M146 Database Systems
Spring 2021

Georgia Koutrika

SQL Execution

```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%';

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

--------------------------------------------------------------------
| Id| Operation                    | Name         |Rows|By
--------------------------------------------------------------------
| 0 | SELECT STATEMENT             |              |  3 | 189 | 7(15)| 00:00:01 |
|*1 |  HASH JOIN                   |              |  3 | 189 | 7(15)| 00:00:01 |
|*2 |   HASH JOIN                  |              |  3 | 141 | 5(20)| 00:00:01 |
| 3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |  3 |  60 | 2 (0)| 00:00:01 |
|*4 |     INDEX RANGE SCAN         | EMP_NAME_IX  |  3 |     | 1 (0)| 00:00:01 |
| 5 |    TABLE ACCESS FULL         | JOBS         | 19 | 513 | 2 (0)| 00:00:01 |
| 6 |   TABLE ACCESS FULL          | DEPARTMENTS  | 27 | 432 | 2 (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

Join methods: techniques to execute a join of two tables

Access paths: techniques for retrieving data from the database.

Execution plans and optimizers

SQL Execution

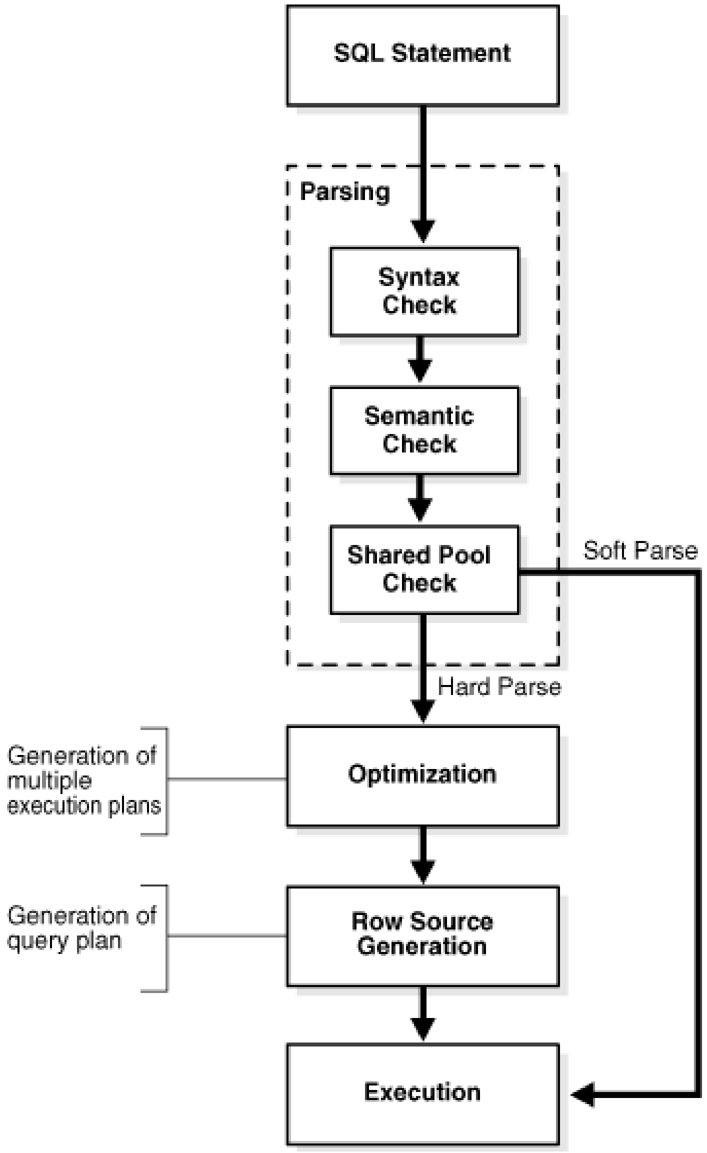
```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%';

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

--------------------------------------------------------------------------------
| Id| Operation                    | Name        |Rows|Bytes|Cost(%CPU)|Time    |
--------------------------------------------------------------------------------
| 0 | SELECT STATEMENT             |             |  3 | 189 | 7(15)| 00:00:01 |
|*1 |  HASH JOIN                   |             |  3 | 189 | 7(15)| 00:00:01 |
|*2 |   HASH JOIN                  |             |  3 | 141 | 5(20)| 00:00:01 |
| 3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |  3 |  60 | 2 (0)| 00:00:01 |
|*4 |     INDEX RANGE SCAN         | EMP_NAME_IX |  3 |     | 1 (0)| 00:00:01 |
| 5 |    TABLE ACCESS FULL         | JOBS        | 19 | 513 | 2 (0)| 00:00:01 |
| 6 |   TABLE ACCESS FULL          | DEPARTMENTS | 27 | 432 | 2 (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```



Other execution plans and optimizers

Cypher

```
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

```
+-----------------+------------------------+----------------+------+---------+------------------------+
| Operator        | Details                | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
+-----------------+------------------------+----------------+------+---------+------------------------+
| +ProduceResults | p                      |             13 |    1 |       0 |                    0/0 |
| |               +------------------------+----------------+------+---------+------------------------+
| +Filter         | p.name = $autostring_0 |             13 |    1 |     125 |                    0/0 |
| |               +------------------------+----------------+------+---------+------------------------+
| +NodeByLabelScan | p:Person              |            125 |  125 |     126 |                    0/0 |
+-----------------+------------------------+----------------+------+---------+------------------------+

Total database accesses: 251, total allocated memory: 0
```

Access path

Other execution plans and optimizers

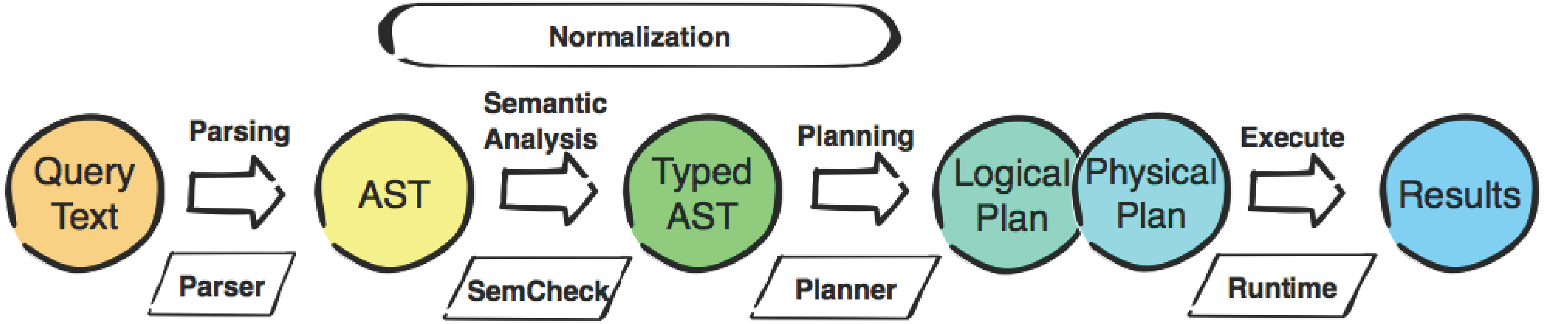
neo4j

Cypher

```
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

```
+-------------------+-----------------------------+----------------+------+---------+-------------------------+
| Operator          | Details                     | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
+-------------------+-----------------------------+----------------+------+---------+-------------------------+
| +ProduceResults   | p                           |             13 |    1 |       0 |                     0/0 |
| |                 +-----------------------------+----------------+------+---------+-------------------------+
| +Filter           | p.name = $autostring_0      |             13 |    1 |     125 |                     0/0 |
| |                 +-----------------------------+----------------+------+---------+-------------------------+
| +NodeByLabelScan  | p:Person                    |            125 |  125 |     126 |                     0/0 |
+-------------------+-----------------------------+----------------+------+---------+-------------------------+

Total database accesses: 251, total allocated memory: 0
```



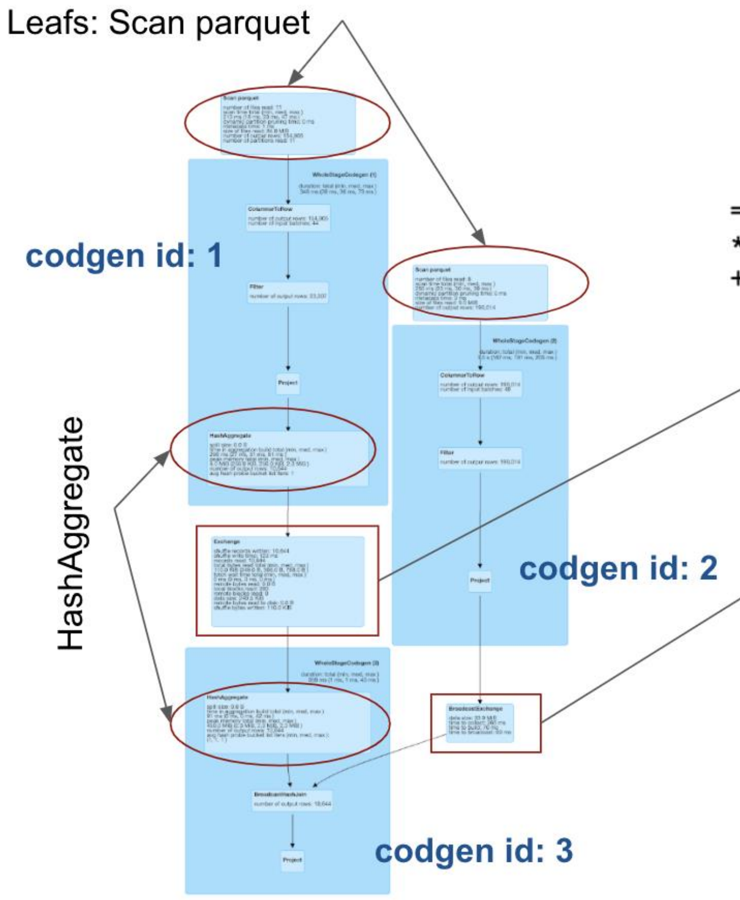
Cypher query execution occurs in the following steps:

1. Convert the input query string into an abstract syntax tree (AST)
2. Optimize and normalize the AST
3. Create a query graph from the normalized AST
4. Create a logical plan
5. Rewrite the logical plan
6. Create an execution plan from the logical plan
7. Execute the query using the execution plan

Other execution plans and optimizers

3.0 APACHE Spark

```
query = (
  questionsDF
  .filter(col('year') == 2019)
  .groupBy('user_id')
  .agg(
     count('*').alias('cnt')
  )
  .join(usersDF, 'user_id')
)
```



Join Method

```
== Physical Plan
* Project (14)
+- * BroadcastHashJoin Inner BuildRight (13)
   :- * HashAggregate (7)
   :  +- Exchange (6)
   :     +- * HashAggregate (5)
   :        +- * Project (4)
   :           +- * Filter (3)
   :              +- * ColumnarToRow (2)
   :                 +- Scan parquet  (1)
   +- BroadcastExchange (12)
      +- * Project (11)
         +- * Filter (10)
            +- * ColumnarToRow (9)
               +- Scan parquet  (8)
```

codgen id: 3

Aggregate Method

codgen id: 1

codgen id: 2

Access path

https://towardsdatascience.com/mastering-query-plans-in-spark-3-0-f4c334663aa4
http://spark.apache.org/docs/latest/sql-ref-syntax-qry-explain.html

Other execution plans and optimizers

```
EXPLAIN SELECT DISTINCT s.product_key, p.product_description
 FROM store.store_sales_fact s, public.product_dimension p
 WHERE s.product_key = p.product_key
 AND s.product_version = p.product_version
 AND s.store_key IN (
 SELECT store_key
 FROM store.store_dimension
 WHERE store_state = 'MA')
 ORDER BY s.product_key;
```

VERTICA

```
GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 3M, Rows: 15M] (PATH ID:
on
(PATH ID: 3) Inner (BROADCAST)
key)
| |       Materialize at Input: s.store_key
| |       Materialize at Output: s.product_key
| |       Execute on: All Nodes
| | +-- Outer -> JOIN HASH [Cost: 906K, Rows: 30M] (PATH ID: 4) Inner (BROADCAST)
| | |       Join Cond: (s.product_key = p.product_key) AND (s.product_version = p.product_version)
| | |       Execute on: All Nodes
| | | +-- Outer -> STORAGE ACCESS for s [Cost: 893K, Rows: 30M] (PATH ID: 5)
| | | |       Projection: store.store_sales_fact_b0
| | | |       Materialize: s.product_key, s.product_version
| | | |       Execute on: All Nodes
| | | |       Runtime Filters: (SIP2(HashJoin): s.product_key), (SIP3(HashJoin): s.product_version),
(SIP4(HashJoin): s.product_key, s.product_version), (SIP1(HashJoin): s.store_key)
| | | +-- Inner -> STORAGE ACCESS for p [Cost: 365, Rows: 60K] (PATH ID: 6)
| | | |       Projection: public.product_dimension_b0
| | | |       Materialize: p.product_key, p.product_version, p.product_description
| | | |       Execute on: All Nodes
| | +-- Inner -> SELECT [Cost: 68, Rows: 32] (PUSHED GROUPING) (PATH ID: 7)
| | |       Execute on: All Nodes
| | | +---> STORAGE ACCESS for store_dimension [Cost: 68, Rows: 32] (PATH ID: 8)
| | | |       Projection: store.store_dimension_b0
| | | |       Materialize: store_dimension.store_key
n.store_state = 'MA')
```

