Multi-Attribute Indexing

Database Systems Concepts
Silberschatz/ Korth
Sec. 11.9

Fundamentals of Database Systems
Elmasri/Navathe
Sec. 6.4

The Design and Analysis of Spatial Data Structures
Samet

BBIT4/SEM4 Advanced Database Systems

Overview

- **Query Types**
  - **Exact Match**
  - **Partial Match**
  - **Range Match**
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
- Bitmap Index

2

© Stephen Mc Kearney, 2002.

Query Types

- Three important types of query
  - Exact Match Query
  - Partial Match Query
  - Range Match Query
- Query type determines:
  - Search strategy
  - Indexes used

3

- There are three types of query:
  1. Exact Match Queries
  2. Partial Match Queries
  3. Range Match Queries
- The type of the query is important as it determines:
  - **Search Strategy** The type of the query determines whether the DBMS is searching for a single record, a number of records or a sequence of records in a range.
  - **Index Strategy** The type of the query determines whether the DBMS may use an index to search the relation for the required tuples.

Exact Match Queries

- The query matches with a single tuple.

  e.g.

  account(<u>account_number</u>, customer_name, branch_name, balance)

```
SELECT *
FROM account
WHERE account_number = 123456789;
```

Only one tuple has *account_number=123456789*

4

- An exact match query specifies enough information about the content of the tuple to allow the DBMS to identify a single tuple.
- This is normally achieved by specifying a primary key value.
- If there is an index on the primary key then it may be used to answer the query.

© Stephen Mc Kearney, 2002.

Partial Match Queries

- The query matches with a subset of tuples.
  e.g.

account(account_number, customer_name, branch_name, balance)

```
SELECT *
FROM account
WHERE branch_name = 'Bournemouth';
```

Many tuples may have *branch_name='Bournemouth'*

5

- A partial match query specifies enough information about the content of the tuple to allow the DBMS to identify a set of tuples.
- A partial match query occurs when a non-key attribute is included in the WHERE clause.
- If there is an index on the attribute then it may be used to answer the query. However, the index will contain duplicate values (this is possible in a B+-Tree).

BBIT4/SEM4 Advanced Database Systems

Range Match Queries

- The query matches with a range of tuples.
  e.g.

account(account_number, customer_name, branch_name, balance)

```
SELECT *
FROM account
WHERE balance > 1500;
```

Many tuples may have *balance > 1500*

6

- A range match query specifies the lower/upper limits of a range of values in a set of tuples.
- A range match query can occur with a key or non-key attribute in the WHERE clause.
- If there is an index on the attribute then it may be used to answer the query. A B+-Tree is suitable for certain types of range query.

© Stephen Mc Kearney, 2002.

Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- **Strategies for Querying and Indexing**
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
- Bitmap Index

7

Strategies for Indexing 1

- Given a query:

```
SELECT *
FROM account
WHERE branch_name = 'Bournemouth'
AND balance = 1000;
```

- Strategy:
  1. Use index to search for *branch_name='Bournemouth'*
  2. Search result of step (1) for *balance=1000*

8

- The query is a partial match query looking for all accounts in the Bournemouth branch with a balance of 1000.
- The first strategy for answering this query is:
  1. Use an index on *branch_name* to retrieve the set of tuples with *branch_name='Bournemouth'*.
  2. Search the set of tuples, resulting from step 1, for those tuples with *balance=1000*.
- Problem
  - If many records contain *branch_name='Bournemouth'*, step 2 will involve searching many records (a serial/sequential search).

Ref: Silberschatz, sec 11.9

© Stephen Mc Kearney, 2002.

