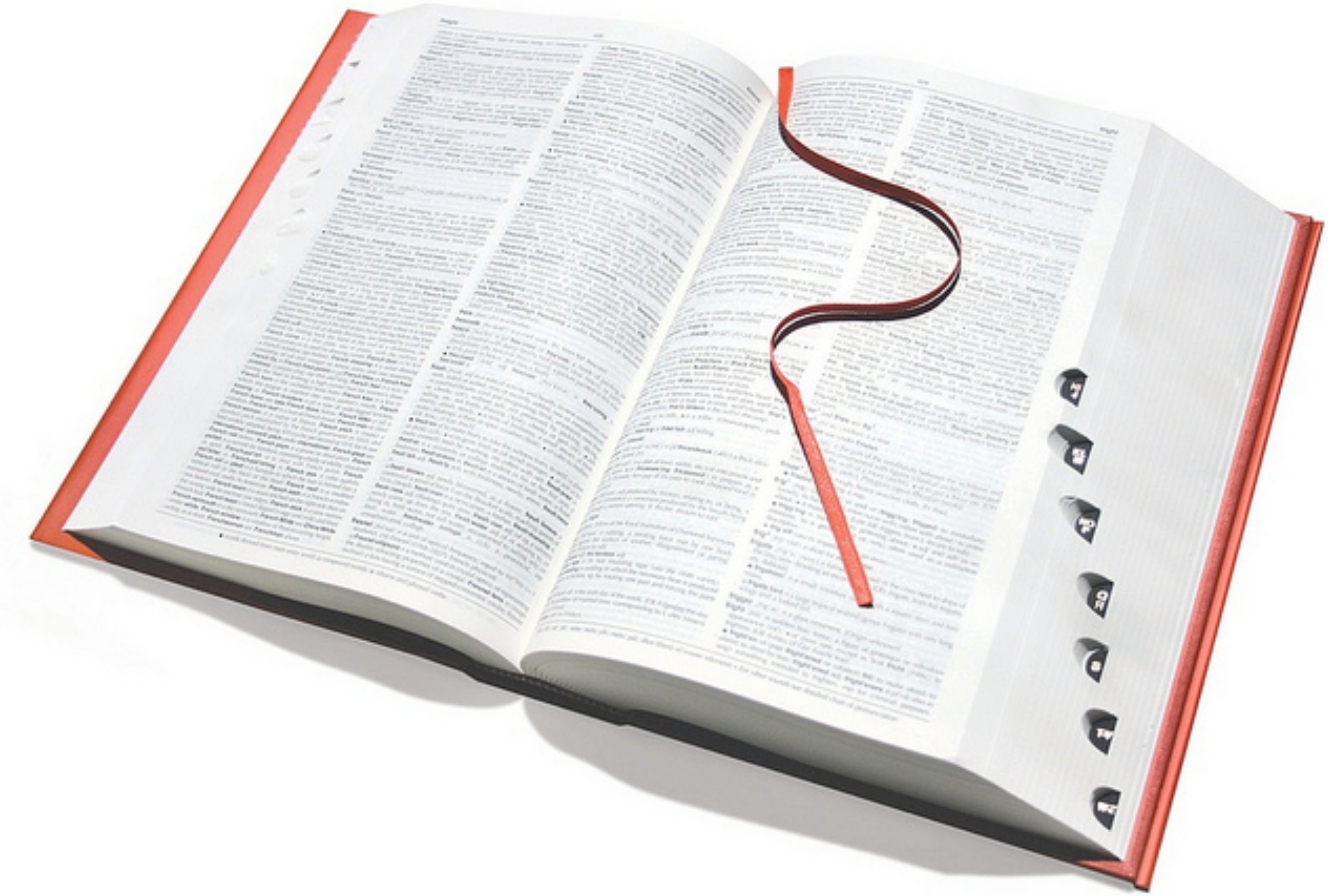
Indexing

Overview

- Indices
- Tree indices
  - B-Tree
  - Bitmap
- Composite
- Administration
- Strategy
- Query Explain

Indexing



What is an Index?

