Database Applications (15-415)

DBMS Internals- Part V
Lecture 13, March 10, 2014

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

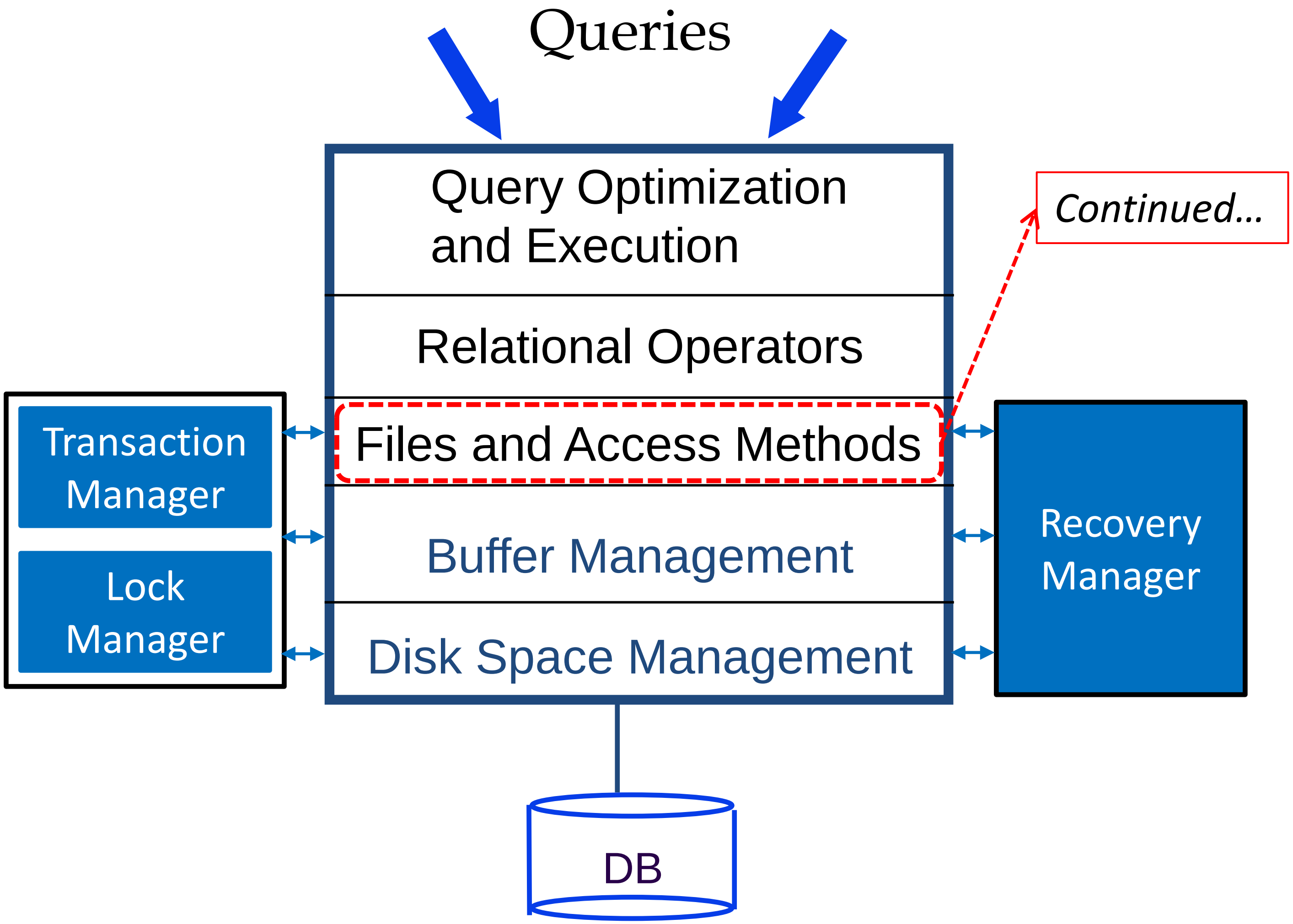
Today…

Welcome Back from Spring Break!

Today…

- Last Session:
  - DBMS Internals- Part IV
    - Tree-based (i.e., B+ Tree) and Hash-based (i.e., Extendible Hashing) indexes
- Today's Session:
  - DBMS Internals- Part V
    - Hash-based indexes (Cont'd) and External Sorting
- Announcements:
  - Project 1 grades are out
  - Midterm grades are out
  - Project 2 is due on March 13 by midnight.

DBMS Layers



Continued…

Outline

- Linear Hashing ✓
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting

Linear Hashing

- Another way of adapting gracefully to insertions and deletions (i.e., pursuing dynamic hashing) is to use Linear Hashing (LH)

- In contrast to Extendible Hashing, LH
  - Does not require a directory
  - Deals naturally with collisions
  - Offers a lot of flexibility w.r.t the timing of bucket split (allowing trading off greater overflow chains for higher average space utilization)

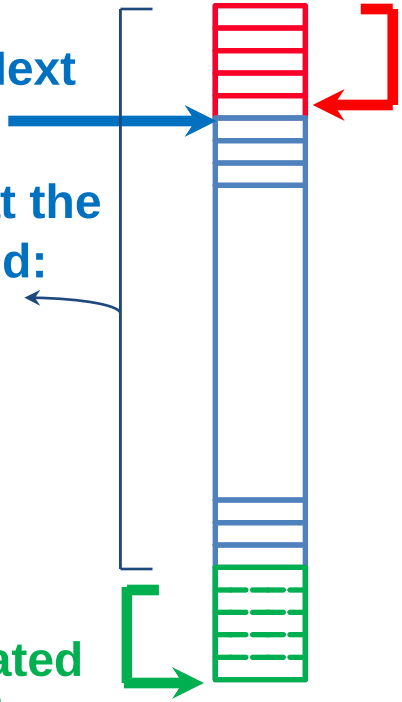
How Linear Hashing Works?

- LH uses a family of hash functions $\boldsymbol{h}_0$, $\boldsymbol{h}_1$, $\boldsymbol{h}_2$, ...
  - $\boldsymbol{h}_i(key) = \boldsymbol{h}(key) \bmod (2^i N)$; N = initial # buckets
  - ***h*** is some hash function (range is *not* 0 to N-1)
  - $\boldsymbol{h}_{i+1}$ doubles the range of $\boldsymbol{h}_i$ (this is *similar to directory doubling*)
  - If $N = 2^{d0}$, for some *d0*, $\boldsymbol{h}_i$ consists of applying ***h*** and looking at the last *di* bits, where $di = d0 + i$

How Linear Hashing Works? (Cont’d)

- LH uses overflow pages, and chooses buckets to split in a *round-robin* fashion

- Splitting proceeds in “rounds”
  - A round ends when all N_R (for round R) initial buckets are split
  - Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split
  - Current round number is referred to as *Level*



Buckets split in this round

Next

Buckets that existed at the beginning of this round: this is the range of h_{Level}

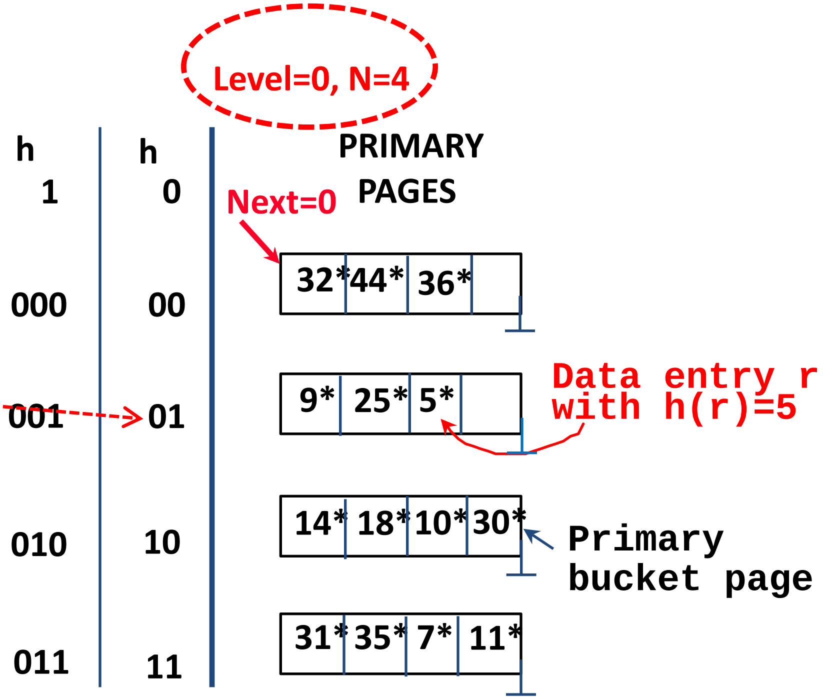
‘split image’ Buckets created in this round

Linear Hashing: Searching For Entries

- To find bucket for data entry *r*, find $\boldsymbol{h}_{Level}(r)$:
  - If $\boldsymbol{h}_{Level}(r)$ in range `*Next* to N_R’, *r* belongs there
  - Else, *r* could belong to bucket $\boldsymbol{h}_{Level}(r)$ or bucket $\boldsymbol{h}_{Level}(r) + N_R$; must apply $\boldsymbol{h}_{Level+1}(r)$ to find out
- Example: search for 5*

Level = 0 ➔ h0

5* = 101 ➔ 01



Level=0, N=4

Next=0

h_1	h_0	PRIMARY PAGES
000	00	32* 44* 36*
001	01	9* 25* 5*
010	10	14* 18* 10* 30*
011	11	31* 35* 7* 11*

Data entry r with h(r)=5

Primary bucket page

Linear Hashing: Inserting Entries

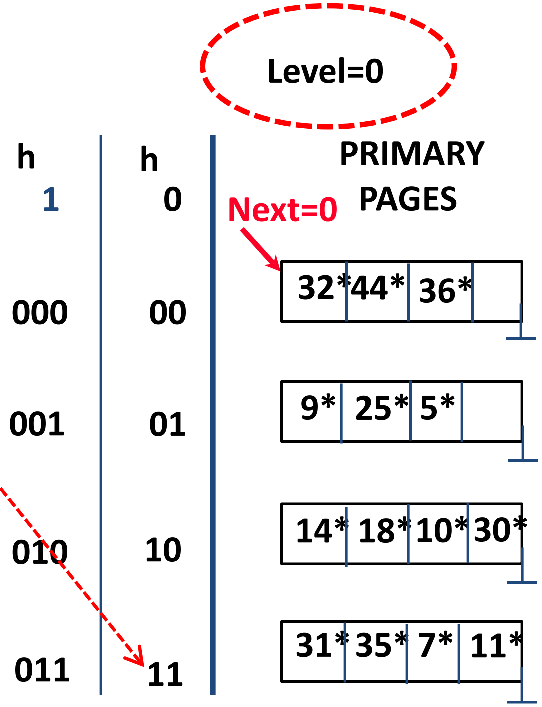
- Find bucket as in search
  - If the bucket to insert the data entry into is full:
    - Add an overflow page and insert data entry
    - (*Maybe*) Split *Next* bucket and increment *Next*