Strategies for Indexing 2

- Given a query:

```
SELECT *
FROM account
WHERE branch_name = 'Bournemouth'
AND balance = 1000;
```

- Strategy:
  1. Use index to search for *balance=1000*
  2. Search result of step (1) for *branch_name='Bournemouth'*

9

- The query is a partial match query looking for all accounts in the Bournemouth branch with a balance of 1000.
- The second strategy for answering this query is:
  1. Use an index on *balance* to retrieve the set of tuples with *balance=1000*.
  2. Search the set of tuples, resulting from step 1, for those tuples with *branch_name='Bournemouth'*.
- Problem
  - If many records contain *balance=1000*, step 2 will involve searching many records (a serial/sequential search).

Ref: Silberschatz, sec 11.9

Strategies for Indexing 3

- Given a query:

```
SELECT *
FROM account
WHERE branch_name = 'Bournemouth'
AND balance = 1000;
```

- Strategy:
  1. Use index to search for *branch_name = 'Bournemouth'*
  2. Use index to search for *balance=1000*
  3. Take the intersection of the results of steps (1) and (2)

10

- The query is a partial match query looking for all accounts in the Bournemouth branch with a balance of 1000.
- The third strategy for answering this query is:
  1. Use an index on *branch_name* to retrieve the set of tuples with *branch_name='Bournemouth'*
  2. Use an index on *balance* to retrieve the set of tuples with *balance=1000*.
  3. Match the results of steps (1) and (2) to produce a set of tuples that have both *branch_name='Bournemouth'* and *balance=1000*.
- Problem
  - If there are many tuples in (1) and (2) but few tuples containing both *branch_name=*'Bournemouth' and *balance=1000*, step 3 will involve searching many records (a sort/merge).

Ref: Silberschatz, sec 11.9

Strategies for Indexing 4

- Given a query:

```
SELECT *
FROM account
WHERE branch_name = 'Bournemouth'
AND balance = 1000;
```

- Strategy:
  1. Use index on (*branch_name, balance*) to search for *branch_name = 'Bournemouth'* and *balance=1000*

11

- The query is a partial match query looking for all accounts in the Bournemouth branch with a balance of 1000.
- The fourth strategy for answering this query is to use an index that has been built using the key (*branch_name, balance*).
- A B+-Tree index on a composite key is the equivalent of indexing a key constructed from the concatenation of both *branch_name* and *balance*.
- Given the query above, an index on the composite key (*branch_name, balance)* will always identify a matching set of tuples.

Ref: Silberschatz, sec 11.9

BBIT4/SEM4 Advanced Database Systems

Overview

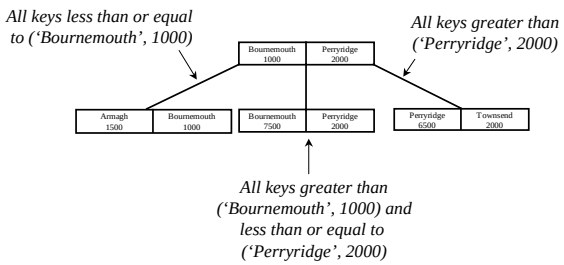
- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- **Composite Key Index**
- Visualising the Index Structure
- Multi-Attribute Index
- Bitmap Index

12

© Stephen Mc Kearney, 2002.

Composite Key Index

B+-Tree on a Composite Key



13

- A B+-Tree on a composite key indexes the key values as if the composite key was a single key.
- For example, in the B+-Tree above, (Bournemouth, 1000) is less than or equal to (Bournemouth, 1000) and so it appears in the first leaf node. However, (Bournemouth, 7500) is greater than (Bournemouth, 1000) and so it appears in the second leaf node.
- The *order* of the individual attributes in the composite key is important. The same as sorting a sentence of words alphabetically.
- "The ordering of the search-key values is the lexicographic ordering. For example, for the case of two attribute search keys, (a1,a2) < (b1,b2) if either a1 < b1, or a1 = b1 and a2 < b2." [Silberschatz et al]
- For example, although the second value of (Armagh, 1500) is greater than the second value of (Bournemouth, 1000), the order of the attributes means that (Armagh, 1500) is less than (Bournemouth, 1000).
- Therefore, the above B+-Tree may be used to search for (*branch_name*) or (*branch_name*, *balance*) but *not* (*balance*). For example, *balance=2000* appears in two paths of the B+-Tree.

