Datawarehouse Space Management

— UNIT-03—

Oracle Documentation

Partition

- Partition allows tables ,indexes into smaller and more manageable pieces called partitions

Transaction Table

Tran_id	Cust_id	Amount	Type	Date	Region
1	7	90	Credit	10 Jan	AP
2	9	78	Debit	10 Jan	AP
3	71	90	Debit	10 Jan	EMEA
4	78	567	Credit	10 Jan	EMEA
5	9	89	Debit	10 Jan	NA

Partition by region

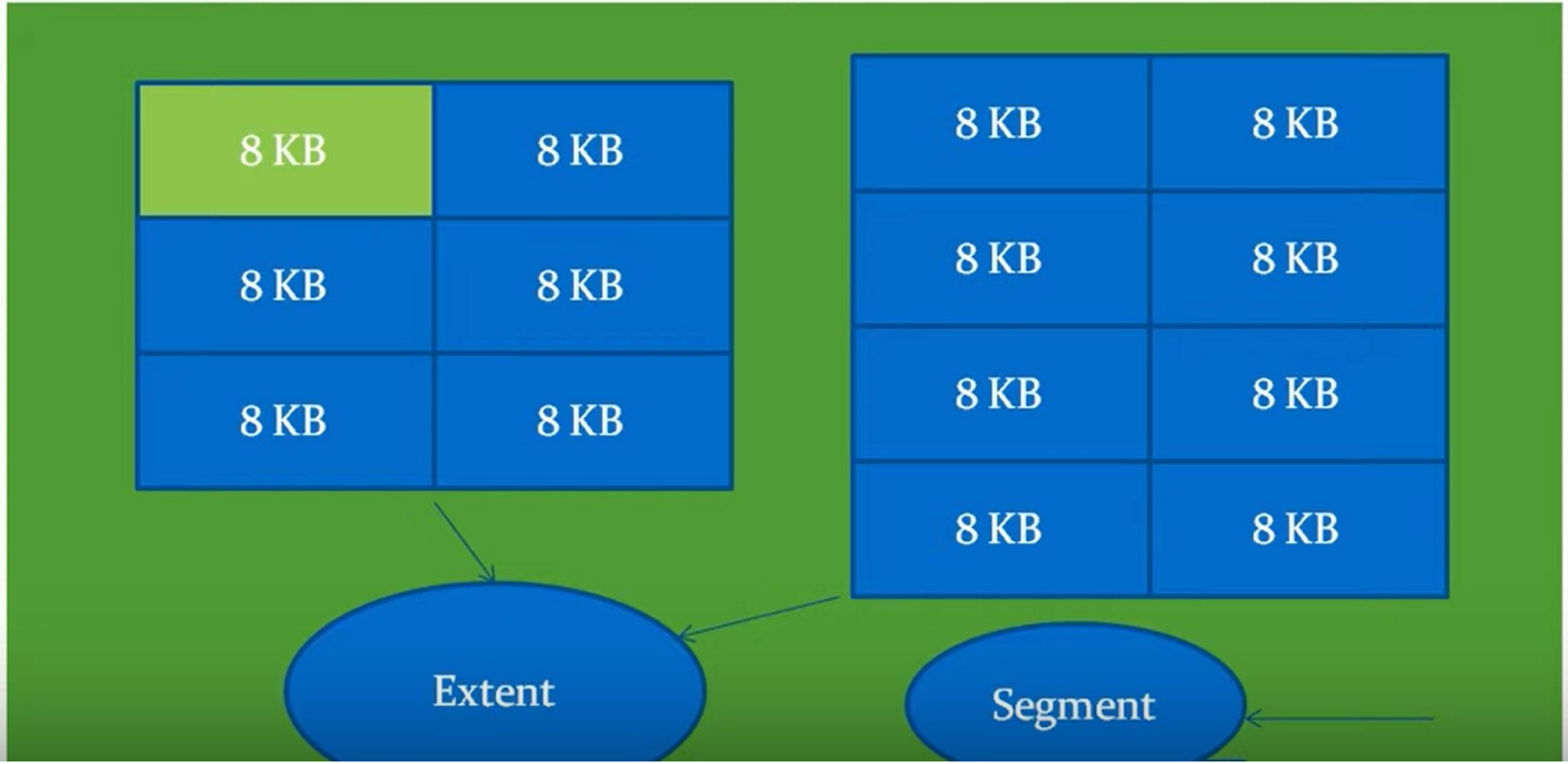
Tran_id	Cust_id	Amount	Type	Date	Region
1	7	90	Credit	10 Jan	AP
2	9	78	Debit	10 Jan	AP

3	71	90	Debit	10 Jan	EMEA
4	78	567	Credit	10 Jan	EMEA

Advantage of Partition

- Performance Improvement
- Manageability
- Availability

Blocks



When to Partition a Table

There are certain situations when you would want to partition a table.

Here are some suggestions for situations when you should consider partitioning a table:

- Tables that are greater than 2 GB.

  These tables should always be considered as candidates for partitioning.

- Tables that contain historical data, in which new data is added into the newest partition.

  A typical example is a historical table where only the current month's data is updatable and the other 11 months are read only.

- Tables whose contents must be distributed across different types of storage devices.

Partitioning Strategies

- Oracle Partitioning offers three fundamental data distribution methods as basic partitioning strategies that control how data is placed into individual partitions.
  These strategies are:
  - Range
  - Hash
  - List
- Using these data distribution methods, a table can either be partitioned as a single level or as a composite-partitioned table:
  - **Single-Level Partitioning**
  - **Composite Partitioning**
- Each partitioning strategy has different advantages and design considerations. Thus, each strategy is more appropriate for a particular situation

Range Partition

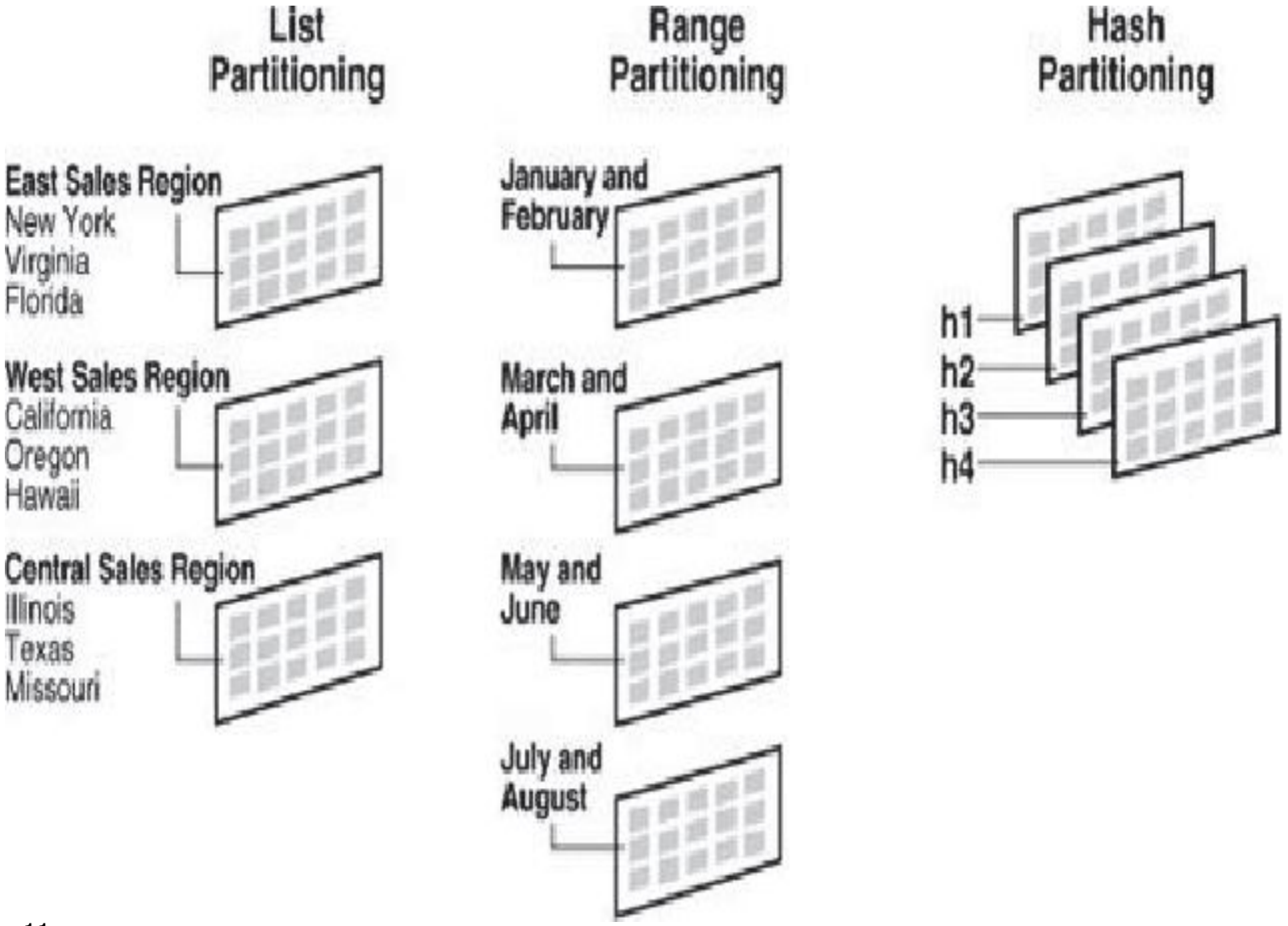
```
create table employees
 (emp_no number(5),emp_name varchar(2))
partition by range(emp_no)
(partition p1 values less than(100),
 partition p2 values less than(200),
 partition p3 values less than(300),
);
```