Operator

Cost

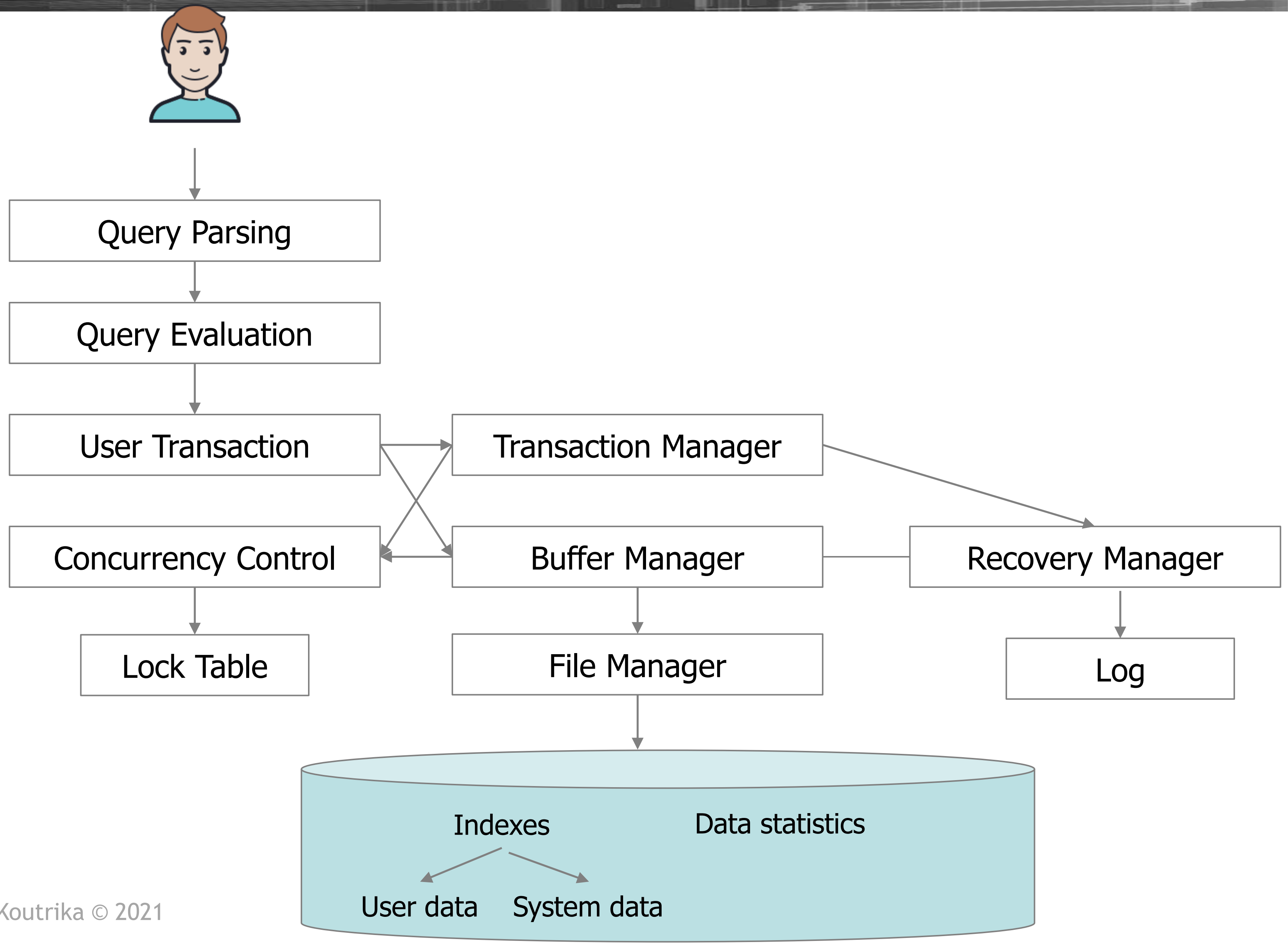
Projection

Column Materialization

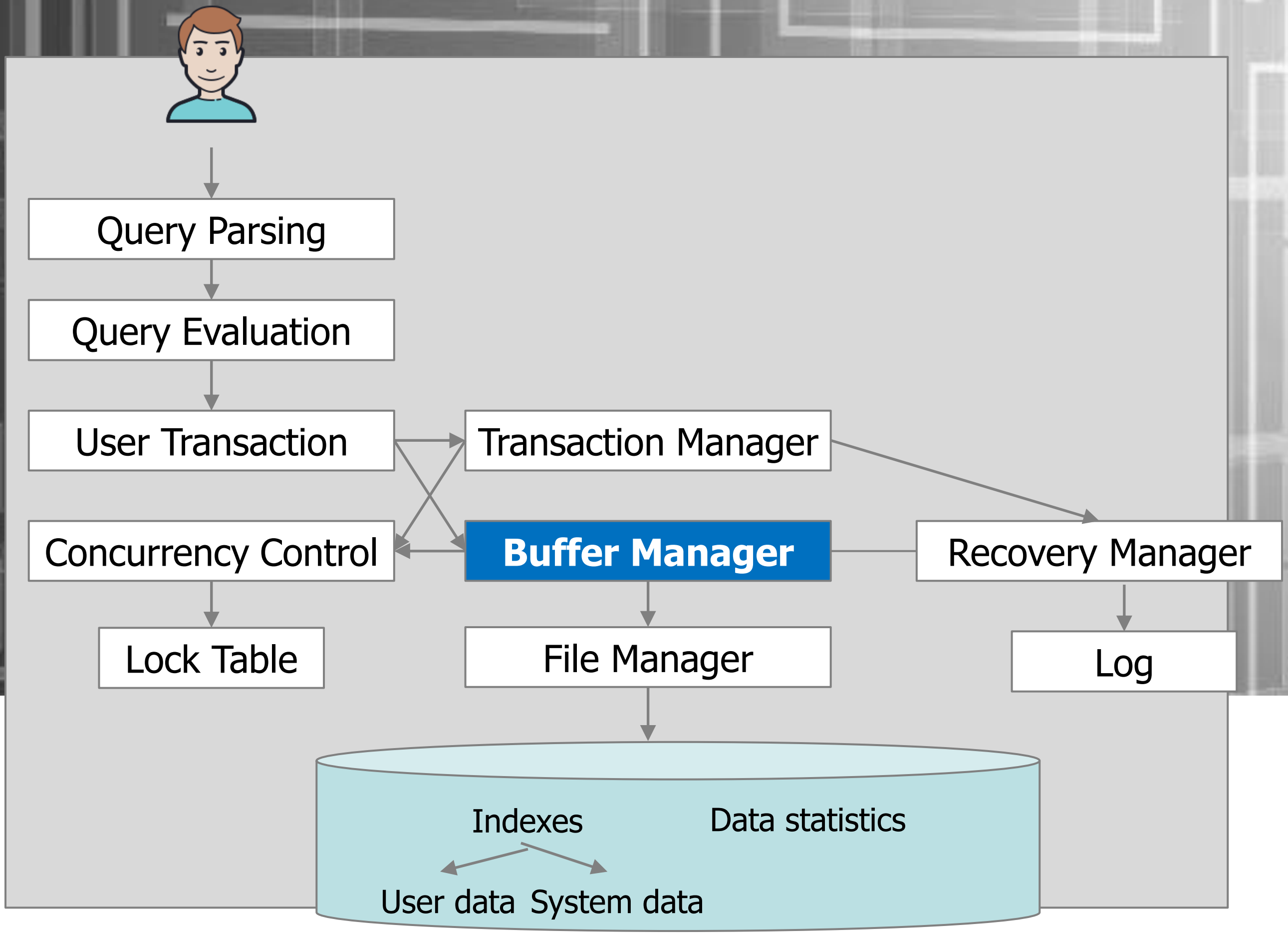
Path ID

http://vertica.com/kb/Reading-Query-Plans/Content/BestPractices/Reading-Query-Plans.htm
https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/AdministratorsGuide/QueryManagement/QueryPlans/QueryPlanInfo/QueryPlansVsql.htm

