UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 3380 Spring 2024

Database Systems

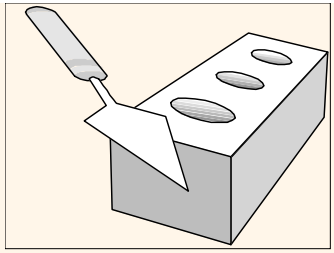
M & W 4:00 to 5:30 PM

Prof. **Victoria Hilford**

PLEASE TURN your webcam ON (must have)

NO CHATTING during LECTURE

VH, unhide Section 15: SET 3 STORAGE III – B PLUS TREE INDEX



COSC 3380

4 to 5:30

PLEASE

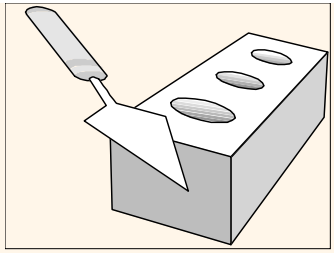
LOG IN

CANVAS

Please close all other windows.

03.25.2024 (18 – Mo)	**ZyBook SET 3- 3**	**Set 3** LECTURE 14 STORAGE III B TREE INDEX
03.27.2024 (19 - We)	**ZyBook SET 3- 4**	**Set 3** LECTURE 15 STORAGE IV HASH INDEX
04.01.2024 (20 - Mo)		**EXAM 3 Practice** **(PART of 20 points)**
04.03.2024 (21 – We)	**TA Download** **ZyBook SET 3 Sections** **(4 PM)** **(PART of 30 points)**	**EXAM 3 Review** **(PART of 20 points)**
04.08.2024 (22 - Mo)		**EXAM 3** **(PART of 50 points)**

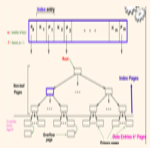
COSC 3380



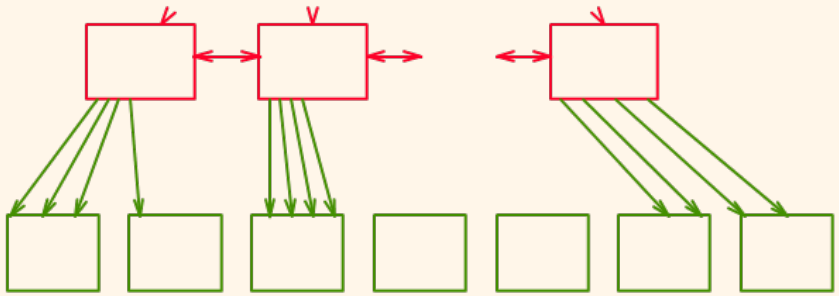
Class 18

03.25.2024	ZyBook SET 3- 3	Set 3
(18 – Mo)		LECTURE 14 STORAGE III B TREE INDEX

$$\text{Cost} = \lceil \log_F N \rceil + 1 = \text{Tree Level}$$



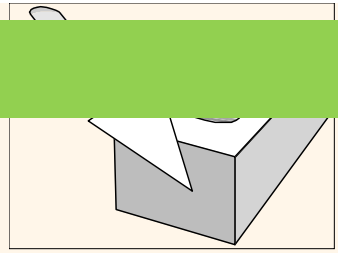
F = # pointers/ Index Page, N = # Leaf (Data Entry k*) blocks



N blocks of Data Entries k*

B blocks of Data Records

From 4:00 to 4:07 PM – 5 minutes.



03.25.2024 (18 – Mo)	ZyBook SET 3- 3	Set 3 LECTURE 14 STORAGE III B TREE INDEX

CLASS PARTICIPATION 20 points — 20% of Total

B PLUS TREE

Class 18 BEGIN PARTICIPATION — VH, publish

Not available until Mar 25 at 4:00pm | Due Mar 25 at 4:07pm | 100 pts

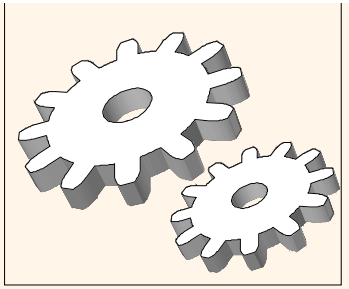
This is a synchronous online class.

Attendance is required.

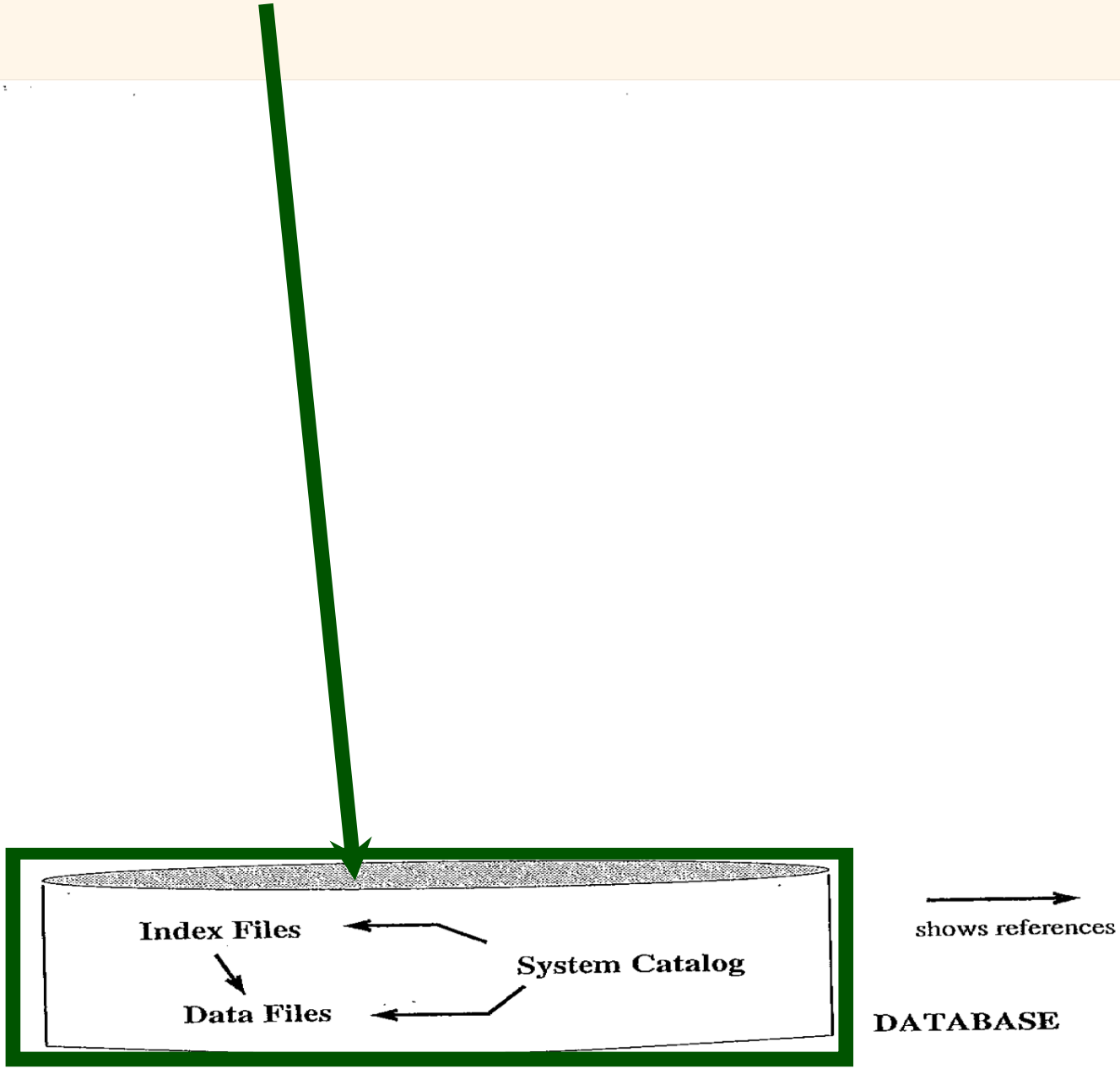
Recording or distribution of class materials is prohibited.

1. At the beginning of selected classes there is an **assessment in the first 10 minutes**. (**beige** BOX in the Detailed Syllabus)
2. At the end of selected classes there is an **assessment in the last 10 minutes**. (**blue** BOX in the Detailed Syllabus)
3. ZyBook sections will be downloaded and used for 30% of Total Score on the dates specified in the Detailed Syllabus.
4. EXAMS are in CANVAS. **No late EXAMS**.
5. I have to be **present in TEAMS in order to take any graded assignment assigned during that class.**

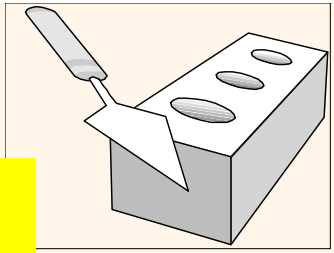
A *DBMS*



Tables
Indexes



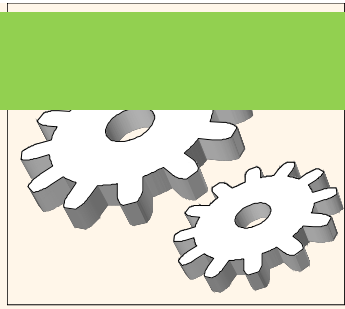
COSC 3380



ZyBook SET 3

10.24.2022 (18 – Mo)	**ZyBook SET 3- 3**	**Set 3** LECTURE 14 STORAGE III B TREE INDEX
10.26.2022 (19 - We)	**ZyBook SET 3- 4**	**Set 3** LECTURE 15 STORAGE IV HASH INDEX
10.31.2022 (20 - Mo)		**EXAM 3 Practice** **(PART of 10 points)**
11.02.2022 (21 – We)	**Download** **ZyBook SET 3 Sections** **(4 PM)** **(PART of 30 points)**	**EXAM 3 Review** **(PART of 10 points)**
11.07.2022 (22 - Mo)		**EXAM 3 – 15 points**

From 4:07 to 5:00 PM – 53 minutes.



COSC 3380

Lecture 14

Tree Structured Indexes

R – number of data records per page/block

B – number of data records blocks

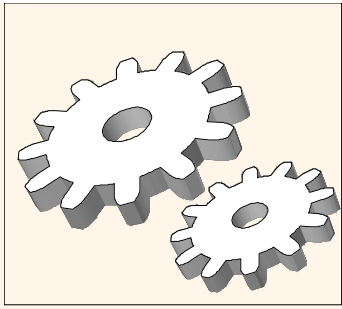
N – number of data entries blocks

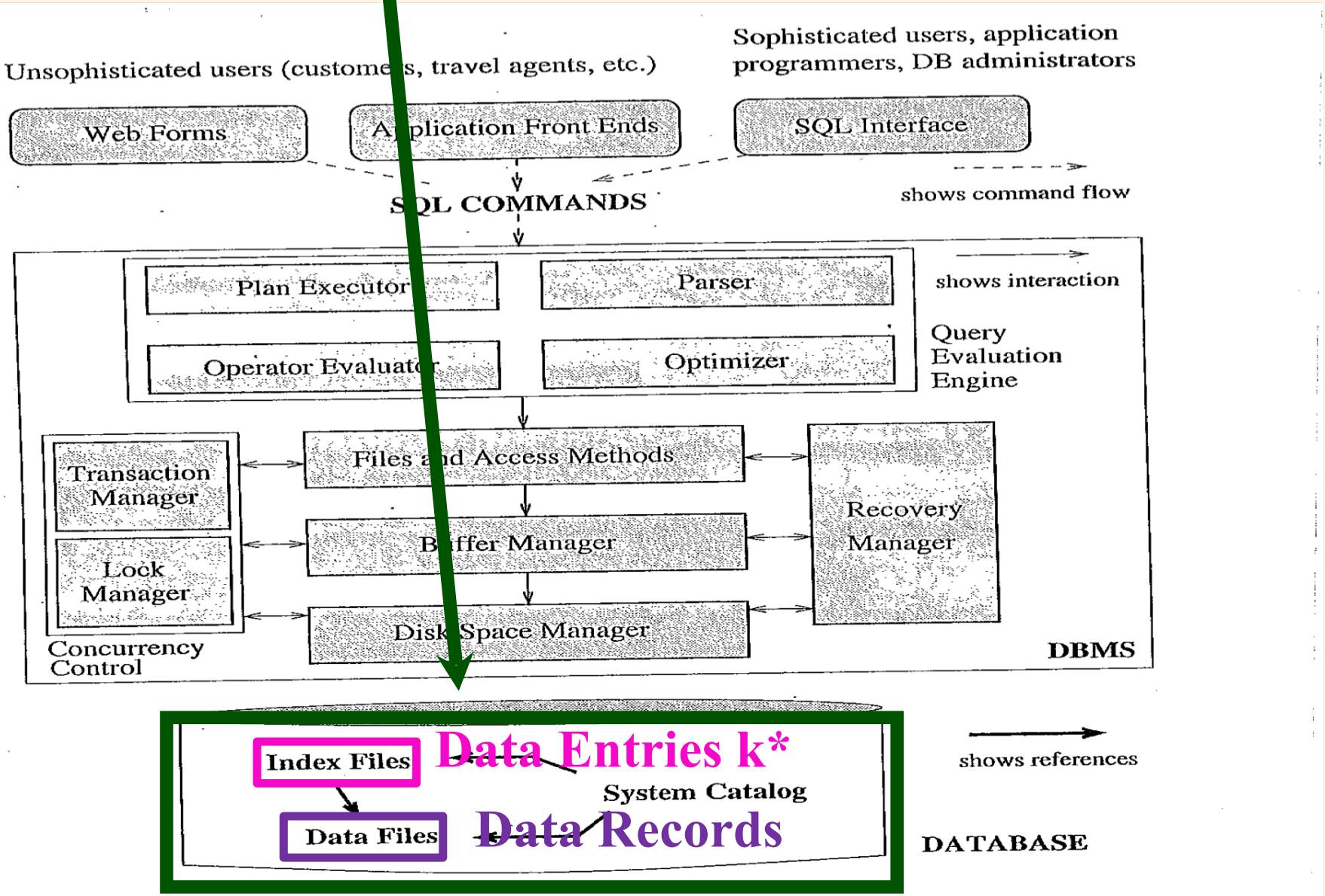
F – fan out - number of data entries per page/block

$$\text{Cost} = \lceil \log_F N \rceil + 1 = \text{Tree Level}$$

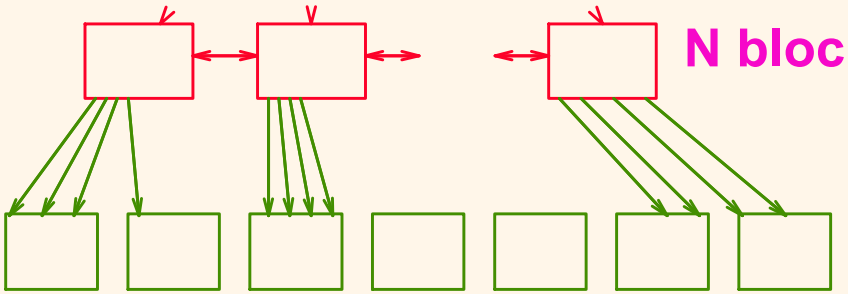
F = # pointers/ **Index Page**, N = # **Leaf** (**Data Entry k***) **blocks**

A DBMS




Unsophisticated users (customers, travel agents, etc.)
Sophisticated users, application programmers, DB administrators
Web Forms
Application Front Ends
SQL Interface
SQL COMMANDS
shows command flow
Plan Executor
Parser
Operator Evaluator
Optimizer
Query Evaluation Engine
shows interaction
Transaction Manager
Lock Manager
Concurrency Control
Files and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
DBMS
Index Files
Data Files
Data Entries k*
System Catalog
Data Records
DATABASE
shows references

