

# CHAPTER 8

# **TOPIC OUTLINE**

**Uses of Indexes**

**Function-Based Indexes**

**Composite Indexes**

**Bitmap Indexes**

**IOTs**

**Bitmap Join Indexes**

# Guidelines for Application-Specific Indexes

You can create indexes on columns to speed up queries. Indexes provide faster access to data for operations that return a small portion of a table's rows.

In general, you should create an index on a column in any of the following situations:

- The column is queried frequently.
- A referential integrity constraint exists on the column.
- A `UNIQUE` key integrity constraint exists on the column.

You can create an index on any column; however, if the column is not used in any of these situations, creating an index on the column does not increase performance and the index takes up resources unnecessarily.

Although the database creates an index for you on a column with an integrity constraint, explicitly creating an index on such a column is recommended.

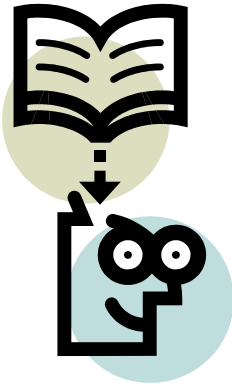
You can use the following techniques to determine which columns are best candidates for indexing:

- Use the `EXPLAIN PLAN` feature to show a theoretical execution plan of a given query statement.
- Use the `V$SQL_PLAN` view to determine the actual execution plan used for a given query statement.

Sometimes, if an index is not being used by default and it would be most efficient to use that index, you can use a query hint so that the index is used.

The following sections explain how to create, alter, and drop indexes using SQL commands, and give guidelines for managing indexes.

# Create Indexes After Inserting Table Data



Typically, you insert or load data into a table (using SQL\*Loader, Import BULK COPY or FOR ALL) before creating indexes. Otherwise, the overhead of updating the index slows down the insert or load operation. The *exception* to this rule is that you must create an index for a cluster before you insert any data into the cluster.

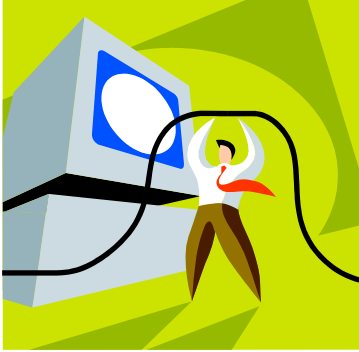
**ORACLE** Enterprise Manager 11g  
Database Control

## Database Instance: winorcl

<a href="#">Home</a>	<a href="#">Performance</a>	<a href="#">Availability</a>	<a href="#">Server</a>	<a href="#">Schema</a>	<a href="#">Data Movement</a>
<b>Storage</b>			<b>Database Configuration</b>		
<a href="#">Control Files</a>			<a href="#">Memory Advisors</a>		
<a href="#">Tablespaces</a>			<a href="#">Automatic Undo Management</a>		
<a href="#">Temporary Tablespace Groups</a>			<a href="#">Initialization Parameters</a>		
<a href="#">Datafiles</a>			<a href="#">View Database Feature Usage</a>		
<a href="#">Rollback Segments</a>					
<a href="#">Redo Log Groups</a>					
<a href="#">Archive Logs</a>					
<a href="#">Migrate to ASM</a>					
<a href="#">Make Tablespace Locally Managed</a>					

If sorting of data due to creation of IOT's, clusters or Sort Clusters or simply large application sort processing cause significant wait events, it will help greatly to create temporary tablespace groups and place multiple temporary tablespaces in them to assist in distributing sort processing.

# Switch Your Temporary Tablespace to Avoid Space Problems Creating Indexes



When you create an index on a table that already has data, Oracle Database must use sort space to create the index. The database uses the sort space in memory allocated for the creator of the index (the amount for each user is determined by the initialization parameter `SORT_AREA_SIZE`), but the database must also swap sort information to and from temporary segments allocated on behalf of the index creation. If the index is extremely large, it can be beneficial to complete the following steps:

1. Create a new temporary tablespace using the `CREATE TABLESPACE` command.
2. Use the `TEMPORARY TABLESPACE` option of the `ALTER USER` command to make this your new temporary tablespace.
3. Create the index using the `CREATE INDEX` command.
4. Drop this tablespace using the `DROP TABLESPACE` command. Then use the `ALTER USER` command to reset your temporary tablespace to your original temporary tablespace.

Under certain conditions, you can load data into a table with the SQL\*Loader "direct path load", and an index can be created as data is loaded.

# Index the Correct Tables and Columns



Use the following guidelines for determining when to create an index:

- Create an index if you frequently want to retrieve less than about 15% of the rows in a large table. This threshold percentage varies greatly, however, according to the relative speed of a table scan and how clustered the row data is about the index key.
- Index columns that are used for joins to improve join performance.
- Small tables do not require indexes; if a query is taking too long, then the table might have grown from small to large.

Some columns are strong candidates for indexing. Columns with one or more of the following characteristics are good candidates for indexing:

- Values are unique in the column, or there are few duplicates.
- There is a wide range of values (good for regular indexes).
- There is a small range of values (good for bitmap indexes).
- The column contains many nulls, but queries often select all rows having a value. In this case, a comparison that matches all the non-null values, such as:  
`WHERE COL_X >= -9.99 *power(10,125)`

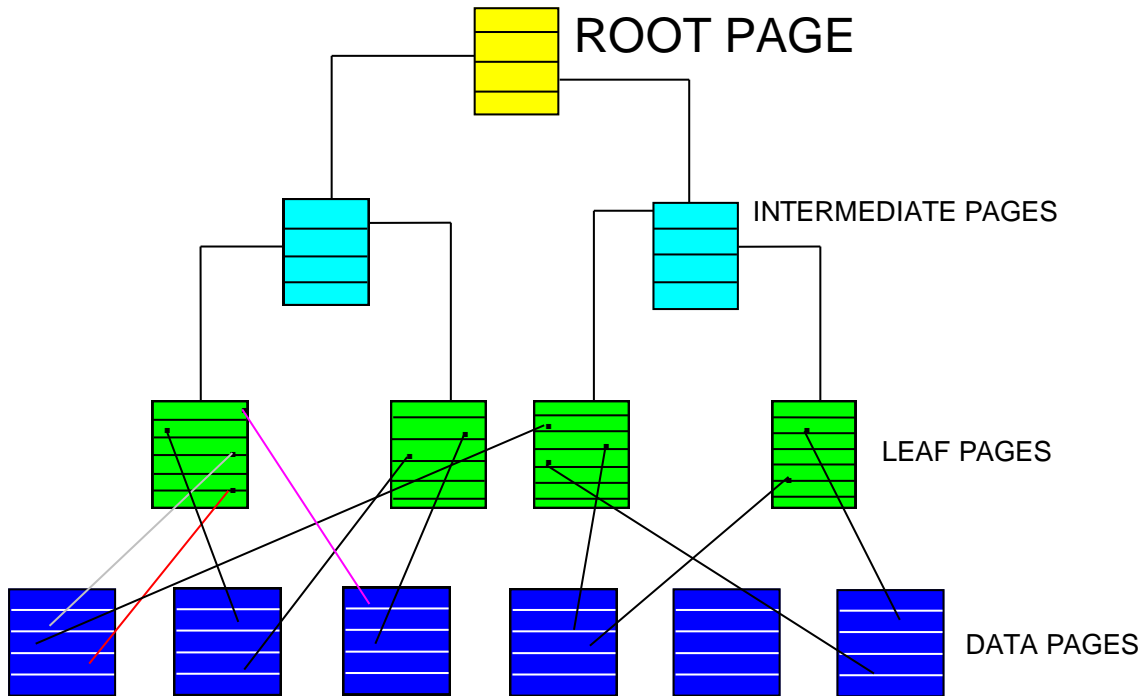
This is preferable to `WHERE COL_X IS NOT NULL`

This is because the first uses an index on `COL_X` (assuming that `COL_X` is a numeric column).

Columns with the following characteristics are less suitable for indexing:

- There are many nulls in the column and you do not search on the non-null values.

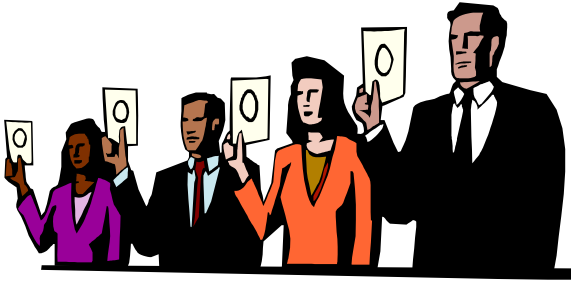
# Limit the Number of Indexes for Each Table



The more indexes, the more overhead is incurred as the table is altered. When rows are inserted or deleted, all indexes on the table must be updated. When a column is updated, all indexes on the column must be updated.

You must weigh the performance benefit of indexes for queries against the performance overhead of updates. For example, if a table is primarily read-only, you might use more indexes; but, if a table is heavily updated, you might use fewer indexes.

# Choose Order of Columns in Compound Indexes



Although you can specify columns in any order in the `CREATE INDEX` command, the order of columns in the `CREATE INDEX` statement can affect query performance. In general, you should put the column expected to be used most often first in the index. You can create a composite index (using several columns), and the same index can be used for queries that reference all of these columns, or just some of them.

Assume that there are five vendors, and each vendor has about 1000 parts.

Suppose that the `VENDOR_PARTS` table is commonly queried by SQL statements such as the following:

```
SELECT * FROM vendor_parts
  WHERE part_no = 457 AND vendor_id = 1012;
```

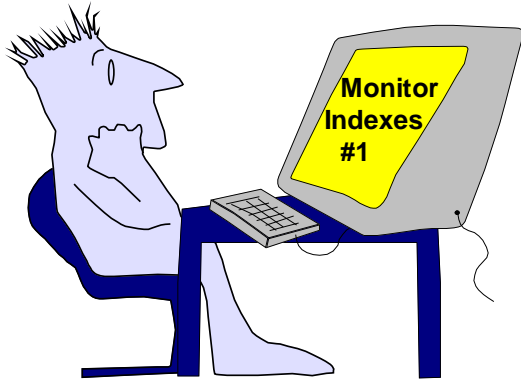
To increase the performance of such queries, you might create a composite index putting the most selective column first; that is, the column with the *most* values:

```
CREATE INDEX ind_vendor_id
  ON vendor_parts (part_no, vendor_id);
```

Composite indexes speed up queries that use the *leading portion* of the index. So in this example, queries with `WHERE` clauses using only the `PART_NO` column also note a performance gain. Because there are only five distinct values, placing a separate index on `VENDOR_ID` would serve no purpose.



