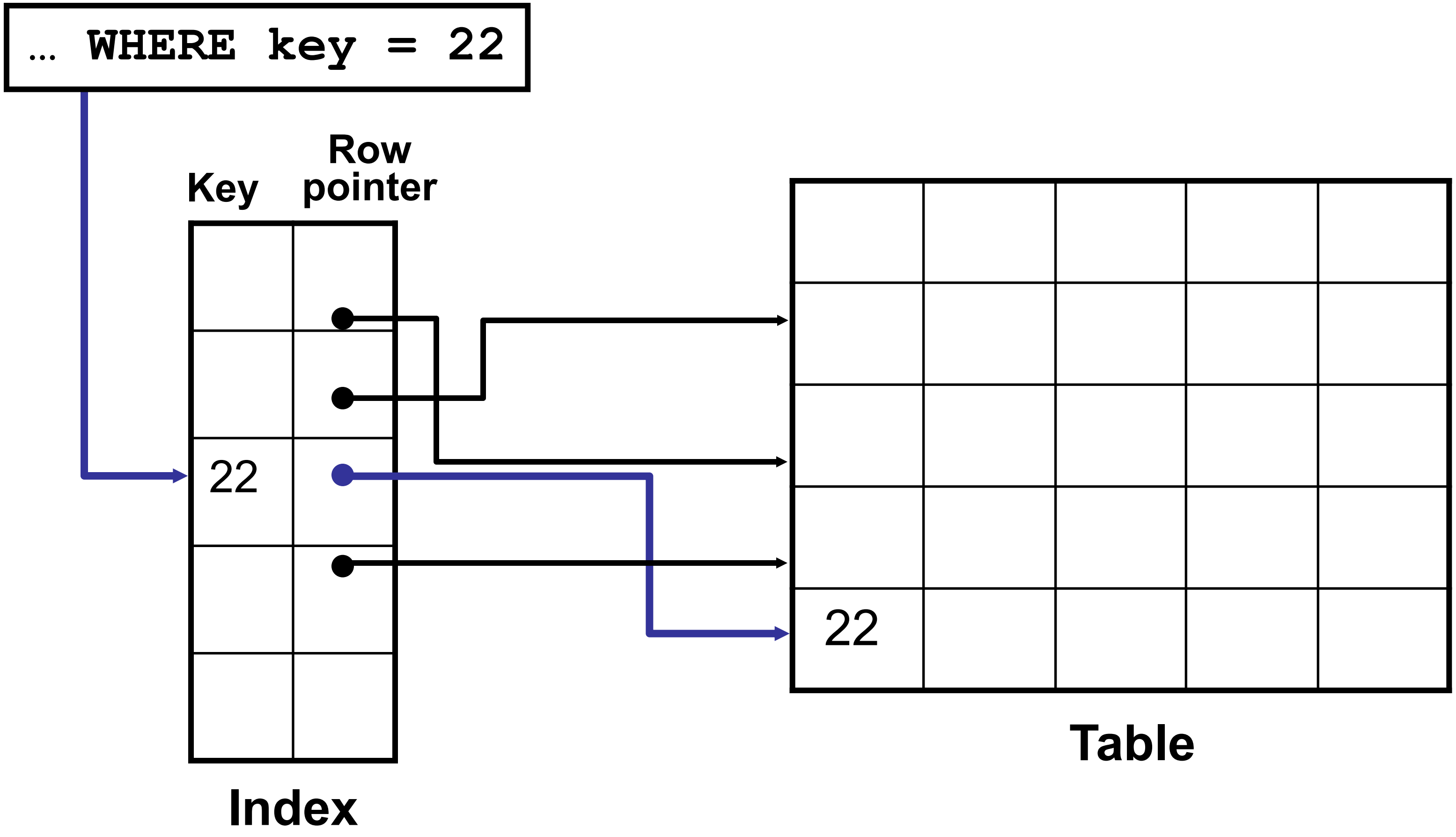
Indexes

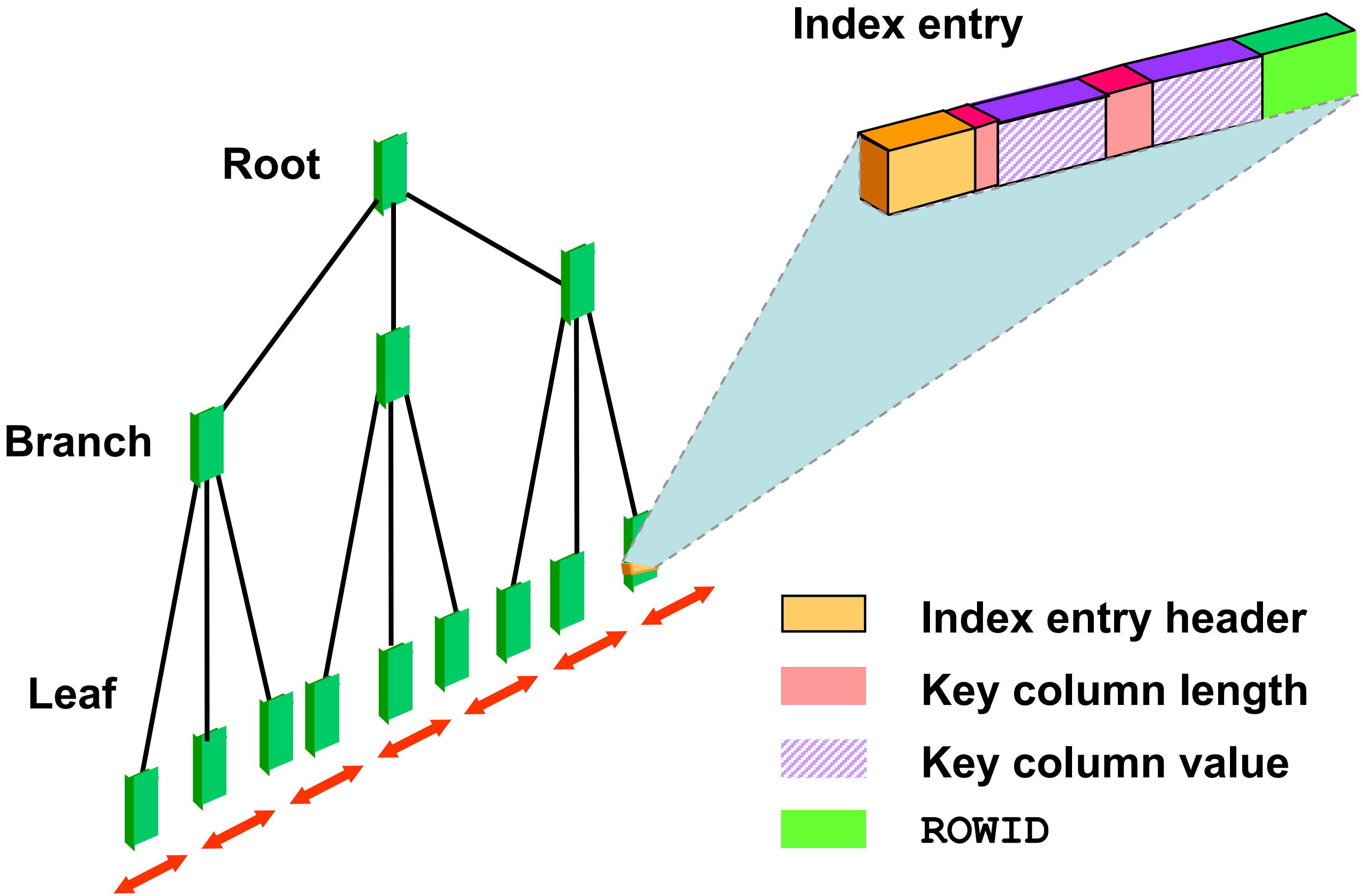


Types of Indexes

These are several types of index structures available to you, depending on the need:

- A B+-tree index is in the form of a balanced tree and is the default index type.
- A bitmap index has a bitmap for each distinct value indexed, and each bit position represents a row that may or may not contain the indexed value. This is best for low-cardinality columns.

B+ tree index



Index entry

Root

Branch

Leaf

🟧 Index entry header

🟥 Key column length

▨ Key column value

🟩 ROWID

B+-Tree Index

Structure of a B+-tree index

At the top of the index is the root, which contains entries that point to the next level in the index. At the next level are branch blocks, which in turn point to blocks at the next level in the index. At the lowest level are the leaf nodes, which contain the index entries that point to rows in the table. The leaf blocks are doubly linked to facilitate the scanning of the index in an ascending as well as descending order of key values.

Format of index leaf entries

An index entry is made up of the following components:

- An entry header, which stores the number of columns and locking information
- Key column length-value pairs, which define the size of a column in the key followed by the value for the column (The number of such pairs is a maximum of the number of columns in the index.)
- `ROWID` of a row that contains the key values

Index Options

- A unique index ensures that every indexed value is unique.

