Chapter 14: Indexing

Database System Concepts, 7th Ed.

Basic concepts

- "Find all instructors in the Physics department" or "Find the total number of credits earned by the student with *ID* 22201"
- Is it efficient to read entire relation of department or students to answer the query?
- No, rather locate the record directly.
- To allow this, we need additional structures associated with files.
- Index of a book?
- For example, to retrieve a *student* record given an *ID*, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.

Basic concepts

- Indices are critical for efficient processing of queries in databases.
- Implementing an index on the *student* relation by keeping a sorted list of students' *ID* would not work well on very large databases, since

(i) the index would itself be very big,

(ii) even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming, and

(iii) updating a sorted list as students are added or removed from the database can be very expensive

Basic concepts

- There are two basic kinds of indices:
- **Ordered indices:** Based on a sorted ordering of the values.
- **Hash indices**: Based on a uniform distribution of values across a range of buckets
- Each techniques must be evaluated on the basis of
  - Access type
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead

Basic concepts

- **Access types**: Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time**: The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

Basic concepts

- One file one index or more than one index?
- Search a book by - Title, Author, Subject
- Search key: An attribute or set of attributes used to look up records in a file is called a **search key.**

Order Indices

- An **ordered index** stores the values of the search keys in sorted order and associates with each search key the records that contain it.
- The records in the indexed file may themselves be stored in some sorted order,
- A file may have several indices, on different search keys.
- If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file. Clustering indices are also called **primary indices**;
- The term *primary index* may appear to denote an index on a primary key, but such indices can in fact be built on any search key. The search key of a clustering index is often the primary key, although that is not necessarily so.

- Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary indices**.
- An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value.
- The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the r

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Clustering index?

Non-Clustering index?

- Index (ID) → Record Pointer
- ---
- 10101 → Srinivasan
- 12121 → Wu
- 15151 → Mozart
- 22222 → Einstein
- 32343 → El Said
- 33456 → Gold
- 45565 → Katz
- 58583 → Califieri
- 76543 → Singh
- 76766 → Crick
- 83821 → Brandt
- 98345 → Kim

- Dept → List of Record IDs
- ---
- Biology → (76766)
- Comp. Sci. → (10101, 45565, 83821)
- Elec. Eng. → (98345)
- Finance → (12121, 76543)
- History → (32343, 58583)
- Music → (15151)
- Physics → (22222, 33456)

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

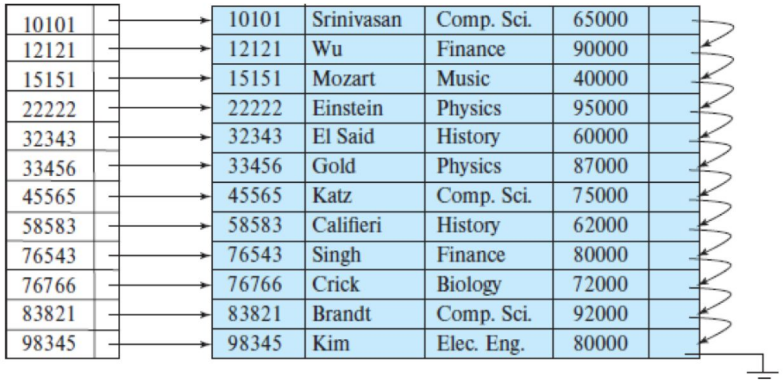
Dept → First Record ID

Biology → 76766
Comp. Sci. → 10101
Elec. Eng. → 98345
Finance → 12121
History → 32343
Music → 15151
Physics → 22222

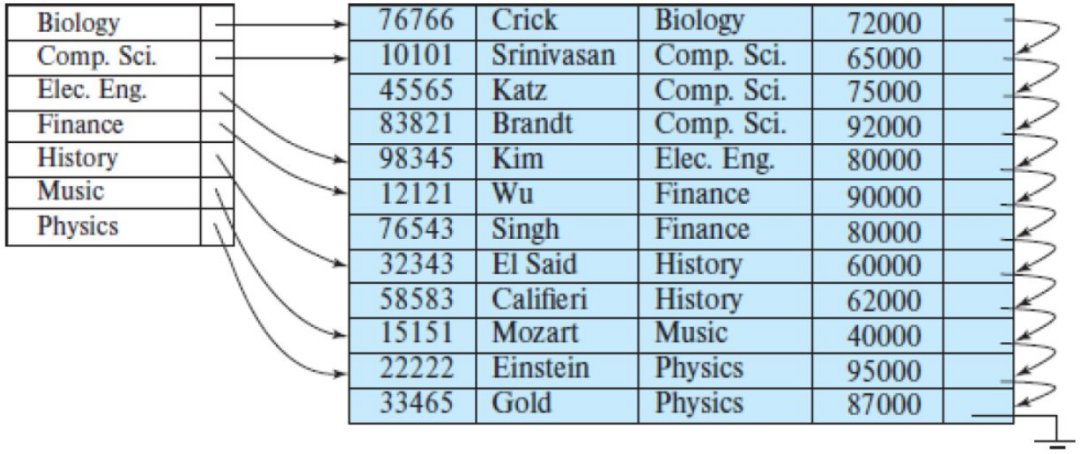
Feature	**Clustering Index**	**Non-Clustering Index**
Physical data storage	Data rows are stored in the **same order** as the index key	Data rows are stored **independently** of the index order
Number per table	Only **one** (because data can only be ordered one way)	Can be **many** (since they don't affect physical order)
Key storage	The index contains the **actual data rows** (leaf nodes store full records)	The index contains **pointers (row IDs)** to actual data rows
Performance	Very efficient for **range queries** and **sorting**	Efficient for **point lookups**, less optimal for ranges
Maintenance	More expensive for **insert/update/delete**, since data must remain ordered	Less costly to maintain, since order of data is unaffected
Default	Often created on the **Primary Key**	Usually created on **non-key attributes** for faster lookups
Example usage	Searching students with RollNo 100–200	Searching by Name or Dept (non-primary key fields)

- **Dense index**:
- In a dense index, an index entry appears for every search-key value in the file.
- In a dense clustering index, the index record contains the search key value and a pointer to the first data record with that search-key value.
- The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.

Order Indices



Dense

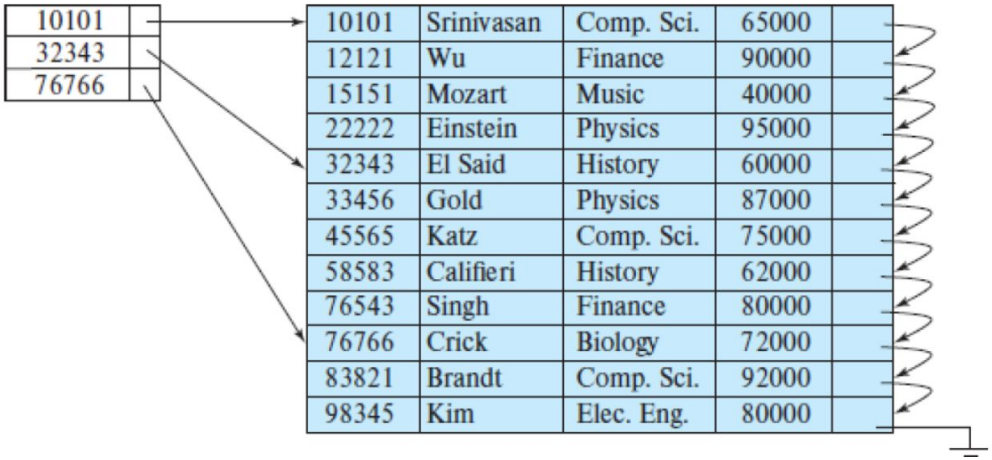


Dense

Order Indices

- **Sparse index**:
- In a sparse index, an index entry appears for only some of the search-key values.
- Sparse indices can be used only if the relation is stored in sorted order of the search key; that is, if the index is a clustering index.
- Each index entry contains a search-key value and a pointer to the first data record with that search-key value.
- To locate a record, we find the index entry with the largest search key value that is less than or equal to the search-key value for which we are looking.
- We start at the record pointed to by that index entry and follow the pointers in the file until we find the desired record.
- Access time VS Space overhead

Order Indices



Sparse

Multilevel indices

- Suppose we build a dense index on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4-kilobyte block. Thus, our index occupies 10,000 blocks.
- If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.
- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low.
- If the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required.
- The search for an entry in the index then requires several disk-block reads.
- Binary search can be used on the index file to locate an entry, but the search still has a large cost.
- For a 10,000-block index, binary search requires 14 random block reads

- To deal with this problem, we treat the index just as we would treat any other sequential file, and we construct a sparse outer index on the original index, which we now call the inner index,

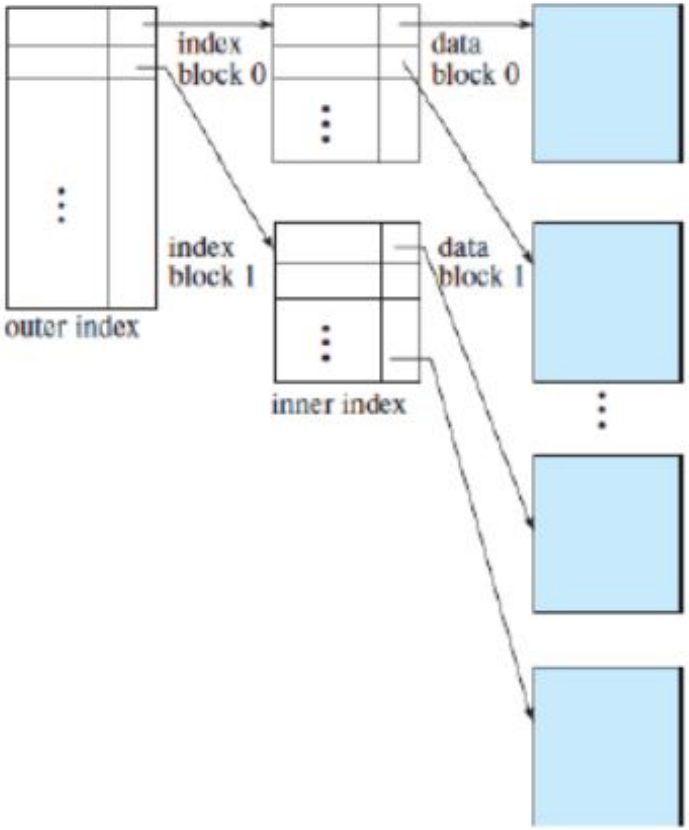


Figure 14.5 Two-level sparse index.

Index update (Insertion)

- Dense indices:
- **1.** If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
- **2.** Otherwise the following actions are taken:
  - a. If the index entry stores pointers to all records with the same search-key value, the system adds a pointer to the new record in the index entry.
  - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

Index update (Insertion)

- Sparse indices: We assume that the index stores an entry for each block.
  - If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index.
  - If the new record has the least search-key value in its block, the system updates the index entry pointing to the block;
  - if not, the system makes no change to the index.

Index update(Deletion)

- Dense indices:
- **1.** If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
- **2.** Otherwise the following actions are taken:
  - a. If the index entry stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index entry.
  - b. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.

Index update(Deletion)

- Sparse indices:
- **1.** If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
- **2.** Otherwise the system takes the following actions:
  - a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
  - b. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.