Index

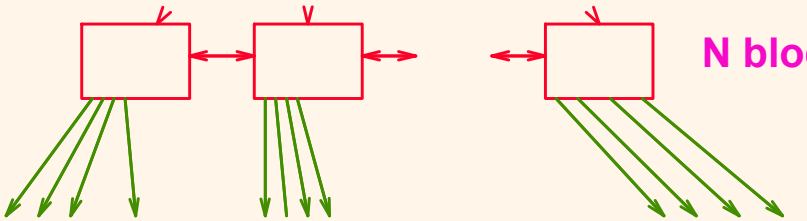


N blocks of Data Entries k*

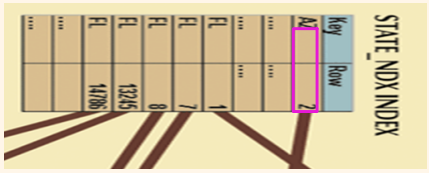
B blocks of Data Records

Data Entries k*

- smaller than **Data Records**
- reside in the leaf node/**Page**



N blocks of Data Entries k*



- **Index rows** contains
- k* = **<k, rid>** of the
- **Data Record k**

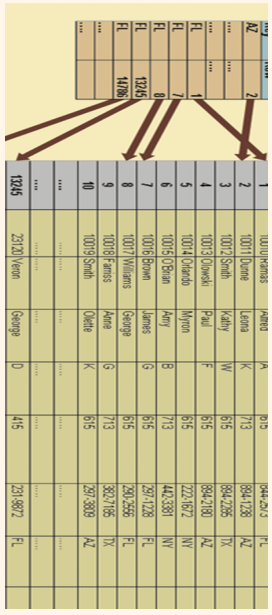
  rid = <pid, slot #>

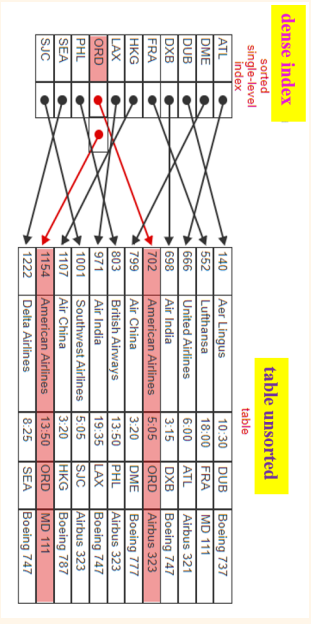
7, <1,1>

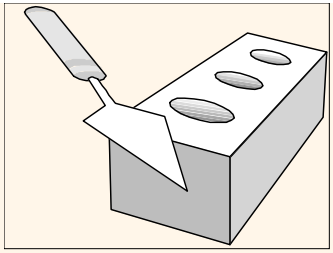
9, <2,1 >

Data Records k

- **Data Record (Record, Tuple)** with key value **k**







15. SET 3 - 3:STORAGE III B TREE INDEX 0% 0%

15.1 Single-level indexes Hidden 0% 0%

Single-level indexes

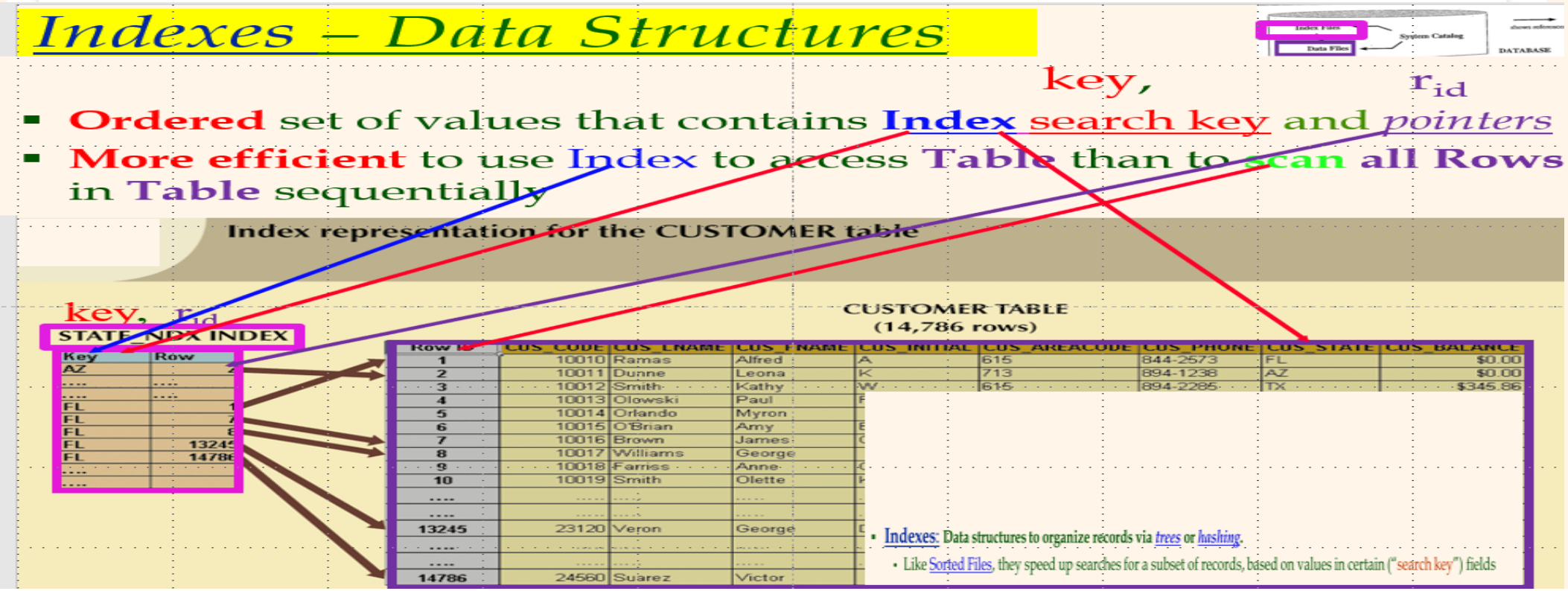
A ***single-level index*** is a file containing column values, along with pointers to rows containing the column value. The pointer identifies the block containing the row. In some indexes, the pointer also identifies the exact location of the row within the block. Indexes are created by database designers with the CREATE INDEX command, described elsewhere in this material.

Single-level indexes are normally sorted on the column value. A sorted index is not the same as an index on a sorted table. Ex: An index on a heap table is a sorted index on an unsorted table.

If an indexed column is unique, the index has one entry for each column value. If an indexed column is not unique, the index may have multiple entries for some column values, or one entry for each column value, followed by multiple pointers.

An index is usually defined on a single column, but an index can be defined on multiple columns. In a ***multi-column index***, each index entry is a composite of values from all indexed columns. In all other respects, multi-column indexes behave exactly like indexes on a single column.

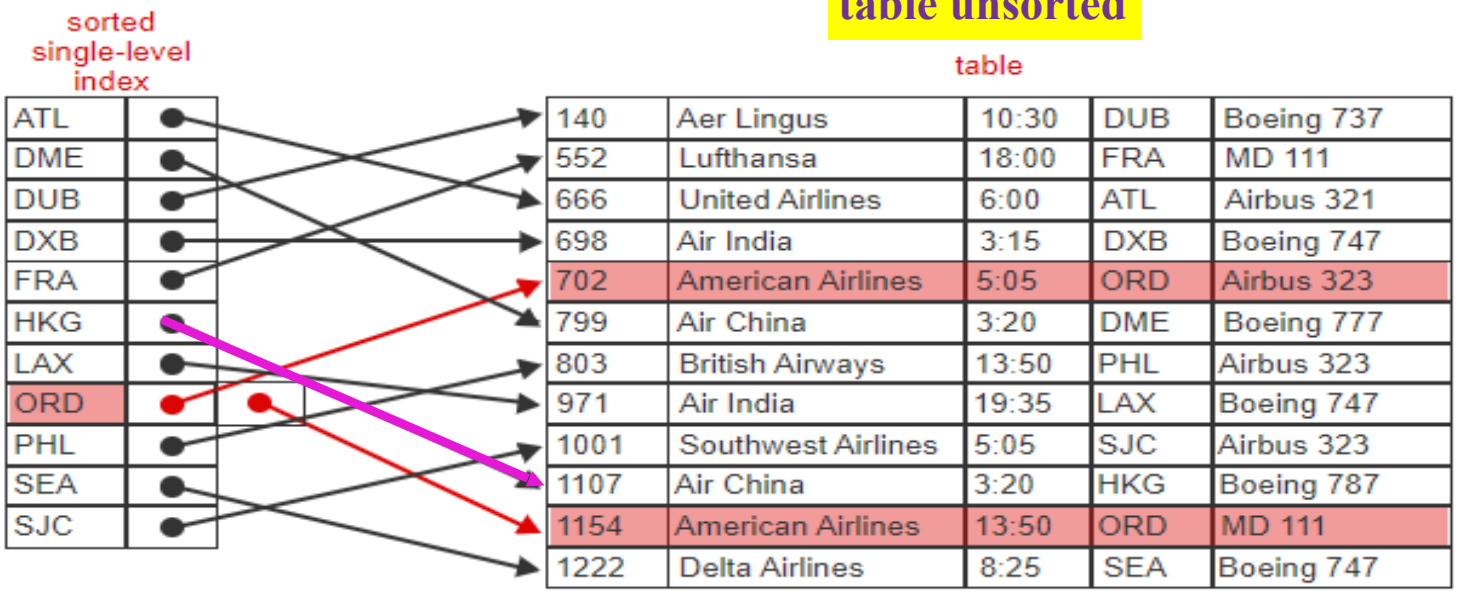
PARTICIPATION ACTIVITY 15.1.1: Single-level index.



PARTICIPATION ACTIVITY 15.1.1: Single-level index.

dense index

table unsorted



PARTICIPATION ACTIVITY 15.1.2: Single-level indexes.

Refer to the index and table in the animation above.

1) Which index entry refers to Air China flight 1107?

- ○ The first entry
- ◉ The sixth entry
- ○ The eleventh entry

Correct

The sixth entry contains the value HKG, with a pointer to the row for Air China flight 1107.

?????

1

sparse clustered index

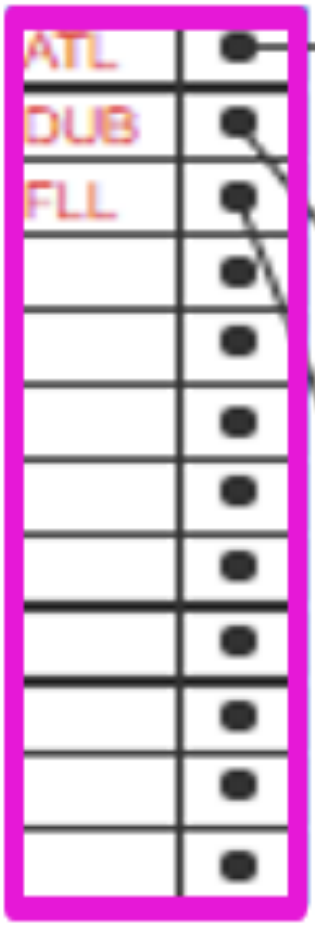
physical design

table sorted

blocks

clustered index

sorted table



666	United Airlines	6:00	ATL	Airbus 321
140	Aer Lingus	10:30	ATL	Boeing 737
799	Air China	3:20	DFW	Boeing 777
698	Air India	3:15	DME	Boeing 747

552	Lufthansa	18:00	DUB	MD 111
1107	Air China	3:20	HKG	Boeing 787
971	Air India	19:35	DXB	Boeing 747
702	American Airlines	5:05	EUG	Airbus 323

1154	American Airlines	13:50	FLL	MD 111
803	British Airways	13:50	FRA	Airbus 323
1222	Delta Air Lines	8:25	FSM	Boeing 747
1001	Southwest Airlines	5:05	FSM	Airbus 323

N blocks of data entries

B blocks of data records

Query processing

To execute a SELECT query, the database can perform a table scan or an index scan:

- A ***table scan*** is a database operation that reads table blocks directly, without accessing an index.
- An ***index scan*** is a database operation that reads index blocks sequentially, in order to locate the needed table blocks.

Hit ratio, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query needs only a few table blocks, so a table scan would be inefficient. Instead, the database:

1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

If the WHERE clause does not contain an indexed column, the database must perform a table scan.

Since a column value and pointer occupy less space than an entire row, an index requires fewer blocks than a table. Consequently, index scans are much faster than table scans. In some cases, indexes are small enough to reside in main memory, and index scan time is insignificant. When hit ratio is low, additional time to read the table blocks containing selected rows is insignificant.

PARTICIPATION ACTIVITY | 15.1.3: Query processing with single-level index.

B blocks of data records

table

140	Aer Lingus	10:30	DUB	Boeing 737

B blocks

971	Air India	19:35	LAX	Boeing 747
1001	Southwest Airlines	5:05	SJC	Airbus 323
1107	Air China	3:20	HKG	Boeing 787
1154	American Airlines	13:50	ORD	MD 111
1222	Delta Airlines	8:25	SEA	Boeing 747

Query processing

To execute a SELECT query, the database can perform a table scan or an index scan:

- A ***table scan*** is a database operation that reads table blocks directly, without accessing an index.
- An ***index scan*** is a database operation that reads index blocks sequentially, in order to locate the needed table blocks.

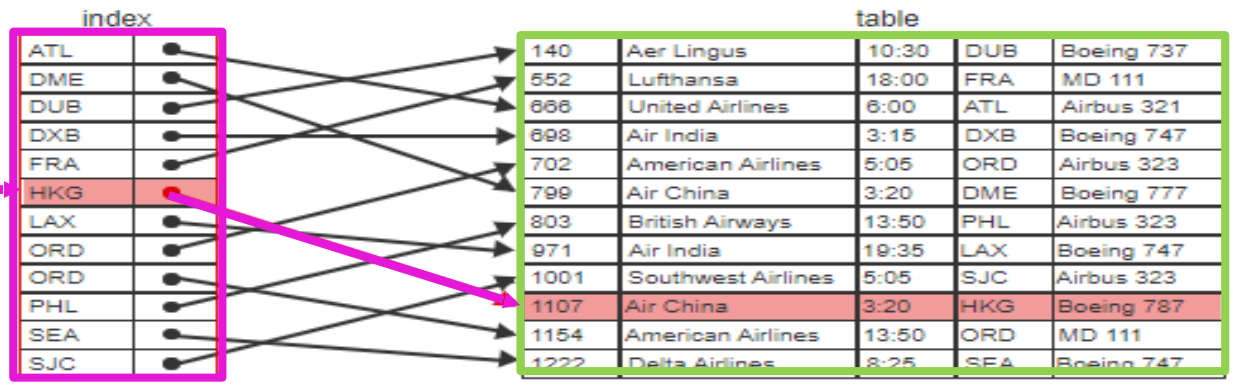
Hit ratio, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query needs only a few table blocks, so a table scan would be inefficient. Instead, the database:

1. Looks for an indexed column in the WHERE clause.
2. Scans the index.
3. Finds values that match the WHERE clause.
4. Reads the corresponding table blocks.

If the WHERE clause does not contain an indexed column, the database must perform a table scan.

Since a column value and pointer occupy less space than an entire row, an index requires fewer blocks than a table. Consequently, index scans are much faster than table scans. In some cases, indexes are small enough to reside in main memory, and index scan time is insignificant. When hit ratio is low, additional time to read the table blocks containing selected rows is insignificant.

PARTICIPATION ACTIVITY | 15.1.3: Query processing with single-level index.



~~*INDEX*~~

~~*Binary Search*~~

N blocks of data entries

1

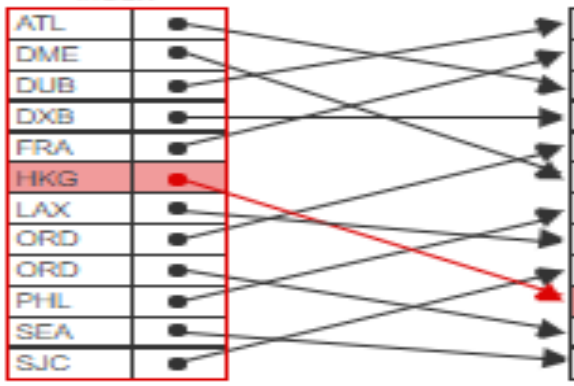
```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'HKG';
```

PARTICIPATION ACTIVITY 15.1.3: Query processing with single-level index.

N = 200

~~INDEX~~

~~Binary Search~~



table

140	Aer Lingus	10:30	DUB	Boeing 737
552	Lufthansa	18:00	FRA	MD 111
666	United Airlines	6:00	ATL	Airbus 321
696	Air India	3:15	DXB	Boeing 747
702	American Airlines	5:05	ORD	Airbus 323
799	Air China	3:20	DME	Boeing 777
803	British Airways	13:50	PHL	Airbus 323
971	Air India	19:35	LAX	Boeing 747
1001	Southwest Airlines	5:05	SJC	Airbus 323
1107	Air China	3:20	HKG	Boeing 787
1154	American Airlines	13:50	ORD	MD 111
1222	Delta Airlines	8:25	SEA	Boeing 747

B = 2,000

```
SELECT FlightNumber, AirlineName
FROM Flight;
```

1

FlightNumber	AirlineName
140	Aer Lingus
552	Lufthansa
666	United Airlines
696	Air India
702	American Airlines
799	Air China
803	British Airways
971	Air India
1001	Southwest Airlines
1107	Air China
1154	American Airlines
1222	Delta Airlines

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE FlightNumber = 803;
```

FlightNumber	AirlineName
803	British Airways

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'HKG';
```

FlightNumber	AirlineName
1107	Air China

PARTICIPATION ACTIVITY 15.1.4: Query processing with single-level indexes.

Refer to the following scenario:

- A table occupies 2,000 blocks. B = 2,000
- FlightNumber is the primary key.
- An index on FlightNumber occupies 200 blocks. N = 200

1) What is the maximum number of blocks necessary to process the SELECT?

- ○ 2
- ◉ 201
- ○ 2,000

200 + 1

Correct

The database reads at most 200 index blocks, plus one table block containing the row for flight 3988.

?????

2

Primary, clustering, and secondary indexes

Indexes on a sorted table may be primary, clustering, or secondary:

- A ***primary index*** is an index on a unique sort column.
- A ***clustering index*** is an index on a non-unique sort column.
- A ***secondary index*** is an index that is not on the sort column.

A sorted table can have only one sort column, and therefore only one primary or clustering index. Tables can have many secondary indexes. All indexes of a heap or hash table are secondary, since heap and hash tables have no sort column.

Indexes may also be dense or sparse:

- A ***dense index*** contains an entry for every table row.

When a table is sorted on an index column, the index may be sparse, as illustrated in the animation below. Primary and clustering indexes are on sort columns and usually sparse. Secondary indexes are on non-sort columns and therefore are always dense.

Sparse indexes are much faster than dense indexes since sparse indexes have fewer entries and occupy fewer blocks. Consider the following scenario:

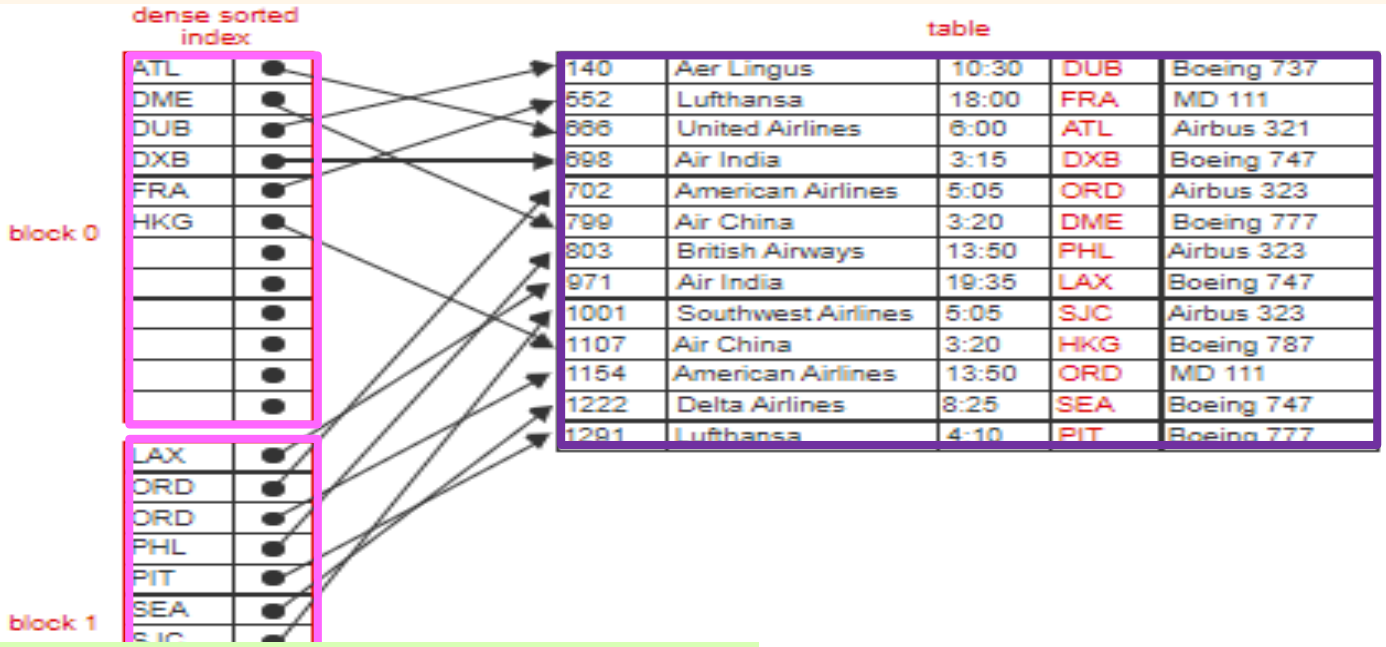
- A table has 10 million rows.
- Each row is 100 bytes.
- Table and index blocks are 4 kilobytes.
- Each index entry is 10 bytes.