- Some points to Keep in mind:
  - Unlike Extendible Hashing, when an insert triggers a split, the bucket into which the data entry is inserted <u>is not necessarily</u> the bucket that is split
  - As in Static Hashing, an overflow page is added to store the newly inserted data entry
  - However, since the bucket to split is chosen in a round-robin fashion, eventually *all* buckets will be split

Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 → h0
43* = 101011 → 11



Level=0

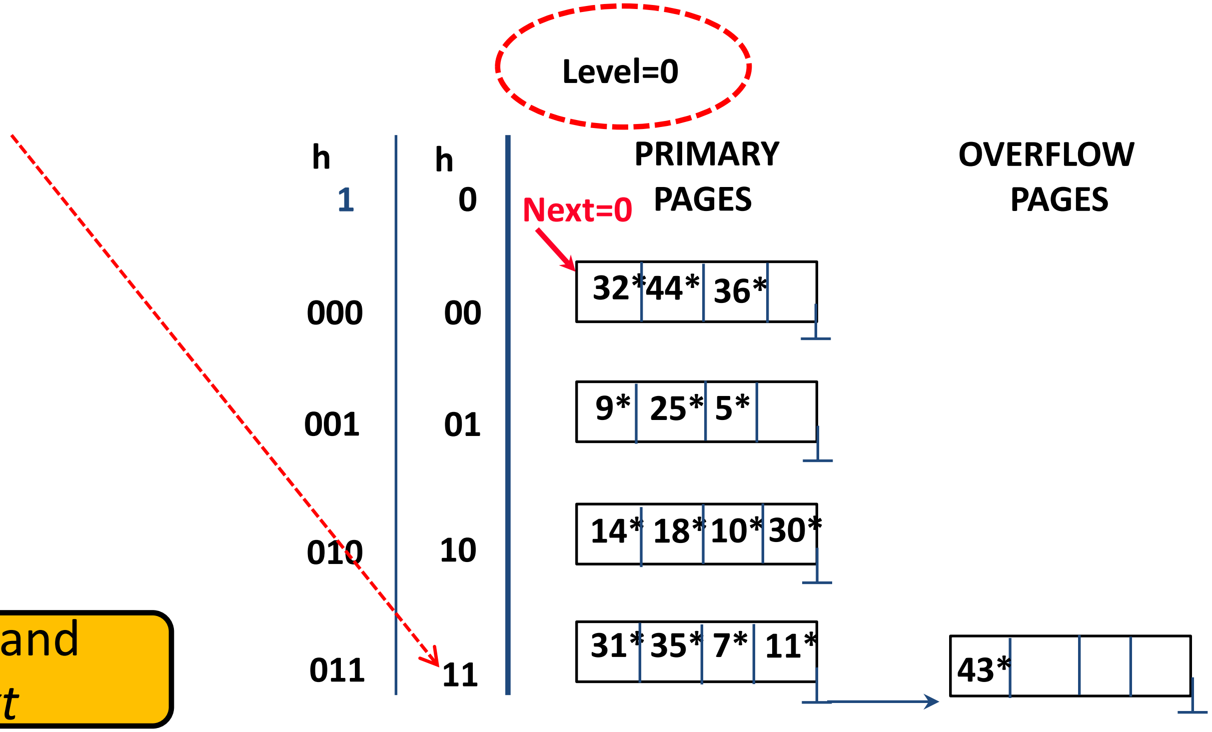
h1	h0	Next=0 PRIMARY PAGES
000	00	32* 44* 36*
001	01	9* 25* 5*
010	10	14* 18* 10* 30*
011	11	31* 35* 7* 11*

Add an overflow page and insert data entry

Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 ➔ h0
43* = 101011 ➔ 11



Level=0

Next=0

h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32* 44* 36*	
001	01	9* 25* 5*	
010	10	14* 18* 10* 30*	
011	11	31* 35* 7* 11*	43*

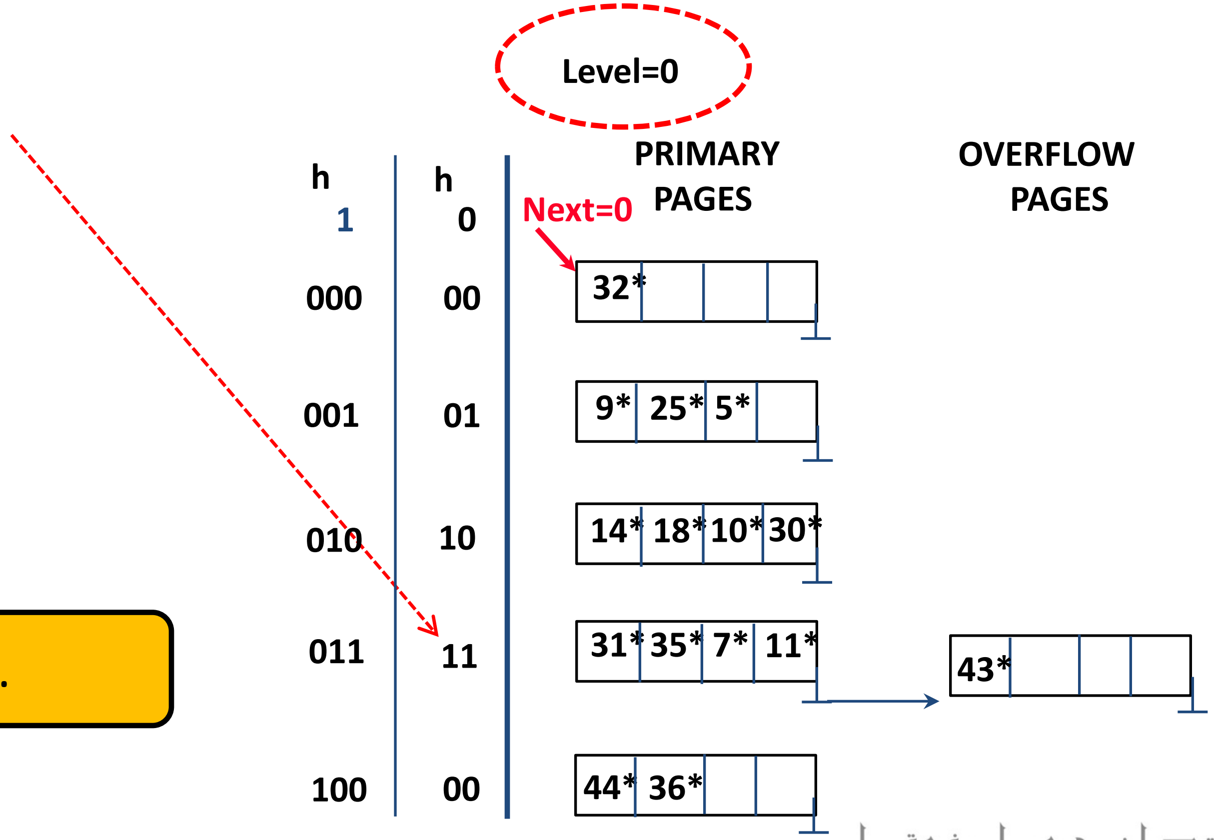
Split *Next* bucket and increment *Next*

Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 ➔ h0

43* = 101011 ➔ 11

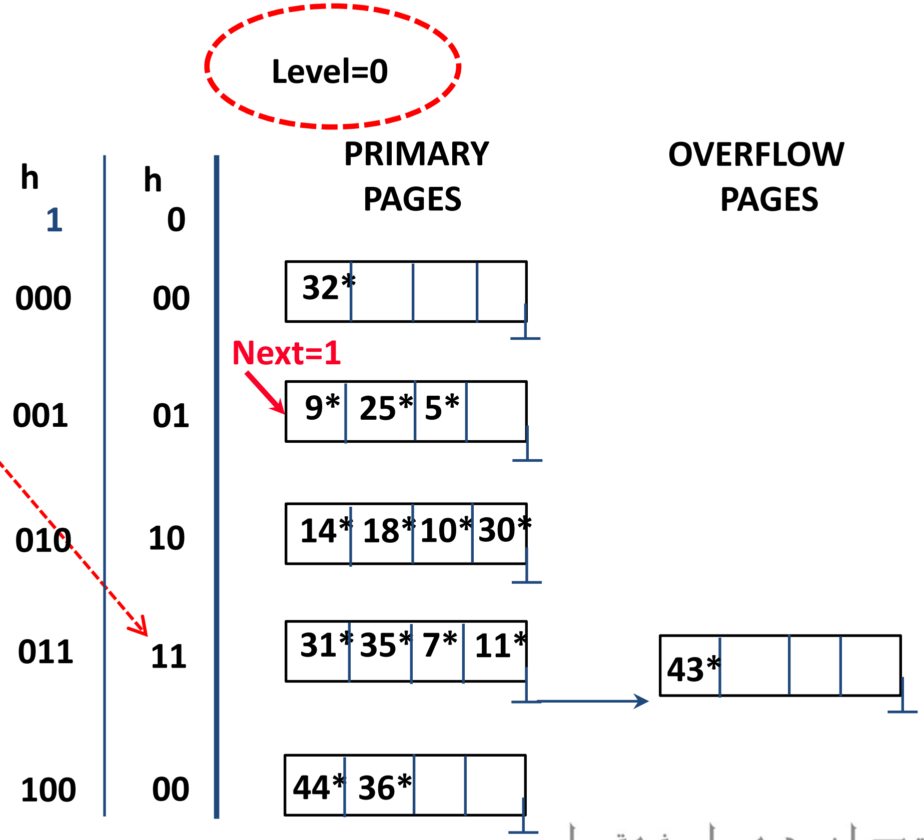


Almost there…

Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 ➔ h0
43* = 101011 ➔ 11

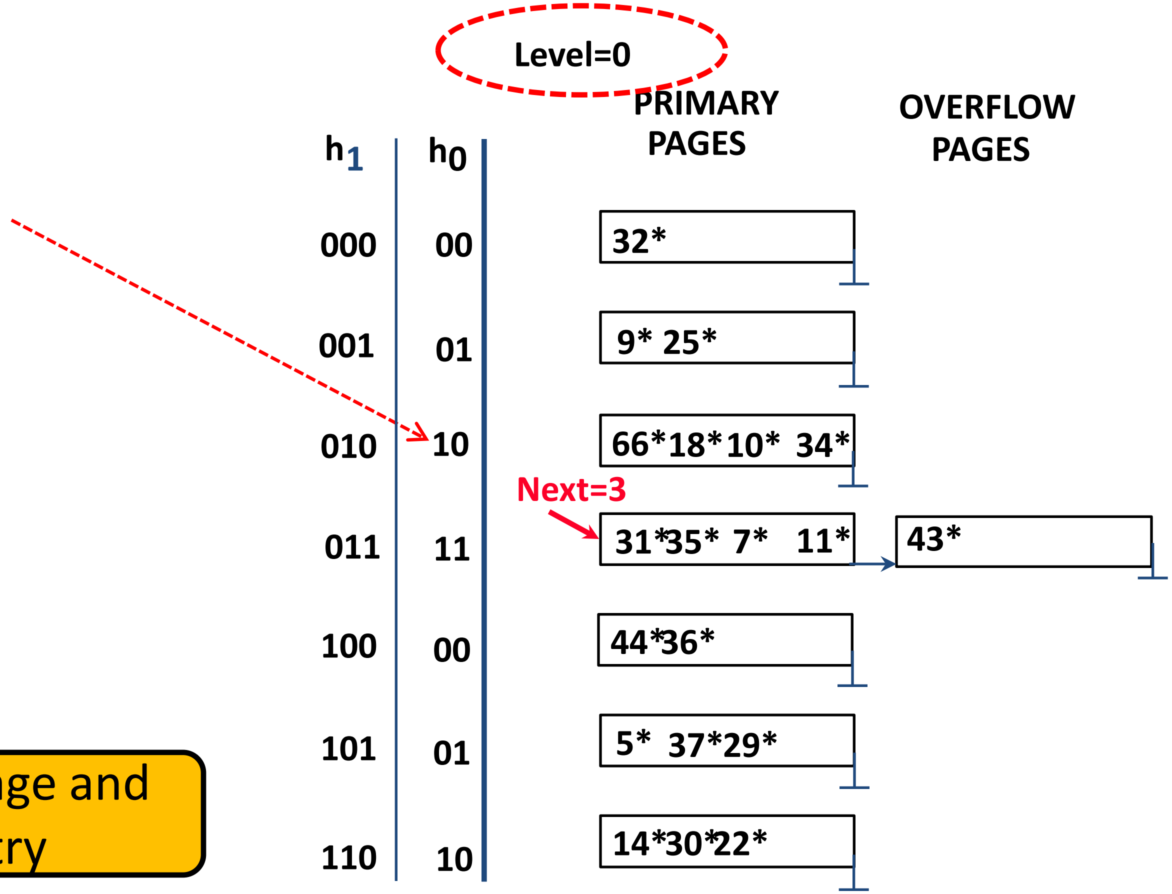


FINAL STATE!

Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 → h0
50* = 110010 → 10



Level=0

Next=3

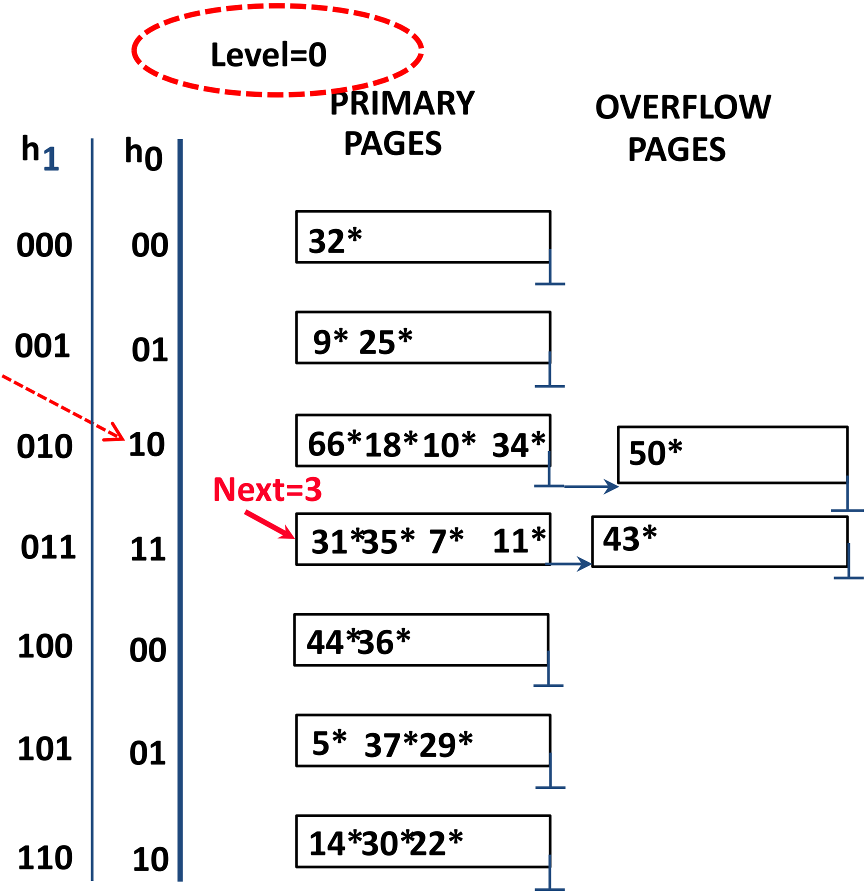
h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	
011	11	31* 35* 7* 11*	43*
100	00	44* 36*	
101	01	5* 37* 29*	
110	10	14* 30* 22*	

