데이터베이스 파일 인덱싱 기법: part 2

다중키 인덱스(Indexes on Multiple Keys)

- EMPLOYEE : Ssn, Dno (department number), Age , Street , City , Zip_code , Salary, Skill_code
- Query: **List the employees in department number 4 whose age is 59**.
  - Case 1: No index on Dno and Age
  - Case 2: Dno has an index, but Age does not
  - Case 3: Age has an index, but Dno does not
  - Case 4: Both Dno and Age are indexed

다중 애트리뷰트의 순서 인덱스

- 순서가 중요하다: (Dno, Age) or (Age, Dno)
- Why?

분할해싱(Partitioned Hashing)

- 다중키에 대한 접근을 허용하는 정적 외부 해싱의 확장
- 해싱이므로 동등 비교(point query)에만 사용, 범위질의(range query)에는 사용 불가능
  - 다중 키 < Dno , Age >를 고려해 보자.
  - Dno가 3 비트로 해시되고, and Age가 5 비트로 해시되면, 총 8 비트로 구성된 버켓 주소를 가지게 된다.
- "Age = 59"인 직원을 탐색하려면?

그리드 파일(Grid File)

- 각 탐색 애트리뷰트에 대해 하나의 선형 눈금(또는 차원)을 갖는 그리드 배열을 구성한다.
- 각 눈금은 해당 애트리뷰트에 대해 균등 분포를 갖도록 만들어진다.
- 각 셀은 셀에 대응하는 레코드들이 저장된 버켓 주소를 가리킨다.
- 선형 눈금자, 그리드 배열은 모두 선형 구조이므로 디스크에 쉽게 저장할 수 있다.
- "Dno $\leq$ 5 and Age > 40"인 직원을 찾아보자
- 동적 파일인 경우, 파일 재조직의 비용이 많이 든다.