B = 250,000 blocks N = 625 blocks

The table occupies 250,000 blocks (10 million rows × 100 bytes/row / 4 kilobytes/block). A sparse index requires 250,000 entries (one entry per table block) and occupies 625 blocks (250,000 entries × 10 bytes/entry / 4 kilobytes/block). The sparse index can easily be retained in main memory.

Primary and clustering i…rse and sparse indexes are fast. As a result, database designers usually create a primary or clustering index on large…

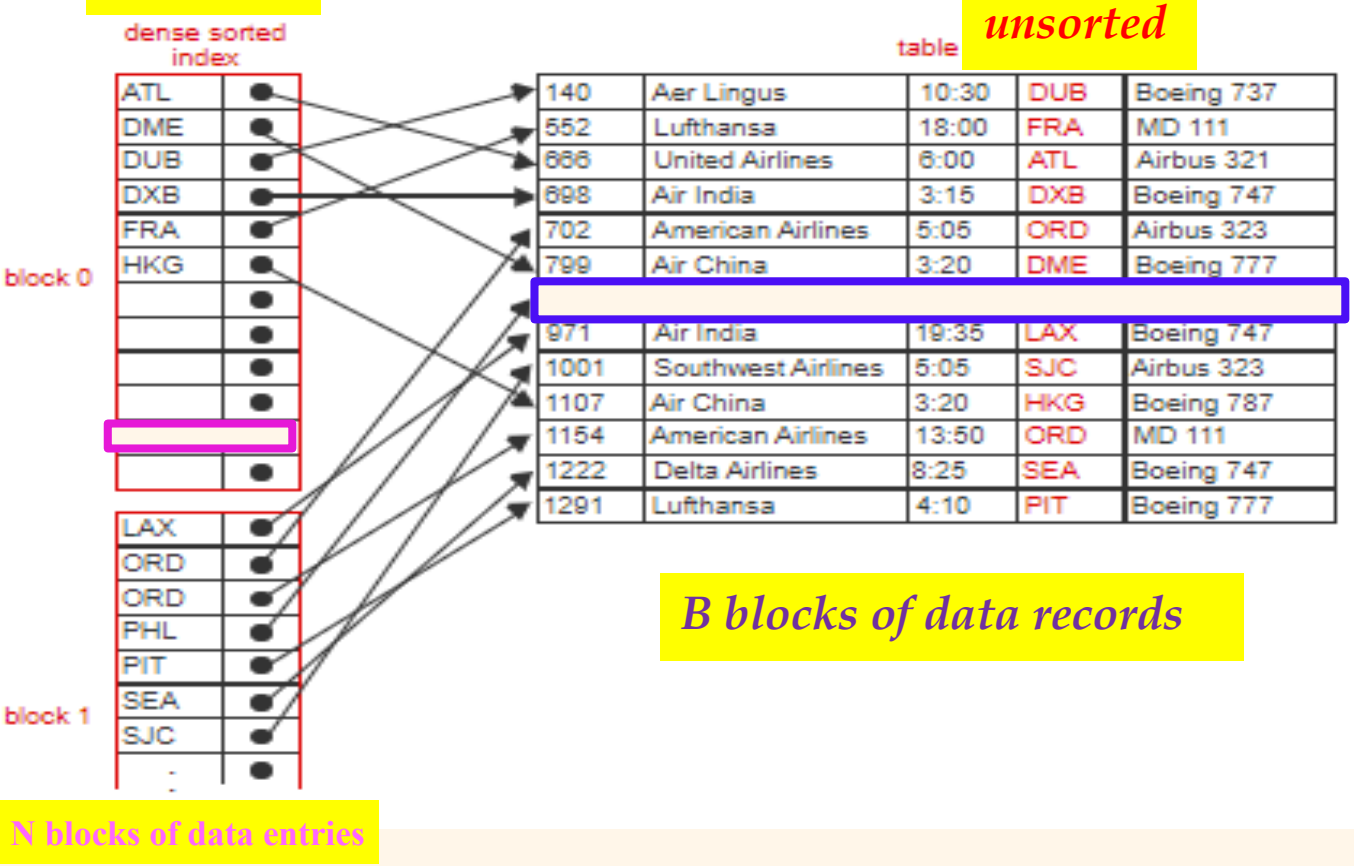
Dense index



 # index entries* = *# data records

PARTICIPATION ACTIVITY | 15.1.9: Insert with dense sorted index.

sorted



PARTICIPATION ACTIVITY | 15.1.10: Inserts, updates, and deletes.

1) Inserts to a sorted table are always faster when the table has no indexes.

- ○ True
- ◉ False

Correct

Inserting a row may generate a new index entry, which slows down the insert. However, if the index is on the sort column, the index helps locate the table block for the insert, which speeds up the query.

?????

Binary search

When hit ratio is low, index scans are always faster than table scans. Consider the following scenario:

- A table has 10 million rows.
- Each row is 100 bytes.
- Each block is 4 kilobytes.
- Each index entry is 10 bytes, including a 6-byte value and a 4-byte pointer.
- Magnetic disk transfer rate is 0.1 gigabytes per second.

If hit ratio is low, an index scan is roughly 10 times faster than a table scan:

- **Table scan:** The table contains 1 billion bytes (10 million rows × 100 bytes/row). The table scan takes around 10 seconds (1 gigabyte / 0.1 gigabytes/sec).
- **Index scan:** The index contains 100 million bytes (10 million rows × 1 entry/row × 10 bytes/entry). The index scan takes around 1 second (0.1 gigabyte / 0.1 gigabytes/sec). The index scan returns pointers to blocks containing rows selected by the query. When hit ratio is low, additional time to read table blocks is insignificant.

Although index scans are faster than table scans, index scans are too slow in many cases. If a single-level index is sorted, each value can be located with a binary search. In a ***binary search***, the database repeatedly splits the index in two until it finds the entry containing the search value:

1. The database first compares the search value to an entry in the middle of the index.
2. If the search value is less than the entry value, the search value is in the first half of the index. If not, the search value is in the second half.
3. The database now compares the search value to the entry in the middle of the selected half, to narrow the search to one quarter of the index.
4. The database continues in this manner until it finds the index block containing the search value.

For an index with N entries, a binary search examines $\log_2 N$ values, rounded up to the nearest integer. In the example above, the index has 10 million entries. The binary search reads at most $\log_2 10{,}000{,}000 = 24$ blocks and requires about 0.001 seconds (24 blocks × 4 kilobytes/block / 0.1 gigabytes/sec).

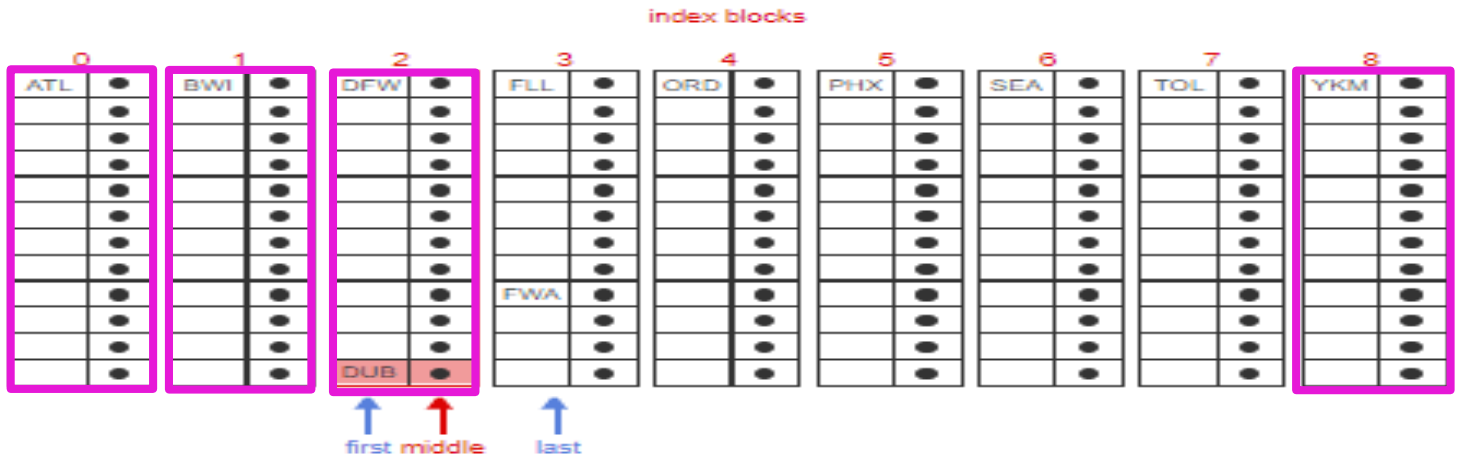
PARTICIPATION ACTIVITY | 15.1.5: Binary search on a sorted index.

INDEX

Binary Search

$log_2 9$

N = 9




```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'DUB';
```

1

Binary search

PARTICIPATION ACTIVITY | 15.1.5: Binary search on a sorted index.

Refer to the following scenario:

- A table has 100 million rows.
- Each row is 400 bytes.
- Each block is 8 kilobytes.
- Each index entry is 20 bytes.
- Magnetic disk transfer rate is 0.1 gigabytes per second.

R = 8,000 / 400 =
20 data records / page

8KB =
8,000 B

1) Assuming no free space, a table scan requires approximately how many seconds?

- ○ 0.4
- ○ 40
- ◉ 400

Correct

100,000,000 rows × 400 bytes/row / 0.1 gigabytes/sec = 400 seconds.

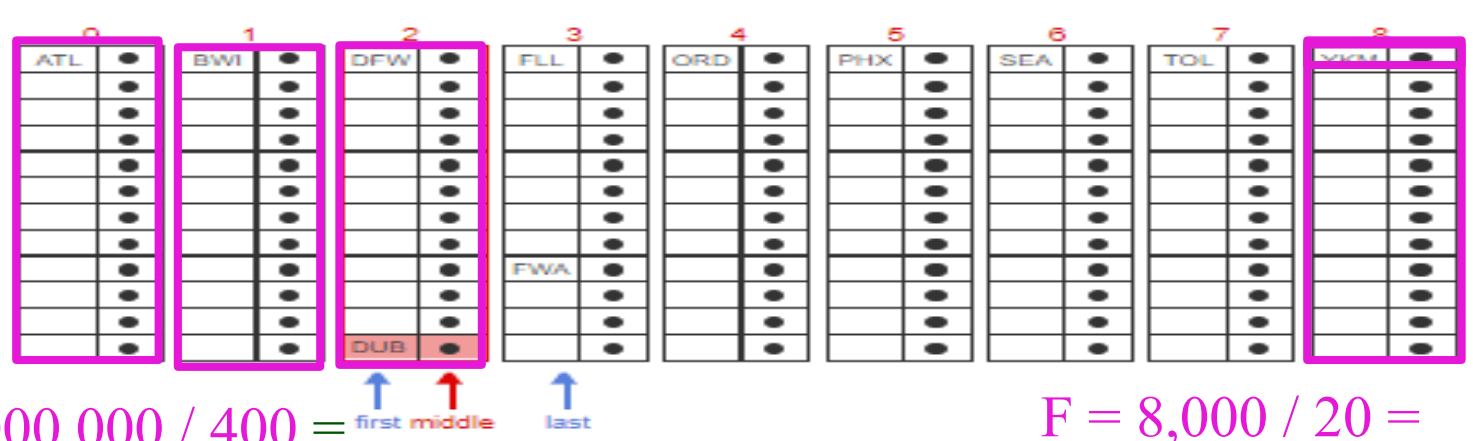
Binary search

PARTICIPATION ACTIVITY | 15.1.5: Binary search on a sorted index.

INDEX

Binary Search

$log_2 N$



20 B

8KB = 8,000 B

N = 100,000,000 / 400 = 250,000 blocks

F = 8,000 / 20 = 400 data entries / block

```
SELECT FlightNumber, AirlineName
FROM Flight
WHERE DepartureAirport = 'DUB';
```

PARTICIPATION ACTIVITY | 15.1.6: Binary search. ?????

Refer to the following scenario:

- A table has 100 million rows.
- Each row is 400 bytes.
- Each block is 8 kilobytes.
- Each index entry is 20 bytes.
- Magnetic disk transfer rate is 0.1 gigabytes per second.

R = 8,000 / 400 = 20 data records / block

8KB = 8,000 B

2) Assuming the index is sorted, a binary search for one row reads approximately how many blocks?

- (●) log_2 250,000
- ( ) log_2 5,000,000
- ( ) log_{10} 250,000

Correct

Each index block contains 8,000 bytes / 20 bytes/entry = 400 entries. The index has 100,000,000 entries / 400 entries/block = 250,000 blocks. A binary search requires log 250,000 (base 2).

B = 100,000,000 / 20 = 50,000,000 blocks

Primary, clustering, and secondary indexes

Indexes on a sorted table may be primary, clustering, or secondary:

- A ***primary index*** is an index on a unique sort column.
- A ***clustering index*** is an index on a non-unique sort column.
- A ***secondary index*** is an index that is not on the sort column.

A sorted table can have only one sort column, and therefore only one primary or clustering index. Tables can have many secondary indexes. All indexes of a heap or hash table are secondary, since heap and hash tables have no sort column.

Indexes may also be dense or sparse:

- A ***dense index*** contains an entry for every table row.
- A ***sparse index*** contains an entry for every table block.

When a table is sorted on an index column, the index may be sparse, as illustrated in the animation below. Primary and clustering indexes are on sort columns and usually sparse. Secondary indexes are on non-sort columns and therefore are always dense.

Sparse indexes are much faster than dense indexes since sparse indexes have fewer entries and occupy fewer blocks. Consider the following scenario:

- A table has 10 million rows.
- Each row is 100 bytes.
- Table and index blocks are 4 kilobytes.
- Each index entry is 10 bytes.

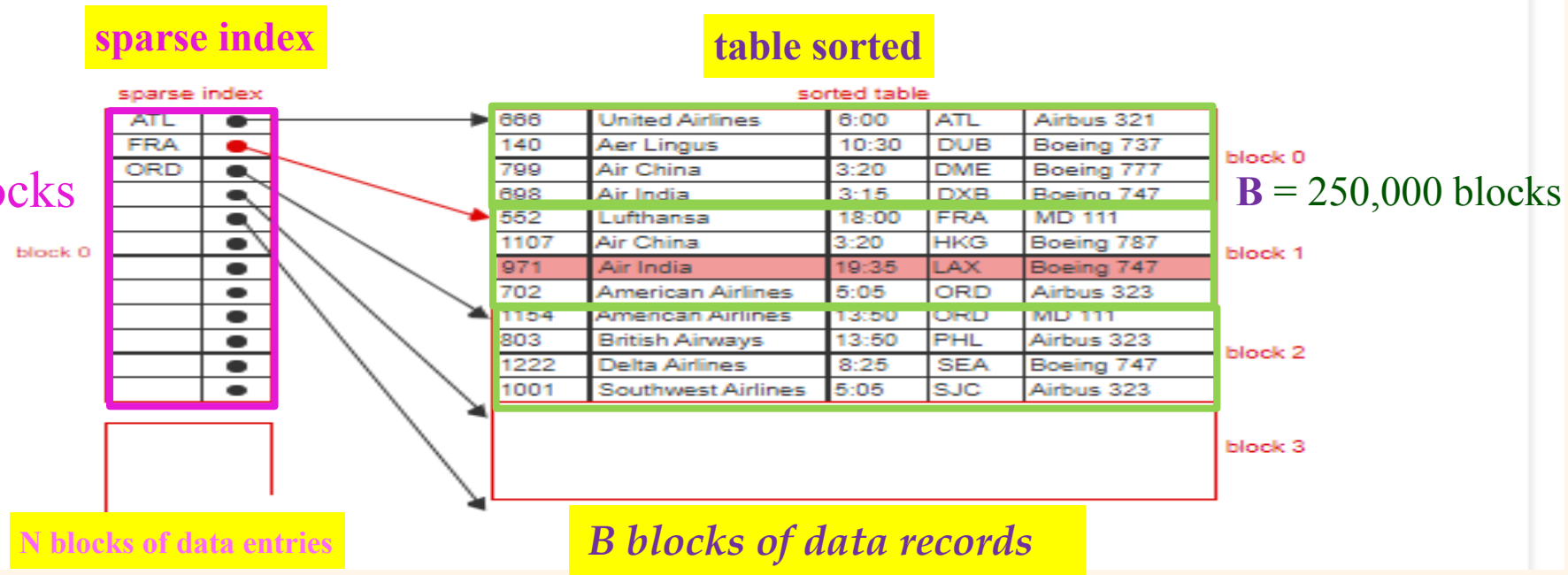
B = 250,000 blocks

The table occupies 250,000 blocks (10 million rows × 100 bytes/row / 4 kilobytes/block). A sparse index requires 250,000 entries (one entry per table block) and occupies 625 blocks (250,000 entries × 10 bytes/entry / 4 kilobytes/block). The sparse index can easily be retained in main memory.

N = 625 blocks

Primary and clustering indexes are usually sparse and sparse indexes are fast. As a result, database designers usually create a primary or clustering index on large tables.

PARTICIPATION ACTIVITY | 15.1.7: Dense and sparse indexes.



