DATABASE ADMINISTRATION

T2: DATABASE OPTIMIZATION [C5]

Table Tuning and Index-based Optimization

CHAPTER PLAN

- 1. Physical Design and Tools
  - 1.1 Physical Design - Optimization Objectives
  - 1.2 Physical Design Process
  - 1.3 Execution Plan Tool
  - 1.4 CBO Scopes
- **2. Table and Index Access Optimization**
  - **2.1 Buffering & Table (scan) Access Opt. Techniques**
  - **2.2 Index-based Optimization Techniques**
- 3. JOIN and SORT Operation Optimization
  - 3.1 JOIN Optimization Techniques
  - 3.2 SORT Optimization Techniques
- 4. More advanced techniques
  - Transformed Subqueries: subquery refactoring
  - Subquery optimization: temporary tables and materialized views
  - Access optimization with Parallel Query

2. TABLE and INDEX-based Optimization

- TABLE Storage and Access Tuning Techniques
- INDEX based Optimization:
    - to access tables;
    - to join tables;
    - to sort rows.

2.1 TABLE Optimization Techniques

- Table buffering
- Table STORAGE and table access paths OPERATIONS:
  - Full table scan;
  - Table access by index;
  - Table access by rowid;
  - Table access by hash;
  - Table access by cluster.

Storage Type	Access Path
HEAP	FTS, TABLE ACCESS BY INDEX ROWID
IOT	INDEX SCAN
CLUSTER HASH [SINGLE TABLE]	TABLE ACCESS BY HASH
CLUSTER INDEX [MULTIPLE TABLES]	TABLE ACCESS BY CLUSTER

- Table Tuning TECHNIQUES

SQL Table Storage and Table Access Paths

- The physical design decision to choose the table storage type could be determined by some specific table access **strategies**:
  - *S1: Unordered table + primary key index + secondary indexes:*
    - Unordered tables could be stored as:
      - *Heap* storage structures;
      - *Hash* storage structures;
    - Indexing criteria could be determined by:
      - the initial primary and unique key constraints;
      - the columns coming from *equals* or *range* predicates within WHERE clauses of SELECT queries;
  - *S2: Ordered table + secondary indexes:*
    - Tables could be:
      - partially ordered: by using CLUSTER-like storage;
      - totally ordered: by using primary key index storage.

SQL Table Storage and Table Access Paths
When to use: heap, hash, cluster or index storage types

- When to use HEAP (unordered) storage:
  - massive data workloads occurs immediately after creation or at scheduled points in time;
  - the final table size is relatively small;
  - the access probability is the same for each and every table row;
  - most frequently access requests could be easily covered by additional indexes.
- When to use HASH (unordered) storage:
  - the access requests are very stable and are based on equals-like predicates;
  - hash-result values produced by hash-functions are highly controllable to accurately predict memory-page count number (the number of data blocks);
- When to use CLUSTER-ordered storage:
  - to pre-determine aggregation of row groups for aggregation queries;
  - to pre-determine JOIN-ing row groups (physical denormalization instead of logical denormalization) of related tables;
- When to use INDEX-ordered storage:
  - primary-key index is the main access criteria;
  - to pre-determine row ordering on final resultset.

PL/SQL Hash Function Example

```
CREATE OR REPLACE
FUNCTION Get_Hash_Value(key_value VARCHAR) RETURN Number IS
  l_sum number := 0;
  l_tmp number := 0;
begin
    FOR i in 1..length(key_value) LOOP
      l_tmp := ascii(substr(key_value,i,1));
      l_sum := l_sum + l_tmp;
    END LOOP;
    return mod(l_sum,13);
END;
/
```