A high-level architecture of a DBMS

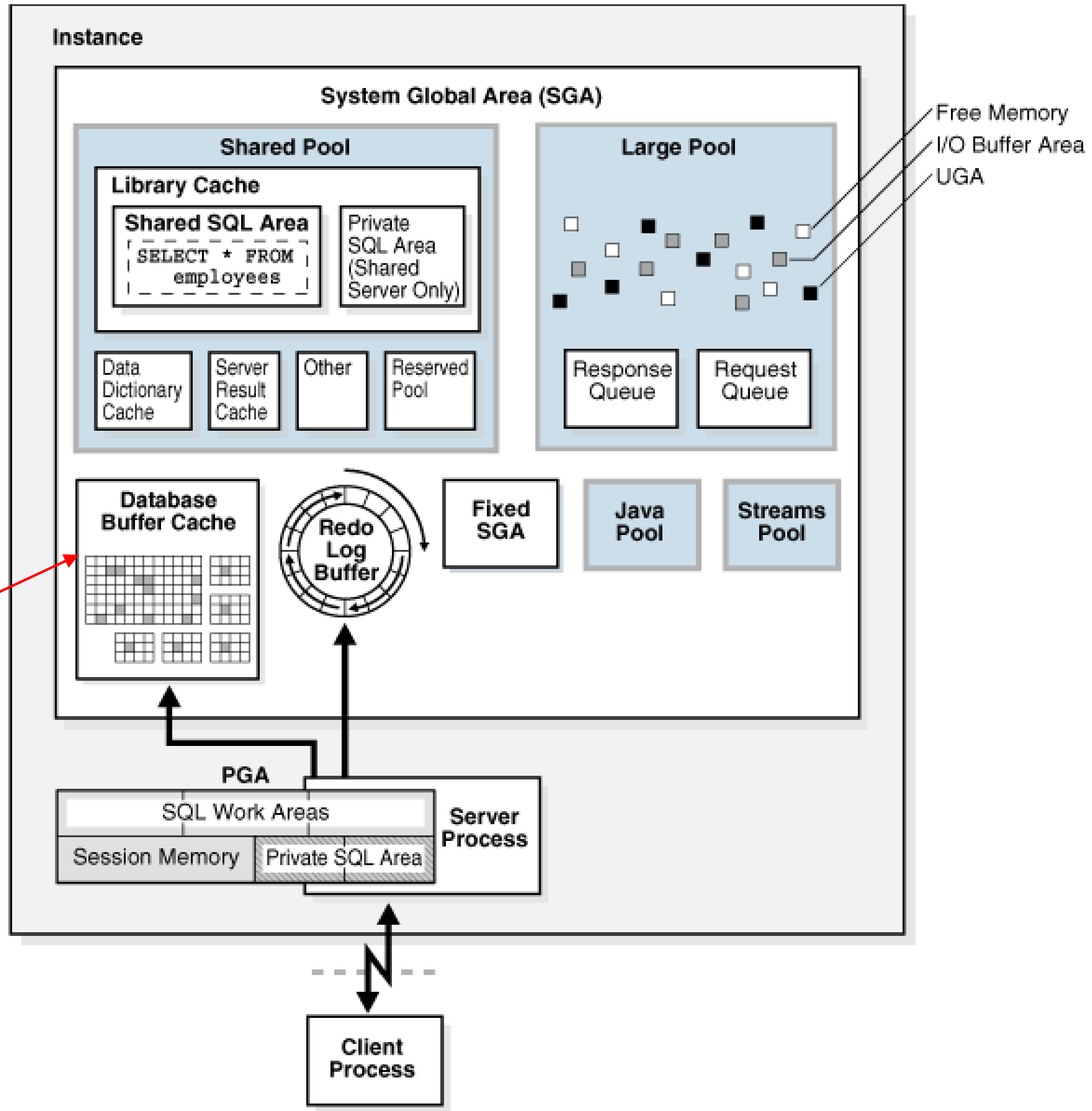


BUFFER MANAGEMENT

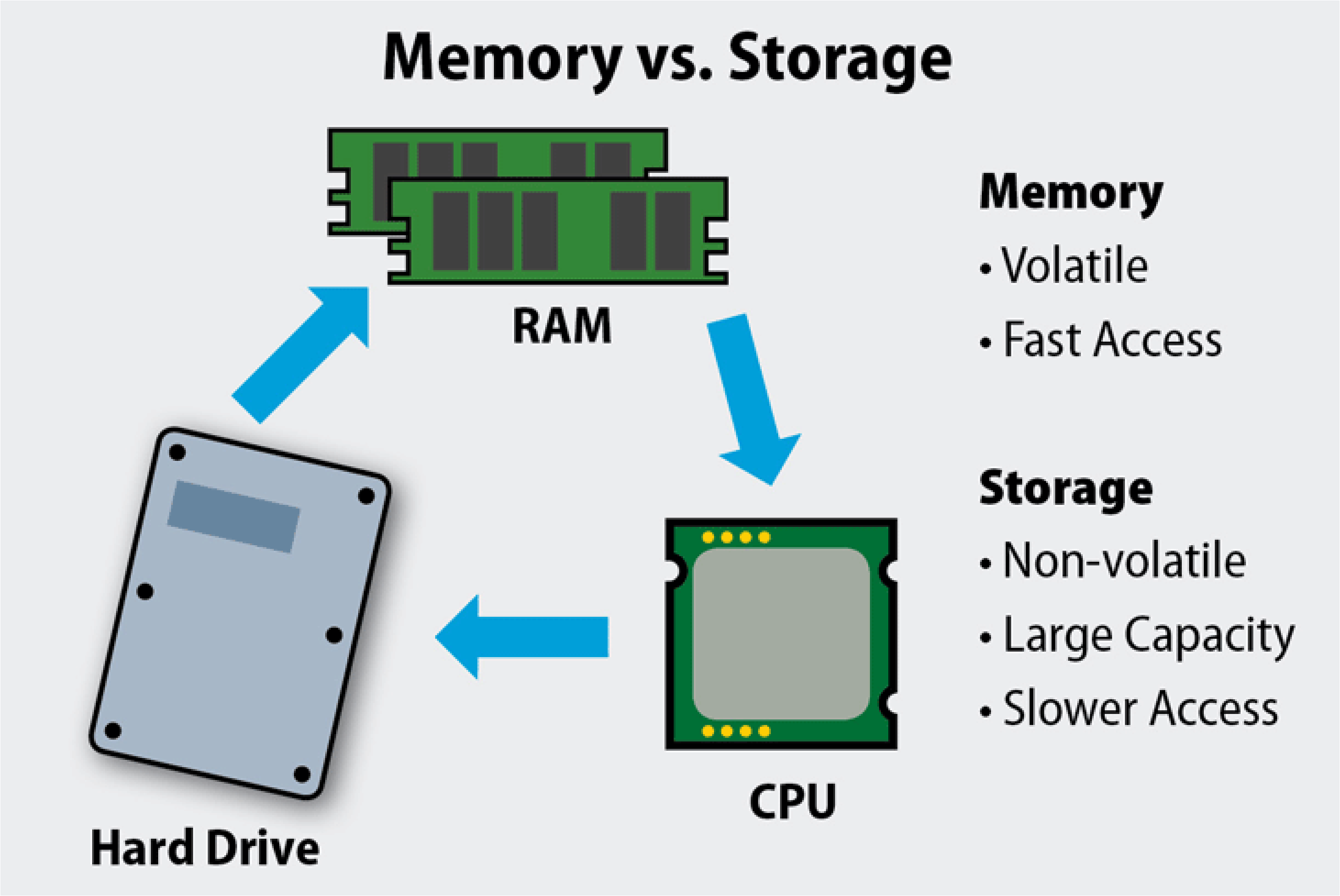


Buffer Basics

Buffer
Keeps data from the disk



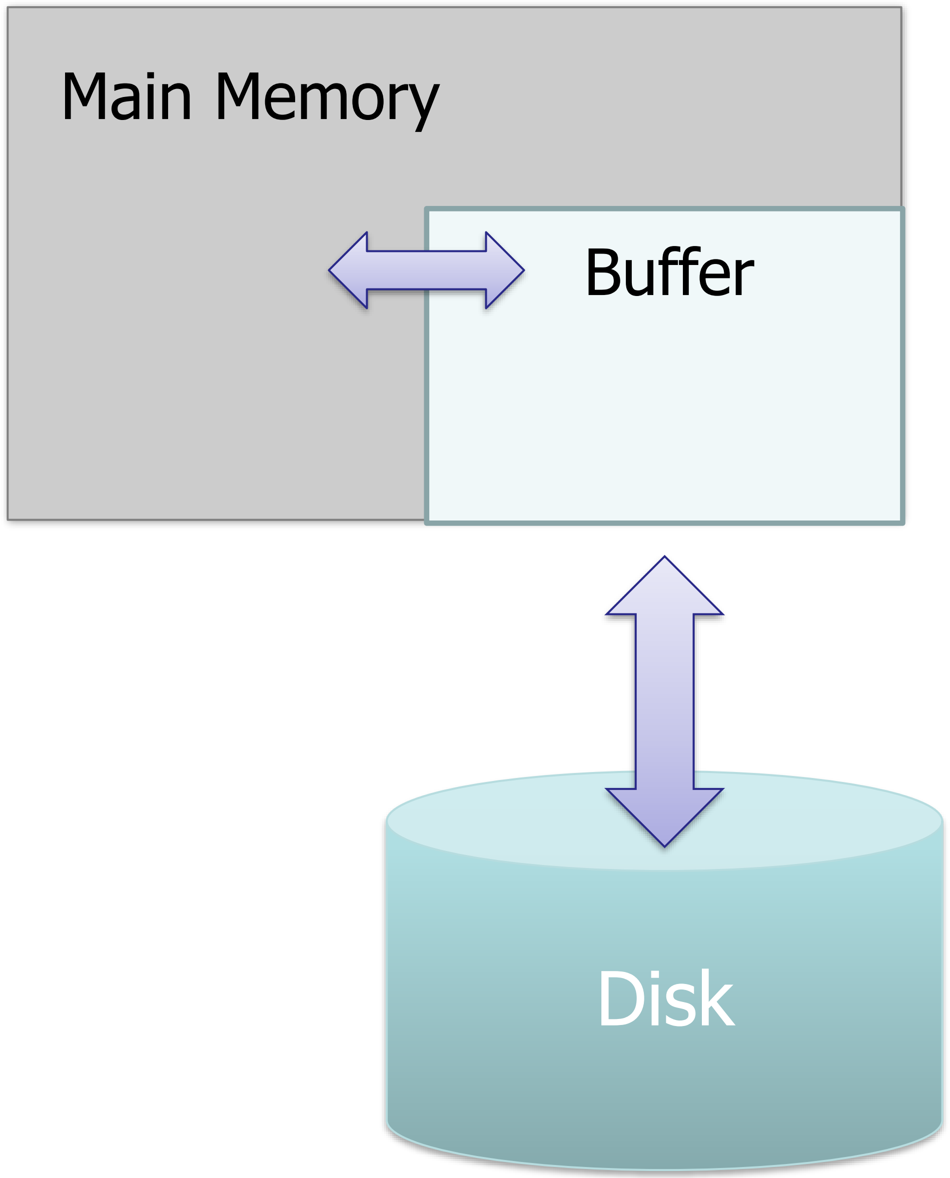
High-level: Disk vs. Main Memory



The Buffer

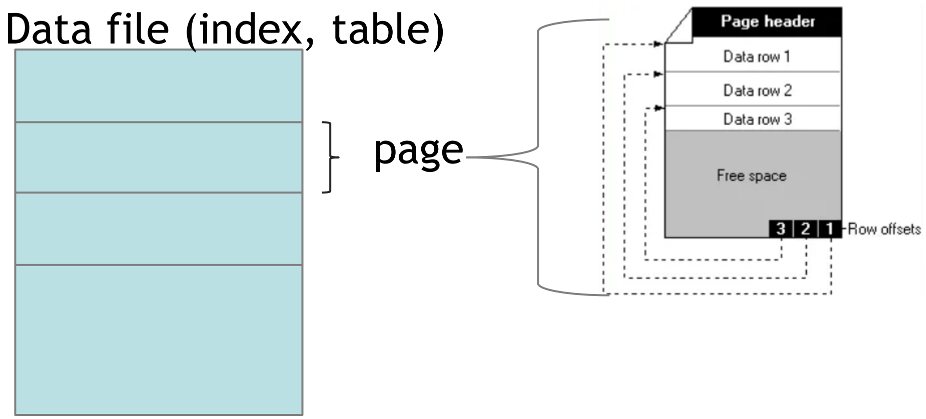
- A **<u>buffer</u>** is a region of physical memory used to store **intermediate data between disk and processes**

- *Key idea*: Reading / writing to disk is slow- need to cache data!



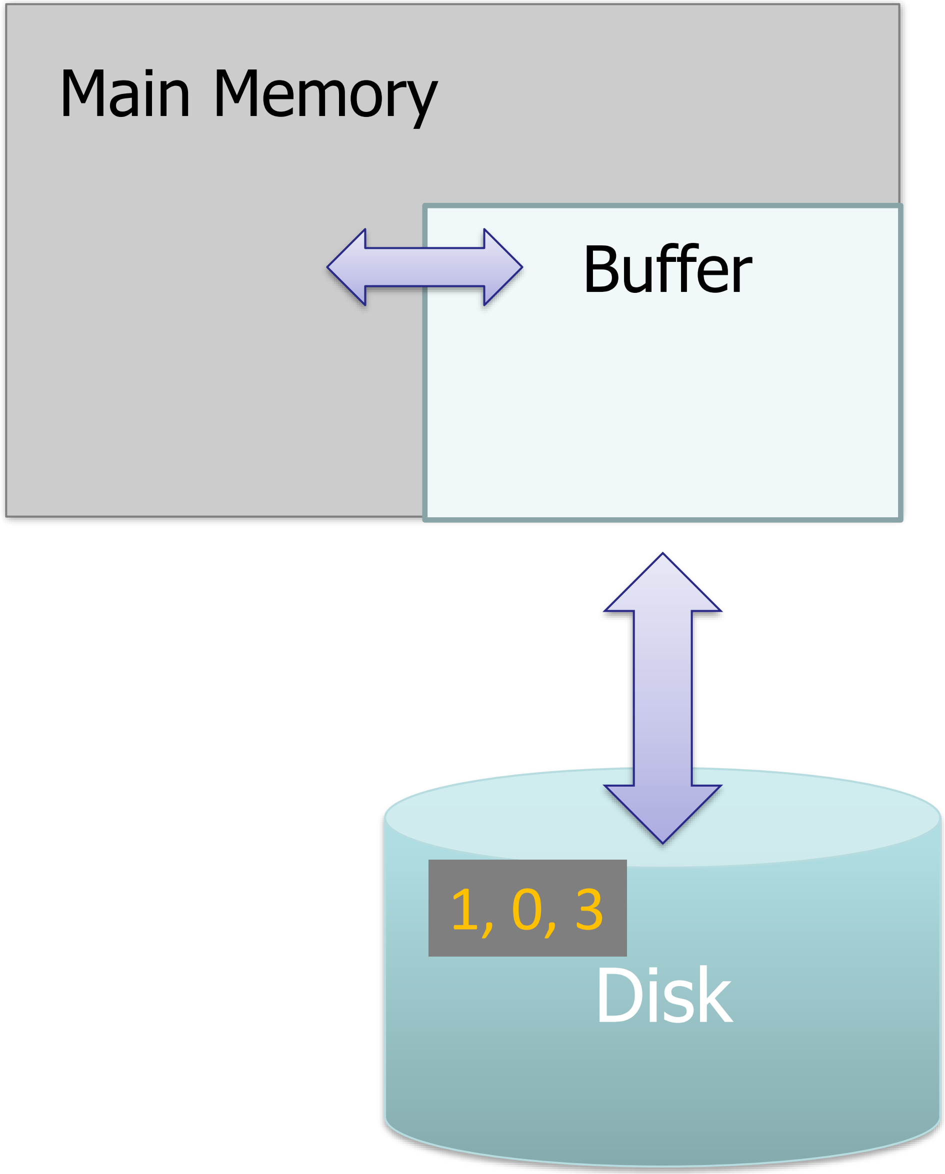
The (Simplified) Buffer

- We’ll consider a buffer located in **main memory** that operates over **pages** and **files**