Error : ORA-14400: inserted partition key does not map to any partition

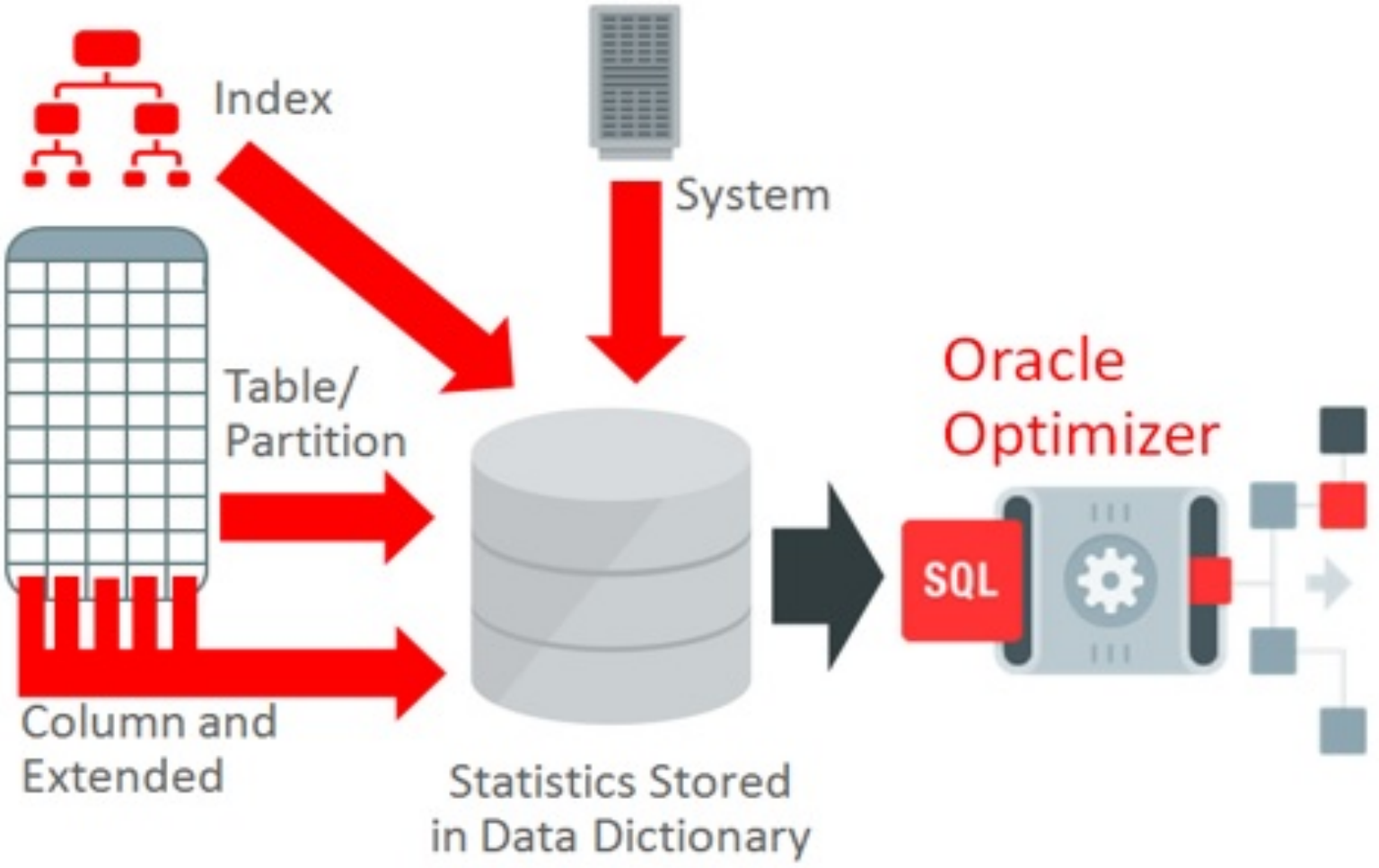
```
partition p4 values less than (maxvalue)
```

When and why ?

- When the dataset is above 2gb
- Use Range partitioning when you plan to access medium data aggregations/dumps on a frequent basis based on dates (Using the partition key column)
- When you have rolling months of data in which only current month of data is updatable while the rest is read only.

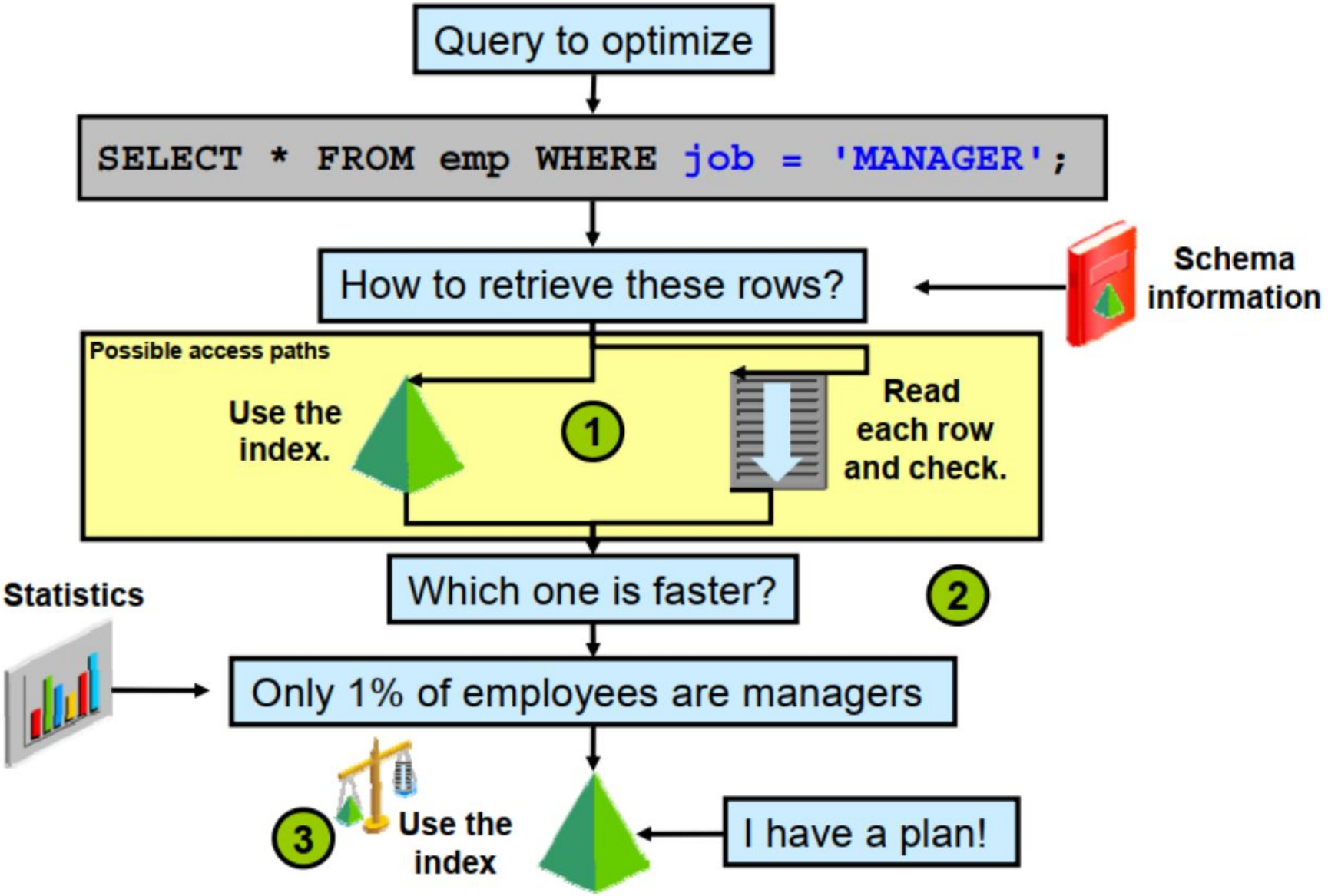

List Partitioning
East Sales Region
New York
Virginia
Florida
West Sales Region
California
Oregon
Hawaii
Central Sales Region
Illinois
Texas
Missouri
Range Partitioning
January and February
March and April
May and June
July and August
Hash Partitioning
h1
h2
h3
h4

Query Optimizer



https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL213

How Oracle Optimizer Works



INDEXING

- Indexes are used to improve the performance of your query
- Save us from full table scan

Follow these guidelines for creating indexes

- Do not create indexes on small tables i.e. where numbers of rows are less.
- Do not create indexes on those columns which contain many null values.
- Limit the number of indexes on tables because, although they speed up queries, but at the same time DML operations becomes very slow as all the indexes have to updated whenever an Update, Delete or Insert takes place on tables.

Table

Emp_id	Name	Salary	Hir_date	Cmsn	Manager	Dept
4532 → 1	Abhinav	120000	01-01-19	0.10	Null	10
2	Ankur	230000	01-04-18	0.20	1	10
.	.		.	.	.	.
7501	.		.	.	.	.
.	.		.	.	.	.
9870	Shridahr	32192	02-01-20	0.50	1	10

Select * from employees where emp_id = 4532

Index

Emp_id
1
2
.
4532
.
9870

Table

Emp_id	Name	Salary	Hir_date	Cmsn	Mgr	Dept
1	Abhinav	120000	01-01-19	0.10	Null	10
2	Ankur	230000	01-04-18	0.20	1	10
.	.		.	.	.	.
3896	.		.	.	.	.
4532	.		.	.	.	.
9870	Shridahr	32192	02-01-20	0.50	1	10

Select * from employees where emp_id = 4532

Types of Indexes

- B-tree indexes
- Bitmap and bitmap join indexes
- Function-based indexes
- Application domain indexes

- Default index is B-Tree index

Creating Indexes (B-Tree) Simple Index

To create an Index give the create index command. For example the following statement creates an index on emp_id column of employees table.

Example:

Syntax

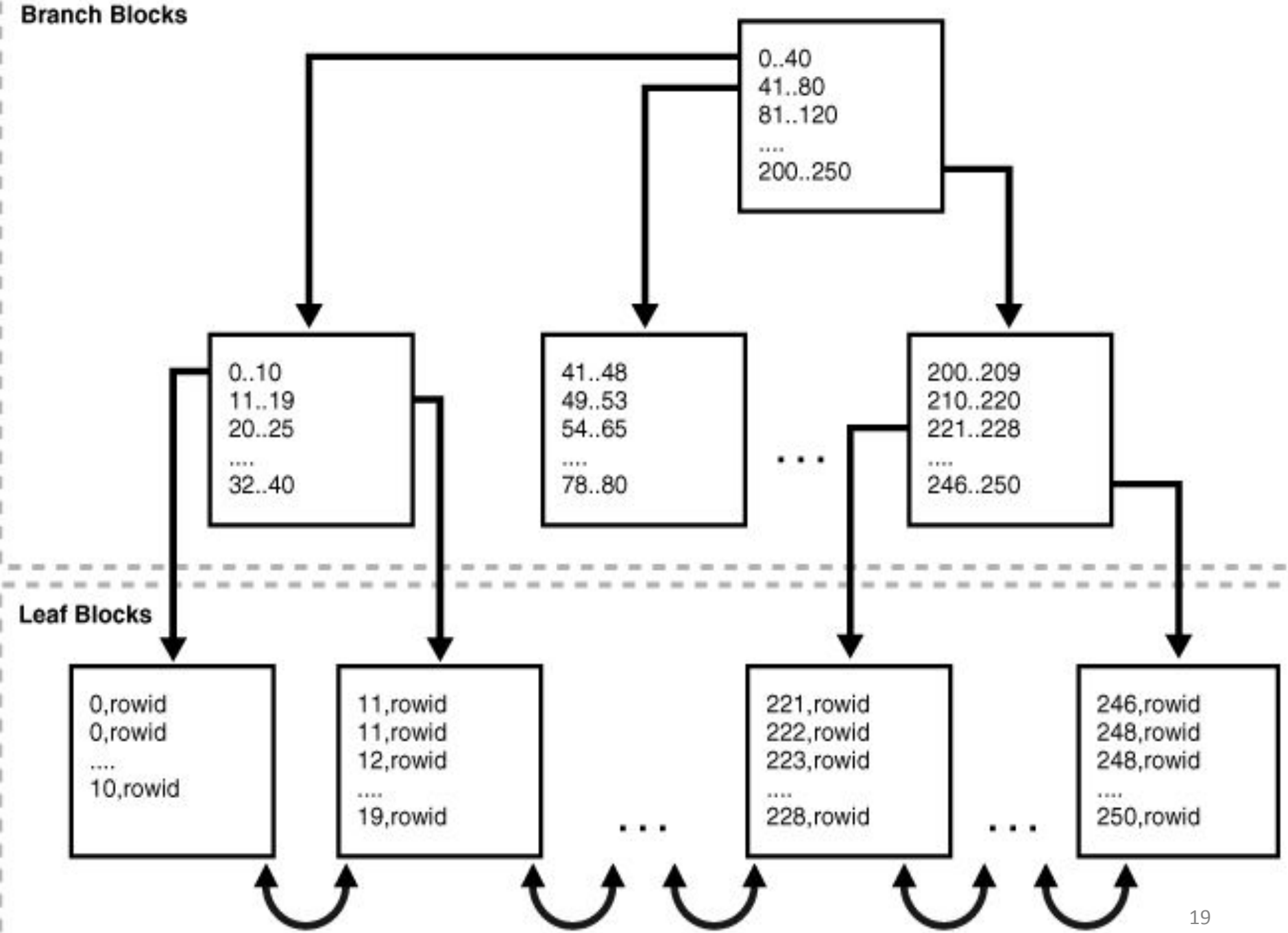
Create <index_type> index <index_name> on <table_name(attribute_name)>;