# Drop Indexes That Are No Longer Required



You might drop an index if:

- It does not speed up queries. The table might be very small, or there might be many rows in the table but very few index entries.
- The queries in your applications do not use the index.
- The index must be dropped before being rebuilt.

When you drop an index, all extents of the index's segment are returned to the containing tablespace and become available for other objects in the tablespace.

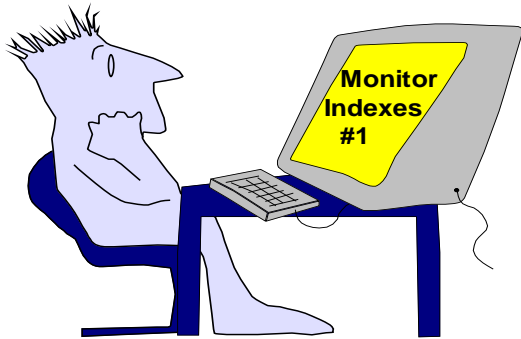
Use the SQL command `DROP INDEX` to drop an index. For example, the following statement drops a specific named index:

```
DROP INDEX Emp_ename;
```

If you drop a table, then all associated indexes are dropped.

To drop an index, the index must be contained in your schema or you must have the `DROP ANY INDEX` system privilege.

# Creating Indexes: Basic Examples



You can create an index for a table to improve the performance of queries issued against the corresponding table. You can also create an index for a cluster. You can create a *composite* index on multiple columns up to a maximum of 32 columns. A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block.

Oracle Database automatically creates an index to enforce a `UNIQUE` or `PRIMARY KEY` integrity constraint. In general, it is better to create such constraints to enforce uniqueness, instead of using the obsolete `CREATE UNIQUE INDEX` syntax.

Use the SQL command `CREATE INDEX` to create an index.

In this example, an index is created for a single column, to speed up queries that test that column:

```
CREATE INDEX emp_ename ON emp_tab(ename);
```

In this example, several storage settings are explicitly specified for the index:

```
CREATE INDEX emp_ename ON emp_tab(ename)
  TABLESPACE users
  STORAGE (INITIAL      20K
           NEXT         20k
           PCTINCREASE 75)
  PCTFREE      0
  COMPUTE STATISTICS;
```

In this example, the index applies to two columns, to speed up queries that test either the first column or both columns:

```
CREATE INDEX emp_ename ON emp_tab(ename, empno) COMPUTE STATISTICS;
```

# When to Use Function-Based Indexes



A function-based index is an index built on an expression. It extends your indexing capabilities beyond indexing on a column. A function-based index increases the variety of ways in which you can access data.

The expression indexed by a function-based index can be an arithmetic expression or an expression that contains a PL/SQL function, package function, C callout, or SQL function. Function-based indexes also support linguistic sorts based on collation keys, efficient linguistic collation of SQL statements, and case-insensitive sorts.

Like other indexes, function-based indexes improve query performance. For example, if you need to access a computationally complex expression often, then you can store it in an index. Then when you need to access the expression, it is already computed. You can find a detailed description of the advantages of function-based indexes in ["Advantages of Function-Based Indexes"](#).

Function-based indexes have all of the same properties as indexes on columns. However, unlike indexes on columns which can be used by both cost-based and rule-based optimization, function-based indexes can be used by only by cost-based optimization. Other restrictions on function-based indexes are described in ["Restrictions for Function-Based Indexes"](#).

## Advantages of Function-Based Indexes

Function-based indexes:

- *Increase the number of situations where the optimizer can perform a range scan instead of a full table scan.* For example, consider the expression in this `WHERE` clause:

```
CREATE INDEX Idx ON Example_tab(Column_a + Column_b);  
SELECT * FROM Example_tab WHERE Column_a + Column_b < 10;
```

The optimizer can use a range scan for this query because the index is built on `(column_a + column_b)`. Range scans typically produce fast response times if the predicate selects less than 15% of the rows of a large table.

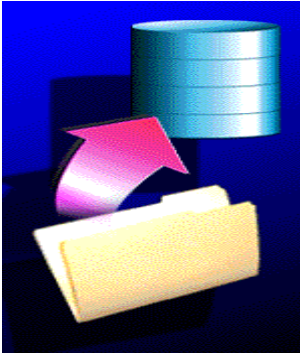
*Precompute the value of a computationally intensive function and store it in the index.* An index can store computationally intensive expression that you access often. When you need to access a value, it is already computed, greatly improving query execution performance.

- *Create indexes on object columns and REF columns.* Methods that describe objects can be used as functions on which to build indexes. For example, you can use the `MAP` method to build indexes on an object type column.
- *Create more powerful sorts.* You can perform case-insensitive sorts with the `UPPER` and `LOWER` functions, descending order sorts with the `DESC` keyword, and linguistic-based sorts with the `NLSSORT` function.

Another function-based index calls the object method `distance_from_equator` for each city in the table. The method is applied to the object column `Reg_Obj`. A query could use this index to quickly find cities that are more than 1000 miles from the equator:

```
CREATE INDEX Distance_index  
ON Weatherdata_tab (Distance_from_equator (Reg_obj));  
  
SELECT * FROM Weatherdata_tab  
WHERE (Distance_from_equator (Reg_Obj)) > '1000';
```

# Examples of Function-Based Indexes



## Example: Function-Based Index for Case-Insensitive Searches

The following command allows faster case-insensitive searches in table EMP\_TAB.

```
CREATE INDEX Idx ON Emp_tab (UPPER(Ename));
```

The `SELECT` command uses the function-based index on `UPPER(e_name)` to return all of the employees with name like `:KEYCOL`.

```
SELECT * FROM Emp_tab WHERE UPPER(Ename) like :KEYCOL;
```

## Example: Precomputing Arithmetic Expressions with a Function-Based Index

The following command computes a value for each row using columns A, B, and C, and stores the results in the index.

```
CREATE INDEX Idx ON Fbi_tab (A + B * (C - 1), A, B);
```

The `SELECT` statement can either use index range scan (since the expression is a prefix of index `IDX`) or index fast full scan (which may be preferable if the index has specified a high parallel degree).

```
SELECT a FROM Fbi_tab WHERE A + B * (C - 1) < 100;
```

## NON-UNIQUE INDEXES

**TASK:** Create an Index so that ORACLE can retrieve data based upon an Index search versus a table scan

**SQL:**

```
CREATE INDEX xname  
ON STAFF (name)
```

**RESULT:** Index xname is built with the key field of NAME.

### NOTES:

- A general guideline is to create an index on anything you want to search frequently.
- Primary keys should always be indexed uniquely.
- Foreign keys should also be indexed, to help you find the information they point to more efficiently.
- In the case of joins , the system can perform the join much faster if the columns are sorted.
- Must be the table owner in order to create or have Index privilege on table .

## UNIQUE INDEXES

**TASK:** Create a unique index on the primary key field  
DEPTNUMB of the ORG table

**SQL:**  
**CREATE UNIQUE INDEX XORG**  
**ON ORG (DEPTNUMB)**

**RESULT:** Index XORG is created which will not allow duplicate  
Department numbers in the ORG table.

### NOTES:

1. Enforces uniqueness of Rows
2. Speeds Joins (Referential checks)
3. Speeds Data Retrieval
4. Speeds ORDER BY and GROUP BY

## COMPOSITE INDEXES

**TASK:** Create an index on name and department on the STAFF table

**SQL:**  
**CREATE INDEX xstaff\_dept**  
**ON STAFF (NAME,DEPT)**

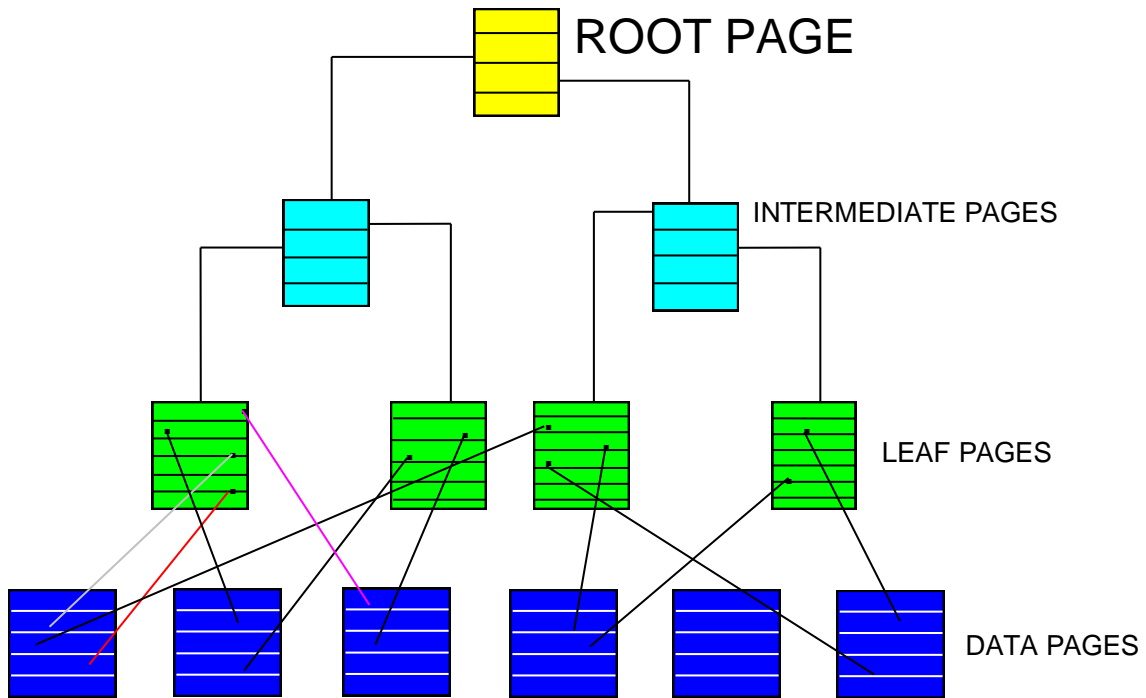
**RESULT:** Index xstaff\_dept is created which will not allow duplicate name and dept columns in the STAFF table.