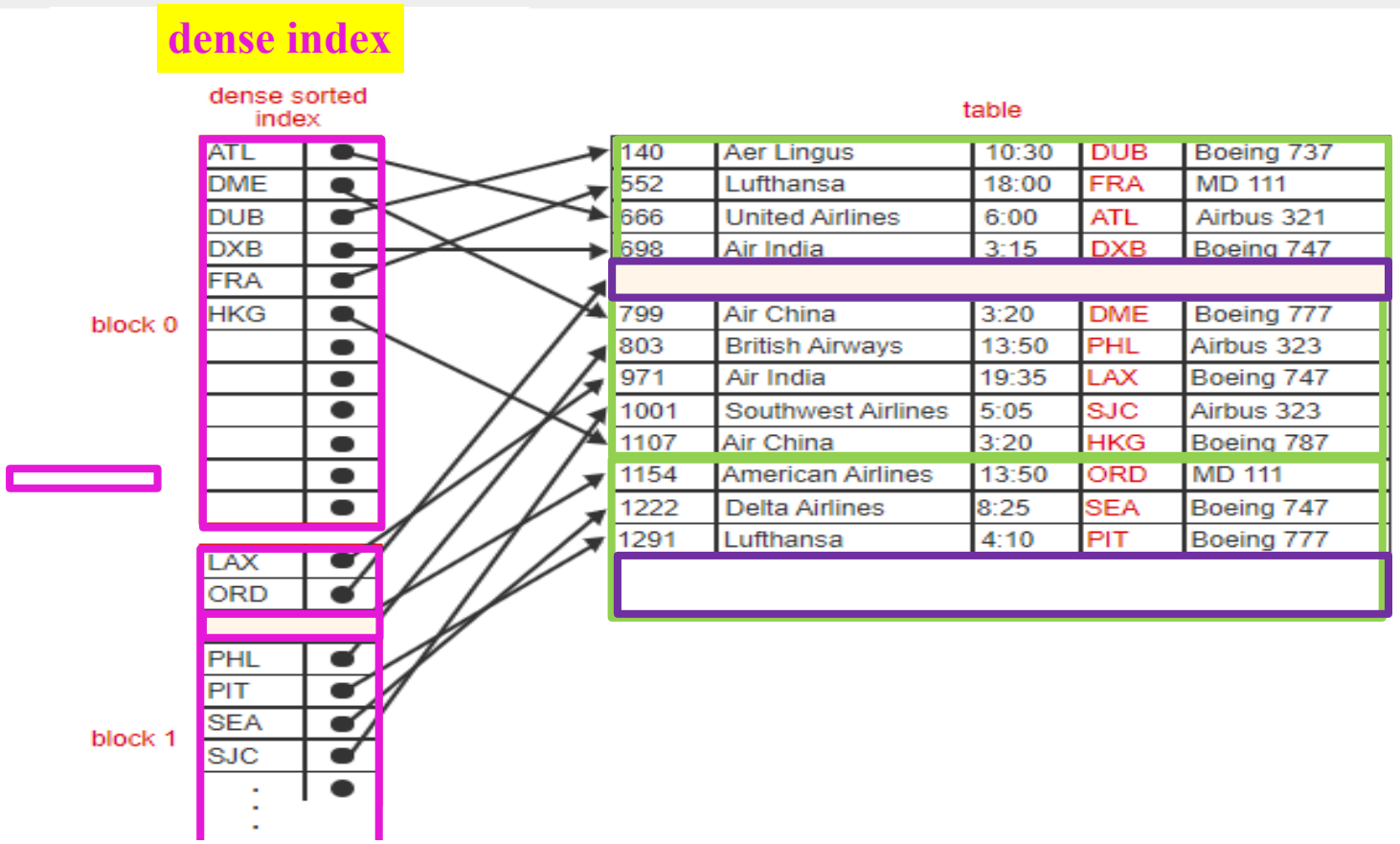
Inserts, updates, and deletes

Inserts, updates, and deletes to tables have an impact on single-level indexes. Consider the behavior of dense indexes:

- **Insert**. When a row is inserted into a table, a new index entry is created. Since single-level indexes are sorted, the new entry must be placed in the correct location. To make space for the new entry, subsequent entries must be moved, which is too slow for large tables. Instead, the database splits an index block and reallocates entries to the new block, creating space for the new entry.
- **Delete**. When a row is deleted, the row's index entry must be deleted. The deleted entry can be either physically removed or marked as 'deleted'. Since single-level indexes are sorted, physically removing an entry requires moving all subsequent entries, which is slow. For this reason, index entries are marked as 'deleted'. Periodically, the database may reorganize the index to remove deleted entries and compress the index.
- **Update**. An update to a column that is not indexed does not affect the index. An update to an indexed column is like a delete followed by an insert. The index entry for the initial value is deleted and an index entry for the updated value is inserted.

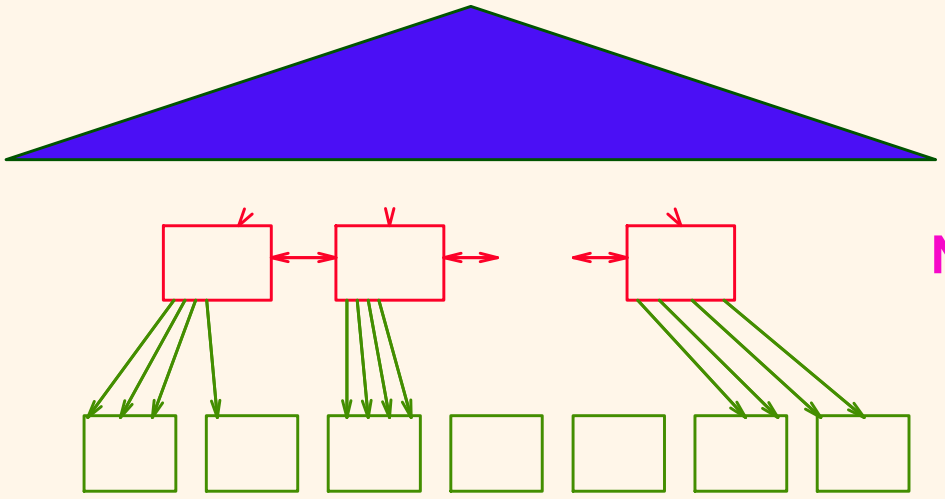
With a sparse index, each entry corresponds to a table block rather than a table row. Index entries are inserted or deleted when blocks split or merge. Since blocks contain many rows, block splits and mergers occur less often than row inserts and deletes. Aside from frequency, however, the behavior of sparse and dense indexes is similar.

PARTICIPATION ACTIVITY | 15.1.9: Insert with dense sorted index.



- *Auxiliary information* used to quickly direct the search to Data Entries k*: **Tree Structured Indexing** techniques support both *range searches* and *equality searches*.

- *ISAM*: static structure;
- *B+ Tree*: dynamic, adjusts gracefully under **inserts** and **deletes**.



N blocks of Data Entries k*

B blocks of Data Records

Indexes

blocks of Search Tree

N blocks of Data Entries k*

B blocks of Data Records

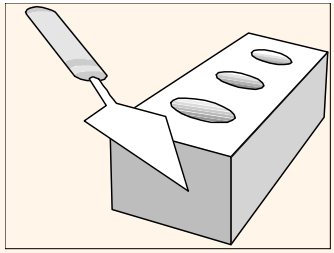
Index rep ation for the CUSTOMER table

CUSTOMER TABLE
(14,786 rows)

STATE_NDX INDEX

Key	Row
AZ	1
....	....
....	....
FL	1
FL	7
FL	8
FL	13245
FL	14786
....	....

Row ID	CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE	CUS_STATE	CUS_BALANCE
1	10010	Ramas	Alfred	A	615	844-2573	FL	$0.00
2	10011	Dunne	Leona	K	713	894-1238	AZ	$0.00
3	10012	Smith	Kathy	W	615	894-2285	TX	$345.86
4	10013	Olowski	Paul	F	615	894-2180	AZ	$536.75
5	10014	Orlando	Myron		615	222-1672	NY	$0.00
6	10015	O'Brian	Amy	B	713	442-3381	NY	$0.00
7	10016	Brown	James	G	615	297-1228	FL	$221.19
8	10017	Williams	George		615	290-2556	FL	$768.93
9	10018	Farriss	Anne	G	713	382-7185	TX	$216.55
10	10019	Smith	Olette	K	615	297-3809	AZ	$0.00
....	.....	.....	.....	.....	.....	.....	.....	.....
....	.....	.....	.....	.....	.....	.....	.....	.....
13245	23120	Veron	George	D	415	231-9872	FL	$675.00
....	.....	.....	.....	.....	.....	.....	.....	.....
....	.....	.....	.....	.....	.....	.....	.....	.....
14786	24560	Suarez	Victor		435	342-9876	FL	$342.00



15. SET 3 - 3:STORAGE III B TREE INDEX 0% 0%

15.2 Multi-level indexes Hidden 0% 0%

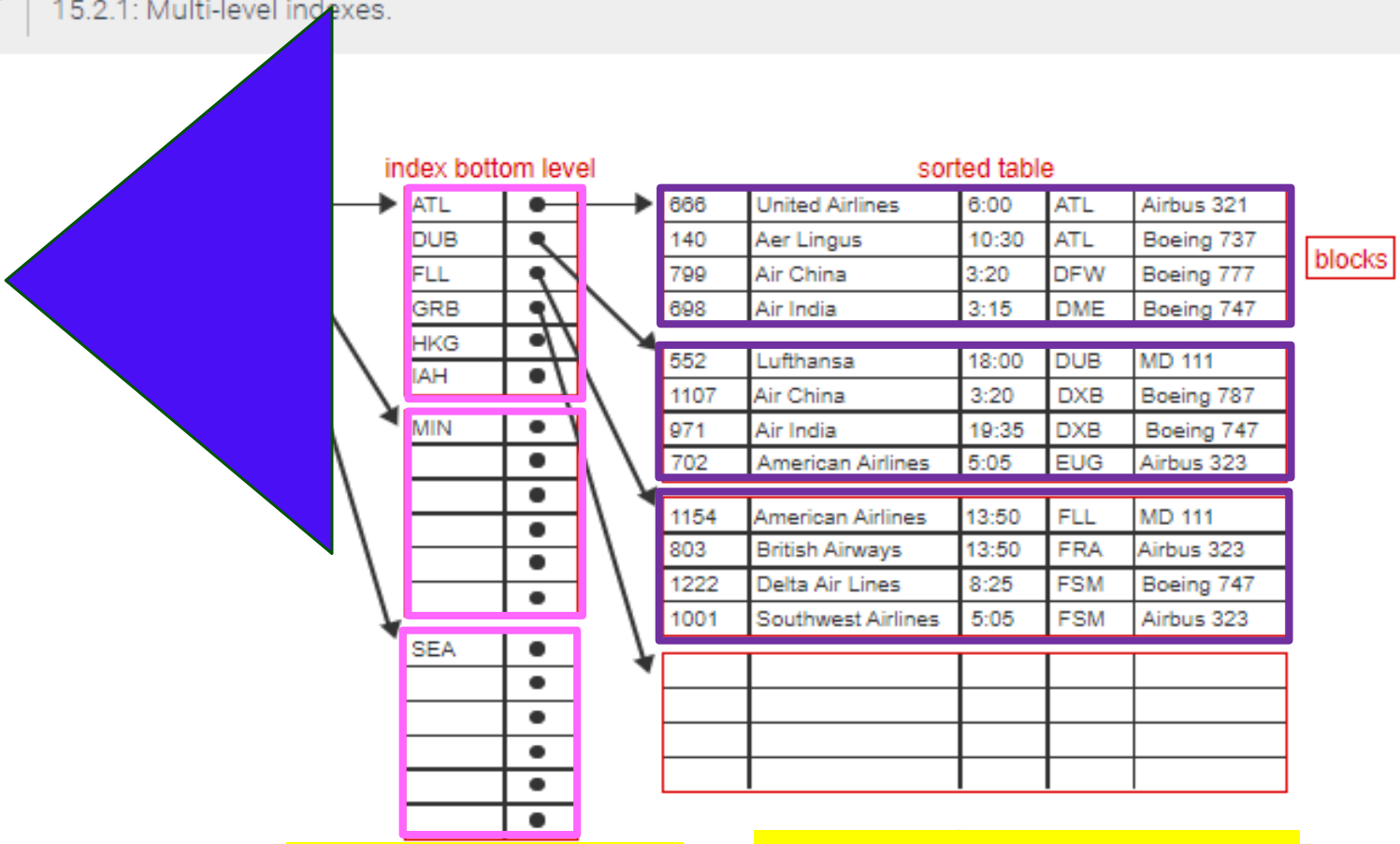
Multi-level indexes

A ***multi-level index*** stores column values and row pointers in a hierarchy. The bottom level of the hierarchy is a sorted single-level index. The bottom level is sparse for primary and clustering indexes, or dense for secondary indexes.

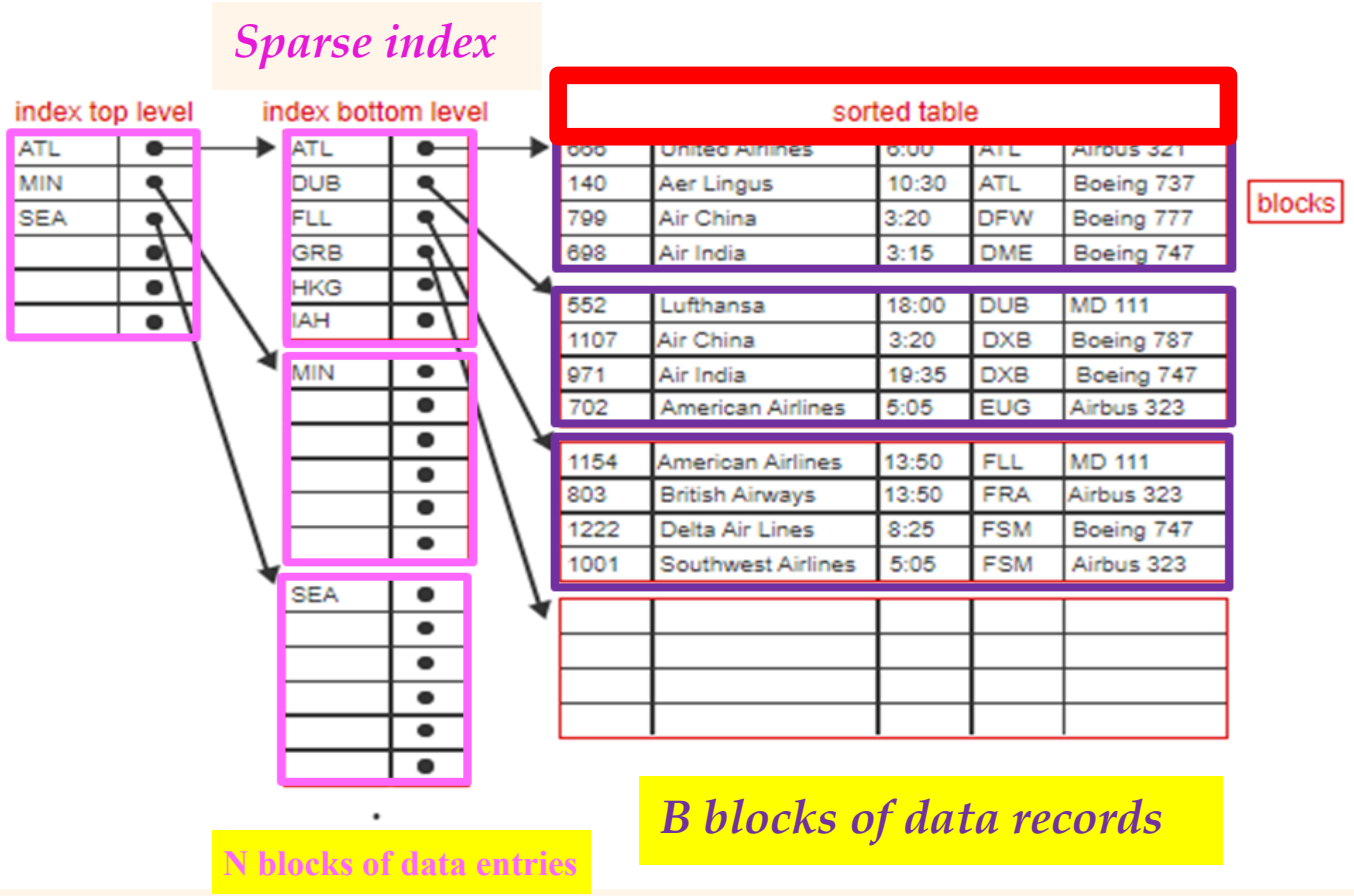
Each level above the bottom is a sparse sorted index to the level below. Since all levels above the bottom are sparse, levels rapidly become smaller. The top level always fits in one block.

To locate a row containing an indexed value, the database first reads the top-level block. The database compares the indexed value to entries in the block and locates the next level block containing the value. Continuing in this manner, the database eventually locates the bottom-level block containing the value. The bottom-level block contains a pointer to the correct table block.

PARTICIPATION ACTIVITY | 15.2.1: Multi-level indexes.



PARTICIPATION ACTIVITY 15.2.1: Multi-level indexes.



PARTICIPATION ACTIVITY 15.2.2: Multi-level indexes.

1) The bottom level of a multi-level index is always sparse.

- ○ True
- ◉ False

Correct

The bottom level of a multi-level index is like a single-level index. The bottom level is usually sparse when the table is sorted on the indexed column. The bottom level is dense when the table is not sorted on the indexed column.

?????

Query processing

- A table has 10 million rows.
- Each row is 100 bytes.
- Each index entry is 10 bytes.
- Table and index blocks are 4 kilobytes.

The table contains 250,000 blocks = 10 million rows × 100 bytes per row / 4,000 bytes per block.

A dense, single-level index contains 25,000 blocks = 10 million entries × 10 bytes per entry / 44,000 bytes per block.

A dense, multi-level index contains 3 levels = log 400 entries per index block 10,000,000 rows, rounded up.

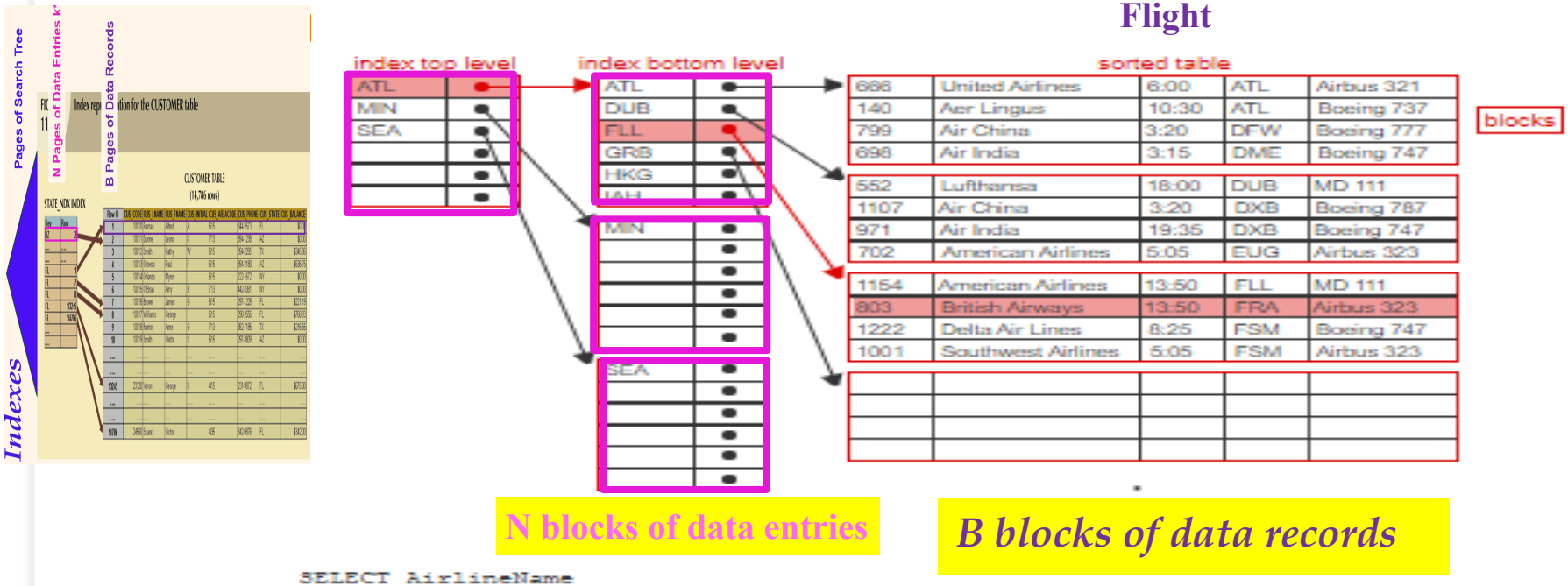
A query searches for rows containing a specific value of an indexed column. Assuming query hit ratio is low:

- A table scan reads at most 250,000 table blocks.
- A single-level index scan reads at most 25,000 index blocks plus a few table blocks.
- A binary search of a sorted, single-level index reads at most 24 index blocks plus a few table blocks.
- A multi-level index search reads 3 index blocks plus a few table blocks.