CREATE UNIQUE INDEX emp1 ON EMP (ename);

DBA_INDEXES. UNIQUENESS → 'UNIQUE'

- Bitmap index

CREATE BITMAP INDEX emp2 ON EMP (deptno);

DBA_INDEXES.INDEX_TYPE → 'BITMAP'

An index can have its key values stored in ascending or descending order.

CREATE INDEX emp3 ON emp (sal DESC);

DBA_IND_COLUMNS.DESCEND → 'DESC'

Index Options

- A composite index is one that is based on more than one column.

```
CREATE INDEX emp4 ON emp (empno, sal);
```

DBA_IND_COLUMNS.COLUMN_POSITION → 1,2 …

- A function-based index is an index based on a function’s return value.

```
CREATE INDEX emp5 ON emp (SUBSTR(ename, 3, 4));
```

DBA_IND_EXPRESSIONS.COLUMN_EXPRESSION → ‘SUBSTR …’

- A compressed index has repeated key values removed.

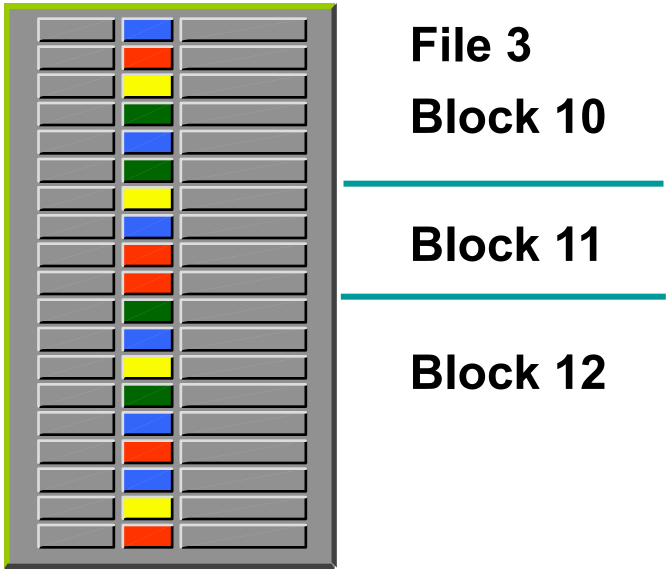
```
CREATE INDEX emp6 ON emp (empno, ename, sal) COMPRESS 2;
```

DBA_INDEXES.COMPRESSION → ‘ENABLED’

DBA_INDEXES.PREFIX_LENGTH → 2

Bitmap Indexes

Table



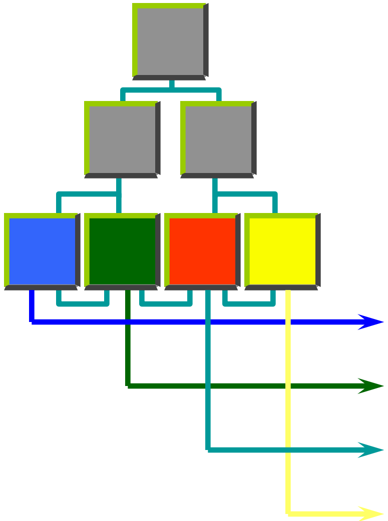
File 3

Block 10

Block 11

Block 12

Index



Key	Start ROWID	End ROWID	Bitmap
<Blue,	10.0.3,	12.8.3,	1000100100010010100>
<Green,	10.0.3,	12.8.3,	0001010000100100000>
<Red,	10.0.3,	12.8.3,	0100000011000001001>
<Yellow,	10.0.3,	12.8.3,	0010001000001000010>

Bitmap Indexes

Structure of a bitmap index

A bitmap index is also organized as a B-tree, but the leaf node stores a bitmap for each key value instead of a list of `ROWID`s. 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.

As shown in the diagram, the leaf node of a bitmap index contains the following:

An entry header that contains the number of columns and lock info

Key values consisting of length and value pairs for each key column

Start `ROWID`

End `ROWID`

A bitmap segment consisting of a string of bits. (The bit is set when the corresponding row contains the key value and is unset when the row does not contain the key value. The Oracle server uses a patented compression technique to store bitmap segments.)