### NOTES:

- A composite index specifies two or more columns as the index
- Composite columns are searched as a unit
- Column order in the CREATE INDEX statement doesn't have to match column order in the CREATE TABLE statement
- May create either a unique, non-unique or even a clustered index using a composite index.



## NONCLUSTERED INDEXES



### NOTES:

- Row keys are searched starting with a high key search in the root page
- The leaf level contains pointers to the rows on the data page
- The pointers add a level between the index and the data

# INDEXES WITH PCTFREE

TASK: Limit the number of Index pages which are filled with rows at index creation time

SQL:

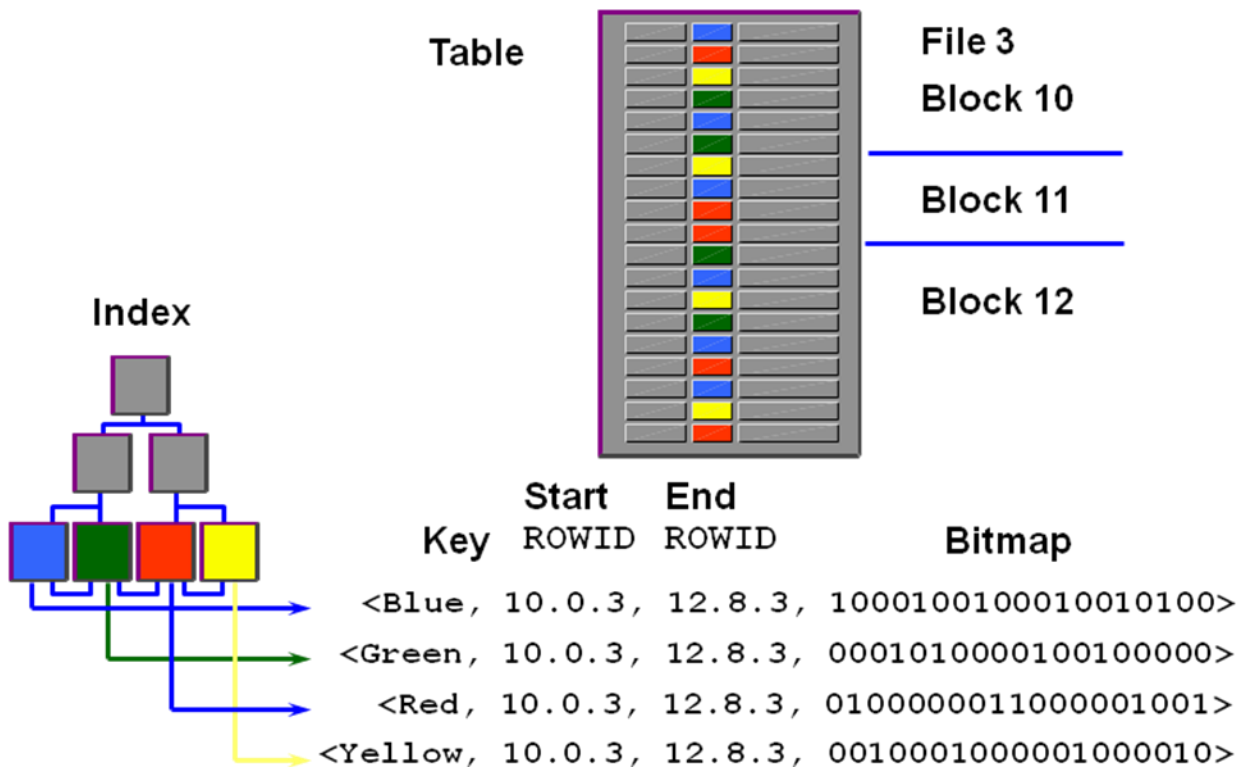
```
CREATE UNIQUE INDEX xstaff_indx  
ON STAFF (id,name)  
PCTFREE 10
```

RESULT: 90% of each Index page is filled at creation with rows from the STAFF table. This minimizes index page splitting and allows for fine-tuning for performance.

NOTES:

PCTFREE of 10% will keep 10% of the index page available for updates and inserts. This would be useful for tables where the index values will not change a great deal through inserts, deletions, or updates..

# BITMAP INDEXES



Bitmap Indexes are used for:

- Used for Low-cardinality columns
- Good for multiple predicates
- Use minimal storage space
- Best for read-only systems
- Good for very large tables

# CREATING AND MAINTAINING BITMAP INDEXES

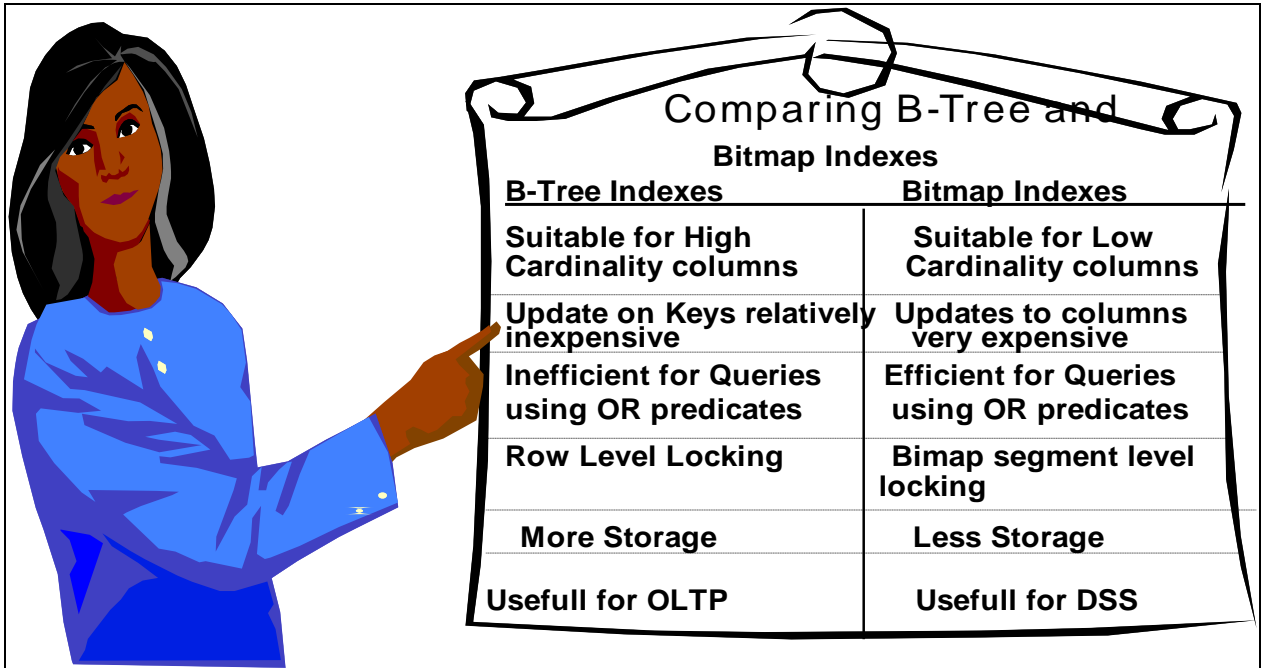


```
CREATE BITMAP INDEX INV_BIKE_COLOR  
ON INVENTORY (COLOR)  
STORAGE (INITIAL 200K NEXT 200K  
PCTINCREASE 0) TABLESPACE ORASERVE;
```

- **CREATE\_BITMAP\_AREA\_SIZE** defines the amount of memory for an index creation. The default is 8mb. The higher the cardinality the more memory you will need.
  - **BITMAP\_MERGE\_AREA\_SIZE** defines the amount of memory used to merge bitmaps retrieved from a range scan of an index. The default is 1 MB.

In a DSS environment, data is usually maintained by using bulk inserts and updates. Index maintenance is deferred until the end of each DML operation. For example, if you insert 1,000 rows into a table that has a bitmapped index, the bitmapped column information and the ROWID information from the inserted rows are placed into the reference to the sort buffer, and then the updates of all 1,000 index entries are batched. The **SORT\_AREA\_SIZE** parameter will need to be monitored when using these indexes.

# INDEX COMPARISONS



The illustration shows a woman with dark hair and a blue top pointing her right index finger towards a scroll. The scroll is titled 'Comparing B-Tree and Bitmap Indexes' and contains a table comparing the two index types. The scroll is tied with a black ribbon at the top.

<b>B-Tree Indexes</b>	<b>Bitmap Indexes</b>
Suitable for High Cardinality columns	Suitable for Low Cardinality columns
Update on Keys relatively inexpensive	Updates to columns very expensive
Inefficient for Queries using OR predicates	Efficient for Queries using OR predicates
Row Level Locking	Bitmap segment level locking
More Storage	Less Storage
Usefull for OLTP	Usefull for DSS

Bitmap indexes should be investigated and tested for Data Warehousing applications, Financials and all low-cardinality accesses to tables where the predicates are not widely dispersed.