The multi-level index search reads one index block per level. Usually the top two levels are small and retained in memory. Since the index has three levels, the query reads just one index block from storage media.

Because multi-level indexes are faster than single-level indexes on most queries, databases commonly use multi-level rather than single-level indexes.

PARTICIPATION ACTIVITY | 15.2.4: Query processing with a multi-level index.



666	United Airlines	6:00	ATL	Airbus 321
140	Aer Lingus	10:30	ATL	Boeing 737
799	Air China	3:20	DFW	Boeing 777
698	Air India	3:15	DME	Boeing 747
552	Lufthansa	18:00	DUB	MD 111
1107	Air China	3:20	DXB	Boeing 787
971	Air India	19:35	DXB	Boeing 747
702	American Airlines	5:05	EUG	Airbus 323
1154	American Airlines	13:50	FLL	MD 111
803	British Airways	13:50	FRA	Airbus 323
1222	Delta Air Lines	8:25	FSM	Boeing 747
1001	Southwest Airlines	5:05	FSM	Airbus 323

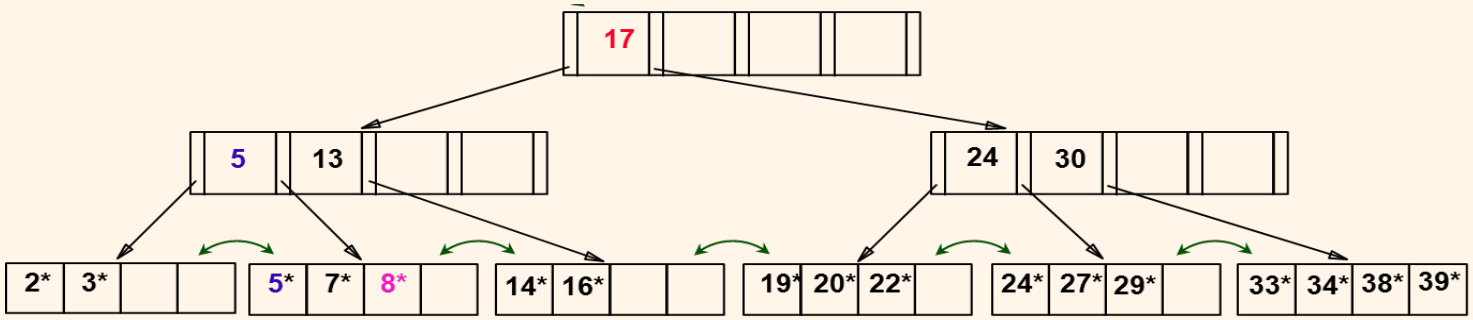
```
SELECT AirlineName
FROM Flight
WHERE DepartureAirport = 'FRA';
```

Multilevel ***Index***

Number of levels

A dense index has more bottom-level entries than a sparse index, and may have more levels. Assuming a table with 10 million rows and 400 index entries per block, a dense index has three levels:

- Level 3 is dense and has 25,000 blocks = 10 million rows / 400 index entries per block
- Level 2 is sparse and has 63 blocks = 25,000 level 3 blocks / 400 index entries per block
- Level 1 is sparse and has one block containing 63 index entries.



Level 1

Level 2

Level 3 N blocks

F = 400

10,000,000 rows

10,000,000 index entries

10,000,000 index entries / 400 = N level 3 = 25,000

25,000 index entries / 400 = level 2 = 63

63 index entries / 400 = level 1 = 1

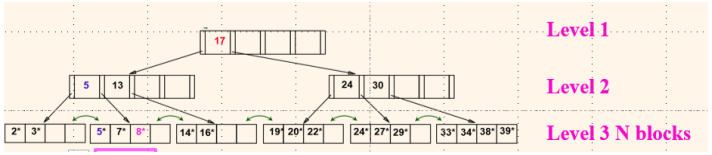
Number of levels

A dense index has more bottom-level entries than a sparse index, and may have more levels. Assuming a table with 10 million rows and 400 index entries per block, a dense index has three levels:

- Level 3 is dense and has 25,000 blocks = 10 million rows / 400 index entries per block
- Level 2 is sparse and has 63 blocks = 25,000 level 3 blocks / 400 index entries per block
- Level 1 is sparse and has one block containing 63 index entries.

The number of index entries per block is called the ***fan-out*** of a multi-level index. The number of levels in a multi-level index can be computed from fan-out, number of rows, and rows per block:

- For a dense index, number of levels = $\log_{\text{fan-out}}$ (number of rows)
- For a sparse index, number of levels = $\log_{\text{fan-out}}$ (number of rows / rows per block)



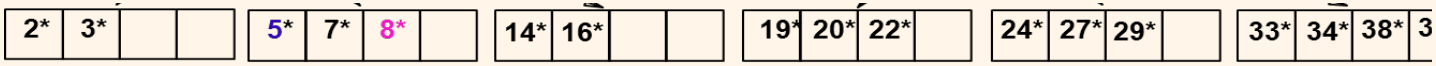
In both cases, log is a fractional number and must be rounded up to the nearest integer. Both formulas assume minimal free space in the index.

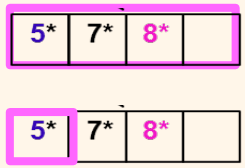
Dense indexes usually have four levels or less. Sparse indexes usually have three levels or less.

Table 15.2.1: Number of levels.

dense index		rows	
		1 million	1 billion
index entries per block	100	3	5
	400	3	4

sparse index		rows	
		1 million	1 billion
index entries per block	100	3	4
	400	2	3





8,000 bytes

26 bytes

F = 8,000 / 26
= 308 ~ 300

PARTICIPATION ACTIVITY 15.2.3: Number of levels.

Assume a table has 1,000,000 rows. Index entries are 26 bytes, and index blocks are 8 kilobytes.

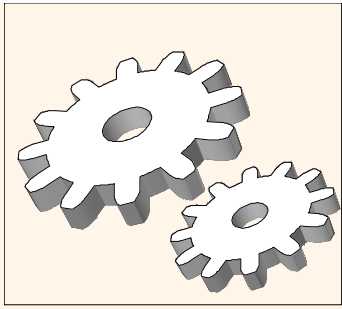
1) What is the fan-out for a multi-level index?

- ○ Approximately 200
- ◉ Approximately 300
- ○ Approximately 400

Correct

Fan-out is the number of entries per block = 8,000 bytes per index block / 26 bytes per index entry, or roughly 300.

?????



ISAM (Indexed Sequential Access Method)

14.1 Physical design

Present Note

MySQL storage engines

Logical design specifies tables, columns, and keys. The logical design process is described elsewhere in this material. ***Physical design*** specifies indexes, table structures, and partitions. Physical design affects query performance but never affects query results.

A ***storage engine*** or ***storage manager*** translates instructions generated by a query processor into low-level commands that access data on storage media. Storage engines support different index and table structures, so physical design is dependent on a specific storage engine.

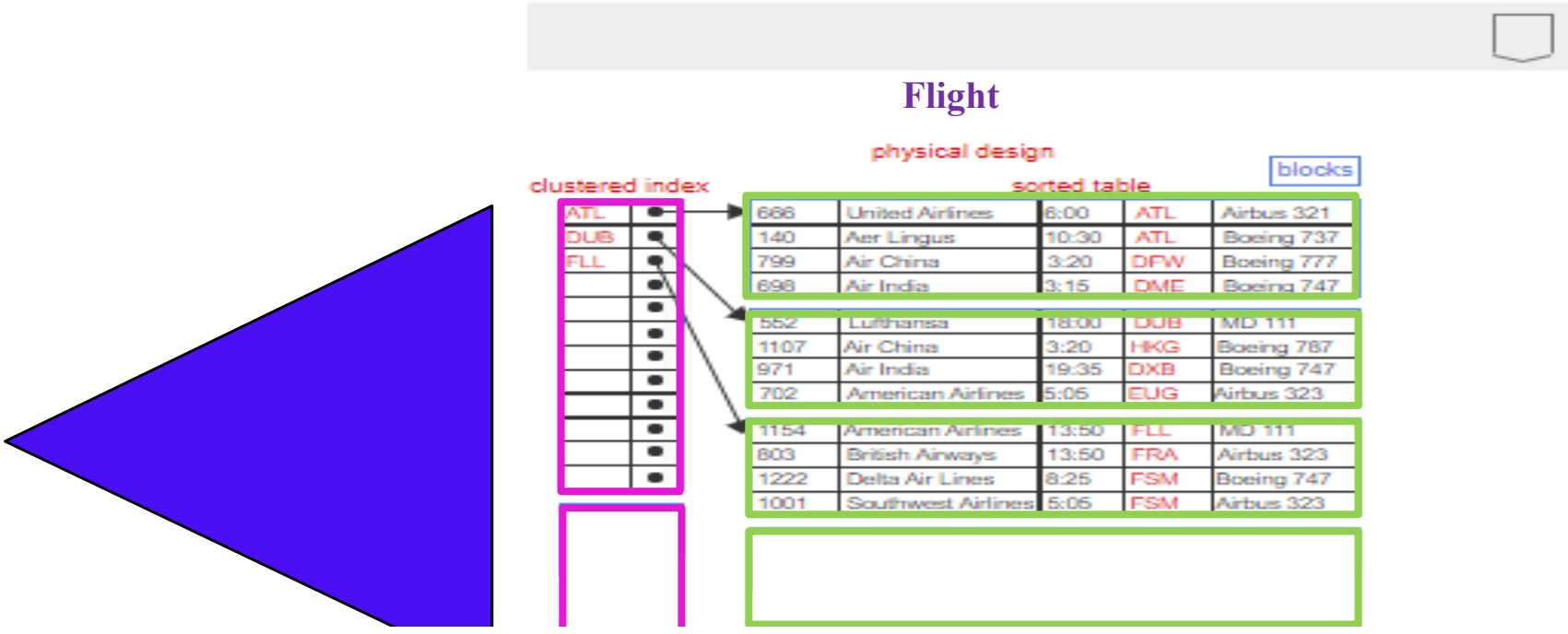
MySQL can be configured with several different storage engines, including:

- InnoDB is the default storage engine installed with the MySQL download. InnoDB has full support for transaction management, foreign keys, referential integrity, and locking.
- MyISAM has limited transaction management and locking capabilities. MyISAM is commonly used for analytic applications with limited data updates.
- MEMORY stores all data in main memory. MEMORY is used for fast access with databases small enough to fit in main memory.

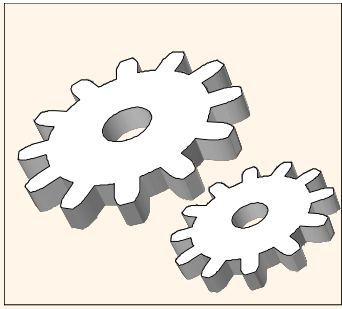
Different databases and storage engines support different table structures and index types. Ex:

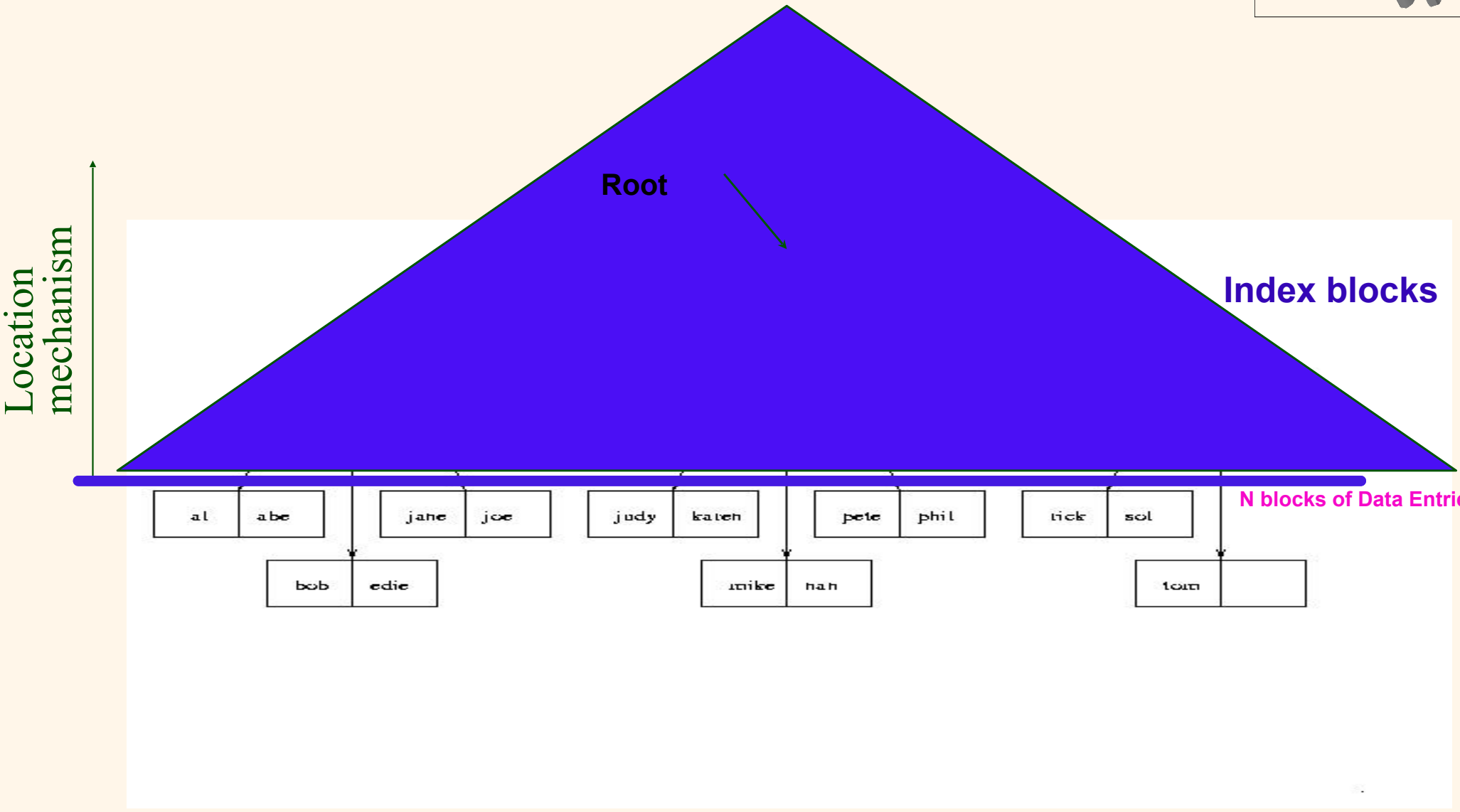
- Table structure. Oracle Database supports heap, sorted, hash, and cluster tables. MySQL with InnoDB supports only heap and sorted tables.
- Index type. MySQL with InnoDB or MyISAM supports only B+tree indexes. MySQL with MEMORY supports both B+tree and hash indexes.

This section describes the physical design process and statements for MySQL with InnoDB. The process and statements can be adapted to other databases and storage engines, but details depend on supported index and table structures.

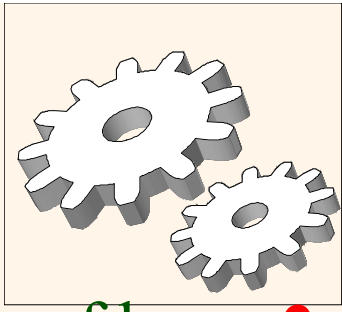


Index Sequential Access Method - ISAM



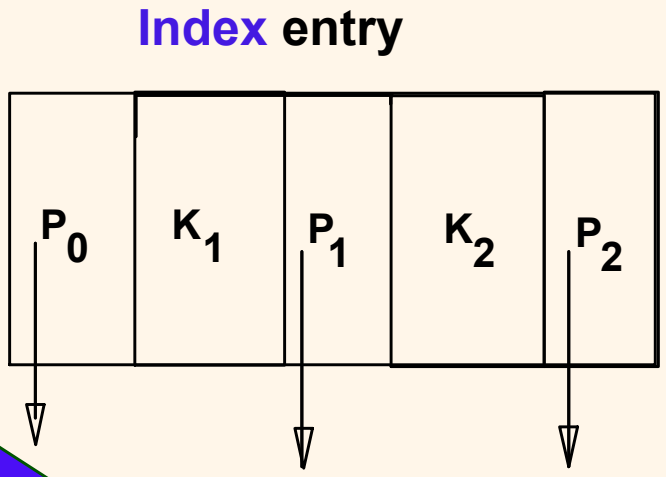


Example ISAM Tree



- Each node/page(Index k & Data Entry k*) can hold 2 entries.

m = number of keys **2**



F = fanout **3**

N = 6 blocks of Data Entries k*

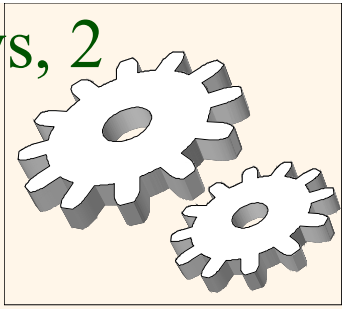
*Inserting 23**

*Inserting 48**

Inserting 41, 42** ...

