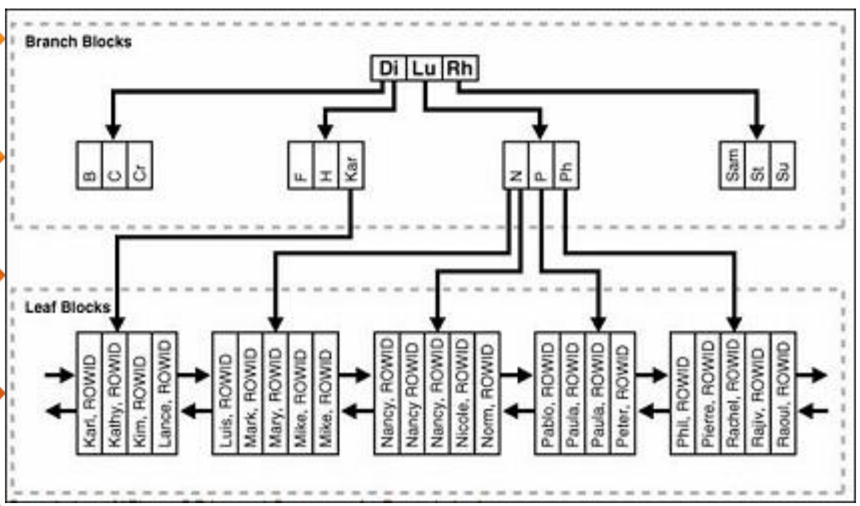
Access Paths

INFS602 Physical Database Design

Learning Outcomes

- Understand the role played by access paths in improving performance
- Understand the trade-offs between choosing *Indexing*, *Hashing* and *Clustering* in Physical Design
- Examine Oracle's Access Path mechanisms

Indexing

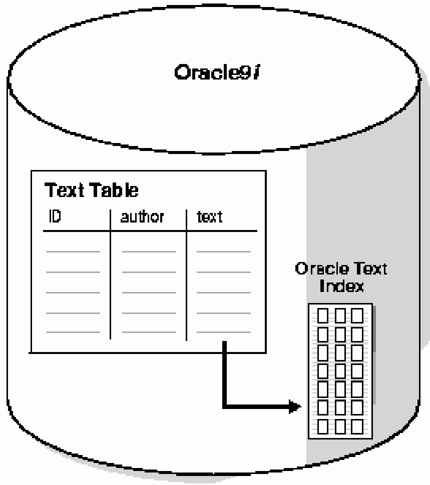


INDEX

ALCOHOL CAN BE A GAS!

What is an Index?

- An index is a fast access path to data
- An index is a table data structure created to improve performance in some cases
- An index consists of two fields, the index key and a pointer which gives the address of the actual record
- The index is sorted on the index key
- For very large files, it is necessary to have a multi-level index (called B tree indexes) as the index itself may be large



Indexes

- Indexes are created on one or more columns of a table.
- After it is created, an index is
  - automatically maintained and used by Oracle.
- Changes to table data (such as adding new rows, updating rows, or deleting rows) are automatically incorporated into all relevant indexes with complete transparency to the users.

Index Properties

Two kinds of blocks:

- Branch Blocks - contain index data that points to lower-level index blocks

  They store:

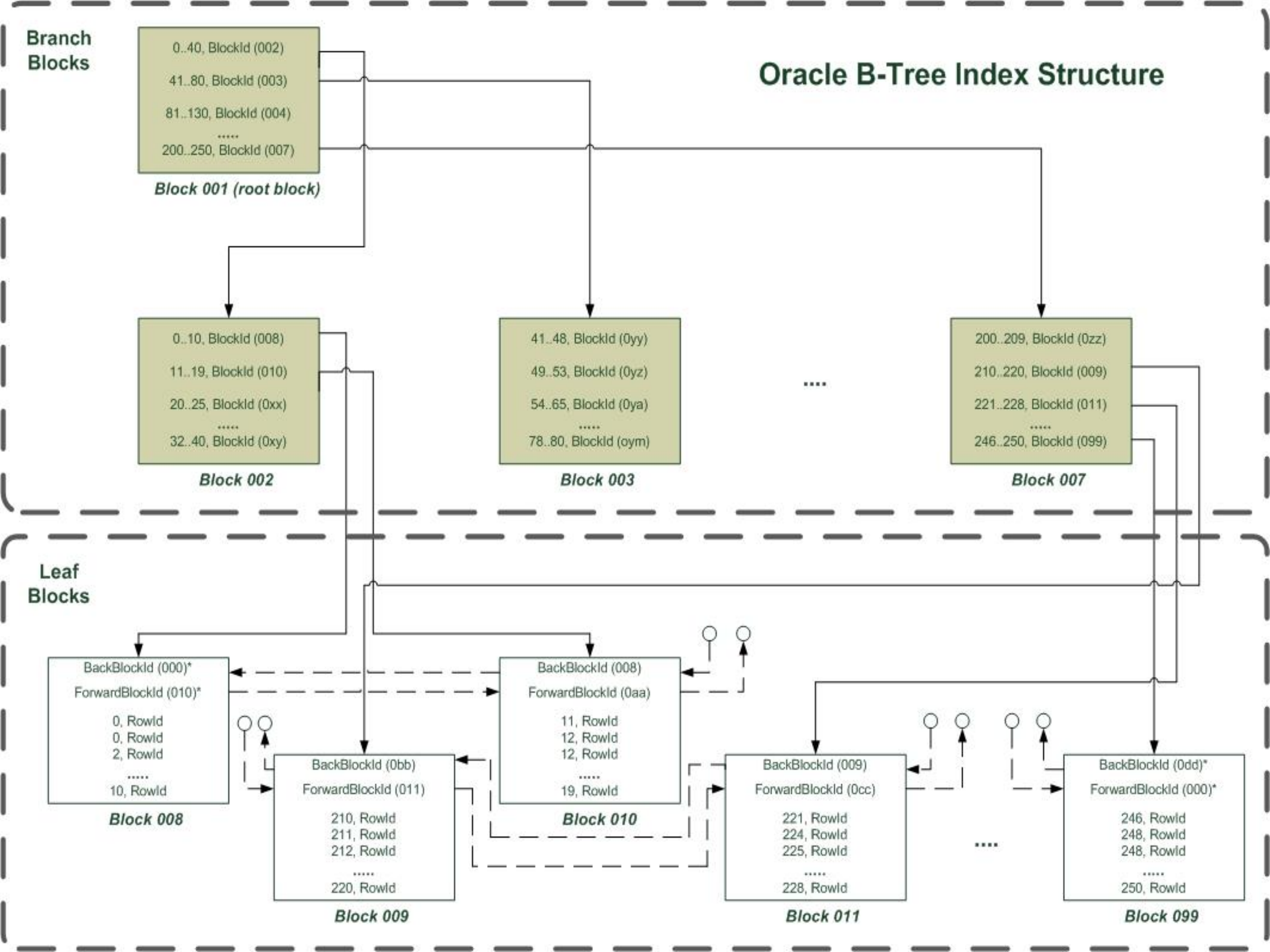
  - The minimum key prefix needed to make a branching decision between two keys
  - The pointer to the child block containing the key

- Leaf blocks - contain every indexed data and a corresponding rowid for the actual row

  They store:

  - The complete index key value for each table row
  - ROWID pointer for each table row

  All <Key, RowID> pairs are linked to their left and right siblings (forward and backward chain) . They are sorted by (key, ROWID).