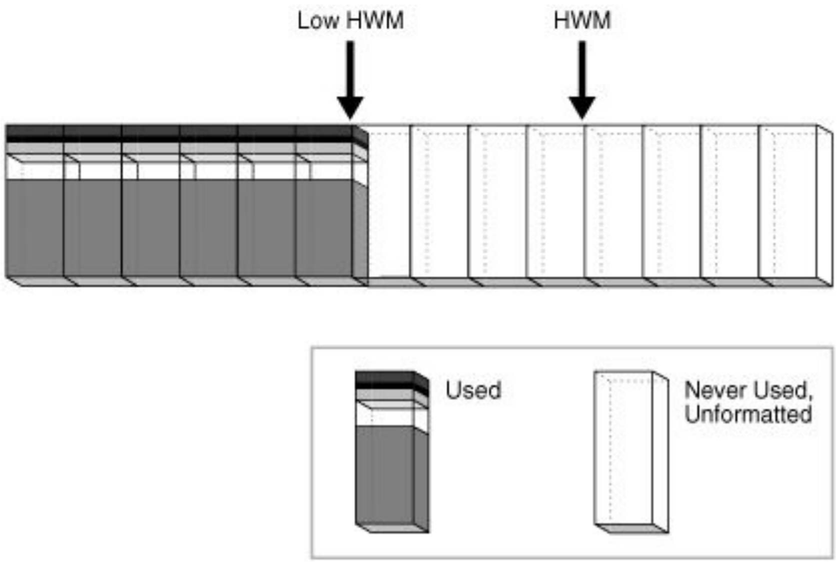
Table Scan Operations

- Table access methods (OEP-operations):
  - table scan;
    - TABLE ACCESS FULL;
    - TABLE ACCESS SAMPLE;
  - access by rowid:
    - TABLE ACCESS BY USER ROWID;
    - TABLE ACCESS BY RANGE ROWID;
  - access by index:
    - TABLE ACCESS BY INDEX ROWID;
  - partitioning access for TABLE ACCESS FULL:
    - PARTITION RANGE SINGLE
    - PARTITION RANGE ITERATOR
  - full tables scan with PARALLEL QUERY

SQL Query blocks determining FULL TABLE SCAN Operations [FTS]

- WHERE clause is missing (no filter predicate);
- WHERE clause with NULL condition;
- WHERE clause with predicates against unindexed columns;
- WHERE clause with LIKE condition (patterns like '%...');
- WHERE clause with NOT equals condition;
- WHERE clause with BIF (built-in-functions) invalidating index;
- **OEP hints** to enable table scan (full):
  - ALL_ROWS hint;
  - PARALLEL hint;
  - FULL(table_name | alias) hint;

FTS up to: **HIGH WATER MARK** indicator



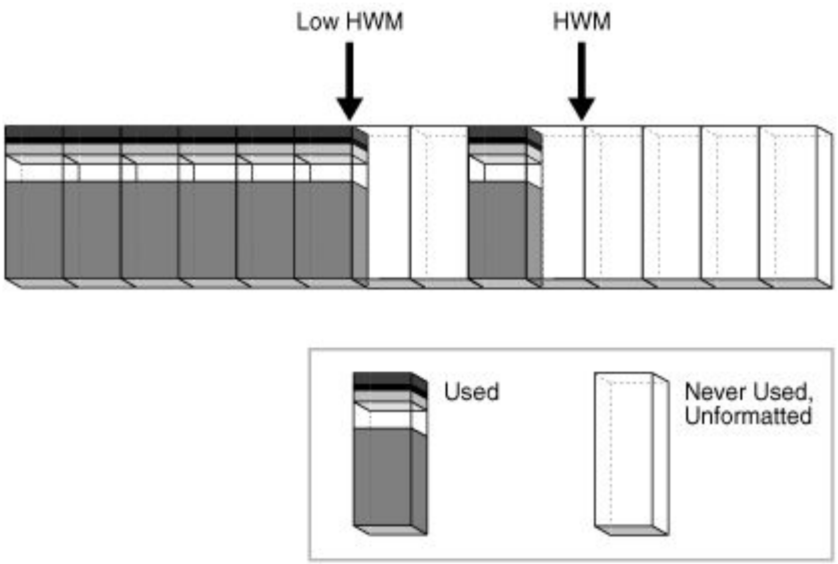


Table Access Tuning TECHNIQUES

Tuning Technique	Table access operation	Notes
KEEP buffer	FTS (in memory)	Row resequencing and de-fragmentation to HWS
PARTITION ACCESS	PARTITION RANGE SINGLE PARTITION RANGE ITERATOR	Row resequencing and partition
Change access mode: access by ROWID	TABLE ACCESS BY INDEX ROWID	
Change access mode: access by ROWID	TABLE ACCESS BY USER|RANGE ROWID	
Enable PARALLEL mode	Enable PARALLEL mode: PX	Multi-block I/O parameter