- An *index* is a <u>optional</u> data structure that allows us to map a key value to a physical location
  - Requires additional storage
  - Requires additional accesses for retrieval
  - Must be updated with database

- Provides “indirect” random access via an index table/ structure to any tuple
- Performance is enhanced if index fits into main memory

Types of Indices

- A *primary index* is based on key (unique) attributes that are used to physically order the data.
- One or more *secondary indexes* can be based on nonordering attributes, both unique and nonunique.
- B-Tree (default)
- Bitmap - store rowids associated with a key value as a bitmap
- Sparse Index
  - Key and pointer for every *block*
  - Pointer is usually *direct* to record
- Dense Index
  - Key and pointer for every *record*
  - Pointer is usually *indirect* (uses primary index)

Cluster

- Method for storing more than 1 table per block
- i.e.

  SELECT last_name, department_name

  FROM Employee JOIN Department USING dept_id;
- Cluster this data by dept_id
  - All the rows in Employee for dept_id = 10 will be stored with all the rows from Department = 10
  - 1 I/O to get all data
  - Create cluster key – dept_id
  - Index on cluster key becomes Cluster Index

This applies to Oracle

Clustered vs Non-Clustered

- Clustered index
  - Physically reorders the way the records are stored in the table
  - One per table
  - Leaf nodes contain the data pages
  - PK creates if no other clustered index exists
- Non-clustered
  - Logical order of rows does not match physical order of rows
  - Leaf nodes contain index rows

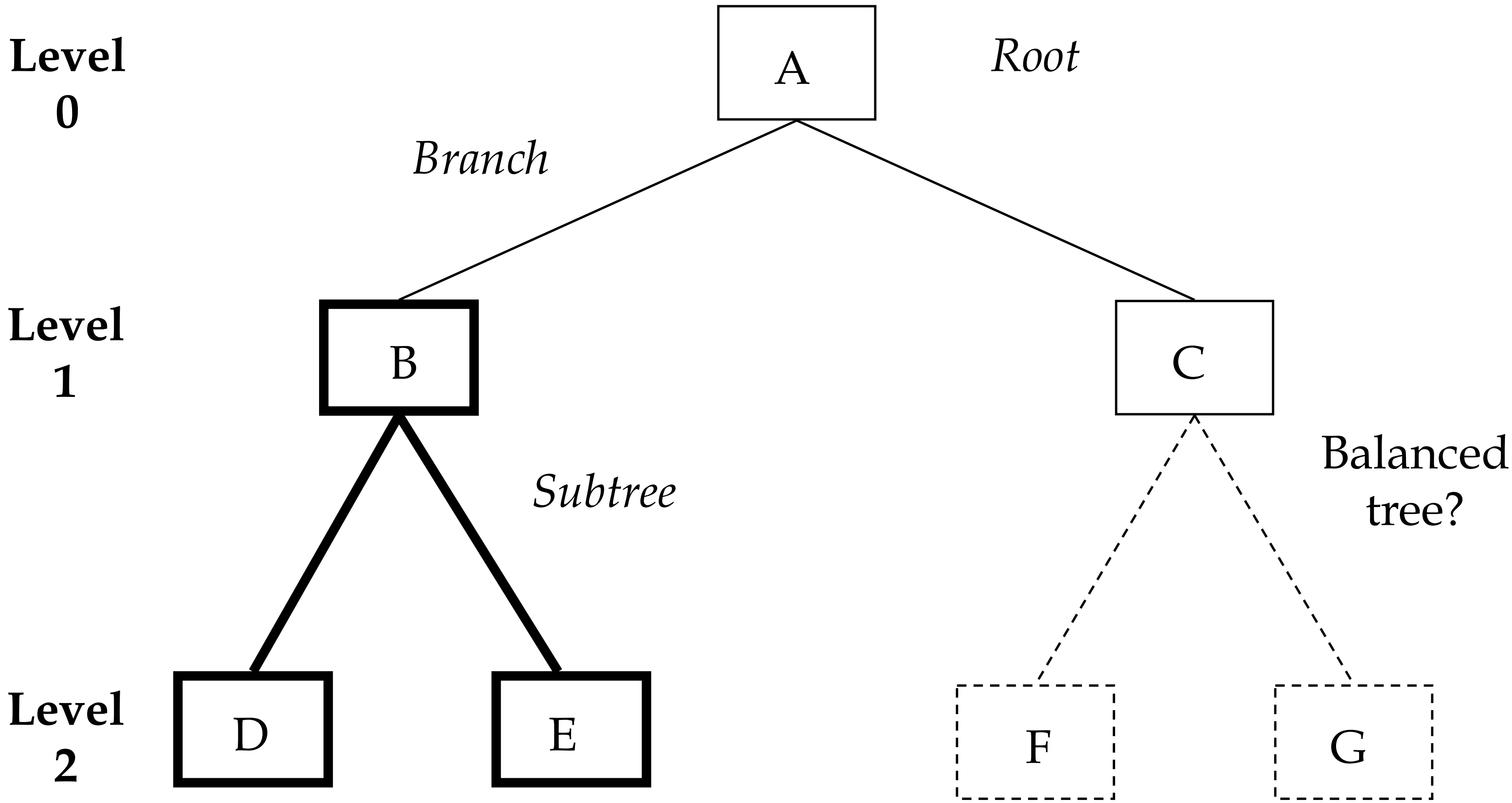
This applies to MS SQL Server and Sybase