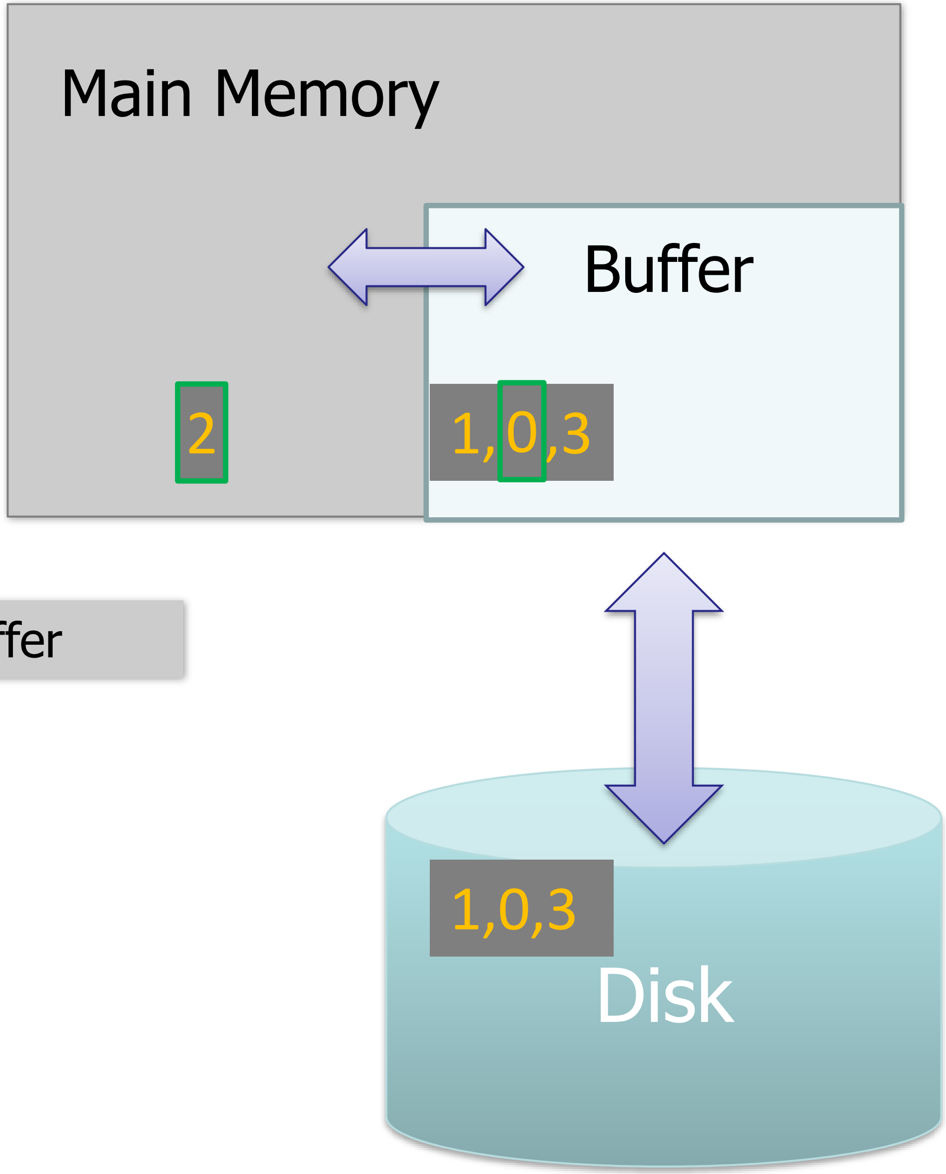


Basic Buffer Operations

- **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*



Basic Buffer Operations

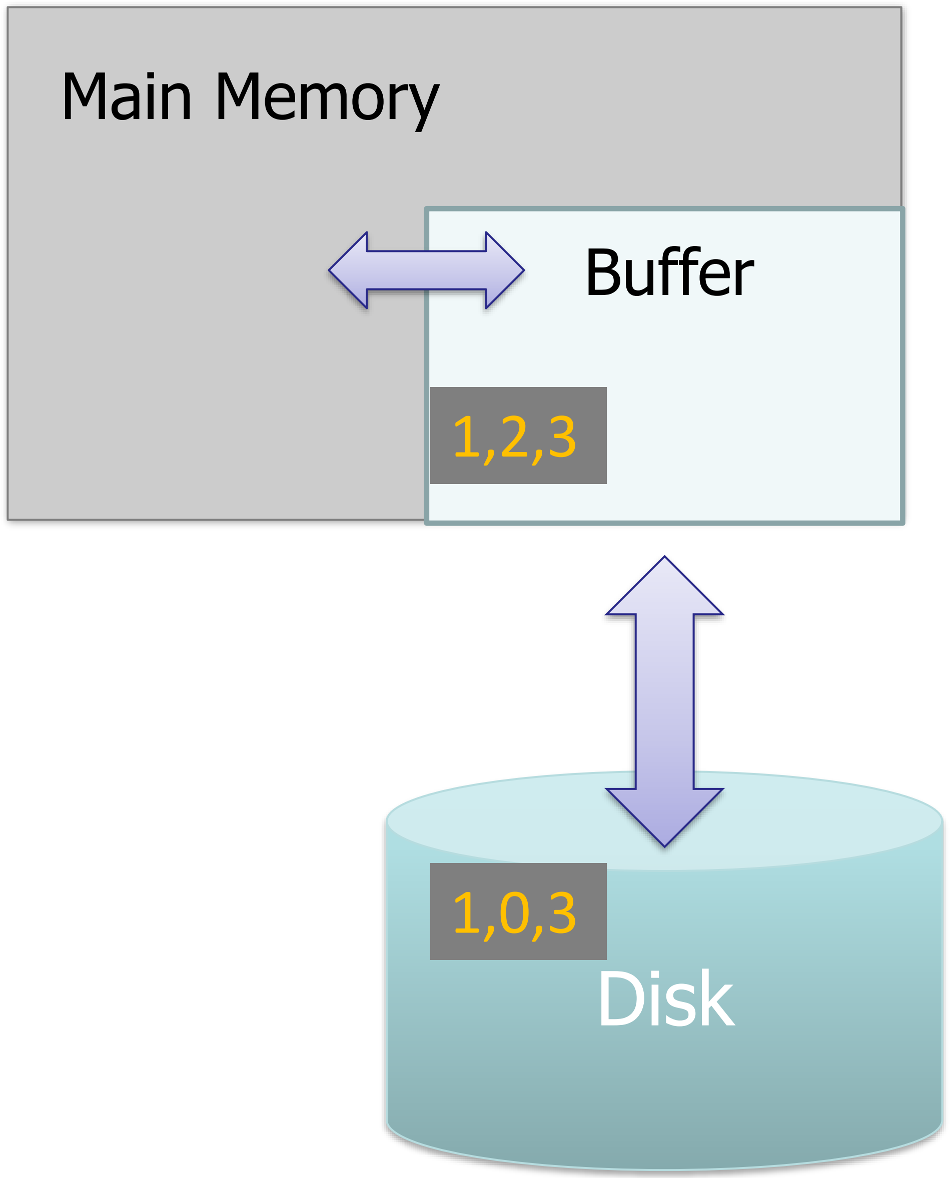


- **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*

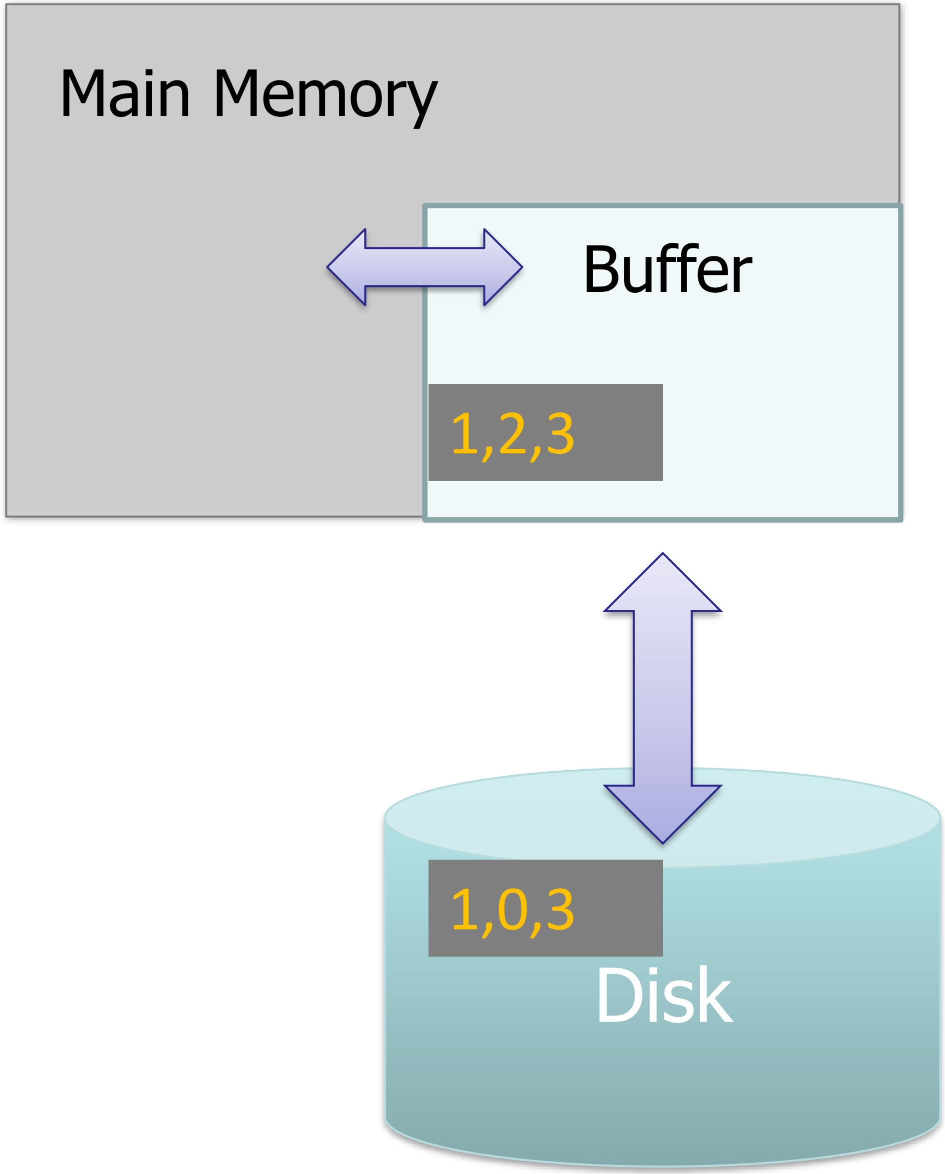
Processes can then read from / write to the page in the buffer

Basic Buffer Operations

- **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
- **<u>Flush(page):</u>** Evict page from buffer & write to disk



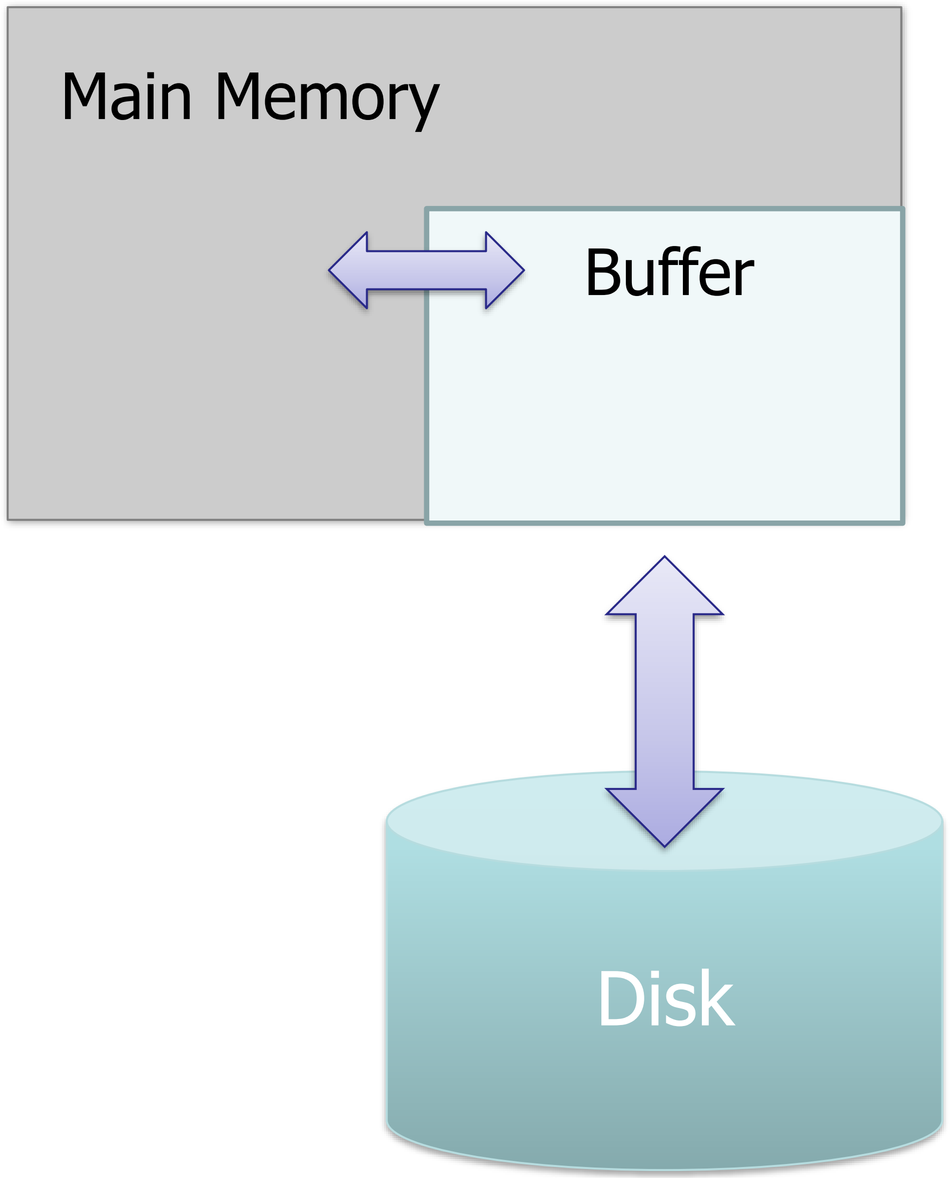
Basic Buffer Operations



- **<u>Read(page):</u>** Read page from disk -> buffer *if not already in buffer*
- **<u>Flush(page):</u>** Evict page from buffer & write to disk
- **<u>Release(page):</u>** Evict page from buffer *without* writing to disk

Managing Disk: The DBMS Buffer

- Database maintains its own buffer
  - Why? The OS already does this…



The Buffer Manager

- A **buffer manager** handles supporting operations for the buffer:
  - Primarily, handles & executes the "replacement policy"
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

How to pick a page to release?