m = number of keys, 2

F = fanout, 3



Root

1

40

Level 1

3 blocks of Search Tree Index k

1

20 | 33

51 | 63

Level 2

Primary

1

6 blocks of Leaf Data Entries k*

10* | 15*

20* | 27*

33* | 37*

40* | 46*

51* | 55*

63* | 97*

Level 3

1

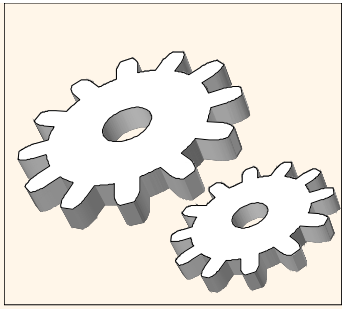
Overflow blocks

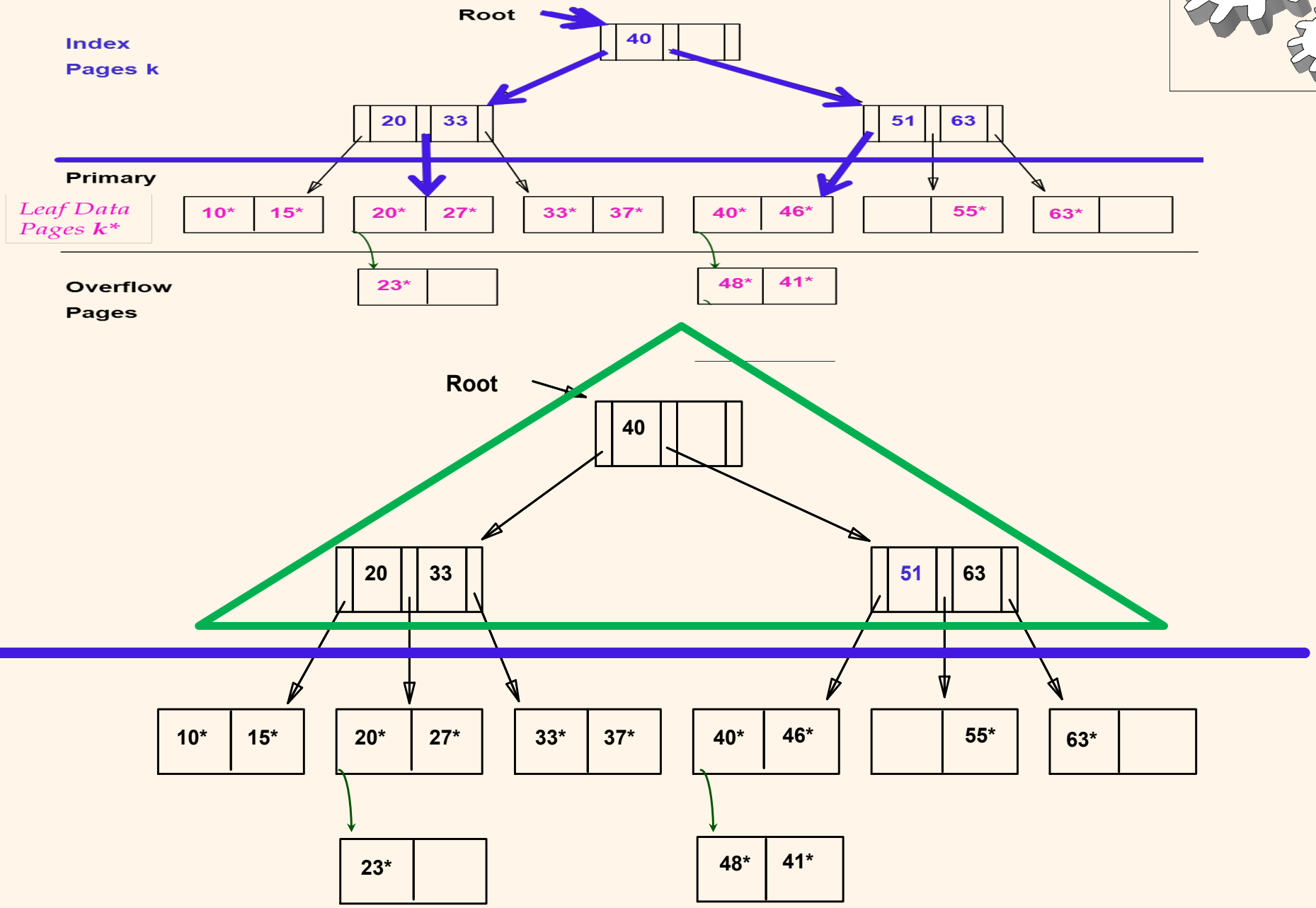
23*

48* | 41*

42*

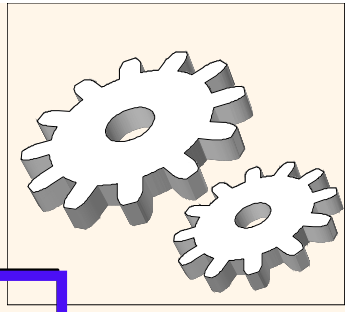
... Then Deleting 42*, 51*, 97*





? *Note that 51 appears in Index levels, but not in leaf!*

ISAM (Indexed Sequential Access Method)



? *N Leaf Data Entries blocks contain Data Entries* ***k****.*

Comments on ISAM

File creation: Leaf (Data Entries k*) blocks allocated sequentially,
Sorted by search key; then **Index blocks** allocated,
then space for **Overflow blocks**.

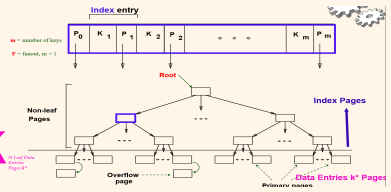
Index blocks k: <search key value, **pid**>;
`direct' search for *Leaf blocks*.

Leaf blocks k*: <search key value, **rid**>;
`points' to ***Data Record Page***.

Search k*: Start at Root; use key comparisons to go to Leaf Page.

$$\text{Cost} = \lceil \log_F N \rceil + 1 = \textbf{Tree Level}$$

F = # pointers/ **Index Page**, **N** = # **Leaf** (**Data Entry k**



Insert k*: Find Leaf (Data) Page belongs to, and put it there.

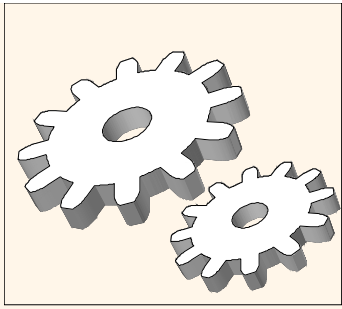
Delete k*: Find and remove from Leaf Page; if empty **Overflow Page**, de-alloca

Leaf (Data) blocks

Index blocks

Overflow blocks

? **Static Tree structure**: *inserts/deletes affect only Leaf (Data k*)*



B+ Tree: Most Widely Used Index

14.1 Physical design

Present Note

MySQL storage engines

Logical design specifies tables, columns, and keys. The logical design process is described elsewhere in this material. ***Physical design*** specifies indexes, table structures, and partitions. Physical design affects query performance but never affects query results.

A ***storage engine*** or ***storage manager*** translates instructions generated by a query processor into low-level commands that access data on storage media. Storage engines support different index and table structures, so physical design is dependent on a specific storage engine.

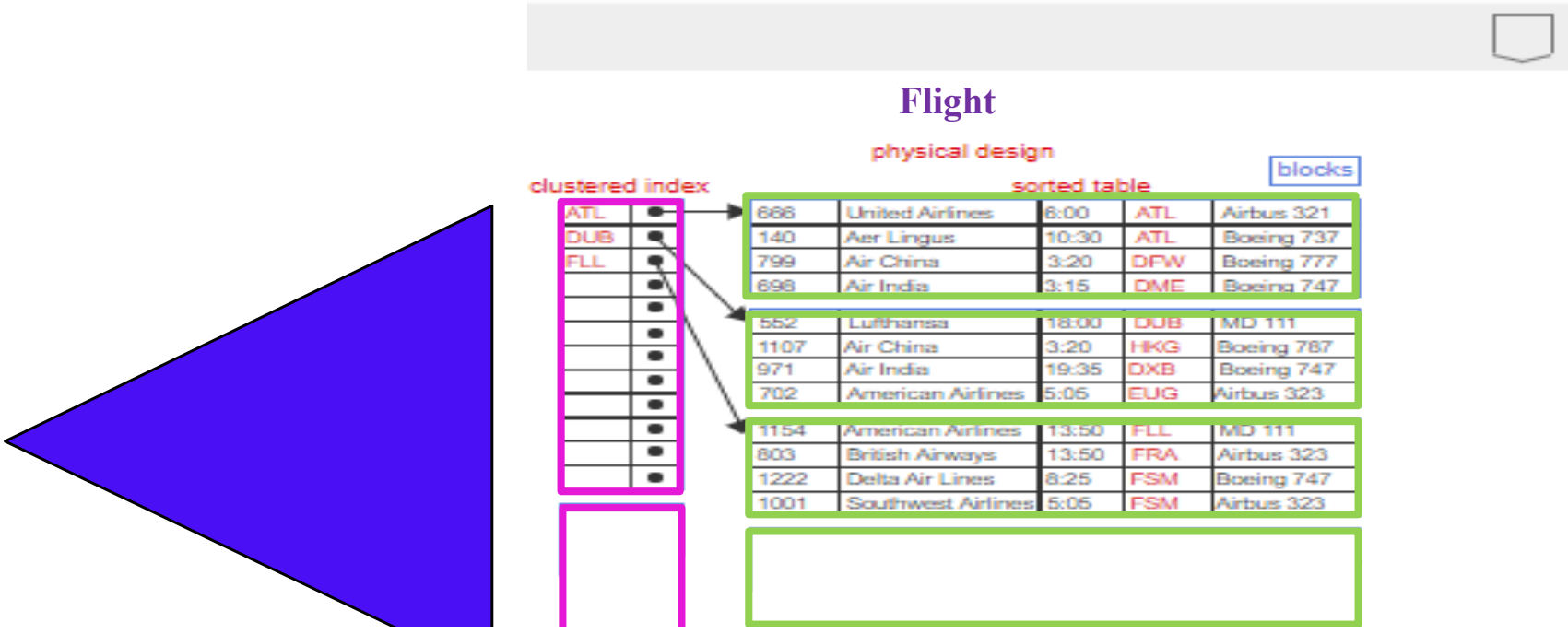
MySQL can be configured with several different storage engines, including:

- InnoDB is the default storage engine installed with the MySQL download. InnoDB has full support for transaction management, foreign keys, referential integrity, and locking.
- MyISAM has limited transaction management and locking capabilities. MyISAM is commonly used for analytic applications with limited data updates.
- MEMORY stores all data in main memory. MEMORY is used for fast access with databases small enough to fit in main memory.

Different databases and storage engines support different table structures and index types. Ex:

- Table structure. Oracle Database supports heap, sorted, hash, and cluster tables. MySQL with InnoDB supports only heap and sorted tables.
- Index type. MySQL with InnoDB or MyISAM supports only B+tree indexes. MySQL with MEMORY supports both B+tree and hash indexes.

This section describes the physical design process and statements for MySQL with InnoDB. The process and statements can be adapted to other databases and storage engines, but details depend on supported index and table structures.



B+ Tree: Most Widely Used Index

Cost to get to the **Leaf Page** is the **Level** of the **B+ Tree**

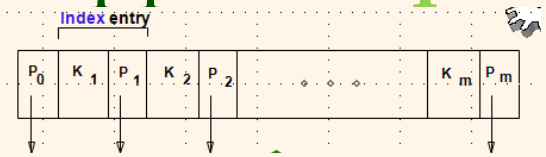
$$\lceil \log_F N \rceil + 1$$

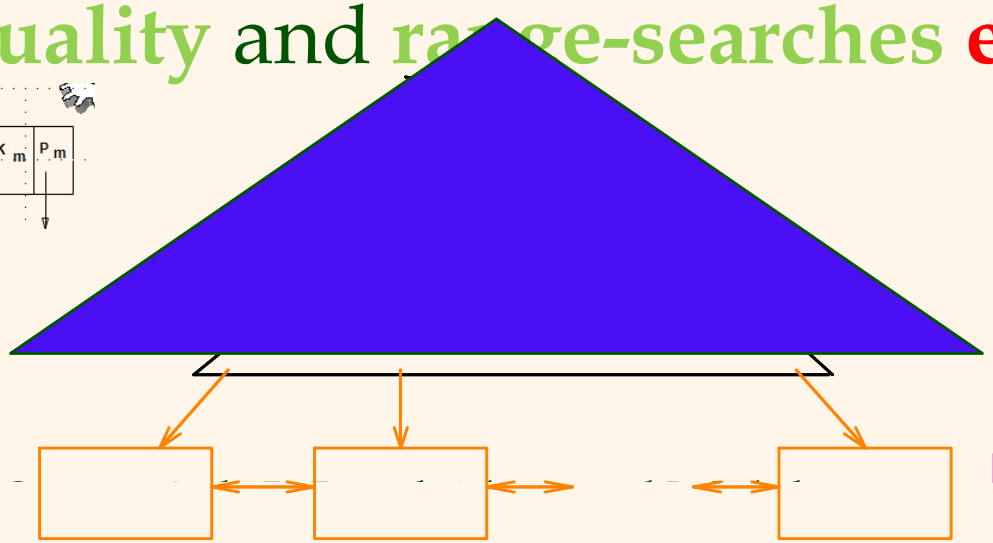
Keeps **Tree** *height-balanced.* (**F** = fanout, **N** = # leaf **blocks**)

Minimum 50% occupancy (except for Root Node/Page). Each Node contains **d** <= ***m*** <= 2**d** entries. The parameter **d** is called the *order* of the **Tree**.

F = **m**+1 – number of Pointers P_0 to P_m to other **Index blocks**

Supports **equality** and **range-searches** **efficiently**.





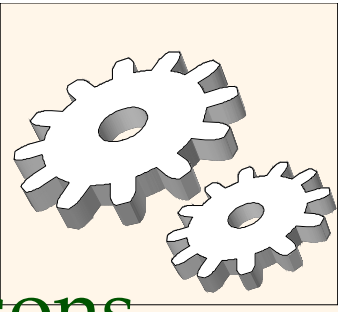
Index Page

(Direct search)

N blocks of Data Entries k*

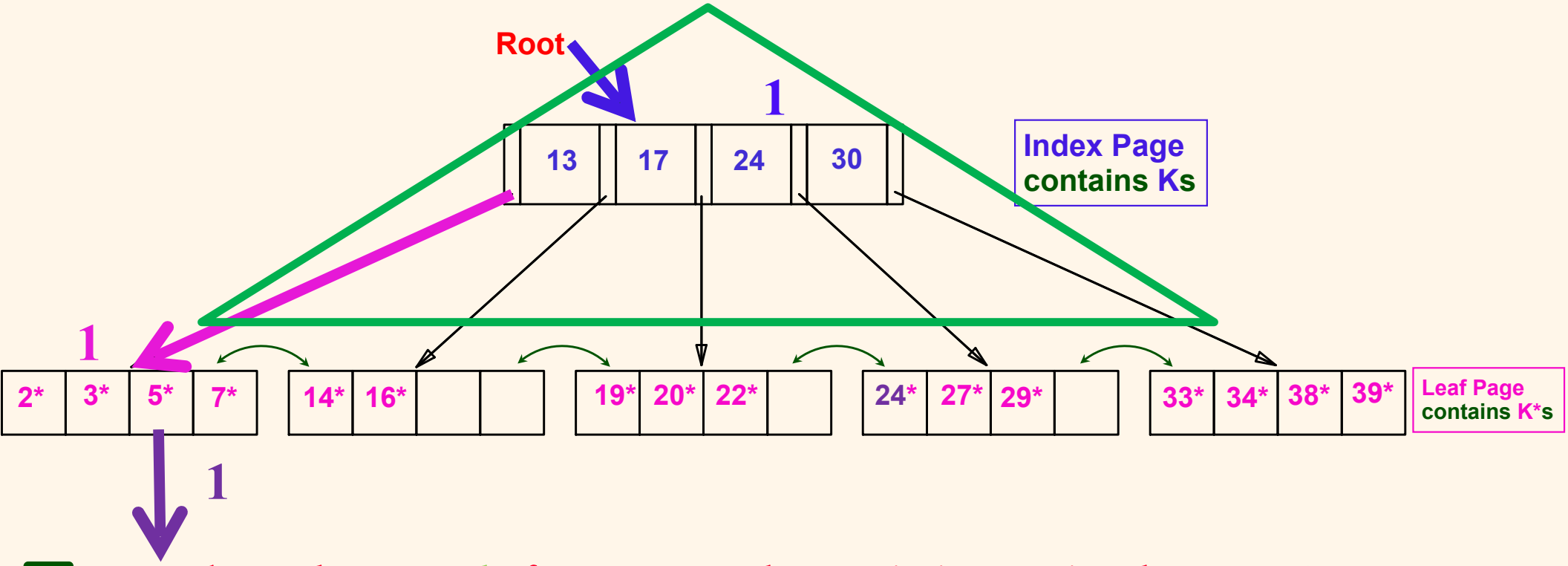
("Sequence set")

Example B+ Tree Index



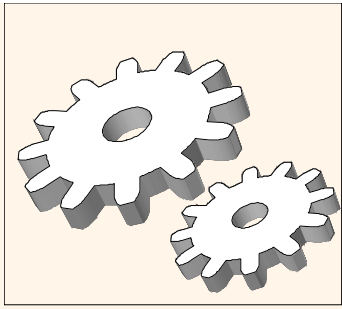
- Search for k* begins at Root, and **key** comparisons direct it to a **leaf Page of Data Entries k*** (as in **ISAM**).
- Search for 5*

Search for 15*, all data entries >= 24* ...



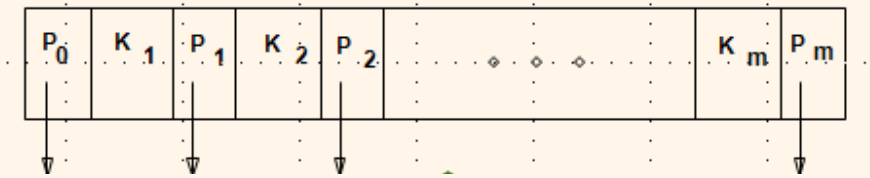
? *Based on the search for 15*, we know it is not in the Tree!*

B+ Tree Indexes in Practice



Typical order: **d** = 100. Typical **fill-factor**: 67%.