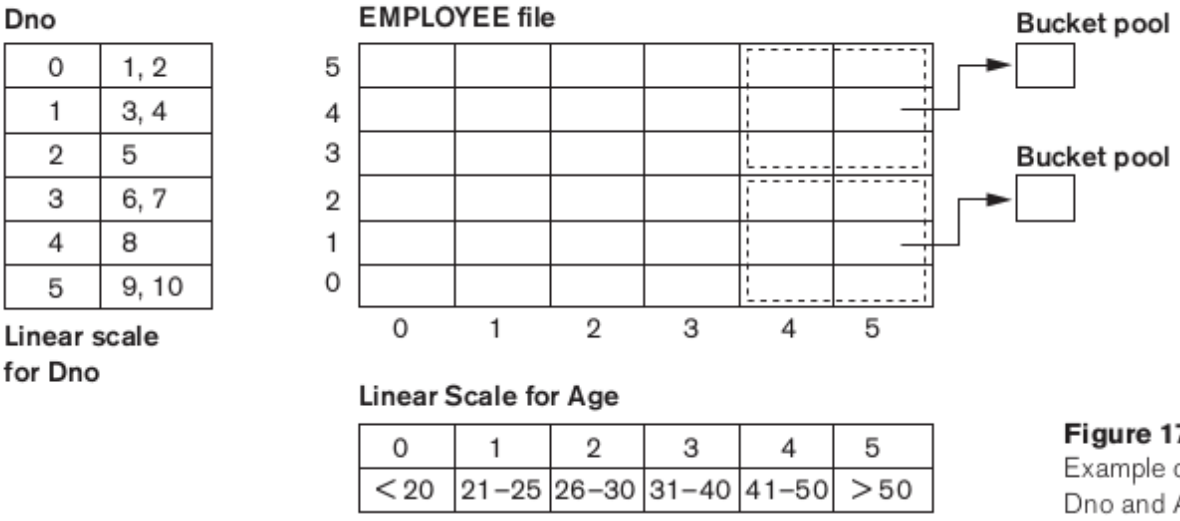
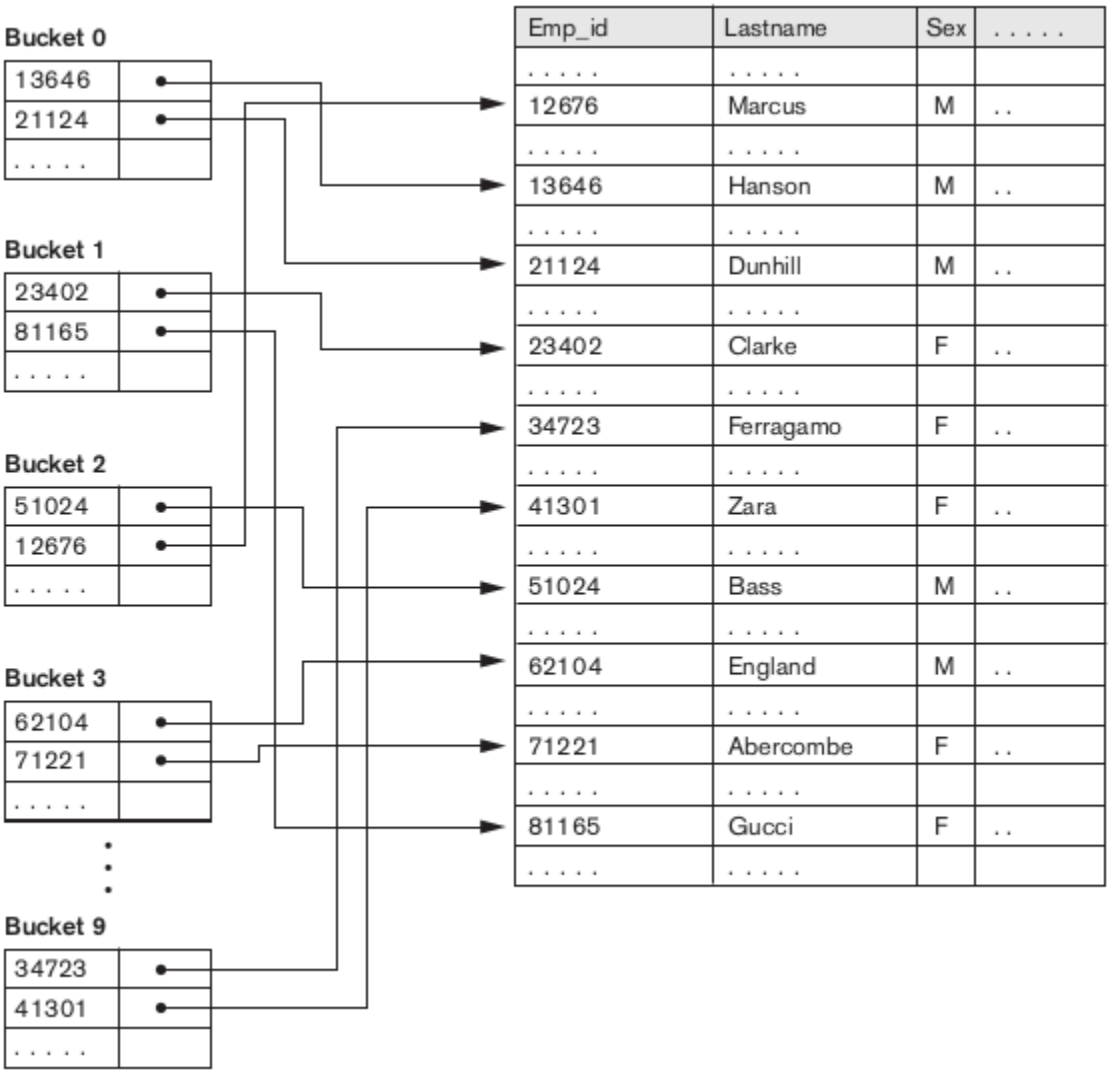


Figure 17.14
Example of a grid array on Dno and Age attributes.

해시 인덱스

- 해시 인덱스를 보조 인덱스로 사용
- CREATE INDEX

Figure 17.15
Hash-based indexing.



비트맵 인덱스(Bitmap index)

- 다중 키에 대한 질의에 널리 사용되는 자료 구조
- 적은 수의 유일한 값을 갖고, 많은 개수의 행을 가진 릴레이션에 사용
- 레코드들은 0부터 n까지의 숫자를 갖는 id를 가져야 한다.
- id를 블록 주소와 블록안의 offset으로 mapping할 수 있는 방법이 있어야 한다.: 예) 상대 화일 구조
- 열 C의 값 V에 대한 비트맵 인덱스: 행 i가 열 C의 값 V를 가지면 1 그렇지 않으면 0이다.