Examples:

- **Least Recently Used** (LRU)
  - Order pages by the time of last accessed
  - Always replace the least recently accessed

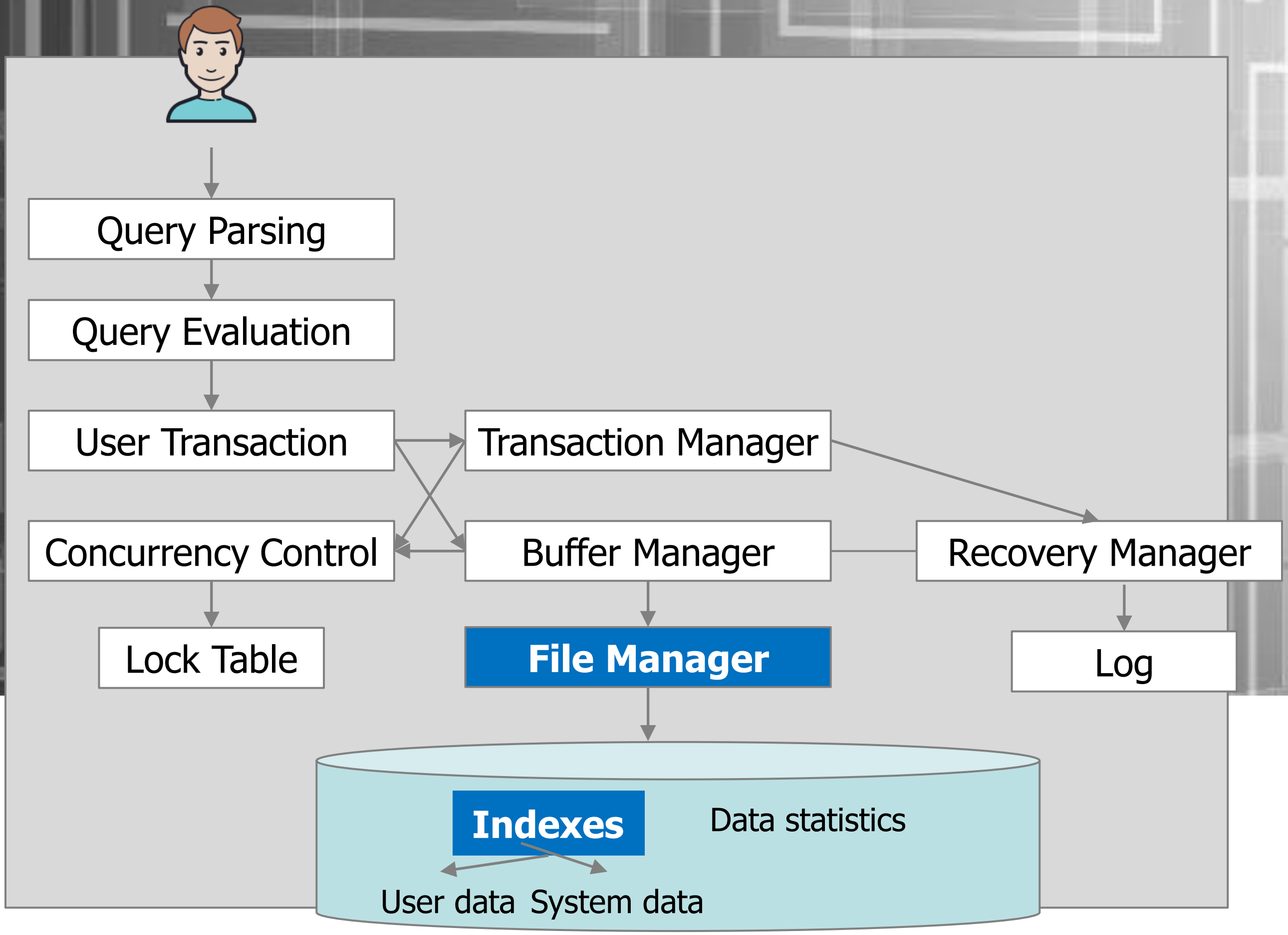
- **Most Recently Used** (MRU)

  MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

Other more sophisticated strategies exist

INDEXES

How does the DB access data fast?



SQL Processing

Access paths/Indexes:

Recall that we see indexes used in the query plan for the execution of a query

```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%';

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

--------------------------------------------------------------------------------
| Id| Operation                     | Name        |Rows|Bytes|Cost(%CPU)|Time    |
--------------------------------------------------------------------------------
| 0 | SELECT STATEMENT              |             |  3 | 189 | 7(15)| 00:00:01 |
|*1 |  HASH JOIN                    |             |  3 | 189 | 7(15)| 00:00:01 |
|*2 |   HASH JOIN                   |             |  3 | 141 | 5(20)| 00:00:01 |
| 3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |  3 |  60 | 2 (0)| 00:00:01 |
|*4 |     INDEX RANGE SCAN          | EMP_NAME_IX |  3 |     | 1 (0)| 00:00:01 |
| 5 |    TABLE ACCESS FULL          | JOBS        | 19 | 513 | 2 (0)| 00:00:01 |
| 6 |   TABLE ACCESS FULL           | DEPARTMENTS | 27 | 432 | 2 (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

Access paths: techniques for retrieving data from the database.

Index Motivation

Person(firstname, lastname, age)

- Suppose we want to **search** for people with a specific lastname

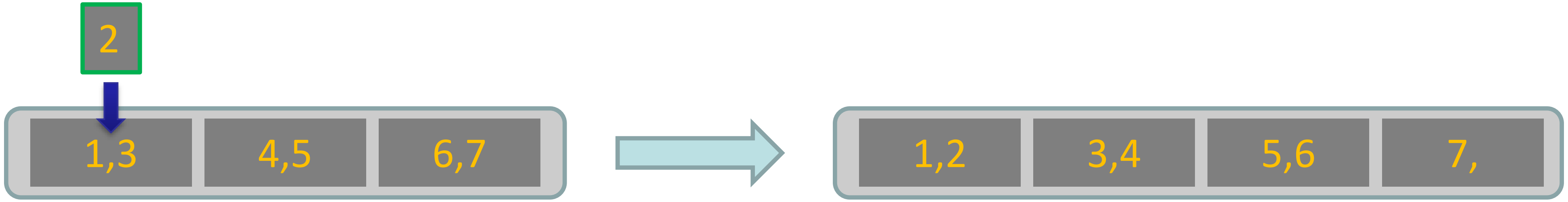
- ***First idea:*** Sort the records by lastname

- How many IO operations to search over ***N sorted*** records?
  - Simple scan: ***O(N)***
  - Binary search: $O(\log_2 N)$

Could we get cheaper search?

Index Motivation

- What about if we want to **insert** a new record, but keep the list sorted?



- We would have to potentially shift ***N*** records, requiring up to ~ **2*N/P** IO operations (where P = # of records per page)!
  - We could leave some "slack" in the pages…

Could we get faster insertions?

Index Motivation

- What about if we want to be able to **search along multiple attributes** (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set
  - … this would take a lot of space

Can we get fast search over multiple attributes
without taking too much space?

Create separate data structures called ***indexes*** to address all these points

What is an index

- An *index* is a **data structure** that maps:
  a tuple of search keys to sets of rows in a database table
  - Provides efficient lookup & retrieval by search key value
  - Usually much faster than searching through all the rows of the database table
- The index is called an access path on the field

Indexes: High-level

- *Example:*

Product(<u>name</u>, maker, price)

On which attributes would you build indexes?

Operations on an Index

- Search: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well.

- Insert / Remove entries
  - Bulk Load / Delete.

Indexing is one the most important features provided by a database for performance

Conceptual Example

What if we want to return all books published after 1867?

```
SELECT *
FROM Books
WHERE Published > 1867
```

Books

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

This table might be very expensive to search over row-by-row…

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Books

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Books

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Composite Keys

- Pros:
  - When they work, they work well
- Cons:
  - Guesses?

Primary Index

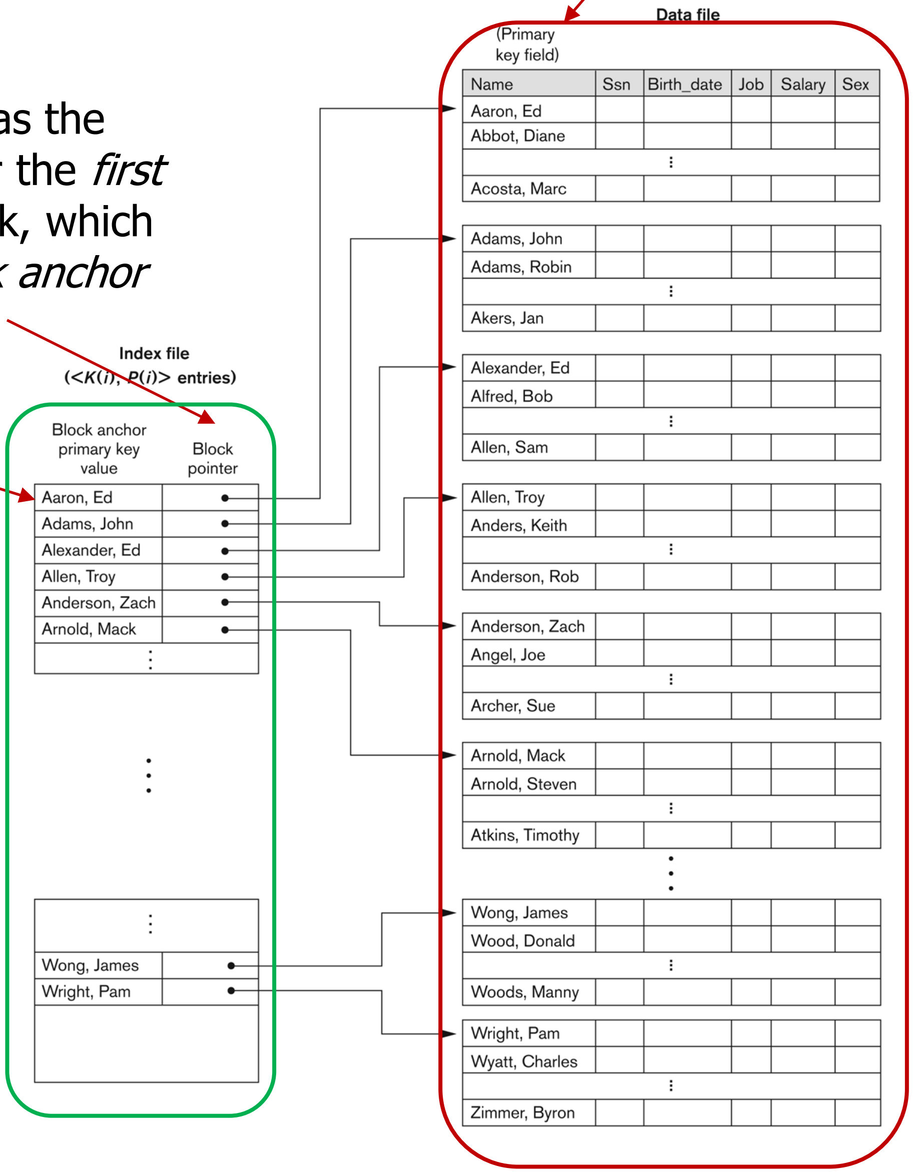
```
CREATE TABLE employees
  ( name VARCHAR(50) PRIMARY KEY,
    ssn VARCHAR(50) NOT NULL,
    birth_date VARCHAR(50),
    salary INT);
```

Primary Index

an ordered data file on the primary key

the index entry has the key field value for the *first record* in the block, which is called the *block anchor*

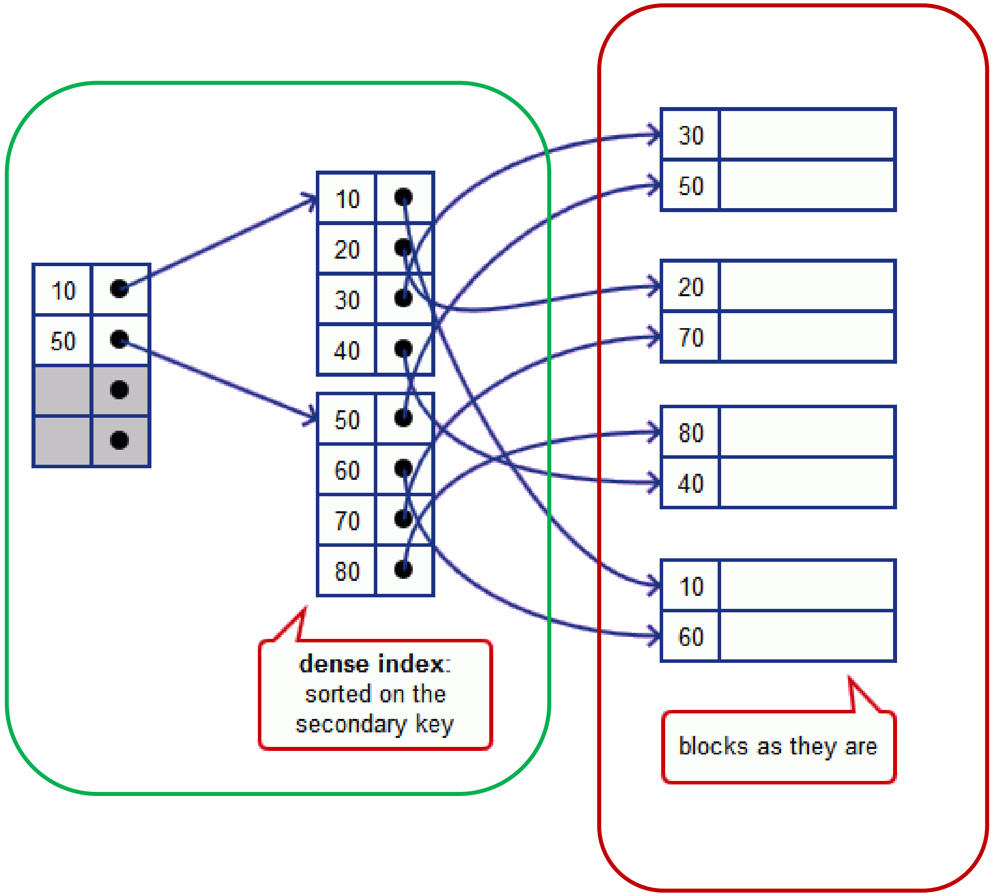
one index entry for each block in the data file



A primary index is a **nondense (sparse) index** since it includes an entry for each disk block of the data file rather than for every search value

Secondary Index

- It provides **a secondary means of accessing a file** for which some primary access already exists.
- It may be on **a candidate key** with unique values in every record, **or a non-key** with duplicate values.
- The index is an ordered file with two fields.
  - The first field is the **key**.
  - The second field is either a **block** pointer or a **record** pointer.
  - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
- Includes one entry **for each record** in the data file; hence, it is a **dense index**



Secondary Index

