Indexes

Indexes

- Indexes are optional structures associated with tables and clusters.
- You can create indexes on one or more columns of a table to speed SQL statement execution on that table.
- Oracle index provides a faster access path to table data.
- Indexes are the primary means of reducing disk I/O when properly used.

Indexes(cont..)

- You can create many indexes for a table as long as the combination of columns differs for each index.
- You can create more than one index using the same columns if you specify distinctly different combinations of the columns.
- For example, the following statements specify valid combinations:

```
CREATE INDEX emp_idx1 ON emp (ename, job);
CREATE INDEX emp_idx2 ON emp (job, ename);
```

Unique and Nonunique Indexes

- Indexes can be unique or nonunique.
- Unique indexes guarantee that no two rows of a table have duplicate values in the key column (or columns).
- Non-unique indexes do not impose this restriction on the column values.

Composite Indexes

- A *composite index* is an index that you create on multiple columns in a table.
- Columns in a composite index can appear in any order and need not be adjacent in the table.
- Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index.
- Therefore, the order of the columns used in the definition is important.*Generally, the most commonly accessed or most selective columns go first*.

Function Based Indexes

Function Based Indexes

- A **function-based index** precomputes the value of the function or expression and stores it in the index.
- A function-based index can be created as either a B-tree or a bitmap index.
- The expression cannot contain any aggregate functions, and it must be `DETERMINISTIC`.

Use Of Function Based Index

- Function-based indexes provide an efficient mechanism for evaluating statements that contain functions in their `WHERE` clauses.
- The value of the expression is computed and stored in the index. When it processes `INSERT` and `UPDATE` statements, however, Oracle must still evaluate the function to process the statement.
- For example, the following index:

```
CREATE INDEX uppercase_idx ON emp (UPPER(empname));
```

can facilitate processing queries such as this:

```
SELECT * FROM emp WHERE UPPER(empname) =
'RICHARD';
```

How Indexes are Stored.

How Indexes are stored

- When you create an index, Oracle automatically allocates an index segment to hold the index's data.
- The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the `CREATE INDEX` statement.
- You can improve performance of queries that use an index by storing an index and its table in different tablespaces located on different disk drives, because Oracle can retrieve both index and table data in parallel.

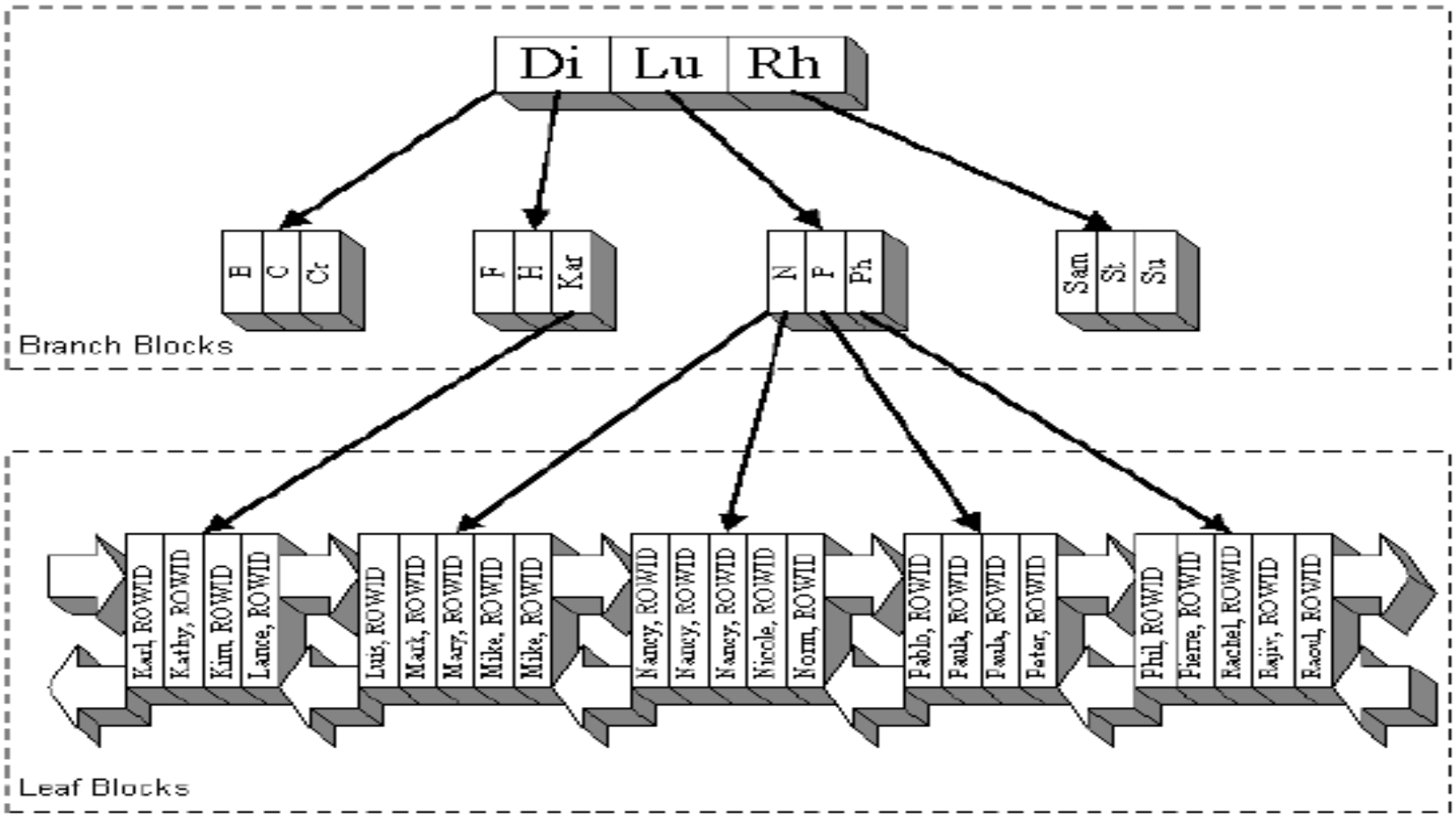
Internal Structure Of Indexes

- Oracle uses *B Trees* to store indexes to speed up data access. If there are no indexes then you have to do a sequential scan on the data to find a value.

Basic principle behind Oracle indexes :

If we had an ordered list of the values, then we could divide it into block wide ranges (leaf blocks). The end points of the ranges along with pointers to the blocks could be stored in a search tree and we could find a value in log(n) time for n entries. This can be illustrated by the following diagram :

Figure 11–7 Internal Structure of a B-tree Index



Structure Of Indexes (contd.)

- The upper blocks (**branch blocks**) of a B-tree index contain index data that points to lower-level index blocks. The lowest level index blocks (**leaf blocks**) contain every indexed data value and a corresponding rowid used to locate the actual row. The leaf blocks are doubly linked.
- For a unique index, there is one rowid for each data value. For a nonunique index, the rowid is included in the key in sorted order, so nonunique indexes are sorted by the index key and rowid.

Index Properties

There are two kinds of blocks :

- Branch blocks for searching
- Leaf blocks that store the values

Branch Blocks : Branch blocks store the following:

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

Leaf Blocks : Leaf blocks store the following:

- The complete key value for every row
- `ROWID`s of the table rows

Advantage of B-Tree structure

The B-tree structure has the following advantages:

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
- B-tree indexes automatically stay balanced.
- B-trees provide excellent retrieval performance for a wide range of queries,including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.

How Indexes Are Searched

Index Unique Scan

- This method is used for returning the data from B-tree indexes.
- The Optimizer chooses a unique scan when all columns of a unique (B-tree) index are specified with *equality* conditions.

Index Unique Scan

Steps in Index Unique Scans

1 Start with the root block.

2 Search the block keys for the smallest key >= value.

3 If key > value, then follow the link before this key to the child block.

4 If key = value, then follow this link to the child block.

5 If there is no key >= value in Step 2, then follow the link after the highest key in the block.

6 Repeat steps 2 through 4 if the child block is a branch block.

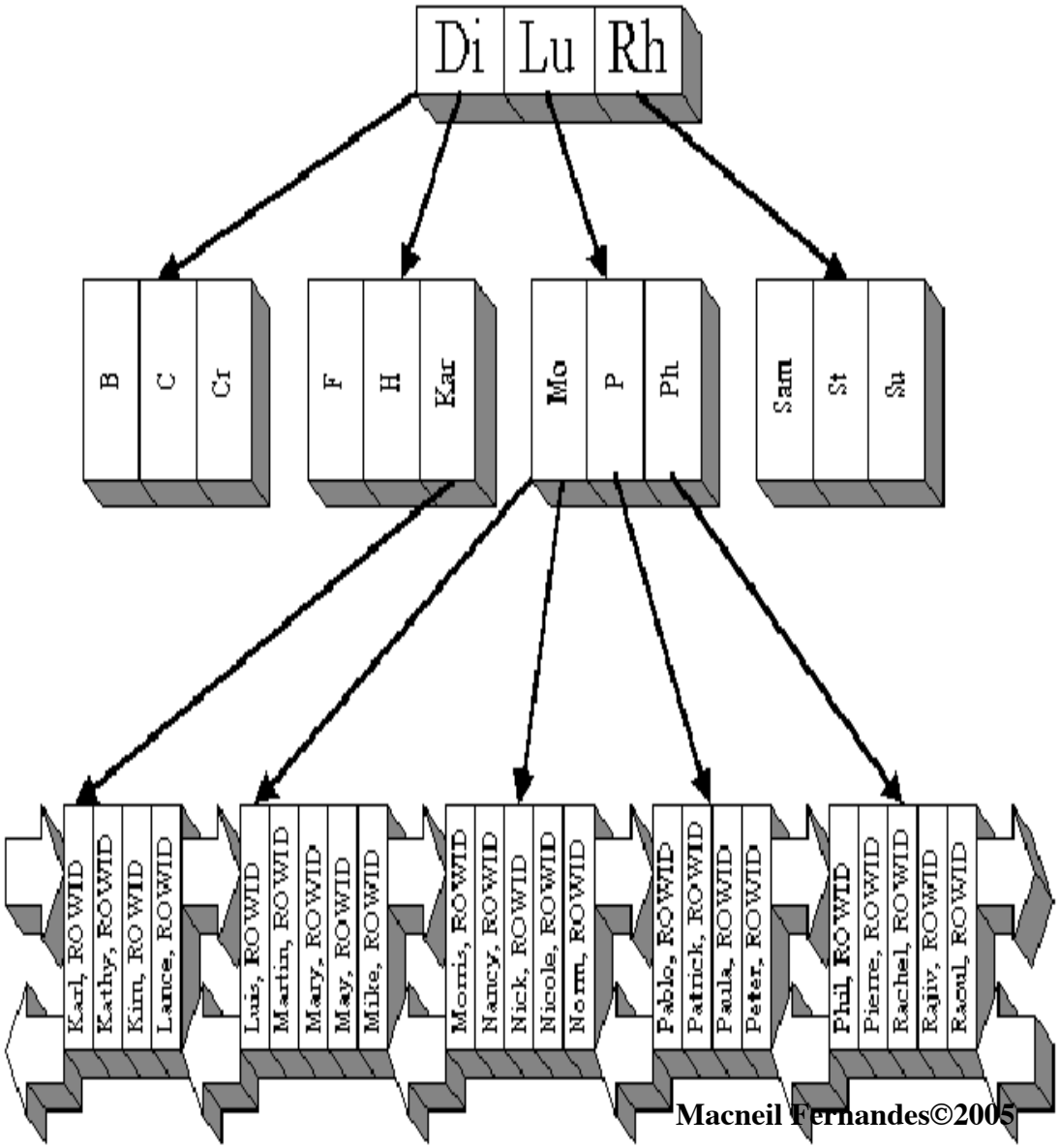
7 Search the leaf block for key equal to the value.

8 If key is found, then return the ROWID. If key is not found, then the row does not exist.

Index Unique Scan

If searching for Patrick:

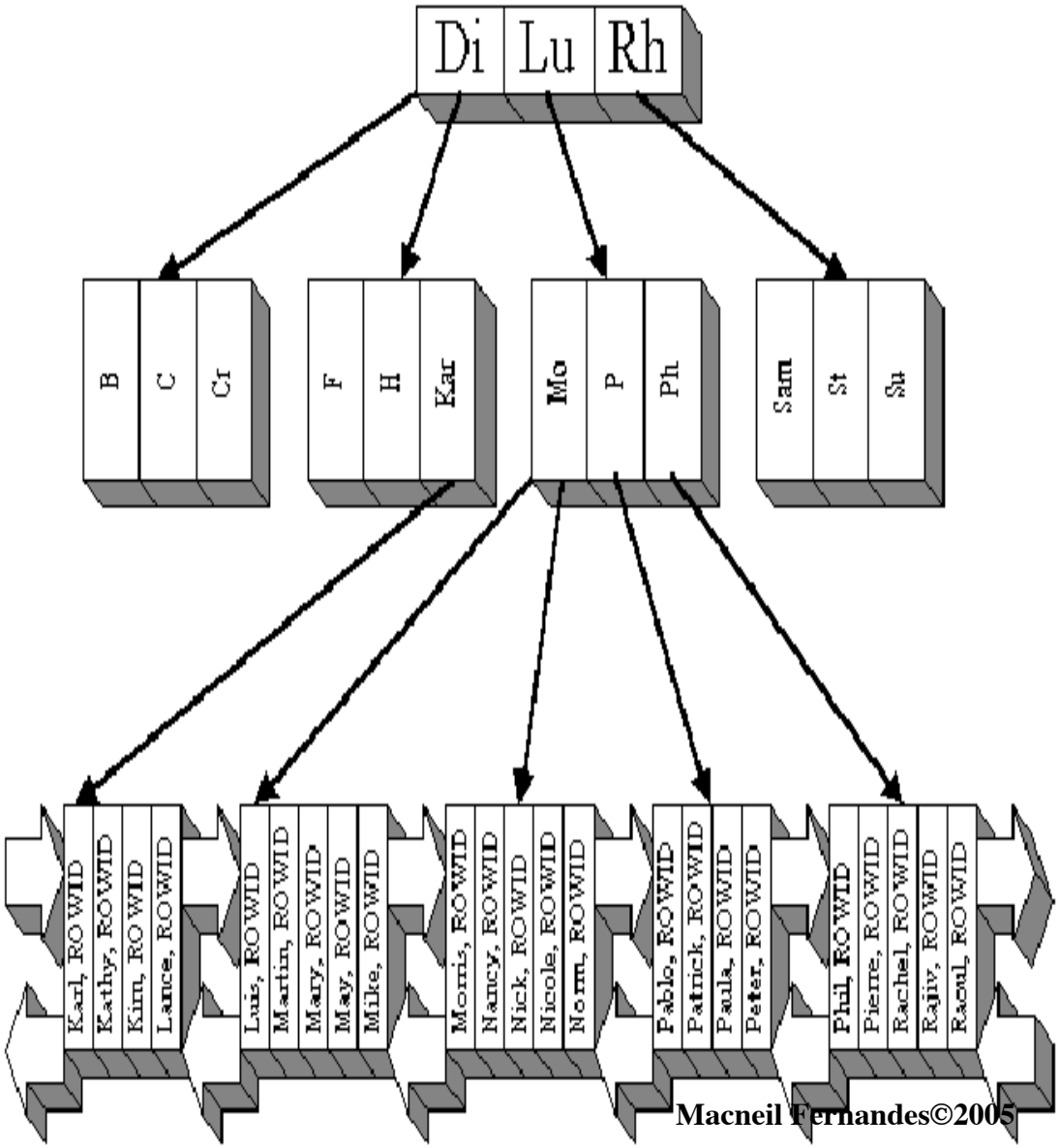
- In the root block, Rh is the smallest key >= Patrick.
- Follow the link before Rh to branch block (Mo, P, Ph).
- In this block, Ph is the smallest key >= Patrick.
- Follow the link before Ph to leaf block (Pablo, Patrick, Paula, Peter).
- In this block, search for key Patrick = Patrick.
- Found Patrick = Patrick, return (KEY, ROWID).



Index Unique Scan

If searching for Meg:

- In the root block, Rh is the smallest key >= Meg.
- Follow the link before Rh to branch block (Mo, P, Ph).
- In this block, Mo is the smallest key >= Meg.
- Follow the link before Mo to leaf block (Luis,… , May, Mike).
- In this block, search for key = Meg.
- Did not find key = Meg, return 0 rows.



Index Range Scan

- Index range scan is a common operation for accessing selective data. It can be bounded (bounded on both sides) or unbounded (on one or both sides).
- Specify an equality condition.
  - order_id = 100 - start key = 100, end key = 100
- Specify an interval bounded by start key and end key.
  - order_id BETWEEN 100 AND 120 - start key = 100, end key = 120
- Specify just a start key or an end key (unbounded range scan).
  - order_id >= 100
  - order_id <= 100

Index Range Scan

Steps in a Bounded Range Scan

1. Start with the root block.
2. Search the block keys for the smallest key >= start key.
3. If key > start key, then follow the link before this key to the child block.
4. If key = start key, then follow this link to the child block.
5. If there is no key >= start key in Step 2, then follow the link after the highest key in the block.
6. Repeat steps 2 through 4 if the child block is a branch block.
7. Search the leaf block keys for the smallest key greater than or equal to the start key.
8. While key <= end key:
   - If the key columns meet all WHERE clause conditions, then return the (value, ROWID).
   - Follow the link to the right.

Index Range Scan

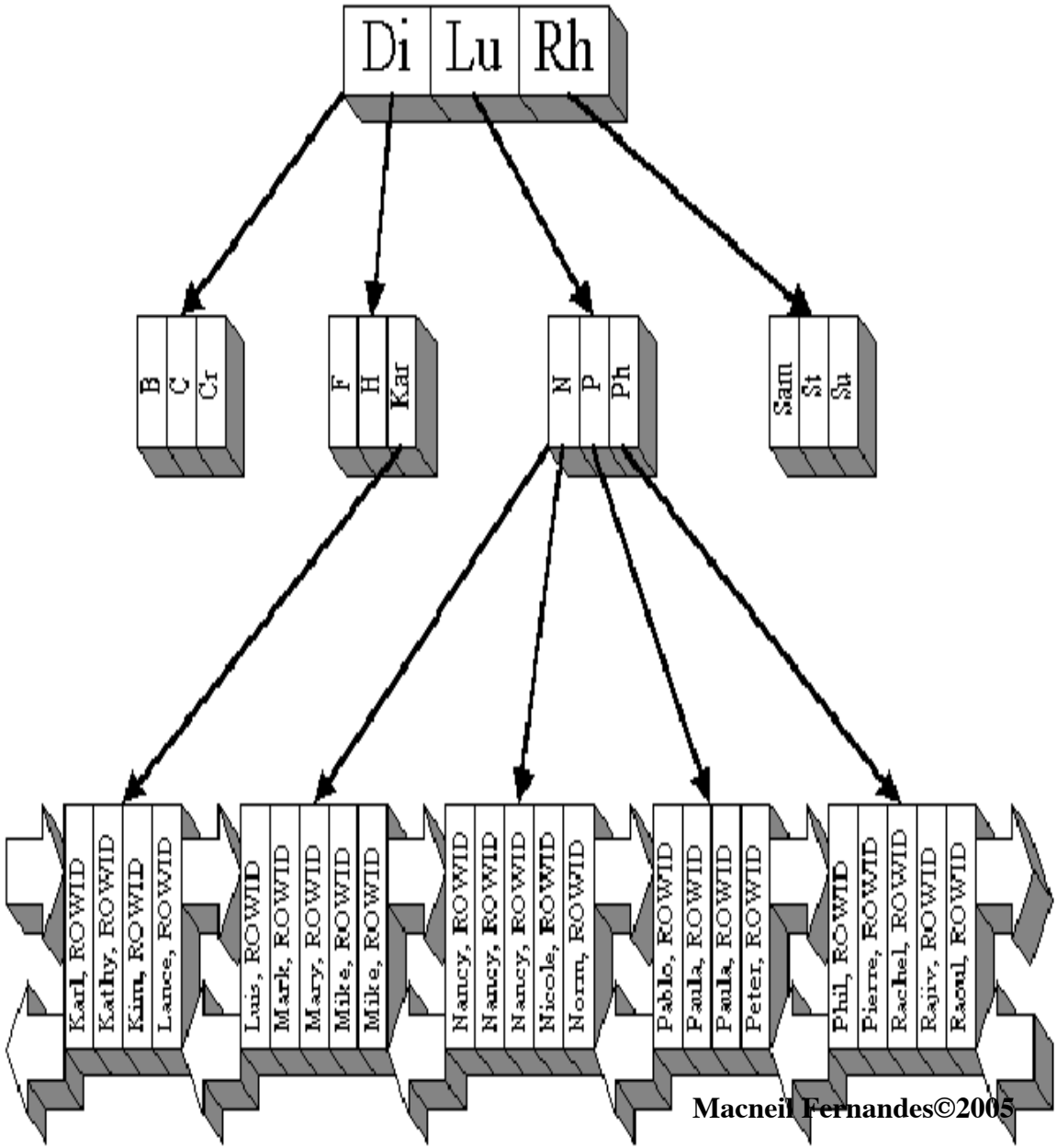
- Range scans bounded on the left (unbounded on the right) start the same as above. However, they do not check for the end point. They continue until they reach the right-most leaf key.

- Range scans bounded on the right traverse the index tree to the left-most leaf key and then follow step #6 and # 7 until they reach a key greater then the specified condition.

Index Range Scan

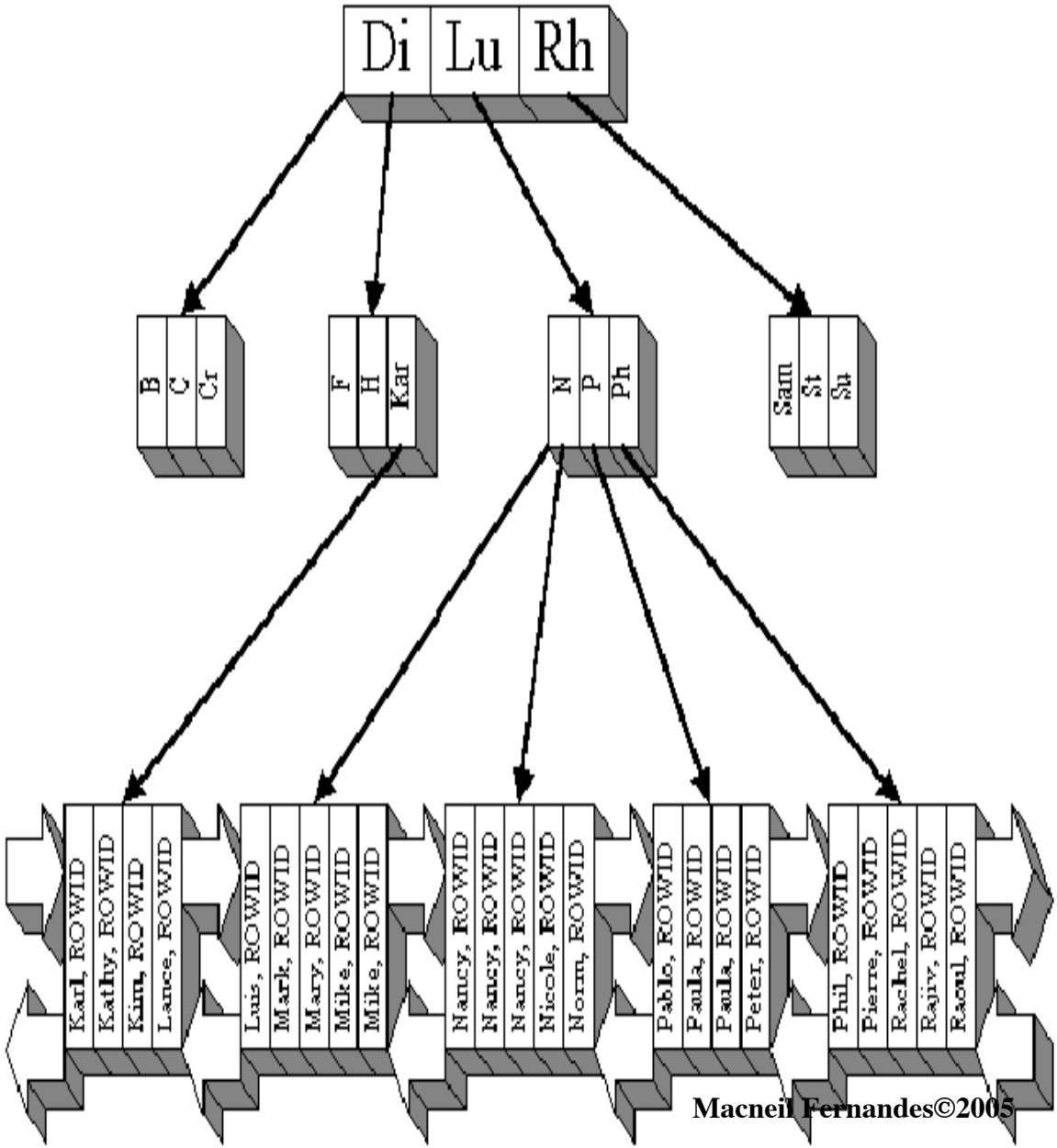
If searching for Nancy:

- Start key = ‘Nancy’, end key = ‘Nancy’.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key >= start key.
- Follow the link before P to leaf block (Nancy, …, Nicole, Norm).
- In this block, Nancy is the smallest key >= start key.



Topic One

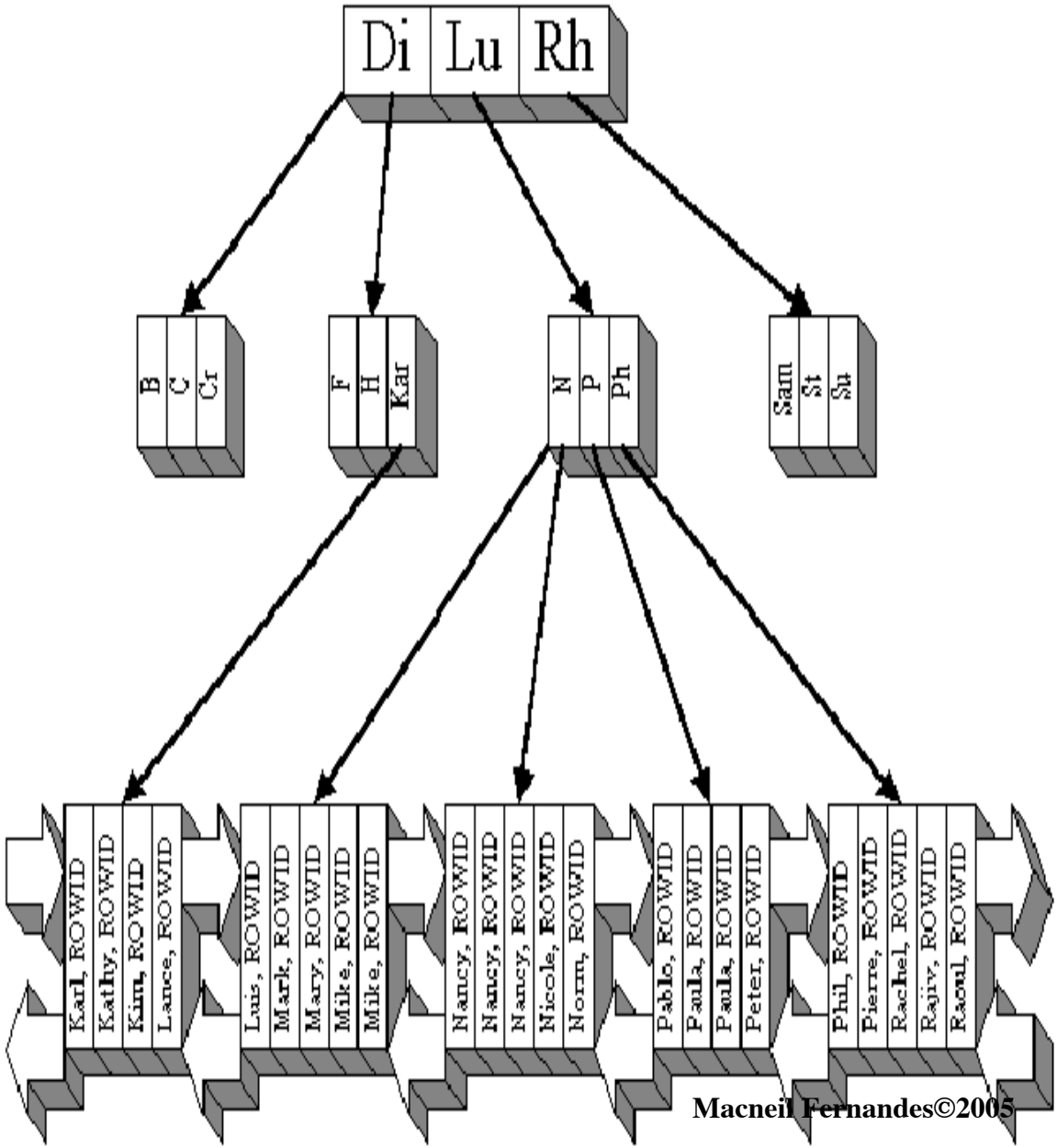
- Because Nancy <= end key, return the (KEY, ROWID).
- Next key Nancy <= end key, return the (KEY, ROWID).
- Next key Nancy <= end key, return the (KEY, ROWID).
- Next key Nicole > end key, terminate the range scan.



Index Range Scan

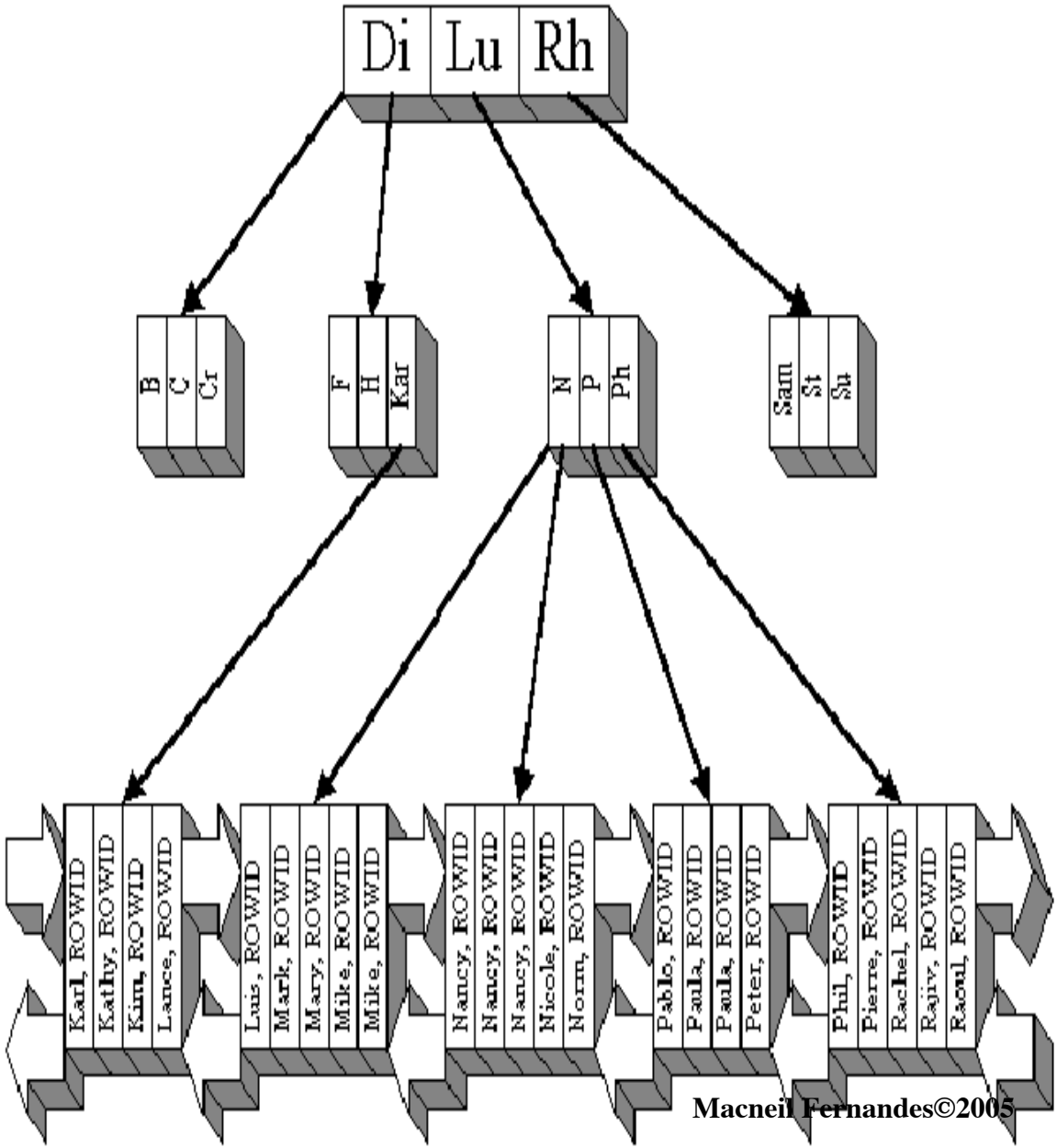
If searching for ‘P%’:

- Start key = ‘P’, end key = ‘Q’.
- In the root block, Rh is the smallest key >= start key.
- Follow the link before Rh to branch block (N, P, Ph).
- In this block, P is the smallest key = start key.
- Follow this link to leaf block (Pablo,…, Peter).
- In this block, Pablo is the smallest key >= start key.



Index Range Scan

- Because Pablo <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).
- Next key Paula <= end key, return the (KEY, ROWID).
- Next key Phil <= end key, return the (KEY, ROWID).
- Next key Pierre <= end key, return the (KEY, ROWID).
- Next key Rachel > end key, terminate the range scan.



Reverse key Indexes

Reverse key Indexes

- Creating a *reverse key index, c*ompared to a standard index, reverses the bytes of each column indexed (except the rowid) while keeping the column order.
- It helps to avoid performance degradation.
- By reversing the keys of the index, the insertions become distributed across all leaf keys in the index.
- Usage of the reverse key arrangement eliminates the ability to run an index range scanning query on the index. Because lexically adjacent keys are not stored next to each other in a reverse-key index, only fetch-by-key or full-index (table) scans can be performed.

Reverse key Indexes

- The REVERSE keyword provides a simple mechanism for creating a reverse key index.
- For example :
  - CREATE INDEX i ON t (a,b,c) REVERSE;
- You can specify the keyword NOREVERSE to REBUILD a reverse-key index into one that is not reverse keyed:
- For example :
  - `ALTER INDEX i REBUILD NOREVERSE;`
- Rebuilding a reverse-key index without the NOREVERSE keyword produces a rebuilt, reverse-key index.

Key Compression

Key Compression

- Key compression *enables compression* of portions of the key column values in an *index*.
- This reduces the storage overhead of repeated values.
- Keys in an index have two pieces
  - **a grouping piece***(Repeating part of the key)*
  - **a unique piece.(***If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece*.**)**
- Key compression is a method of breaking off the *grouping piece* and storing it so it can be *shared by multiple unique pieces*.

Prefix and Suffix Entries

- Key compression breaks the index key into a *prefix entry* (the grouping piece) and a *suffix entry* (the unique piece).
- Compression is achieved by sharing the *prefix entries* among the suffix entries in an index block.
- For example,
  - **In a key made up of three columns (column1, column2, column3) the default prefix is (column1, column2). For a list of values (1,2,3), (1,2,4), (1,2,7), (1,3,5), (1,3,4), (1,4,4) the repeated occurrences of (1,2), (1,3) in the prefix are compressed.**
- Also one can specify the length of the prefix, *(i.e. number of columns to be included in the prefix.)*
- For example,
  - **If you specify prefix length 1, then the prefix is column1 and the suffix is (column2, column3). For the list of values (1,2,3), (1,2,4), (1,2,7),(1,3,5), (1,3,4), (1,4,4) the repeated occurrences of 1 in the prefix are compressed.**

Performance and Storage Considerations

- Key compression leads to
  - **A huge saving in space (stores more keys in each index block)**
  - **Less I/O and**
  - **Better performance.**
  - **Increase in the CPU time required to reconstruct the key column values during an index scan.**

Implementing Key Compression

- Key compression can be useful in the following situations:
  - **For a non-unique index to which `ROWID` is appended to make the key unique. The duplicate key is stored as a prefix entry on the index block without the `ROWID`. The remaining rows become suffix entries consisting of only the `ROWID`.**
  - **For a unique multi-column index.**
- Enable key compression using the *`COMPRESS`* clause. The prefix length (as the number of key columns) can also be specified to identify how the key columns are broken into a prefix and suffix entry.

```
CREATE INDEX emp_ename ON emp(ename)
COMPRESS 1;
```

- The `COMPRESS` clause can also be specified during rebuild. For example, during rebuild you can disable compression as follows:

```
ALTER INDEX emp_ename REBUILD NOCOMPRESS;
```

Bitmap Indexes

Bitmap Indexes

- 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.
- If the bit is set, then it means that the row with the corresponding rowid contains the key value.
- A *mapping function* converts the bit position to an actual rowid, so the bitmap index provides the same functionality as a regular index even though it uses a different representation internally.

Bitmap Indexes

- The advantages of using bitmap indexes are greatest for low ***cardinality columns***: *(i.e. columns in which the number of distinct values is small compared to the number of rows in the table.)*
- If the number of distinct values of a column is less than 1% of the number of rows in the table, or if the values in a column are repeated more than 100 times, then the column is a candidate for a bitmap index.
- For example,
  - **On a table with 1 million rows, a column with 10,000 distinct values is a candidate for a bitmap index.**
- Even columns with a lower number of repetitions and thus higher cardinality can be candidates if they tend to be involved in complex conditions in the `WHERE` clauses of queries.

Bitmap Index Example

- Consider the Table Given Below:
- From the Table Data we find the low cardinality columns to be:
  - **`MARITAL_STATUS` (three possible values),**
  - **`REGION`(three possible values),**
  - **`GENDER`(two possible values),**
- Therefore, it is appropriate to create bitmap indexes on these columns.

CUSTOMER #	MARITAL_STATUS	REGION	GENDER
101	single	east	male
102	married	central	female
103	married	west	female
104	divorced	west	male
105	single	central	female
106	married	central	female

Bitmap Index Example

- The Figure below illustrates the *Bitmap index* for the `REGION` column in this example.
- It consists of three separate bitmaps, one for each region.
- Each entry or bit in the bitmap corresponds to a single row of the `CUSTOMER` table.
- The value of each bit depends upon the values of the corresponding row in the table.

101	single	east	male

The bitmap `REGION='east'` contains a one as its first bit. This is because the region is `east` in the first row of the `CUSTOMER` table.

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

The bitmap REGION='east' has a zero for its other bits because none of the other rows of the table contain east as their value for REGION.

Bitmap Index Example

- Similarly we can represent the *Bitmap index* for the GENDER column.

CUSTOMER #	GENDER
101	male
102	female
103	female
104	male
105	female
106	female

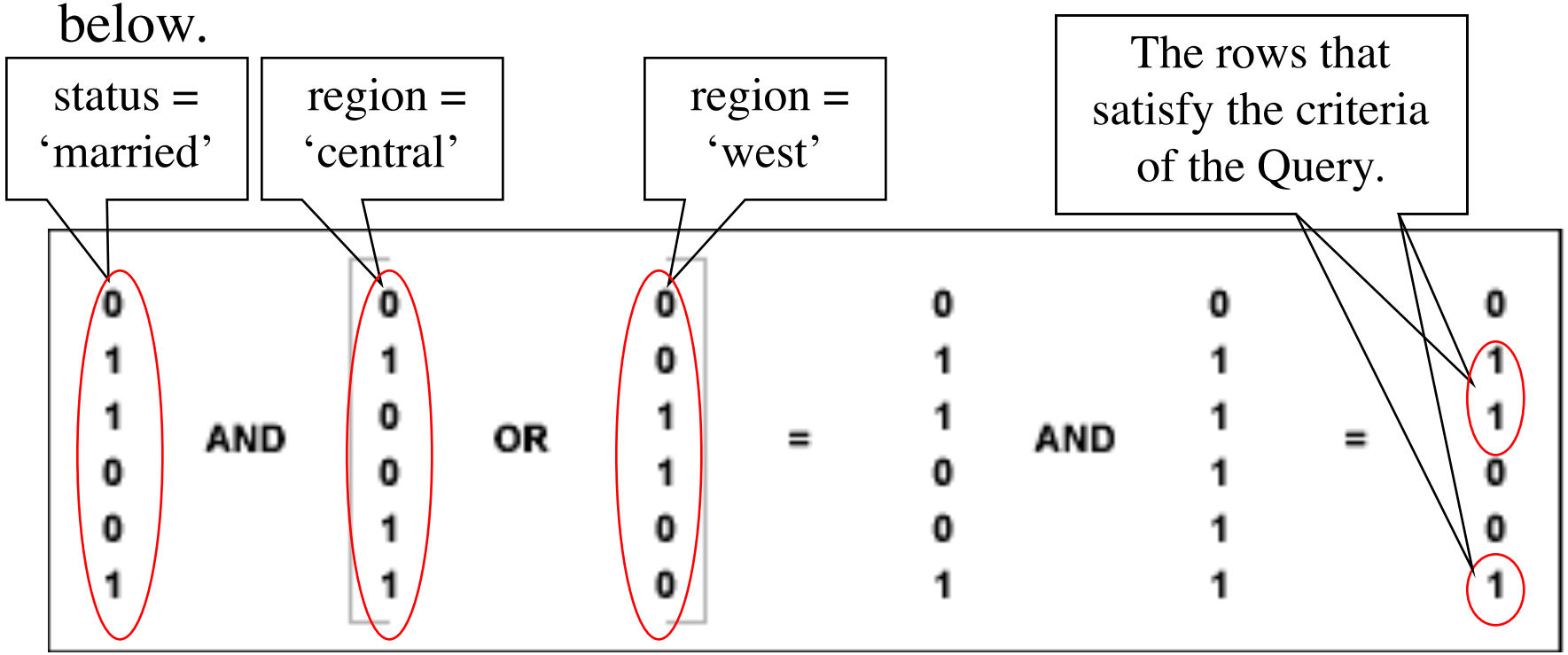
GENDER = 'male'	GENDER = 'female'
1	0
0	1
0	1
1	0
0	1
0	1

- Consider the SQL Query:

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

Bitmap Index Example*(Executing a Query Using Bitmap Indexes)*

- Bitmap indexes can process this query with great efficiency by counting the number of ones in the resulting bitmap, as illustrated below.



- Result of the Query:

COUNT(*)

Bitmap Join Indexes

Bitmap Join Indexes

- The bitmap join index,is a bitmap index on a table `F` based on columns from table `D1,...,Dn,` where `Di` joins with `F` in a star or snowflake schema.
- Table `F` is usually a fact table, table `Di` is usually a dimension table, and the join condition is a join between the primary key column(s) of the dimension tables and the foreign key column(s) in the fact table.
- Bitmap join indexes are much more efficient in storage than materialized join views which do not compress rowids of the fact tables.

Four Join Models

The following is a description of four join models in the star query framework and how they are addressed by bitmap join indexes.

Notation:

`F`*`i`* -- Fact table *i*

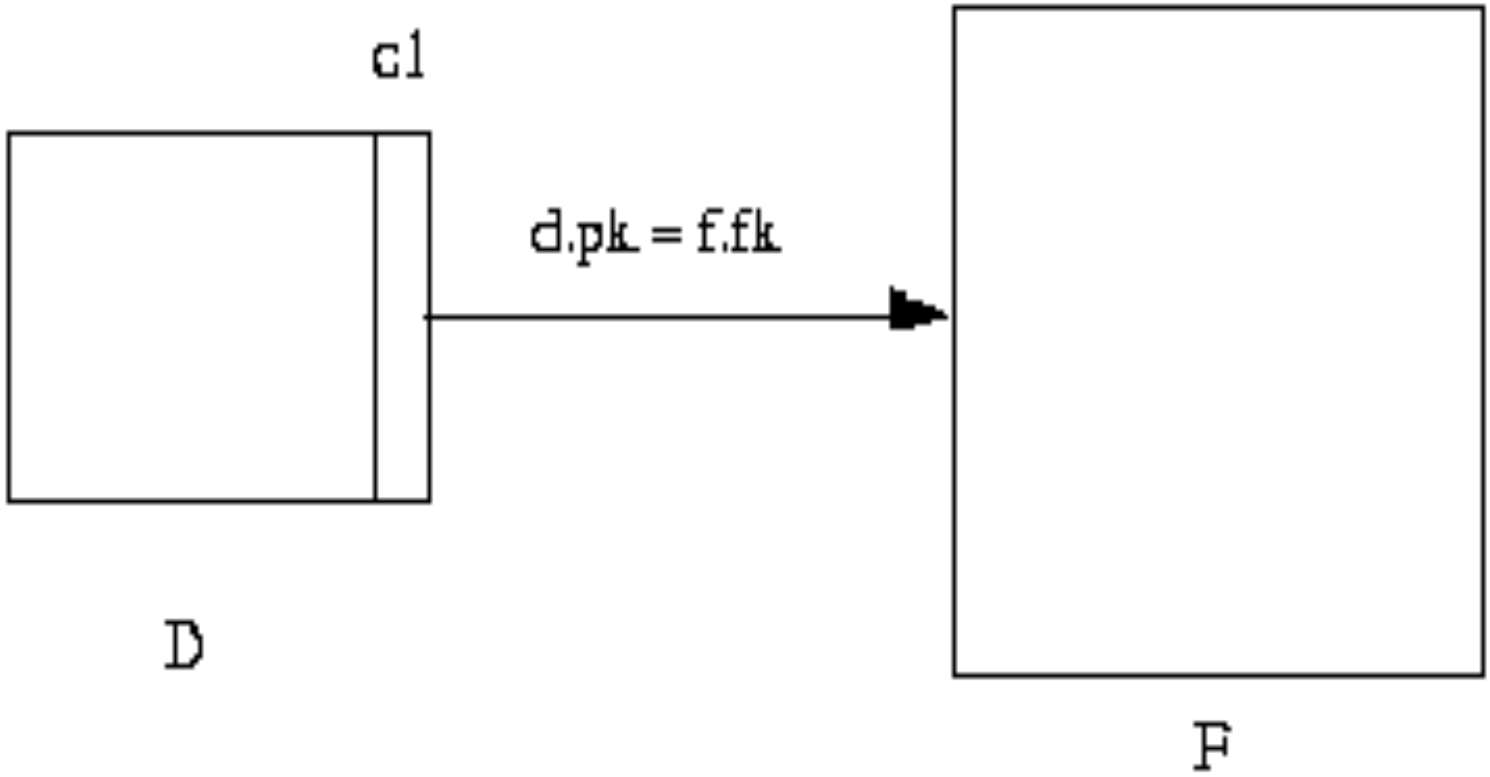
`D`*`i`* -- Dimension table *i*

`pk` -- The primary key column on the dimension table

`fk` -- The fact table column participating in the join with the dimension tables(foreign key).

`sales` -- The measurement column on the fact table

One dimension table column joins with one fact table



•The model shown above is a bitmap join index on `F(D.c1),` can be represented by the following SQL statement:

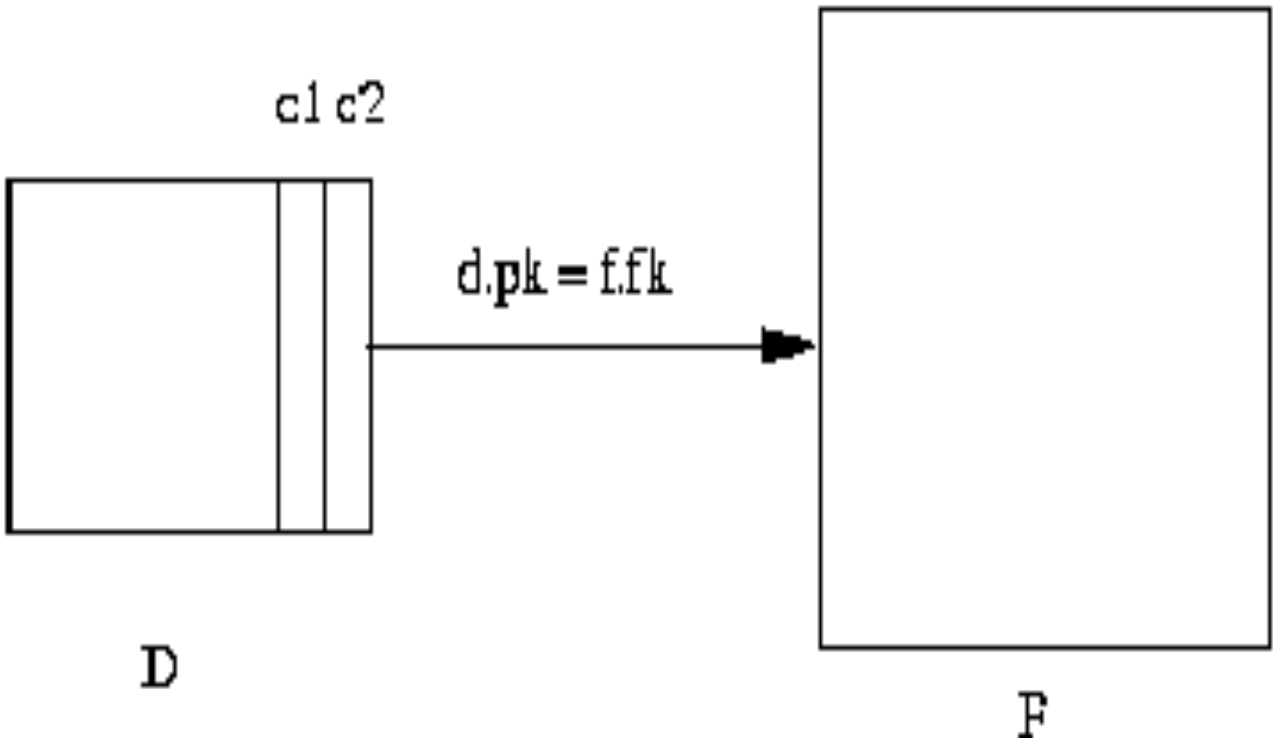
```
CREATE BITMAP INDEX bji ON f (d.c1) FROM f, d WHERE
d.pk = f.fk
```

Then the following query

```
select sum(f.sales) from d, f
where d.pk = f.fk and d.c1 = 2;
```

can be executed by accessing the bitmap join index to avoid the join operation.

Two or more dimension table columns join with one fact table



Above figure shows a simple extension of the previous model, requiring a concatenated bitmap join index to represent it, as follows:

```
CREATE BITMAP INDEX bji ON f (d.c1, d.c2) FROM F, d WHERE
d.pk = f.fk;
```

The result of the following query:

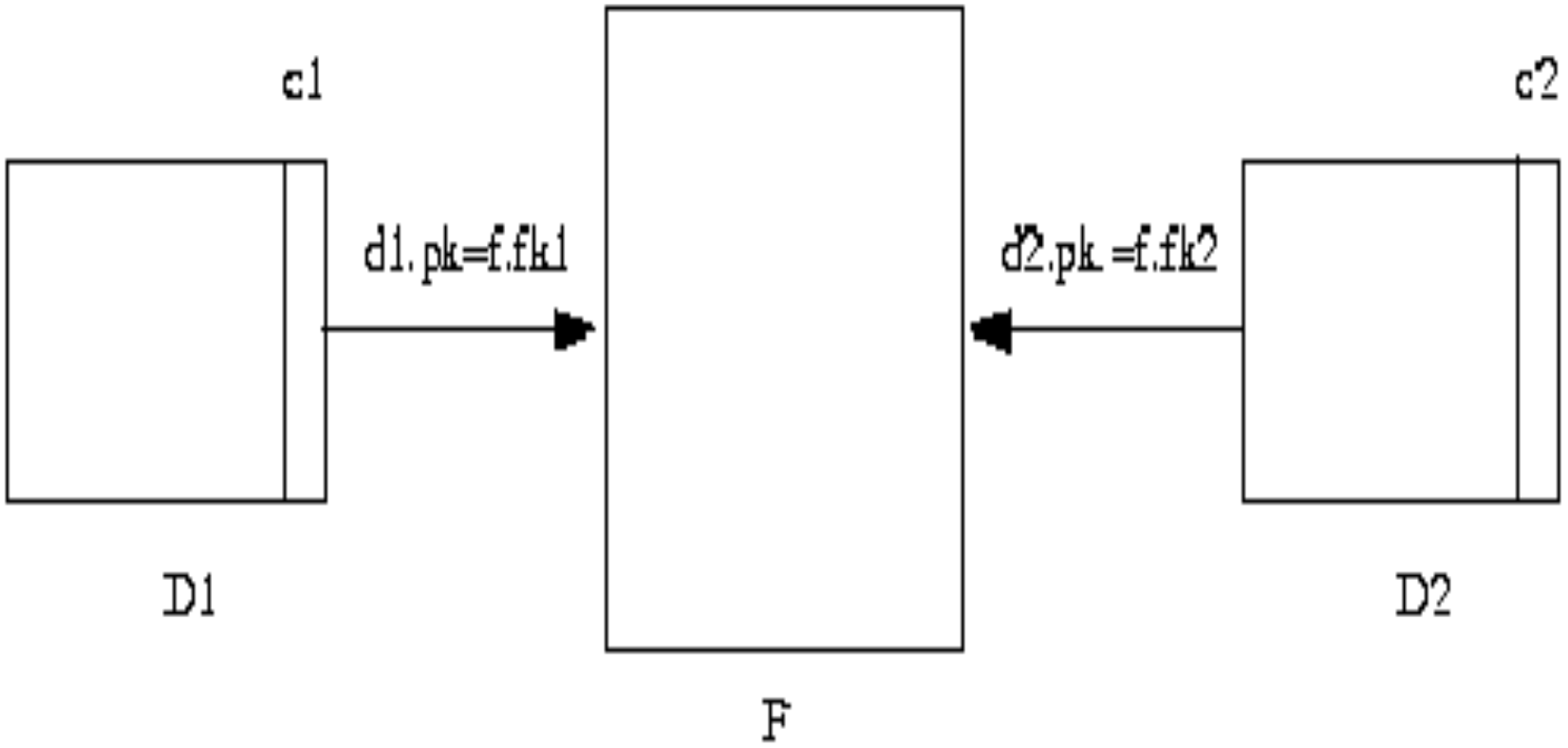
```
select sum(f.sales) from d, f
where d.pk = f.fk and d.c1 = 1 and d.c2 = 3;
```

can be retrieved by accessing the bitmap join index bji.

Another query which references only the leading portion of the index key can also use bitmap join index *bji*:

```
select sum(f.sales) from d, f
where d.pk = f.fk and d.c1 = 1;
```

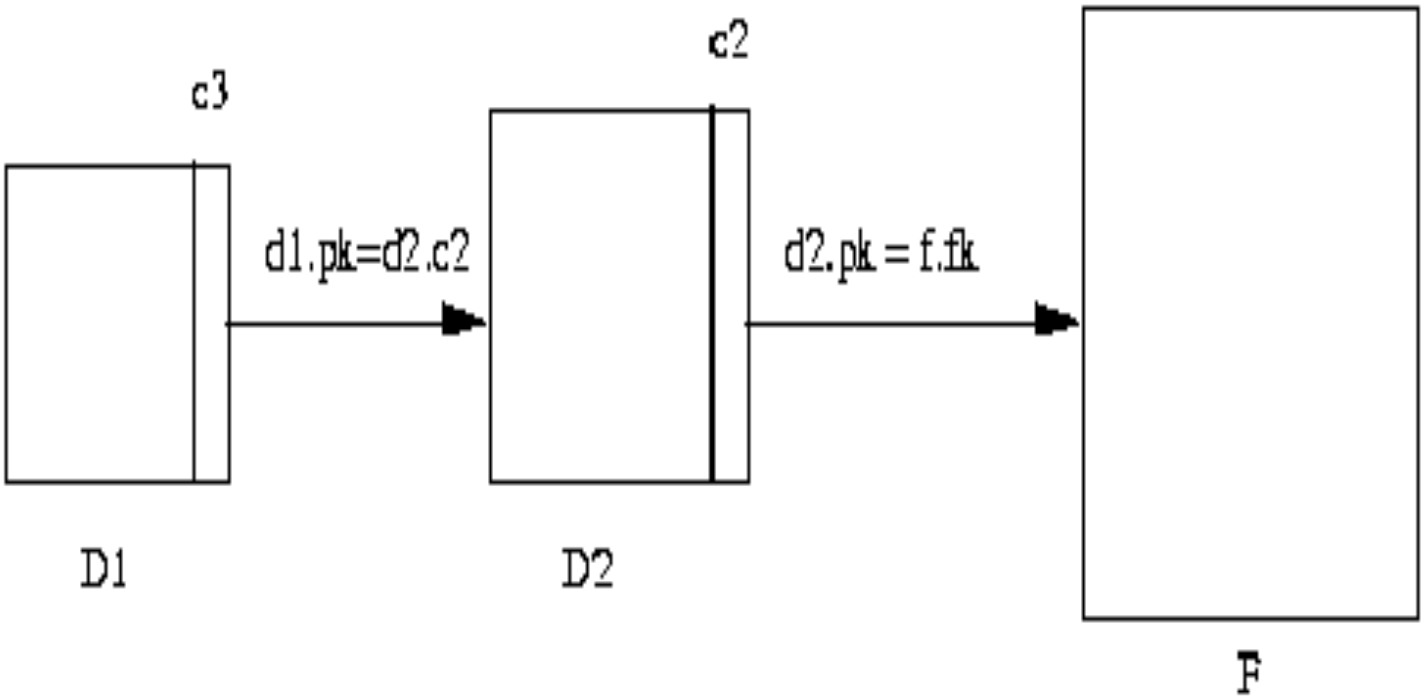
Multiple dimension tables join with one fact table



Above figure shows the third model, which requires a concatenated bitmap join index:

```
CREATE BITMAP INDEX bji ON f (d1.c1, d2.c2) FROM f, d1, d2
WHERE d1.pk = f.fk1 and d2.pk = f.fk2
```

Snow Flake Schema



- The above model involves joins between two or more dimension tables.
- A bitmap join index on `d1.c3` with a join between `d1` and `d2` and a join between `d2` and `f` can be created as follows:

```
CREATE BITMAP INDEX bji
ON f (d1.c3) FROM f, d1, d2
WHERE d1.pk = d2.c2 and d2.pk = f.fk;
```


Macneil Fernandes©2005