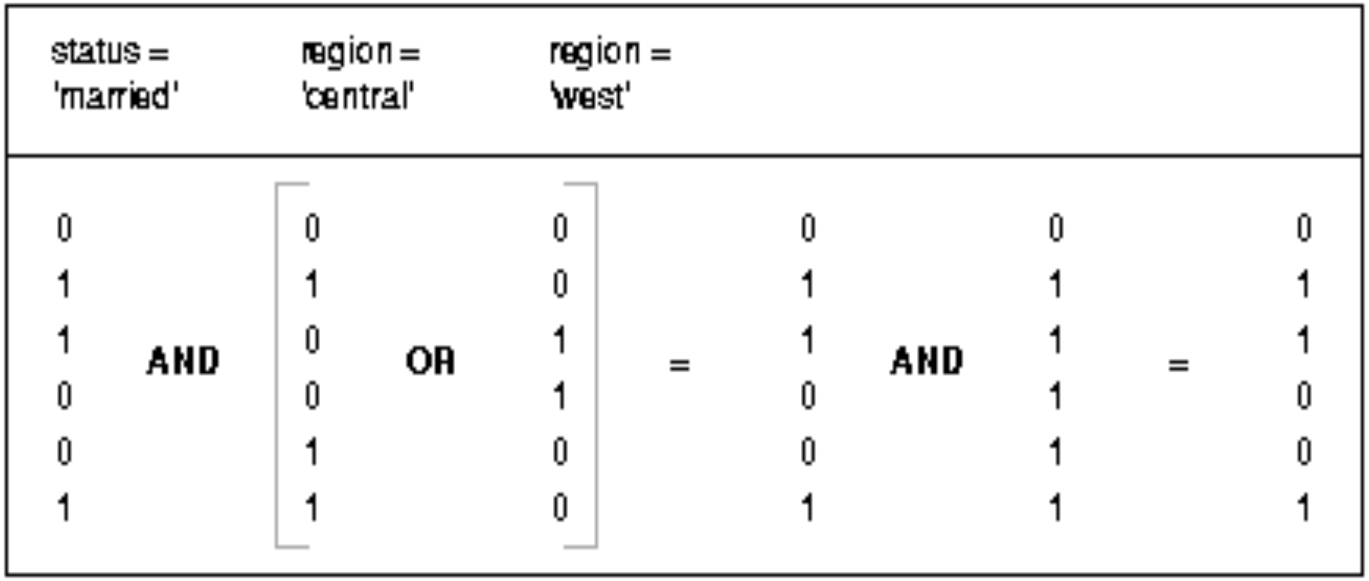
Bitmap Index

Empno	Status	Region	Gender	Info
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

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

Using Bitmap Indexes

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



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

Range queries

AGE	SALARY
25	60
45	60
50	75
50	100
50	120
70	110
85	140
30	260
25	400
45	350
50	275
60	260

```
SELECT * FROM T
WHERE Age BETWEEN 44 AND 55
AND Salary BETWEEN 100 AND 200;
```

Bitvectors for Age

25: 100000001000
30: 000000010000
45: 010000000100
50: 001110000010
60: 000000000001
70: 000001000000
85: 000000100000

Bitvectors for Salary

60: 110000000000
75: 001000000000
100: 000100000000
110: 000001000000
120: 000010000000
140: 000000100000
260: 000000010001
275: 000000000010
350: 000000000100
400: 000000001000

Range queries

AGE	SALARY
25	60
45	60
50	75
50	100
50	120
70	110
85	140
30	260
25	400
45	350
50	275
60	260

SELECT * FROM T
WHERE Age BETWEEN 44 AND 55
AND Salary BETWEEN 100 AND 200;

45: 010000000100
50: 001110000010 OR -> 011110000110

100: 000100000000
110: 000001000000
120: 000010000000
140: 000000100000 OR -> 000111100000

011110000110
000111100000 AND -> 000110000000

Compressed bitmaps

1’s in a bit vector will be very rare. We compress the vector.

Run-length encoding:
run: a sequence of *i* 0’s followed by a 1
1000000100000000010001000000000001

1. Determine how many bits the binary representation of *i* has. This is number *j*.
2. We represent *j* in „unary” by *j-1* 1’s and a single 0.
3. Then we follow with *i* in binary.

Compressed bitmaps

Example: 100000000000001
run with 13 0’s

j = 4 -> in unary: 1110
i in binary: 1101
Encoding for the run: 11101101

Compressed bitmaps

Encoding for *i* = 0: 00
Encoding for *i* = 1: 01

We ignore the trailing 0’s. But not the starting 0’s !

Decoding:
Decode the following: 11101101001011 -> 13, 0, 3

Original bitvector: 0000000000000110001