- average fanout **F**= 133 = **m** + 1

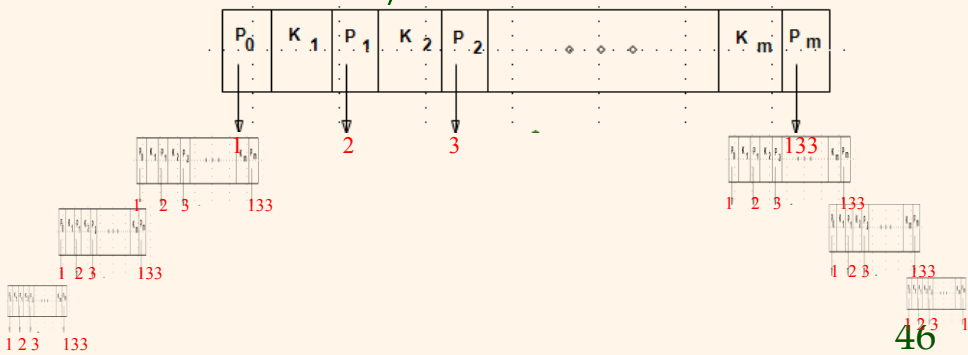


Typical capacities:

- **Height 4**: **133**^{4} = 312,900,700 k*
- **Height 3**: **133**^{3} = 2,352,637 k*

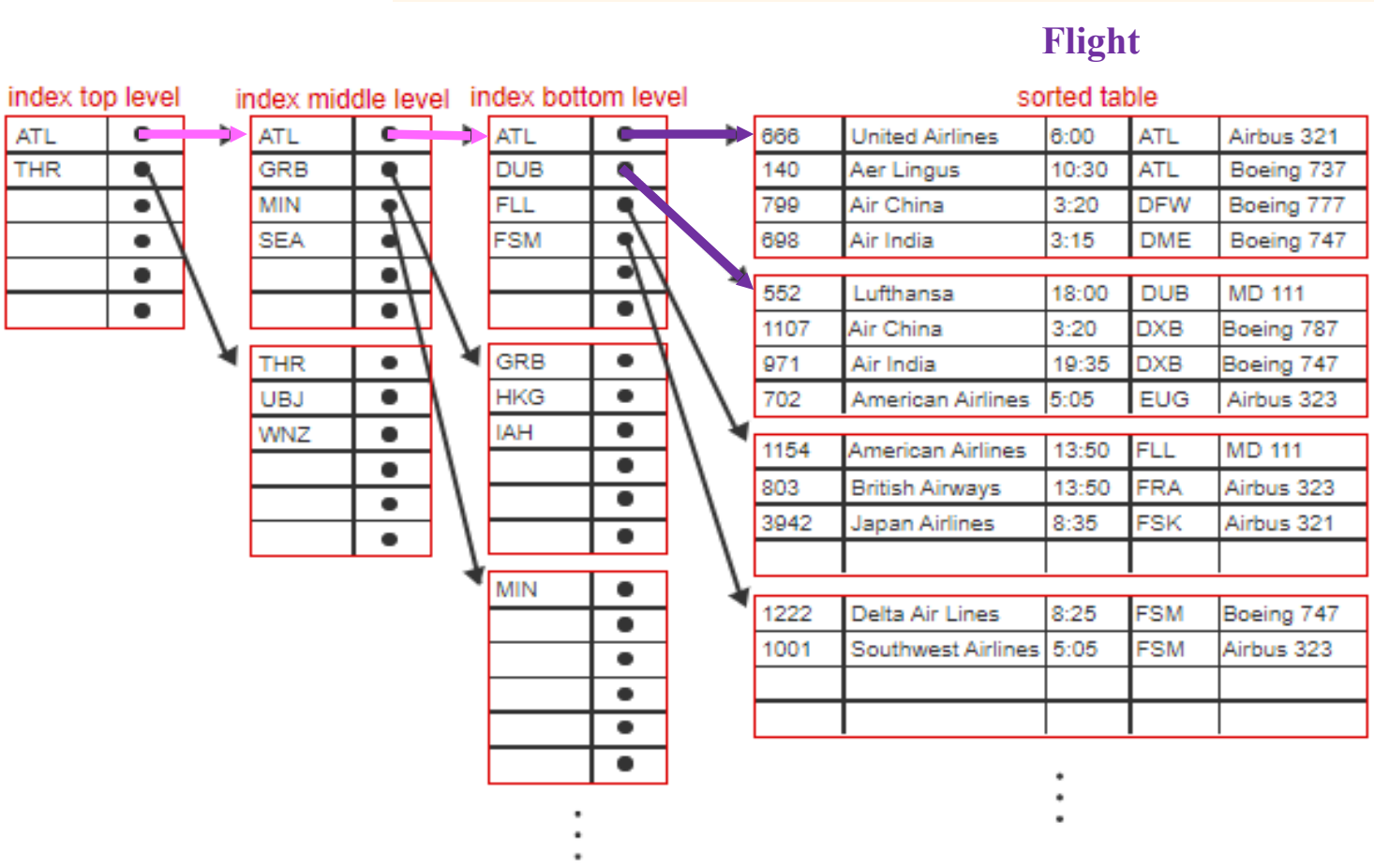
Can often hold top levels in **Buffer Pool**:

- **Level 1** = **1** block = 8 **KB**ytes
- **Level 2** = **133** blocks = 133 X 8 **KB**yte = 1064 **KB** ~ 1 **MB**yte
- **Level 3** = **133** X **133** = **17,689** blocks ~ 133 **MB**ytes



* Root is at **Level 1**!


Balanced indexes
index top level
index middle level
index bottom level
sorted table
ATL
THR
ATL
GRB
MIN
SEA
ATL
DUB
FLL
FSM
THR
UBJ
WNZ
GRB
HKG
IAH
MIN
668 United Airlines 8:00 ATL Airbus 321
140 Aer Lingus 10:30 ATL Boeing 737
799 Air China 3:20 DFW Boeing 777
698 Air India 3:15 DME Boeing 747
552 Lufthansa 18:00 DUB MD 111
1107 Air China 3:20 DXB Boeing 787
971 Air India 19:35 DXB Boeing 747
702 American Airlines 5:05 EUG Airbus 323
1154 American Airlines 13:50 FLL MD 111
803 British Airways 13:50 FRA Airbus 323
3942 Japan Airlines 8:35 FSK Airbus 321
1222 Delta Air Lines 8:25 FSM Boeing 747
1001 Southwest Airlines 5:05 FSM Airbus 323
N blocks of data entries
B blocks of data records




```
INSERT INTO Flight (FlightNumber, AirlineName, DepartureTime, DepartureAirport, AircraftType)
VALUES (3942, 'Japan Airlines', '8:35', 'FSK', 'Airbus 321')
```

PARTICIPATION ACTIVITY 15.2.7: Balanced indexes.

A table has a multi-level index with no free space in any index blocks.

1) If the index is sparse, an insert to the table always generates a new index level.

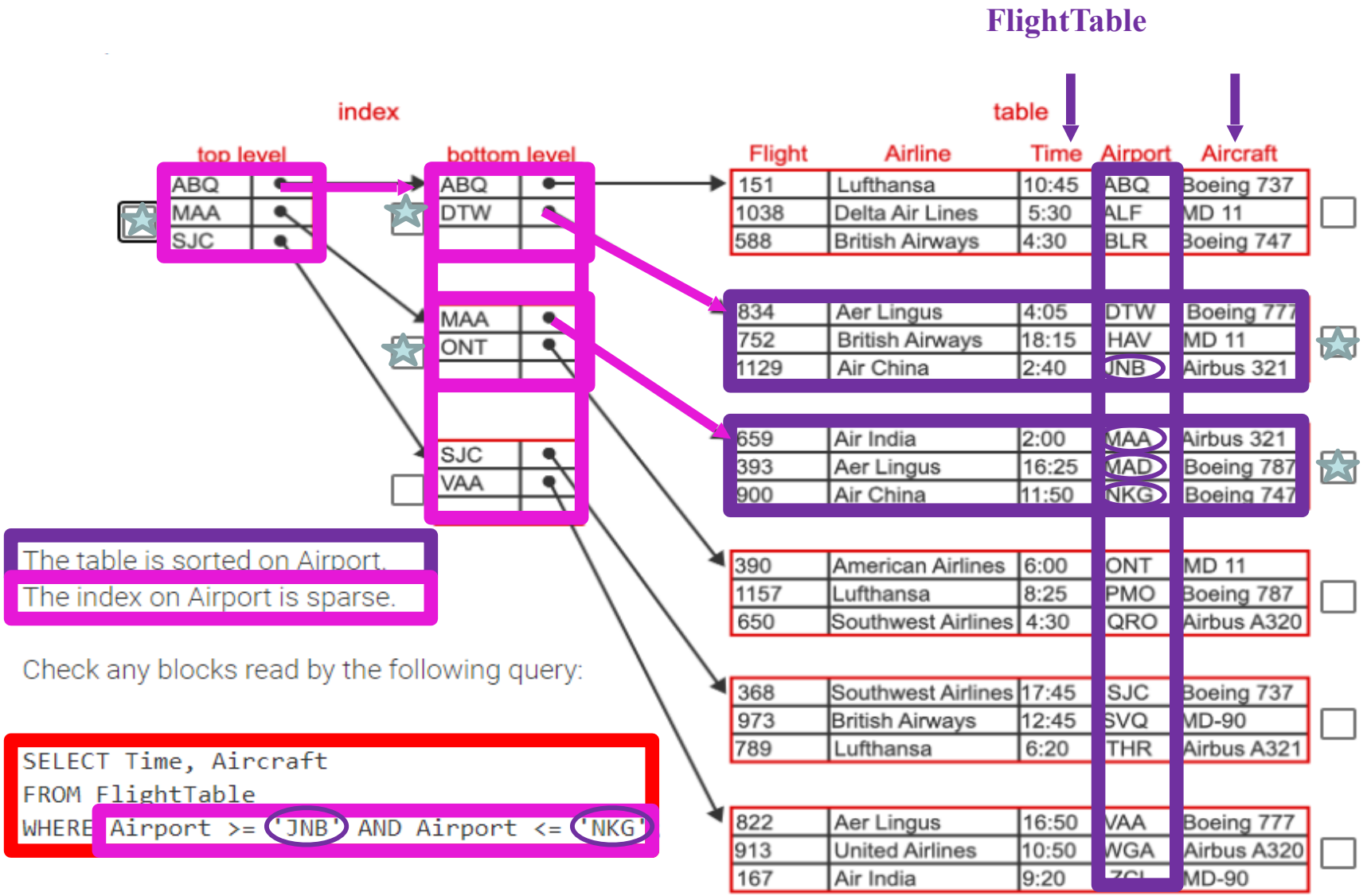
- ○ True
- ◉ False

Correct

Since the index is sparse, a new index entry is generated only when the insert causes a table block split.

?????

CHALLENGE ACTIVITY 15.2.1: Multi-level indexes.



The table is sorted on Airport.

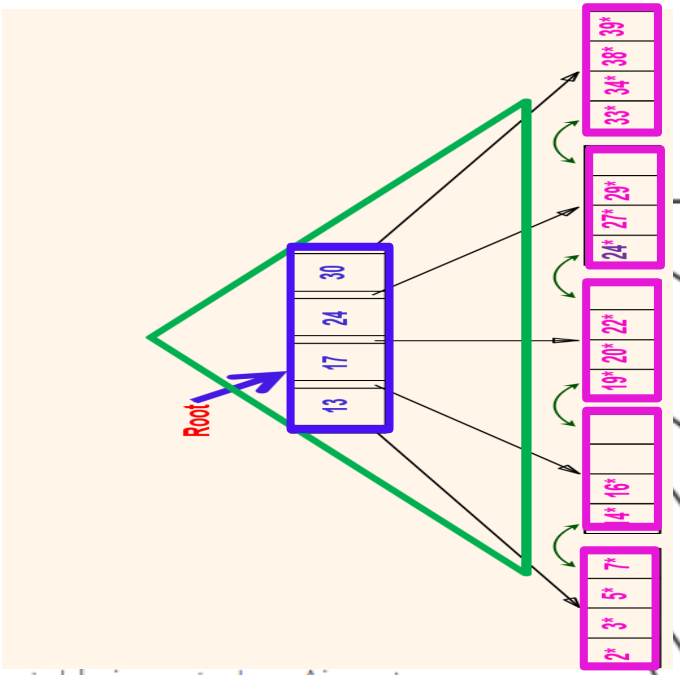
The index on Airport is sparse.

Check any blocks read by the following query:

```
SELECT Time, Aircraft
FROM FlightTable
WHERE Airport >= 'JNB' AND Airport <= 'NKG'
```

1

CHALLENGE ACTIVITY | 15.2.1: Multi-level indexes.



FlightTable

table

Flight	Airline	Time	Airport	Aircraft	
151	Lufthansa	10:45	ABQ	Boeing 737	
1038	Delta Air Lines	5:30	ALF	MD 11	
588	British Airways	4:30	BLR	Boeing 747	
834	Aer Lingus	4:05	DTW	Boeing 777	
752	British Airways	18:15	HAV	MD 11	
1129	Air China	2:40	JNB	Airbus 321	
659	Air India	2:00	MAA	Airbus 321	
393	Aer Lingus	16:25	MAD	Boeing 787	
900	Air China	11:50	NKG	Boeing 747	
390	American Airlines	6:00	ONT	MD 11	
1157	Lufthansa	8:25	PMO	Boeing 787	
650	Southwest Airlines	4:30	QRO	Airbus A320	
368	Southwest Airlines	17:45	SJC	Boeing 737	
973	British Airways	12:45	SVQ	MD-90	
789	Lufthansa	6:20	THR	Airbus A321	
822	Aer Lingus	16:50	VAA	Boeing 777	
913	United Airlines	10:50	WGA	Airbus A320	
167	Air India	9:20	ZCL	MD-90	

The table is sorted on Airport.
The index on Airport is sparse.

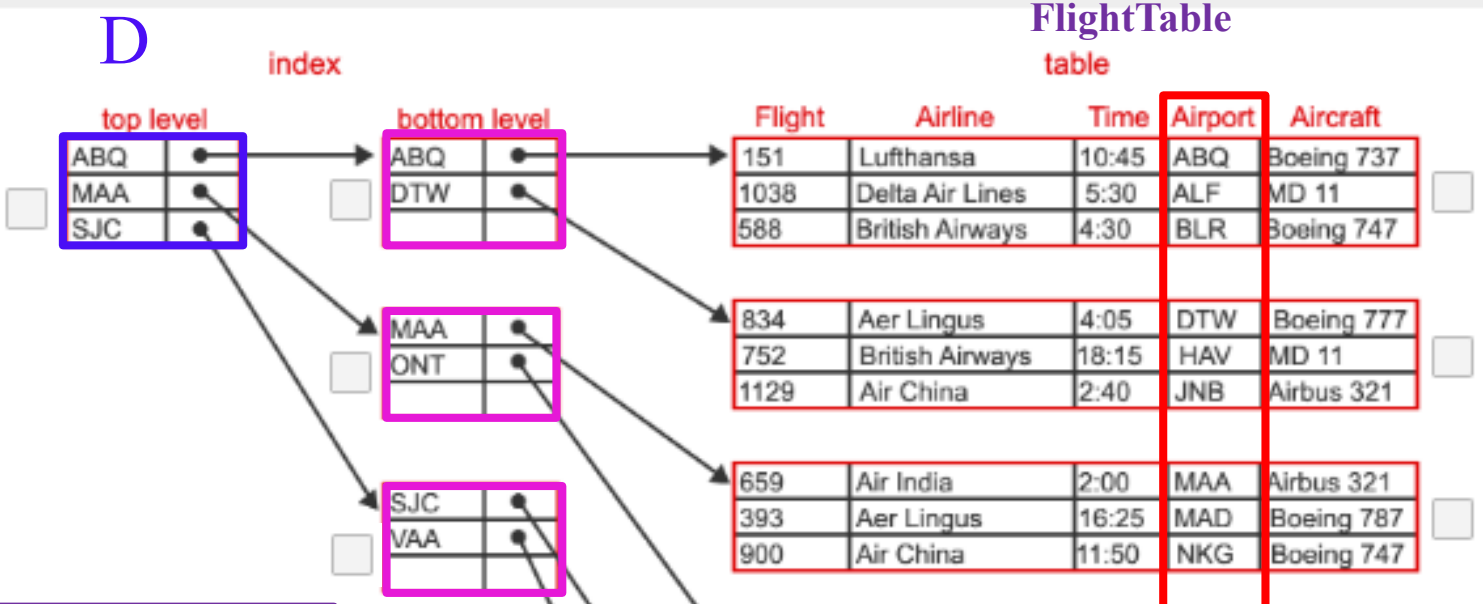
Check any blocks read by the following query:

```
SELECT Time, Airline
FROM FlightTable
WHERE Airport = 'MAD' OR Airport = 'SJC';
```

1

VH

CHALLENGE ACTIVITY | 15.1.1: Multi-level indexes.



The table is sorted on Airport.

The index on Airport is sparse

Check any blocks read by the following query:

```
SELECT Aircraft, Time
FROM FlightTable
WHERE Airport = 'WGA' OR Airport = 'ZCL';
```

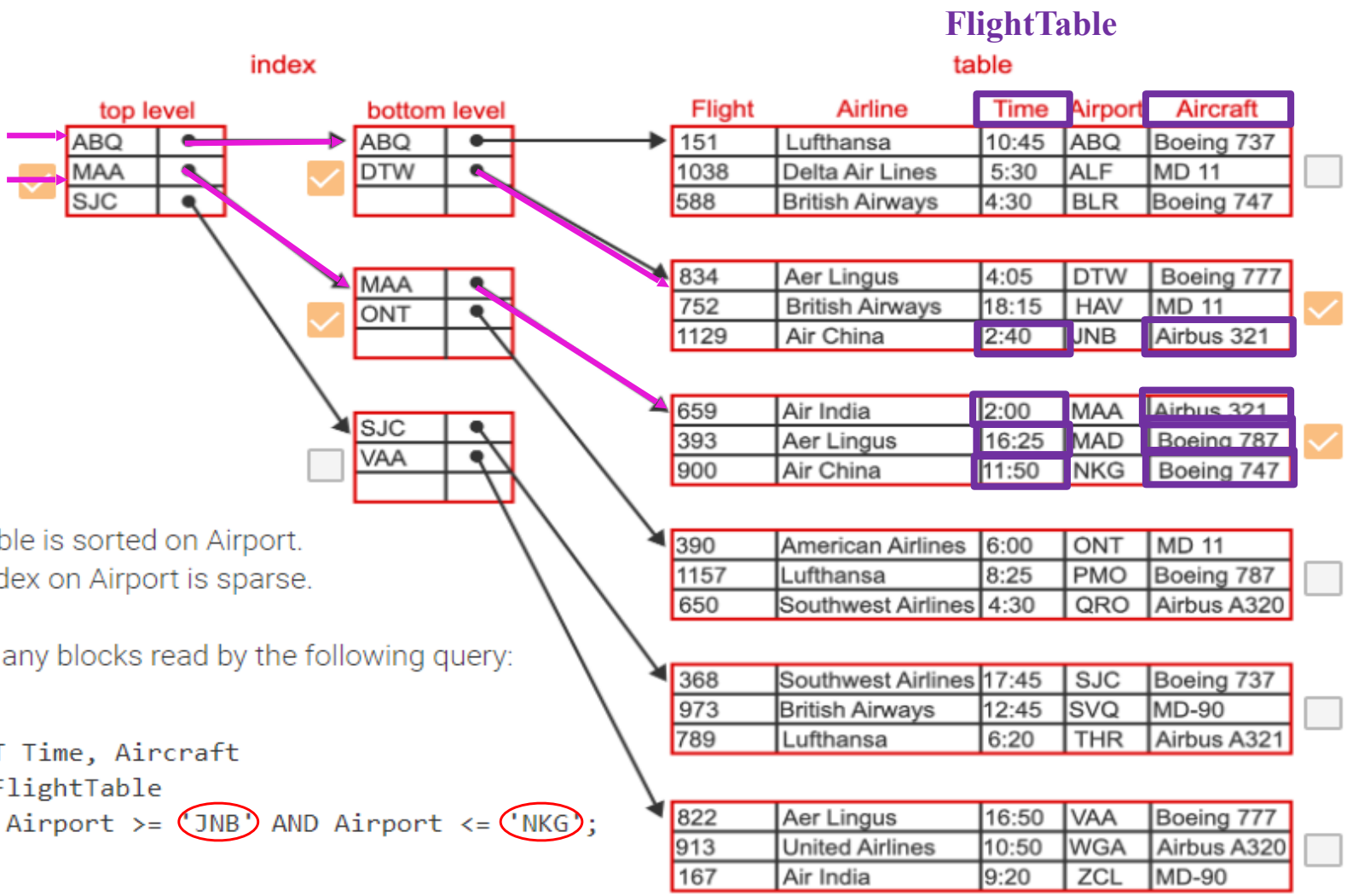
1

Aircraft	Time
Airbus A320	10:50
MD-90	9:20

?????