# CREATING REVERSE KEY INDEXES



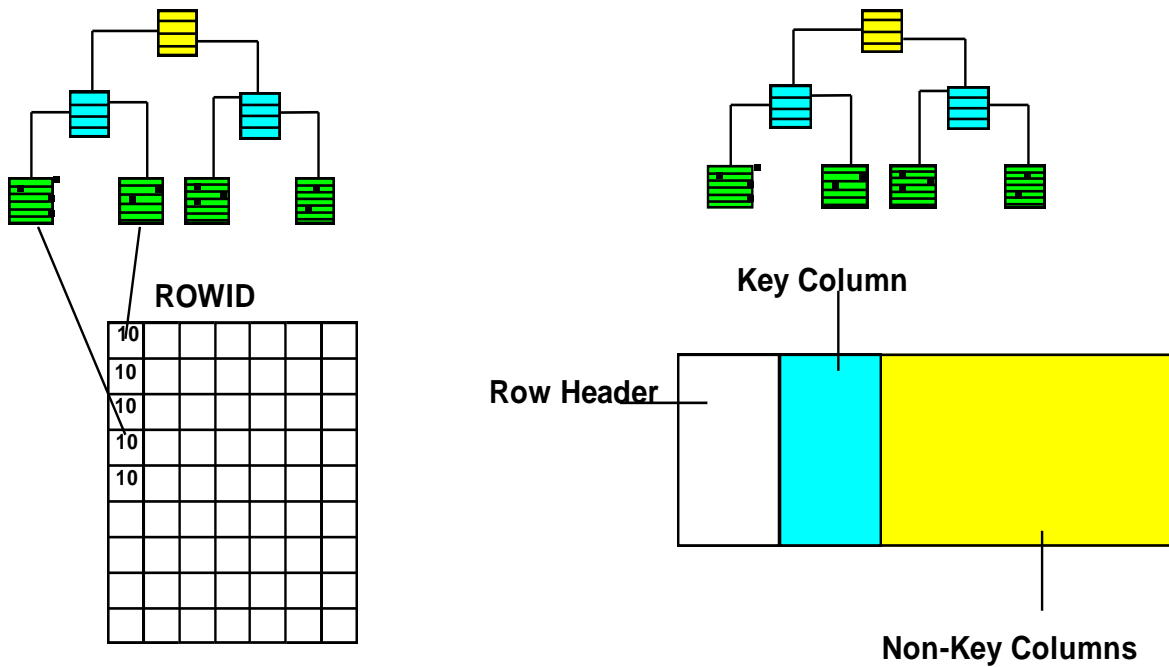
Reducing HOT SPOT Hits

**Create unique index po\_no\_idx on puorder (po\_no)  
REVERSE pctfree 30  
storage (initial 400k next 400k pctincrease 0)  
tablespace jerindx.**

Creating a reverse key index reverses the bytes of each column key value, keeping the column order in case of a composite key.

Use a Reverse key index when an ever-increasing key will repetitively degrade the index height level and cause the HOT SPOT syndrome.

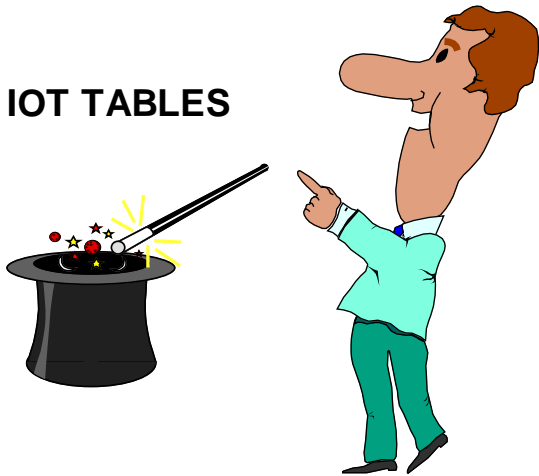
# INDEX-ORGANIZED TABLES



- Index contains both the primary key and other column values
- No duplication of the values for the primary key column (less storage)
- Faster key-based access for queries involving exact matches or range searches or both.
  - Logical Rowid used instead of a physical ROWID. Access to this value is through a UROWID.

# CREATING INDEX-ORGANIZED TABLES

## IOT TABLES

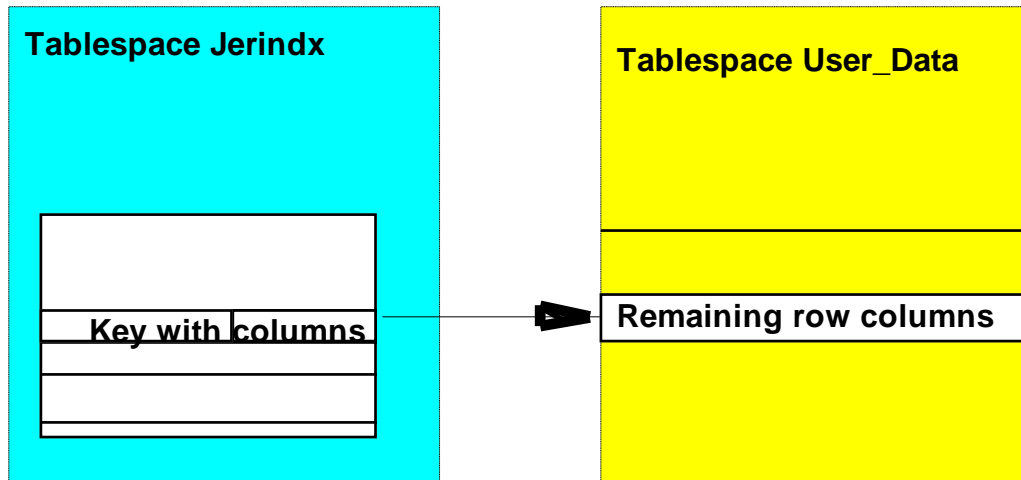


```
CREATE TABLE SALES
(
  OFFICE_ID      NUMBER(3) NOT NULL,
  QTR_END        DATE,
  REVENUE        NUMBER(11,2),
  REVIEW         VARCHAR2(200)
  CONSTRAINT SALES_PK PRIMARY KEY (OFFICE_ID,QTR_END)
)
ORGANIZATION INDEX TABLESPACE JERINDX
PCTTHRESHOLD 20
INCLUDING REVENUE
OVERFLOW TABLESPACE USER_DATA;
```

**IOT-TABLES** store as many rows in a leaf page as **PCTTHRESHOLD** limit will allow. The IOT-Table will store the most frequently used columns with the primary key value. Non key column values that are not used frequently are kept in the overflow tablespace. This is called the overflow area.



# IOT ROW-OVERFLOW STRATEGY



- **PCTTHRESHOLD Clause**

This clause specifies the percentage of space reserved in the index for an index organized table row. If a row exceeds the size calculated based upon this value, all columns after the column named in the INCLUDING clause are moved to the overflow segment. If OVERFLOW is not specified the rows exceeding the threshold are rejected. The default PCTTHRESHOLD = 50.

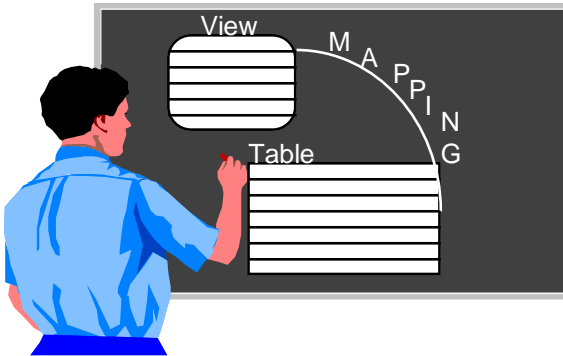
- **INCLUDING Clause**

This clause specifies a column to be placed into the IOT Table and all other row columns to be placed in overflow. Oracle accommodates all non-key columns up to the column specified in the INCLUDING clause in the LEAF block provided it does not exceed the specified threshold.

- **OVERFLOW Clause and Segment**

This clause specifies the non-key columns exceeding the specified THRESHOLD and are subsequently placed in the OVERFLOW segment.

# BITMAP INDEXES ON IOT

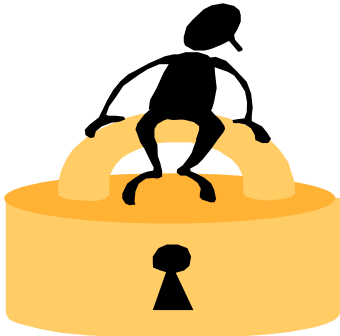


In Oracle19c, you can create a bitmap index on an IOT table. A mapping table is used to resolve ROWIDs needed for the IOT, and the physical ROWIDs needed by the bitmap index. The mapping table is built by specifying the MAPPING TABLE clause when creating the IOT.

Performance problems occur for IOT's when:

- Leaf blocks of an IOT split and cause performance hits
- Analyze indexes and query PCT\_DIRECT\_ACCESS in the DBA\_INDEXES table to see percentage of rows gotten through  
A correct guess on the IOT
- Rebuild mapping tables with the ALTER Table command, including  
The MAPPING TABLE UPDATE BLOCK REFERENCES clause.

# FUNCTION-BASED INDEXES



One of the keys to using indexes

Consider the following query of the STAFF table that identifies information about a specific staff record, based on the uppercase version of the name field.

```
SELECT ID, NAME, JOB, SALARY  
FROM staff  
WHERE UPPER(NAME) = 'RICH';
```

In this case, a normal index on the NAME field would not be optimal because the index contains each staff record's NAME field as it exists in the record itself (in other words, in all uppercase letters, in all lowercase letters, or in a mixed case of letters). Consequently, to do a comparison with an index on the NAME column Oracle needs to convert each key in the index to uppercase letters and then compare the resulting string to the search criteria.

To overcome this overhead processing we create a function-based index:

```
SQL> CREATE INDEX staff_upper_name  
      ON STAFF (UPPER (NAME));
```

In Oracle8i, index-only scans required the index column to be NOT NULL in order to be used. This meant expression used in Function -based indexes could not be used because expression are always defined as NULL. Oracle19c corrects that.

# Bitmap Join Indexes on Multiple Tables

As you become more familiar with using the bitmap join index, you will be able to solve complex business problems that involve multiple tables. The following example shows how to apply the bitmap join index to multiple tables. The syntax is still the same, but it has now been expanded to include multiple columns in the index and multiple tables being joined for the index. The example shown next assumes that the unique constraint on DEPT1.DEPTNO from the example in the earlier listing (where I added a unique constraint to the DEPT1 table) exists and, additionally, that it exists on SALES1.EMPNO (creation not shown).

