  pages in R  
 $p_R = 100$  tuples per page

$N = 500$  pages in S  
 $p_S = 80$  tuples per page

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

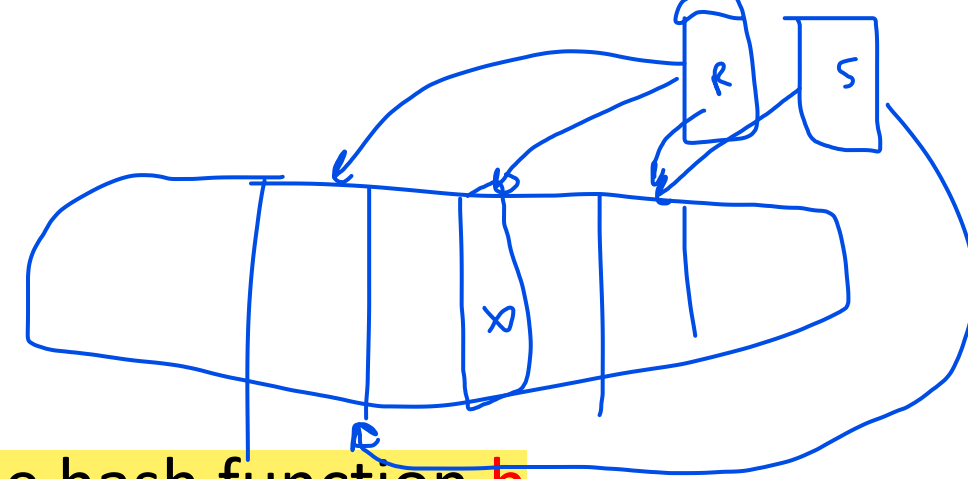
- 100 buffer pages
- Sort R:
  - (pass 0)  $1000/100 = 10$  sorted runs
  - (pass 1) merge 10 runs
  - read + write, 2 passes
  - $4 * 1000 = 4000$  I/O
- Similarly, Sort S:  $4 * 500 = 2000$  I/O
- Second merge phase of sort-merge join
  - another  $1000 + 500 = 1500$  I/O
  - assume uniform  $\sim 2.5$  matches per sid, so  $M+N$  is sufficient
- Total 7500 I/O

- Check yourself:
  - Consider #buffer pages 35, 100, 300
  - Cost of sort-merge = 7500 in all three
  - Cost of block nested 16500, 6500, 2500

# Algorithms for Joins

## 3. HASH JOINS

# Two Phases



## 1. Partition Phase

- partition R and S using the same hash function **h**

## 2. Probing Phase

- join tuples from the same partition (same  $h(..)$  value) of R and S
- tuples in different partition of  $h$  will never join
- use a “different” hash function **h2** for joining these tuples
  - (why different – see next slide first)

lw feh redundancy R feha 5 msln, w S feha 6 yeb2a el total result bkam?  $5 * 6 = 30$ .

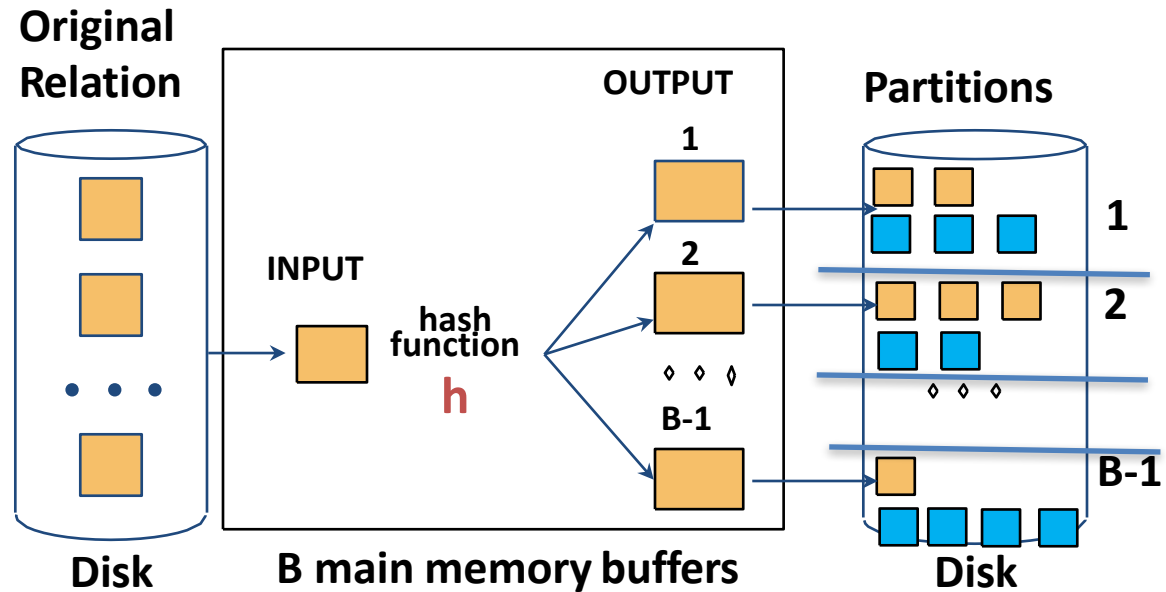
lw 7asal collision, sa3tha  
bs htl3ha output, gher keda  
m7slsh join.