Practice C5_P1

Steps	Notes	SQL Script
1. HWM	High Water Mark	C5_P1.1.TABLE_ACCESS_HWM_FTS.sql
2. FTS Use Cases	OEP Hints: + FULL + PARALLEL + NOCACHE + CARDINALITY	C5_P1.1.TABLE_ACCESS_HWM_FTS.sql
3. Tuning: KEEP buffer	STORAGE (BUFFER_POOL KEEP)	C5_P1.2.TABLE_SCAN_Tuning.sql C5_P1.3.Base_Schema_CREATE_TABLE_KEEP.sql
4. Tuning: PARTITION-ing	PARTITION SCAN	C5_P1.2.TABLE_SCAN_Tuning.sql C5_P1.4.Base_Schema_CREATE_TABLE_STORAGE_RELOCATE.sql

Practice C5_P1

Steps	Notes	SQL Script
5. Tuning: change access mode to INDEX access	TABLE ACCESS BY INDEX ROWID	C5_P1.2.TABLE_SCAN_Tuning.sql
6. Tuning: change access mode to ROWID access	TABLE ACCESS BY RANGE ROWID	C5_P1.2.TABLE_SCAN_Tuning.sql
7. Tuning: reset HWT by re-formatting table storage	ALTER TABLE MOVE TABLESPACE DBMS_REDEFINITION.START_REDEF_TABLE	C5_P1.3.Base_Schema_CREATE_TABLE_STORAGE_RELOCATE.sql C5_P1.5.Base_Schema_ALTER_TABLE_STORAGE_RELOCATE.sql

2.2 INDEX-based optimization

- Index Design Process and Guidelines.
- Index-based OPERATIONS (from explain plan):
  - B*Tree vs. Bitmap Specific operations
- Index-based tuning TECHNIQUES.

Standard INDEX Design FACTORS to optimize SELECT Queries

- **Main factors** (and their relative importance) that **affect index design**:
  - *(1*) filter predicates* (within WHERE clause) could reduce the number of scanned index segments (index slices):
    - through those filter-based columns evaluated as *matching columns*;
  - *(2*) ordering criteria* (ORDER BY clause) included in the index-key structure could remove sorting operations executed after the table data access operations;
  - *(3*) columns determining the resultset structure of data query* (SELECT column clause) included in the index-key structure could remove all table access operations needed:
    - “fat” indexing consequences need to be evaluated.

Standard INDEX Design PROCESS to optimize SELECT Queries

- *1. Analyse WHERE predicates:*
  - *1.1 To choose the columns from equals predicates* [col_name=:value_expression] to be added as first-key index column(s) (choosing matching columns);
  - *1.2* To choose the most selective predicate of *range type* [>, <, >=, <=, BETWEEN] to be added as next-key column(s) in the index structure.

- *2. Choose columns from ORDER BY* clause (taking in consideration also the DESCENDING sub-clause) to complete the index-key structure.

- *3. Add SELECT projection columns* to finalize the index-key structure, without some mandatory order.

Other factors that could affect index design

- Filter/Selection operators from complex predicates like ***range-type predicates*** (containing operators like >, <, >=, <=, or >|>=…<|<= combinations, or BETWEEN) :
    - could conflict with sorting criteria:
        - column order from ORDER BY not matching with the column priority derived from WHERE-predicates;
    - AND operator that link range-predicates could multiply index-slice scanning operations (multiple - index range scans);
    - OR operator that link range-predicates could produce less efficient multiple-index-access operations or even full-index-access operations.