Oracle B-Tree Index Structure
Branch Blocks
0..40, BlockId (002)
41..80, BlockId (003)
81..130, BlockId (004)
......
200..250, BlockId (007)
Block 001 (root block)
0..10, BlockId (008)
11..19, BlockId (010)
20..25, BlockId (0xx)
......
32..40, BlockId (0xy)
Block 002
41..48, BlockId (0yy)
49..53, BlockId (0yz)
54..65, BlockId (0ya)
......
78..80, BlockId (oym)
Block 003
....
200..209, BlockId (0zz)
210..220, BlockId (009)
221..228, BlockId (011)
......
246..250, BlockId (099)
Block 007
Leaf Blocks
BackBlockId (000)*
ForwardBlockId (010)*
0, RowId
0, RowId
2, RowId
......
10, RowId
Block 008
BackBlockId (008)
ForwardBlockId (0aa)
11, RowId
12, RowId
12, RowId
......
19, RowId
Block 010
BackBlockId (0bb)
ForwardBlockId (011)
210, RowId
211, RowId
212, RowId
......
220, RowId
Block 009
BackBlockId (009)
ForwardBlockId (0cc)
221, RowId
224, RowId
225, RowId
......
228, RowId
Block 011
....
BackBlockId (0dd)*
ForwardBlockId (000)*
246, RowId
248, RowId
248, RowId
......
250, RowId
Block 099

Index Unique Scan

Steps in Index Unique Scans

- Start with the root block
- Search the block keys in sequential order for the key range that contains the required value
- If found, then follow the link from this key value range to the child block
- If no key range contains the required value from Step 2, then the row does not exist
- Repeat steps 2 through 4 if the child block is a branch block
- Search the leaf block for key equal to the value
- If key is found, then return the ROWID of the first key that equals the required value
- If key is not found, then the row does not exist

Index Unique Scan

- If searching for 224:
  - In the root block, 200…250 is the range that contains the required key value = 224
  - Follow the link to branch block 7 (200…250)
  - In this branch block, 221…228 is the range that contains the required key value = 224
  - Follow the link to leaf block 11 (221…228)
  - In this leaf block, search for key 224
  - Found 224, return (KEY, ROWID)

Index Range Scan

- If searching for >= 224 (*do this yourself*)

Indexes

- Consider a file with 1 million records. Suppose that record size is 200 bytes, and the block size is 2000 bytes. Also suppose that the size of the index key is 14 bytes and the pointer size is 6 bytes.
- These 10^6 entries would be stored in $10^6/10^2 = 10^4$ blocks (10,000)
- Thus 10^4 entries are needed at the next level of the index – these entries would be stored in $10^4/10^2$ blocks (= 100)
- At the next level of the index 100 entries are needed – these can be stored in 1 block – so no further levels are necessary
- Thus the no. of indexing levels = 3

Indexes

- The example illustrates the power of indexing – a random search on a 200MB file involved just 4 block accesses
- However, it must be noted that, while an index speeds up retrieval it does slow down updates
- Thus we only build indexes on columns which have a high read/write ratio
- Apart from update overhead, indexes also have a fairly substantial space overhead – up to 25% for each index built, depending on the Database System

Bitmap Indexes

- In a regular index a list of rowids is stored for each key corresponding to the rows with that key value.
- In a *bitmap index*, a bitmap for each key value is used instead of a list of rowids.
- Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value
- A mapping function converts the bit position to an actual rowid

Bitmap Indexes

Bitmap Index Example- CUSTOMER Table

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

Bitmap Indexes

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

- An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers live in the central or west regions?"