Ref: Silberschatz, sec 11.9

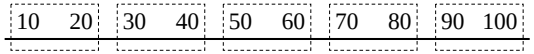
Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- **Visualising the Index Structure**
- Multi-Attribute Index
- Bitmap Index

14

Visualising a Single-Attribute Index

One Dimension



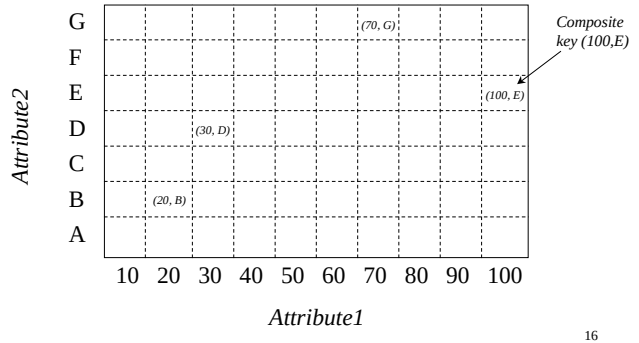
Splitting the data into blocks can only be done on one dimension

15

- A B+-Tree is a single-attribute, single-key, index. That is, it can only index one attribute or a concatenation of attributes.
- This may be thought of as a one-dimensional index.
- The data is ordered on a single dimension and the B+-Tree splits the dimension into a sequence of blocks.
- Therefore, the order of the attributes in a composite index is important.

Visualising a Multi-Attribute Index

Two Dimensions

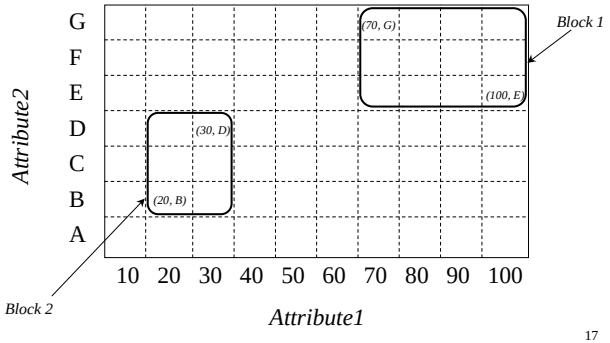


16

- A multi-attribute index has many dimensions. In the example above there are two dimensions.
- If an index was constructed using two dimensions, then searching for Attribute1 = 100 would only require the Attribute1=100 dimension to be searched.
- The problem is how to divide this two dimensional (or n-dimensional) space into a series of blocks that can be stored on the disc.
- Every point on the grid could be a single block but then only one value would be in each block.
- The two-dimensional space must be partitioned into one or more blocks that contain as many keys as possible.