```
CREATE TABLE employees
  ( name VARCHAR(50) PRIMARY KEY,
    ssn VARCHAR(50) NOT NULL,
    birth_date VARCHAR(50),
    salary INT);
```

```
CREATE INDEX idx_ssn
ON employees (ssn);
```

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **covering** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

High-level Categories of Index Types

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called **B+ Trees***

- Hash Tables *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

Real difference between structures: costs of ops *determines which index you pick and why*

INDEXES

B-TREES

What you will learn about in this section

1. B+ Trees: Basics

2. B+ Trees: Design & Cost

3. Clustered Indexes

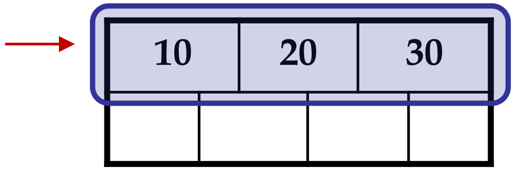
B+ Trees

- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

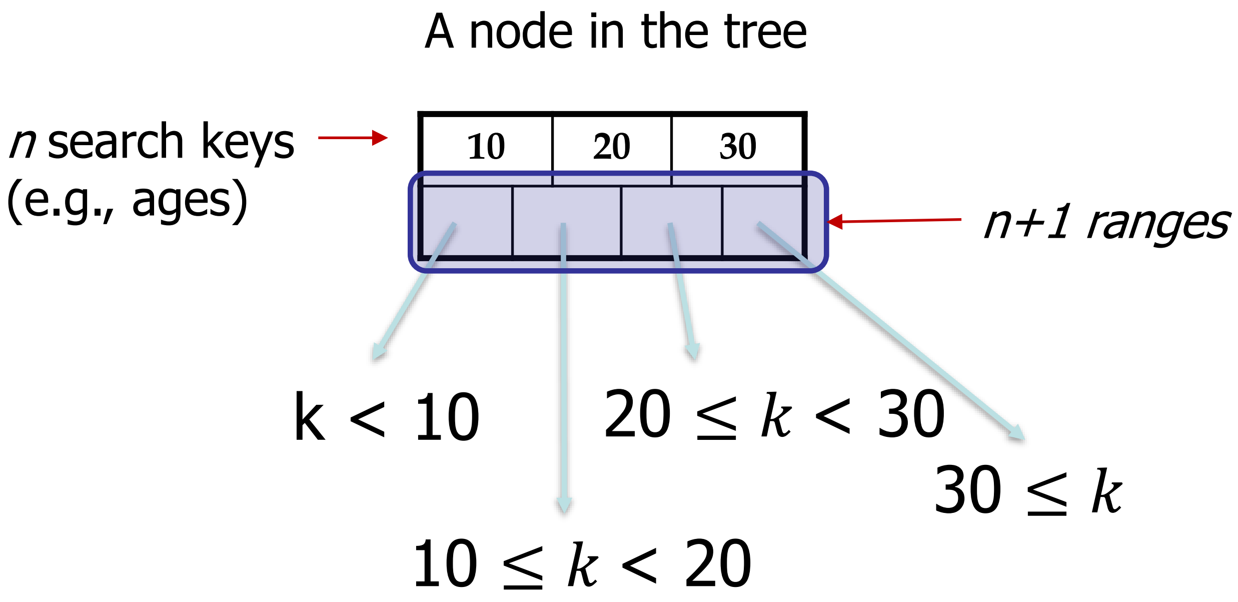
B+ Tree Basics

A node in the tree

n search keys
(e.g., ages)



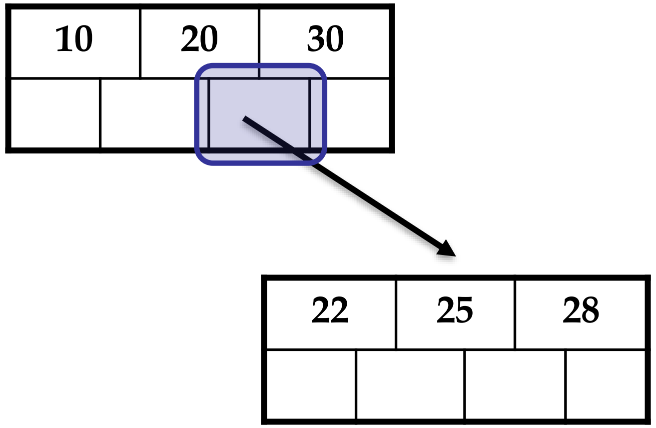
B+ Tree Basics



The *n* keys in a node define *n+1* ranges

B+ Tree Basics

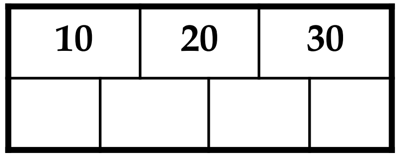
Non-leaf or *internal* node



For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics

A node in the tree

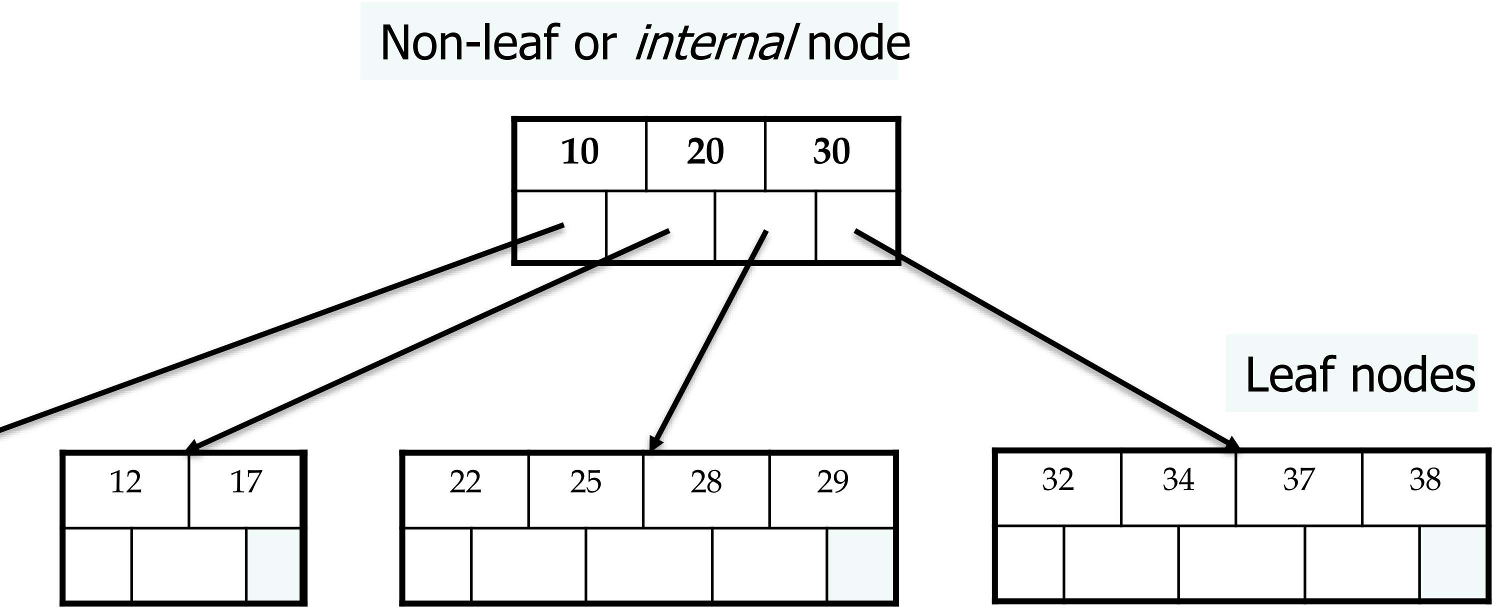


Parameter ***d*** = the degree

Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****

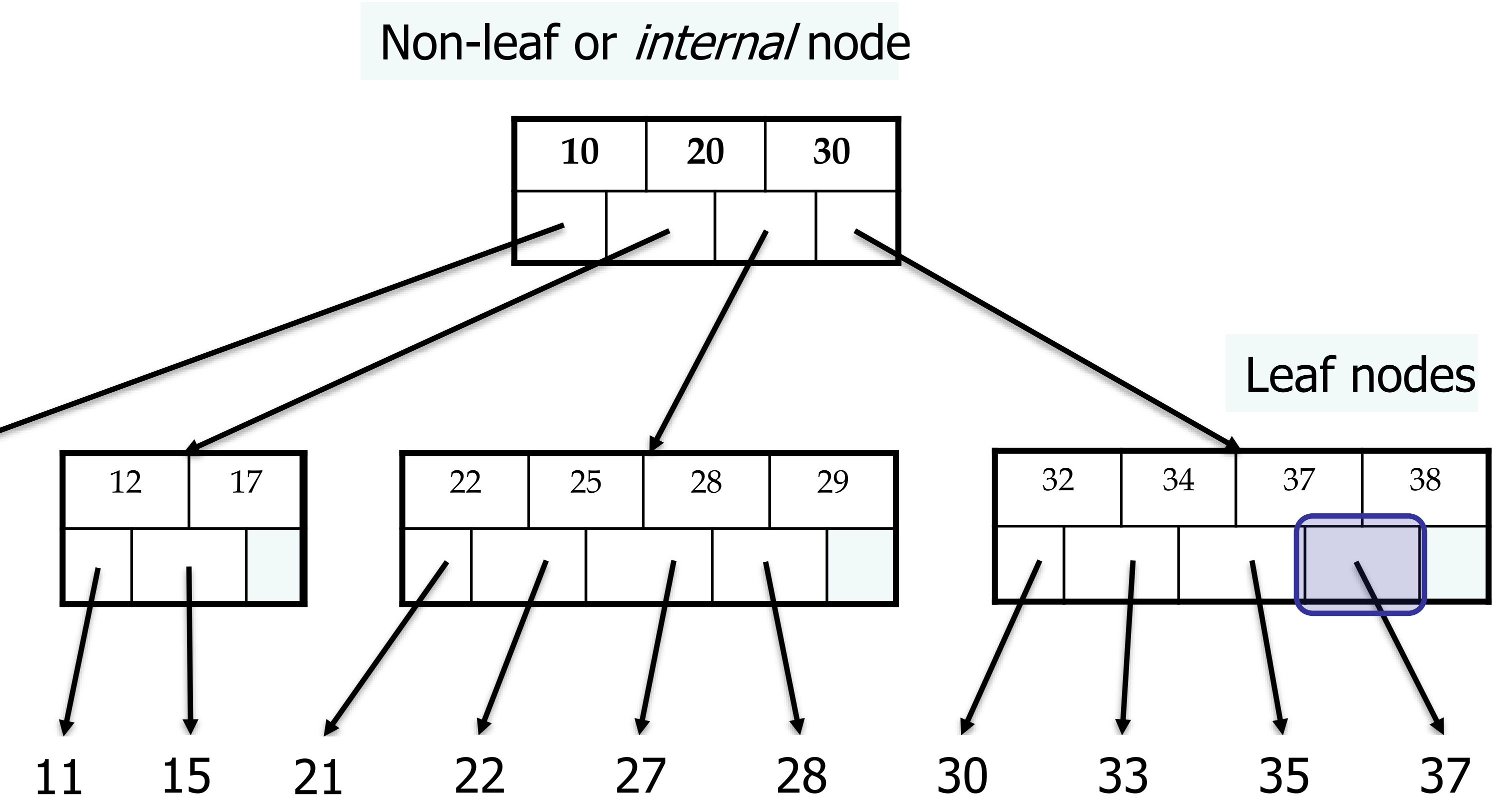
except for root node, which can have between **1 and 2d keys*

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

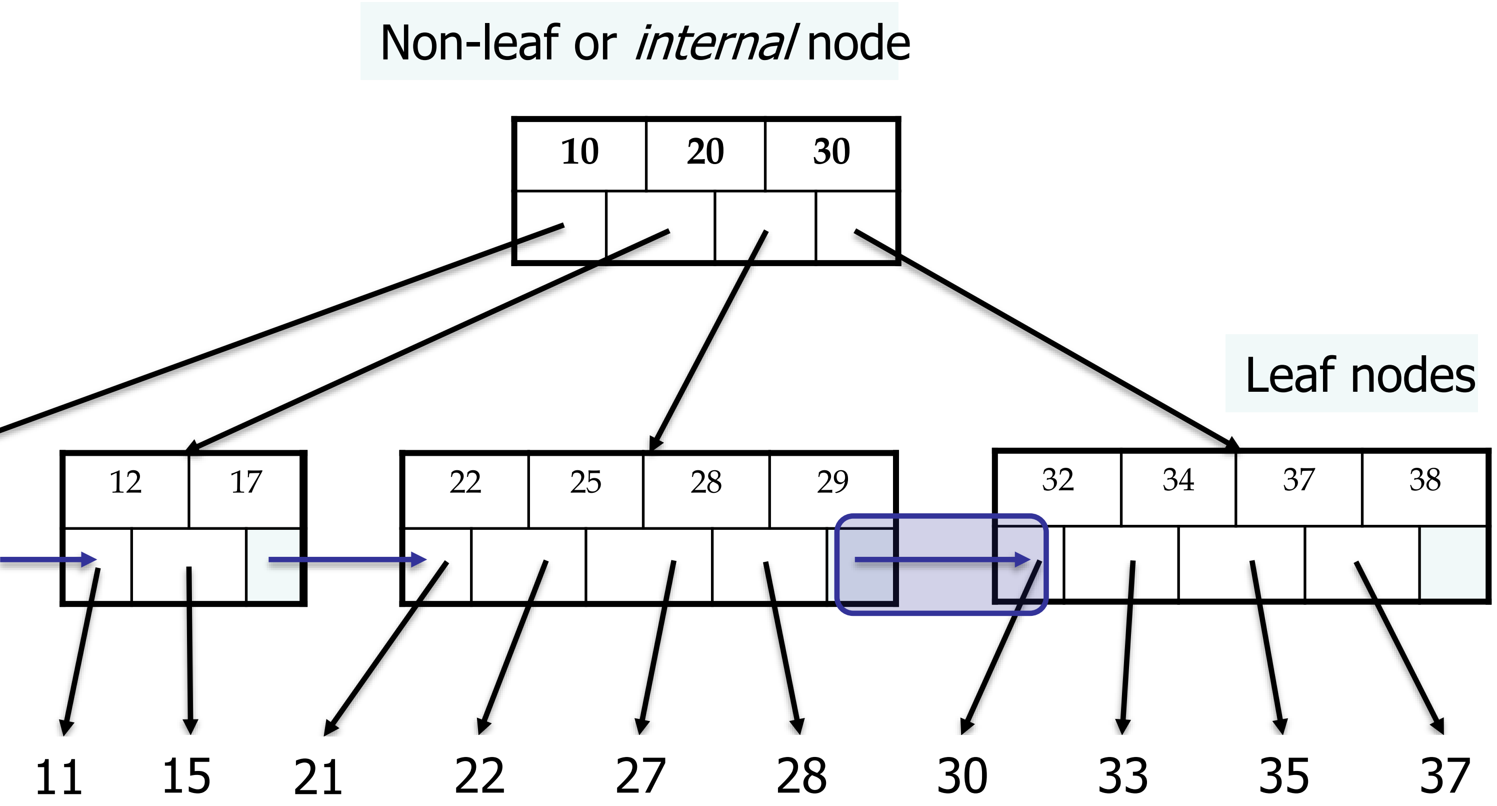
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

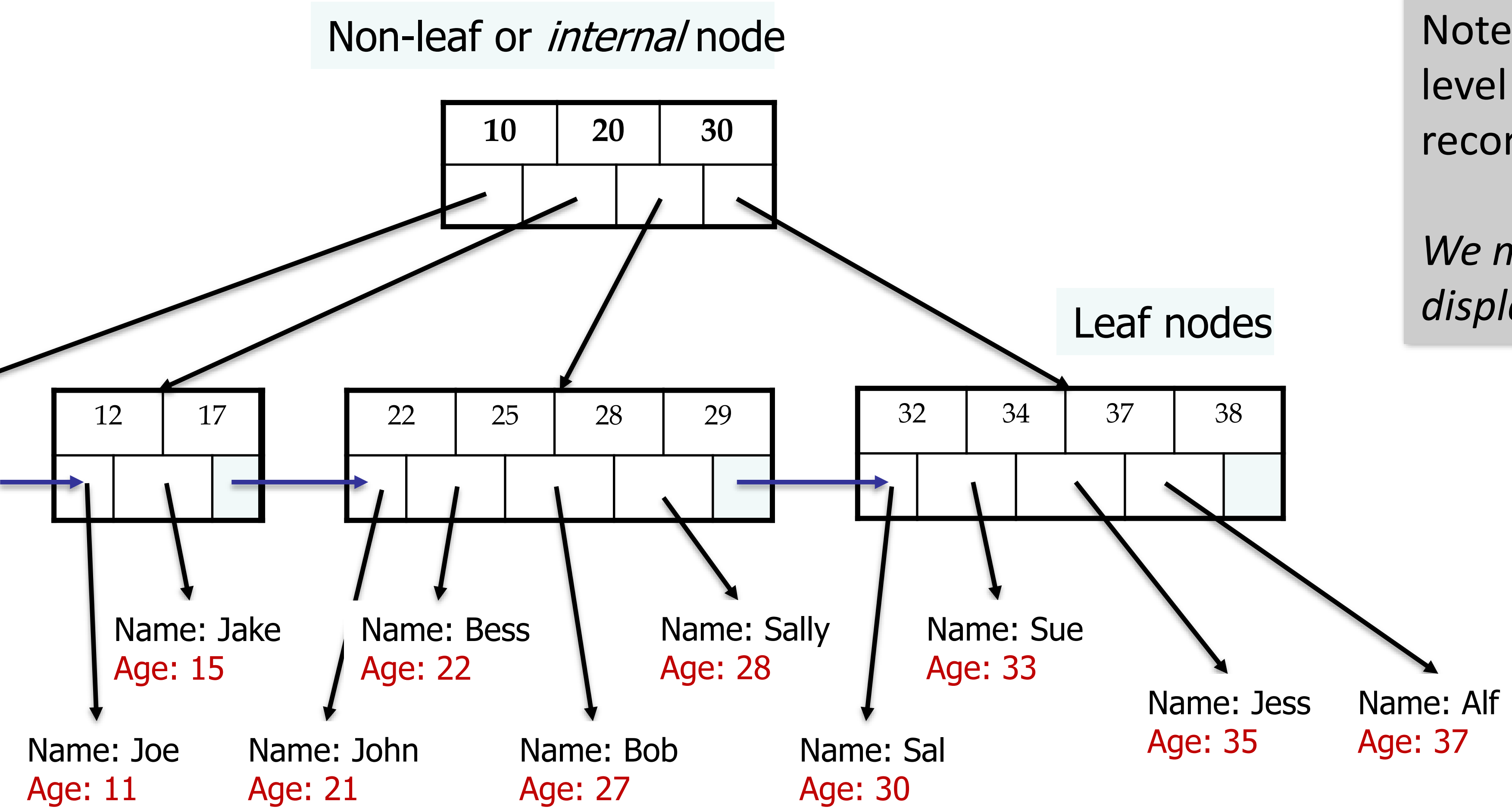


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  age = 25
```