Add an overflow page and insert data entry

Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 ➔ h0
50* = 110010 ➔ 10

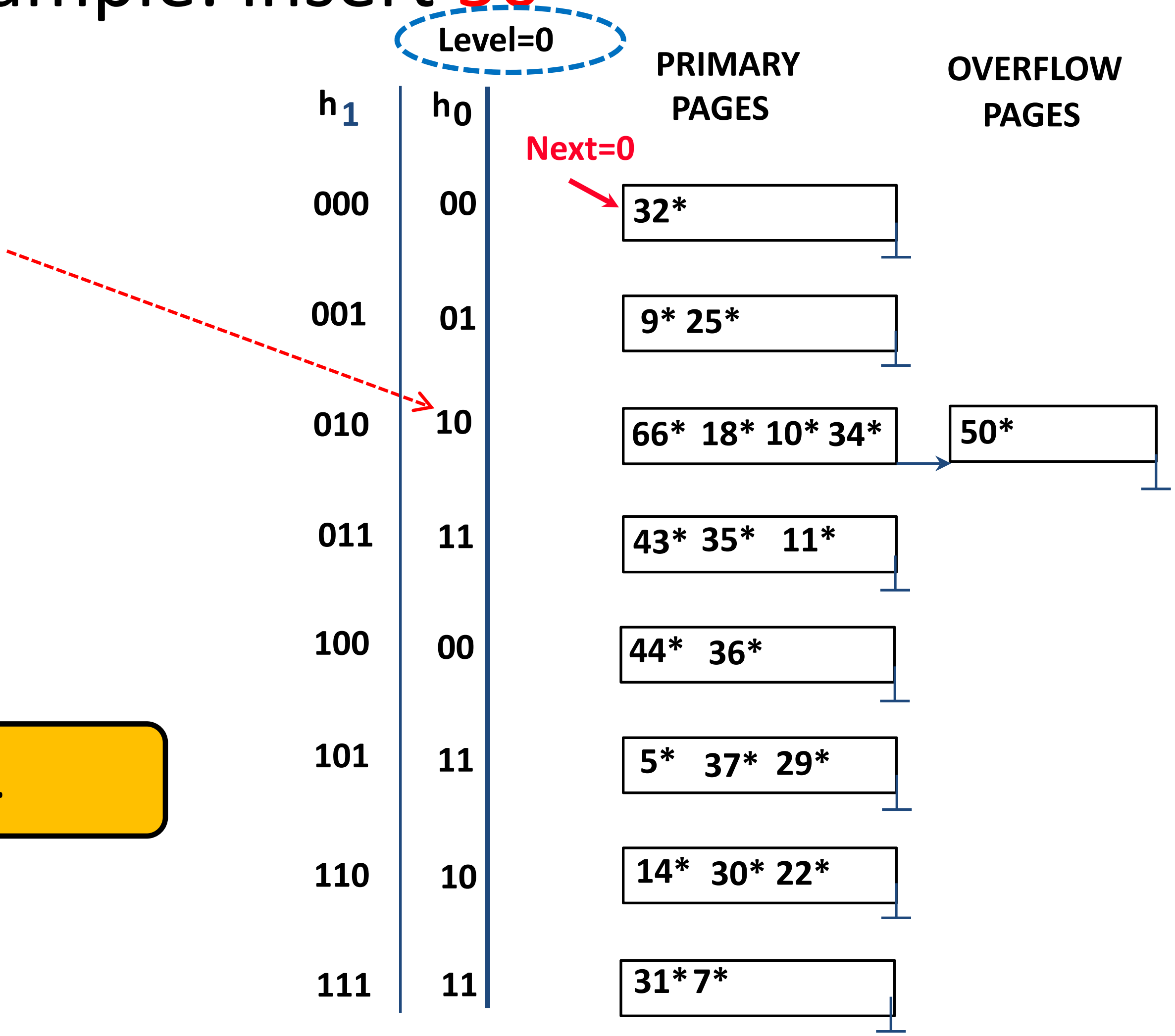


Split *Next* bucket and increment *Next*

Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 ➔ h0
50* = 110010 ➔ 10



Level=0

Next=0

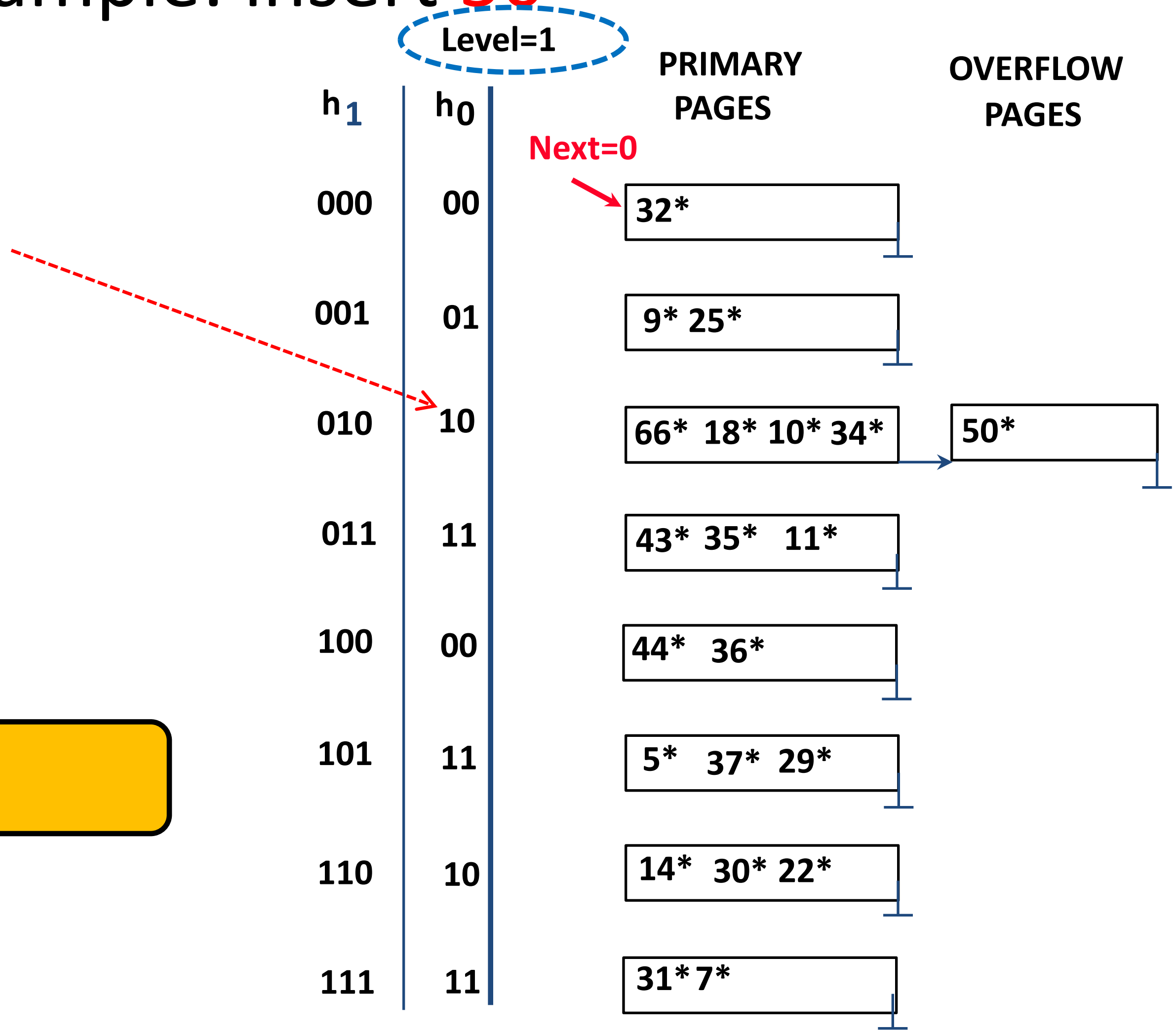
h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	50*
011	11	43* 35* 11*	
100	00	44* 36*	
101	11	5* 37* 29*	
110	10	14* 30* 22*	
111	11	31* 7*	

Almost there…

Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 ➔ h0
50* = 110010 ➔ 10



Level=1

Next=0

h1	h0	PRIMARY PAGES	OVERFLOW PAGES
000	00	32*	
001	01	9* 25*	
010	10	66* 18* 10* 34*	50*
011	11	43* 35* 11*	
100	00	44* 36*	
101	11	5* 37* 29*	
110	10	14* 30* 22*	
111	11	31* 7*	

FINAL STATE!

Linear Hashing: Deleting Entries

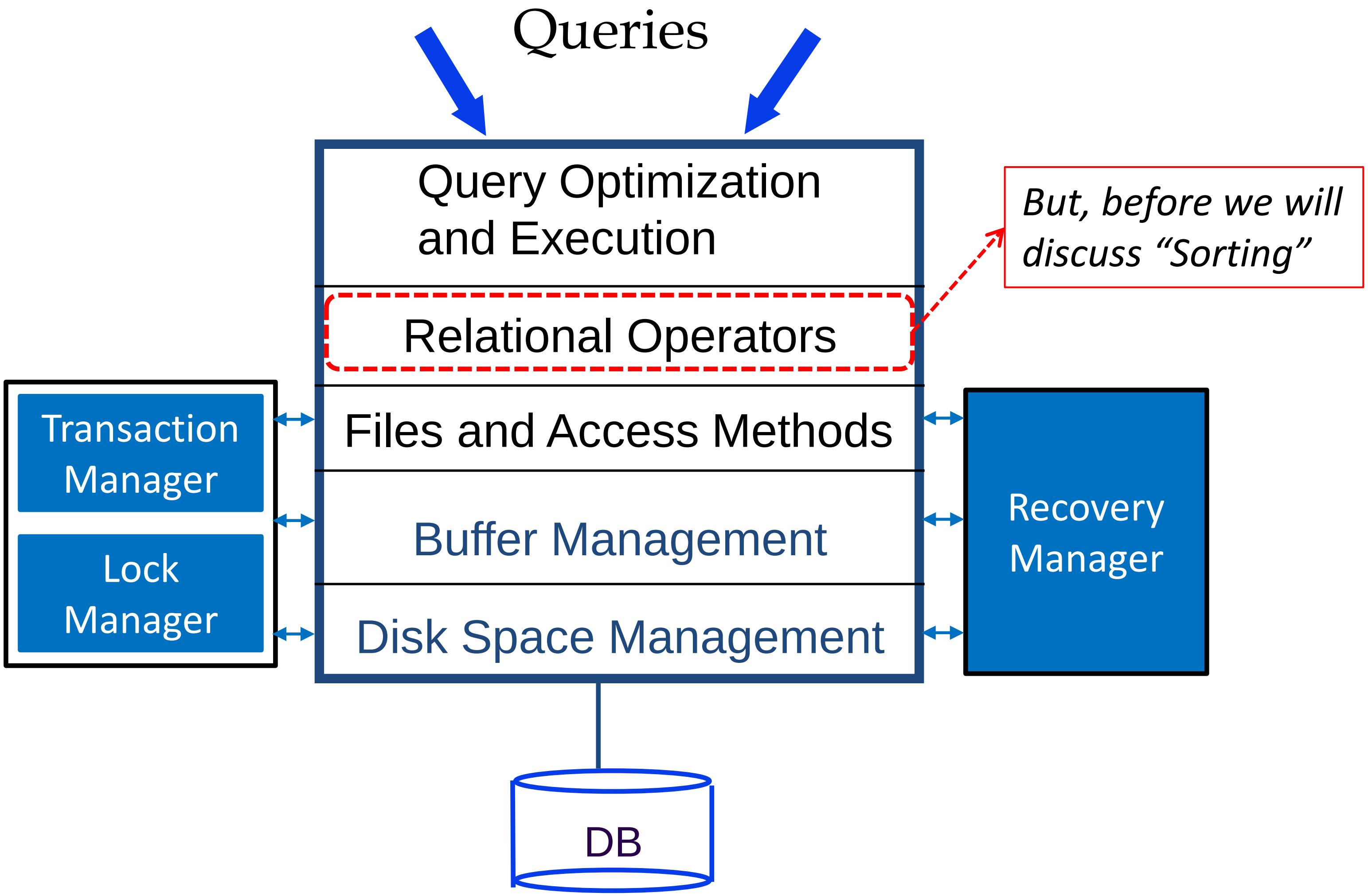
- Deletion is essentially the inverse of insertion

- If the last bucket in the file is empty, it can be removed and *Next* can be decremented

- If *Next* is zero and the last bucket becomes empty
    - *Next* is made to point to bucket ***M***/2 -1 (where ***M*** is the current number of buckets)
    - *Level* is decremented
    - The empty bucket is removed

- The insertion examples can be worked out backwards as examples of deletions!

DBMS Layers



Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

DB

Transaction Manager

Lock Manager

Recovery Manager

But, before we will discuss "Sorting"

Outline

- Linear Hashing
- Why Sorting? ✓
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting

When Does A DBMS Sort Data?

- Users may want answers in some order
  - **SELECT FROM** student ***ORDER BY*** name
  - **SELECT** S.rating, **MIN** (S.age) **FROM** Sailors S ***GROUP BY*** S.rating
- *Bulk loading* a B+ tree index involves sorting
- Sorting is useful in eliminating duplicates records
- The *Sort-Merge* Join algorithm involves sorting (*next session!*)

Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting ✓
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting

In-Memory vs. External Sorting

- Assume we want to sort 60GB of data on a machine with only 8GB of RAM
    - In-Memory Sort (e.g., Quicksort) ?
        - Yes, but data do not fit in memory
        - What about relying on virtual memory?
    - In this case, external sorting is needed
        - In-memory sorting is *orthogonal* to external sorting!