Other factors that could affect index design

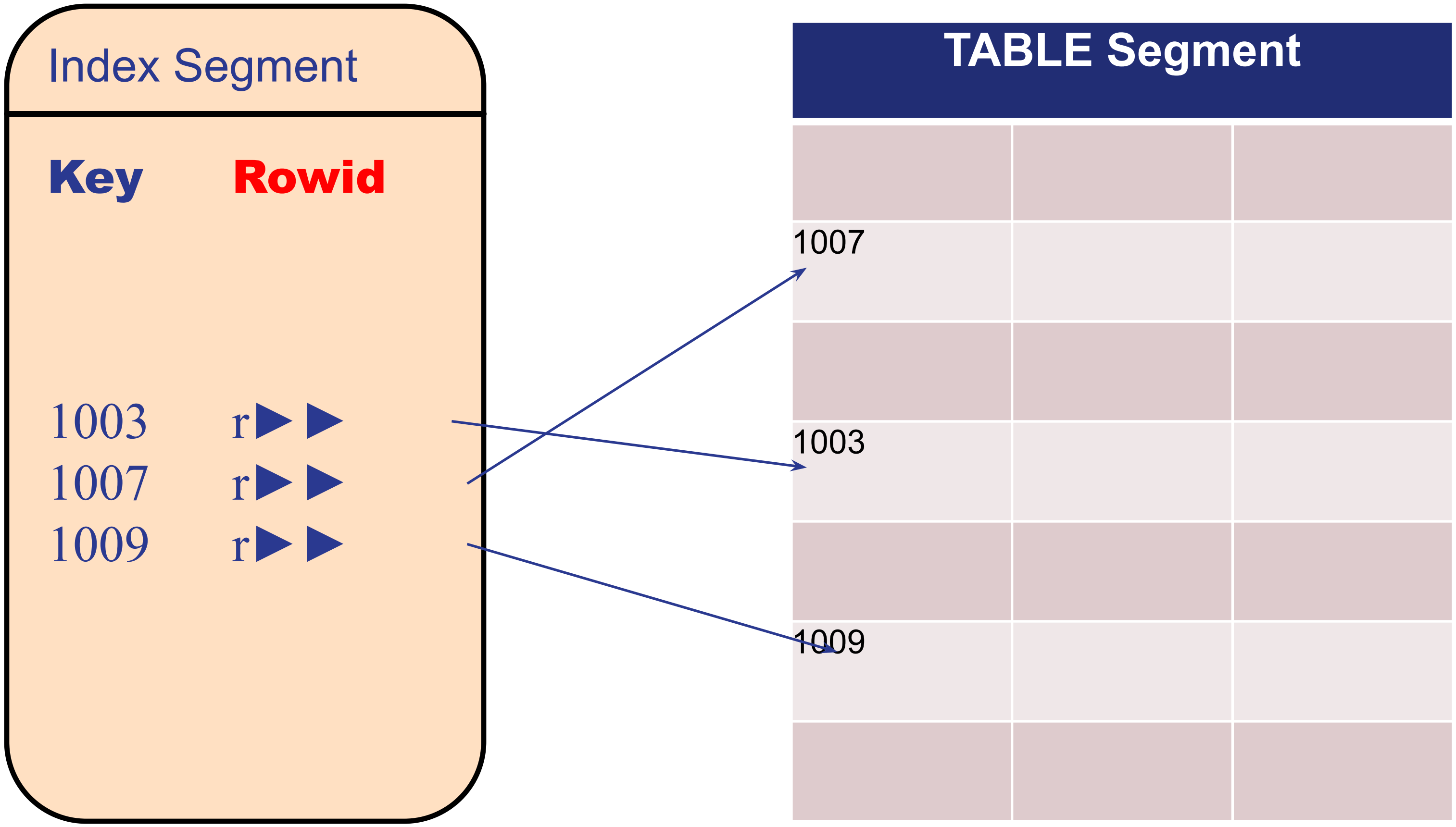
- Selection/Filter operators from complex predicates: such as ***LIKE*** with:
  - *'%XX'* search-pattern could disable a potential index scan or could determine a full index scan (instead of slice/range scan),
  - *but 'XX%'* search-pattern could determine a slice index scan.

- Selection/Filter operators from complex predicates such as ***IN(multiple-value-list)***:
  - could determine multiple - index range scan operations;
  - if there are several *multiple - index range scan* predicates the most selective one could be chosen to determine the first column of the index-key structure.