create index empno_ind on employees (emp_id);

B-Tree Index

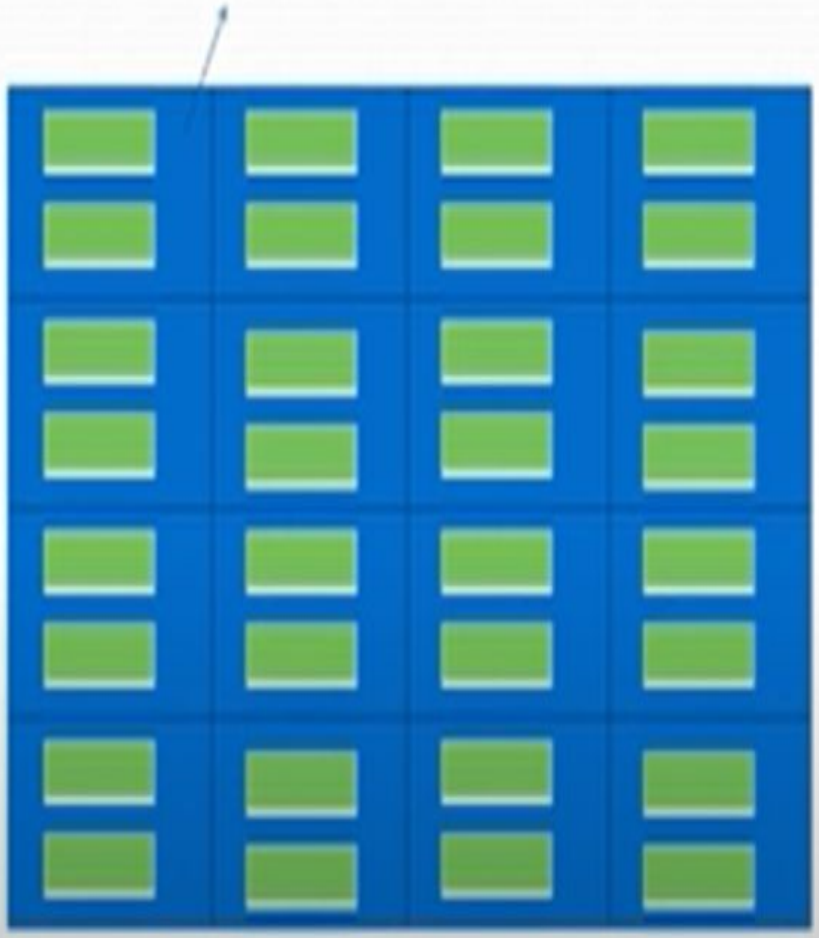
- These indexes are the standard index type.
- It is balance tree index
- They are excellent for primary key and highly-selective indexes.
- B-tree indexes can retrieve data sorted by the indexed columns.
- Types :

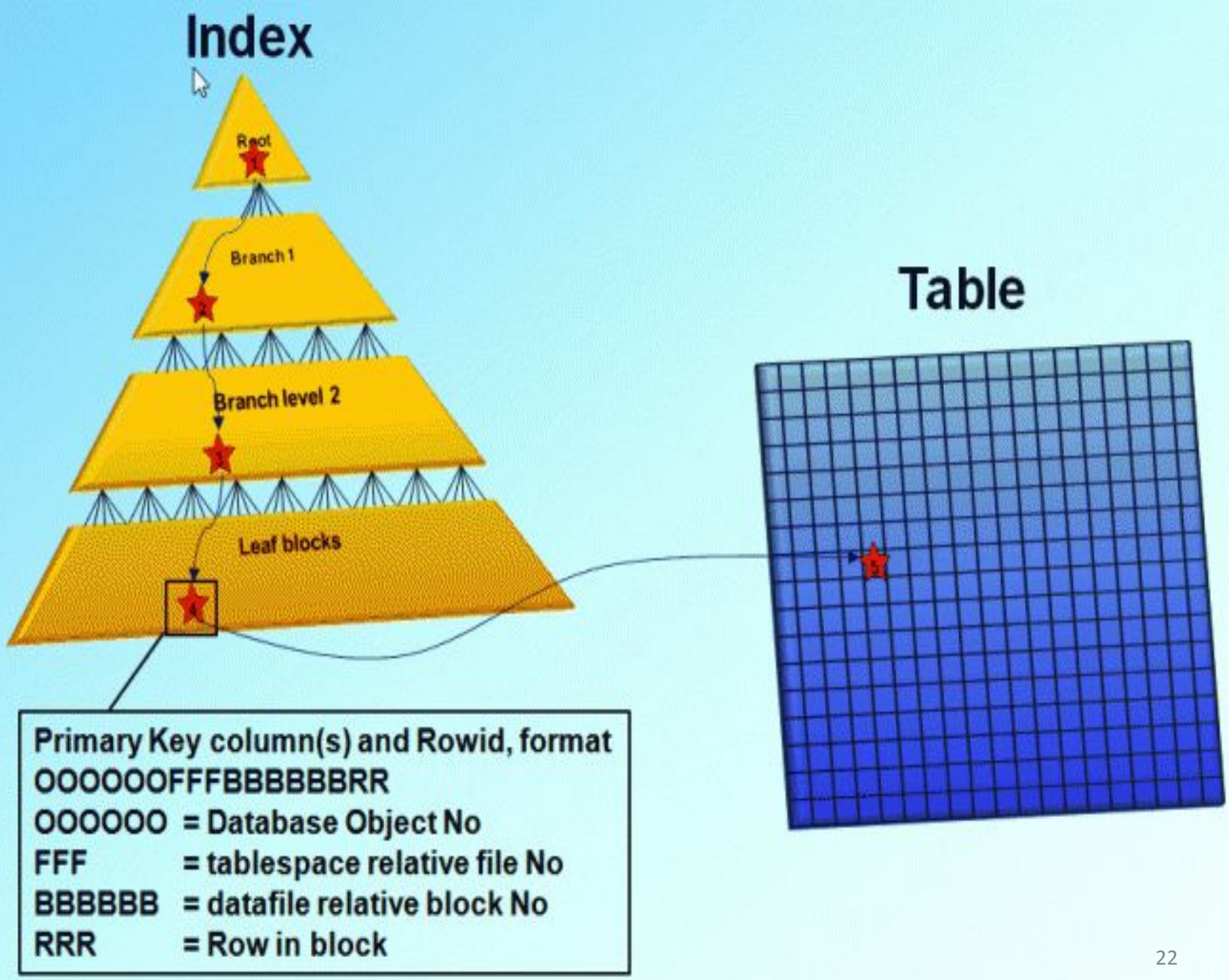
1. Index-organized tables
2. Reverse key indexes
3. Descending indexes
4. B-tree cluster indexes


Branch Blocks
0..40
41..80
81..120
....
200..250
0..10
11..19
20..25
....
32..40
41..48
49..53
54..65
....
78..80
200..209
210..220
221..228
....
246..250
Leaf Blocks
0,rowid
0,rowid
....
10,rowid
11,rowid
11,rowid
12,rowid
....
19,rowid
221,rowid
222,rowid
223,rowid
....
228,rowid
246,rowid
248,rowid
248,rowid
....
250,rowid

- A B-tree index is balanced because all leaf blocks automatically stay at the same depth.
- Thus, retrieval of any record from anywhere in the index takes approximately the same amount of time.
- The **height** of the index is the number of blocks required to go from the root block to a leaf block.
- The **branch level** is the height minus 1.
- In figure, the index has a height of 3 and a branch level of 2.

Id	Rowid
1	AAAKE3AAEAAAAcbAAA
2	AAAKE3AAEAAAAcbAAB
3	AAAKE3AAEAAAAcbAAC
4	AAAKE3AAEAAAAcbAAD
5	AAAKE3AAEAAAAcbAAE
6	AAAKE3AAEAAAAcbAAF
7	AAAKE3AAEAAAAcbAAG




Index
Root
Branch 1
Branch level 2
Leaf blocks
Table
Primary Key column(s) and Rowid, format
OOOOOOFFFBBBBBBRR
OOOOOO = Database Object No
FFF = tablespace relative file No
BBBBBB = datafile relative block No
RRR = Row in block

Creating Indexes (B-Tree) composite key

If two columns are often used together in WHERE conditions then create a composite index on these columns.

For example, suppose we use EMPNO and DEPTNO often together in WHERE condition.

Then create a composite index on these column as given below

Example:

```
create index empdept_ind on emp (empno,deptno);
```

- Multiple indexes can exist for the same table if the permutation of columns differs for each index
- You can create multiple indexes using the same columns if you specify distinctly different permutations of the columns.
- For example, the following SQL statements specify valid permutations:

```
CREATE INDEX employee_idx1 ON employees (last_name,
job_id);

CREATE INDEX employee_idx2 ON employees (job_id,
last_name);
```

Unique and Nonunique Indexes

- Indexes can be **unique** or nonunique.
- Unique indexes guarantee that no two rows of a table have duplicate values in the key column or columns.
- For example, no two employees can have the same employee ID.
- Nonunique indexes permit duplicates values in the indexed column or columns.
- For example, the first_name column of the employees table may contain same values.