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree Exact Search Animation

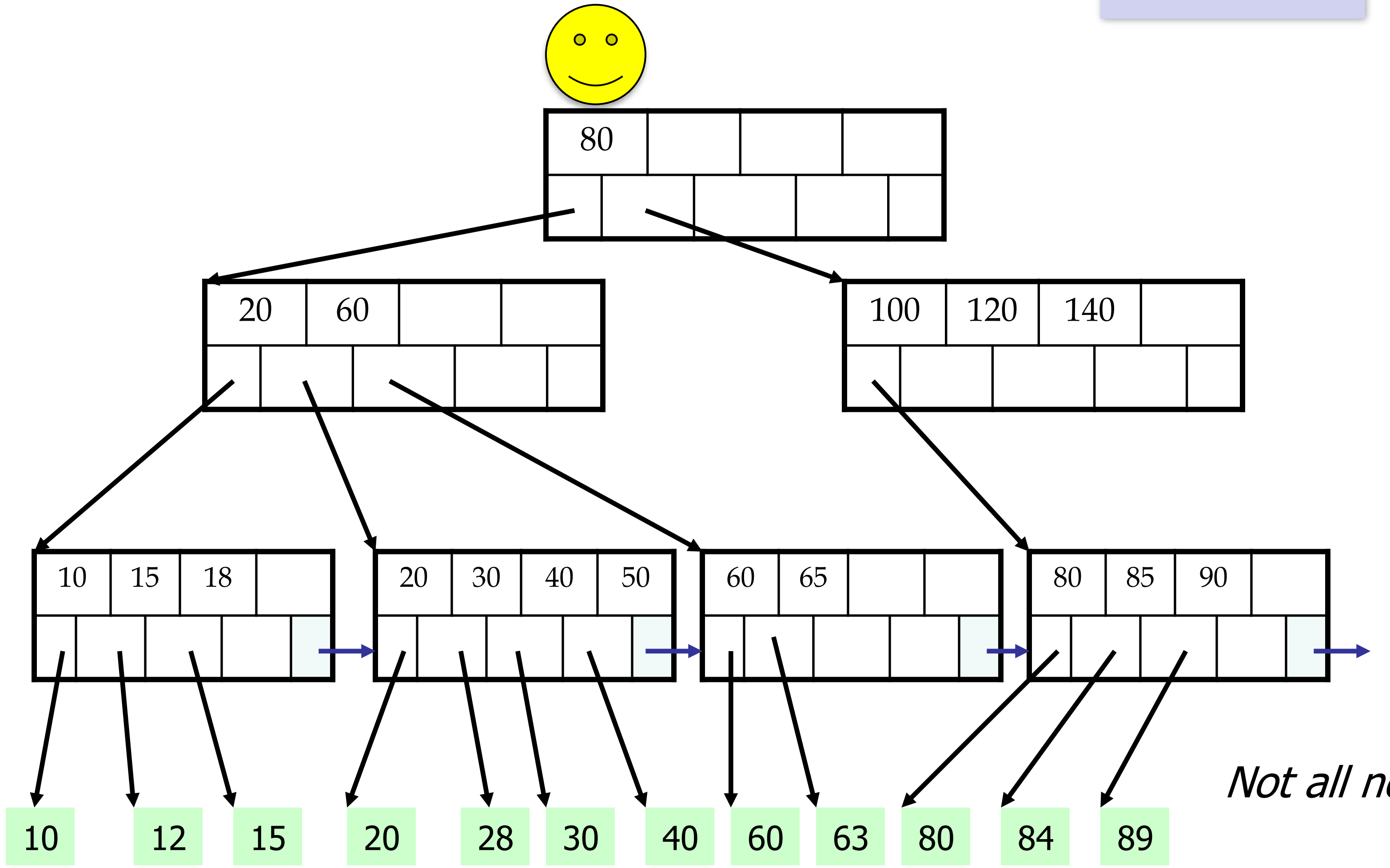
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