Tree Structured Indices

Tree Structured Indices

- *Rooted tree* – forms a hierarchy of index records
- *Leaf nodes* – *l*owest-level (terminal) nodes
- *Siblings* – *n*odes that share a common *parent*
- *Path* – set of pointers (branches) from one node to another
  - 1 unique path to each node
- *Degree* – number of siblings permitted
- *Depth* (height) – number of levels
  - root – level zero

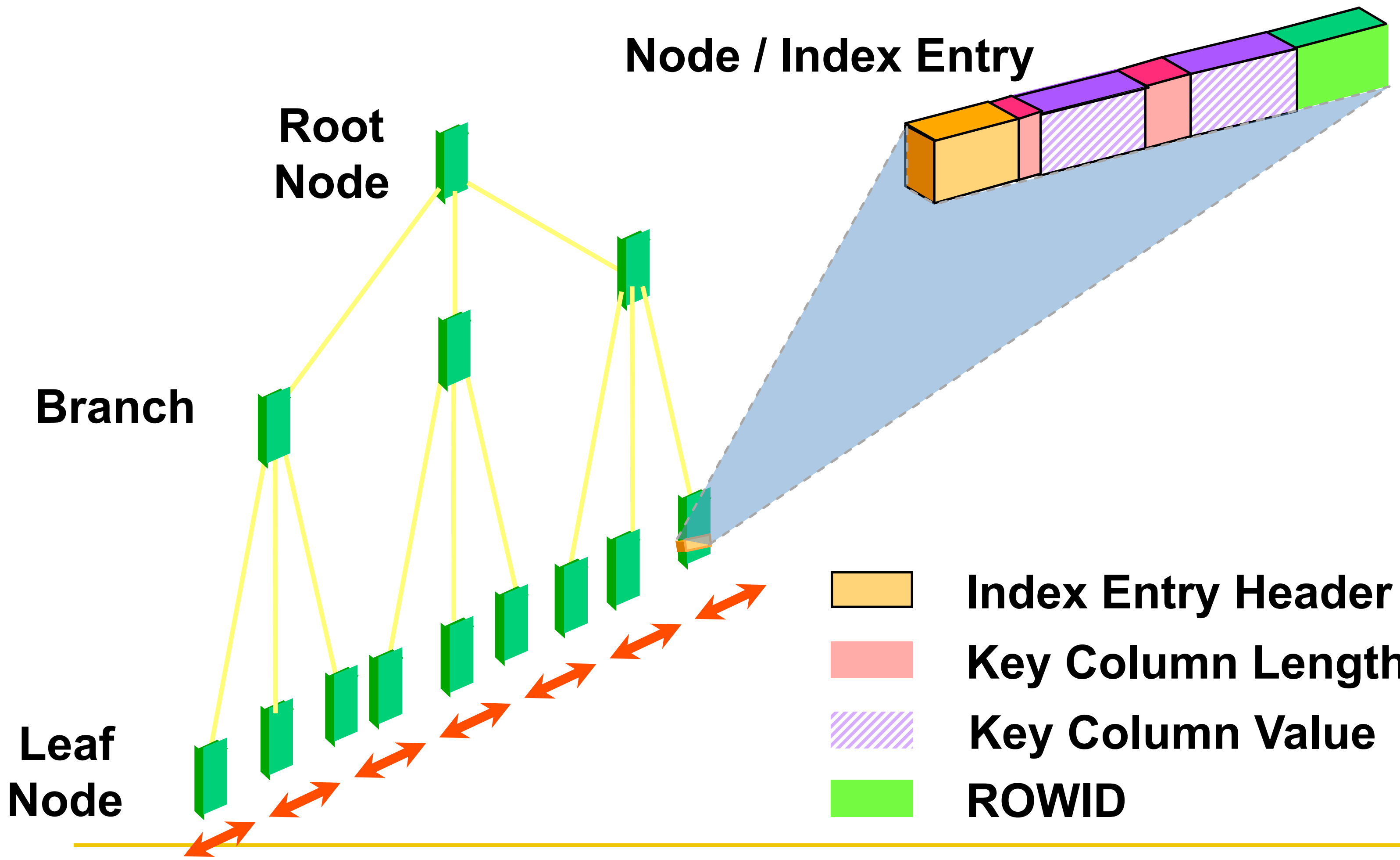