Other factors that could affect index design

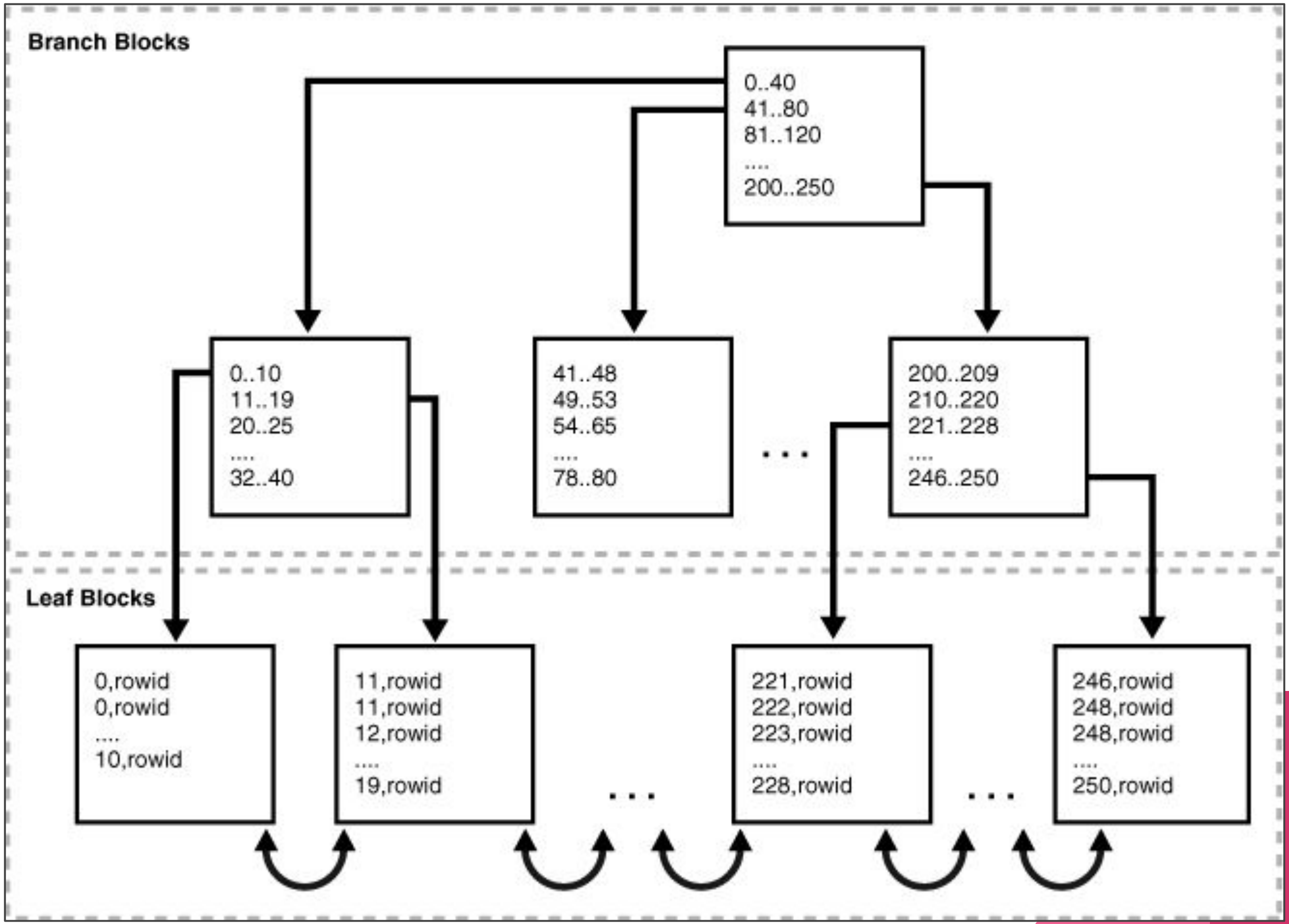
- Not indexable predicates (not “matching predicates”), e.g.:
  - [Col_a] CONCAT [Col_b] > [val];
  - [Col_a] **NOT** BETWEEN [val_1] AND [val_2];
  - [Col_a] LIKE [%val%];
- Nested-JOIN predicates (and the ON clause):
  - *Nested-JOIN* algorithm assumes scanning of the *driving table* (outer table) so that the probe table (inner table) will be iteratively scanned for each source-row from driving table;
  - some WHERE predicates on driving table could reduce the iterative scan operations on probe table, thus the indexed predicates will (indirectly) optimize (through a slice/range index scan) the whole NL process;
  - the actual JOIN-ON predicate analysis could suggest some indexing criteria for probe table (e.g. foreign-key based predicates) so that NL to be more efficiently probed not on actual table but on some index defined on the inner table.