Partitioning a Multi-Attribute Index

Two Dimensions



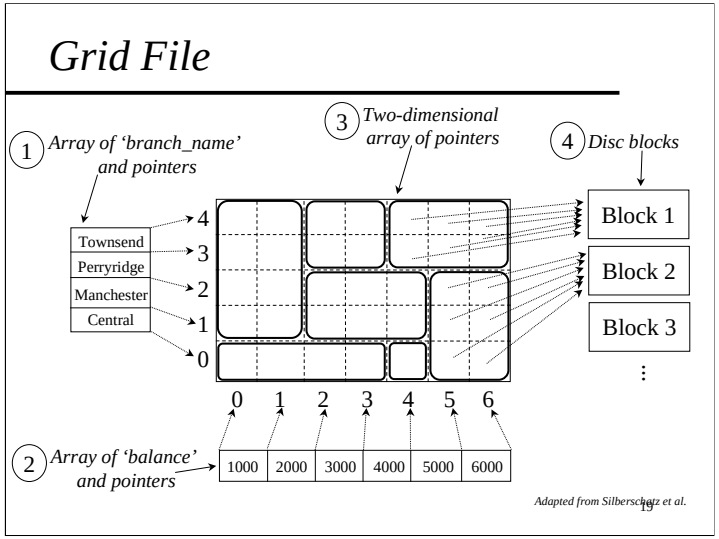
17

- Ideally, the space in a multi-attribute index should be partitioned into the fewest blocks required to store the key values.
- In the above example, there are two blocks. Each block contains two record pointers.
- However, we need a method of deciding how to partition the space and how to track which records are in which blocks.

Overview

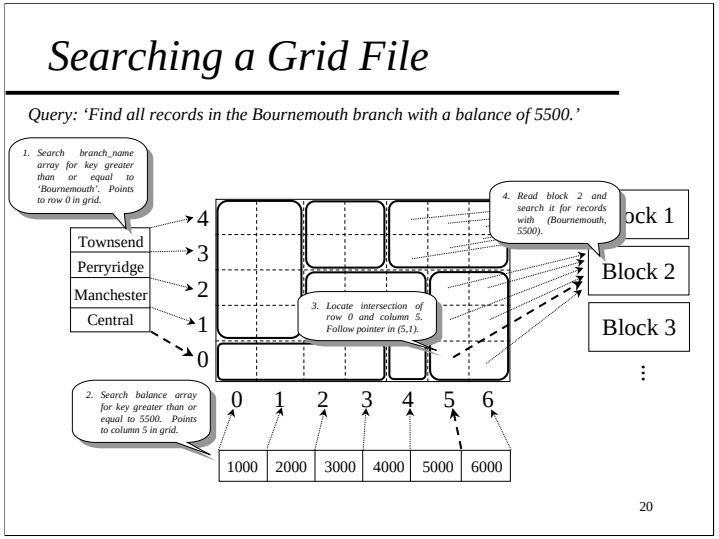
- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
  - **Grid File**
- Bitmap Index

18



- A grid file consists of three components:
  - A series of blocks on disc that contain the data records (4).
  - A multi-dimensional array of pointers (3). A grid of pointers to disc blocks is stored. Each position in the grid corresponds to one combination of attribute values. More than one position in the grid may point to the same disc block. In the above example, positions (4,3), (4,4), (5,3), (5,4), (6,3) and (6,4) all point to block 1.
  - An array of key values and pointers for each attribute (1 & 2). The values of each attribute is stored in an array (similar to the directory in an extendible hashing index) together with a pointer. The pointer points to a row or column in the grid. In the above example, the first location in the 'branch_name' array (1) points to row 0 in the grid.
- The above example shows a grid file with two attributes. However, a grid file may have any number of attributes. Each additional attribute requires an additional array and a new dimension for the grid.

Ref: Silberschatz, sec 11.9.1

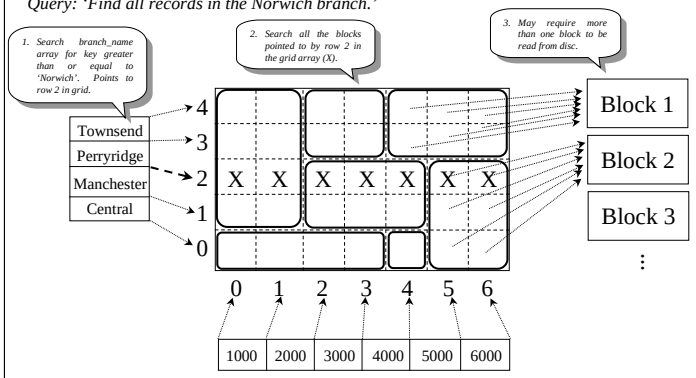


- Searching a grid file involves searching each attribute array, finding the corresponding positions in the grid array and reading the blocks identified by the grid array.