Trees


Level 0
Level 1
Level 2
A
Root
Branch
B
C
Subtree
Balanced tree?
D
E
F
G

Balanced B-Tree

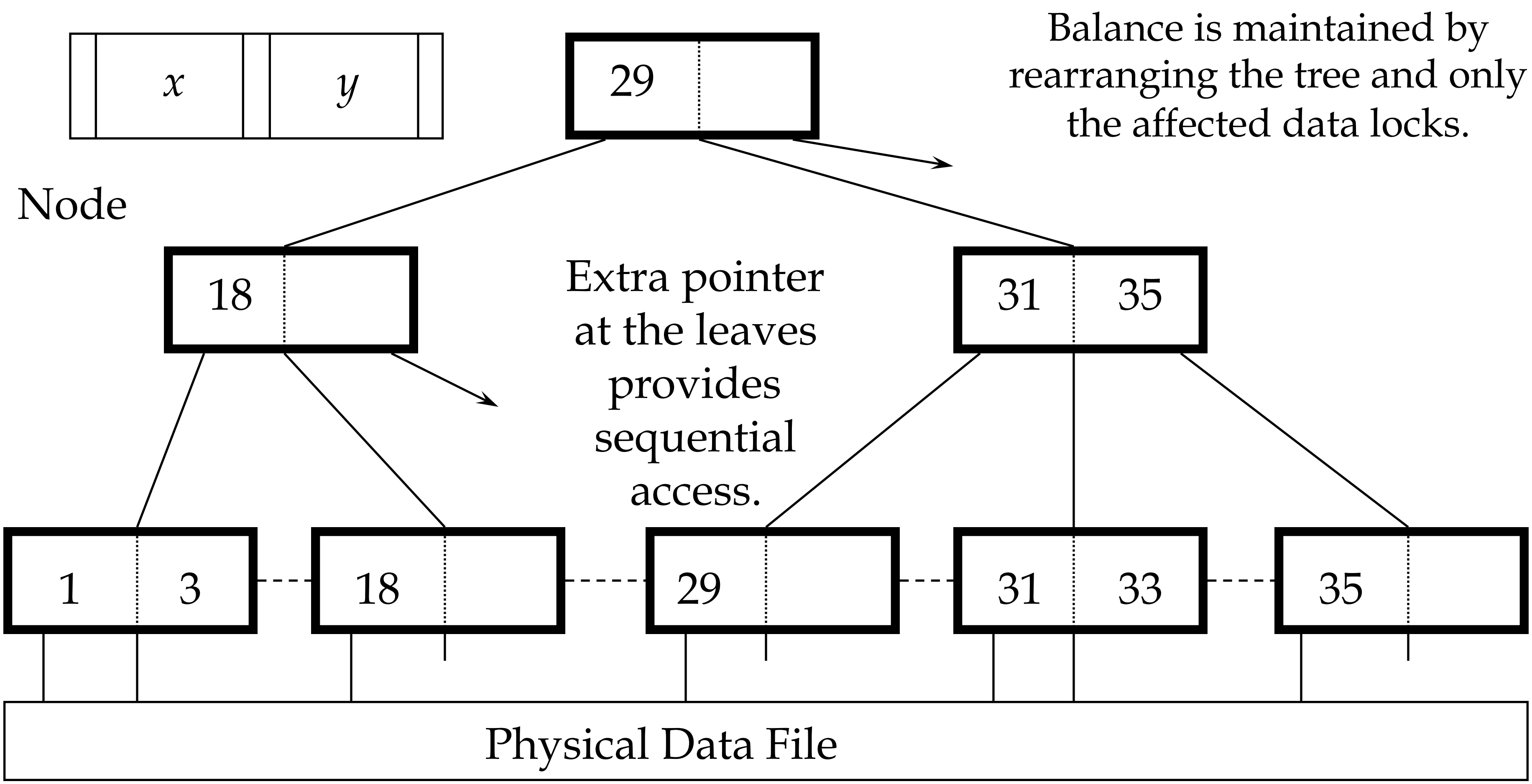
- All paths from the root to a leaf node are the same length
  - Predictable access times
- Each node that is not a leaf has least *n/2* and at most *n* children, where *n* is the order of the B-tree.
- Leaf nodes contain at least *(n-1)/2* and at most *n-1* pointers to data record locations.