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Figure 17.16
Bitmap indexes for Sex and Zipcode.

- "Sex = 'M' and Zipcode = '19046'"인 직원을 찾아보자.
- 조인 조건은 비트맵의 불리안 연산으로 대체될 수 있다.
- 5개의 서로 다른 값을 가진 열과 10개의 서로 다른 값을 가진 열을 조인 조건은 2%의 선택율(selectivity)을 가진다.
- 기본 구조가 히프 인 경우 삽입 비용은 적다.
- 삭제 비용을 줄이기 위해, 삭제시 해당 행에 대한 모든 비트를 0으로 처리 한다. 추후 파일 재구성에서 이를 삭제할 수 있다.
- B^+ 트리의 리프 노드로 사용될 수 있다. 포인터에 대한 압축 표현 방식으로 사용된다.
- Oracle supports bitmap index, MySQL 8.0 does not.
  - create bitmap index genderbitmap on employees(sex)

함수 기반 인덱싱

- Oracle, MySQL 8.0이 지원함
- 함수를 적용한 결과 값들이 인덱스의 키가 된다.
  - Example Query 1:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH"
```

  - Example 1 Index:

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));
```

  - Example Query 2:

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000
```

  - Example 2 Index:

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct))
```

Factors That Influence Physical Database Design

- Analyzing the Database Queries and Transactions.
- Analyzing the Expected Frequency of Invocation of Queries and Transactions.
  - 80-20 rule: approximately 80% of the processing is accounted for by only 20% of the queries and transactions.
- Analyzing the Time Constraints of Queries and Transactions.
  - Example: a transaction may have the constraint that it should terminate within 5 seconds on 95% of the occasions when it is invoked, and that it should never take more than 20 seconds
  - The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files
- Analyzing the Expected Frequencies of Update Operations.
  - The overhead for updating 10 indexes can slow down the insert operations.
- Analyzing the Uniqueness Constraints on Attributes.
  - The existence of an index makes it sufficient to search only the index when checking this uniqueness constraint

Physical Database Design Decisions

- Whether to index an attribute.
  - The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or therenmust be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition.
  - One reason for creating multiple indexes is that some operations can be processed by just scanning the indexes, without having to access the actual data file.
- What attribute or attributes to index on.
  - An index can be constructed on a single attribute, or on more than one attribute if it is a composite index. - If multiple attributes from one relation are involved together in several queries, (for example, (Garment*style*#, Color) in a garment inventory database), a multiattribute (composite) index is warranted.
  - The ordering of attributes within a multiattribute index must correspond to the queries.
  - For instance, the above index assumes that queries would be based on an ordering of colors within a Garment*style*# rather than vice versa.
- Whether to set up a clustered index.
  - At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute.
  - In most RDBMSs, this is specified by the keyword CLUSTER . (If the attribute is a key, a primary index is created, whereas a clustering index is created if the attribute is not a key.)
  - If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed.
  - Range queries benefit a great deal from clustering.
  - If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on.
  - If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should not be clustered, since the main benefit of clustering is achieved when retrieving the records themselves.
  - A clustering index may be set up as a multiattribute index if range retrieval by that composite key is useful in report creation (for example, an index on Zip_code , Store_id , and Product_id may be a clustering index for sales data).
- Whether to use a hash index over a tree index.
  - In general, RDBMSs use B+-trees for indexing.
  - However, ISAM and hash indexes are also provided in some systems.
  - B+-trees support both equality and range queries on the attribute used as the search key.
  - Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.
- Whether to use dynamic hashing for the file.
  - For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes would be suitable. Currently, such schemes are not offered by many commercial RDBMSs.

Spatial Databases from Chap 26

- **Cartographic databases** that store maps include two-dimensional spatial descriptions of their objects from countries and states to rivers, cities, roads, seas, and so on.
- The systems that manage geographic data and related applications are known as **geographic information systems (GISs)**, and they are used in areas such as environmental applications, transportation systems, emergency response systems, and battle management.
- Other databases, such as **meteorological databases for weather information**, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points.

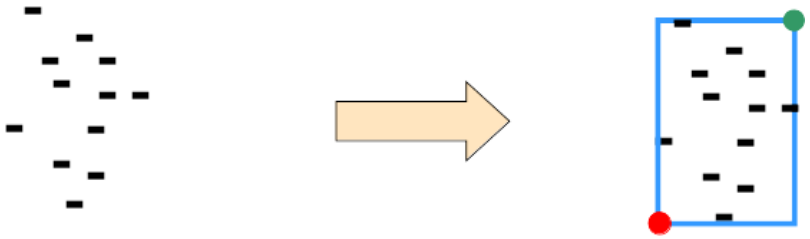
- Spatial queries. For example,
  - “What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?”
  - “List all the customers located within twenty miles of company headquarters”

- A traditional B + -tree index based on customers’ zip codes or other nonspatial attributes cannot be used to process this query since traditional indexes are not capable of ordering multidimensional coordinate data. Therefore, there is a special need for databases tailored for handling spatial data and spatial queries.

Spatial Queries.

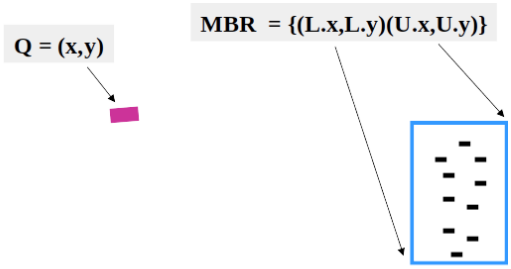
- Spatial queries are requests for spatial data that require the use of spatial operations. The following categories illustrate three typical types of spatial queries:
- **Range queries**. Find all objects of a particular type that are within a given spatial area;
  - for example, find all hospitals within the Metropolitan Atlanta city area.
  - A variation of this query is **to find all objects within a particular distance from a given location**
  - for example, **find all ambulances within a five mile radius of an accident location**.
- **Nearest neighbor queries**.
  - Finds an object of a particular type that is closest to a given location;
  - for example, **find the police car that is closest to the location of a crime**.
  - This can be generalized to find the k nearest neighbors, such as the **5 closest ambulances to an accident location**.
- **Spatial joins or overlays**.
  - Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another.
  - For example, **find all townships located on a major highway between two cities or find all homes that are within two miles of a lake**. The first example spatially joins township objects and highway object, and the second example spatially joins lake objects and home objects.

MBR: Minimum Bounding Rectangle



$$MBR = \{(L.x, L.y)(U.x, U.y)\}$$

Minimum Distance



MINDIST(Q,MBR)

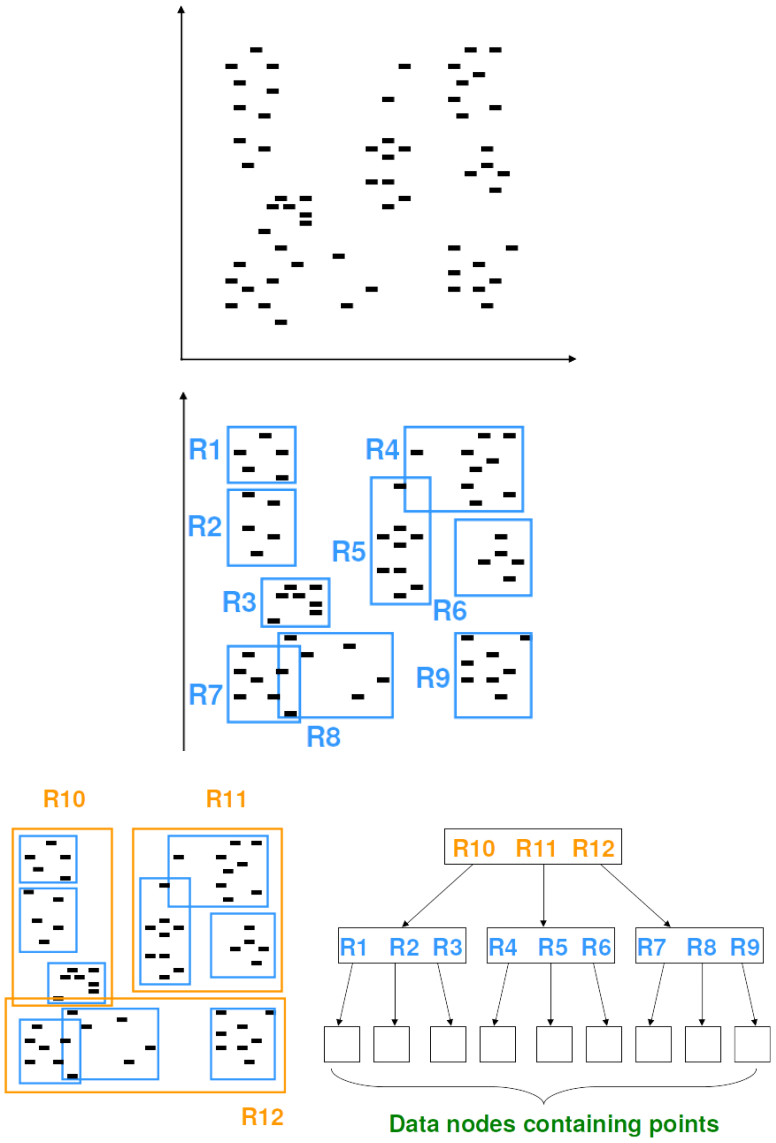
if $L.x < x < U.x$ **and** $L.y < y < U.y$ **then** 0
elseif $L.x < x < U.x$ **then** $\min((L.y - y)^2 , (U.y - y)^2)$
elseif ….

R-Trees.

- The R-tree is a height-balanced tree, which is an extension of the B + -tree for k-dimensions, where k > 1.
- For two dimensions (2-D), spatial objects are approximated in the R-tree by their **minimum bounding rectangle (MBR)**, which is the smallest rectangle, with sides parallel to the coordinate system (x and y) axis, that contains the object.
- We use M to indicate the maximum number of entries that can fit in an R-tree node.

1. The structure of each index entry (or index record) in a leaf node is (I, object-identifier), where I is the MBR for the spatial object whose identifier is object-identifier.
2. Every node except the root node must be **at least half full**. Thus, a leaf node that is not the root should contain m entries (I, object-identifier) where $M/2 \leq m \leq M$.
3. Similarly, a non-leaf node that is not the root should contain m entries (I, child-pointer) where $M/2 \leq m \leq M$, and I is the MBR that contains the union of all the rectangles in the node pointed at by child-pointer.
4. All leaf nodes are at the same level, and the root node should have at least two pointers unless it is a leaf node.
5. All MBRs have their sides parallel to the axes of the global coordinate system.

Construction of R-Trees



Search in R-Trees

- To find data items (rectangles/polygons) intersecting (overlaps) a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node **whose bounding box overlaps the query point/region**, recursively search the child
- Can be **very inefficient in worst case** since multiple paths may need to be searched (**the curse of dimensionality**)
  - but works acceptably in practice.
- Simple extensions of search procedure to handle predicates **contained-in** and **contains**

NN Query in R-Trees

- Think about it?

Insertion in R-Trees

- To insert a data item:
  - Find a leaf to store it, and add it to the leaf
    - To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child **whose overlap with data item bounding box is maximum**
      - Handle overflows by splits (as in B+-trees)
    - Split procedure is different though (see below)
      - Adjust bounding boxes starting from the leaf upwards
- Split procedure:
  - Goal: divide entries of an overfull node into two sets such that the bounding boxes have **minimum total area**
  - Finding the “best” split is **expensive**, use **heuristics** instead

Splitting an R-Tree Node

- Quadratic split divides the entries in a node into two new nodes as follows
  - Find pair of entries with **“maximum separation”**
  - Place these entries in two new nodes
  - Repeatedly find the entry with “maximum preference” for one of the two new nodes, and assign the entry to that node
    - Preference is the **smaller increase in area of bounding box** if the entry is added to the other node
  - Stop when half the entries have been added to one node
    - Then assign remaining entries to the other node

Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a B+-tree deletion.
  - In case of underfull node, **borrow entries from a sibling if possible, else merging sibling nodes**
  - Alternative approach removes all entries from the underfull node, deletes the node, then **reinserts all entries**
  - As always, deletion tends to be **rarer** than insertion for many real world databases.