Indexing

COSC 404 – Database System Implementation

Indexing Overview

An ***index*** is a data structure that allows for fast lookup of records in a file.

An index may also allow records to be retrieved in sorted order.

Indexing is important for file systems and databases as many queries require only a small amount of the data in a file.

Index Terminology

The ***data file*** is the file that actually contains the records.

The ***index file*** is the file that stores the index information.

The ***search key*** is the set of attributes stored by the index to find the records in the data file.

- Note that the search key does not have to be unique - more than one record may have the same search key value.

An ***index entry*** is one index record that contains a search key value and a pointer to the location of the record with that value.

Evaluating Index Methods

Indexes are evaluated for functionality, efficiency, and performance.

The ***functionality*** of an index is measured by the types of queries it supports. Two query types are common:

- exact match on search key
- query on a range of search key values

The ***performance*** of an index is measured by the time required to execute queries and update the index.

- Access time, update, insert, delete time

The ***efficiency*** of an index is measured by the amount of space required to maintain the index structure.

Types of Indexes

Indexes on ***ordered*** versus ***unordered*** files

- An ordered file is sorted on the search key. Unordered file is not.

Dense versus ***sparse*** indexes

- A dense index has an index entry for every record in the data file.
- A sparse index has index entries for only some of the data file records (often indexes by blocks).

Primary (clustering) indexes versus ***secondary*** indexes

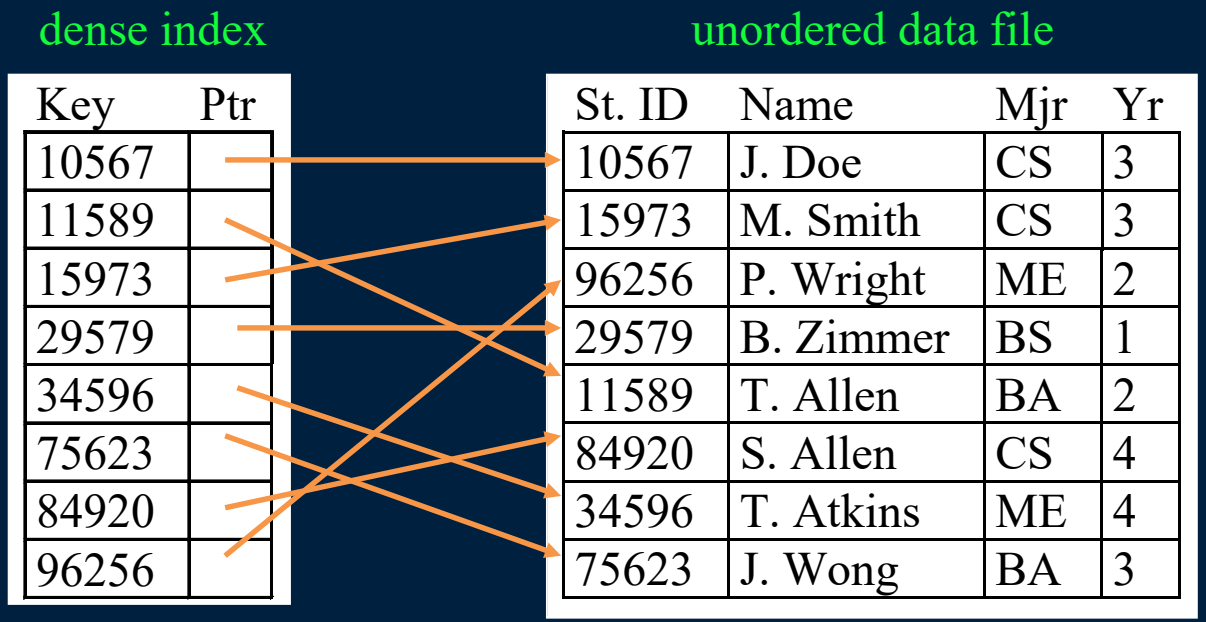
- A primary index sorts the data file by its search key. The search key ***DOES NOT*** have to be the same as the primary key.
- A secondary index does not determine the organization of the data file.

Single-level versus ***multi-level*** indexes

- A single-level index has only one index level.
- A multi-level index has several levels of indexes on the same file.