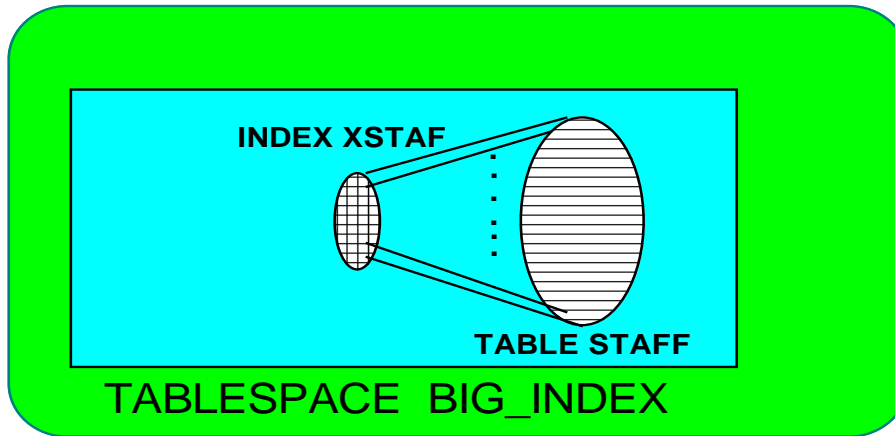
```
Create bitmap index emp_dept_location_ms
on      empl (dept1.loc, sales1.marital_status)
from    empl, dept1, sales1
where   empl.deptno = dept1.deptno
and     empl.empno = sales1.empno;
```

The query in this next listing is now able to use the bitmap join index appropriately:

```
select empl.empno, empl.ename, dept1.loc, sales1.marital_status
from    empl, dept1, sales1
where   empl.deptno = dept1.deptno
and     empl.empno = sales1.empno;
```

# INDEX BEFORE OR AFTER LOADING DATA

When to create Indexes?



Issues to consider when creating indexes and loading data:

- Indexes slow down data loading
- Usually more efficient to index data after loading it
- The sort area may not be large enough for extremely large tables

Consider using the NOSORT option if data has been loaded in ascending key order:

```
ORACLE> CREATE INDEX BIG_BIGNO ON BIG(BIGNO) NOSORT  
TABLESPACE BIG_INDEX
```

## CLUSTERING TABLES

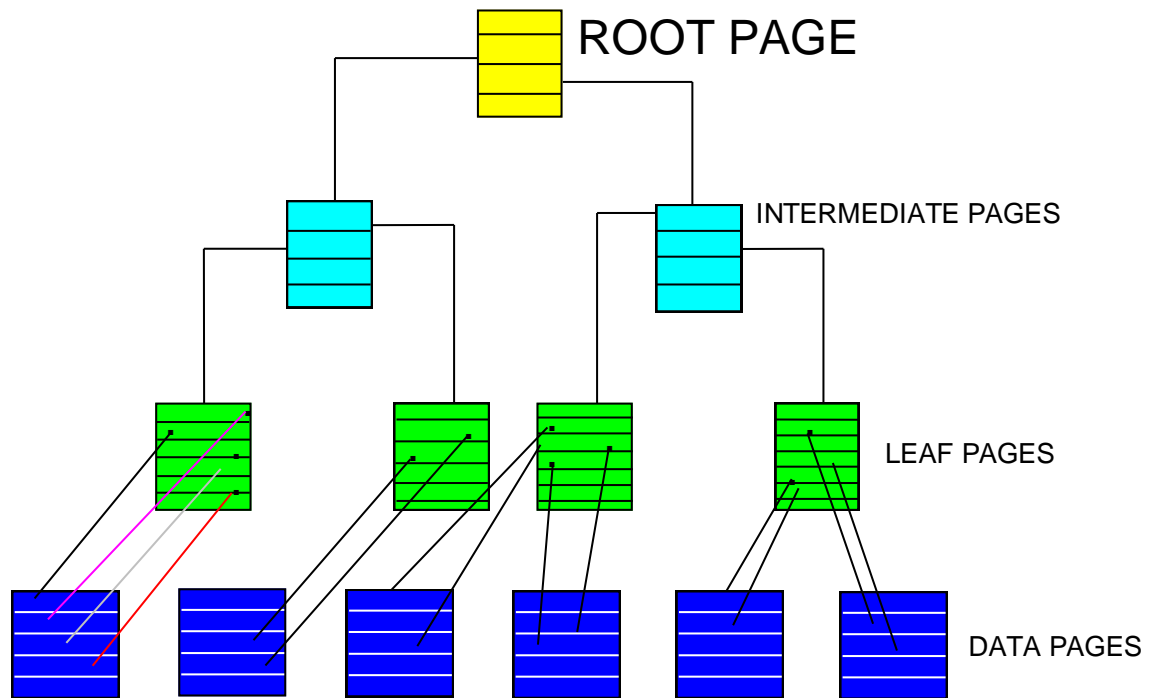
Clustering tables are used for:

- JOIN operations
- reduced storage requirements
- 'clustering' of key row values

### CLUSTER BLOCK

DEPTNO	
01	MYDEPT table row
01	MYEMP table row
01	MYEMP table row
02	MYDEPT table row
02	MYEMP table row
02	MYEMP table row

## CLUSTERED INDEXES



### NOTES:

1. Rows from more than one table are grouped in order in a data page based upon the Cluster Index
2. The leaf level contains pointers to the rows on the data page
3. The pointers add a level between the index and the data

## CREATING A CLUSTER INDEX

**Task:** Create the CLUSTER INDEX for the clus\_dept CLUSTER and ORG and STAFF tables

**\*\*\*\*Important:** You must have created the cluster before you can create the cluster index.

**SQL:**

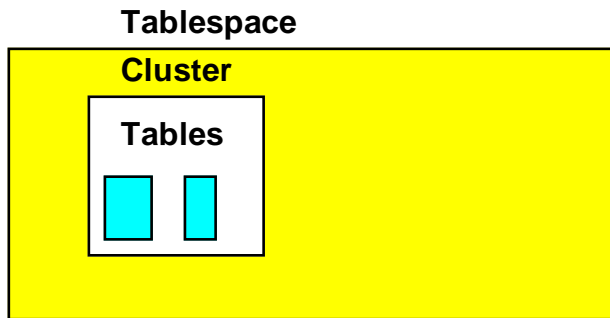
```
CREATE INDEX staff_deptindx  
on CLUSTER clus_dept
```

**Notes:**

To create the cluster, tables, and indexes you must have the appropriate privileges to do so. You must have either UNLIMITED TABLESPACE system privilege or CREATE ANY CLUSTER system privilege. To create a table you must have CREATE TABLE privilege or be a system administrator. To create indexes you must have the CREATE INDEX privilege at the very least.



## CREATING CLUSTERS



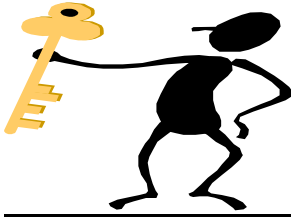
Clusters are created independently from your base tables

- Based upon common key from base tables
- Base tables clustered column must be NOT NULL
- Need to create an index on cluster key
- Typically created for empty tables

### *Format*

```
Create cluster cluster-name  
(cluster_key1 datatype, cluster_key2 datatype,...)  
[space space_name]  
{size logical_block_size}  
[compress |no compress]
```

## EXAMPLES:



1.) CREATE CLUSTER CUST\_CLUS (custid number(4))  
- custid is the cluster key column

2.) CREATE INDEX custndx  
on CLUSTER CUST\_CLUS

3.) CREATE TABLE CUSTOMER  
CLUSTER CUST\_CLUS (custid)  
AS  
SELECT \* FROM CUST

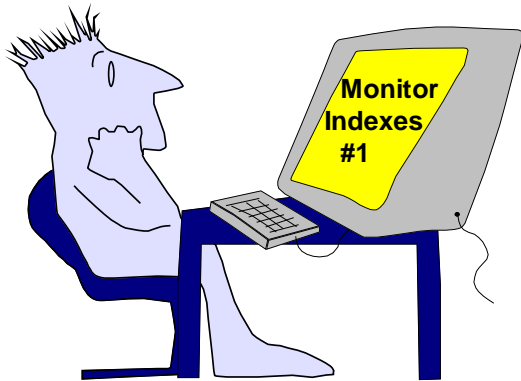
CREATE TABLE ORDERS  
CLUSTER CUST\_CLUS(custid)  
AS  
SELECT \* FROM ORD

3.) If you do not need the previous unclustered tables and want the same names:

DROP TABLE CUST  
RENAME CUSTOMER TO CUST  
DROP TABLE ORD  
RENAME ORDER TO ORD

4.) You cannot query a cluster, only the base tables

## MONITORING INDEX SPACE IN BLOCKS



Index space usage can be examined using several methods. Lets assume we have an primary key index called YEIDX which we want to evaluate:

```
SQL> ANALYZE INDEX YEIDX VALIDATE STRUCTURE
```

Index analyzed.

```
SQL> SELECT HEIGHT, BLOCKS, NAME, DISTINCT_KEYS,  
MOST_REPEATED_KEY MRP,PCT_USED FROM SYS.INDEX_STATS  
WHERE NAME = 'YEIDX';
```

HEIGHT	BLOCKS	NAME	DISTINCT_KEYS	MRP	PCT_USED
2	130	YEIDX	10000	1	88

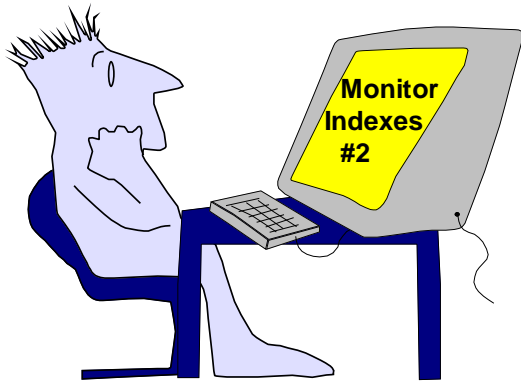
1 ROW SELECTED

```
SQL> DROP INDEX YEIDX;  
SQL> CREATE INDEX YEIDX ...;
```

The method above should be performed several times to establish a base average efficiency of space usage. Once this base average is calculated it should be compared against a new percentage. When the new percentage falls below the base average, then drop and recreate to conserve space.