Outline

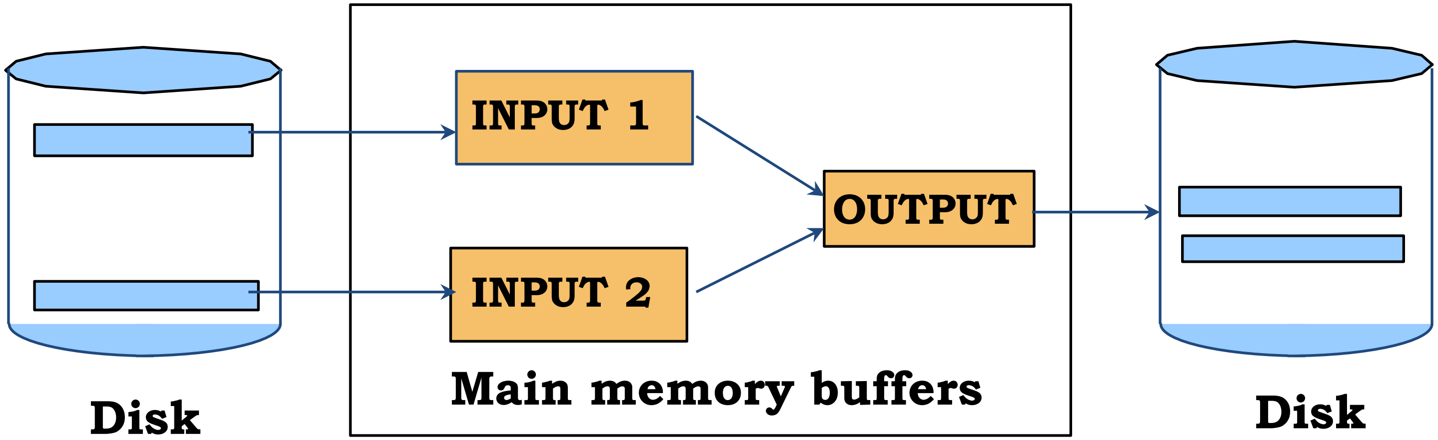
- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting ✓
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting

A Simple Two-Way Merge Sort

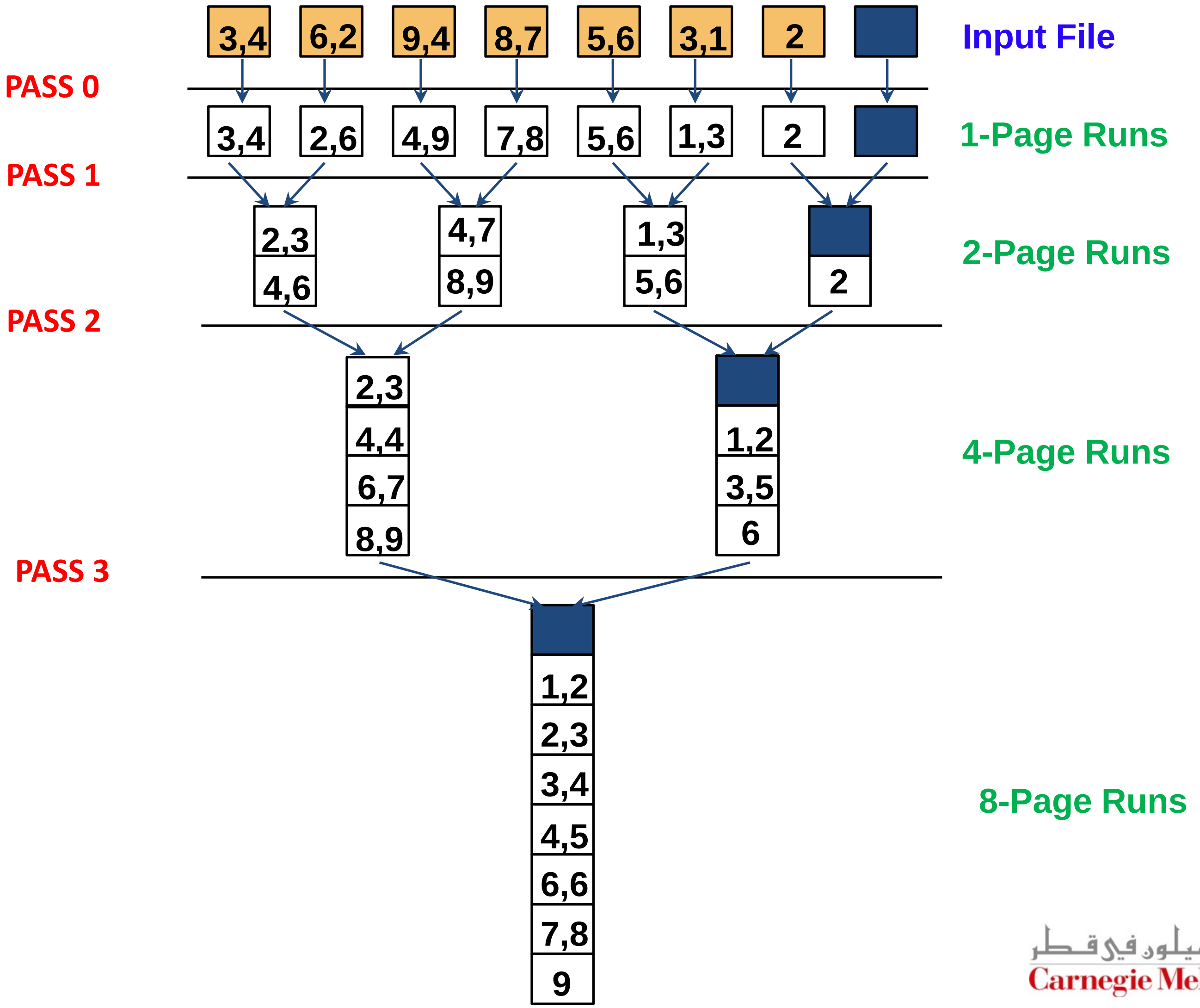
- IDEA: Sort sub-files that can fit in memory and merge
- Let us refer to each sorted sub-file as a *run*
- Algorithm:
  - Pass 1: Read a page into memory, sort it, write it
    - 1-page runs are produced
  - Passes 2, 3, etc.,: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left

A Simple Two-Way Merge Sort

- Algorithm:
  - Pass 1: Read a page into memory, sort it, write it
    - How many buffer pages are needed? ONE
  - Passes 2, 3, etc.,: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left
    - How many buffer pages are needed? THREE



2-Way Merge Sort: An Example



3,4 6,2 9,4 8,7 5,6 3,1 2

Input File

PASS 0

3,4 2,6 4,9 7,8 5,6 1,3 2

1-Page Runs

PASS 1

2,3 4,6 | 4,7 8,9 | 1,3 5,6 | 2

2-Page Runs

PASS 2

2,3 4,4 6,7 8,9 | 1,2 3,5 6

4-Page Runs

PASS 3

1,2 2,3 3,4 4,5 6,6 7,8 9

8-Page Runs

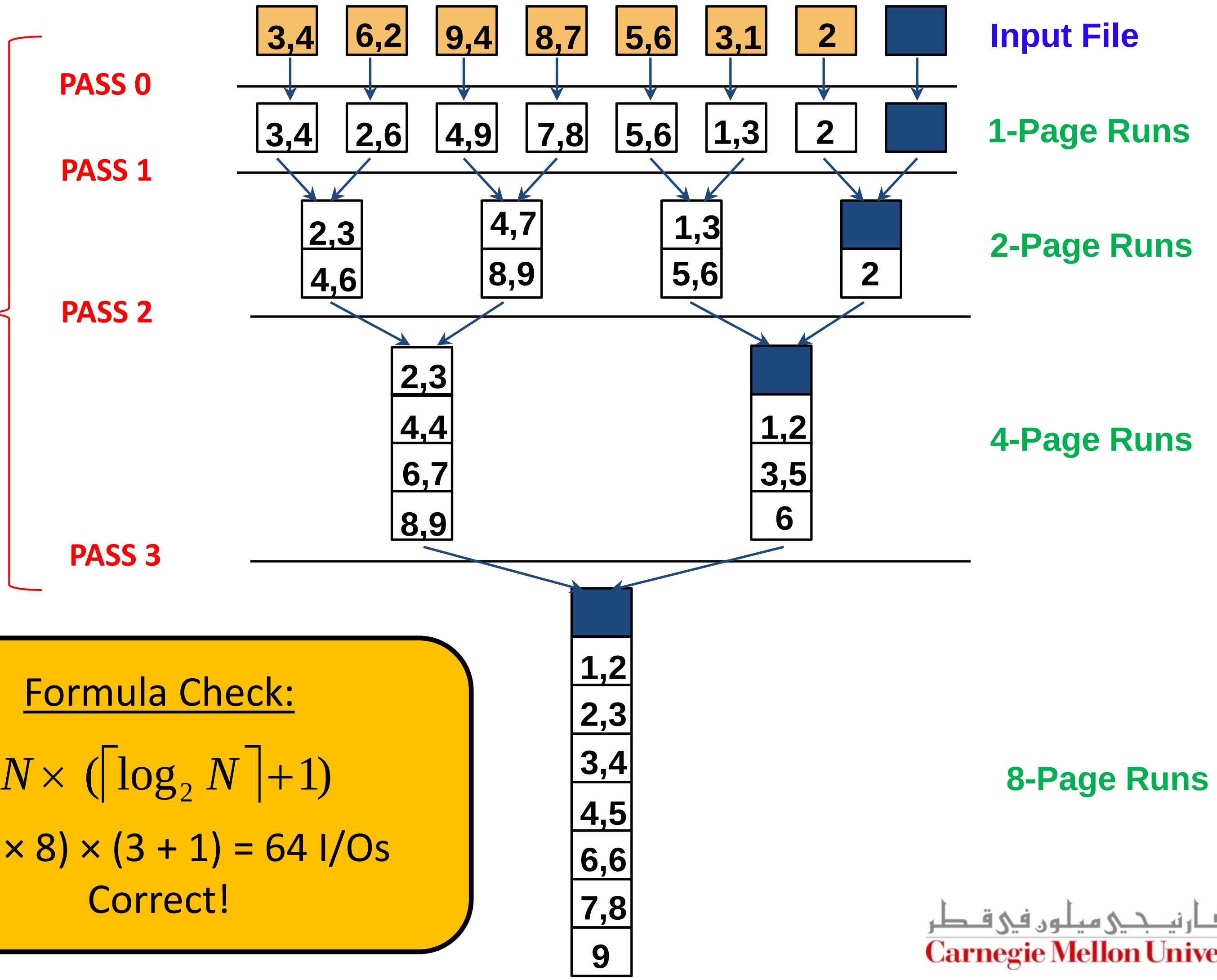
2-Way Merge Sort: I/O Cost Analysis

- If the number of pages in the input file is 2^k
  - How many runs are produced in pass 0 and of what size?
    - 2^k 1-page runs
  - How many runs are produced in pass 1 and of what size?
    - 2^{k-1} 2-page runs
  - How many runs are produced in pass 2 and of what size?
    - 2^{k-2} 4-page runs
  - How many runs are produced in pass k and of what size?
    - 2^{k-k} 2^k-page runs (or 1 run of size 2^k)
  - For *N* number of pages, how many passes are incurred?
    - $\boxed{\lceil \log_2 N \rceil + 1}$
  - How many pages do we read and write in each pass?
    - 2*N*
  - *What is the overall cost?*
    - $\boxed{2N \times (\lceil \log_2 N \rceil + 1)}$