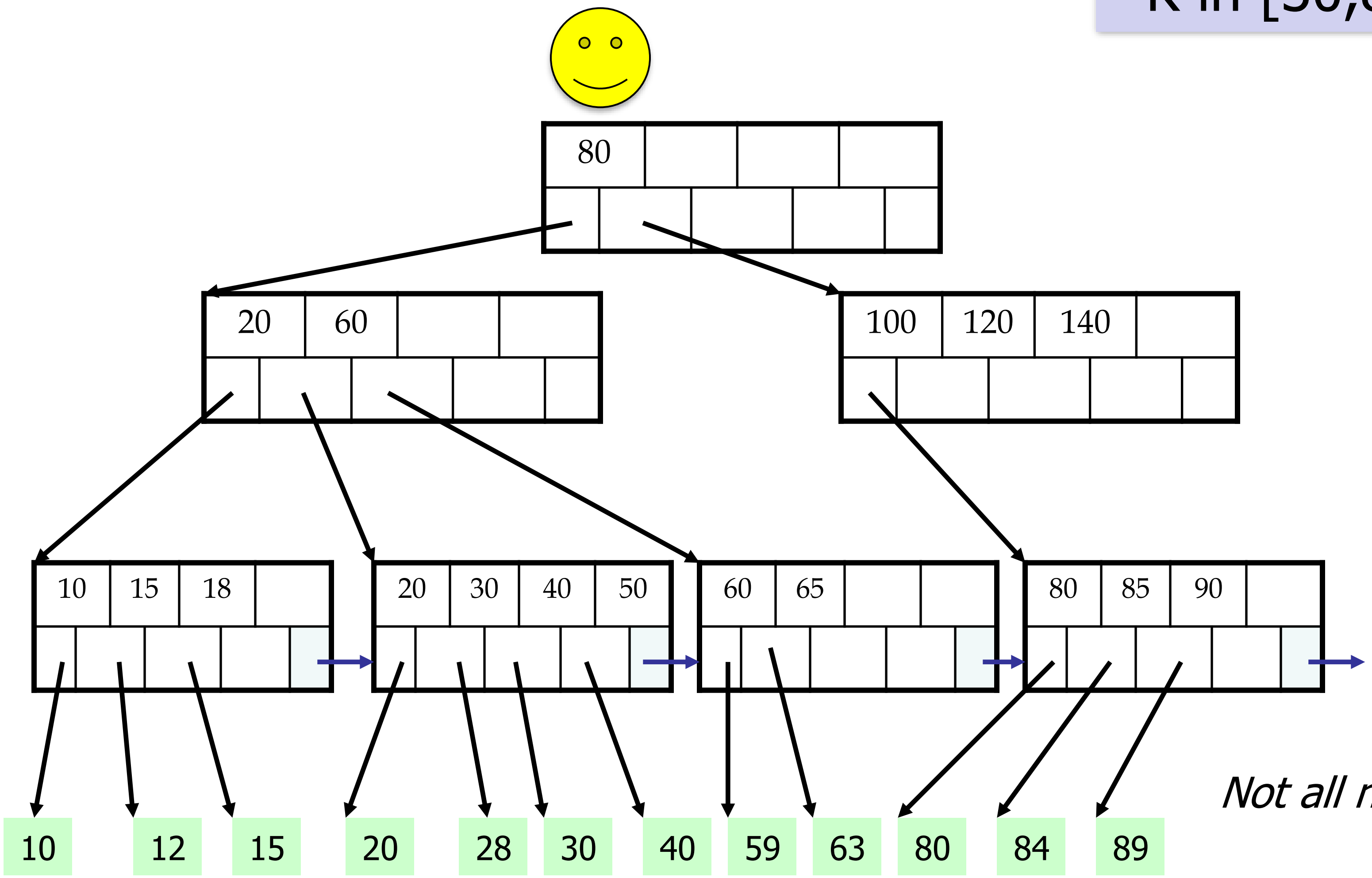
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

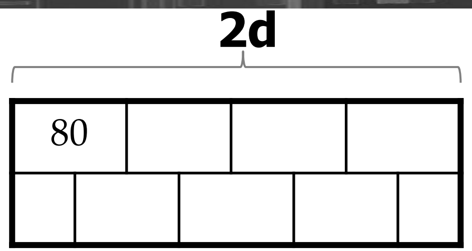
To the data!



Not all nodes pictured

B+ Tree Design

<u>Degree</u> d



Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys***
The root node can have between ***1*** *and 2d keys*

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

E.g.: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

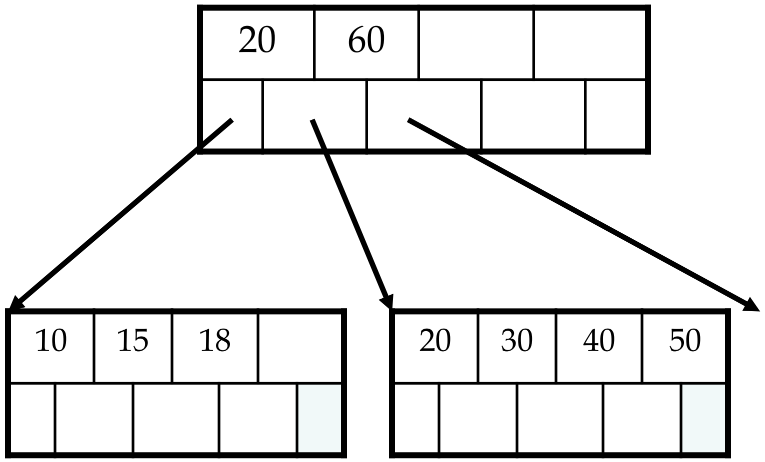
B+ Tree: High Fanout = Smaller & Lower IO

<u>Fanout</u> f
the number of pointers to child nodes coming out of a node

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)

- This means that the **depth of the tree is small**
  - getting to any element requires very few IO operations!
  - Most or all of the B+ Tree in main memory!

Fanout *depends on the data* (assume it's constant for simplicity in calculations)



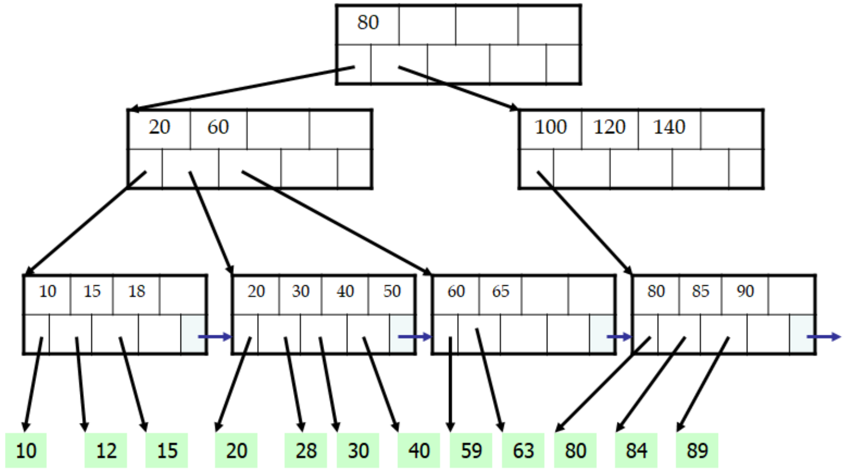
B+ Trees in Practice

<u>Fill-factor</u> F: the percentage of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 134
- Typical capacities:
  - Height 1: 134 = 134 records

What is the relationship between fill factor F and fanout f?

$(2d+1)*F = f$



Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ f^h = ***N / F***

→ We need a B+ Tree of height $h = \left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have ***B*** available buffer pages, by the same logic:
  - We can store L_B levels of the B+ Tree in memory
  - where L_B ***is the number of levels such that the sum of all the levels' nodes fit in the buffer:***
    - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers

- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as "Cost(OUT)".

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Cost(OUT) has one subtle but important twist… let's watch again

B+ Tree Range Search Animation

K in [30,85]?

How many IOs did our friend do?

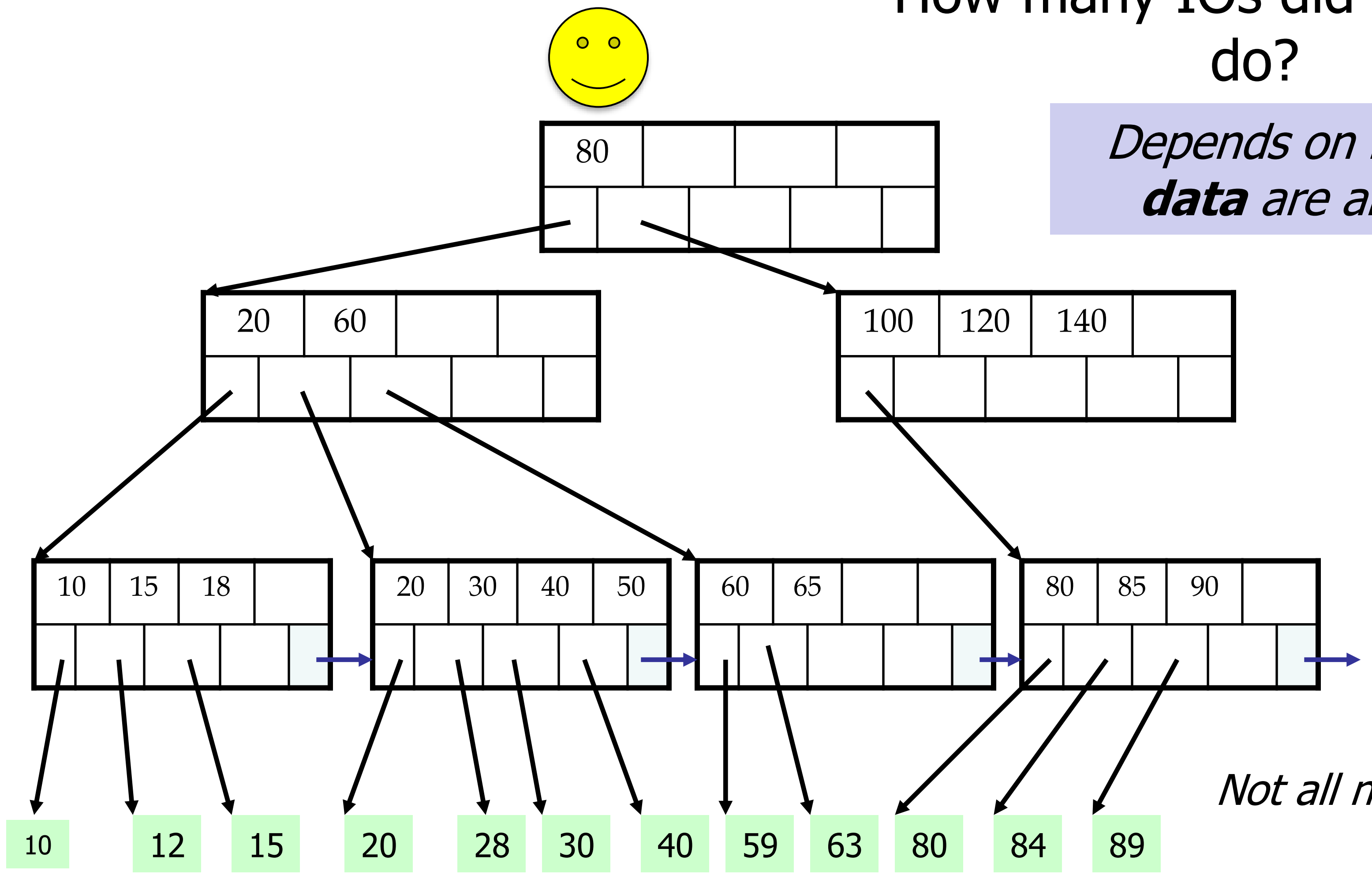
*Depends on **how the data** are arranged*

30 < 80

30 in [20,60)

30 in [30,40)

To the data!

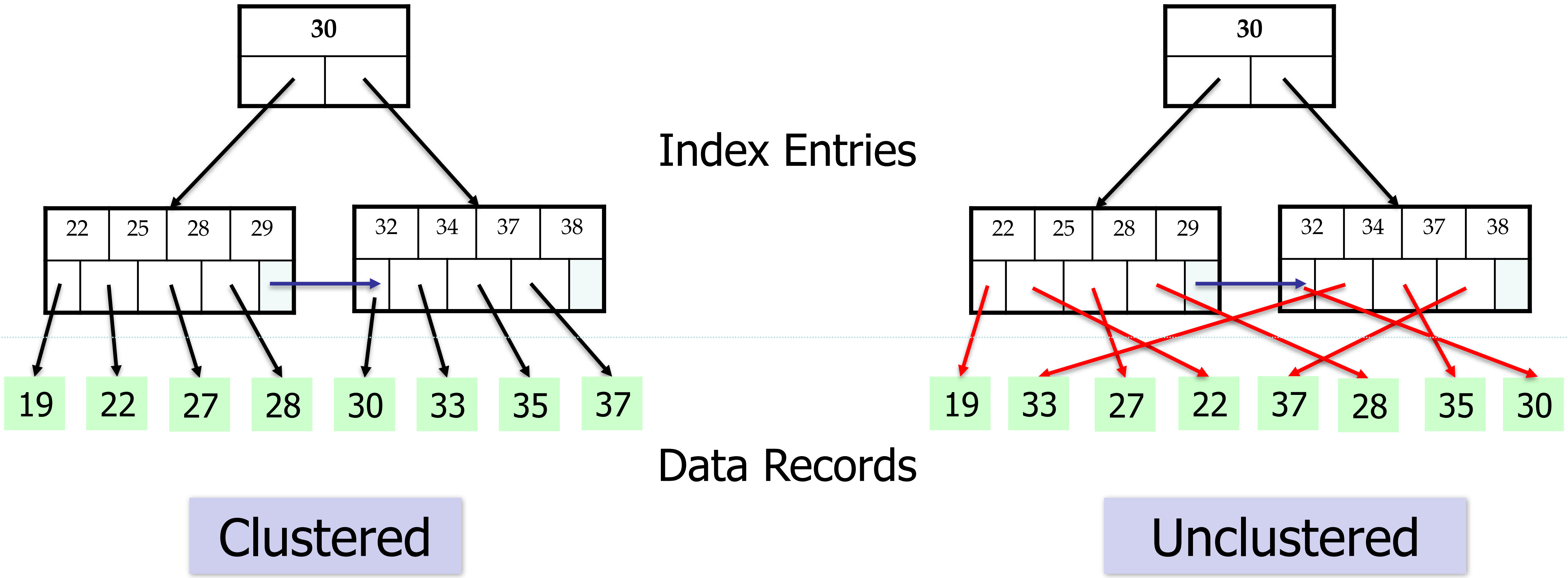


Not all nodes pictured

Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**

- For exact search, no difference between clustered / unclustered

- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

Fast Insertions & Self-Balancing

- We won't go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

Summary

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which supports very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too