VH

CHALLENGE ACTIVITY | 15.2.1: Multi-level indexes.



The table is sorted on Airport.
The index on Airport is sparse.

Check any blocks read by the following query:

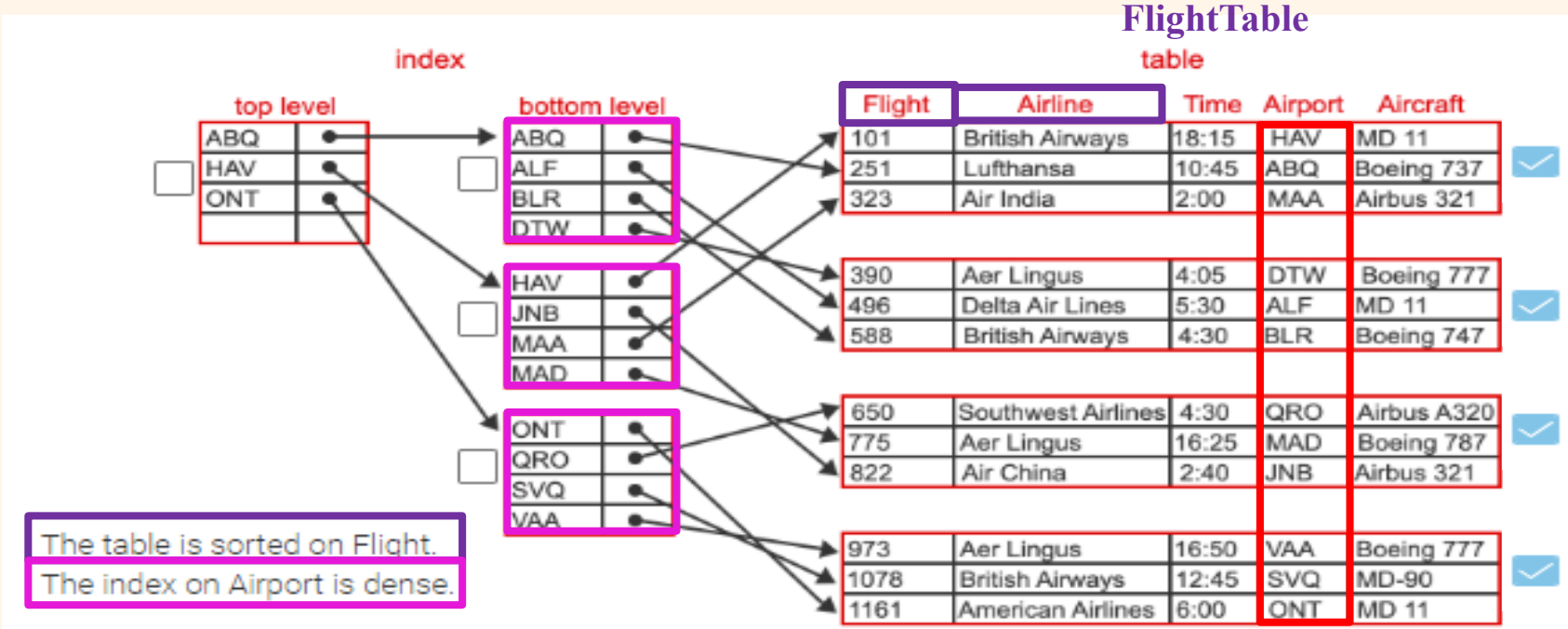
```
SELECT Time, Aircraft
FROM FlightTable
WHERE Airport >= 'JNB' AND Airport <= 'NKG';
```

1

?????

VH

CHALLENGE ACTIVITY | 15.1.1: Multi-level indexes.



Check any blocks read by the following query:

```
SELECT Airline, Flight
FROM FlightTable
WHERE Airline > 'Lufthansa';
```

2

Index on Airport

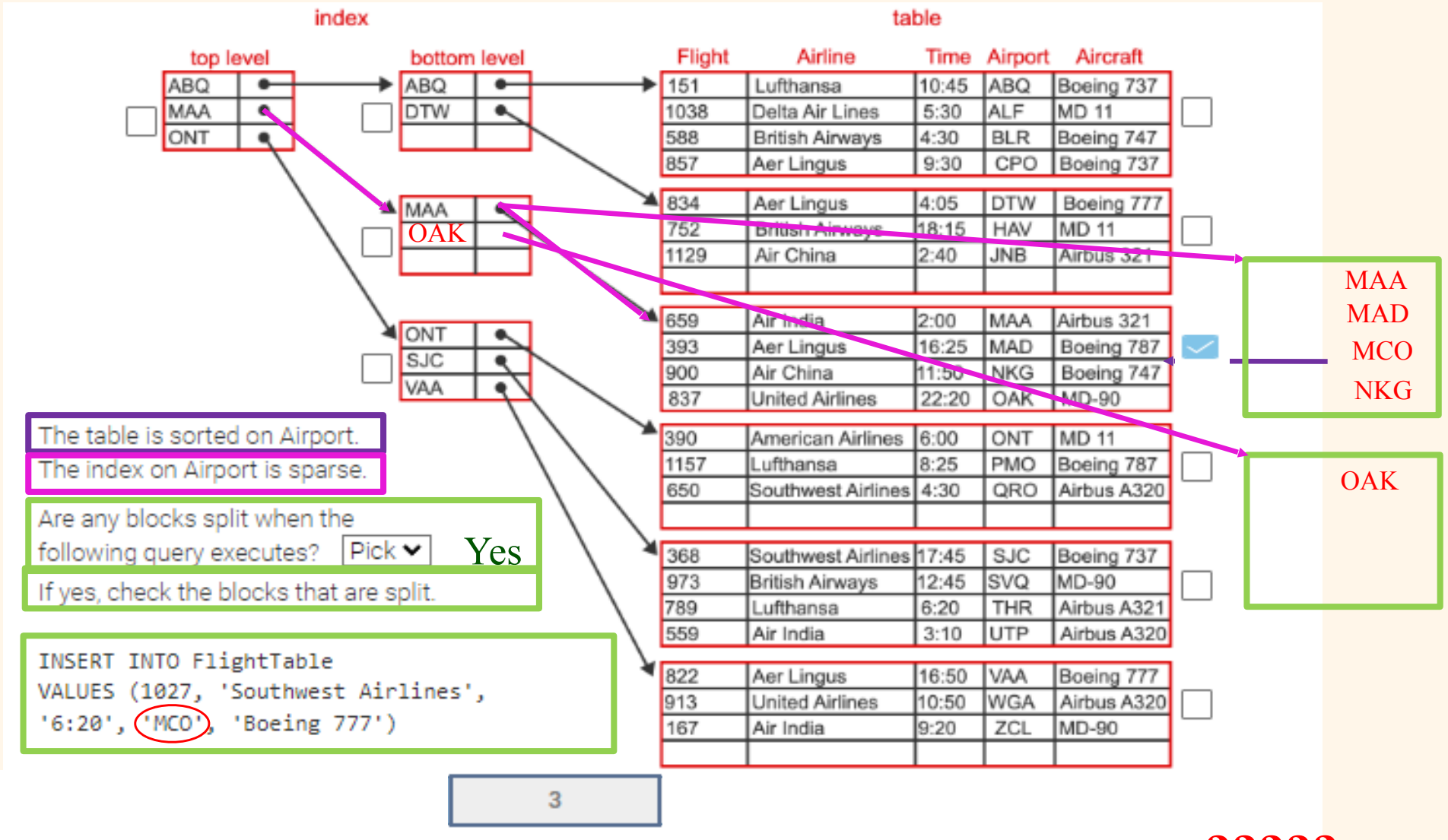
All table blocks **B**

?????

VH

CHALLENGE ACTIVITY | 15.1.1: Multi-level indexes. | 1027, 'Southwest Airline", '6:20', '**MCO**', 'Boeing 777'

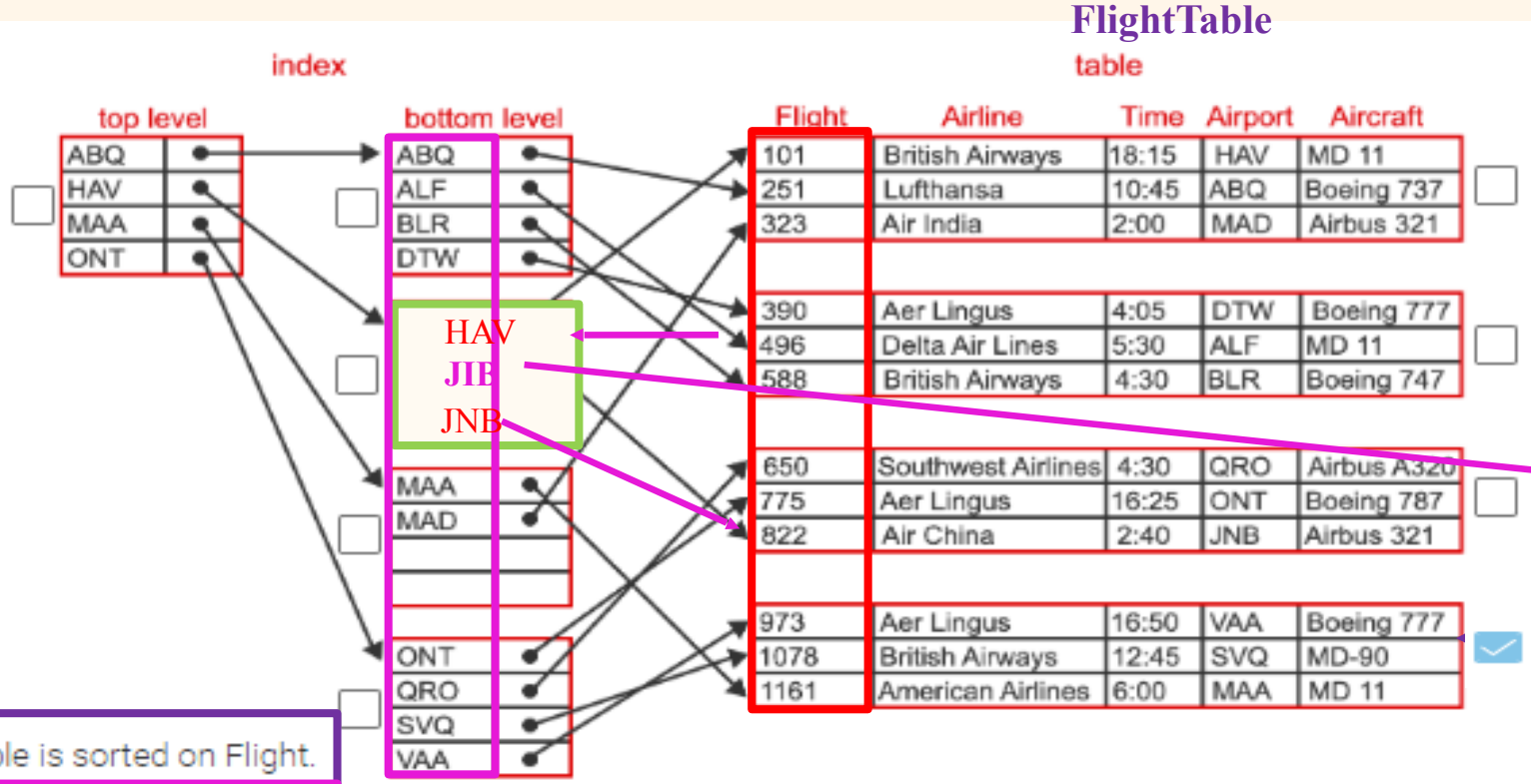
FlightTable



?????

CHALLENGE ACTIVITY | 15.1.1: Multi-level indexes.

995, 'American Airlines", '2:15', '**JIB**', 'Boeing 747'



Check any blocks that split when the following query executes:

```
INSERT INTO FlightTable
VALUES (995, 'American Airlines', '2:15', 'JIB', 'Boeing 747')
```

4

VH

?????

TA time (Alvaro)
(CA 15.1.1 Step 1– Multi-level indexes)

TA time (Alvaro)
(CA 15.1.1 Step 2– Multi-level indexes)

TA time (Alvaro)
(CA 15.1.1 Step 3– Multi-level indexes)

TA time (Alvaro)
(CA 15.1.1 Step 4– Multi-level indexes)

B+Tree Index Practice Questions

15. SET 3 - 3:STORAGE III B TREE INDEX

Branches that are similar in length in an index hierarchy are ____.

a. dense

b. balanced

c. sparse

d. primary

????

9

15. SET 3 - 3:STORAGE III B TREE INDEX

How are blocks read with a single-level index scan?

a. A scan reads all index blocks to find table blocks that contain selected rows. The table blocks are then read.

b. A scan is initiated to read all table blocks. Index blocks are then read.

c. The search reads one index block plus selected table blocks.

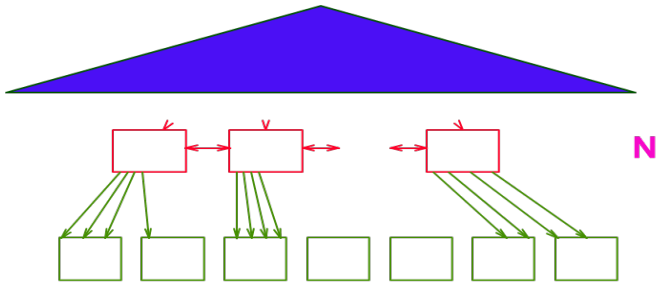
d. A scan is initiated to read all table blocks that contain index blocks with selected data.

????

10

At 5:00 PM .

03.25.2024	ZyBook SET 3- 3	Set 3
(18 – Mo)		LECTURE 14 STORAGE III B TREE INDEX



Cost = $\lceil \log_F N \rceil + 1$ = Tree Level

F = # pointers/ Index Page, N = # Leaf (Data Entry k*) blocks

N blocks of Data Entries k*

B blocks of Data Records

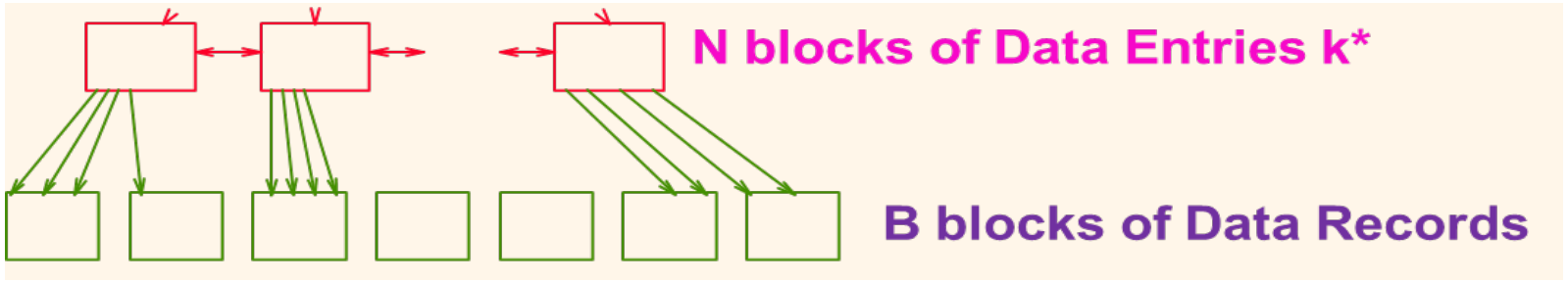
VH work on
SET 3 – 3: Storage III B Tree Index

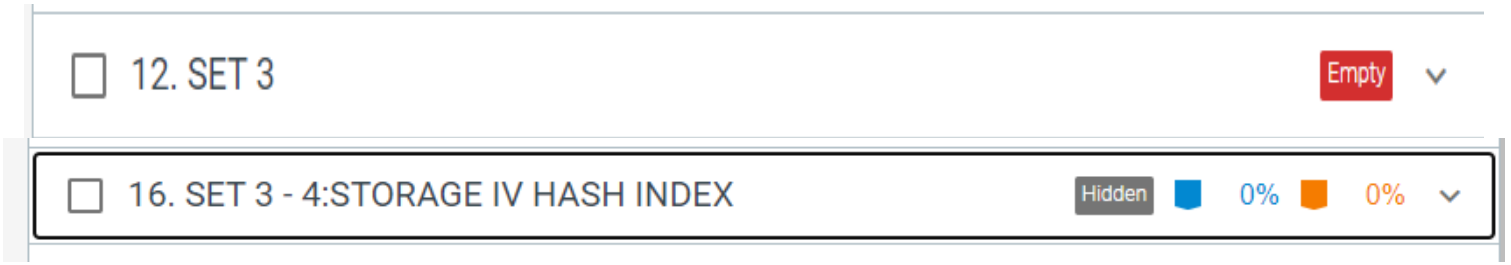
Next

03.27.2024	ZyBook SET 3 - 4	Set 3
(19 - We)		LECTURE 15 STORAGE IV HASH INDEX

HASH

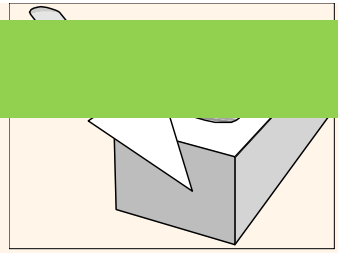
O(1)





VH, unHIDE

From 5:05 to 5:15 PM – 5 minutes.



03.25.2024	ZyBook SET 3- 3	Set 3
(18 – Mo)		LECTURE 14 STORAGE III B TREE INDEX

CLASS PARTICIPATION 20 points — 20% of Total

B PLUS TREE

Class 18 END PARTICIPATION

Not available until Mar 25 at 5:05pm | Due Mar 25 at 5:15pm | 100 pts

VH, publish

This is a synchronous online class.

Attendance is required.

Recording or distribution of class materials is prohibited.

1. At the beginning of selected classes there is an **assessment in the first 10 minutes**. (**beige** BOX in the Detailed Syllabus)
2. At the end of selected classes there is an **assessment in the last 10 minutes**. (**blue** BOX in the Detailed Syllabus)
3. ZyBook sections will be downloaded and used for 30% of Total Score on the dates specified in the Detailed Syllabus.
4. EXAMS are in CANVAS. **No late EXAMS**.
5. I have to be **present in TEAMS in order to take any graded assignment assigned during that class.**

At 5:15 PM.

End Class 18

VH, unhide ZyBook Section 16.

VH, Download Attendance Report
Rename it:
3.25.2024 Attendance Report FINAL

VH, upload Class 18 to CANVAS.