B-Tree Index



Node / Index Entry
Root
Node
Branch
Leaf
Node
Index Entry Header
Key Column Length
Key Column Value
ROWID

B+ Trees



x
y
Node
29
Balance is maintained by rearranging the tree and only the affected data locks.
18
Extra pointer at the leaves provides sequential access.
31
35
1
3
18
29
31
33
35
Physical Data File

Reverse Key Index (B-Tree)

Index on EMP (EMPNO)

EMPNO	KEY ROWID (BLOCK# ROW# FILE#)
1257	0000000F.0002.0001
2877	0000000F.0006.0001
4567	0000000F.0004.0001
6657	0000000F.0003.0001
8967	0000000F.0005.0001
9637	0000000F.0001.0001
9947	0000000F.0000.0001
...	...
...	...

EMP table

EMPNO	ENAME	JOB	...
7499	ALLEN	SALESMAN	
7369	SMITH	CLERK	
7521	WARD	SALESMAN	...
7566	JONES	MANAGER	
7654	MARTIN	SALESMAN	
7698	BLAKE	MANAGER	
7782	CLARK	MANAGER	
...	...	...	...
...	...	...	...

What about range queries?

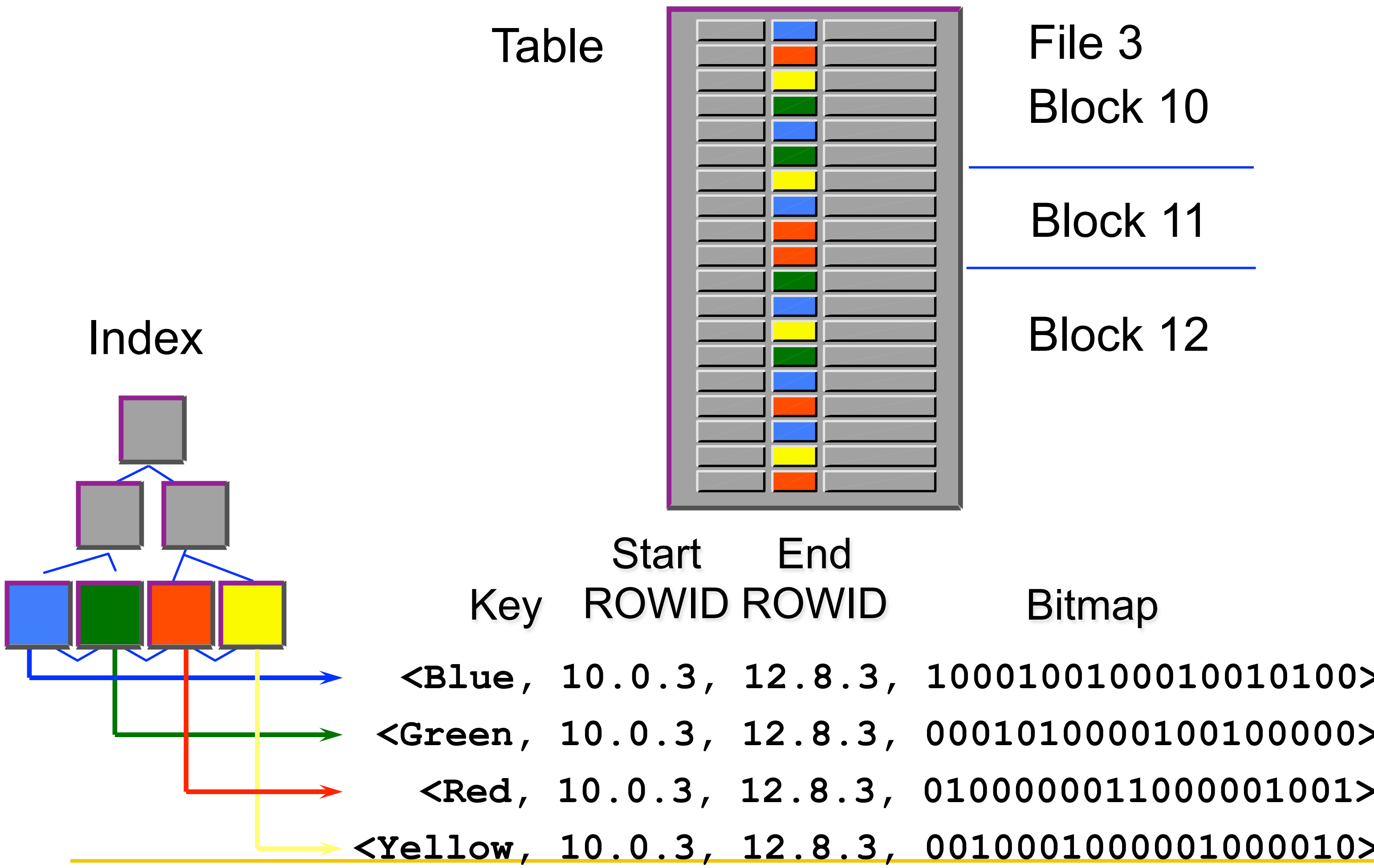
Use of B-Tree Indices

- Particularly suited to high-cardinality attributes
- The B-tree index is the default in most cases
- "For each primary key and unique constraint, Oracle automatically (implicitly) creates a B-tree index."
- B-tree indexes use a large amount of space
- Index can be larger than data table

Bitmap Indices

- A newer form of indexing that is very appropriate for data warehouses
- Appropriate for attributes with low cardinality (e.g., < 1%)
  - Examples: gender, Y/N attributes, categorical attributes ...

Bitmap Index



B-Tree vs Bitmap

B-Tree	Bitmap
Suitable for high-cardinality columns	Suitable for low-cardinality columns
Updates on keys relatively inexpensive	Updates to key columns very expensive
Inefficient for queries using OR predicates	Efficient for queries using OR predicates
Useful for OLTP	Useful for DSS

Composite Indices

Composite

- Given two attributes A1 and A2 …
- If one has an index (idx_a1) and the other does not, we can access the index first and then search the selected rows for a specific value of A2
- i.e. idx_last_name

```
SELECT last_name, first_name
        FROM Employee
        WHERE last_name = 'Smith' AND
        first_name LIKE 'F%';
```