INDEX ROLE



Index Segment
Key
Rowid
1003 r▶ ▶
1007 r▶ ▶
1009 r▶ ▶
TABLE Segment
1007
1003
1009

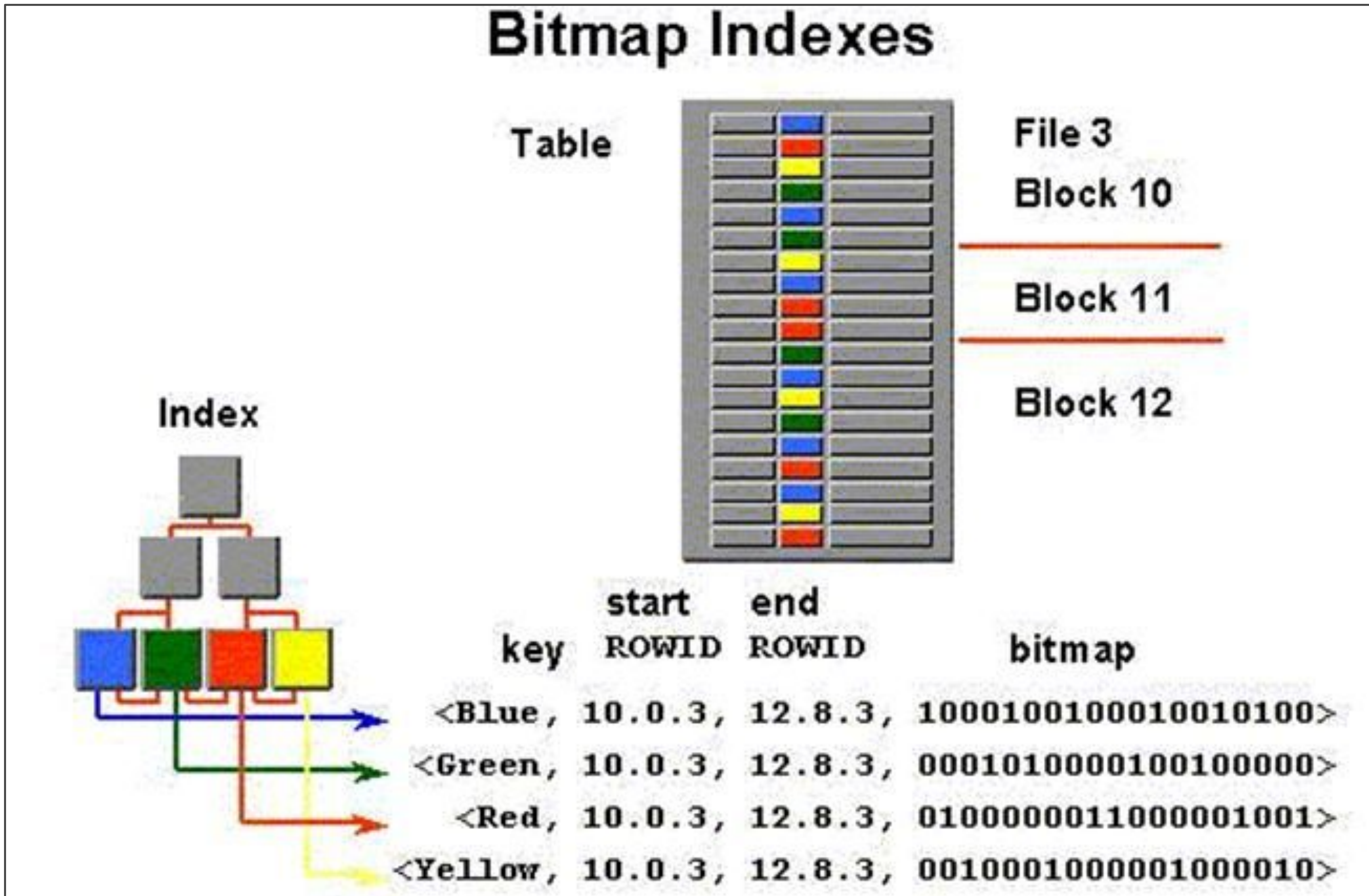
ORACLE **B-TREE**



B*TREE INDEX-based Operations

- **(preceding TABLE ACCESS BY INDEX)**
- B-tree Index Access
  - INDEX FULL SCAN
    - MIN|MAX
    - DESCENDING
  - INDEX RANGE SCAN
    - MIN|MAX
    - DESCENDING
  - INDEX SKIP SCAN
    - DESCENDING
  - INDEX UNIQUE SCAN
  - INDEX FAST FULL SCAN
  - INDEX JOIN