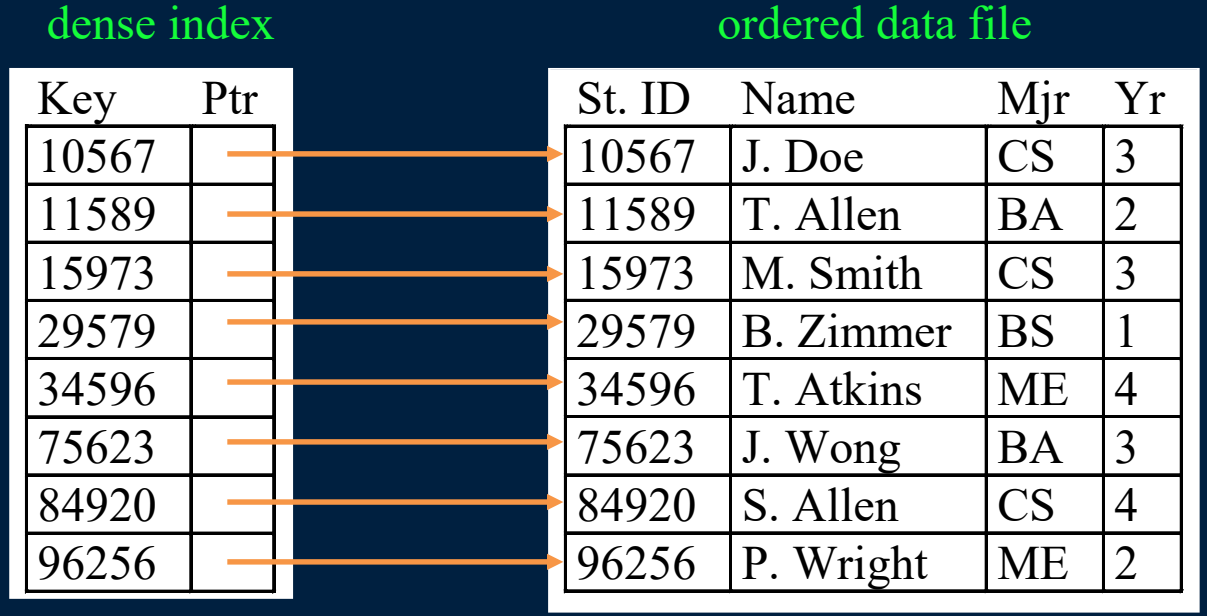
(Secondary) Index on Unordered File

Dense, single-level index on an unordered file



Primary Index on Ordered File

Dense, primary, single-level index on an ordered file



Index on Unordered/Ordered Files

An index on an unordered file makes immediate sense as it allows us to access the file in sorted order without maintaining the records in sorted order.

- Insertion/deletion are more efficient for unordered files.
  - Append record at end of file or move record from end for delete.
- Must only update index after data file is updated.
- Searching for a search key can be done using binary search on the index.

What advantage is there for a primary index on an ordered file?

- Less efficient to maintain an ordered file PLUS we must now also maintain an ordered index!

Answer: The index will be smaller than the data file as it does not store entire records. Thus, it may be able to fit entirely in memory.

Index Performance Example

We will calculate the increased performance of a dense index on an unordered/ordered file with the following parameters:

- Each disk block stores 4000 bytes.
- Each index entry occupies 20 bytes.
  - 10 bytes for search key, 10 bytes for record pointer
  - Index entries/block = 4000 bytes / 20 bytes/entry = 200 entries/block.
- Each record has size 1000 bytes.
  - Data records/block = 4000 bytes / 1000 bytes/record = 4 records/block.
- The data file contains 100,000 records.

How many block reads to retrieve a record based on its key?

How many block reads if have no index?

Index Performance Example (2)

Answer:

#dataBlocks = 100,000 records / 4 records/block = 25,000 blocks

#indexBlocks = 100,000 records / 200 entries/block = 500 blocks

Search index using a binary search = $\log_2 N = \log_2(500) = 8.97$ blocks

# of blocks retrieved = 9 index blocks + 1 data block = **10 blocks**

Time to find record using linear search (unordered file) = N/2

= 25,000 blocks/2 = **12,500 blocks** retrieved on average

Time to find record using binary search (ordered file) = $\log_2 N$

= $\log_2(25000) = 14.60$ blocks = **15 blocks**

Index Performance

Question: What statement is true for a non-empty, indexed table when searching for a single record?

A) Using an index is always faster than scanning the file if the data is on a hard drive

B) Using an index is always faster than scanning the file if the data is on a SSD

C) Binary searching an index is more suited to a hard drive than a SSD.

D) None of the above.