ORACLE stores the information from method one in a SYS view which is based on dynamic tables, therefore, the view is initialized every time a new session begins.

## MONITORING INDEXES



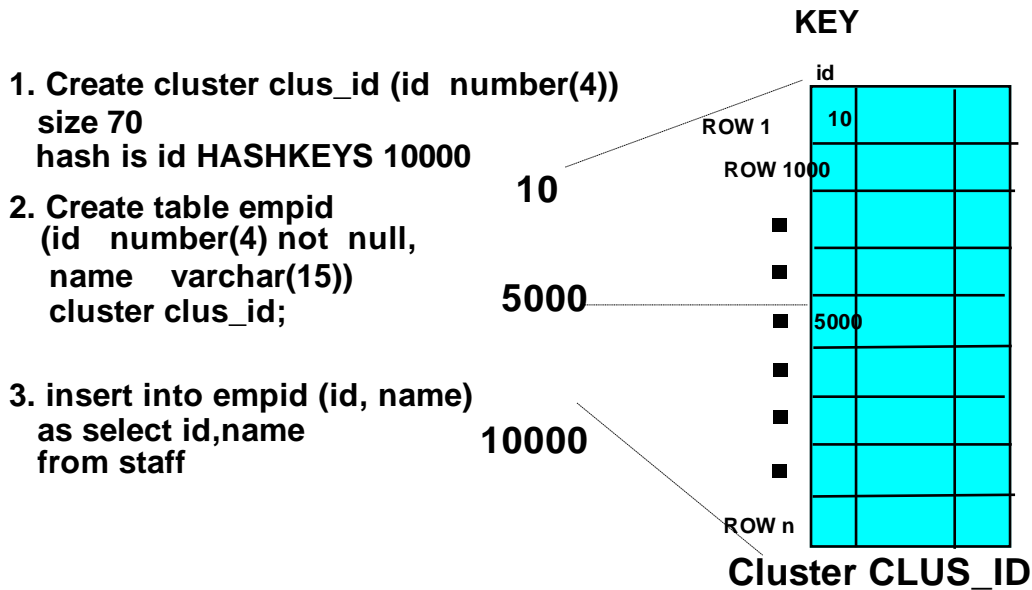
Method two for monitoring indexes can be performed to gather certain information that is not generated in method one, e.g. index level, average number of leaf blocks/key, average number of data blocks/key, clustering factor, minimum key value, and maximum key value.

ORACLE stores the information from method two in several data dictionaries:

- ALL\_INDEXES
- DBA\_INDEXES
- USER\_INDEXES

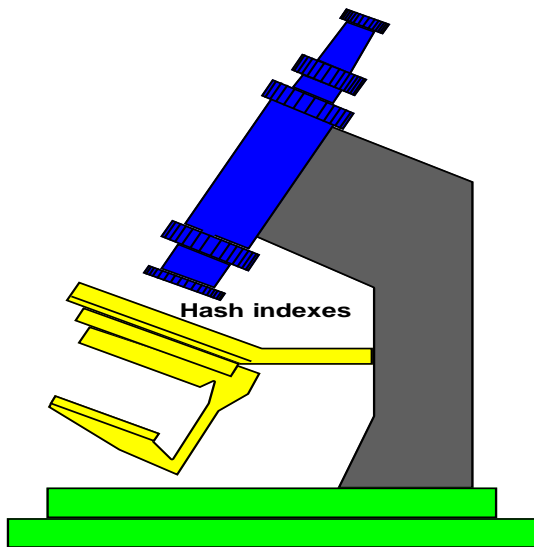
Information for a particular index in these data dictionary views are regenerated every time the ANALYZE INDEX command is executed

# HASH CLUSTERS



- A hash cluster is a way of storing and accessing table data
- Applies a hash function to one or more columns
- Rows are stored as a result of the hash function
- Hashing typically reduces the number of I/Os to return data blocks
- Rows are stored as a result of the hash function
- Used when it is acceptable to pre-allocate storage space for a table
- Used when query optimization is very important
- Used when the hashed columns are frequently referenced in the where clause with an equal sign (=)
- Tables must be clustered before they are hashed

# ANALYZING A HASH CLUSTER



Hash clusters can be used in place of indexes many times because their search for a key is done through an algorithm versus a key scan. To ensure your hash index is working properly you can use the following steps.

```
ORACLE> Analyze cluster clus_id compute statistics;
```

```
ORACLE>  SELECT CLUSTER_NAME,KEY_SIZE,
              AVG_BLOCKS_PER_KEY  BLOCKS,
              CLUSTER_TYPE,FUNCTION,
              HASHKEYS  HASH
              FROM USER_CLUSTERS;
```

CLUSTER_NAME	KEY_SIZE	BLOCKS	CLUS	FUNCTION	HASH
CLUS_ID	70	50	HASH	DEFAULT	10000

# SORTED HASH CLUSTERS

- **New data structure used to store data sorted by nonprimary key columns:**
  - **Cluster key values are hashed.**
  - **Rows corresponding to a particular cluster key value are sorted according to the sort key.**
- **Used to guarantee that row order is returned by queries without sorting data:**
  - **Rows are returned in ascending or descending order for a particular cluster key value.**
  - **ORDER BY clause is not mandatory to retrieve rows in ascending order.**

## **Sorted Hash Cluster: Overview**

When the Oracle database stores data in a heap-organized table, the rows are not stored in a user-controlled order. Rather, the decision about where to place a row is dependent on storage heuristics. The Oracle database does not guarantee the return order of the rows unless the query includes an ORDER BY clause. Inside a sorted hash cluster, the rows are organized in lists of sorted rows. Each list corresponds to a particular value of the hash key columns defined by the corresponding sorted hash cluster. Within each list, the rows are stored in the order specified by the sort key columns defined by the corresponding sorted hash cluster. This is also the default return order when querying the table by using the hash key columns in the predicate.

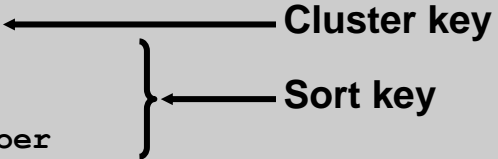
Sorted hash clusters offer the benefit of eliminating the CPU time and private memory needed to sort the data for queries that require a guaranteed returned order between SQL statements.

When querying data in a sorted hash-clustered table by cluster key columns with an ORDER BY clause that references only the sort key columns or one of their prefixes, the optimizer avoids the sorting overhead because the rows are returned sorted by the sort key columns. However, for the same kind of queries, if you have an ORDER BY clause on a suffix of the sort key columns or nonsort key columns, additional sorting is required, assuming that no indexes are defined on the table. For this reason, when you create a sorted hash cluster, select its order key columns carefully.

## IMPLEMENTING SORTED HASH CLUSTERS

```
CREATE CLUSTER calls_cluster
( origin_number  NUMBER
, call_timestamp NUMBER SORT
, call_duration  NUMBER SORT)
HASHKEYS 10000
SINGLE TABLE HASH IS origin_number
SIZE 50;

CREATE TABLE calls
( origin_number  NUMBER
, call_timestamp NUMBER
, call_duration  NUMBER
, other_info     VARCHAR2(30))
CLUSTER calls_cluster(
origin_number, call_timestamp, call_duration
);
```

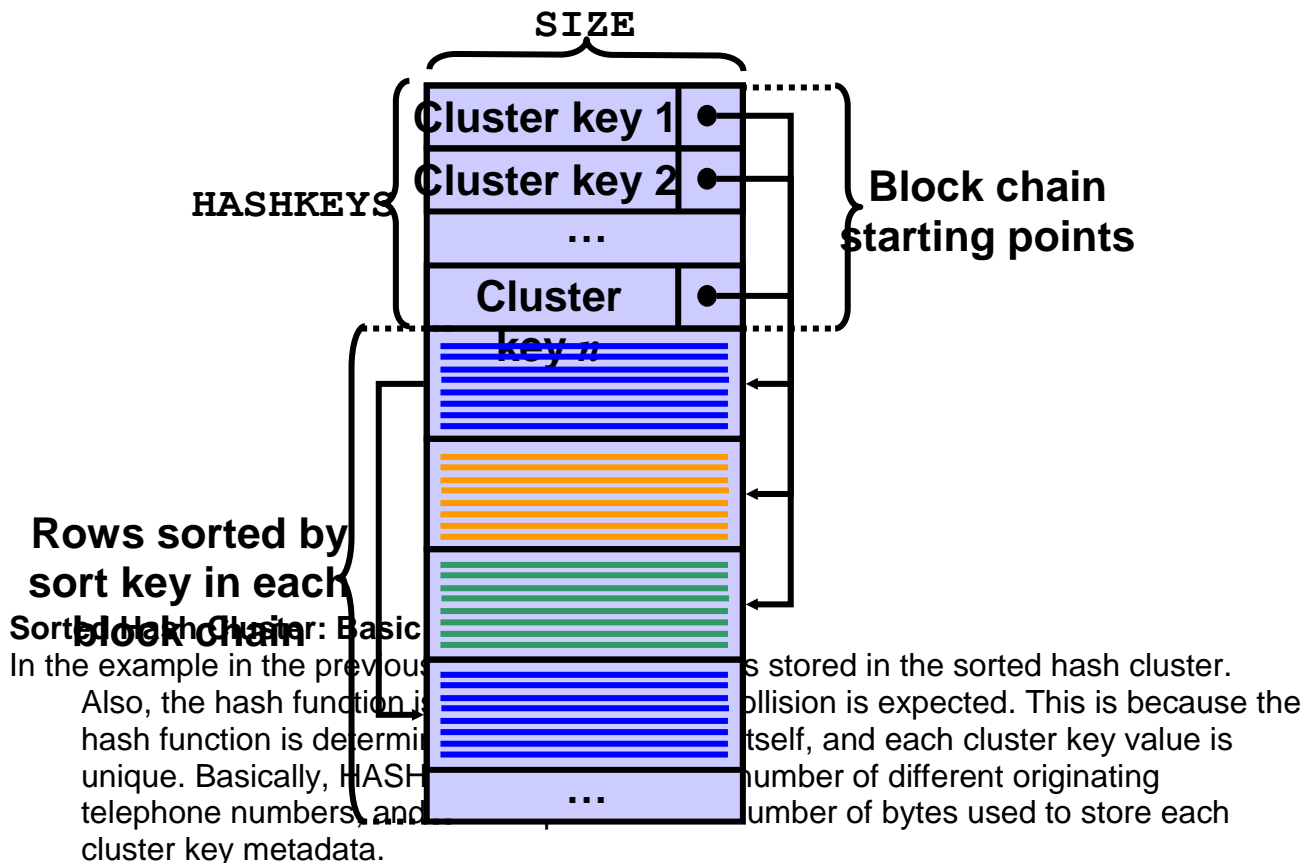


ORIGIN\_NUMBER, and the sort key columns are CALL\_TIMESTAMP and CALL\_DURATION. Another difference with traditional hash clusters is that for sorted hash clusters, the SIZE parameter specifies how many metadata entries to store for a particular hash key value. The size of one metadata entry is mainly determined by the size of the cluster key columns.