Purpose of BITMAP INDEX

- Expedite the performance of queries.
- BITMAP Index are different from B-TREE INDEX in the manner they store data
- BTREE stores data in Tree Format
- While BITMAP index are stored as Dimensional arrays
- Btree Indexes are created on High cardinality columns
- BITMAP Indexes are created on low cardinality columns

id	Gender	Status
1	Male	Single
2	Female	Divorced
3	Male	Married
4	Female	Divorced
5	Male	Married
6	Female	Single
7	Female	Single

Male	Female
1	0
0	1
1	0
0	1
1	0
0	1
0	1
1	0
1	0

Single	Married	Divorced
1	0	0
0	0	1
0	1	0
0	0	1
0	1	0
1	0	0
1	0	0
0	1	0
0	1	0

BITMAP JOIN INDEX

- While BITMAP index is for a single table
- BITMAP join index can be created on join of two or more tables
- Space efficient way for reducing the volume of data that is to be joined
- In DWH environment the join is an equi join between Primary key of a dimension table and a foreign key of a fact table

Tran_id	Cust_id	Prod_id	amount	profit
1	1	1	100	10
2	2	4	900	100
3	2	2	800	200
4	1	3	7900	1000
5	4	5	780	200

Cust_id	Name	Gender
1	Vivek	M
2	Sonam	F
3	Rashi	F
4	Rohan	M

```
Select cust_id,sum(amount) from transcation t,
customer c where c.cust_id=t.cust_id and c.gender='M'
```

```
CREATE BITMAP INDEX TRAN_CUST_GEND_IDX
ON TRANSACTION(customer. gender) FROM from transcation
t, customer c where c.cust_id=t.cust_id ;
```

TRAN_ROW_ID	MALE	FEMALE
1	1	0
2	0	1
3	0	1
4	1	0
5	1	0

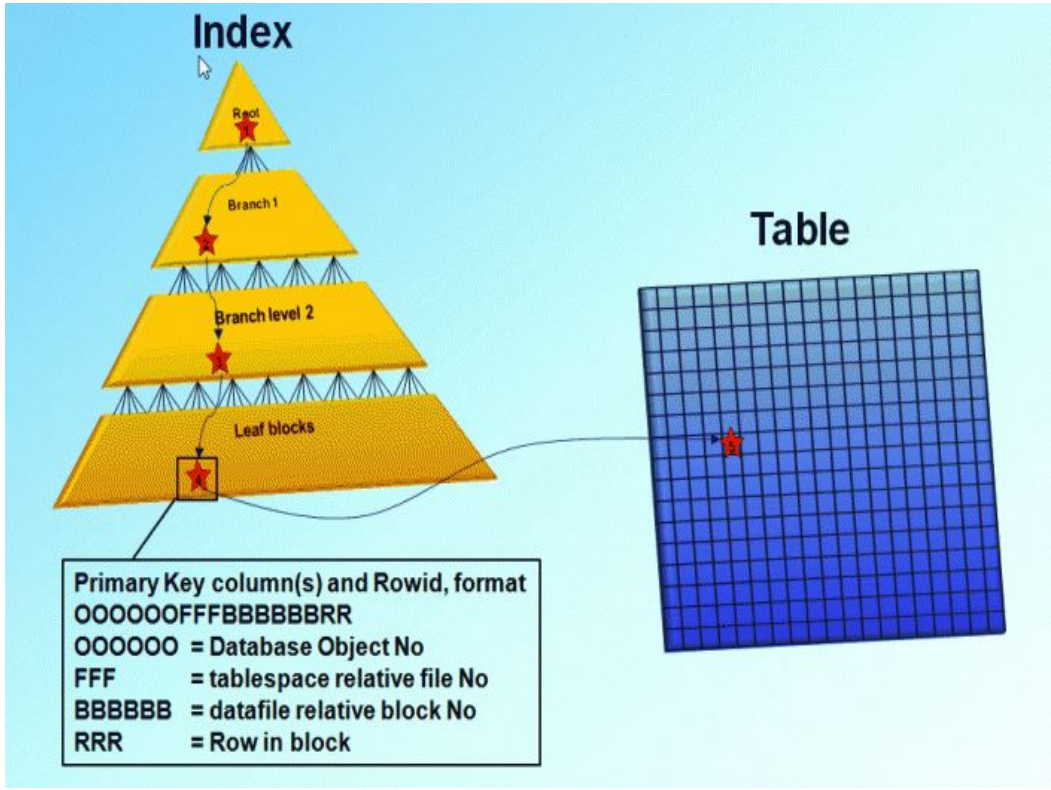
For detail Description of indexing referred this link of oracle document

- https://www.oracletutorial.com/oracle-index/

Index Scan Method(refer index_scan_method.doc)

- It is a method to retrieve rows traversing through index

1. Index unique Scan
2. Index Range Scan
3. Full index Scan
4. Fast full index Scan
5. Index slip Scan



Index Skip Scan

- An **index skip scan** uses logical subindexes of a composite index.
- The database "skips" through a single index as if it were searching separate indexes.
- Skip scanning is beneficial if there are few distinct values in the leading column of a composite index and many distinct values in the nonleading key of the index.
- The database may choose an index skip scan when the leading column of the composite index is not specified in a query predicate.

Index Skip Scan

SELECT * FROM sh.customers WHERE cust_email =
'Abbey@company.com';

a composite index exists on the columns
(cust_gender, cust_email)

- The customers table has a column cust_gender whose values are either M or F.

F,Wolf@company.com,rowid

F,Wolsey@company.com,rowid

F,Wood@company.com,rowid

F,Woodman@company.com,rowid

F,Yang@company.com,rowid

F,Zimmerman@company.com,rowid

M,Abbassi@company.com,rowid

M,Abbey@company.com,rowid

- The database logically splits the index into one subindex with the key F and a second subindex with the key M.
- When searching for the record for the customer whose email is Abbey@company.com, the database searches the subindex with the value F first and then searches the subindex with the value M.

- Conceptually, the database processes the query as follows:

```
SELECT * FROM sh.customers WHERE cust_gender = 'F'
  AND cust_email = 'Abbey@company.com'
UNION ALL
SELECT * FROM sh.customers WHERE cust_gender =
  'M'
  AND cust_email = 'Abbey@company.com';
```

Ascending and Descending Indexes

- In an ascending index, Oracle Database stores data in ascending order.
- By default,

✔ character data is ordered by the binary values contained in each byte of the value,

✔ numeric data from smallest to largest number, and

✔ date from earliest to latest value.

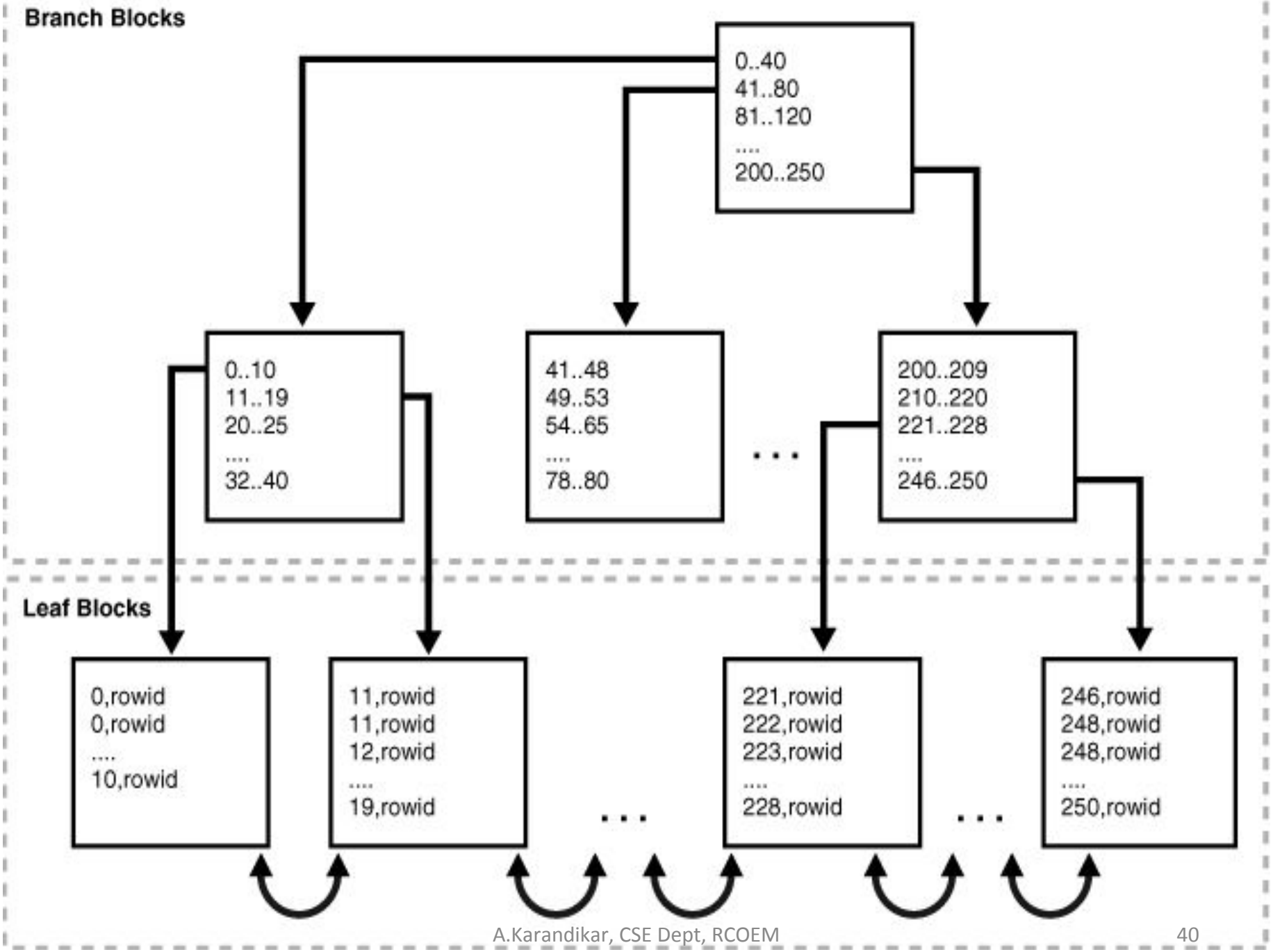
Ascending and Descending Indexes

- For an example of an ascending index, consider the following SQL statement:

CREATE INDEX emp_deptid_ix ON hr.employees(department_id);

- Oracle Database sorts the hr.employees table on the department_id column. It loads the ascending index with the department_id and corresponding rowid values in ascending order, starting with 0. When it uses the index, Oracle Database searches the sorted department_id values and uses the associated rowids to locate rows having the requested department_id value.

- By specifying the **DESC** keyword in the CREATE INDEX statement, you can create a descending index. In this case, the index stores data on a specified column or columns in descending order.
- Descending indexes are useful when a query sorts some columns ascending and others descending.
- The default search through a descending index is from highest to lowest value.
- If the index in Figure on the employees.department_id column were descending, then the leaf blocking containing 250 would be on the left side of the tree and block with 0 on the right.


Branch Blocks
0..40
41..80
81..120
....
200..250
0..10
11..19
20..25
....
32..40
41..48
49..53
54..65
....
78..80
200..209
210..220
221..228
....
246..250
Leaf Blocks
0,rowid
0,rowid
....
10,rowid
11,rowid
11,rowid
12,rowid
....
19,rowid
221,rowid
222,rowid
223,rowid
....
228,rowid
246,rowid
248,rowid
248,rowid
....
250,rowid

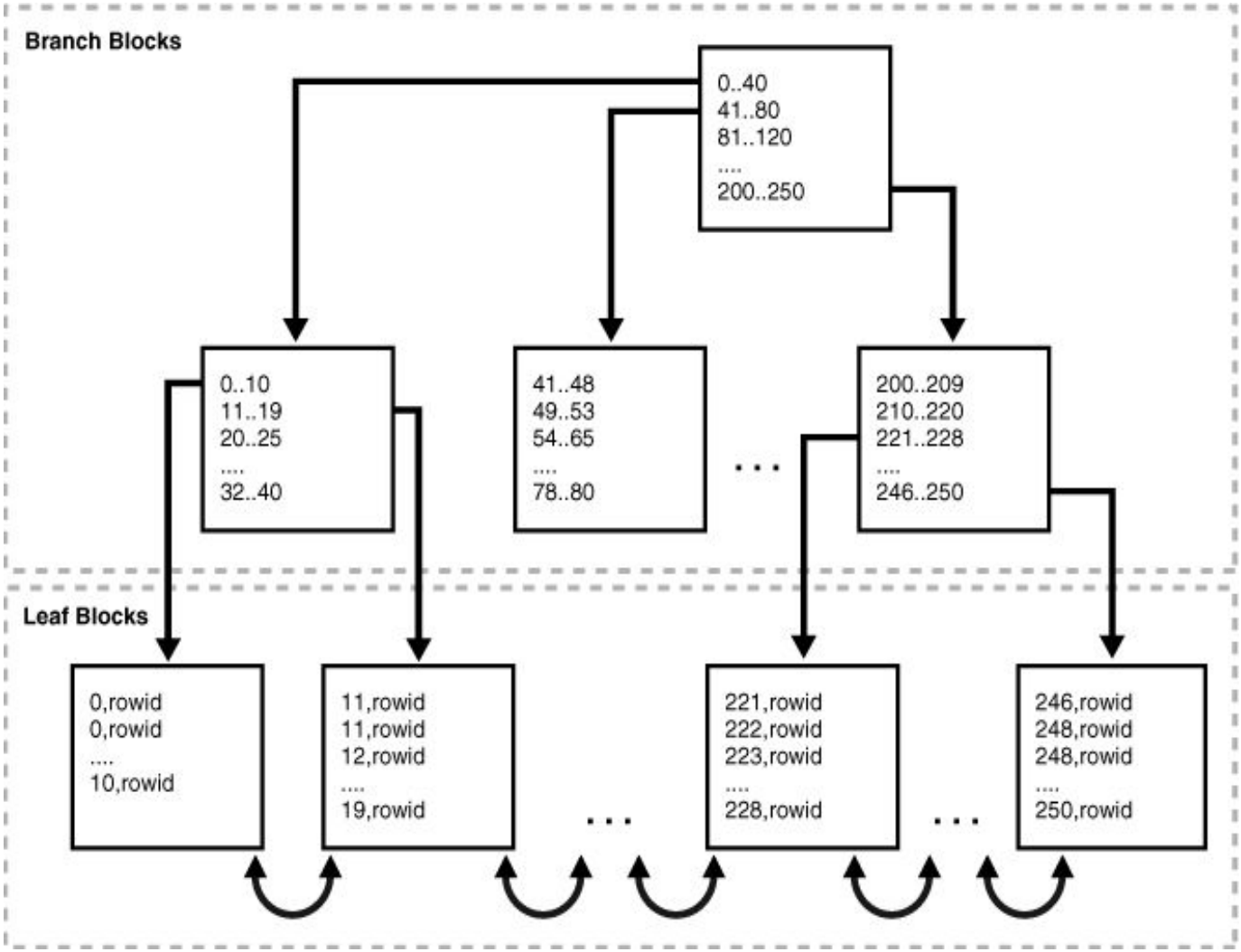
- If a user queries hr.employees for last names in ascending order (A to Z) and department IDs in descending order (high to low),

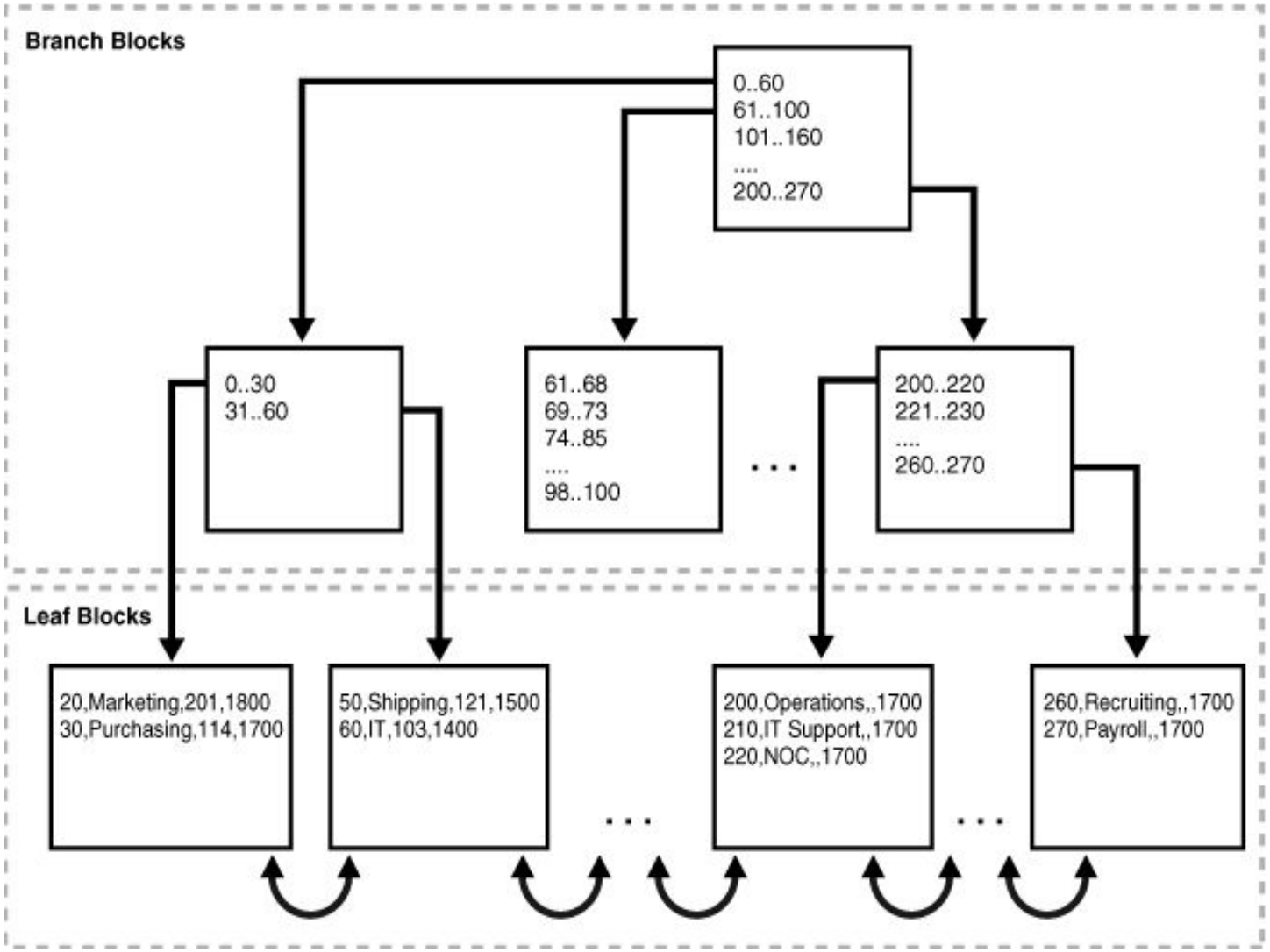
CREATE INDEX emp_name_dpt_ix ON hr.employees(last_name ASC, department_id DESC);

- avoid the extra step of sorting it.

Index Organized Tables(IOT)

- An index-organized table is a table stored in a **variation** of a B-tree index structure.
- In a heap-organized table, rows are inserted where they fit.
- In an index-organized table, rows are stored in an index defined on the primary key for the table.


Branch Blocks
0..40
41..80
81..120
....
200..250
0..10
11..19
20..25
....
32..40
41..48
49..53
54..65
....
78..80
200..209
210..220
221..228
....
246..250
Leaf Blocks
0,rowid
0,rowid
....
10,rowid
11,rowid
11,rowid
12,rowid
....
19,rowid
221,rowid
222,rowid
223,rowid
....
228,rowid
246,rowid
248,rowid
248,rowid
....
250,rowid


Branch Blocks
0..60
61..100
101..160
....
200..270
0..30
31..60
61..68
69..73
74..85
....
98..100
200..220
221..230
....
260..270
Leaf Blocks
20,Marketing,201,1800
30,Purchasing,114,1700
50,Shipping,121,1500
60,IT,103,1400
200,Operations,,1700
210,IT Support,,1700
220,NOC,,1700
260,Recruiting,,1700
270,Payroll,,1700

- Index-organized tables provide faster access to table rows by primary key
- The presence of non-key columns of a row in the leaf block avoids an **additional data block I/O**. For example, the salary of employee 100 is stored in the index row itself.
- Also, because rows are stored in primary key order, range access by the primary key or prefix involves minimal block I/Os.
- Another benefit is the avoidance of the **space overhead** of a separate primary key index.
- Index-organized tables are useful when related pieces of data must be stored together or data must be physically stored in a specific order.

Index Organised Tables
Heap tables versus IOTs

Heap-organised table

```
CREATE TABLE team
(
        team_key     VARCHAR2(3),
        team_name  VARCHAR2(50),
        country_key VARCHAR2(3)
        CONSTRAINT team_pk
        PRIMARY KEY (team_key);
)
ORGANIZATION HEAP;
```

```
SELECT team_name
FROM team
WHERE team_key = 'FER';
```

```
SELECT STATEMENT
        TABLE ACCESS (BY INDEX ROWID) OF
'TEAM'
                INDEX (UNIQUE SCAN) OF
'TEAM_PK'
```

Index-organised table

```
CREATE TABLE team
(
        team_key     VARCHAR2(3),
        team_name  VARCHAR2(50),
        country_key VARCHAR2(3)
        CONSTRAINT team_pk
        PRIMARY KEY (team_key);
)
ORGANIZATION INDEX;
```

```
SELECT team_name
FROM team
WHERE team_key = 'FER';
```

```
SELECT STATEMENT
        INDEX (UNIQUE SCAN) OF 'TEAM_PK'
```

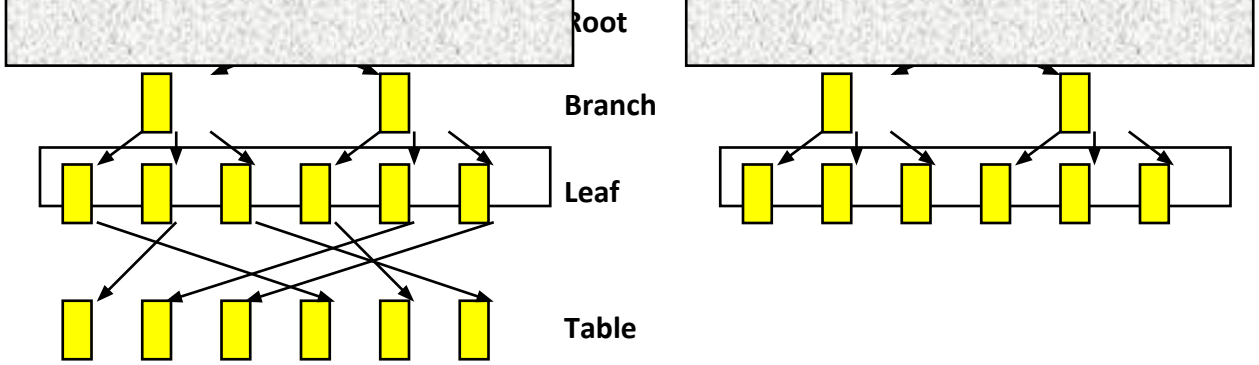
Heap tables versus IOTs

Heap-organised table

```
CREATE TABLE team
(
        team_key     VARCHAR2(3),
        team_name  VARCHAR2(50),
        country_key VARCHAR2(3)
        CONSTRAINT team_pk
        PRIMARY KEY (team_key);
)
ORGANIZATION HEAP;
```

Index-organised table

```
CREATE TABLE team
(
        team_key     VARCHAR2(3),
        team_name  VARCHAR2(50),
        country_key VARCHAR2(3)
        CONSTRAINT team_pk
        PRIMARY KEY (team_key);
)
ORGANIZATION INDEX;
```



Heap-Organized Table	Index-Organized Table
The **rowid** uniquely identifies a row. Primary key constraint may optionally be defined.	Primary key uniquely identifies a row. Primary key constraint must be defined.
Physical rowid in `ROWID` **pseudocolumn** allows building secondary indexes.	Logical rowid in `ROWID` pseudocolumn allows building secondary indexes.
Individual rows may be accessed directly by rowid.	Access to individual rows may be achieved indirectly by primary key.
Sequential **full table scan** returns all rows in some order.	A **full index scan** or fast full index scan returns all rows in some order.

- An index-organized table stores all data in the same structure and does not need to store the rowid.

leaf block 1

20,Marketing,201,1800

30,Purchasing,114,1700

Leaf block 2

50,Shipping,121,1500

60,IT,103,1400

- A scan of the index-organized table rows in primary key order reads the blocks in the following sequence:

  Block 1

  Block 2

- To contrast data access in a heap-organized table to an index-organized table, suppose

block 1 of a heap-organized departments table contains rows as follows:

50,Shipping,121,1500

20,Marketing,201,1800

Block 2 contains rows for the same table as follows:

30,Purchasing,114,1700

60,IT,103,1400

- A B-tree index leaf block for this heap-organized table contains the following entries, where the first value is the primary key and the second is the rowid:

  20,AAAPeXAAFAAAAAyAAD
  30,AAAPeXAAFAAAAAyAAA
  50,AAAPeXAAFAAAAAyAAC
  60,AAAPeXAAFAAAAAyAAB

- A scan of the table rows in primary key order reads the table segment blocks in the following sequence:

  Block 1

  Block 2

  Block 1

  Block 2

- Thus, the number of block I/Os in this example is double the number in the index-organized example.

```
CREATE TABLE countries_demo
   ( country_id      CHAR(2)
     CONSTRAINT country_id_nn_demo NOT NULL
   , country_name    VARCHAR2(40)
   , currency_name   VARCHAR2(25)
   , currency_symbol VARCHAR2(3)
   , region          VARCHAR2(15)
   , CONSTRAINT    country_c_id_pk_demo
                   PRIMARY KEY (country_id ) )
   ORGANIZATION INDEX
```

Index-Organized Tables with Row Overflow Area

- When creating an index-organized table, you can specify a separate segment as a row overflow area.
- In index-organized tables, B-tree index entries can be large because they contain an entire row, so a separate segment to contain the entries is useful.
- In contrast, B-tree entries are usually small because they consist of the key and rowid.

- If a row overflow area is specified, then the database can divide a row in an index-organized table into the following parts:

✔ The index entry

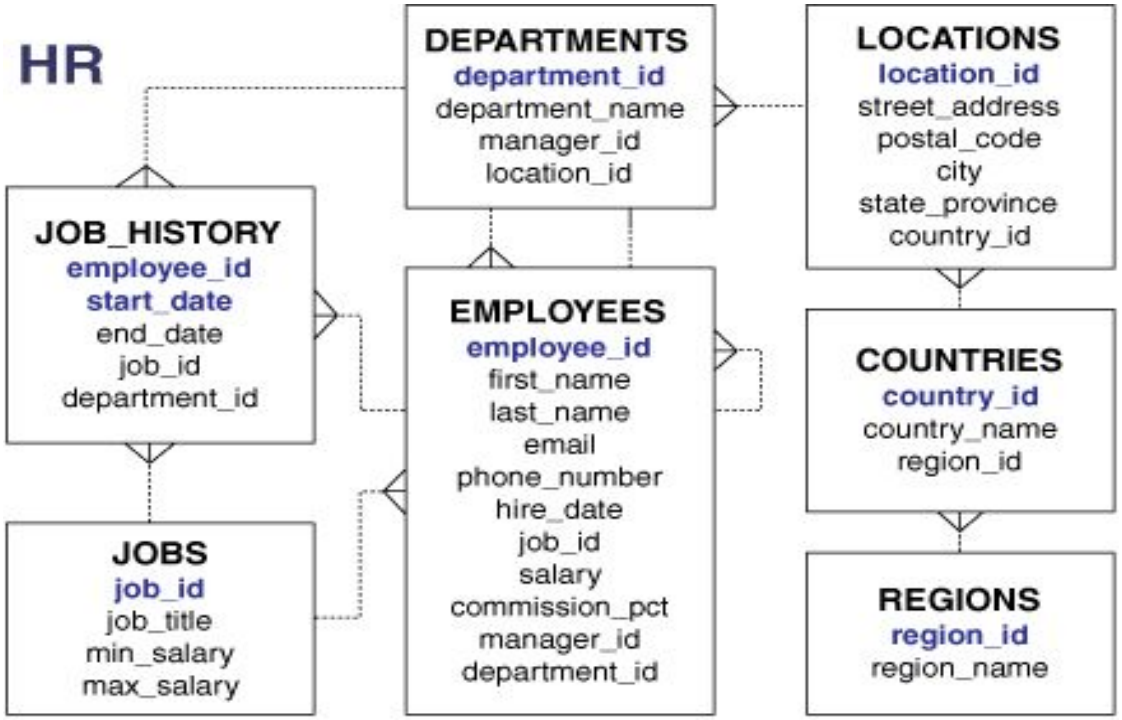
This part contains column values for all the primary key columns, a physical rowid that points to the overflow part of the row, and optionally a few of the non-key columns. This part is stored in the index segment.

✔ The overflow part

This part contains column values for the remaining non-key columns. This part is stored in the overflow storage area segment.

```
CREATE TABLE countries_demo
   ( country_id      CHAR(2)
     CONSTRAINT country_id_nn_demo NOT NULL
   , country_name    VARCHAR2(40)
   , currency_name   VARCHAR2(25)
   , currency_symbol VARCHAR2(3)
   , region          VARCHAR2(15)
   , CONSTRAINT    country_c_id_pk_demo
                   PRIMARY KEY (country_id ) )
   ORGANIZATION INDEX
   INCLUDING   country_name
```

HR Schema


HR
DEPARTMENTS
department_id
department_name
manager_id
location_id
LOCATIONS
location_id
street_address
postal_code
city
state_province
country_id
JOB_HISTORY
employee_id
start_date
end_date
job_id
department_id
EMPLOYEES
employee_id
first_name
last_name
email
phone_number
hire_date
job_id
salary
commission_pct
manager_id
department_id
COUNTRIES
country_id
country_name
region_id
JOBS
job_id
job_title
min_salary
max_salary
REGIONS
region_id
region_name

Secondary Indexes on Index-Organized Tables

- A secondary index is an index on an index-organized table.
- In a sense, it is an index on an index.
- The secondary index is an independent schema object and is stored separately from the index-organized table.

- Rows in index leaf blocks can move within or between blocks because of insertions.
- Rows in index-organized tables do not migrate as heap-organized rows do.
- Because rows in index-organized tables do not have permanent physical addresses, the database uses logical rowids based on primary key.
- A logical rowid is a base64-encoded representation of the table primary key. The logical rowid length depends on the primary key length.

- For example, assume that the departments table is index-organized. The location_id column stores the ID of each department. The table stores rows as follows, with the last value as the location ID:

  10,Administration,200,1700
  20,Marketing,201,1800
  30,Purchasing,114,1700

  40,Human Resources,203,2400

- A secondary index on the location_id column might have index entries as follows, where the value following the comma is the logical rowid:

  1700,*BAFAJqoCwR/+

  1700,*BAFAJqoCwQv+

  1800,*BAFAJqoCwRX+

  2400,*BAFAJqoCwSn+

- Secondary indexes provide fast and efficient access to index-organized tables using columns that are neither the primary key nor a prefix of the primary key.
- For example, a query of the names of departments whose ID is greater than 1700 could use the secondary index to speed data access.

Data dictionary views on Indexes

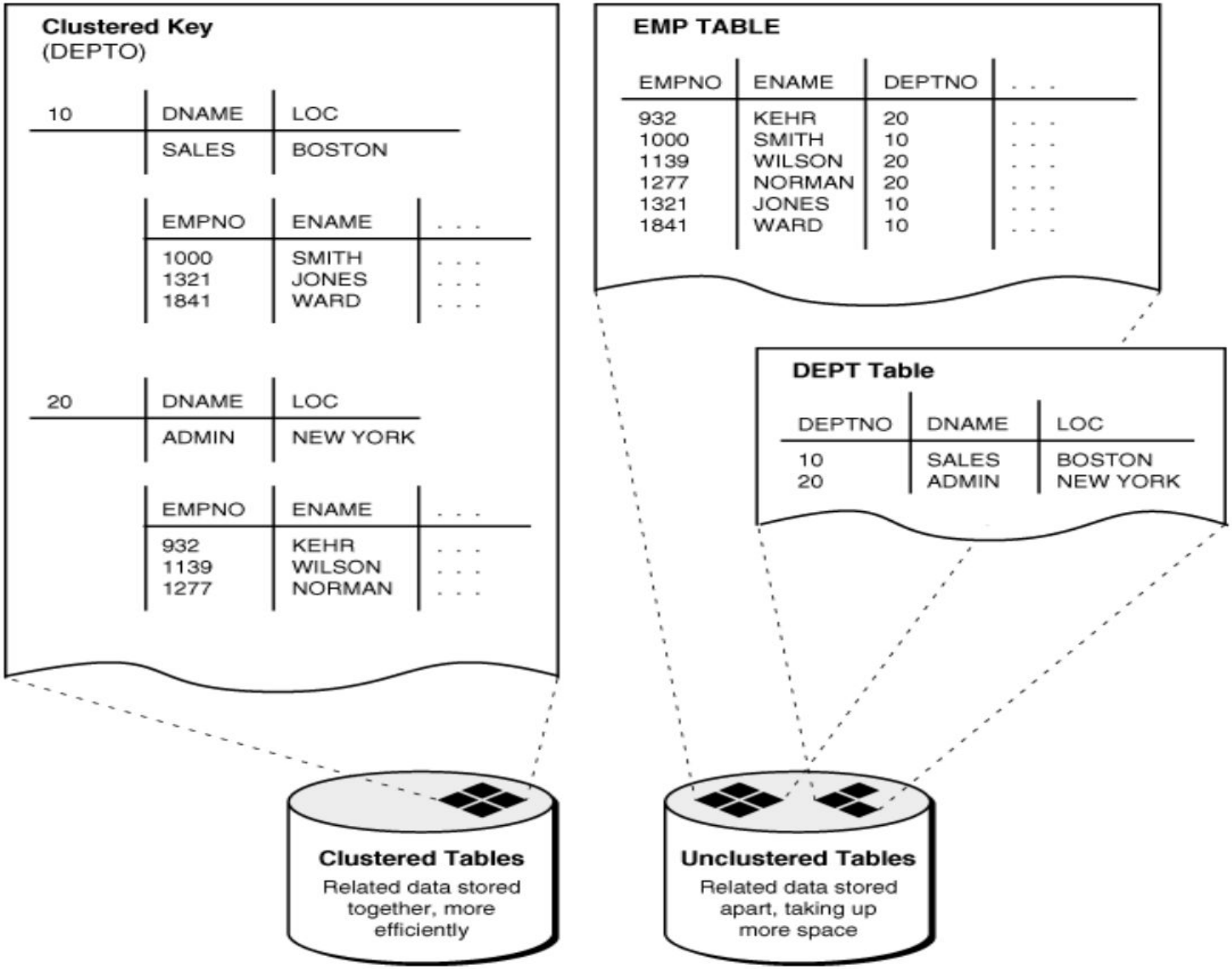
DBA_INDEXES ALL_INDEXES USER_INDEXES	DBA view describes indexes on all tables in the database. ALL view describes indexes on all tables accessible to the user. USER view is restricted to indexes owned by the user. Some columns in these views contain statistics that are generated by the DBMS_STATS package or ANALYZE statement.
DBA_IND_COLUMNS ALL_IND_COLUMNS USER_IND_COLUMNS	These views describe the columns of indexes on tables. Some columns in these views contain statistics that are generated by the DBMS_STATS package or ANALYZE statement.

Cluster

- A **cluster** provides an optional method of storing table data.
- A cluster is made up of a group of tables that share the same data blocks.
- The tables are grouped together because they share common columns and are often used together.

For example

- The **emp** and **dept** table share the deptno column.
- When you cluster the emp and dept tables Oracle Database physically stores all rows for each department from both the emp and dept tables in the same data blocks.


Clustered Key
(DEPTO)
10
DNAME
LOC
SALES
BOSTON
EMPNO
ENAME
. . .
1000
SMITH
. . .
1321
JONES
. . .
1841
WARD
. . .
20
DNAME
LOC
ADMIN
NEW YORK
EMPNO
ENAME
. . .
932
KEHR
. . .
1139
WILSON
. . .
1277
NORMAN
. . .
EMP TABLE
EMPNO
ENAME
DEPTNO
. . .
932
KEHR
20
. . .
1000
SMITH
10
. . .
1139
WILSON
20
. . .
1277
NORMAN
20
. . .
1321
JONES
10
. . .
1841
WARD
10
. . .
DEPT Table
DEPTNO
DNAME
LOC
10
SALES
BOSTON
20
ADMIN
NEW YORK
Clustered Tables
Related data stored together, more efficiently
Unclustered Tables
Related data stored apart, taking up more space

Advantage

- Disk I/O is reduced and access time improves for joins of clustered tables
- The **cluster key** is the column, or group of columns, that the clustered tables have in common.
- You specify the columns of the cluster key when creating the cluster.
- You subsequently specify the same columns when creating every table added to the cluster.
- Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value. So less place is required

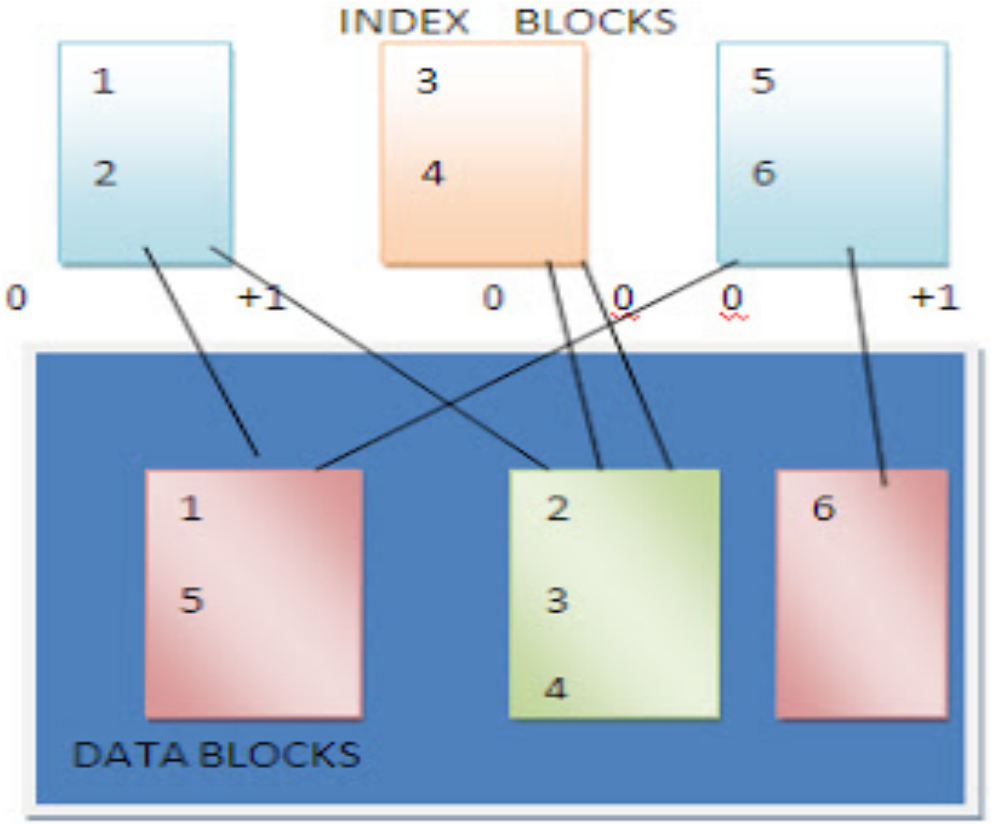
Index Clustering Factor

- The clustering factor is a number which represent the degree to which data is randomly distributed in a table.

  In simple terms it is the number of "block switches" while reading a table using an index.

- The Index clustering factor is a number which represents the degree to which the data in table is synchronized with the entries in the index.
- It gives a rough measure of how many I/Os the database would perform if it were to read every row in that table via the index in index order.
- If the rows of a table on disk are sorted in the same order as the index keys, the database will perform a minimum number of I/Os on the table to read the entire table via the index.

Clustering Factor



Data Block 1				
100	Steven	**King**	SKING	...
156	Janette	**King**	JKING	...
115	Alexander	**Khoo**	AKHOO	...
.				
.				
.				
116	Shelli	**Baida**	SBAIDA	...
204	Hermann	**Baer**	HBAER	...
105	David	**Austin**	DAUSTIN	...
130	Mozhe	**Atkinson**	MATKINSO	...
166	Sundar	**Ande**	SANDE	...
174	Ellen	**Abel**	EABEL	

Data Block 2				
149	Eleni	**Zlotkey**	EZLOTKEY	...
200	Jennifer	**Whalen**	JWHALEN	...
.				
.				
.				
137	Renske	**Ladwig**	RLADWIG	...
173	Sundita	**Kumar**	SKUMAR	...
101	Neena	**Kochar**	NKOCHHAR	...

- Rows are stored in the blocks in order of last name (shown in bold).
- Assume that an index exists on the last name column. Each name entry corresponds to a rowid.
- Conceptually, the index entries would look as follows:

Abel,block1row1

Ande,block1row2

Atkinson,block1row3

Austin,block1row4

Baer,block1row5

. . .

- Assume that a separate index exists on the employee ID column.
- Conceptually, the index entries might look as follows, with employee IDs distributed in almost random locations throughout the two blocks:

100,block1row50

101,block2row1

102,block1row9

103,block2row19

104,block2row39

105,block1row4

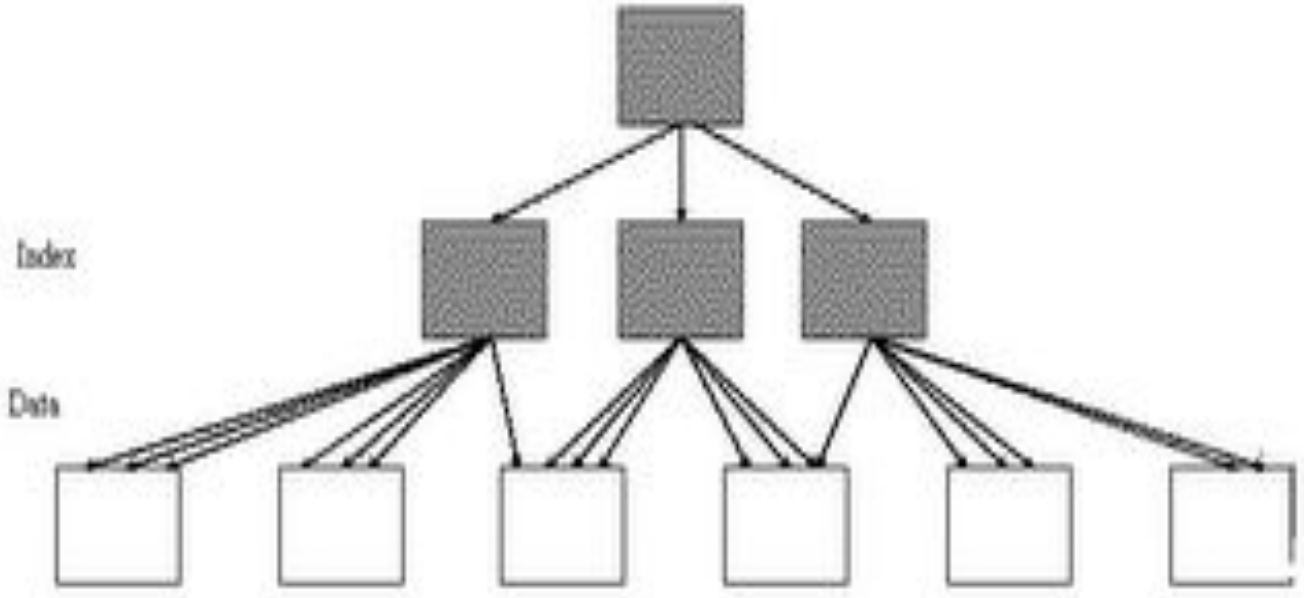
. . .

- To calculate the clustering factor of an index during the gathering of index statistics, Oracle does the following :
  For each entry in the index

  (

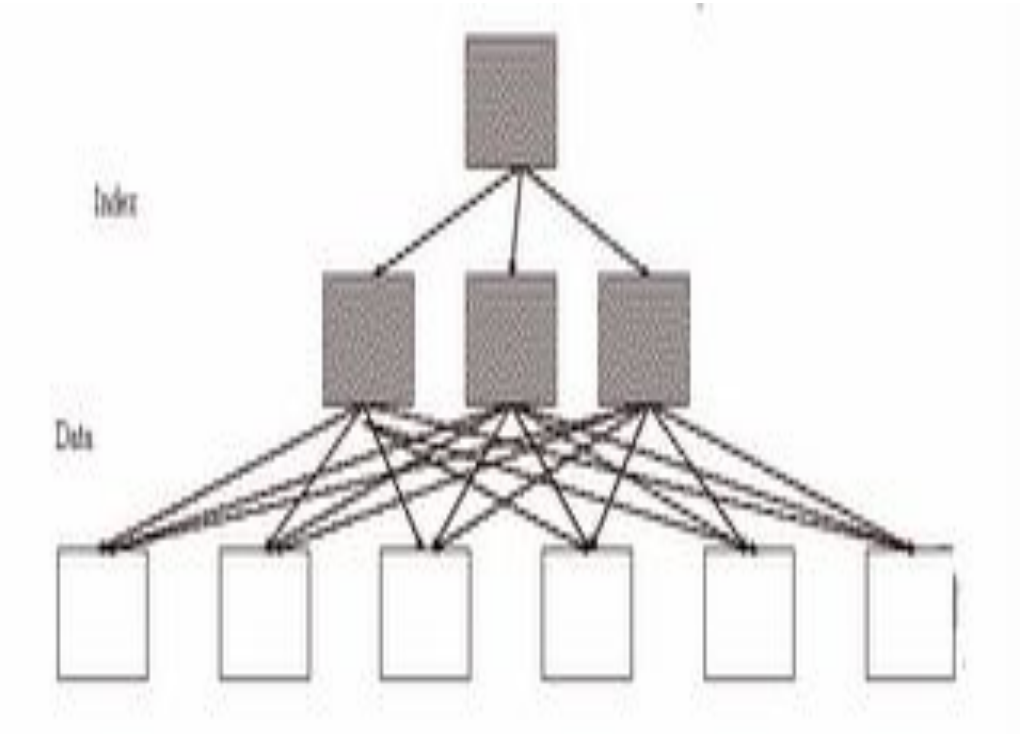
  Compare the entry's table rowid block with the block of the previous index entry.
  If the block is different, Oracle increments the clustering factor by 1.
)
- The minimum possible clustering factor is equal to the number of distinct blocks identified through the index's list of rowid's.
- The maximum clustering factor is the number of entries in the index i.e. each rowid points to a different block in the table.

Good CF



- Clustering factor nearly equal to no. of blocks

Bad CF



- Clustering_factor nearly equal to no. of rows

```
SQL> SELECT INDEX_NAME,
  CLUSTERING_FACTOR
  FROM ALL_INDEXES
  WHERE INDEX_NAME IN
  ('EMP_NAME_IX','EMP_EMP_ID_PK');

INDEX_NAME          CLUSTERING_FACTOR
------------------- -----------------
EMP_EMP_ID_PK                    19
EMP_NAME_IX                      2
```

decision to use an index vs. a full-table scan

```
Select    customer_name
from    customer
where customer_state = ‘New Mexico’;
```

An index scan is faster for this query if the percentage of customers in New Mexico is small and the values are clustered on the data blocks.

Four factors work together to help the CBO decide whether to use an index or a full-table scan:

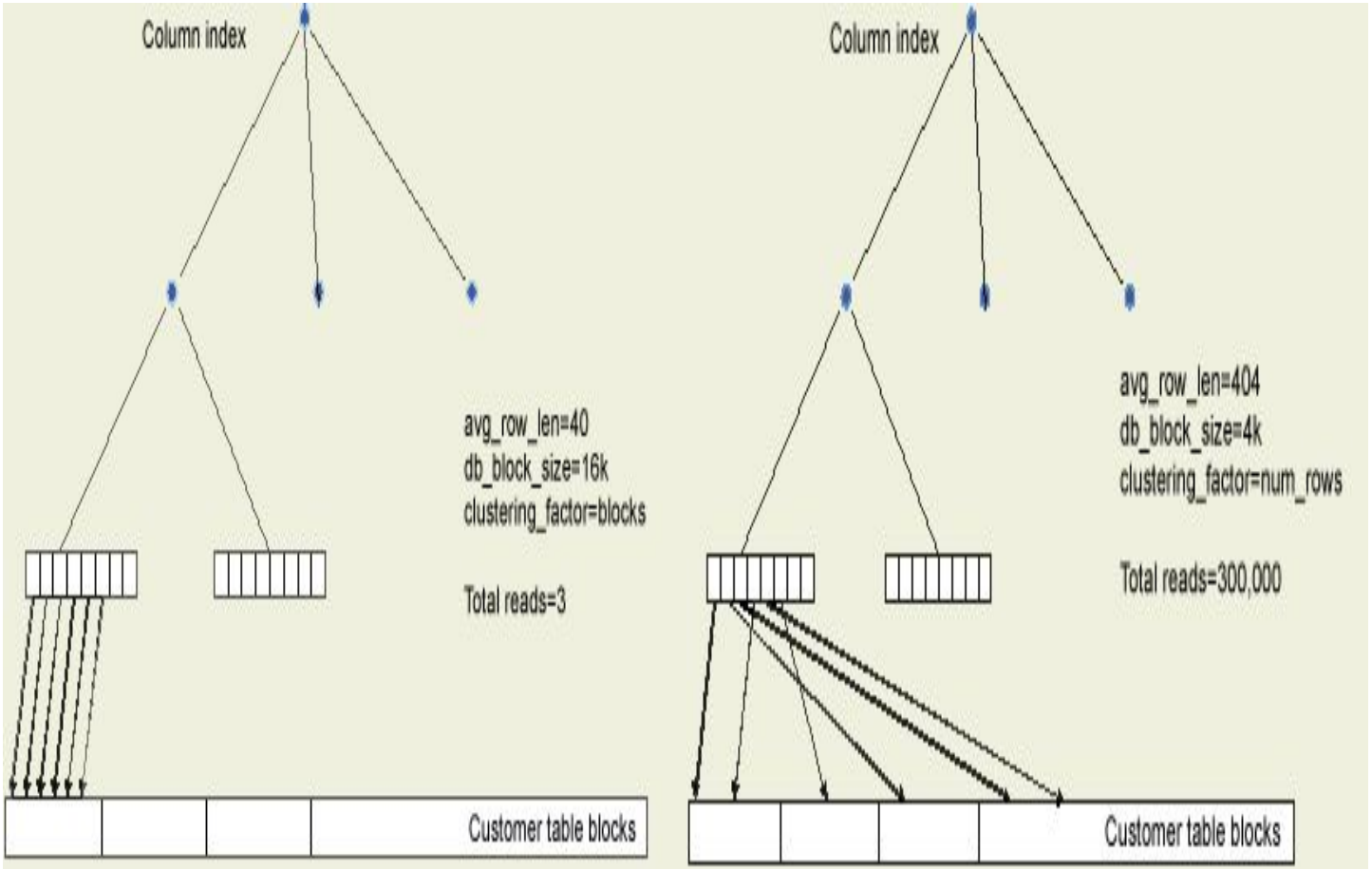
✔ the selectivity of a column value,

✔ the *db_block_size*,

✔ the *avg_row_len*,

✔ the cardinality.

- An index scan is usually faster if a data column has high selectivity and a low *clustering_factor.*


Column index
avg_row_len=40
db_block_size=16k
clustering_factor=blocks
Total reads=3
Customer table blocks
Column index
avg_row_len=404
db_block_size=4k
clustering_factor=num_rows
Total reads=300,000
Customer table blocks

Try It !!!

SQL> create table sac as select * from all_objects;

Table created.

SQL> create index obj_id_indx on sac(object_id);

Index created.

```
SQL> select clustering_factor from user_indexes
where index_name='OBJ_ID_INDX';

CLUSTERING_FACTOR
-----------------
545

SQL> select count(*) from sac;

COUNT(*)
----------
38956

SQL> select blocks from user_segments where
segment_name='OBJ_ID_INDX';

BLOCKS
----------
96
```

- The above example shows that index has to jump 545 times to give you the full data had you performed full table scan using the index.

Note:
- A good CF is equal (or near) to the values of number of blocks of table.

- A bad CF is equal (or near) to the number of rows of table.

Is it true???

- Rebuilding of index can improve the CF.

Then how to improve the CF?

- To improve the CF, it's the table that must be rebuilt (and reordered).
- If table has multiple indexes, careful consideration needs to be given by which index to order table.

Example

- ORDERS table which grows every day.
- index on the order date and another one on the customer id.
- Because orders don’t get deleted there are no holes in the table so that each new order is added to the end. The table grows chronologically.

Example (contd..)

- Which index will have a good CF and which will have a bad CF??
- WHY???

- An index on the **order date** has a very **low** clustering factor because the index order is essentially the same as the table order.
- The index on **customer id** has a **higher** clustering factor because the index order is different from the table order; the table row will be inserted at the end of the table, the corresponding index entry somewhere in the middle of the index—according to the customer id.