Composite

- If both A1 and A2 have individual indexes, we could use each index to retrieve the appropriate rows and then compute the intersection

    idx_last_name and idx_first_name

- A composite index on A1 and A2 could be used to directly retrieve the required rows, but it is more query specific

    idx_last_first_name

Administration

Create Index

```
CREATE INDEX scott.emp_lname_idx
ON scott.employees(last_name)
PCTFREE 30
STORAGE(INITIAL 200K NEXT 200K
PCTINCREASE 0 MAXEXTENTS 50)
TABLESPACE indx01;
```

Free space left in block before new is created

Keep indices in separate tablespace

Rebuilding Indices

Use this command to:

- Move an index to a different tablespace
- Improve space utilization by removing deleted entries
- Change a reverse key index to a normal B-tree index and vice versa

```
ALTER INDEX scott.ord_region_id_idx REBUILD
TABLESPACE indx02;
```

Dropping Indices

- Drop and re-create an index before bulk loads
- Drop indexes that are infrequently needed and build them when necessary
- Drop and recreate invalid indexes

```
DROP INDEX scott.dept_dname_idx;
```

Indexing Strategy

Indexing Strategy

- Most RDBMS create an index on PKs
- Specify indices for *foreign keys* that are used for joining tables
- Index on *nonkey attributes* that are used frequently for selection criteria or grouping
- OLAP indexed more than OLTP
- Proactive – anticipating usage and build accordingly
- Reactive – based on optimizer and query implementation plan
- Too many on OLTP can slow updates
- Too few on OLAP can slow queries

Oracle Indexing Guidelines

- Balance query and DML needs
- Place in separate tablespace
  - Old – Maximize I/O while minimize disk accesses
- Use uniform extent sizes: multiples of five blocks or MINIMUM EXTENT size for tablespace.
- Consider NOLOGGING for large indices.
- Set high PCTFREE if new key values are likely to be within the current range.

Query Explain Plan

Query Explain

- Each DBMS has a query optimizer
- Before a query is run, query optimizer develops an execution plan
  - Which columns used as index keys, have unique values
  - How many rows each table has
- Tools and statements help us view the execution plan and change queries accordingly
  - SQL Developer write the query and click: 

Query Explain Examples

```
SELECT employee_id, last_name, first_name
 FROM employees
 WHERE last_name = 'Feeney' AND
  first_name = 'Kevin’;
```

- No index on last_name, first_name

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		3	
TABLE ACCESS (FULL)	EMPLOYEES	3	4
Filter Predicates			
AND			
LAST_NAME='Feeney'			
FIRST_NAME='Kevin'			

- Index on last_name, first_name

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		2	
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEES	2	2
INDEX (RANGE SCAN)	EMP_NAME_IX	1	1
Access Predicates			
AND			
LAST_NAME='Feeney'			
FIRST_NAME='Kevin'			

Query Explain Examples

```
SELECT employee_id, last_name
FROM employees JOIN jobs USING (job_id)
WHERE job_title = 'Stock Clerk'
ORDER BY last_name;
```

- No index on job_title; index on job_id (employees & jobs)
  - Total Cost = 5

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		5	
SORT (ORDER BY)		5	5
NESTED LOOPS			5
NESTED LOOPS		4	4
TABLE ACCESS (FULL)	JOBS	3	3
Filter Predicates			
JOBS.JOB_TITLE='Stock Clerk'			
INDEX (RANGE SCAN)	EMP_JOB_IX	0	1
Access Predicates			
EMPLOYEES.JOB_ID=JOBS.JOB_			
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEES	1	1

Query Explain Examples Con't

- Index on job_title and job_id (employees & jobs)
  - Total Cost = 4

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		4	
SORT (ORDER BY)		4	4
NESTED LOOPS			4
NESTED LOOPS		3	3
TABLE ACCESS (BY INDEX ROWID)	JOBS	2	2
INDEX (RANGE SCAN)	JOBS_JOB_TITLE_IX	1	1
Access Predicates			
JOBS.JOB_TITLE='Stock Cle			
INDEX (RANGE SCAN)	EMP_JOB_IX	0	1
Access Predicates			
EMPLOYEES.JOB_ID=JOBS.JOB_			
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEES	1	1