In the second step, you create the actual table specifying the link to the sorted hash cluster with the CLUSTER clause. You must specify the cluster key columns in the correct order followed by the sort key columns in the correct order. The example in the slide represents the following scenario: A telecommunications company needs to store call records for a fixed number of originating telephone numbers through a telecommunications switch. From each originating telephone number, there can be an unlimited number of telephone calls. Calls are stored as they are made and processed later in a “first-in, first-out” order when bills are generated for each originating telephone number. Each call is identified by a time-stamp number.



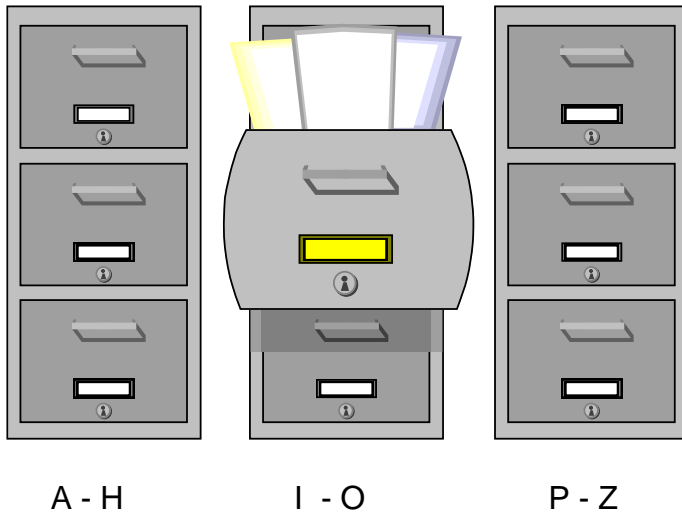
## SORTED HASH CLUSTER SCHEMATIC



As you can see, the first part of the sorted hash cluster segment is reserved to store the metadata entries. Each metadata entry contains a link to the list of its corresponding rows. Each list is made up of a series of Oracle blocks that are linked together. Each list is sorted according to the sort key columns.

Whenever you want to retrieve the rows for a corresponding cluster key value, the cluster key value is hashed to its metadata entry location, which gives the sorted list of rows that you are selecting.

# PARTITIONED TABLES



Partitioned tables are used independently of partitioned views which came into being in version 7.3 of Oracle's database. You can either create separate tables with the same structure or one table to hold all the partitioned data as shown.

```
ORACLE> CREATE TABLE SURNAME_A_H
(SURNAME VARCHAR2(20),
CITY      VARCHAR2(20));
ORACLE> CREATE TABLE SURNAME_I_Q
AS SELECT * FROM SURNAME_A_H;
ORACLE> CREATE TABLE SURNAME_R_Z
AS SELECT * FROM SURNAME_A_H;
ORACLE> ALTER TABLE SURNAME_A_H
ADD constraint con1 check(upper(substr(surname,1,1)) between 'A' and 'H');
ORACLE> ALTER TABLE SURNAME_I_Q
ADD constraint con1 check(upper(substr(surname,1,1)) between 'I' and 'Q');
ORACLE> ALTER TABLE SURNAME_R_Z
ADD constraint con1 check(upper(substr(surname,1,1)) between 'R' and 'Z');
ORACLE>create view surname as
select * from surname_a_h
union all
select * from surname_i_q
union all
select * from surname_r_z;
```

or

## PARTITIONED TABLE WITH ONE CREATE



### ONE PARTITIONED TABLE

```
ORACLE> CREATE TABLE SURNAME_PART
(SURNAME VARCHAR2(15),
CITY      VARCHAR2(20))
PARTITION BY RANGE (SURNAME)
(PARTITION SURNAME_A_H VALUES LESS THAN ('H'),
PARTITION SURNAME_I_Q VALUES LESS THAN ('Q'),
PARTITION SURNAME_R_Z VALUES LESS THAN ('Z'),
PARTITION OTHER VALUES LESS THAN (MAXVALUE));
```

With this table, you can see how partitioned tables come into their own. Normally all of the data would have to be stored in a single table. By using partitions, you can store different parts of the data in different places – i.e., different tablespaces,. Partitioning has these advantages:

- Reduces the possibility of data corruption
- Increases performance by spreading data over different disk drives
- Permits independence of backups
- Allows the database to support large tables
- Allow you to support highly intensive queries against your data.

## **TYPES OF PARTITIONING**

The example shown above utilized a partitioning type known as RANGE partitioning.

Oracle supports two other types of partitions (HASH and LIST). Also, within partitioning, a sub-partition can exist. Sub-partitions also referred to as composite partitions, can be either HASH or LIST.

## **RANGE**

Range partitions are the most common. Table and index partitions are based on a list of columns allowing to the database to store each occurrence in a given partition. These partitions are typically used within data warehousing systems. The most common range boundary is based off of dates.

Each partition is defined with an upper boundary. The storage location of each occurrence is then found by comparing the partitioning key of the occurrence with this upper boundary. This upper boundary is non-inclusive; in other words, the key of each occurrence must be less than this limit for the record to be stored in this partition.

## **HASH**

Hash partitions are ideal when there is no real method to divide a table based on a range. Hash partitions utilize a hashing algorithm to programmatically take a column value and store that value within a given partition. Each partition is defined with an upper boundary. The storage location of each occurrence is then found by comparing the partitioning key of the occurrence with this upper boundary. This upper boundary is non-inclusive; in other words, the key of each occurrence must be less than this limit for the record to be stored in this partition. This type of partitioning is recommended when it is difficult to define the criteria for the distribution of data.

## **LIST**

List partitions have a hard-coded LIST of values that will exist within any partition. A common usage would be with states. A state partition table would commonly have 50 partitions, one for each state.

SUB-Partitions are utilized most often when the partition strategy does not provide small enough partition units to achieve maintenance goals. When this is true, sub-partitions can further divide a table based another column.

# Examples



RANGE: A max partition will capture any values beyond the stated ranges – including NULLS

```
Create table range_partition  
( date_col date)  
partition by RANGE (date_col)  
(  
partition p_jan_2001 values less than (to_date('01022001','ddmmyyyy')),  
partition p_feb_2001 values less than (to_date('01032001','ddmmyyyy')),  
partition pmax values less than (maxvalue)  
);
```

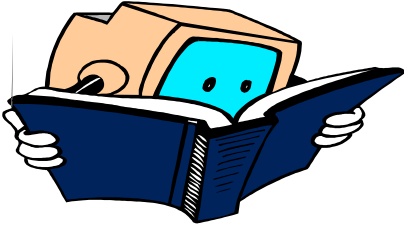
HASH – Hash partitions are most optimal when 8, 16, or 32 partitions are used.

```
Create table hash_partition  
(account_id varchar2(30))  
partition by HASH (account_id) partitions 16
```

LIST

```
Create table list_partition  
(state_id varchar2(2))  
partition by LIST (state_id)  
(  
partition P_MI values ('MI'),  
partition P_CO values ('CO')  
);
```

## LAB 5 - MORE ABOUT CREATING TABLES AND INDEXES



Create 7 tables in the appropriate tablespace. The seven tables are called: **PRESIDENT, ELECTION, ADMIN\_PR\_VP, ADMINISTRATION, PRES\_HOBBY, PRES\_MARRIAGE, STATE.** See pres.sql. The specs for these tables can be found on subsequent pages of this lab. Remember to check the sample against the specs to make sure they match up. Create the appropriate Primary key indexes for these tables. Find the foreign key column in the ADMINISTRATION table which references PRESIDENT and create a foreign key (RI) on it . (HINT: These tables must be empty before creating (RI) constraints.)