Ref: Silberschatz, sec 11.9.1

Searching a Grid File

Query: 'Find all records in the Norwich branch.'



21

- Unlike a one-dimensional index (e.g. the B+-Tree), a grid file can be used to answer queries on any subset of attributes in the index.
- In above example, the index is used to search on the *branch_name* only. The number of blocks that have to be read from disc is reduced to only those blocks pointed to by row 2 in the grid array (i.e. three blocks).
- As with the B+-Tree the grid file can be used to perform sequential searches. For example, searching for all balances less than 3000 would search blocks pointed to in columns 0, 1 and 2.

Ref: Silberschatz, sec 11.9.1

BBIT4/SEM4 Advanced Database Systems

Grid File

- Advantages
  - Allows multiple attributes to be indexed.
  - Allows exact match queries.
  - Allows partial match queries.
  - Allows range queries.
- Disadvantages
  - Requires additional space to store arrays.
  - Insertion and deletion can be difficult.
  - When blocks are full, the grid array must be reorganised.

22

- The main advantage of the grid file is that it supports exact match queries, partial match queries and range queries.
- This means that a single gird file can replace a number of ordinary indexes.
- However, the grid file is more complex than normal indexes and requires more space to store the various arrays and grids.
- When a block is full it must be split. Splitting a block requires the pointers in the grid array to be reorganised. Selecting the point to split a block can be difficult as the block may have to be split on more than one attribute.
- It is important that the keys in the index have uniform values.

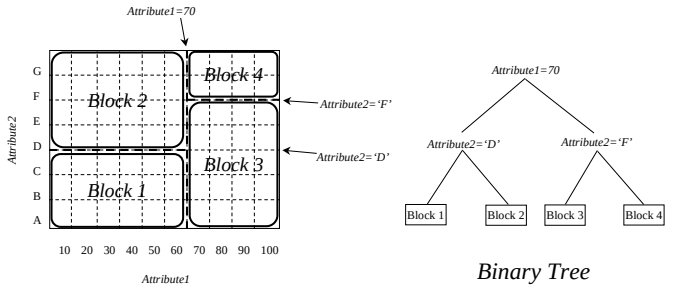
© Stephen Mc Kearney, 2002.

Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
  - **K-D Tree**
- Bitmap Index

23

K-D Tree

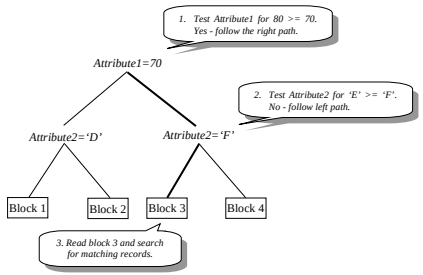


24

- A K-D Tree is a multi-attribute index that builds a binary tree. Each split in the tree divides one of the attributes in half.
- In the above example, the tree is first split at *attribute1=70*. All keys that are less than 70 are in the left-hand tree and all keys that are greater than or equal to 70 are in the right-hand tree.
- The K-D Tree is dynamic. When a block is full it is split by taking the mid-point of one of the attributes. To provide better distribution of the data in the tree, each level on the K-D Tree splits on a different attribute.

Exact Match Search in a K-D Tree

Query: Find all records with Attribute1=80 and Attribute2='E'.

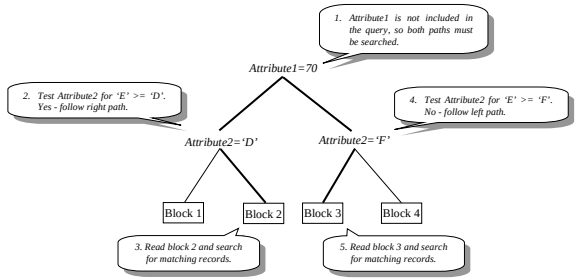


25

- When searching a K-D Tree for an exact match only one path is followed in the tree.