2-Way Merge Sort: An Example

$\lceil \log_2 8 \rceil + 1$

= 4 passes



Formula Check:

$$2N \times (\lceil \log_2 N \rceil + 1)$$

= (2 × 8) × (3 + 1) = 64 I/Os

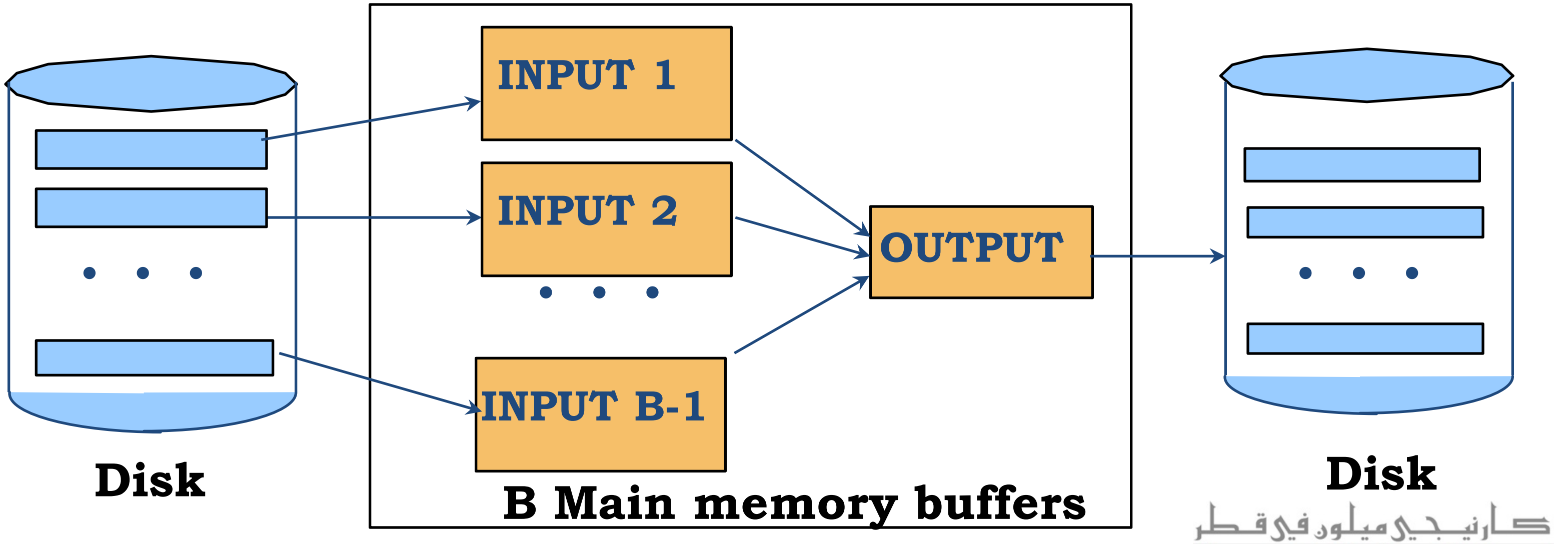
Correct!

Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting ✓
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting

B-Way Merge Sort

- How can we sort a file with *N* pages using ***B*** buffer pages?
  - Pass 0: use *B* buffer pages
    - This will produce $\lceil N / B \rceil$ sorted B-page runs
  - Pass 2, …, etc.: merge *B-1* runs



B-Way Merge Sort: I/O Cost Analysis

- I/O cost = 2N × Number of passes

- Number of passes = $1+\lceil \log_{B-1}\lceil N / B \rceil \rceil$

- Assume the previous example (i.e., 8 pages), *but* using 5 buffer pages (instead of 2)
    - I/O cost = 32 (*as opposed to 64*)

- Therefore, increasing the number of buffer pages minimizes the number of passes and accordingly the I/O cost!

Number of Passes of B-Way Sort

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

High Fan-in during merging is crucial!

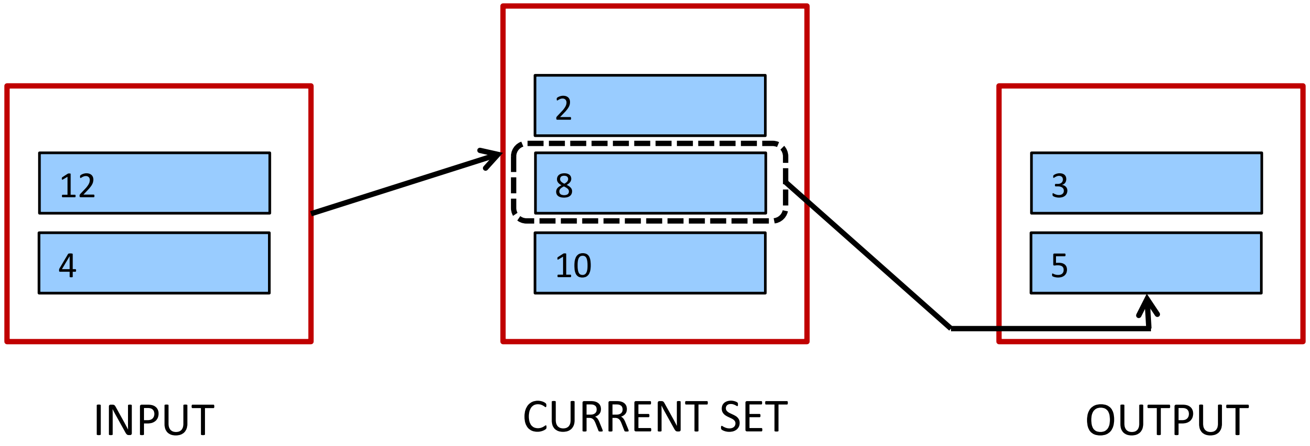
How else can we minimize I/O cost?

Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering ✓
- Using B+ Trees for External Sorting

Replacement Sort

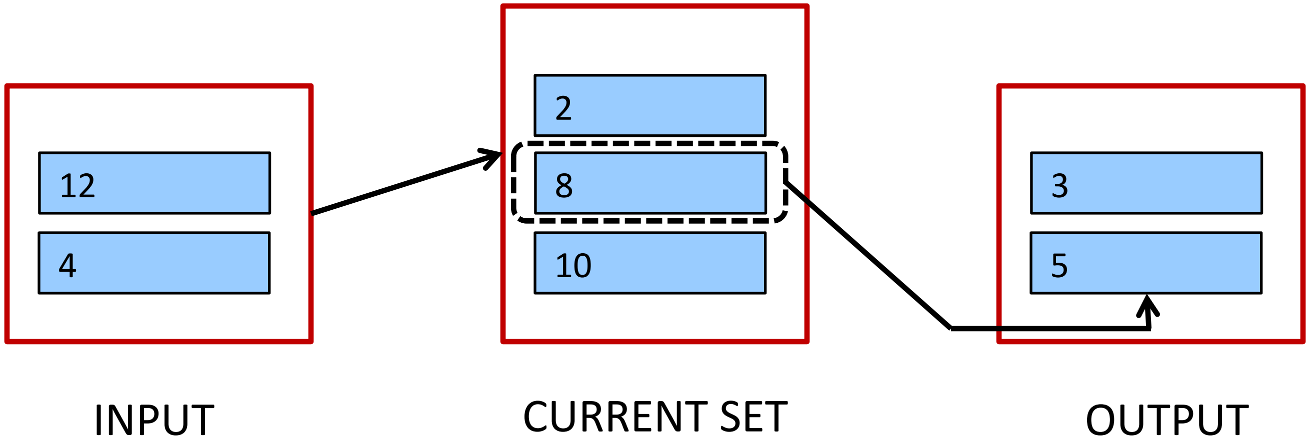
- With a more aggressive implementation of B-way sort, we can write out runs of ~2×B internally sorted pages
  - This is referred to as replacement sort



IDEA: Pick the tuple in the *current set* with the smallest value that is greater than the largest value in the *output buffer* and append it to the *output buffer*

Replacement Sort

- With a more aggressive implementation of B-way sort, we can write out runs of ~2×B internally sorted pages
    - This is referred to as replacement sort



When do we terminate the current *run* and start a new one?

Blocked I/O and Double Buffering

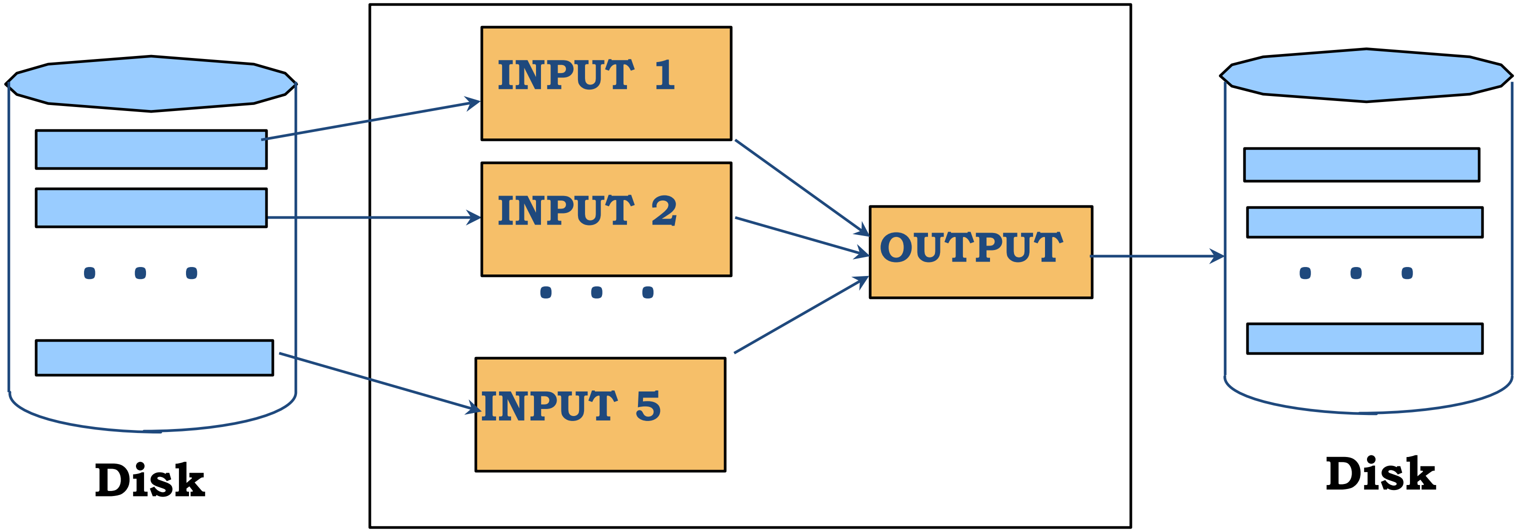
- So far, we assumed random disk access

- Would cost change if we assume that reads and writes are done sequentially?
  - Yes