### Table Specifications:

We want to collect these items of data about presidents of the United States into these tables:

ITEM	TYPE	LENGTH	NULLS ALLOWED
=====			
PRESIDENT NAME	CHAR	20	NO
YEAR BORN	NUMBER(4)		NO
NUMBER YEARS SERVED	NUMBER(4)		NO
AGE AT DEATH	NUMBER(4)		YES
PARTY	CHAR	20	NO
STATE BORN	CHAR	20	NO
STATE	CHAR	20	NO
YEAR STATE ENTERED U.S.	NUMBER(4)		NO
ADMINISTRATION STATE ENTERED	NUMBER(4)		YES
ADMINISTRATION NUMBER	NUMBER(4)		NO
YEAR INAUGURATED	NUMBER(4)		NO
VICE-PRESIDENT NAME	CHAR	20	NO
PRESIDENT'S HOBBIES	CHAR	20	YES
PRESIDENT'S SPOUSE	CHAR	20	YES
YEAR MARRIED	NUMBER(4)		YES
PRESIDENT'S AGE AT MARRIAGE	NUMBER(4)		YES
SPOUSE'S AGE AT MARRIAGE	NUMBER(4)		YES
NUMBER OF CHILDREN	NUMBER(4)		YES
ELECTION YEAR	NUMBER(4)		NO
CANDIDATE	CHAR	20	NO
VOTES	NUMBER(4)		NO
WINNER/LOSER INDICATOR	CHAR	1	NO

## LAB 5 - MORE ABOUT CREATING TABLES AND INDEXES



### TABLE COLUMN NAMES AND RELATIONSHIPS

The logical data base design process suggest tables with the following columns:

#### **PRESIDENT:**

pres\_name, birth\_yr, yrs\_serv, death\_age, party, state\_born

#### **ADMINISTRATION:**

admin\_nr, pres\_name, year\_inaugurated

#### **ADMIN PR VP:**

admin\_nr, pres\_name, vice\_pres\_name

#### **STATE:**

state, admin\_entered, year\_entered

#### **PRES HOBBY:**

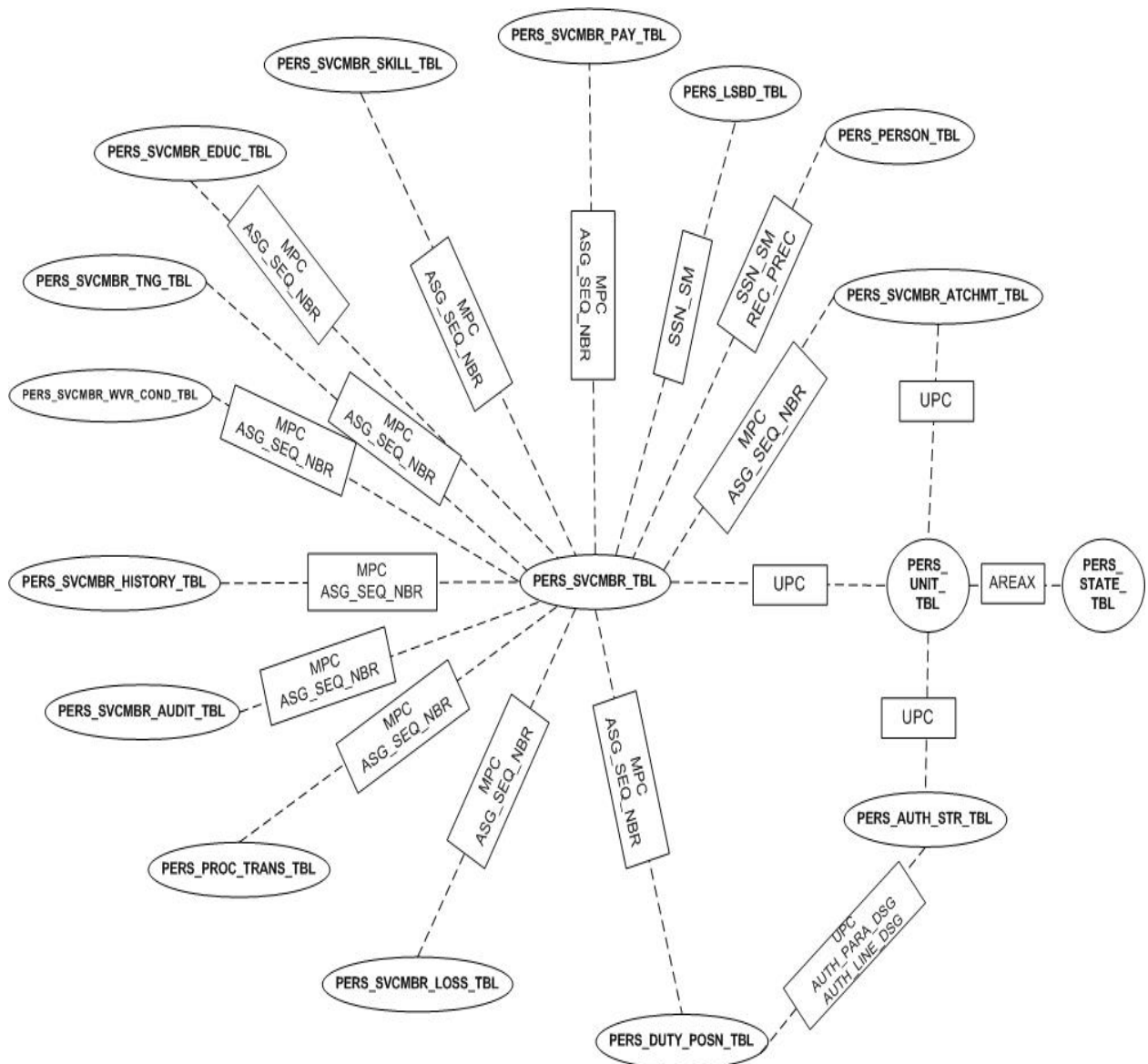
pres\_name, hobby

#### **PRES MARRIAGE**

pres\_name, spouse\_name, pr\_age, sp\_age, nr\_children, mar\_year

#### **ELECTION:**

election\_year, candidate, votes, winner\_loser\_indic



This is the Data Model for the Sidpers schema. You will be creating different indexes and compare the performance between the different tables. Before executing the queries on the next page, clear the buffer cache of data so the queries are equivalent.

Log in as sidpers:

```
SQL> ALTER SYSTEM FLUSH BUFFER_CACHE;
```



## LAB 5 - MORE ABOUT CREATING TABLES AND INDEXES



### PRESIDENT TABLE QUERIES

1. Insert your name into the PRESIDENT table. Execute uinsert.SQL to load the tables. No changes are needed to these inserts. See: uinsert.SQL.
2. Execute 'SELECT' statements to verify the data exists in each of your tables.
3. List the president's name, year born, year married, and spouse's name of all married presidents. This is a join of 2 tables. President and Pres\_Marriage. Use a Natural Join.
4. Create a non unique index on the candidate column in the Election table..
5. Create a view which contains the President's name, his spouse and any hobbies he may have had. This is a join of 3 tables..
6. Create exact duplicate tables of the following tables in the sidpers schema using CTAS (Create table as select command).  
Pers\_person\_tbl will be called person  
Pers\_svcmbtr\_tbl will be called svcmbtr  
pers\_unit\_tbl will be called unit.  
These tables will have no indexes on them.
  - a. Perform a join of the three tables (using the above data model to identify keys) and use AUTOTRACE to see what kind of performance in cpu, I/O and cost you achieve.
  - b. Create unique indexes on the person.table (ssn\_sm and rec\_prec)
  - c. Create non unique indexes on the (svcmbtr) table on the ssn\_sm column.
  - d. Create unique indexes on the svcmbtr table on the mpc and asg\_seq\_nbr columns.
  - e. Create non unique indexes on the up column in the svcmbtr tbl.
  - f. Create unique indexes on the unit table on the upc column.
  - g. Clear the buffer cache so that no data is in them  
SQL> alter system flush buffer\_cache;
  - h. Now run the same join as above and identify performance benefits.

7. Create a join of the person and svcnbr table where the user requires the ssn\_sm, rec\_prec, name\_ind, states\_us, dob and apft\_score (physical fitness score) from the person table and requires the mpc, asg\_seq\_nbr, gr\_abbr\_code and cum\_ret\_pt for their report. Join them on with a using clause. Using Autotrace, record the costs and i/o effort
8. Create an IOT from the person table called person\_iot which includes the above data (ssn\_sm, rec\_prec, name\_ind, states\_us, dob and apft\_score) with the primary key of ssn\_sm and rec\_prec and sends eth\_gp, marlt\_stat, race\_pop\_gp, rel\_denom, sex, hgt\_ind, wt\_ind, loc\_data\_pers to an overflow tablespace called user\_data.
9. Create another IOT from the svcnbr table called svcnbr\_iot which includes the mpc, asg\_seq\_nbr, ssn\_sm, rec\_prec, upc, gr\_abbr\_code, cum\_ret\_pt and date\_rec\_stat with the primary key of mpc and asg\_seq\_nbr while sending the following columns (date\_asgn\_loss\_rsn, org\_ident, expir\_date\_ing, date\_init\_procrmt, retn\_wvr, date\_mand, rem, date\_end\_eval\_period, afqt\_score\_gps) to an overflow tablespace called user\_data.
10. Do the same join as above and identify cost and i/o.
11. Create an exact copy of the President table using the create table as select (CTAS) command.
12. Run the following statement before you create a bitmap index. Create a bitmap index called person\_bit on the person table which includes sex, marlt\_stat (marital status), hgt\_ind, wt\_ind and dob. This is required because of new requirements which require certain human characteristics for specific jobs.
13. Create a SPOUSE table that contains spouse name and marriage year. Load it from the PRES\_Marriage table using a CTAS.
14. Run this query which searches for an individual name using a like clause. Use SQLPLUS or SQL DEVELOPER with the autotrace on.  
SQL>select name\_ind from person where name\_ind like 'JER%'; Record the cost and type of access.  
Then create a function based index called person\_name\_ix on the person table on the name\_ind column which ensures that a user searches for the name in upper case. What are the results? Did the function-based index get used? If not, why don't you think it did?

15. Select index\_name from user\_indexes where table\_name like big; How many indexes on big and on what columns? Do the following select:  
SQL> explain plan for  
    select bname from big where bname like 'BET%'; Evaluate the plan.  
Now let's drop the index on bname. SQL> drop index big\_bname;  
Rerun the explain plan above. Does it have a different plan?  
Now Create a bitmap index on the bname column in the big table.  
Rerun the explain plan for select bname from big where bname like 'BET%';

## LAB 5 - MORE ABOUT CREATING TABLES AND INDEXES



16. Add a column to the STATE table for state flower.
17. Create a table for flowers containing columns for flower name and color. Insert your own state flower data.
18. Add a column for height in the FLOWER table.
19. Delete the FLOWER table
20. Which Presidents never won an election? (Use a subquery)

### Optional Queries:

1. Summarize total votes by candidate without regard to election year. Show in descending total vote sequence.
2. Add yourself as a candidate in the election table, then using a complex insert, add yourself to the PRESIDENT table as the next president adding appropriate values as needed using data (your name) from the election table.