Partial Match Search in a K-D Tree

Query: Find all records with Attribute2='E'.



26

- When searching a K-D Tree for a partial match more than one path may be followed in the tree.

© Stephen Mc Kearney, 2002.

K-D Tree

- Advantages
  - The number of blocks grows as the data is added.
  - The index adapts to the distribution of values.
  - Exact match, partial match and range queries may be performed.
- Disadvantages
  - Special procedures for insert and delete are required.
  - The tree is not balanced.

27

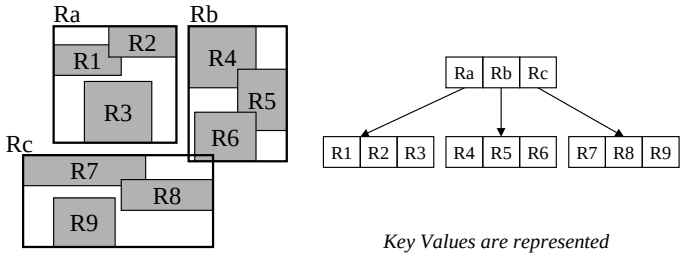
Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
  - **R-Tree**
- Bitmap Index

28

© Stephen Mc Kearney, 2002.

R-Tree - A Spatial Access Method

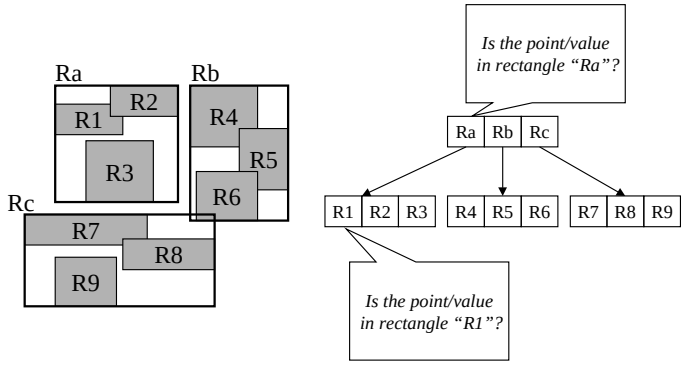


Key Values are represented as rectangles, for example,

R1 = (x:10, y:60, w:40, h:10)

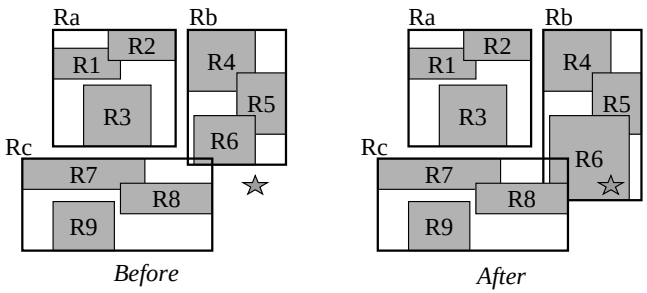
Adapted from Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, Y. Theodoridis: R-trees Have Grown Everywhere

R-Tree - Search



Unlike a B+-Tree, R-Tree nodes can overlap so a search may take many paths down the tree

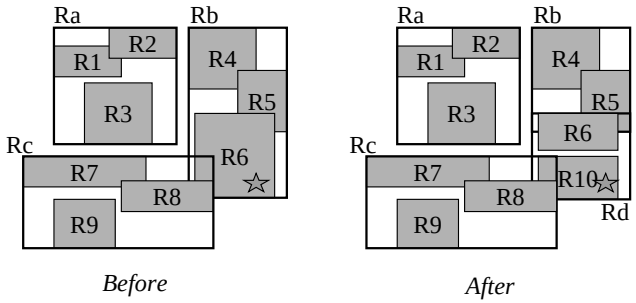
R-Tree - Insert



Extend the node that requires the smallest change to include the new key

BBIT4/SEM4 Advanced Database Systems

R-Tree - Node Full & Split



Extend the node that requires the smallest change to include the new key

© Stephen Mc Kearney, 2002.