el cost hyb2a  $2 * (R + S)$

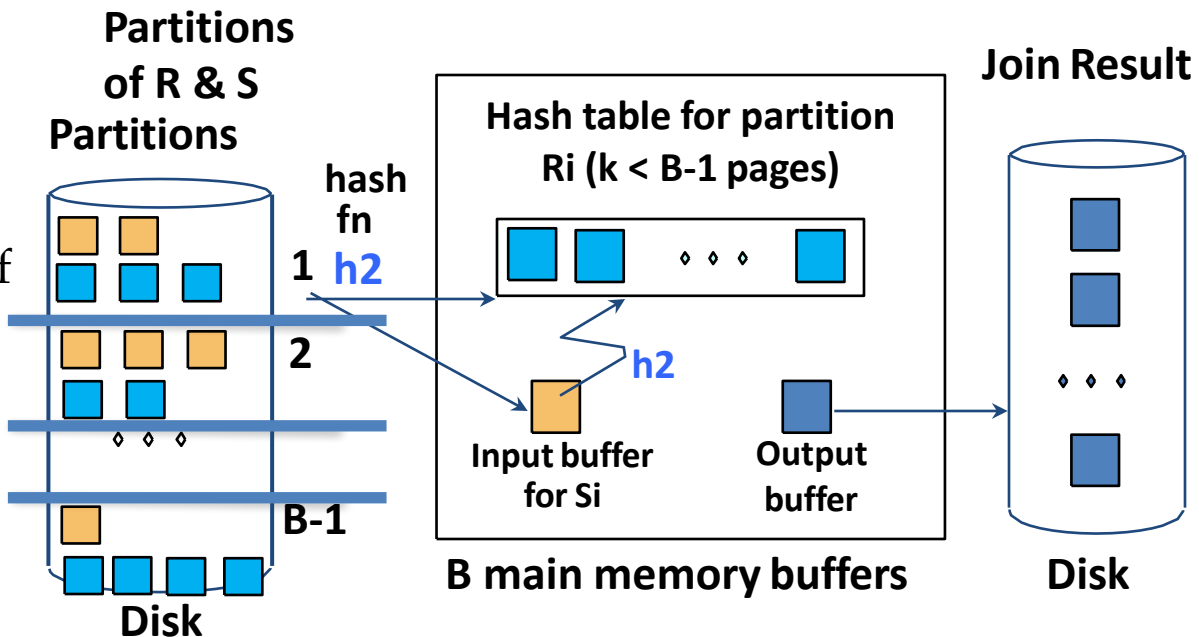


# Hash-Join

- Partition both relations using hash function  $h$
- $R$  tuples in partition  $i$  will only match  $S$  tuples in partition  $i$



- ❖ Read in a partition of  $R$ , hash it using  $h_2 (\neq h)$ .
- ❖ Scan matching partition of  $S$ , search for matches.



# Cost of Hash-Join

- In partitioning phase
  - read+write both relns;  $2(M+N)$
  - In matching phase, read both relns;  $M+N$  I/Os
  - remember – we are not counting final write
- In our running example, this is a total of 4500 I/Os
  - $3 * (1000 + 500)$
  - Compare with the previous joins

# Sort-Merge Join vs. Hash Join

- Both can have a cost of  $3(M+N)$  I/Os
  - if sort-merge gets enough buffer
- Hash join holds smaller relation in buffer-better if limited buffer
- Hash Join shown to be highly parallelizable
- Sort-Merge less sensitive to data skew
  - also result is sorted

Other operator algorithms

# Algorithms for Selection

```
SELECT *  
FROM Reserves R  
WHERE R.rname = 'Joe'
```

- No index, unsorted data
  - Scan entire relation
  - May be expensive if not many 'Joe's
- No index, sorted data (on 'rname')
  - locate the first tuple, scan all matching tuples
  - first binary search, then scan depends on matches
- B+-tree index, Hash index
  - Discussed earlier
  - Cost of accessing data entries + matching data records
  - Depends on clustered/unclustered
- More complex condition like  $\text{day} < 8/9/94 \text{ AND } \text{bid} = 5 \text{ AND } \text{sid} = 3$ 
  - Either use one index, then filter
  - Or use two indexes, then take intersection, then apply third condition
  - etc.

# Algorithms for Projection

```
SELECT DISTINCT  
      R.sid, R.bid  
FROM   Reserves R
```

- Two parts
  - Remove fields: **easy**
  - Remove duplicates (if distinct is specified): **expensive**
- Sorting-based
  - Sort, then scan adjacent tuples to remove duplicates
  - Can eliminate unwanted attributes in the first pass of merge sort
- Hash-based
  - Exactly like hash join
  - Partition only one relation in the first pass
  - Remove duplicates in the second pass
- Sort vs Hash
  - Sorting handles skew better, returns results sorted
  - Hash table may not fit in memory – sorting is more standard
- Index-only scan may work too
  - If all required attributes are part of index

# Algorithms for Set Operations

- Intersection, cross product are special cases of joins
- Union, Except
  - Sort-based
  - Hash-based
  - Very similar to joins and projection

# Algorithms for Aggregate Operations

- SUM, AVG, MIN etc.
  - again similar to previous approaches
- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan /
- With grouping:
  - Sort on group-by attributes
  - or, hash on group-by attributes
  - can combine sort/hash and aggregate
  - can do index-only scan here as well