- How can we incorporate this fact into our cost model?
  - Use bigger units (this is referred to as Blocked I/O)
  - Mask I/O delays through pre-fetching (this is referred to as double buffering)

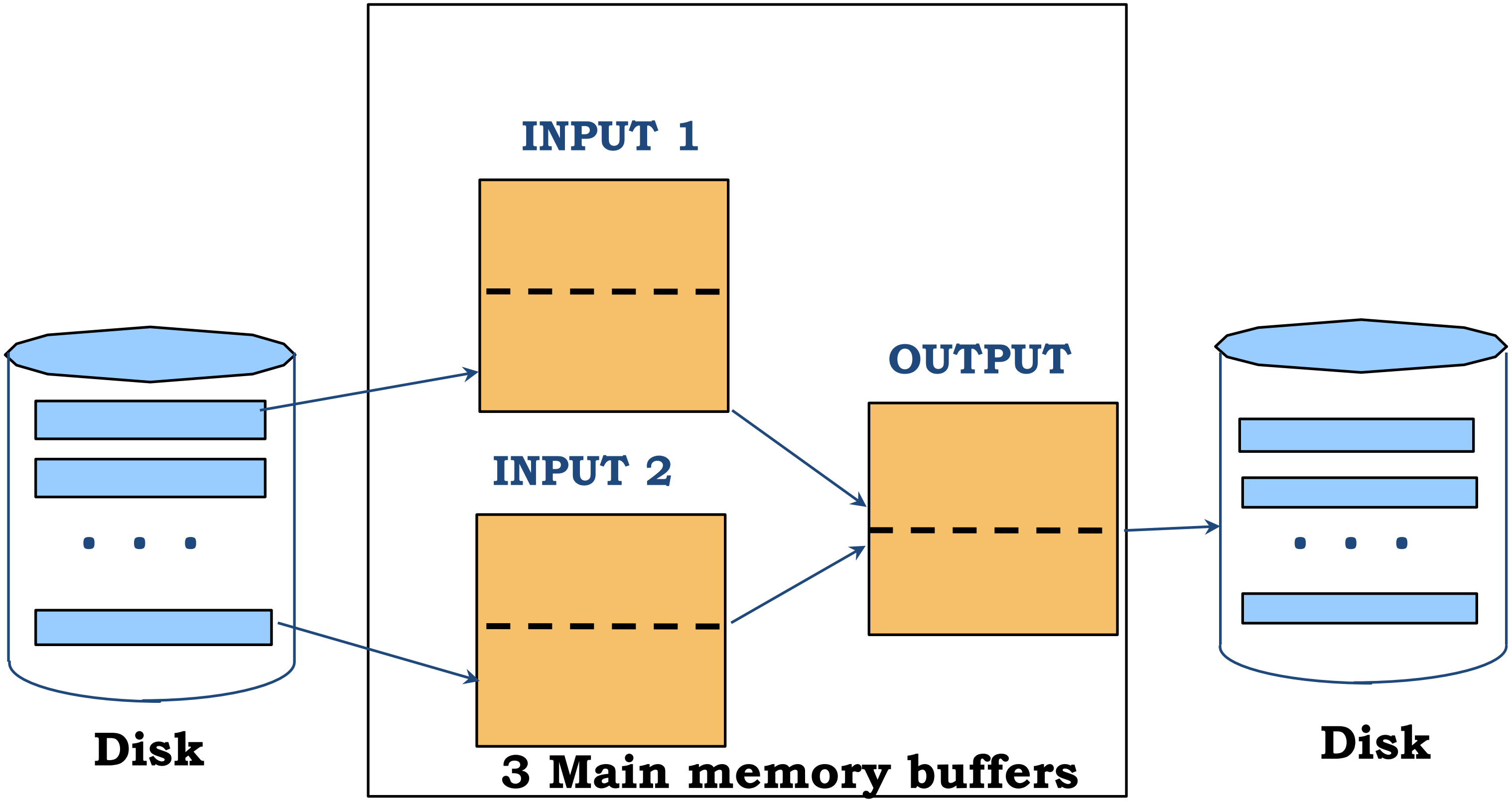
Blocked I/O

- Normally, we go with '*B*' buffers of size (say) 1 page



Blocked I/O

- Normally, we go with '*B*' buffers of size (say) 1 page
- INSTEAD: let us go with *B/b* buffers, of size '*b*' pages



Blocked I/O

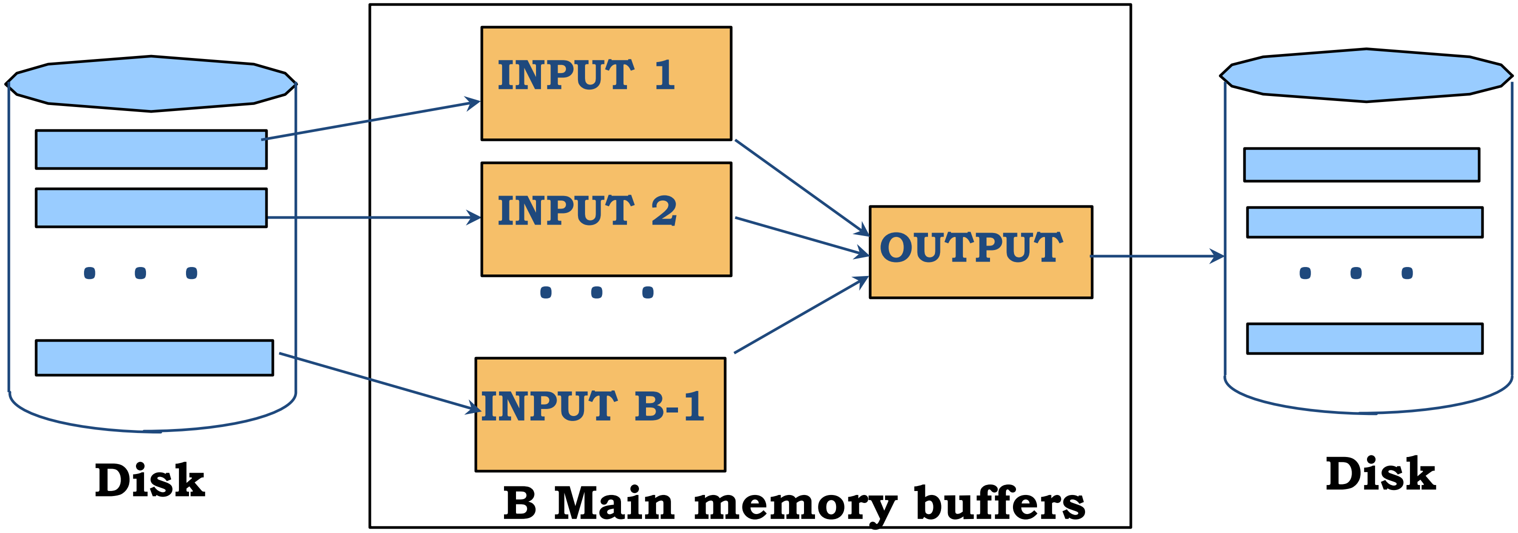
- Normally, we go with '*B*' buffers of size (say) 1 page
- INSTEAD: let us go with *B/b* buffers, of size '*b*' pages

- What is the main advantage?
  - Fewer random accesses (as some of the page will be arranged sequentially!)

- What is the main disadvantage?
  - Smaller fan-out and accordingly larger number of passes!

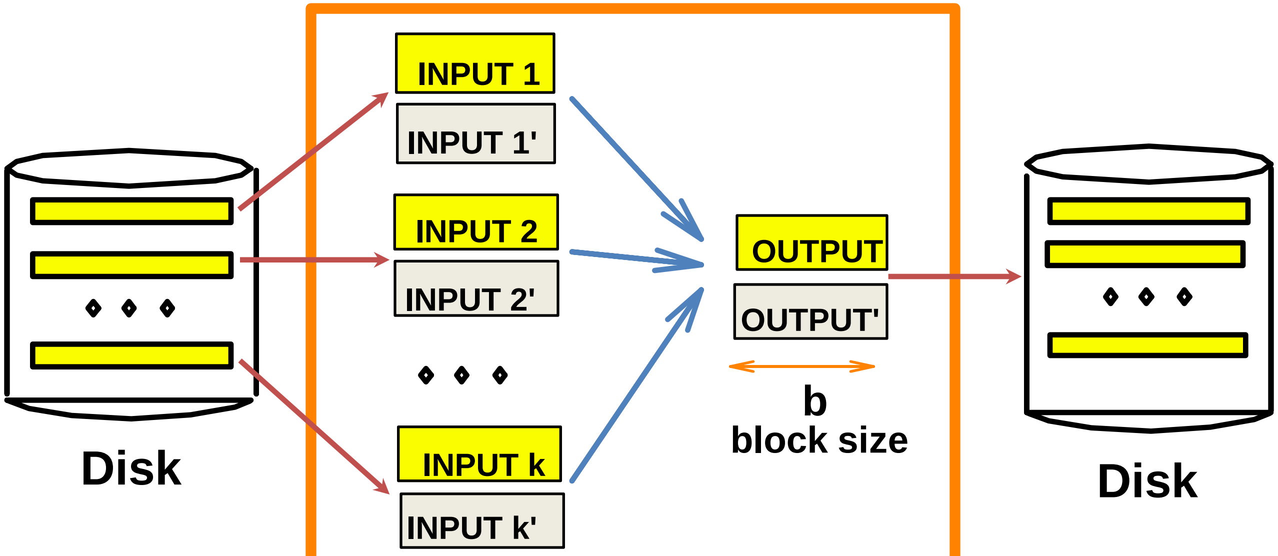
Double Buffering

- Normally, when, say ‘INPUT1’ is exhausted
  - We issue a ‘read’ request and
  - We wait …



Double Buffering

- INSTEAD: *pre-fetch* INPUT1’ into a `*shadow block*’
  - When INPUT1 is exhausted, issue a ‘read’
  - BUT, also proceed with INPUT1’
  - Thus, the CPU can never go idle!