ORACLE **BITMAP**



BITMAP Index-based Operations

- BITMAP Index Access:
  - BITMAP INDEX SINGLE VALUE
  - BITMAP INDEX RANGE VALUE
  - BITMAP INDEX FULL SCAN
  - BITMAP INDEX FAST FULL SCAN
- Other BITMAP Index operations:
  - BITMAP CONVERSION
  - BITMAP OR
  - BITMAP AND
  - BITMAP MINUS
  - BITMAP MERGE
- *index_combine* hint is used to force a bitmap access path for the table - if multiple bitmap indexes exist.

Index-based Access optimization Techniques

SQL operation	**Index-based Tuning Technique**	**Notes**
SQL predicate: - equals predicates; - range predicates; - LIKE predicates; - FBI predicates;	CREATE [B*TREE] INDEX CREATE BITMAP INDEX	LIKE predicate invalidates index usage NULL predicate invalidates index usage BIF ON expressions or VIRTUAL columns
ORDER BY	CREATE [B*TREE] INDEX CREATE CLUSTER	Avoid SORT ORDER BY (in memory) operations
SELECT-Projection	CREATE [B*TREE] INDEX	Cumulative: predicate + order_by_criteria + projection_columns
JOIN	CREATE B*TREE INDEX CREATE BITMAP INDEX	

Practice C5_P2

Steps	Notes	SQL Script
Gather Index Stats	DBMS_STATS. GATHER_INDEX_STATS	C5_P2.1.INDEX_ACCESS_Operations_and _Stats.sql
B*TREE Index Operations	FULL SCAN RANGE SCAN UNIQUE SCAN FAST FULL SCAN	C5_P2.1.INDEX_ACCESS_Operations_and _Stats.sql
BITMAP Index Operations	BITMAP AND BITMAP MINUS BITMAP MERGE	C5_P2.1.INDEX_ACCESS_Operations_and _Stats.sql

Practice C5_P2

Steps	Notes	SQL Script
INDEX and predicates	Equals predicates Range predicates LIKE predicates BIF predicates	C5_P2.2.INDEX_Based_Tuning.sql
INDEX-based tuning for sorting	ORDER BY	C5_P2.2.INDEX_Based_Tuning.sql
INDEX-based tuning for projection	SELECT Project	C5_P2.2.INDEX_Based_Tuning.sql
INDEX-based tuning for JOINs	NL and SORT-MERGE	C5_P2.2.INDEX_Based_Tuning.sql
INDEX-based tuning for NULL predicates	B-TREE and NULLs BITMAP and NULLs	C5_P2.2.INDEX_Based_Tuning.sql
Parallel INDEX SCAN	/*+ parallel_index(t i)*/	C5_P2.2.INDEX_Based_Tuning.sql

References

- Craig S. Mullins, *Database Administration The Complete Guide to DBA Practices and Procedures* Second Edition, Addison-Wesley, 2013
- Tony Hasler, *Expert Oracle SQL Optimization, Deployment, and Statistics*, Apress, 2014
- Christian Antognini, *Troubleshooting Oracle Performance*, Apress, 2014
- Donald Burleson, *Oracle High-Performance SQL Tuning* 1st Edition, McGraw-Hill Education; 1 edition (August 17, 2001)

References

- Lahdenmaki, Tapio, Leach, Michael, *Relational database index design and optimizers: DB2, Oracle, SQL server et al*, John Wiley & Sons, 2005
- Harrison, Guy, *Oracle performance survival guide: a systematic approach to database optimization,* Prentice Hall, 2009
- Allen, Grant, Bryla, Bob, Kuhn, Darl, *Oracle SQL Recipes*: *A Problem-Solution Approach*, Apress, 2009
- Caffrey, Mellanie et.al. *Expert Oracle Practices: Oracle Database Administration from the Oak Table,* Apress, 2010

REFERENCES

- Oracle OTN Docs [link 11gR2]
- Oracle-Base Docs [link 11gR2]
- Toad World Docs [link]
- http://www.orafaq.com/tuningguide/hints.html