Carnegie Mellon University

09 Sorting & Aggregations

Intro to Database Systems
15-445/15-645
Fall 2021

Andrew Crotty
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #2 is due Sunday, Oct 3rd @ 11:59pm

Project #2 is due Sunday, Oct 17th @ 11:59pm
→ **Q&A Session** on Thursday, Sept 30th from 5-6pm
→ See the Piazza post for details

Mid-Term Exam is Wednesday, Oct 13th
→ During regular class time from 3:05-4:25pm
→ More details next week…

B+TREE CONCURRENCY CONTROL

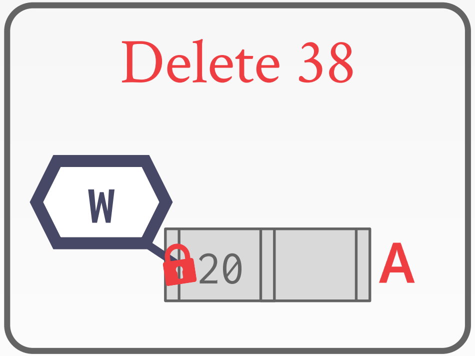
We want to allow multiple threads to read and update a B+Tree at the same time.

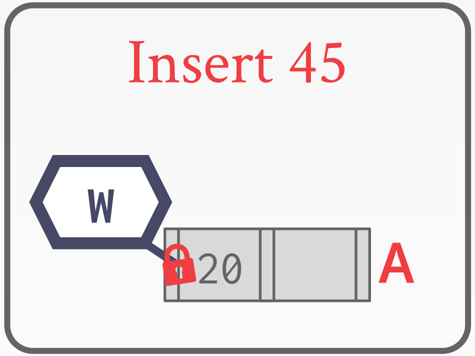
We need to protect against two types of problems:

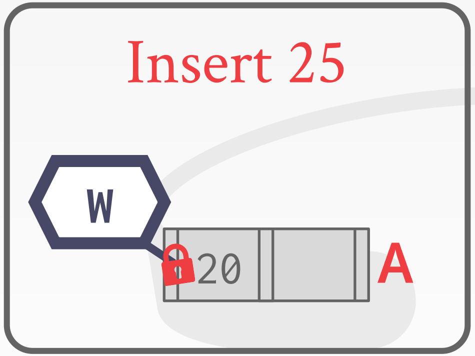
- → Threads trying to modify the contents of a node at the same time.
- → One thread traversing the tree while another thread splits/merges nodes.

OBSERVATION

What was the first step that all the update examples did on the B+Tree?







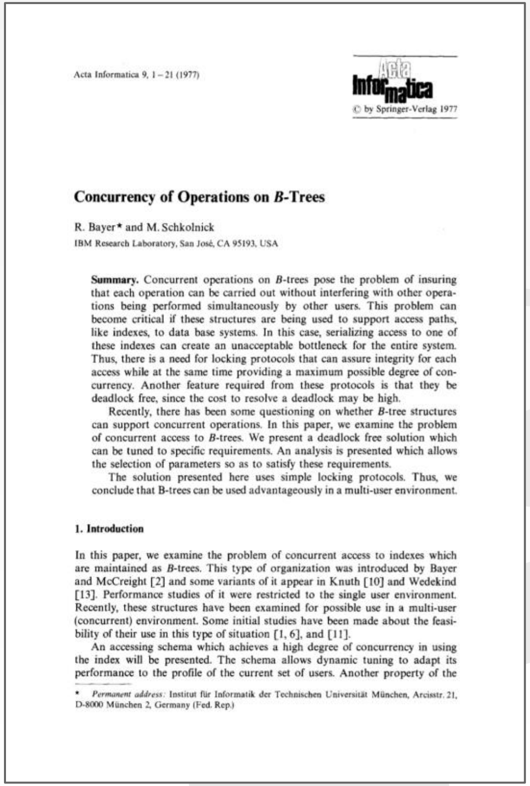
Taking a write latch on the root every time becomes a bottleneck with higher concurrency.

BETTER LATCHING ALGORITHM

Most modifications to a B+Tree will not require a split or merge.

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches.

If you guess wrong, repeat traversal with the pessimistic algorithm.



Acta Informatica 9, 1–21 (1977)

Acta Informatica
© by Springer-Verlag 1977

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick
IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

BETTER LATCHING ALGORITHM

Search: Same as before.

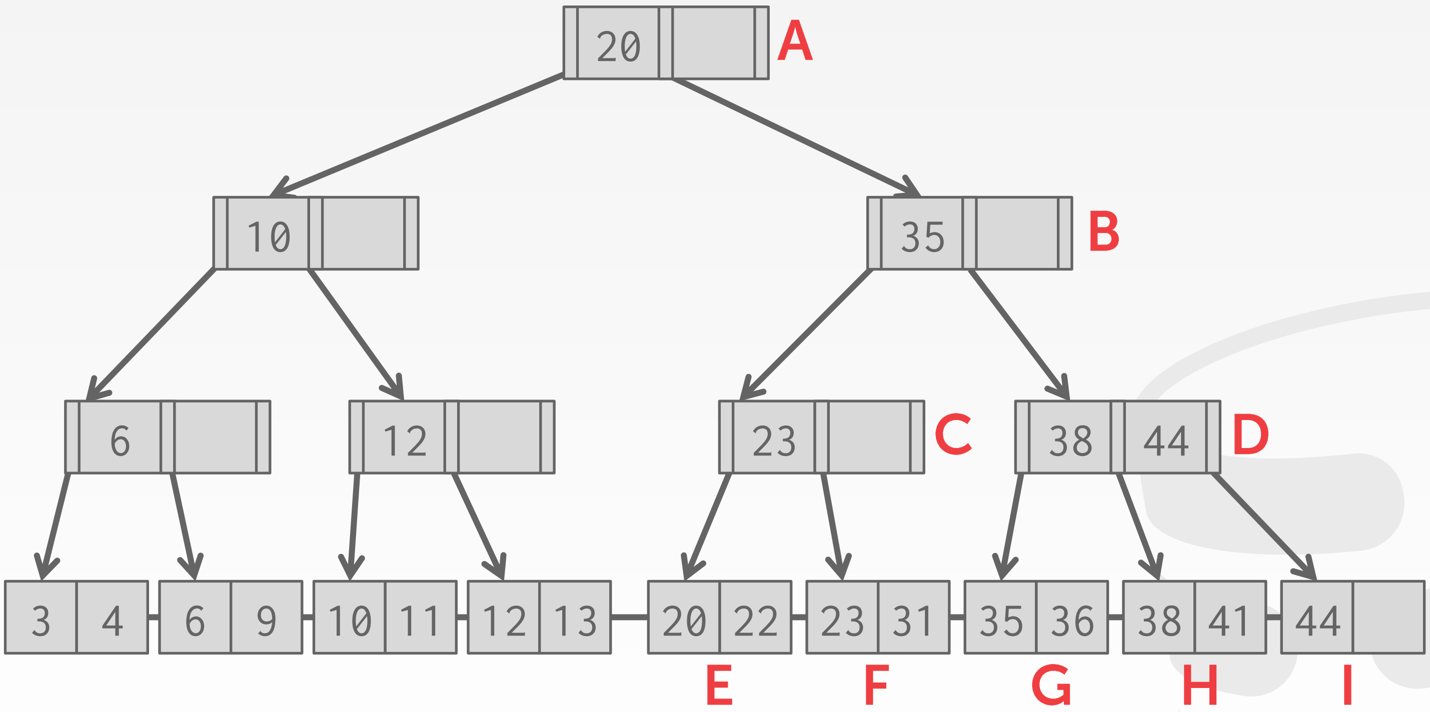
Insert/Delete:

→ Set latches as if for search, get to leaf, and set **W** latch on leaf.

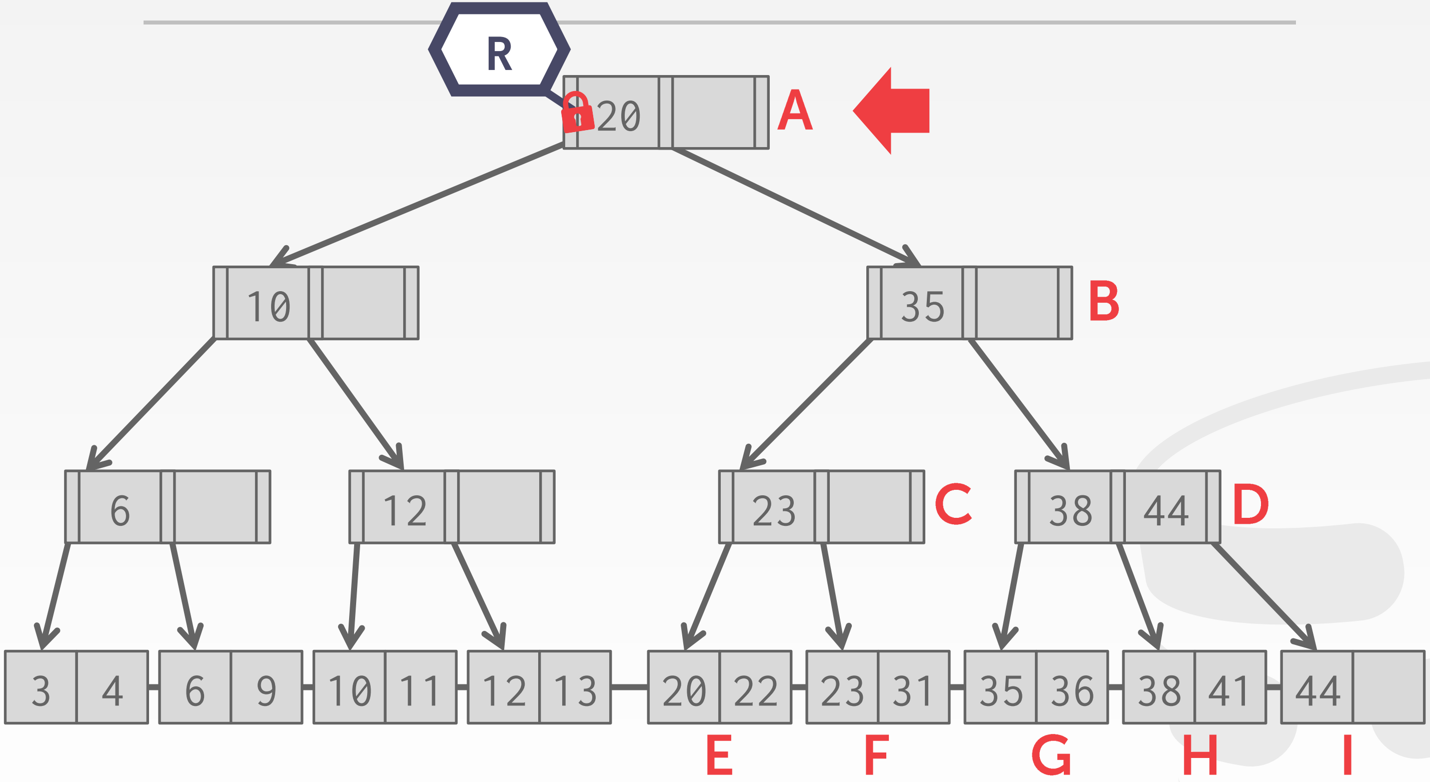
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

This approach optimistically assumes that only leaf node will be modified; if not, **R** latches set on the first pass to leaf are wasteful.

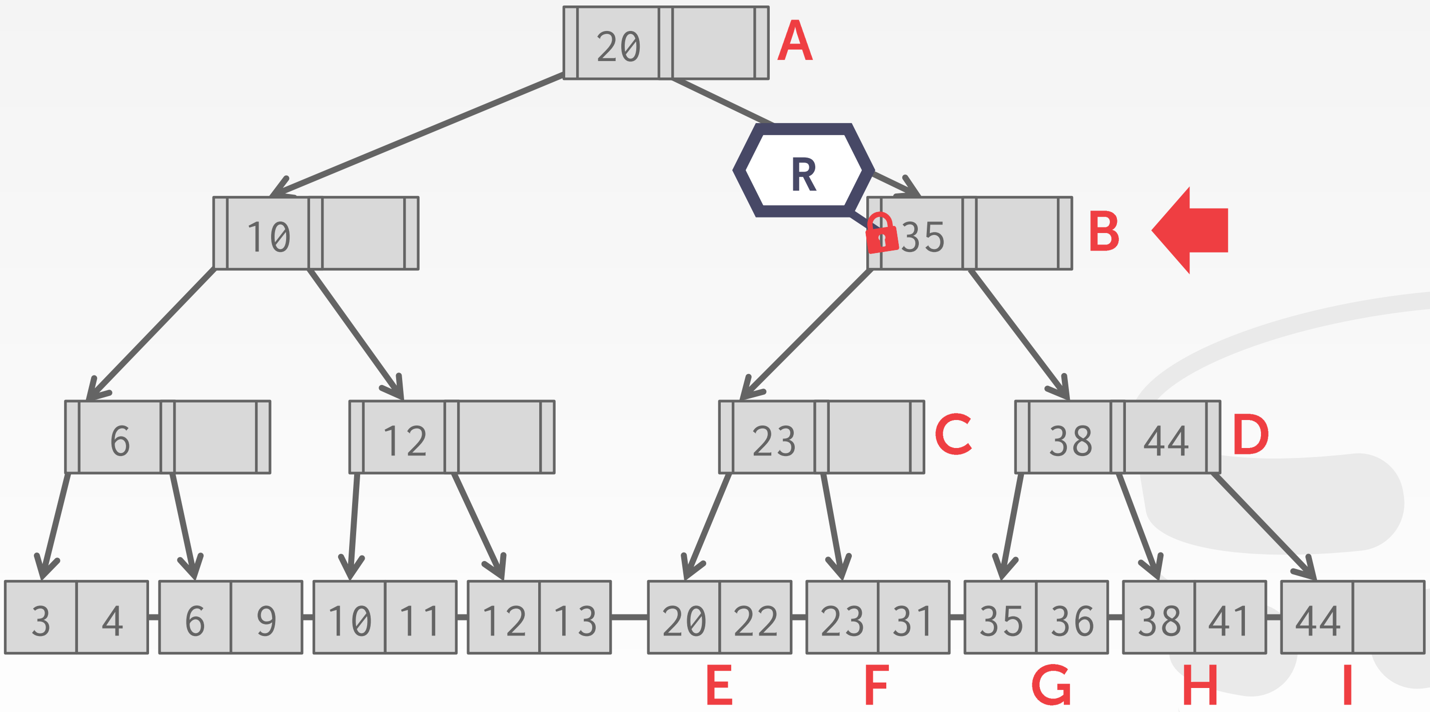
EXAMPLE #2 – DELETE 38



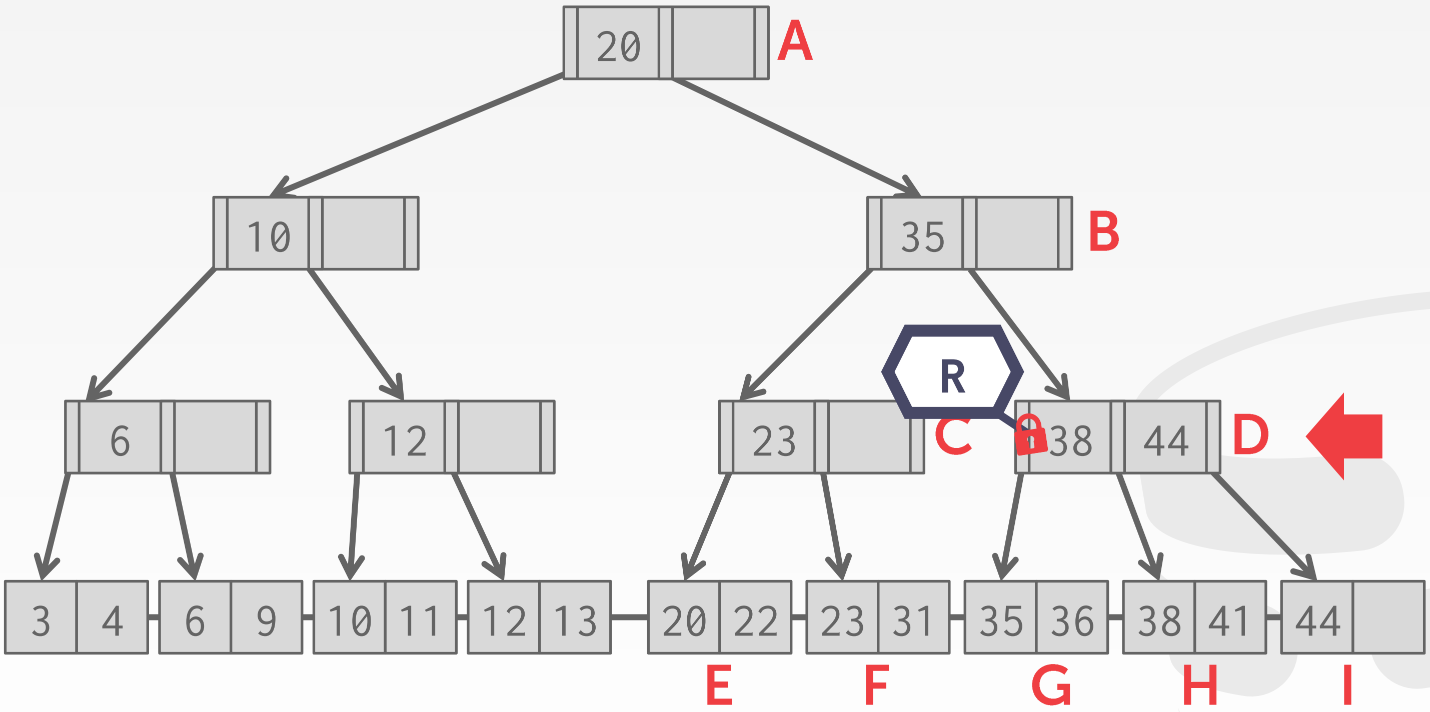
EXAMPLE #2 – DELETE 38



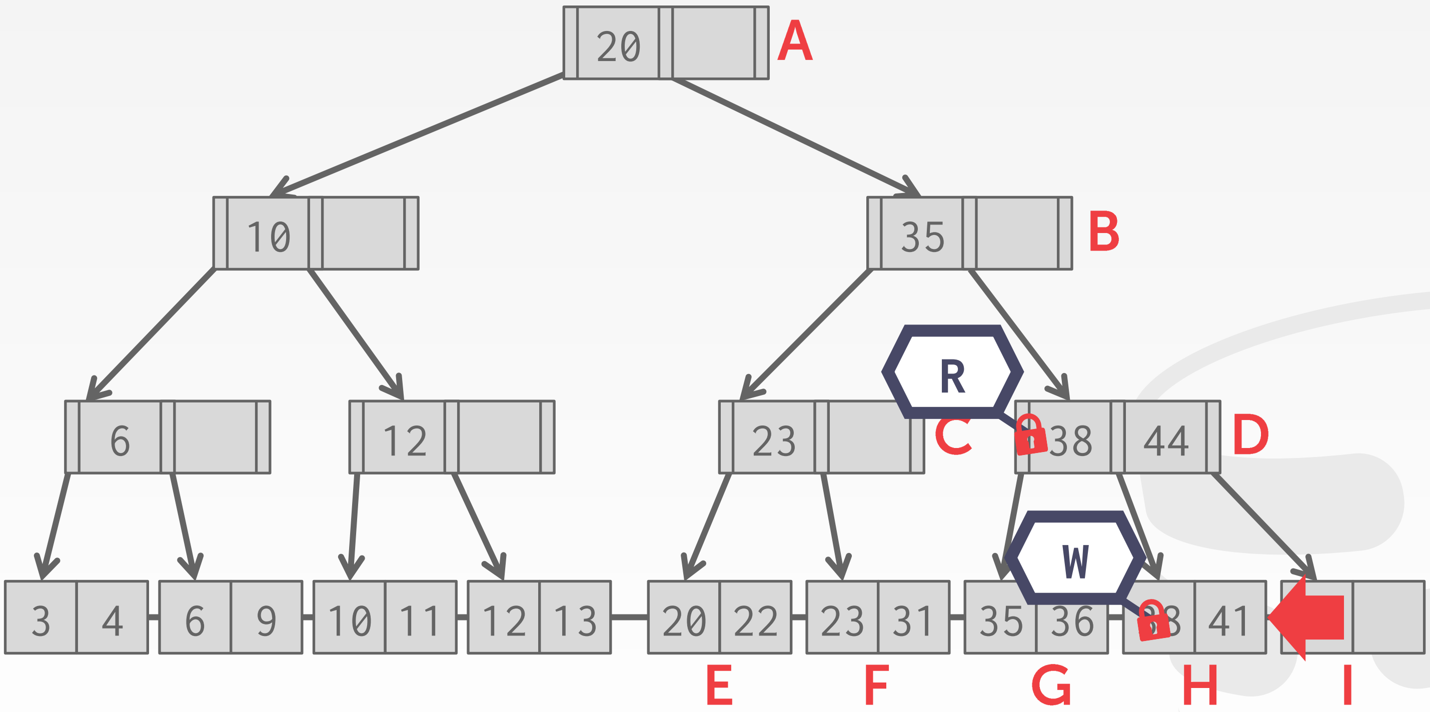
EXAMPLE #2 – DELETE 38



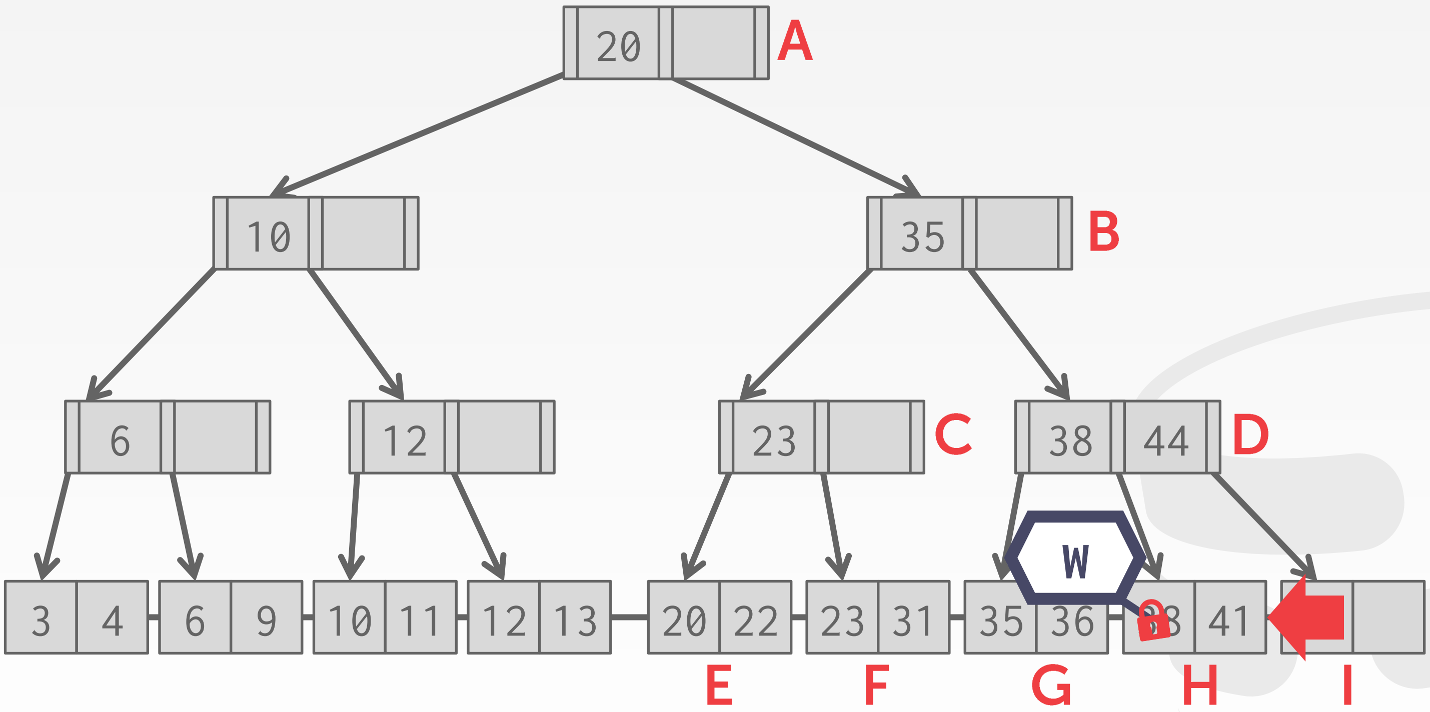
EXAMPLE #2 – DELETE 38



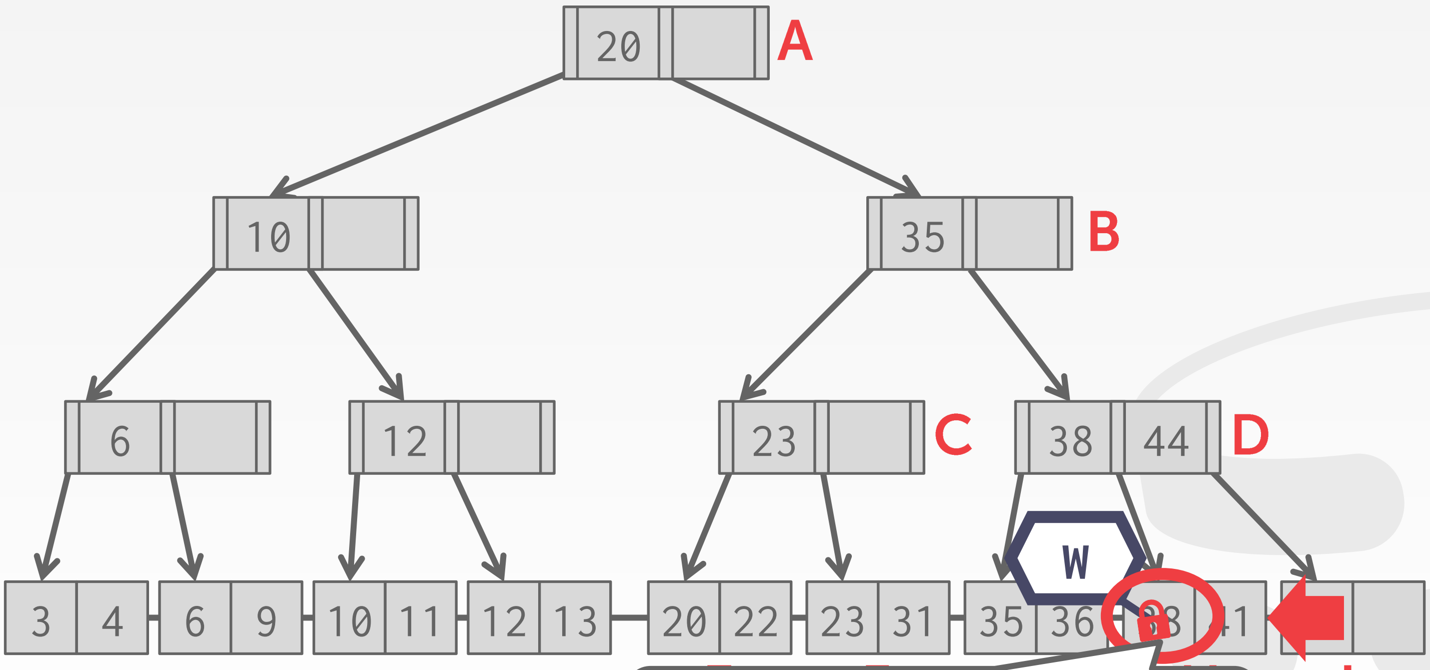
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

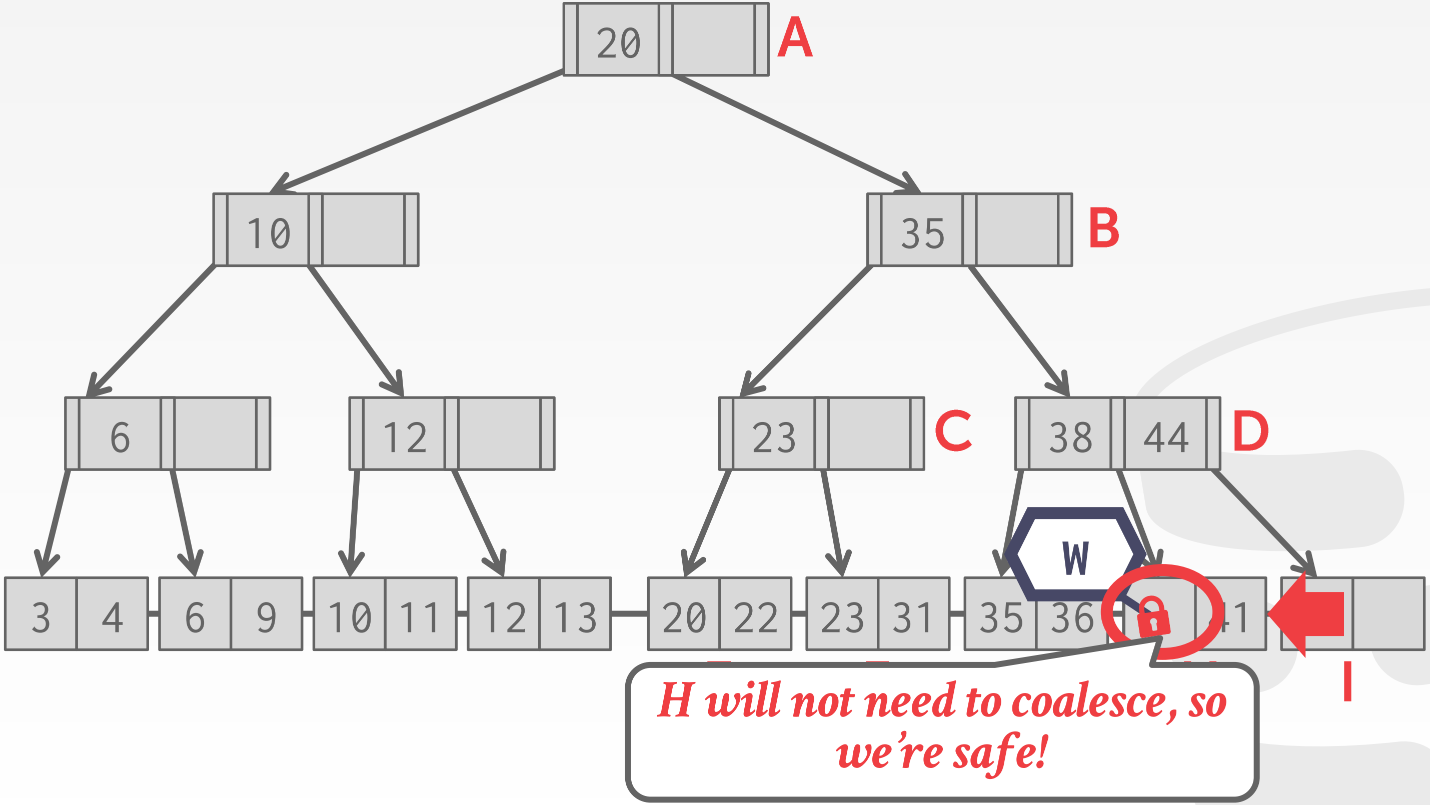


EXAMPLE #2 – DELETE 38

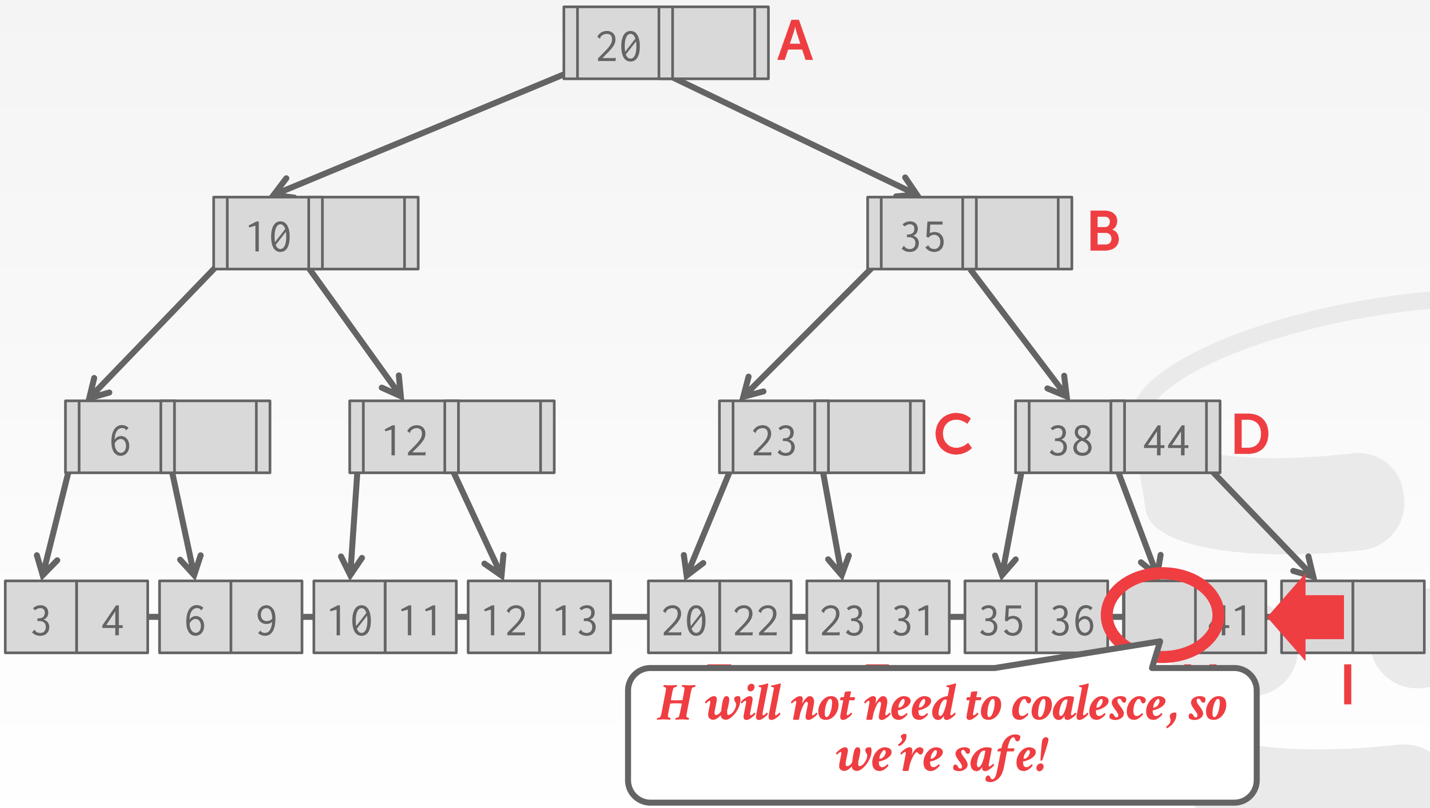


H will not need to coalesce, so we're safe!

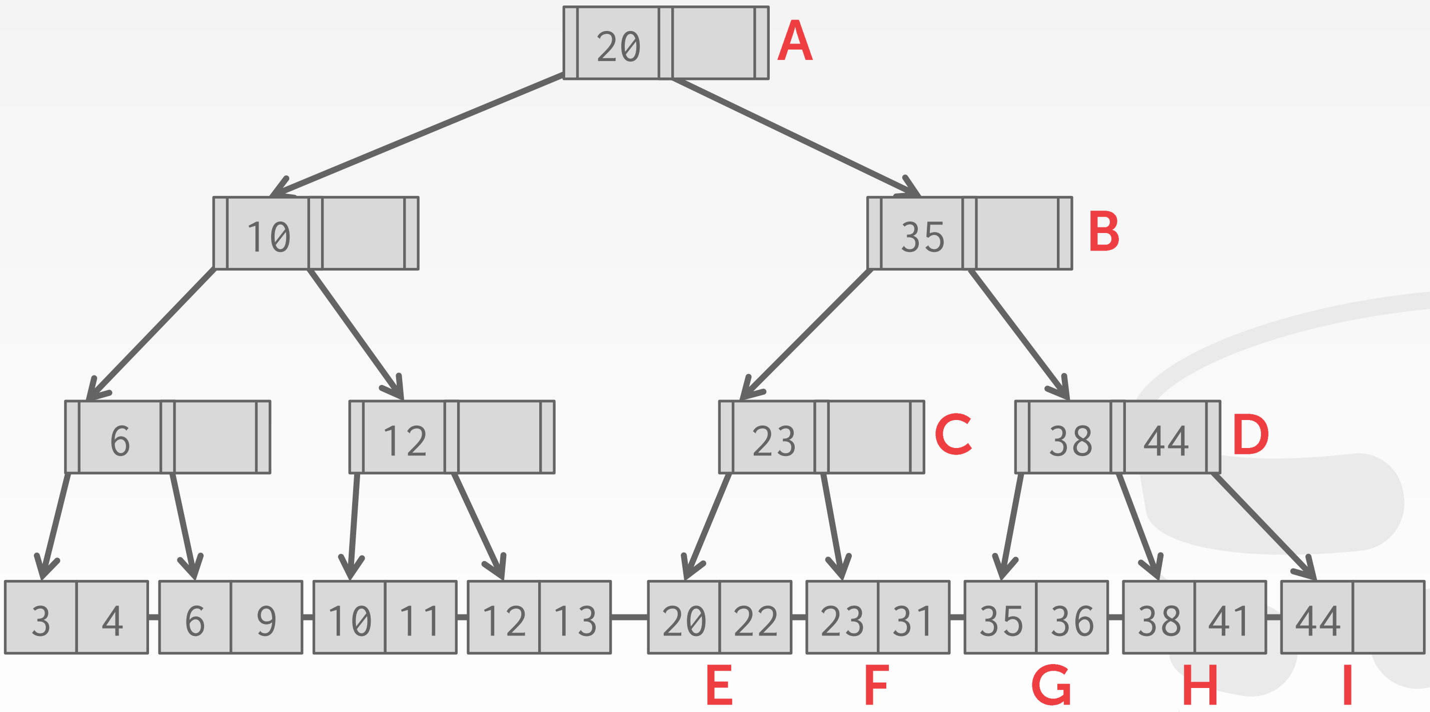
EXAMPLE #2 – DELETE 38



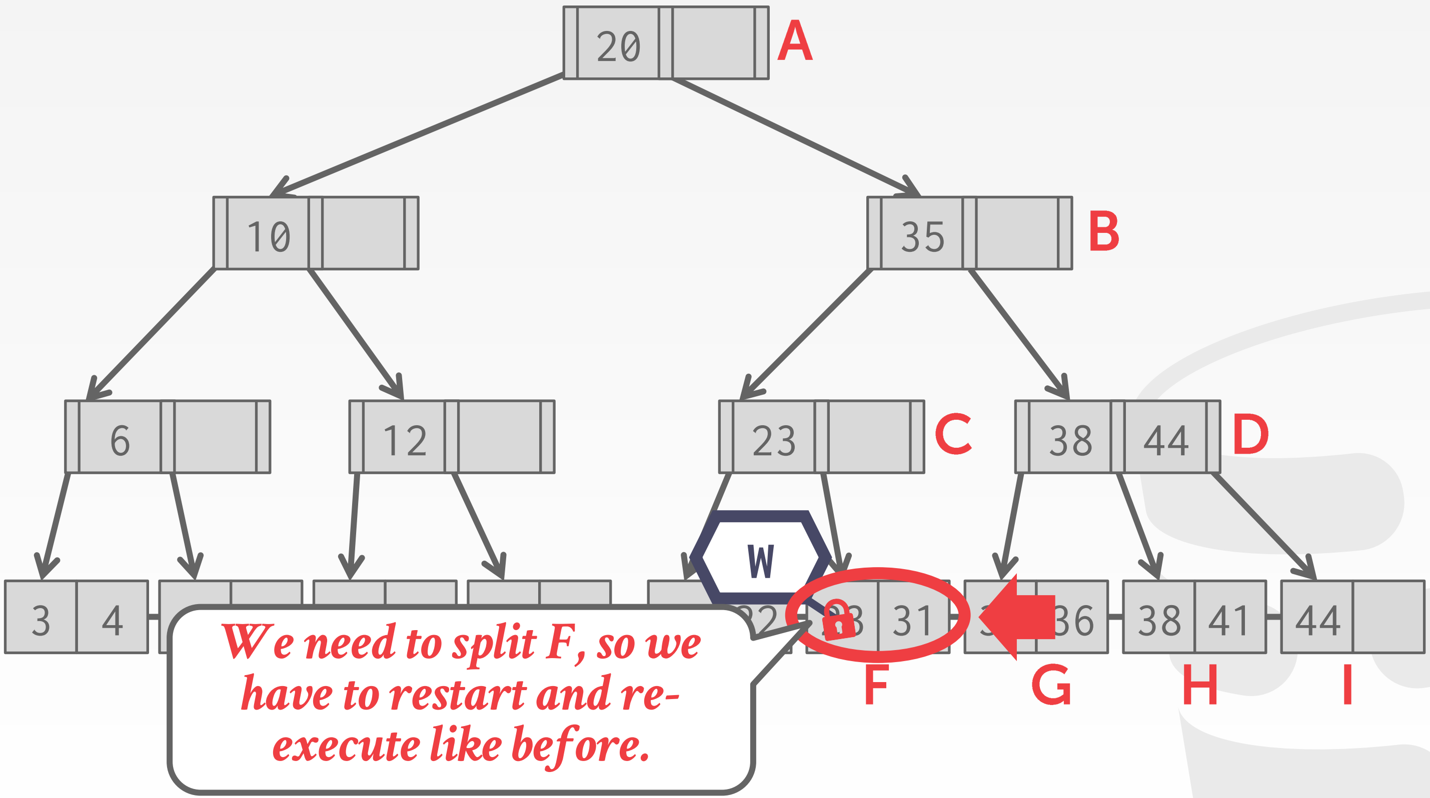
EXAMPLE #2 – DELETE 38



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

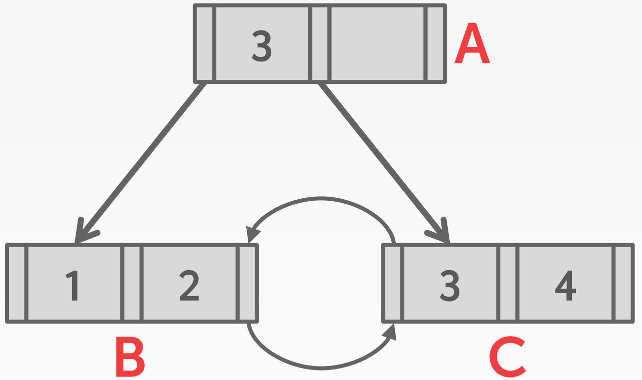
The threads in all the examples so far have acquired latches in a "top-down" manner.

→ A thread can only acquire a latch from a node that is below its current node.

→ If the desired latch is unavailable, the thread must wait until it becomes available.

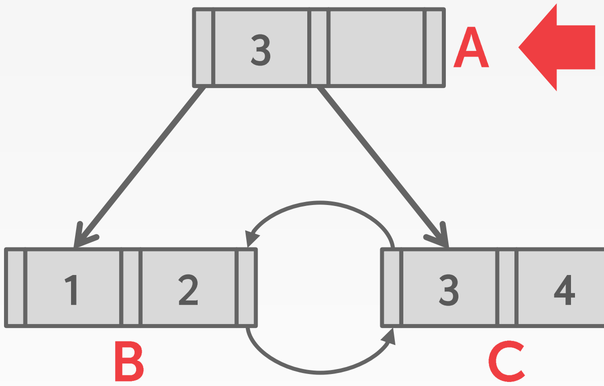
But what if we want to move from one leaf node to another leaf node?

LEAF NODE SCAN EXAMPLE #1



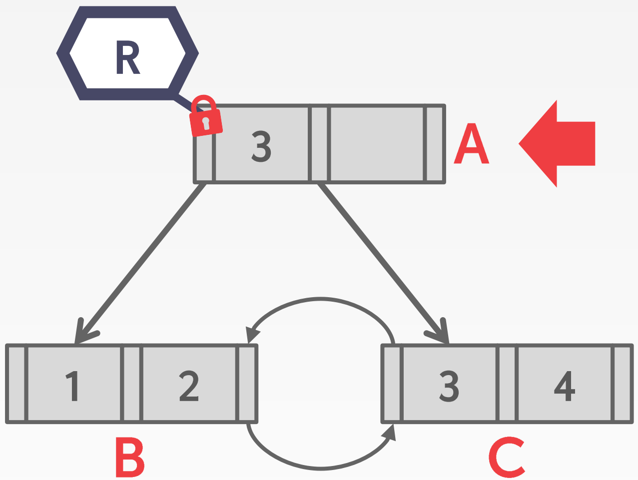
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



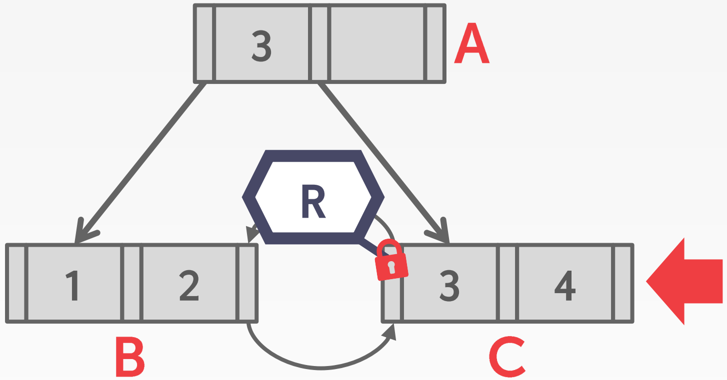
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



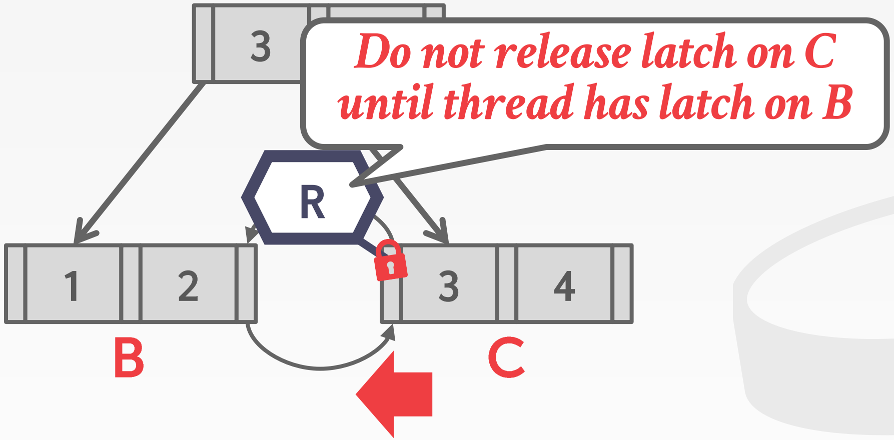
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



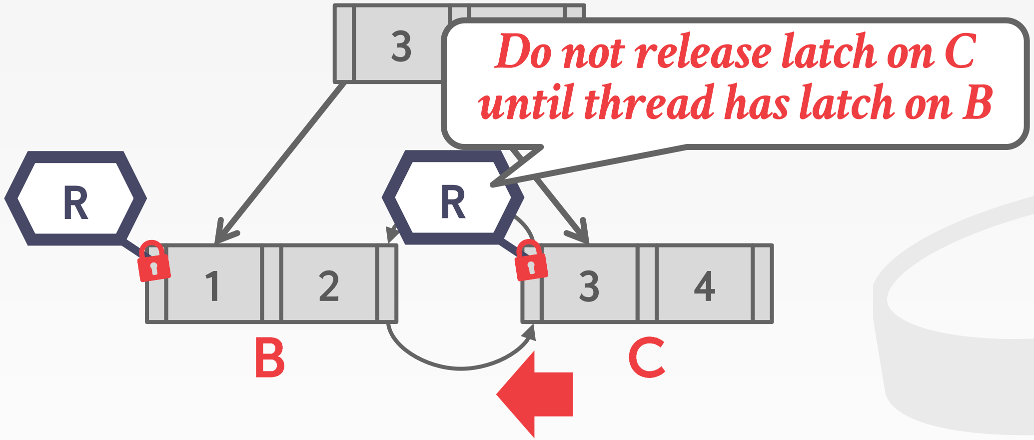
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



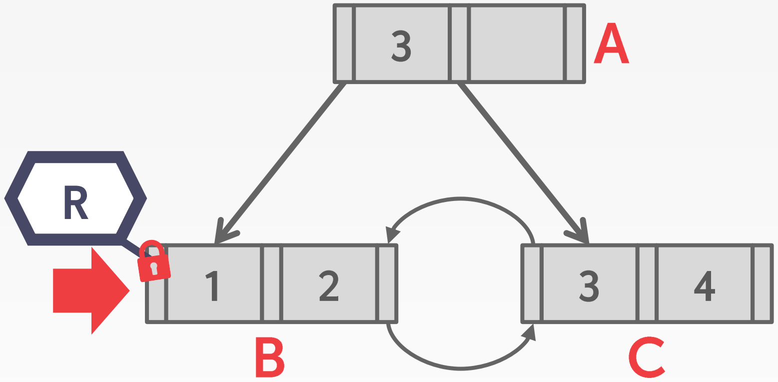
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #1

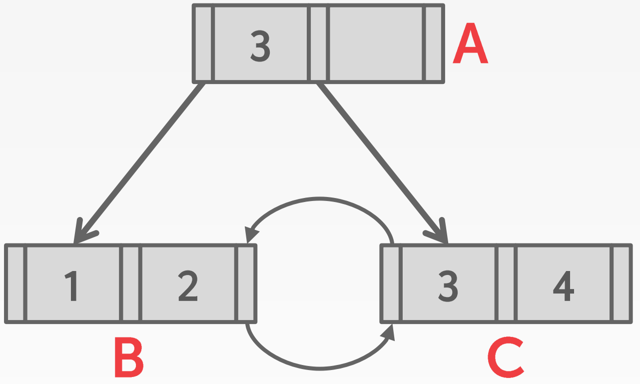
T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

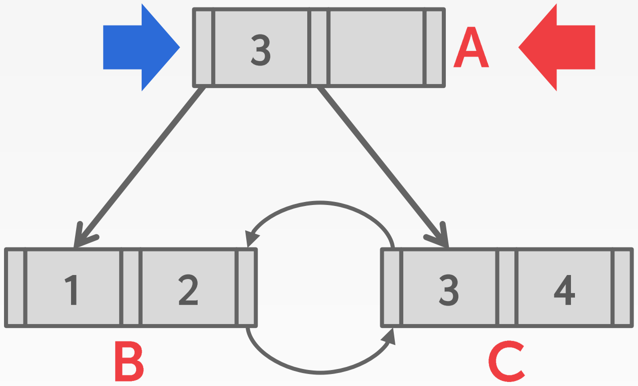
T_2: Find Keys > 1



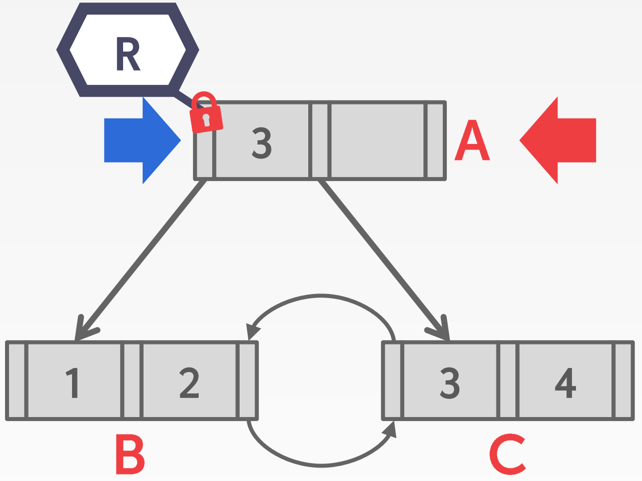
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



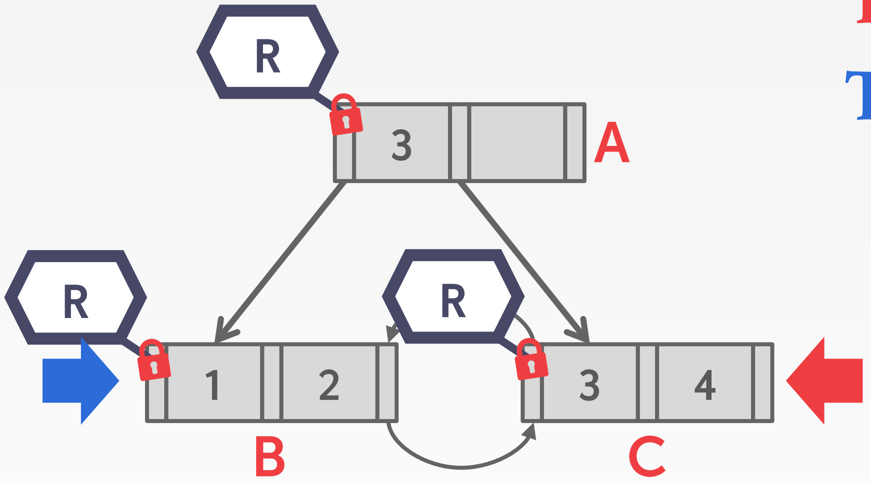
T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

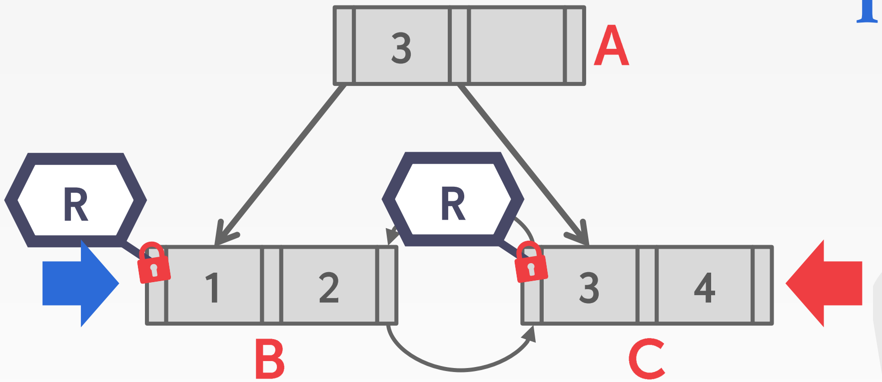
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

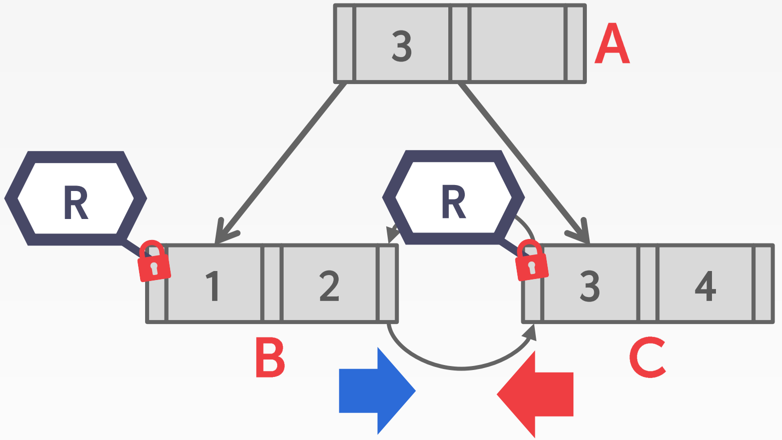
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

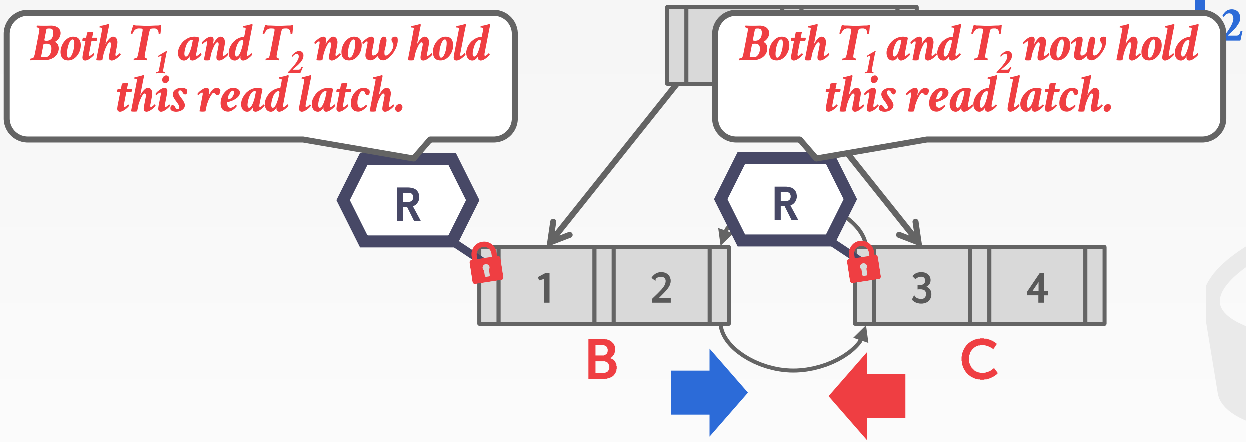
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

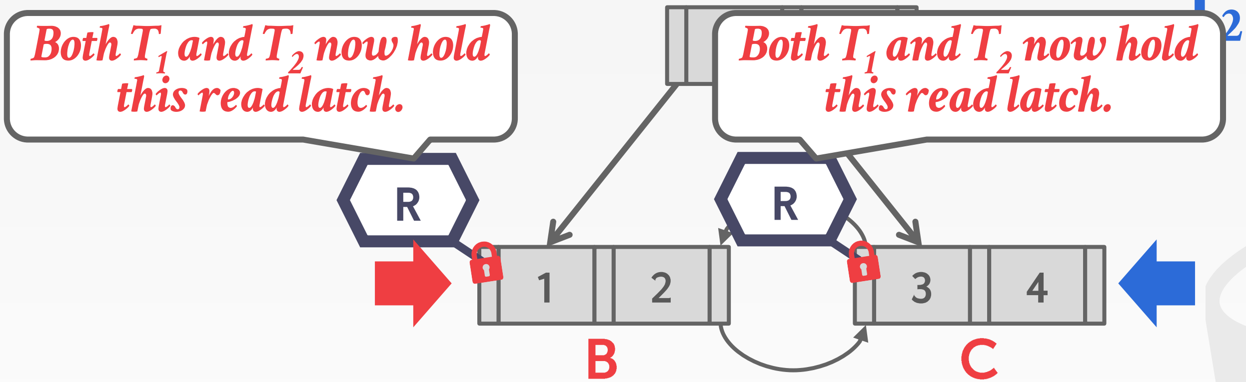
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

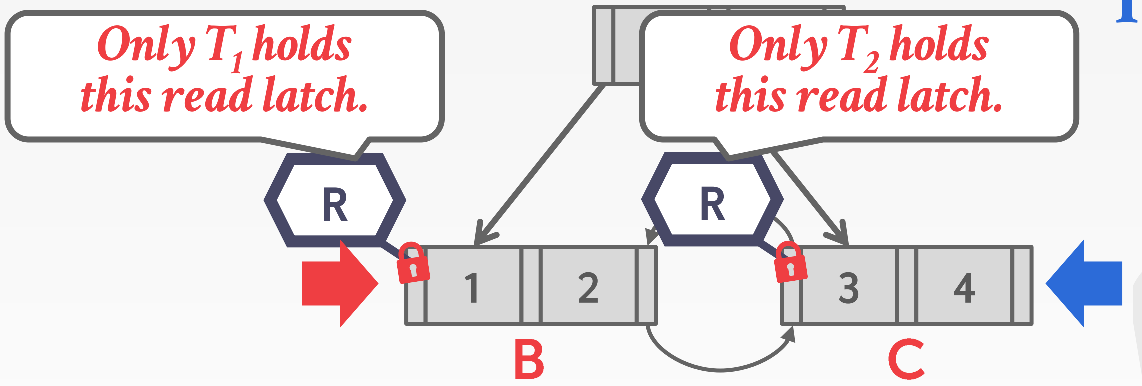
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCANS

Latches do not support deadlock detection or avoidance. The only way we can deal with this problem is through coding discipline.

The leaf node sibling latch acquisition protocol must support a “no-wait” mode.

The DBMS's data structures must cope with failed latch acquisitions.

CONCLUSION

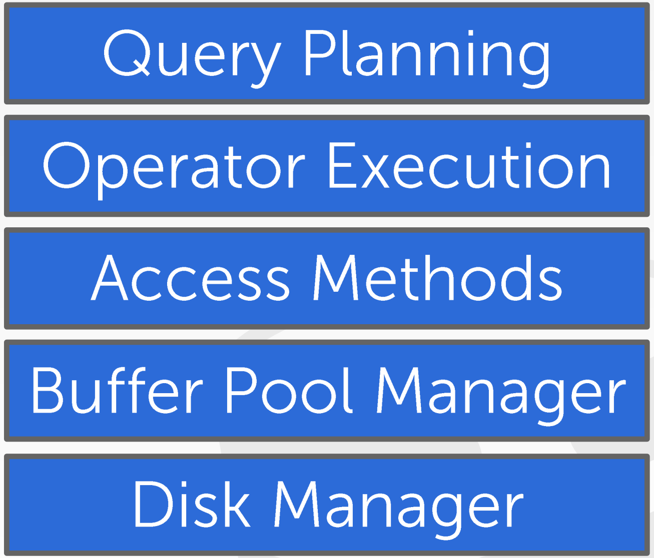
Making a data structure thread-safe is notoriously difficult in practice.

We focused on B+Trees, but the same high-level techniques are applicable to other data structures.

COURSE STATUS

We are now going to talk about how to execute queries using the DBMS components we have discussed so far.

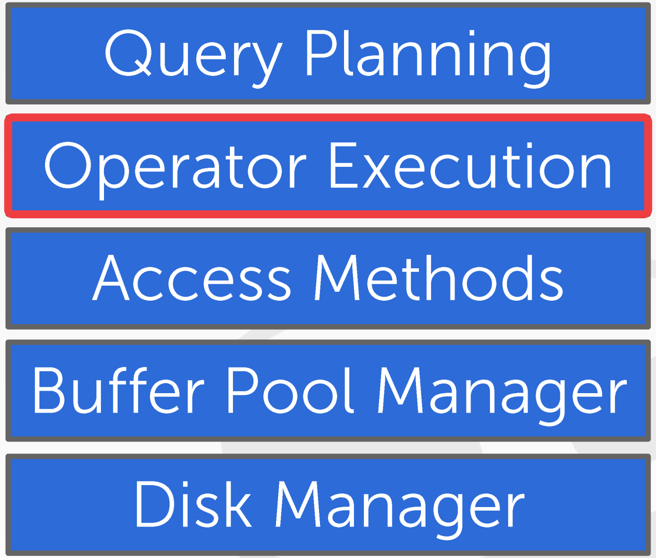
Next four lectures:
→ Operator Algorithms
→ Query Processing Models
→ Runtime Architectures



COURSE STATUS

We are now going to talk about how to execute queries using the DBMS components we have discussed so far.

Next four lectures:
→ Operator Algorithms
→ Query Processing Models
→ Runtime Architectures



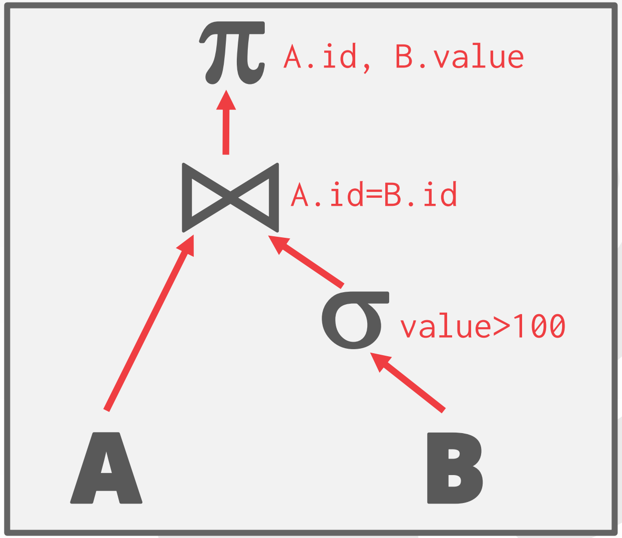
QUERY PLAN

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

The output of the root node is the result of the query.

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



DISK-ORIENTED DBMS

Just like it cannot assume that a table fits entirely in memory, a disk-oriented DBMS cannot assume that query results fit in memory.

We are going to rely on the buffer pool to implement algorithms that need to spill to disk.

We are also going to prefer algorithms that maximize the amount of sequential I/O.

TODAY'S AGENDA

External Merge Sort

Aggregations

WHY DO WE NEED TO SORT?

Relational model/SQL is unsorted.

WHY DO WE NEED TO SORT?

Relational model/SQL is unsorted.

Queries may request that tuples are sorted in a specific way (**ORDER BY**).

WHY DO WE NEED TO SORT?

Relational model/SQL is unsorted.

Queries may request that tuples are sorted in a specific way (**ORDER BY**).

But even if a query does not specify an order, we may still want to sort to do other things:
→ Trivial to support duplicate elimination (**DISTINCT**)
→ Bulk loading sorted tuples into a B+Tree index is faster
→ Aggregations (**GROUP BY**)
→ …

SORTING ALGORITHMS

If data fits in memory, then we can use a standard sorting algorithm like quicksort.

SORTING ALGORITHMS

If data fits in memory, then we can use a standard sorting algorithm like quicksort.

If data does not fit in memory, then we need to use a technique that is aware of the cost of reading and writing disk pages…

EXTERNAL MERGE SORT

Divide-and-conquer algorithm that splits data into separate **runs**, sorts them individually, and then combines them into longer sorted runs.

Phase #1 – Sorting

→ Sort chunks of data that fit in memory and then write back the sorted chunks to a file on disk.

Phase #2 – Merging

→ Combine sorted runs into larger chunks.

SORTED RUN

A run is a list of key/value pairs.

Key: The attribute(s) to compare to compute the sort order.

Value: Two choices

→ Tuple (***early materialization***).

→ Record ID (***late materialization***).

SORTED RUN

A run is a list of key/value pairs.

Key: The attribute(s) to compare to compute the sort order.

Value: Two choices

→ Tuple (***early materialization***).

→ Record ID (***late materialization***).

Early Materialization

K1	*<Tuple Data>*
K2	*<Tuple Data>*

⋮

SORTED RUN

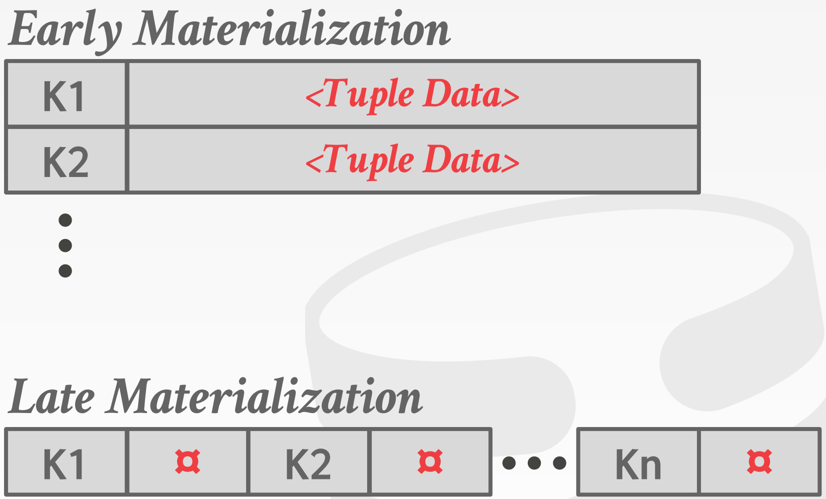
A run is a list of key/value pairs.

Key: The attribute(s) to compare to compute the sort order.

Value: Two choices

→ Tuple (***early materialization***).

→ Record ID (***late materialization***).

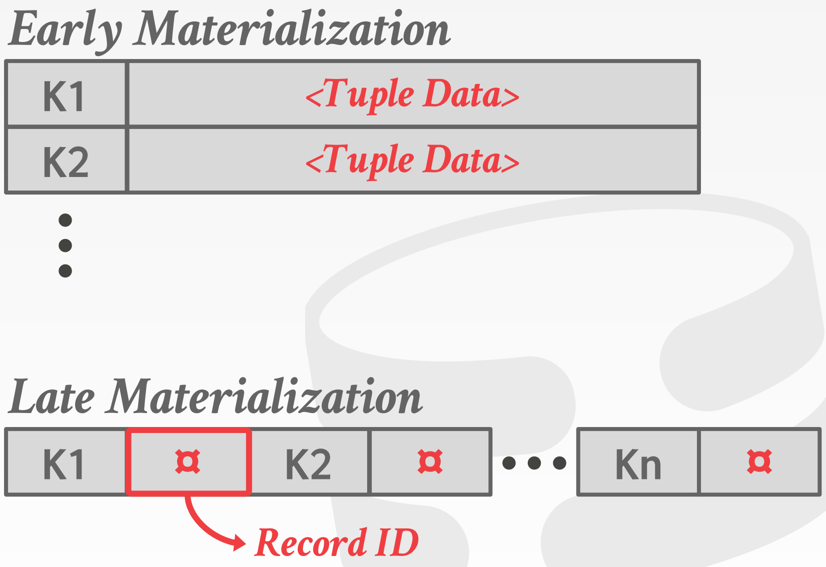


SORTED RUN

A run is a list of key/value pairs.

Key: The attribute(s) to compare to compute the sort order.

Value: Two choices
→ Tuple (***early materialization***).
→ Record ID (***late materialization***).



2-WAY EXTERNAL MERGE SORT

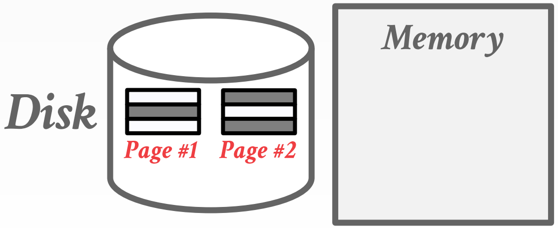
We will start with a simple example of a 2-way external merge sort.

→ "2" is the number of runs that we are going to merge into a new run for each pass.

Data is broken up into ***N*** pages.

The DBMS has a finite number of ***B*** buffer pool pages to hold input and output data.

2-WAY EXTERNAL MERGE SORT

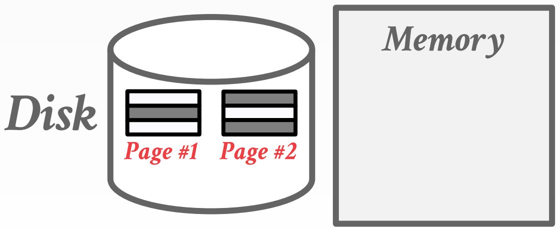


2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

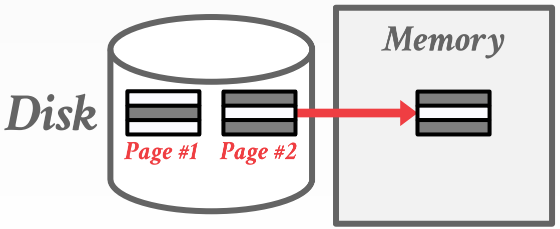


2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory

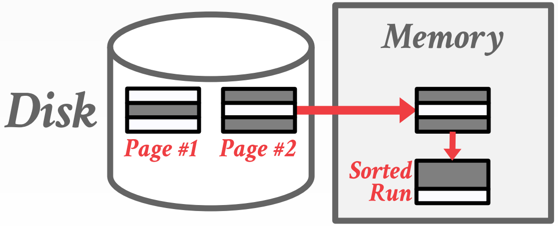
→ Sort pages into runs and write them back to disk



2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk

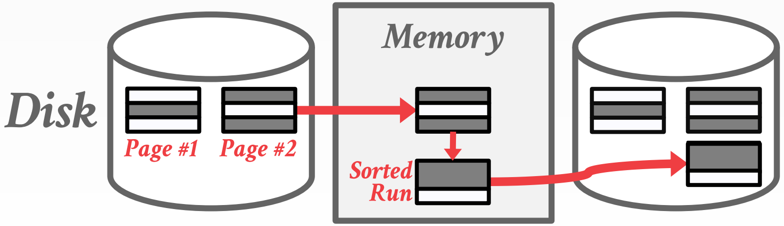


2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

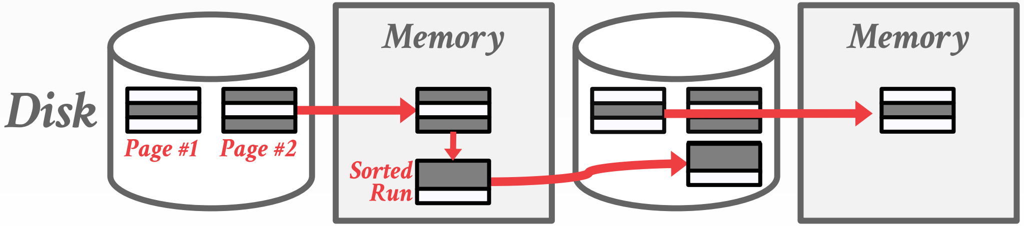


2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory

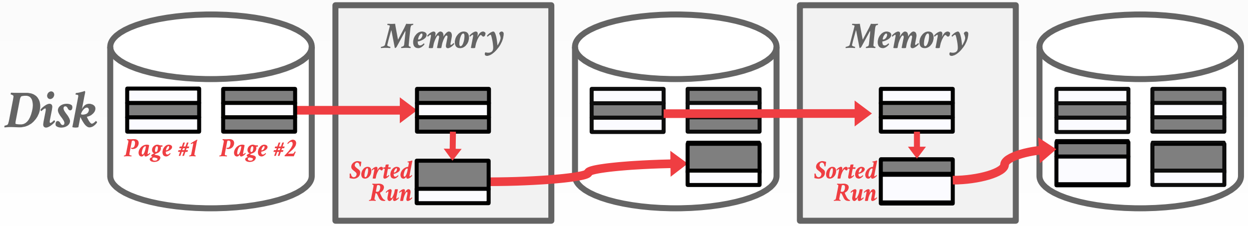
→ Sort pages into runs and write them back to disk



2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk



2-WAY EXTERNAL MERGE SORT

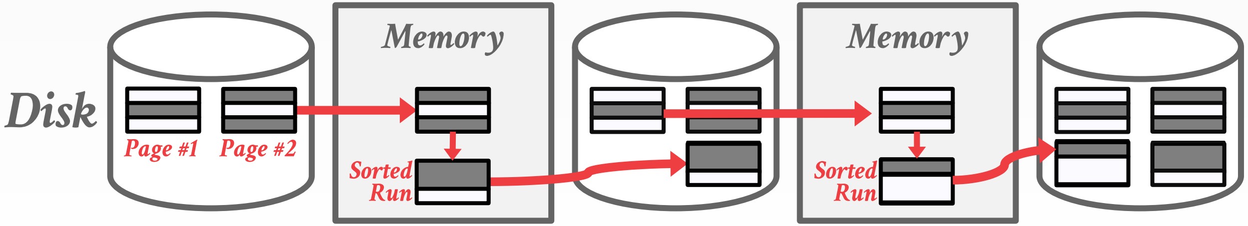
Pass #0

→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long



2-WAY EXTERNAL MERGE SORT

Pass #0

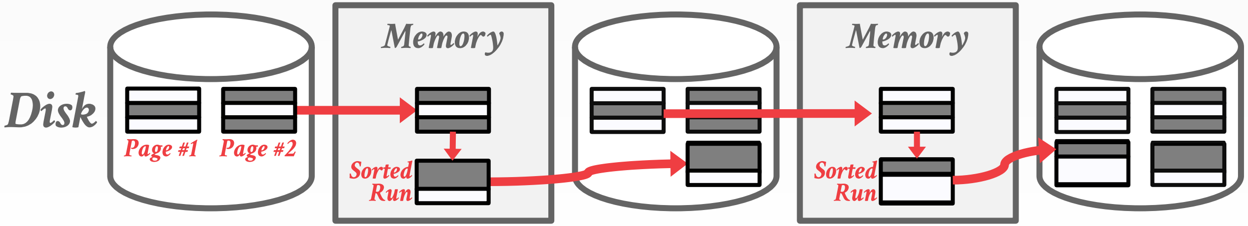
→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long

→ Uses three buffer pages (2 for input pages, 1 for output)



2-WAY EXTERNAL MERGE SORT

Pass #0

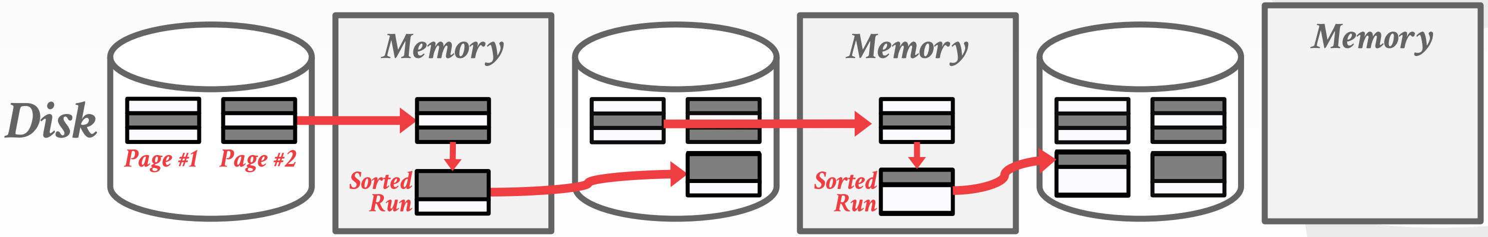
→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long

→ Uses three buffer pages (2 for input pages, 1 for output)



2-WAY EXTERNAL MERGE SORT

Pass #0

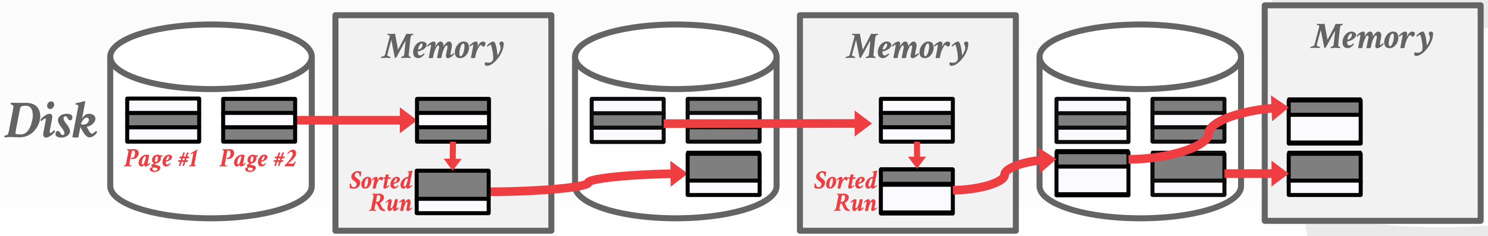
→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long

→ Uses three buffer pages (2 for input pages, 1 for output)



2-WAY EXTERNAL MERGE SORT

Pass #0

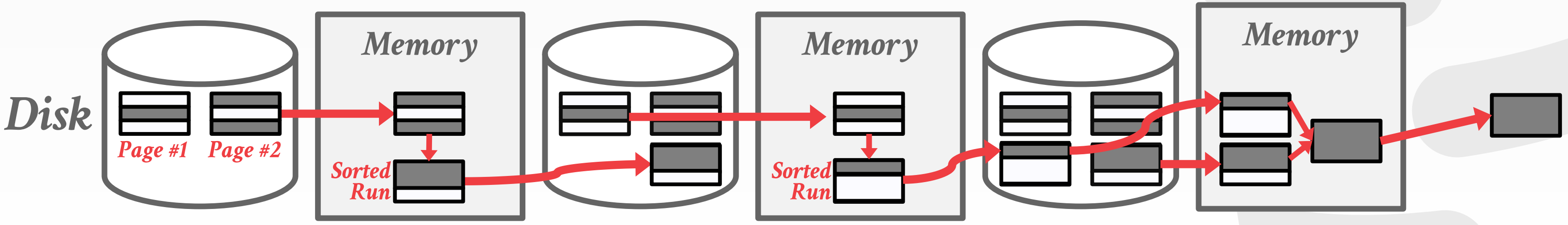
→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long

→ Uses three buffer pages (2 for input pages, 1 for output)



2-WAY EXTERNAL MERGE SORT

Pass #0

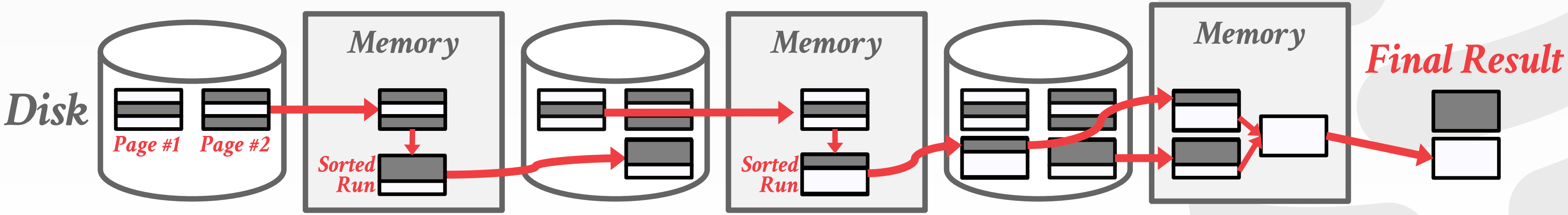
→ Read all ***B*** pages of the table into memory

→ Sort pages into runs and write them back to disk

Pass #1,2,3,…

→ Recursively merge pairs of runs into runs twice as long

→ Uses three buffer pages (2 for input pages, 1 for output)



2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\text{\# of passes})}$

2-WAY EXTERNAL MERGE SORT

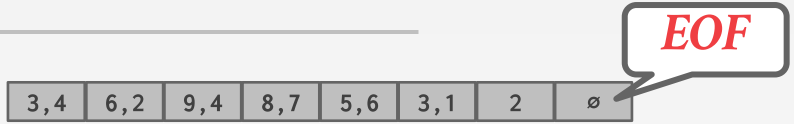
3,4	6,2	9,4	8,7	5,6	3,1	2	∅

In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\#\ of\ passes)}$

2-WAY EXTERNAL MERGE SORT



In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\text{\# of passes})}$

2-WAY EXTERNAL MERGE SORT

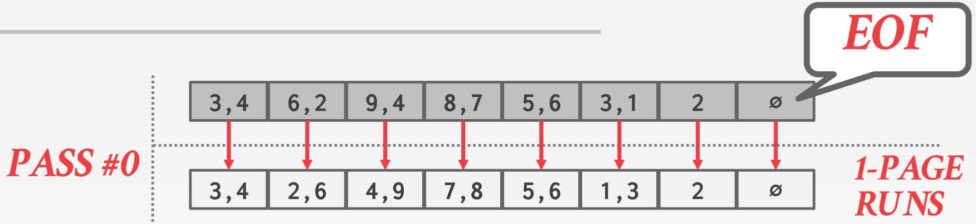
In each pass, we read and write every page in the file.

Number of passes

$= 1 + \lceil \log_2 N \rceil$

Total I/O cost

$= 2N \cdot (\text{\# of passes})$

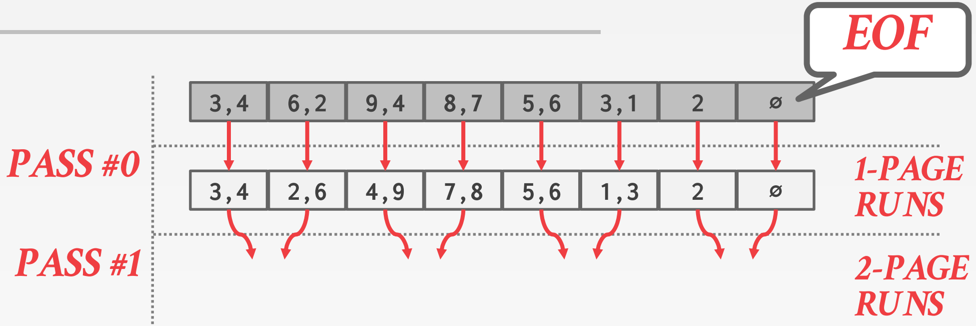


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= 1 + \lceil \log_2 N \rceil$

Total I/O cost
$= 2N \cdot (\text{\# of passes})$

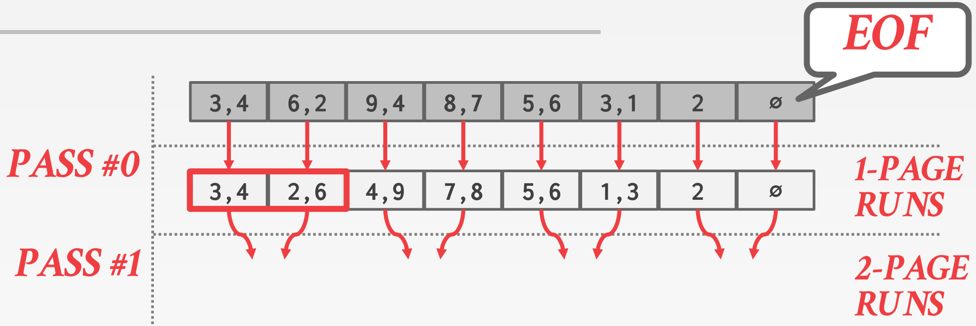


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= 1 + \lceil \log_2 N \rceil$

Total I/O cost
$= 2N \cdot (\text{\# of passes})$



2-WAY EXTERNAL MERGE SORT

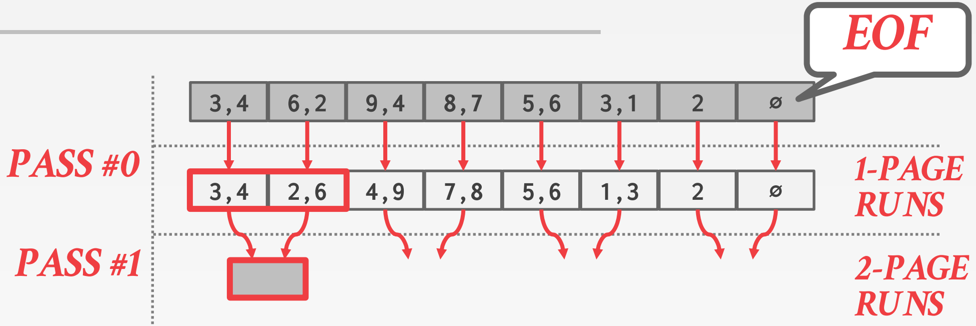
In each pass, we read and write every page in the file.

Number of passes

$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost

$= \mathbf{2N \cdot (\#\ of\ passes)}$

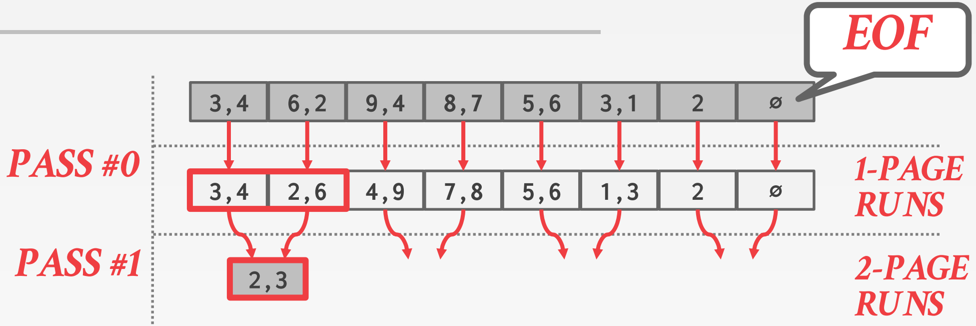


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\#\ of\ passes)}$

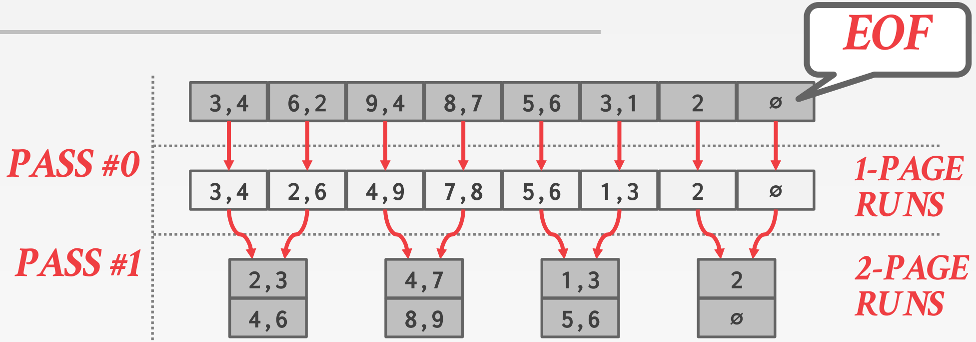


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\#\ of\ passes)}$

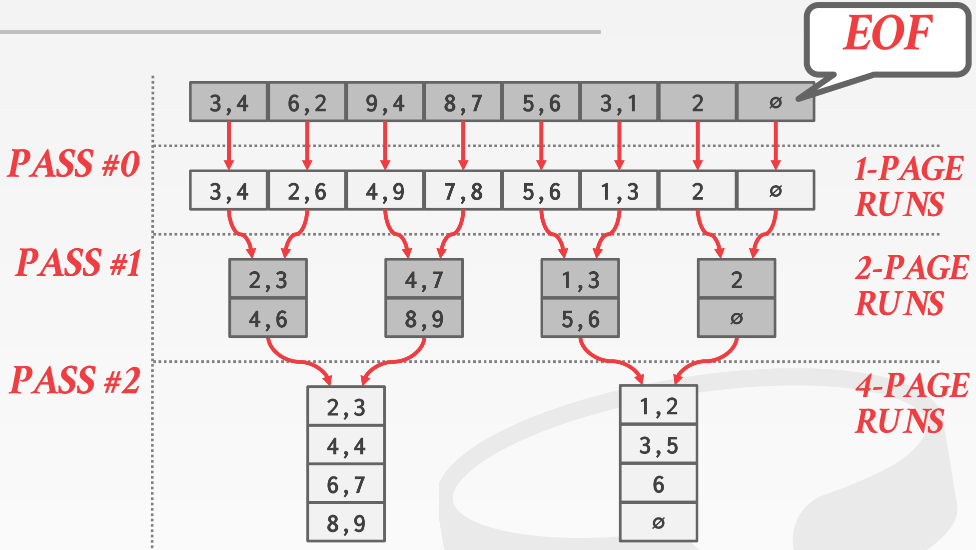


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost
$= \mathbf{2N \cdot (\# \text{ of passes})}$

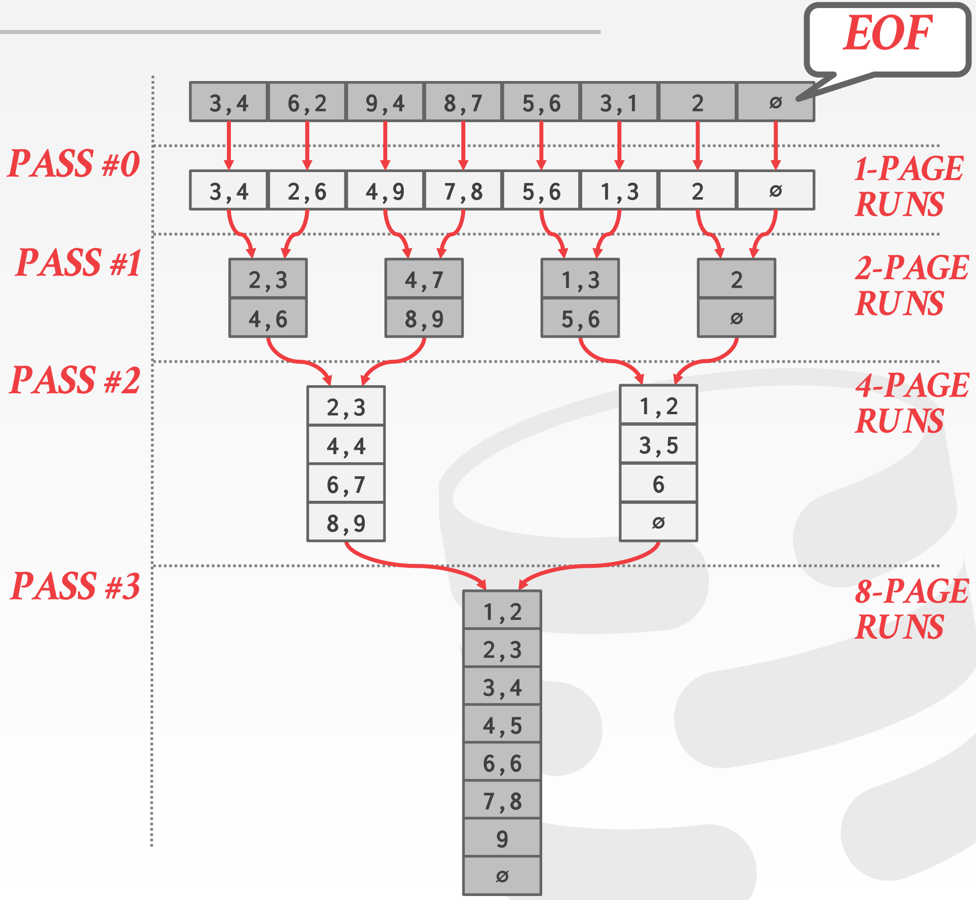


2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

Number of passes
$= 1 + \lceil \log_2 N \rceil$

Total I/O cost
$= 2N \cdot (\text{\# of passes})$



2-WAY EXTERNAL MERGE SORT

This algorithm only requires three buffer pool pages to perform the sorting (***B*=3**).

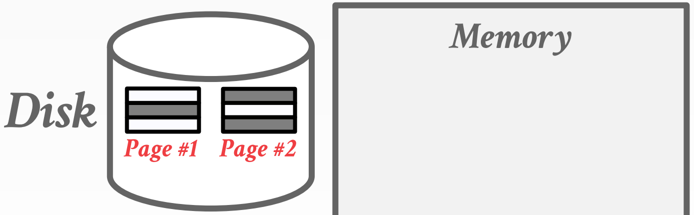
→ Two input pages, one output page

But even if we have more buffer space available (***B*>3**), it does not effectively utilize them if the worker must block on disk I/O…

DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

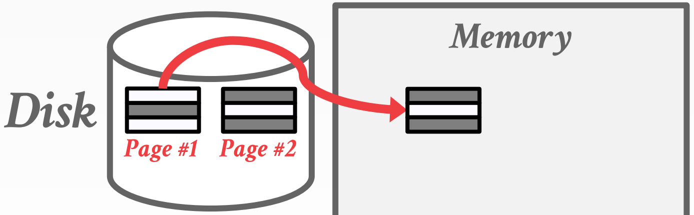
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

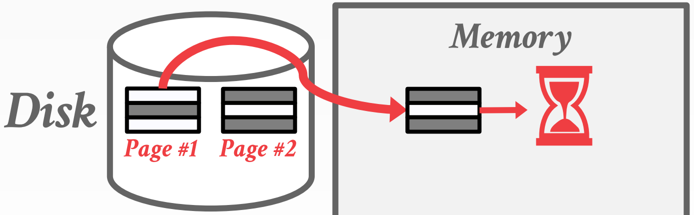
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

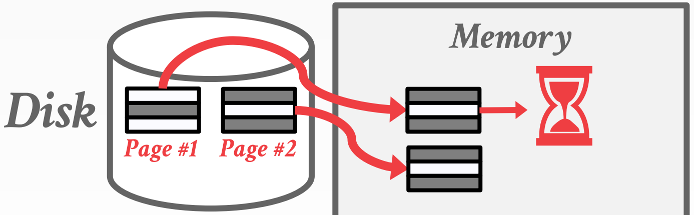
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

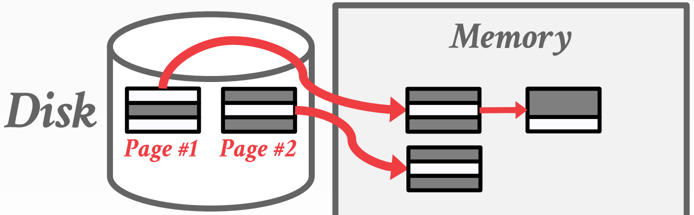
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

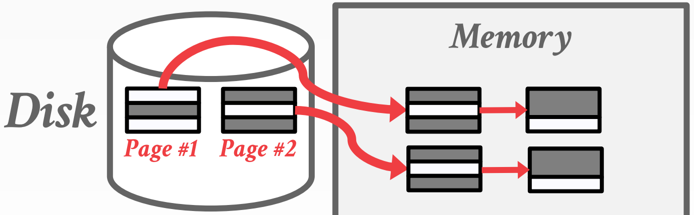
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING OPTIMIZATION

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



GENERAL EXTERNAL MERGE SORT

Pass #0

→ Use ***B*** buffer pages

→ Produce $\lceil N / B \rceil$ sorted runs of size ***B***

Pass #1,2,3,…

→ Merge ***B*-1** runs (i.e., K-way merge)

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

Total I/O Cost = $2N \cdot$ **(# of passes)**

GENERAL EXTERNAL MERGE SORT

Pass #0

→ Use ***B*** buffer pages

→ Produce $\lceil N / B \rceil$ sorted runs of size ***B***

Pass #1,2,3,…

→ Merge ***B*-1** runs (i.e., K-way merge)

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

Total I/O Cost = $2N \cdot$ (# of passes)

GENERAL EXTERNAL MERGE SORT

Pass #0

→ Use ***B*** buffer pages

→ Produce $\lceil N / B \rceil$ sorted runs of size ***B***

Pass #1,2,3,…

→ Merge ***B*-1** runs (i.e., K-way merge)

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

Total I/O Cost = $2N \cdot$ (# of passes)

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

→ **Pass #0:** $\lceil N / B \rceil$ = $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages).

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

→ **Pass #0:** $\lceil N / B \rceil$ = ⌈108 / 5⌉ = 22 sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** $\lceil N' / B\text{-}1 \rceil$ = ⌈22 / 4⌉ = 6 sorted runs of 20 pages each (last run is only 8 pages).

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

→ **Pass #0:** $\lceil N / B \rceil$ = ⌈108 / 5⌉ = 22 sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** $\lceil N' / B\text{-}1 \rceil$ = ⌈22 / 4⌉ = 6 sorted runs of 20 pages each (last run is only 8 pages).

→ **Pass #2:** $\lceil N'' / B\text{-}1 \rceil$ = ⌈6 / 4⌉ = 2 sorted runs, first one has 80 pages and second one has 28 pages.

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

→ **Pass #0:** $\lceil N / B \rceil$ = ⌈108 / 5⌉ = 22 sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** $\lceil N' / B\text{-}1 \rceil$ = ⌈22 / 4⌉ = 6 sorted runs of 20 pages each (last run is only 8 pages).

→ **Pass #2:** $\lceil N'' / B\text{-}1 \rceil$ = ⌈6 / 4⌉ = 2 sorted runs, first one has 80 pages and second one has 28 pages.

→ **Pass #3:** Sorted file of 108 pages.

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108**, ***B*=5**

→ **Pass #0:** $\lceil N / B \rceil = \lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** $\lceil N' / B\text{-}1 \rceil = \lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages).

→ **Pass #2:** $\lceil N'' / B\text{-}1 \rceil = \lceil 6 / 4 \rceil$ = 2 sorted runs, first one has 80 pages and second one has 28 pages.

→ **Pass #3:** Sorted file of 108 pages.

$$1+\lceil \log_{B-1}\lceil N / B \rceil \rceil = 1+\lceil \log_4 22 \rceil = 1+\lceil 2.229... \rceil$$

$$= \textbf{4 passes}$$

USING B+TREES FOR SORTING

If the table that must be sorted already has a B+Tree index on the sort attribute(s), then we can use that to accelerate sorting.

Retrieve tuples in desired sort order by simply traversing the leaf pages of the tree.

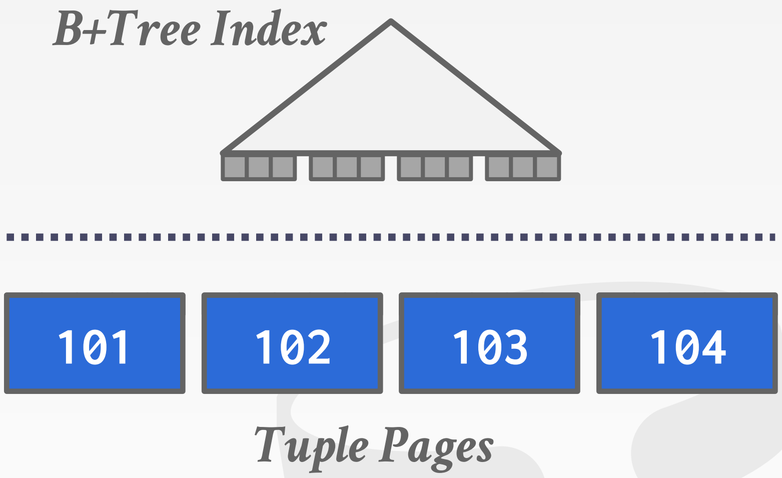
Cases to consider:

→ Clustered B+Tree
→ Unclustered B+Tree

CASE #1 – CLUSTERED B+TREE

Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.

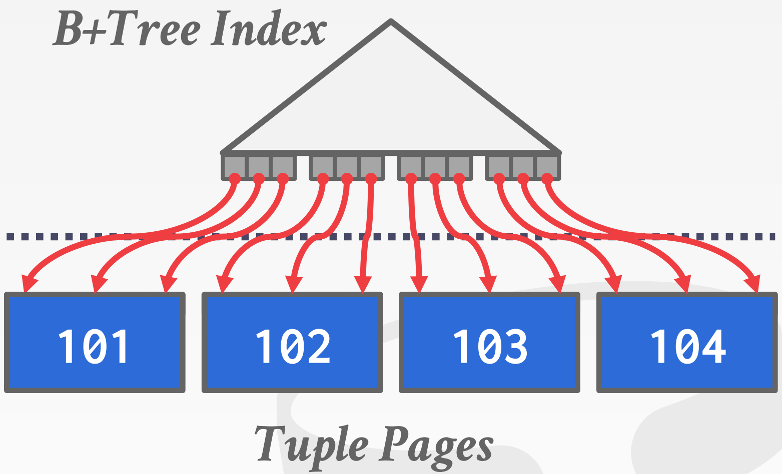
This is always better than external sorting because there is no computational cost, and all disk access is sequential.



CASE #1 – CLUSTERED B+TREE

Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.

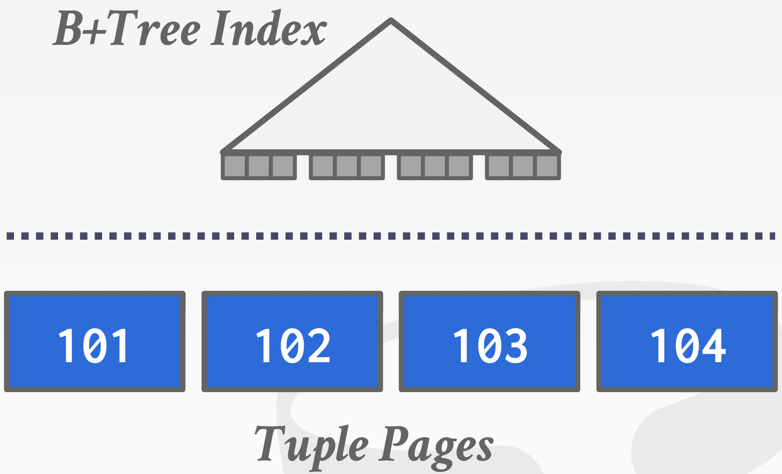
This is always better than external sorting because there is no computational cost, and all disk access is sequential.



CASE #2 – UNCLUSTERED B+TREE

Chase each pointer to the page that contains the data.

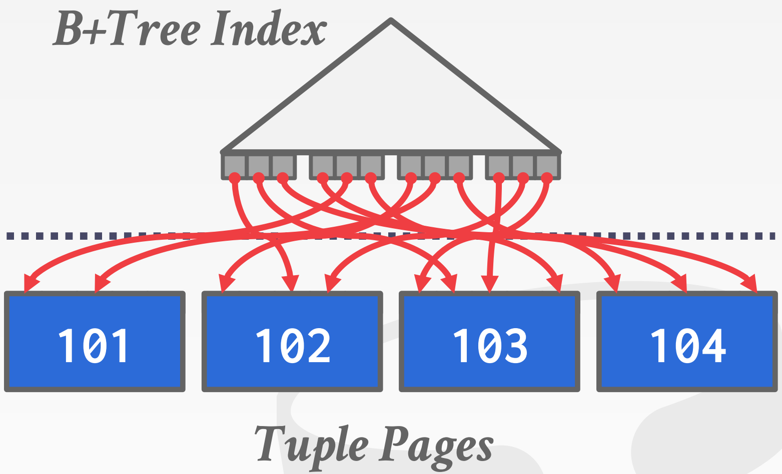
This is almost always a bad idea.
In general, one I/O per data record.



CASE #2 – UNCLUSTERED B+TREE

Chase each pointer to the page that contains the data.

This is almost always a bad idea.
In general, one I/O per data record.



AGGREGATIONS

Collapse values for a single attribute from multiple tuples into a single scalar value.

Two implementation choices:

→ Sorting
→ Hashing

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

SORTING AGGREGATION

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

SORTING AGGREGATION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
15-445
15-721
15-826

Eliminate Dupes

SORTING AGGREGATION

```sql
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
 ORDER BY cid
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445

Sort

cid
15-445
~~15-445~~
15-721
15-826

Eliminate Dupes

ALTERNATIVES TO SORTING

What if we do not need the data to be ordered?

→ Forming groups in **GROUP BY** (no ordering)

→ Removing duplicates in **DISTINCT** (no ordering)

ALTERNATIVES TO SORTING

What if we do not need the data to be ordered?
→ Forming groups in **GROUP BY** (no ordering)
→ Removing duplicates in **DISTINCT** (no ordering)

Hashing is a better alternative in this scenario.
→ Only need to remove duplicates, no need for ordering.
→ Can be computationally cheaper than sorting.

HASHING AGGREGATE

Populate an ephemeral hash table as the DBMS scans the table. For each record, check whether there is already an entry in the hash table:

→ **DISTINCT**: Discard duplicate
→ **GROUP BY**: Perform aggregate computation

If everything fits in memory, then this is easy.

If the DBMS must spill data to disk, then we need to be smarter…

EXTERNAL HASHING AGGREGATE

Phase #1 – Partition

→ Divide tuples into buckets based on hash key
→ Write them out to disk when they get full

Phase #2 – ReHash

→ Build in-memory hash table for each partition and compute the aggregation

PHASE #1 – PARTITION

Use a hash function h_1 to split tuples into **partitions** on disk.

→ A partition is one or more pages that contain the set of keys with the same hash value.

→ Partitions are "spilled" to disk via output buffers.

Assume that we have ***B*** buffers.

We will use ***B-1*** buffers for the partitions and ***1*** buffer for the input data.

PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Filter →

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns →

cid
15-445
15-826
15-721
15-445

⋮

PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

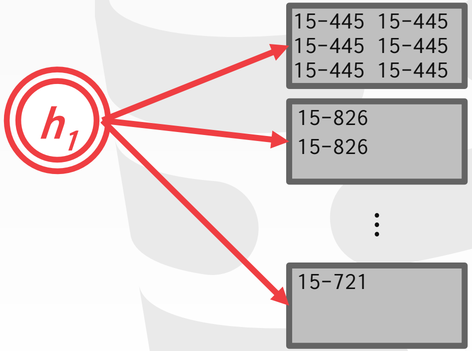
Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445
⋮

B-1 partitions



PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

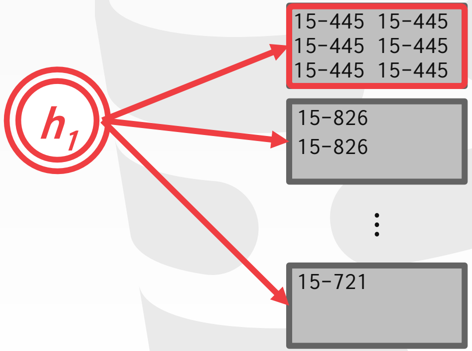
Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445
⋮

B-1 partitions



PHASE #1 – PARTITION

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

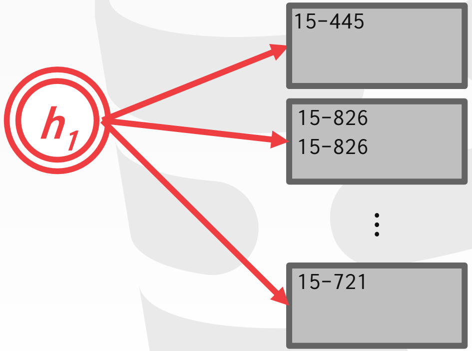
Filter

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

Remove Columns

cid
15-445
15-826
15-721
15-445
⋮

B-1 partitions



PHASE #2 – REHASH

For each partition on disk:

→ Read it into memory and build an in-memory hash table based on a second hash function h_2.

→ Then go through each bucket of this hash table to bring together matching tuples.

This assumes that each partition fits in memory.

PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

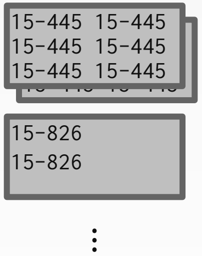
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



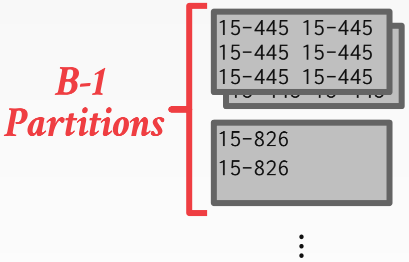
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



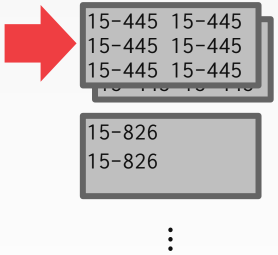
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets

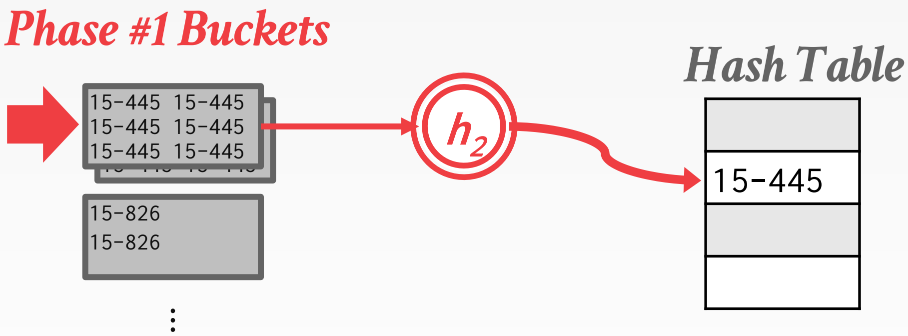


PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

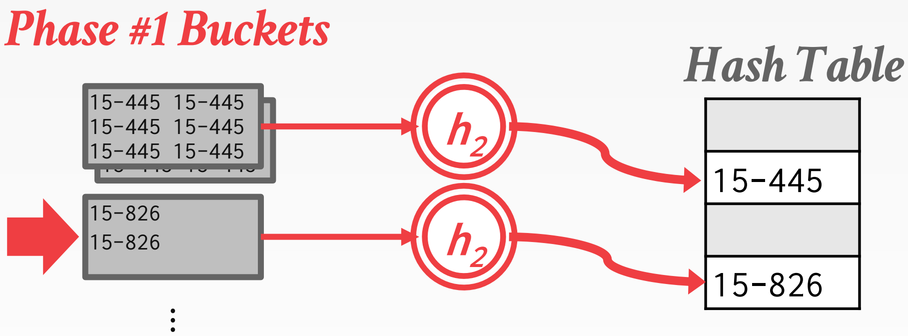
enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C



PHASE #2 – REHASH

```sql
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```



enrolled(sid,cid,grade)

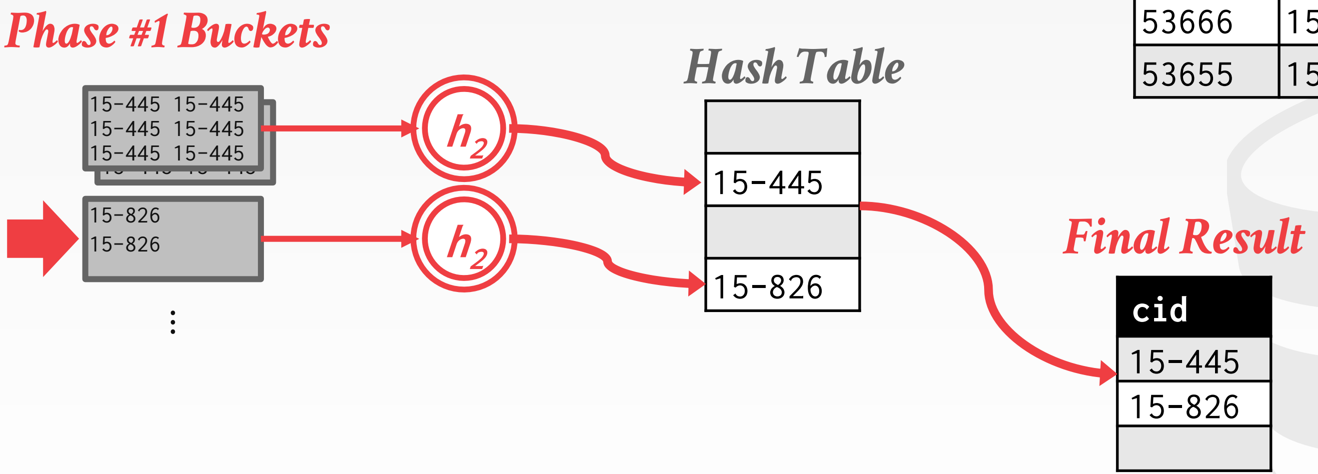
sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

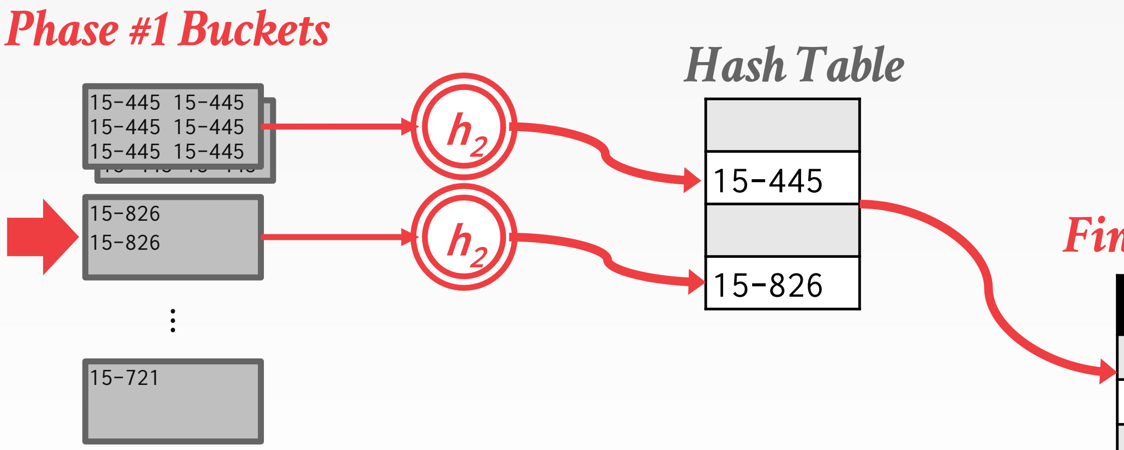


PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C



Final Result

cid
15-445
15-826

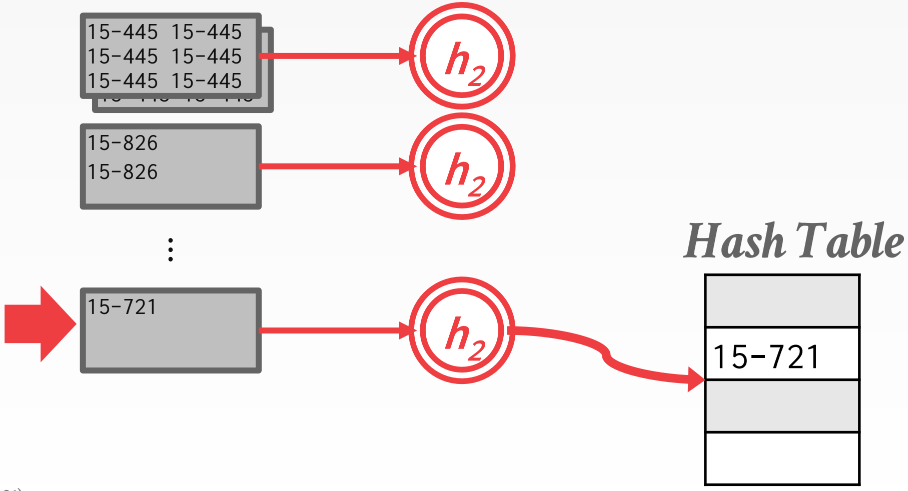
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



Final Result

cid
15-445
15-826

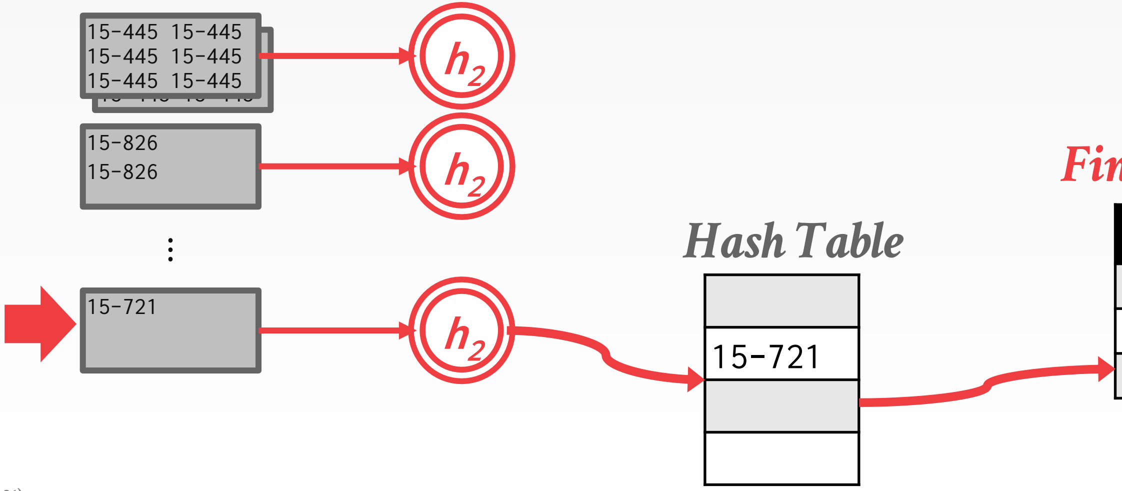
PHASE #2 – REHASH

```
SELECT DISTINCT cid
  FROM enrolled
 WHERE grade IN ('B','C')
```

enrolled(sid,cid,grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

Phase #1 Buckets



Final Result

cid
15-445
15-826
15-721

HASHING SUMMARIZATION

During the ReHash phase, store pairs of the form (**GroupKey→RunningVal**)

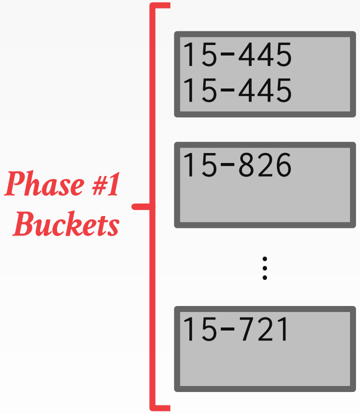
When we want to insert a new tuple into the hash table:

→ If we find a matching **GroupKey**, just update the **RunningVal** appropriately

→ Else insert a new **GroupKey→RunningVal**

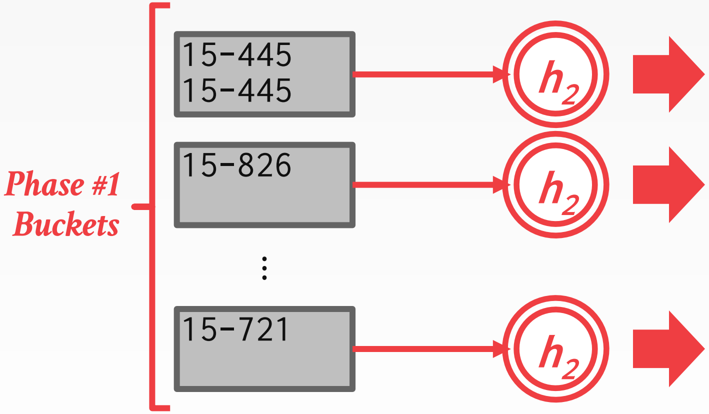
HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```



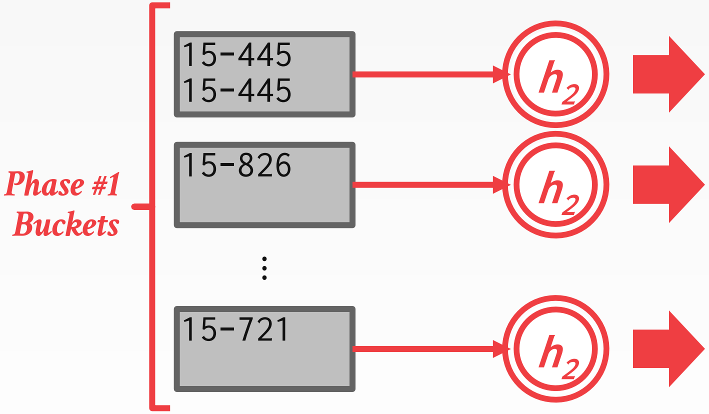
HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```



HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

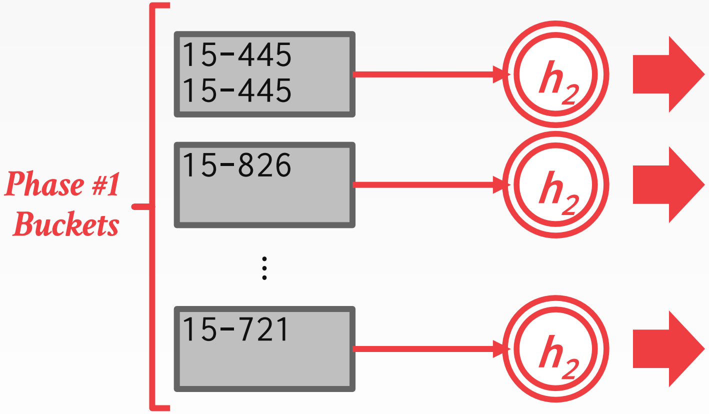


Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```



Hash Table

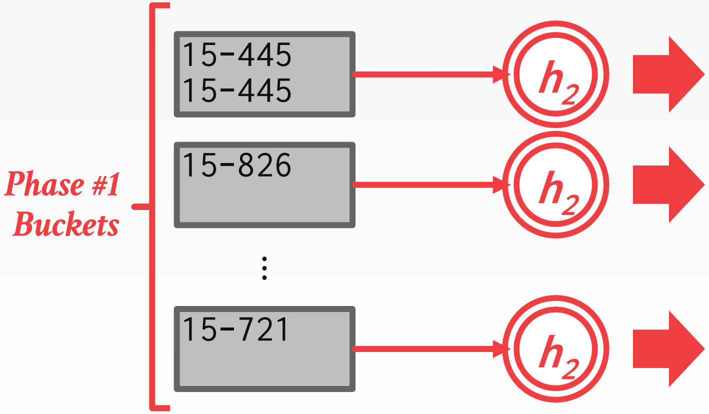
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

Running Totals

AVG(col) → (COUNT,SUM)
MIN(col) → (MIN)
MAX(col) → (MAX)
SUM(col) → (SUM)
COUNT(col) → (COUNT)



Hash Table

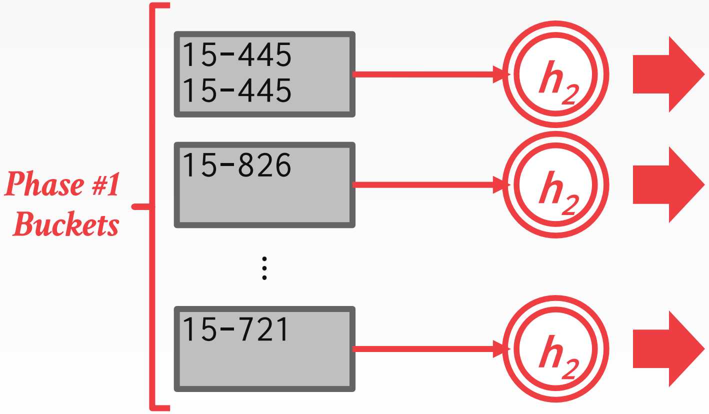
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

Running Totals

AVG(col) → (COUNT,SUM)
MIN(col) → (MIN)
MAX(col) → (MAX)
SUM(col) → (SUM)
COUNT(col) → (COUNT)



Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Final Result

cid	AVG(gpa)
15-445	3.66
15-826	3.33
15-721	2.89

CONCLUSION

Choice of sorting vs. hashing is subtle and depends on optimizations done in each case.

We already discussed the optimizations for sorting:

→ Chunk I/O into large blocks to amortize costs
→ Double-buffering to overlap CPU and I/O

NEXT CLASS

Nested Loop Join

Sort-Merge Join

Hash Join