B main memory buffers, k-way merge

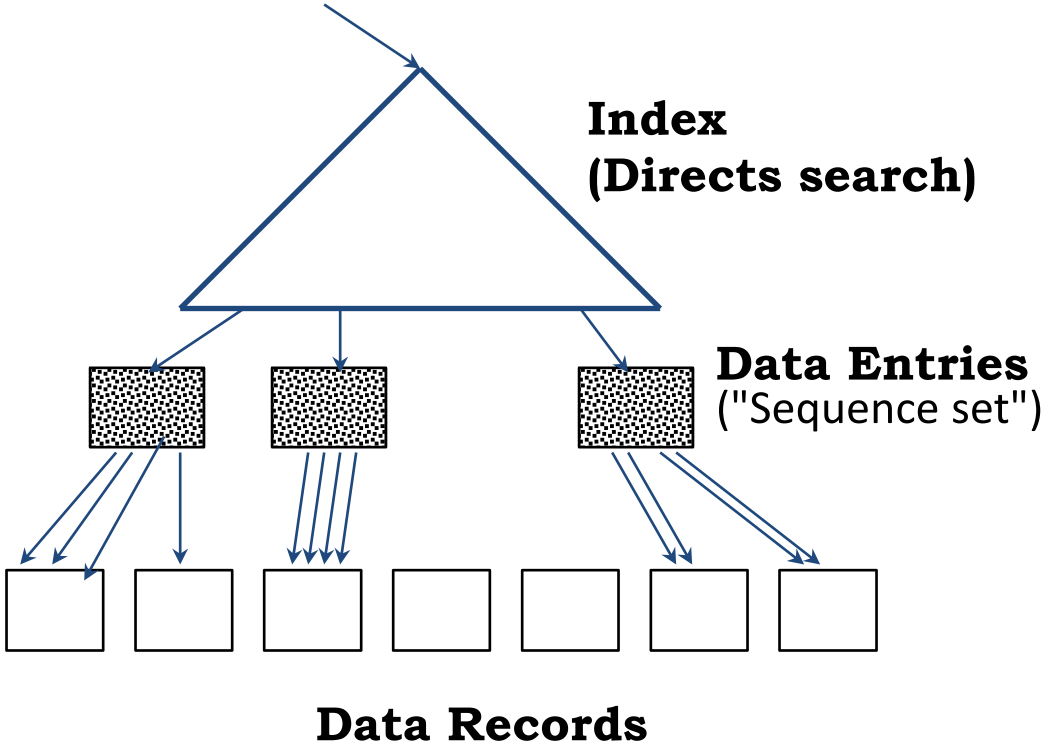
Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
- Using B+ Trees for External Sorting ✓

Using B+ Trees for External Sorting

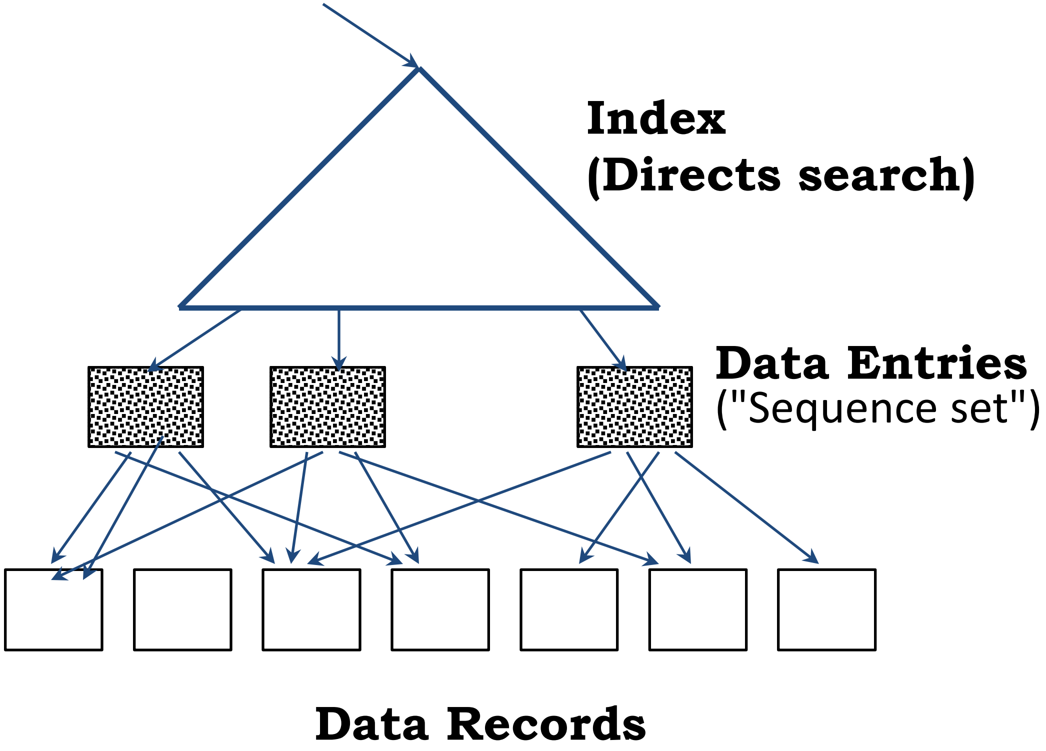
- Scenario: the relation to be sorted has a B+ tree index on its primary key
- IDEA: retrieve records in order by traversing leaf pages
- Is this a good idea?
  - What if the B+ tree is clustered?
  - What if the B+ tree in un-clustered?
  - What about different indexing alternatives?

Using Clustered B+ Trees for Sorting



- What if Alternative (1) is in use?
  - Cost: root to the left-most leaf, then retrieve all leaf pages
- What if Alternative (2) or (3) is in use?
  - Cost: root to the left-most leaf, then fetch each page just once

Using Un-clustered B+ Trees for Sorting



- What if Alternative (1) is in use?
  - Cost: root to the left-most leaf, then retrieve all leaf pages
- What if Alternative (2) or (3) is in use?
  - Cost: root to the left-most leaf, then fetch pages
    - Worst-case: 1 I/O per each data record!

Using B+ Trees for External Sorting

- Scenario: the relation to be sorted has a B+ tree index on its primary key

- IDEA: Can retrieve records in order by traversing leaf pages

- Is this a good idea?
    - What if the B+ tree is clustered?
        - Good idea!
    - What if the B+ tree in un-clustered?
        - Could be a very bad idea!

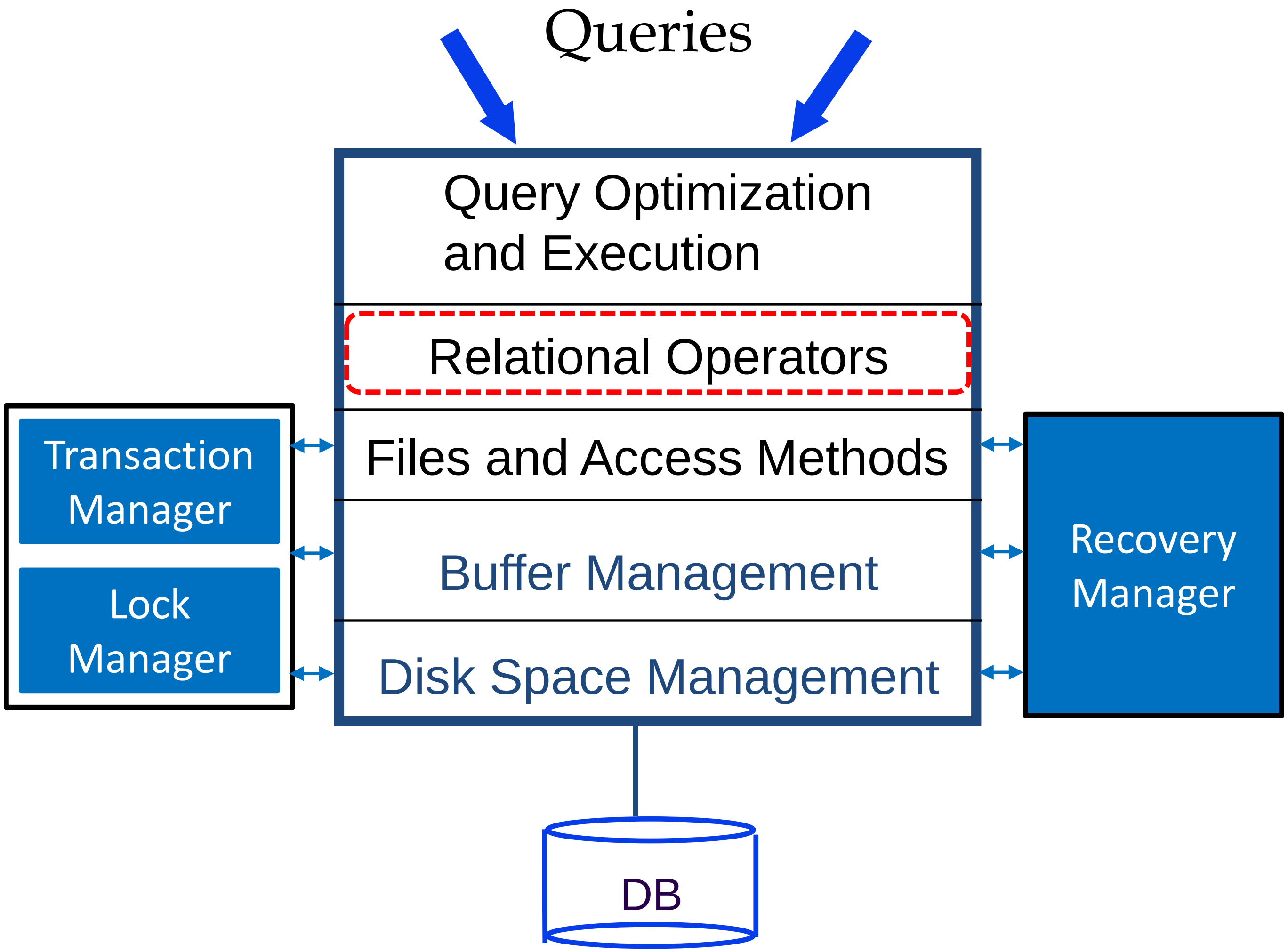
Summary

- External sorting is important; a DBMS may dedicate part of its buffer pool for sorting!

- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs
  - # of runs merged at a time depends on ***B,*** and ***block size***
    - Larger block size means less I/O cost per page
    - Larger block size means smaller # runs merged
  - In practice, # of runs is rarely more than 2 or 3

- Clustered B+ tree is good for sorting; un-clustered tree is usually very bad!

Next Class


Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB