How Indexes Work

Added Material

Indexes

- ***Index***: a data structure that improves the speed of locating records that are to be retrieved, updated, or deleted
- Assist performance of primary key and unique constraints because they support a fast and efficient way of determining if the new record would be a duplicate.
- Without indexes, the DBMS must read through all the rows in order to find the rows required by an SQL statement (a process known as a *full table scan*).

Index Challenges

- Indexes require storage.
  - However, storage costs have been decline for years.
- Indexes must be maintained.
  - Most indexes are maintained automatically by the DBMS.
    - However, NULL values are not indexed.
  - INSERTs require index record creation
  - DELETEs require index record removal
  - UPDATEs of indexed columns require index record updates

B-tree Indexes

- *B-tree (balanced tree) indexes*:
  - Data structures with an inverted tree structure
    - *Root* node at the top of the structure
    - *Branch layer* nodes in the middle of the structure
    - *Leaf layer* nodes at the bottom of the structure,
      - One logical record per key value
      - Maintained in key sequence, allowing sequential processing without the need to sort the table data
    - Usually contain row ids (RIDs) as pointers to table rows
  - Nodes connected with RBA pointers
  - Automatically reorganized as structure expands/contracts

Example: Table Content

ROWID	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT
1	173	Sundita	Kumar	6100	0.1
2	172	Elizabeth	Bates	7300	0.15
3	171	William	Smith	7400	0.15
4	170	Tayler	Fox	9600	0.2
5	169	Harrison	Bloom	10000	0.2
6	168	Lisa	Ozer	11500	0.25
7	150	Peter	Tucker	10000	0.3
8	151	David	Bernstein	9500	0.25
9	149	Eleni	Zlotkey	10500	0.2
10	145	John	Russell	14000	0.4
11	147	Alberto	Errazuriz	12000	0.3
12	146	Karen	Partners	13500	0.3
13	152	Peter	Hall	9000	0.25
14	155	Oliver	Tuvault	7000	0.15
15	153	Christopher	Olsen	8000	0.2
16	148	Gerald	Cambrault	11000	0.3
17	154	Nanette	Cambrault	7500	0.2
18	156	Janette	King	10000	0.35
19	161	Sarath	Sewall	7000	0.25
20	160	Louise	Doran	7500	0.3
21	159	Lindsey	Smith	8000	0.3
22	158	Allan	McEwen	9000	0.35
23	157	Patrick	Sully	9500	0.35
24	162	Clara	Vishney	10500	0.25
25	163	Danielle	Greene	9500	0.15
26	164	Mattea	Marvins	7200	0.1
27	165	David	Lee	6800	0.1
28	166	Sundar	Ande	6400	0.1
29	167	Amit	Banda	6200	0.1
30	179	Charles	Johnson	6200	0.1
31	177	Jack	Livingston	8400	0.2
32	176	Jonathon	Taylor	8600	0.2
33	175	Alyssa	Hutton	8800	0.25
34	174	Ellen	Abel	11000	0.3

Diagram of B-tree Index

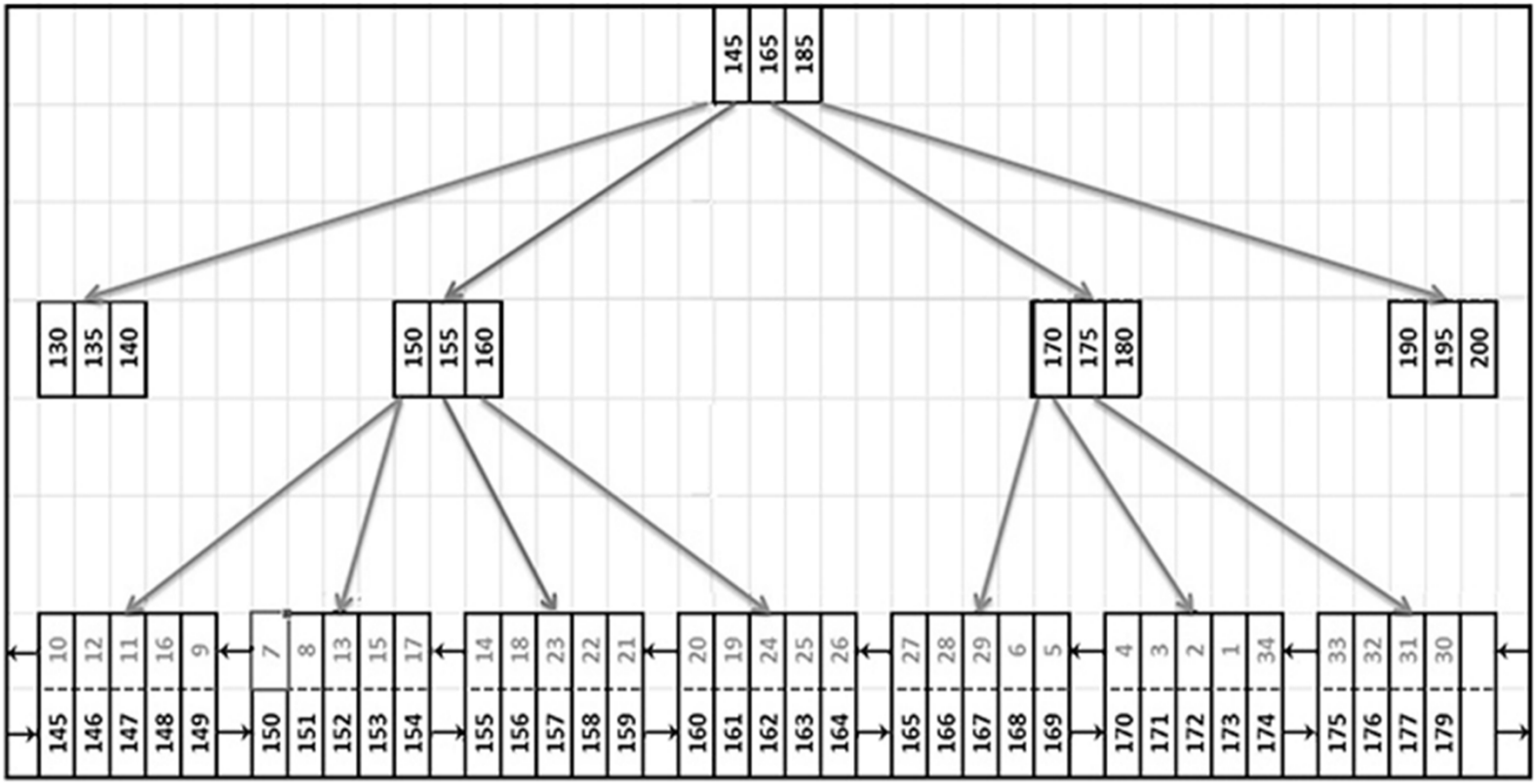

Diagram Explanation

- Top level (root), contains 3 values: 145, 165, and 185
  - These point to the branch layer (one level in this example), representing:
    - Less than 145
    - Greater than or equal to 145, but less than 165
    - Greater than or equal to 165, but less than 185
    - Greater than 185
  - Each branch layer node also contains 3 values

Searching the Index

- Search path for the row with key 156 (follow red arrows on the diagram):
  - In root node, 156 is between 145 and 165, so pointer followed to branch node with 150 | 155 | 160
  - In branch node, 156 is between 155 and 160, so pointer followed to leaf node holding keys for 155 through 159
  - Leaf node searched sequentially for matching value (156 is second value in the node)
  - Row ID 18 (value stored with key 156) is used to access the row in the table

Using Index for Skip Sequential Processing

- When processing a query like this one:

```
SELECT LAST_NAME
   FROM EMPLOYEE
  WHERE EMPLOYEE_ID BETWEEN 157 AND 160;
```

- Index used to find the first value (157) among the leaf nodes.
- Leaf node scanned sequentially to find the rest of the rows in the requested range.
  - Key 160 is in another leaf node, but leaf nodes are connected by pointers, so the RDBMS can keep scanning

Effective Use of B-tree Indexes

- Most effective when they are ***selective***
  - ***Index selectivity*** is measured as the ratio of key values to the number of rows in the table
  - Unique indexes have selectivity of 1.0 (the maximum ratio)
  - For b-tree indexes, selectivity of 0.7 or higher is considered good (but vendor recommendations vary somewhat)

Keep Index Downsides in Mind

- As tables are updated, indexes must be maintained
- For tables smaller than 1,000 rows or so, indexes other than those required to enforce primary key and unique constraints, are seldom useful to the DBMS
- For tables with short rows, consider an index-organized table (IOT), if supported by the vendor.
  - This option puts all the table column values in the index leaf nodes (instead of the RowID) and thus the entire table is built into the index

Bitmap Indexes

- *Bitmap indexes* are special indexes that use bit arrays (bit maps) to store index values
  - *Bit* (binary digit) can have only two values – 0 or 1
  - A *bit array* is merely a sting of bits
  - Bits are typically organized into bytes, each consisting of 8 bits
- Bitmap indexes within an RDBMS contain a string of bits to represent each possible value of an attribute, with each bit within a string representing a row in the base table
- Most appropriate for columns with low cardinality

Example: Table Contents

RowID	Region	Year/Qtr	Product Line	Transaction ID	Amount
1	South	2015-1	Systems	19574	$12,754
2	North	2015-2	Accessories	84737	$6,274
3	West	2015-4	Software	25141	$47,250
4	East	2015-3	Accessories	85743	$5,280
5	North	2016-1	Systems	64532	$187,205
6	North	2016-1	Parts	32767	$299
7	South	2015-2	Software	12843	$5,000
8	North	2015-1	Parts	48732	$598
9	West	2015-2	Systems	18749	$96,200
10	South	2015-3	Software	79453	$86,327
11	East	2015-1	Systems	63890	$275,900
12	North	2015-4	Accessories	48327	$2,785

Viable Columns for Indexing

- Columns with low cardinality (bit-map candidates):
  - Region,
  - Year/Qty
  - Product Line
- Columns with high cardinality (b-tree candidates):
  - Transaction ID (primary key)
  - Amount
    - However, it might be frequently updated
    - Could be divided into ranges (e.g. < 5K, 5K – 10K, 10K – 20K, etc.) that would potentially be low cardinality

Bit-map Indexes

REGION	TABLE ROW											
	1	2	3	4	5	6	7	8	9	10	11	12
North	0	1	0	0	1	1	0	1	0	0	0	1
South	1	0	0	0	0	0	1	0	0	1	0	0
East	0	0	0	1	0	0	0	0	0	0	1	0
West	0	0	1	0	0	0	0	0	1	0	0	0

YEAR/ QTR	TABLE ROW											
	1	2	3	4	5	6	7	8	9	10	11	12
2015-1	1	0	0	0	0	0	0	1	0	0	1	0
2015-2	0	1	0	0	0	0	1	0	1	0	0	0
2015-3	0	0	0	1	0	0	0	0	0	1	0	0
2015-4	0	0	1	0	0	0	0	0	0	0	0	1
2016-1	0	0	0	0	1	1	0	0	0	0	0	0

PRODUCT LINE	TABLE ROW											
	1	2	3	4	5	6	7	8	9	10	11	12
Accessories	0	1	0	1	0	0	0	0	0	0	0	1
Parts	0	0	0	0	0	1	0	1	0	0	0	0
Software	0	0	1	0	0	0	1	0	0	1	0	0
Systems	1	0	0	0	1	0	0	0	1	0	1	0

- One row per possible data value
- Each column:
  - Represents a table row
  - Contains a single ‘1’ bit

Bitmap Index Usability

- Benefits
  - Much more compact than b-tree indexes
  - Easily compared with one another using Boolean logic
- Challenges
  - Bitmap indexes are quite expensive to maintain as data values change in the base tables
    - Most implementations do not have updatable bitmap indexes – they must be rebuilt
    - Best used on read-only tables, such as in analytical databases, where indexes are rebuilt whenever tables are re-loaded