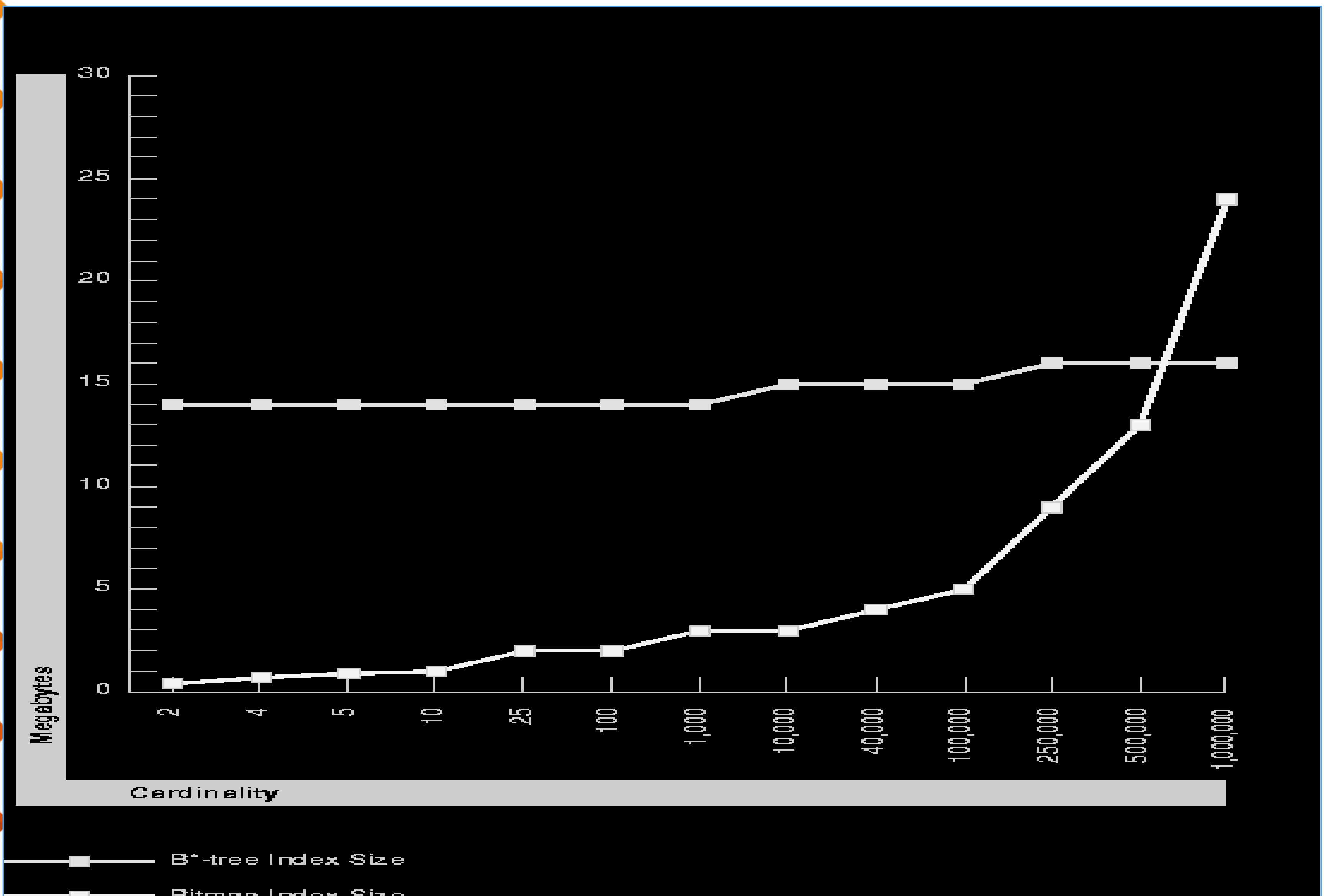
```
SELECT COUNT(*) FROM CUSTOMER WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');
```

Bitmap Indexes

status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

- The advantages of using bitmap indexes are greatest for low cardinality columns: that is, columns in which the number of distinct values is small compared to the number of rows in the table
- For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index

Bitmap Index Size



BTree Indexes vs. Bitmap Indexes

- Use BTree indexes for columns with high cardinality (e.g. customer name, phone number, etc.)

- Use bitmap indexes on columns with low cardinality (typically columns whose values are repeated more than a hundred times)

BTree vs Bitmap Indexes

BTree	**Bitmap**
Suitable for high cardinality columns	Suitable for low cardinality columns
Update on keys relatively inexpensive	Updates on keys very expensive
Inefficient for queries using OR predicates	Efficient for queries using OR predicates
Useful for OLTP	Useful for data warehousing

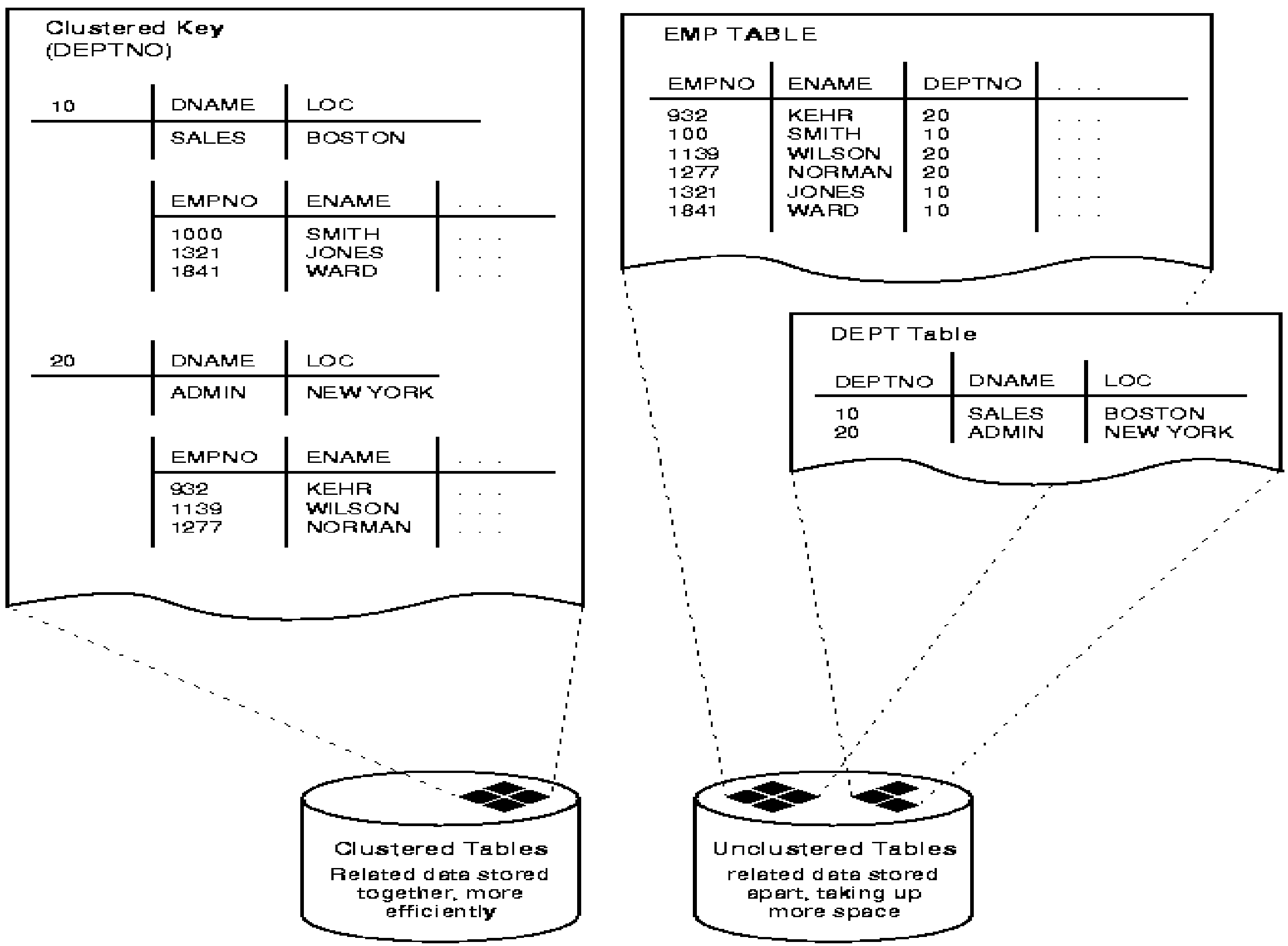
BTree vs. Bitmap Indexes

- The following statements create indexes in Oracle:

1. BTree CREATE INDEX emp_ename_idx ON emp(ename)
2. Bitmap CREATE bitmap INDEX emp_region_idx ON emp(region)

Clusters

- Clusters store data from two or more related tables together in an interleaved fashion



Clustering

- In clusters rows from more than one table are stored in each data block
- Disk I/O is reduced and access time improves for joins of clustered tables (in a master-detail relationship)
- However tables that are *inserted* into or *modified* often are not good candidates for clustering
- In Oracle, clusters must be created before creating the tables that make up the cluster

```
CREATE CLUSTER emp_dept (deptno NUMBER(3)) PCTUSED 80 PCTFREE 5
    SIZE 600)
```

- The cluster size specifies the average number of bytes required to store the rows for a cluster key value

Clustering

- Once the cluster is created the next step is to create the tables and the cluster index

CREATE TABLE dept (deptno NUMBER(3) PRIMARY KEY, ...) CLUSTER emp_dept (deptno)

CREATE TABLE emp (empno NUMBER(5) PRIMARY KEY, ename VARCHAR2(15) NOT NULL, . . . deptno NUMBER(3) REFERENCES dept) CLUSTER emp_dept (deptno)

CREATE INDEX emp_dept_index ON CLUSTER emp_dept

INITRANS 2 MAXTRANS 5 PCTFREE 5;

Hash Clustering

- Hashing goes one step better than indexing by calculating the locations of rows (rather than having to search using an index)

- Hashing provides optimal performance when the search criterion is an equality condition on a column value

  e.g. Select all details of employee with empno =1234

Hash Clustering

- When a table is hashed its rows are stored in a number of *buckets*
- Each *bucket* contains rows which have the same hash key value
- A hashing *function* is used to map rows to *buckets*
- In order to find a row with only one block access (optimal value) the hashing function must distribute rows evenly across buckets

Hash Clustering

- The best hashing function is usually the remainder function used by the algorithm below

Algorithm

1. Determine the number of buckets to be allocated to the file
2. Select a prime number that is approx. equal to this number
3. Divide each hash key value by this number
4. Use the remainder as the bucket address

Hash Clustering Example

- **Consider this Products table** **(15 products)**

Product #	Description	Finish
100	Stereo Cabinet	Maple
125	Coffee Table	Walnut
153		
207		
219		
221		
286		
314		
363		
394		
418		
434		
488		
500		
515		

Hash Clustering Example

- If two rows fit into a bucket (block), then we need at least 8 buckets (15/2 = 7.5)
- Use a hash function based on 13 (prime):

Key	Prime Divisor	Product	Remainder (hash)
100	13	7	9
125	13	9	8
153	13	11	10
207	13	15	12
219	13	16	11
221	13	17	0
286	13	22	0
314	13	24	2
363	13	27	12
394	13	30	4
418	13	32	2
434	13	33	5
488	13	37	7
500	13	38	6
515	13	39	8

Bucket #	Key value
0	221 286
1	
2	314 418
3	
4	394
5	434
6	500
7	488
8	125 515
9	100
10	153
11	219
12	207 363

Hashing Performance Characteristics

- Hashing provides very good performance for searching, modifying and deleting rows when the search criterion is an equality condition on the hash key
- In order to ensure good performance, the number of overflows must be minimized, this is done by selecting the hash key carefully
- Hashing clusters the table, so only ONE hash key (which can be composite) can be used (contrast with indexing) for a given table
- Hashing does not support range searching at all (full table scan needed)

Oracle's Hashing Mechanism

- Oracle has a built-in hash function based on the "Remainder" method
- This function requires two parameters:
  1. Hashkeys – which specifies the total number of buckets required (Oracle will round this value up to the nearest prime number)
  2. Size – this gives the total space in bytes required to store the average number of rows associated with a bucket
- Oracle enables more than one table to be hashed in the same cluster

```
CREATE CLUSTER personnel (deptno NUMBER )
    SIZE 512 HASHKEYS 500 STORAGE (INITIAL 100K NEXT 50K);
CREATE TABLE emp (empno NUMBER PRIMARY KEY, ename
    VARCHAR2(10) NOT NULL ,job VARCHAR2(9), hiredate DATE , sal
    NUMBER(10,2), deptno NUMBER(2) NOT NULL )
    CLUSTER personnel (deptno)
CREATE TABLE dept (deptno NUMBER(2), dname VARCHAR2(9), loc
    VARCHAR2(9)) CLUSTER personnel (deptno);
```

Oracle's Hashing Mechanism

- To define a single table hash cluster use:

```
CREATE CLUSTER personnel (deptno NUMBER)
    SIZE 512 SINGLE TABLE HASHKEYS 500
```

- In cases when the cluster key is a unique identifier that is uniformly distributed over its range, you can by-pass Oracle's internal function and simply specify the hash column

```
CREATE CLUSTER emp_cluster (empno NUMBER)
            . . .
SIZE 55
HASH IS empno HASHKEYS 10007
```

Oracle's Hashing Mechanism

- If the cluster key values are not evenly distributed you can create your own hash function
- For example, if your cluster key is the employee's home area code, it is likely that many employees will hash to the same hash value
- To alleviate this problem, you can place the following expression in the HASH IS clause of the CREATE CLUSTER command:

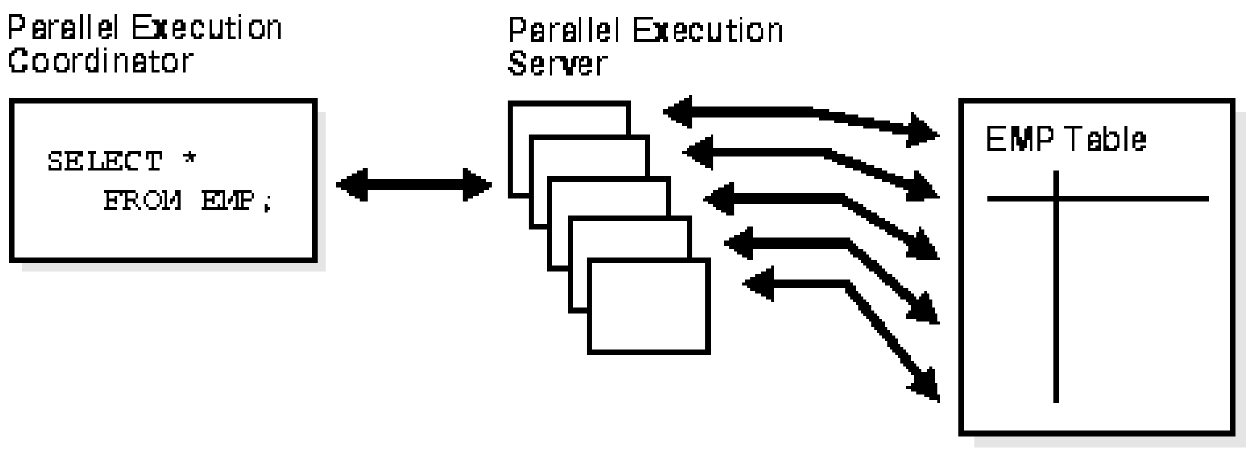
```
MOD (emp.home_area_code + emp.home_prefix + emp.home_suffix), 101)
```

Using Parallelism to improve Performance

- Many DBMSs parallelize data-intensive operations to improve performance
- Two types of parallelism are possible:

1. Intra-operation parallelism – here a single operation is parallelised

Parallel Full Table Scan



Using Parallelism

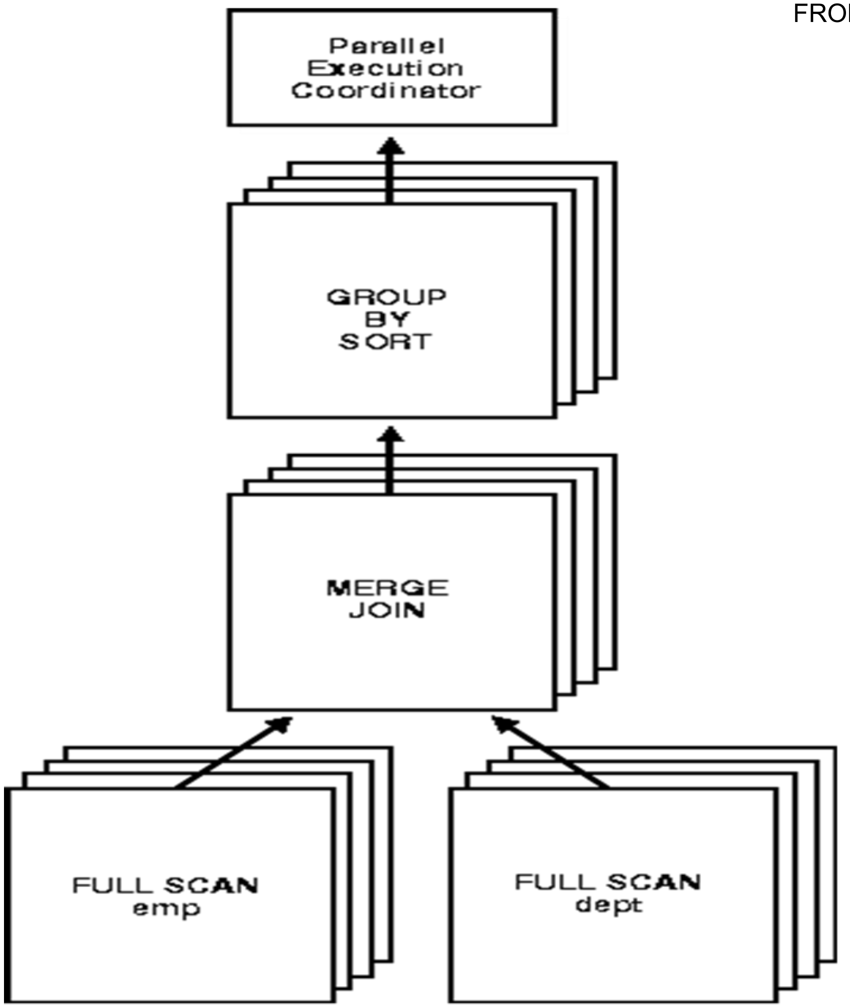
- Parallelism can also be used to perform many operations simultaneously

SELECT dname, MAX(sal), AVG(sal)

FROM emp, dept

GROUP BY dname;

WHERE emp.deptno = dept.deptno



Parallelism in Oracle

- Oracle supports both intra-operation and inter-operation parallelism
- Oracle allows the user to set the degree of parallelism
- The degree of parallelism is specified at the statement level (with hints or the PARALLEL clause), at the table or index level or by default based on the number of CPUs.

```
ALTER TABLE customer PARALLEL 4
ALTER CLUSTER dept_emp PARALLEL 4
ALTER INDEX iemp PARALLEL
SELECT /*+ PARALLEL(emp, 4) */ COUNT(*) FROM emp
```

References

1. Fundamentals of Database Systems, Elmasri/Navathe, Chapters 5 and 6
2. Modern Database Management, Hoffer/Prescott/Topi, Chapter 6
3. Oracle 10g Concepts
4. Oracle 10g Administrator's Guide
5. Oracle 10g Tuning Guide

Review Questions

1. What is the decision to choose B* indexes over bitmap indexes based on?
2. Discuss the situations in which Hashing is preferred over Indexing.
3. How does clustering improve performance?
4. Discuss two different ways that parallelism can be used to improve performance in a large database environment