R-Tree

- Advantages
  - Tree structure
  - Similar to B^+-Tree
  - Good search performance
  - Relatively simple to implement
  - Lots of variations
    - R^+-Tree
    - R^*-Tree
- Disadvantages
  - Not always appropriate for alphanumeric data
    - Designed for spatial data
      - Map co-ordinates
  - Nodes can overlap
    - Searches can cover more than one path
  - Split algorithms can be complex

33

Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
  - **Partitioned Hashing**
- Bitmap Index

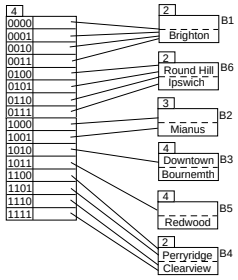
34

© Stephen Mc Kearney, 2002.

Partitioned Hashing

Extendible Hash Index

Values	Hash
(Bournemouth, 1000)	10 10
(Downtown, 1200)	10 10
(Manchester, 1500)	10 00
...	



Hash function returns binary code with two parts.

35

- A partition hash function takes one or more attributes and returns a sequence of binary bits.
- In the example above, the the binary code for each set of attributes values has two parts - one for each attribute value. When the bits are combined they can be used to insert the key values in an extendible hashing index.
- When querying, if only one attribute value is known, then those entries in the extendible hashing index matching the sequence of known binary bits are searched.
- For example, if the query requires branches in Bournemouth, then using the partition hash function produces 10 for Bournemouth and all entries in the directory beginning with 10 are searched. If the query requires balances of 1500, then using the partition hash function produces 00 and all entries in the directory ending in 00 are searched.
- The partitioned hash function cannot be used for range searching.

Ref: Silberschatz, sec 11.9.2

BBIT4/SEM4 Advanced Database Systems

Overview

- Query Types
  - Exact Match
  - Partial Match
  - Range Match
- Strategies for Querying and Indexing
- Composite Key Index
- Visualising the Index Structure
- Multi-Attribute Index
  - Partitioned Hashing
- **Bitmap Index**

36

© Stephen Mc Kearney, 2002.

Table Suitable for Bitmap Index

CUST #	MARITAL	REGION	GENDER	INCOME_ LEVEL
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

Most attributes have low cardinality

A B+-Tree would be very large on these attributes and not very efficient.

37

Bitmap Index on Region

Cust#	REGION='east'	REGION='central'	REGION='west'
101	1	0	0
102	0	1	0
103	0	0	1
104	0	0	1
105	0	1	0
106	0	1	0

A bit is 1 if the tuple has the value represented by the column.

Very little space is required to store these bits.

38

Executing a Query

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

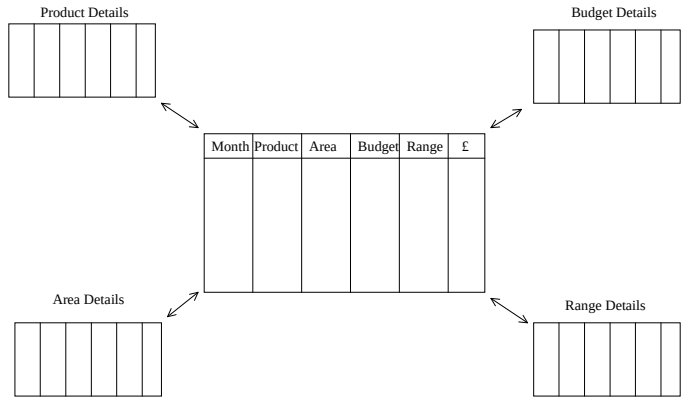
status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

101
102
103
104
105
106

Very efficient bit matching

39

Star Schema



40