Sparse Index on Ordered Files

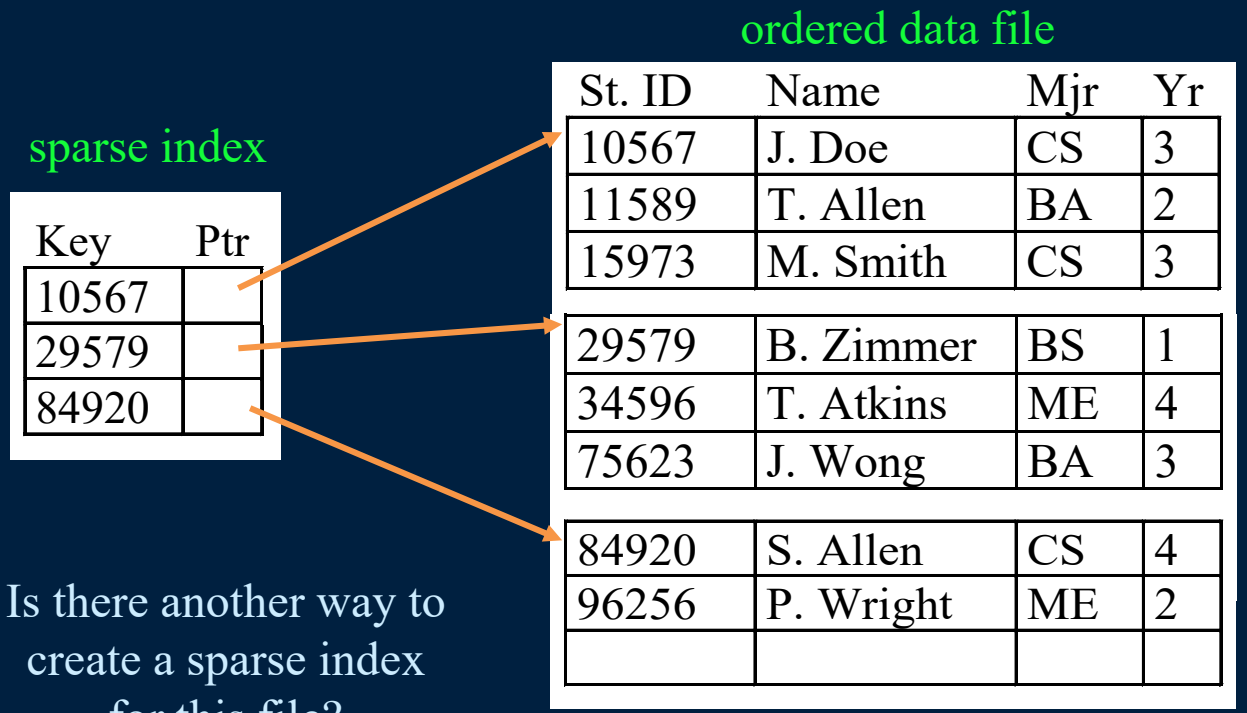
A ***sparse*** index only contains a subset of the search keys that are in the data file.

A better index for an ordered file is a sparse index since we can take advantage of the fact that the data file is already sorted.

- The index will be smaller as not all keys are stored.
  - Fewer index entries than records in the file.
  - Binary search over index can be faster as fewer index blocks to read than unordered file approach.

For an ordered file, we will store ***one search key per block*** of the data file.

Sparse Index on an Ordered File



Is there another way to create a sparse index for this file?

Sparse Index versus Dense Index

A sparse index is much more space efficient than a dense index because it only stores one search key per block.

- If a block can store 10 data records, then a sparse index will be 10 times smaller than a dense index!
- This allows more (or all) of the index to be stored in main memory and reduces disk accesses if the index is on disk.

A dense index has an advantage over a sparse index because it can answer queries like: does search key ***K*** exist? without accessing the data file (by using only the index).

- Finding a record using a dense index is easier as the index entry points directly to the record. For a sparse index, the block that may contain the data value must be loaded into memory and then searched for the correct key.

Index Performance Question

Calculate the performance of a sparse index on an **ordered** file with the following parameters:

- Each disk block stores 2000 data bytes.
- Each index entry occupies 8 bytes.
- Each record has size 100 bytes.
- The data file contains 1,000,000 records.

How many block reads to retrieve a record based on its key?

How many block reads for a dense index?

How many block reads if there is no index?

Multi-level Index

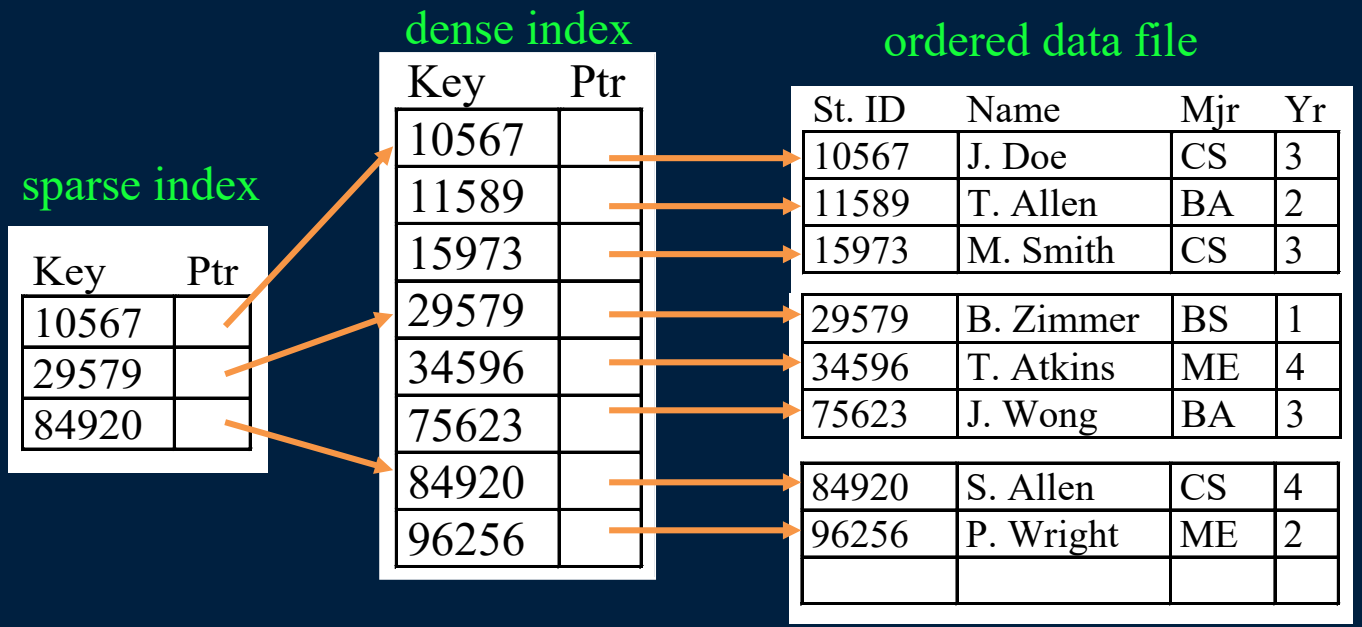
A ***multi-level index*** has more than one index level for the same data file.

- Each level of the multi-level index is smaller, so that it can be processed more efficiently.
- The first level of a multi-level index may be either sparse or dense, but all higher levels must be sparse. Why?

Having multiple levels of index increases the level of indirection, but is often quicker because the upper levels of the index may be stored entirely in memory.

- However, index maintenance time increases with each level.

Multi-level Index on an Ordered File



Multi-level Index Performance Question

Calculate the performance of a multi-level index on an **ordered** file with the following parameters:

- Each disk block stores 2000 data bytes.
- Each index entry occupies 8 bytes.
- Each record has size 100 bytes.
- The data file contains 10,000,000 records.
- There are 3 levels of multi-level index.
  - First level is a sparse index - one entry per block.

How long does it take to retrieve a record based on its key?

Compare this to a single level sparse index.

Indexes with Duplicate Search Keys

What happens if the search key for our index is not unique?

- The data file contains many records with the same search key. This is possible because we may index a field that is not a primary key of the relation.

Both sparse and dense indexes still apply:

- 1) Dense index with entry for every record
- 2) Sparse index containing one entry per block

Note: Search strategy changes if have many records with same search key. May need to search adjacent blocks.

Secondary Indexes

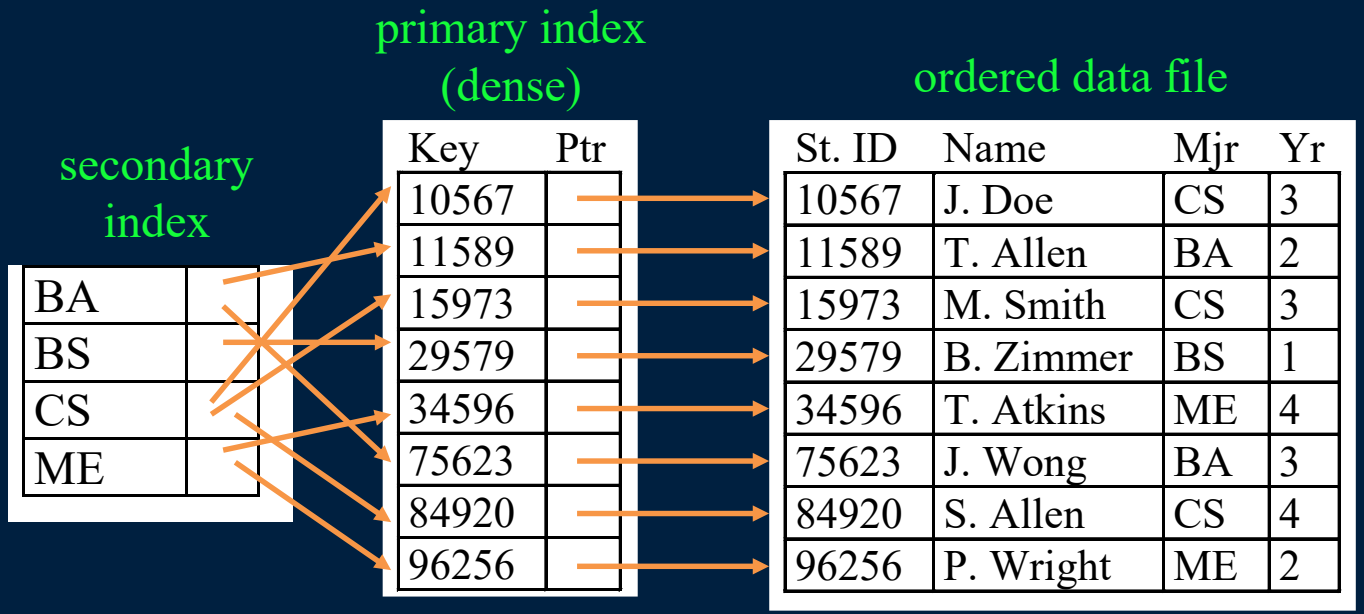
A ***secondary index*** is an index whose search key does not determine the ordering of the data file.

- A data file can have only one primary index but many secondary indexes.

Secondary index entries often refer to the primary index instead of the data records directly.

- Advantage - simpler maintenance of secondary index.
  - Secondary index changes only when primary index changes not when the data file changes.
- Disadvantage - less efficient due to indirection.
  - Multiple levels of indirection as must use secondary index, then go to primary index, then access record in data file.

Secondary Index Example



Secondary Indexes
Handling Duplicate Search Keys

A secondary index may have duplicate search keys.

Techniques for handling duplicates:

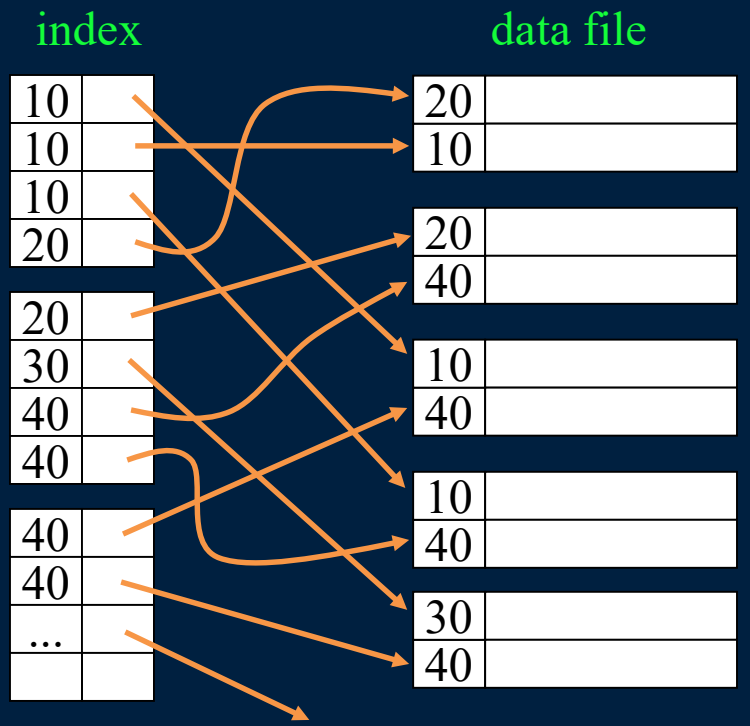
- 1) Create an index entry for each record (dense)
  - Wastes space as key value repeated for each record
- 2) Use buckets (blocks) to store records with same key
  - The index entry points to the first record in the bucket.
  - All other matching records are retrieved from the bucket.

Handling Duplicates
Secondary Index - One Entry per Record

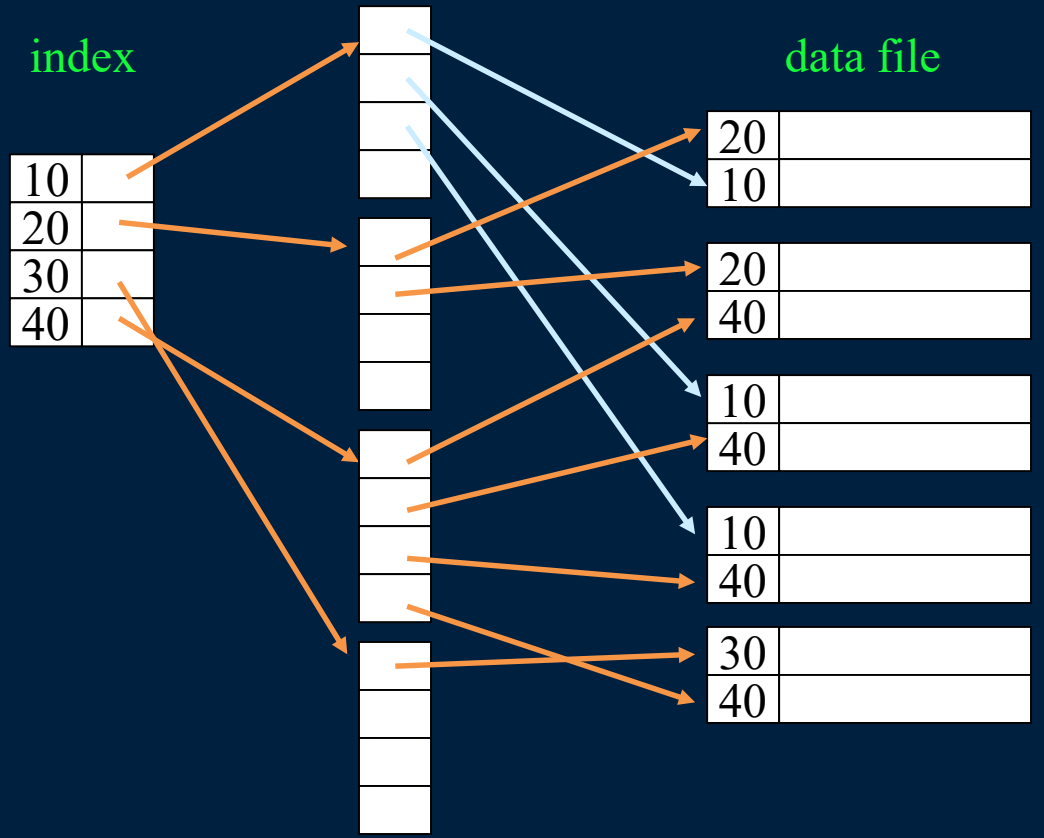
Problem:

Excess overhead!

- disk space
- search time



Handling Duplicates
Secondary Index - Buckets (as blocks)



Secondary Indexes Discussion

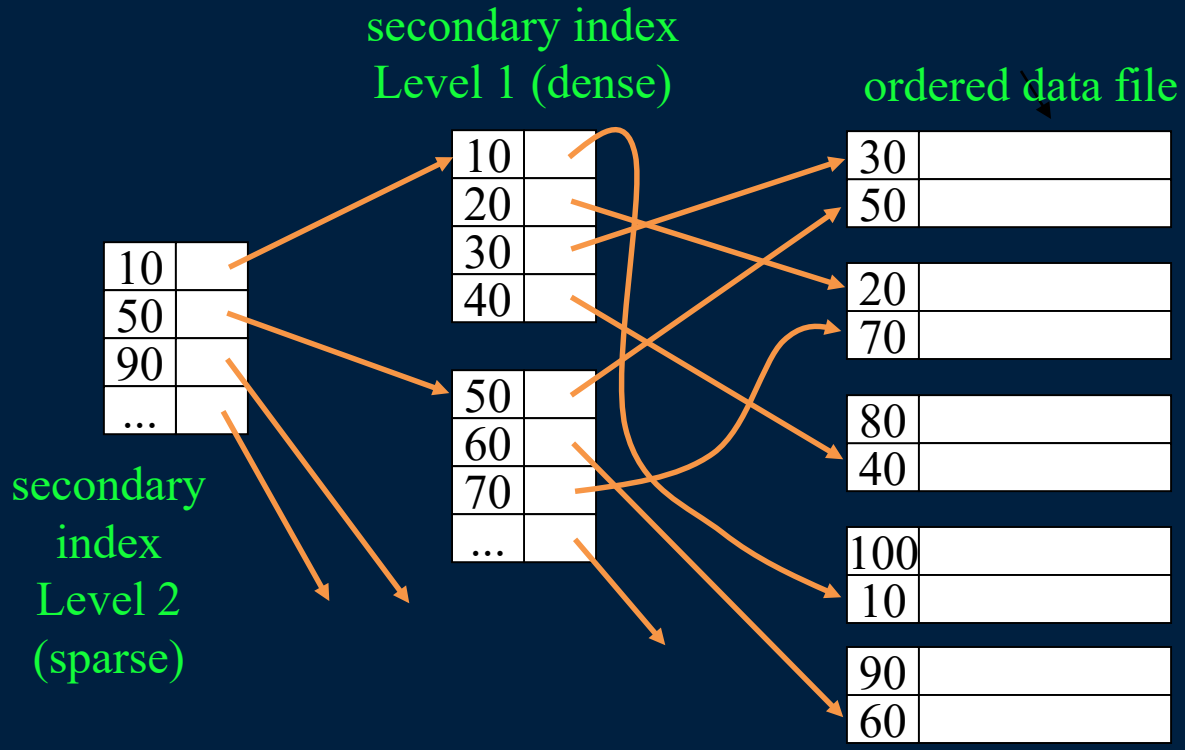
It is not possible to have a sparse secondary index. There must be an entry in the secondary index for **EACH KEY VALUE**.

- However, it is possible to have a multi-level secondary index with upper levels sparse and the lowest level dense.

Secondary indexes are especially useful for indexing foreign key attributes.

The bucket method for handling duplicates is preferred as the index size is smaller.

Multi-level Secondary Index



Secondary Indexes
Buckets in Query Processing

Consider the query:

```
select * from student
where Major = "CS" and Year = "3"
```

If there were secondary indexes on both `Major` and `Year`, then we could retrieve the buckets for `Major="CS"` and `Year="3"` and compare the records that are in both.

- We then retrieve only the records that are in both buckets.

Question: How would answering the query change if:

- a) There were no secondary indexes?
- b) There was only one secondary index?

Secondary Index Example

We will calculate the increased performance of a secondary index on a data file with the following parameters:

- Each disk block stores 4000 bytes.
- Each index entry occupies 20 bytes.
  - 10 bytes for search key, 10 bytes for record pointer
  - Assume 200 index records fit in a disk block.
  - **Assume one index entry per record.**
- Each record has size 1000 bytes.
  - Assume 4 data records fit in a disk block.
- The data file contains 1,000,000 records.

How long does it take to retrieve a record based on its key?

How much faster is this compared to having no index?

Secondary Index Example (2)

Answer:

#indexBlocks = 1,000,000 records / 200 entries/block = 5,000 blocks

#dataBlocks = 1,000,000 records / 4 records/block = 250,000 blocks

Search index using a binary search

= $\log_2 N = \log_2(5000) = 12.28$ blocks

# of blocks retrieved

= 13 blocks + 1 primary index block + 1 data block = **15 blocks**

Time to find record using linear scan (unordered file) = N/2

= 250,000 /2 = **125,000 blocks** retrieved on average

Note that need to do full table scan (250,000 blocks) ALWAYS if want to find all records with a given key value (not just one).

Lesson: Secondary indexes allow significant speed-up because the alternative is a linear search of the data file!

Secondary Index

Question: A secondary index is constructed that refers to the primary index to locate its records. What is the minimum number of blocks that must be processed to retrieve a record using the secondary index?

A) 0

B) 1

C) 2

D) 3

E) 4

Index Maintenance

As the data file changes, the index must be updated as well.

The two operations are ***insert*** and ***delete***.

Maintenance of an index is similar to maintenance of an ordered file. The only difference is the index file is smaller.

Techniques for managing the data file include:

- 1) Using overflow blocks
- 2) Re-organizing blocks by shifting records
- 3) Adding or deleting new blocks in the file

These same techniques may be applied to both sparse and dense indexes.

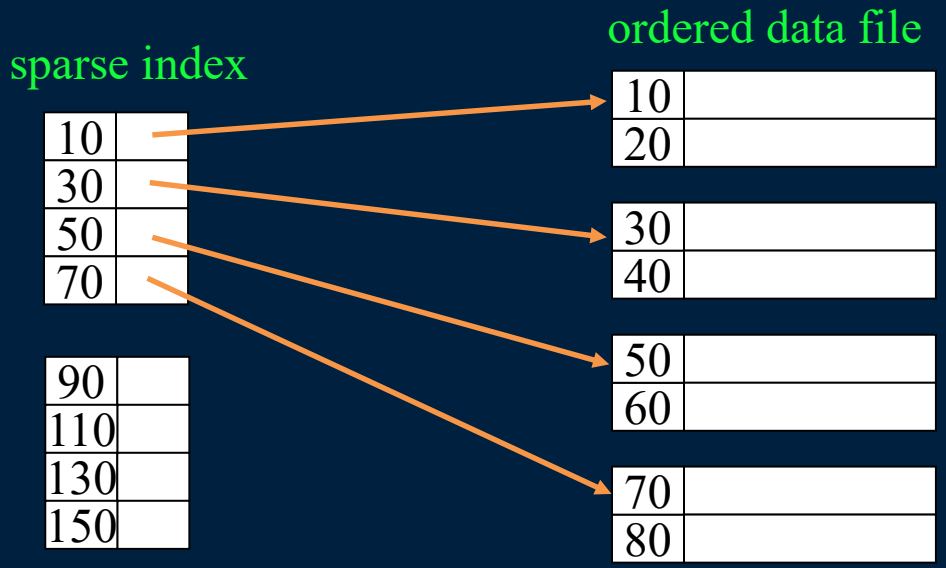
Index Maintenance Summary

In the process of handling inserts and deletes in the data file, any of the previous 3 techniques may be used on the data file.

The effect of these techniques on the index file are as follows:

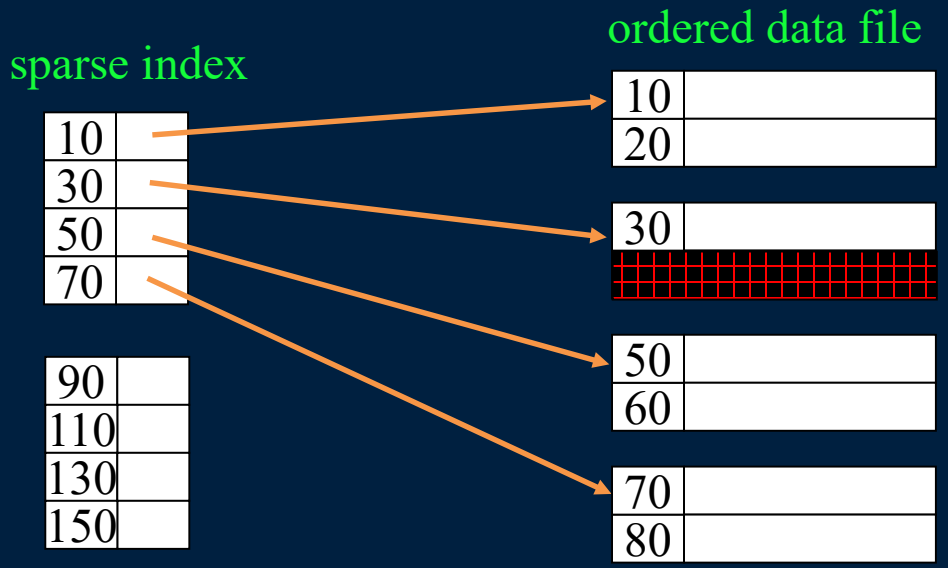
- Create/delete overflow block for data file
  - No effect on both sparse/dense index (overflow block not indexed).
- Create/delete new sequential block for data file
  - Dense index unaffected, sparse index needs new index entry for block.
- Insert/Delete/Move record
  - Dense index must either insert/delete/update entry.
  - Sparse index may only have to update entry if the smallest key value in the block is changed by the operation.

Index Maintenance
Record Deletion with a Sparse Index



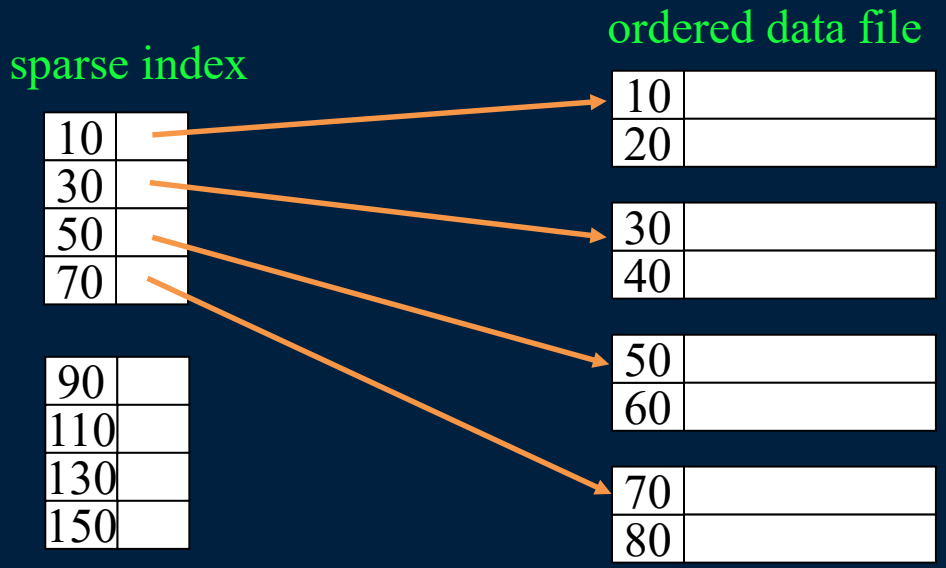
Delete record with key
40 from data file.

Index Maintenance
Record Deletion with a Sparse Index (2)



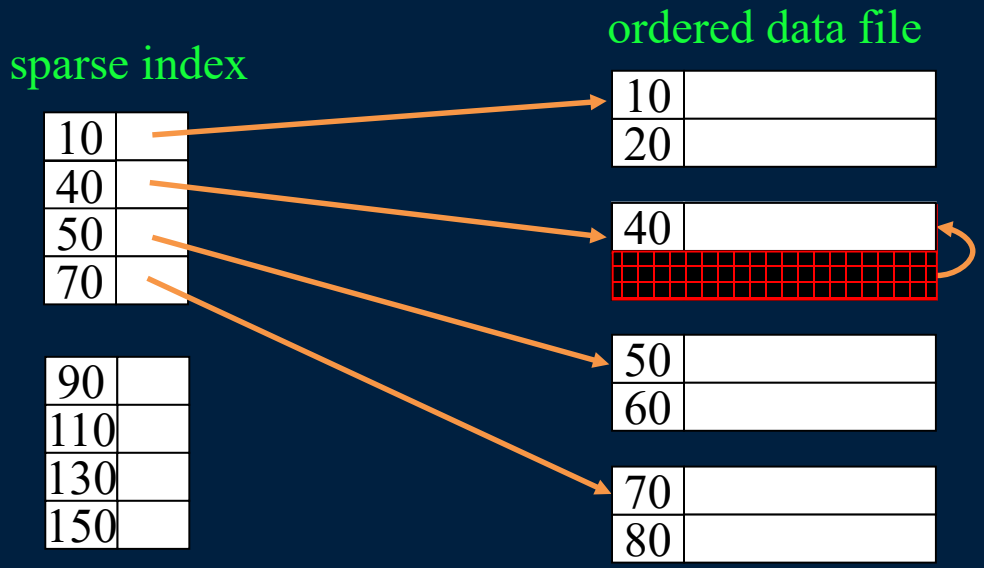
Record deleted.
No change to index.

Index Maintenance
Record Deletion with a Sparse Index (3)



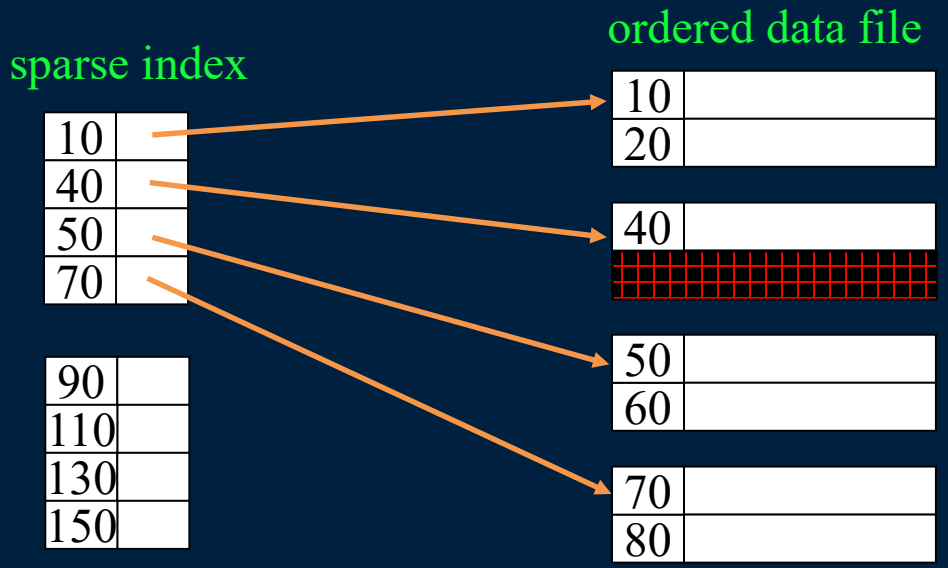
Delete record with key
30 from data file.

Index Maintenance
Record Deletion with a Sparse Index (4)



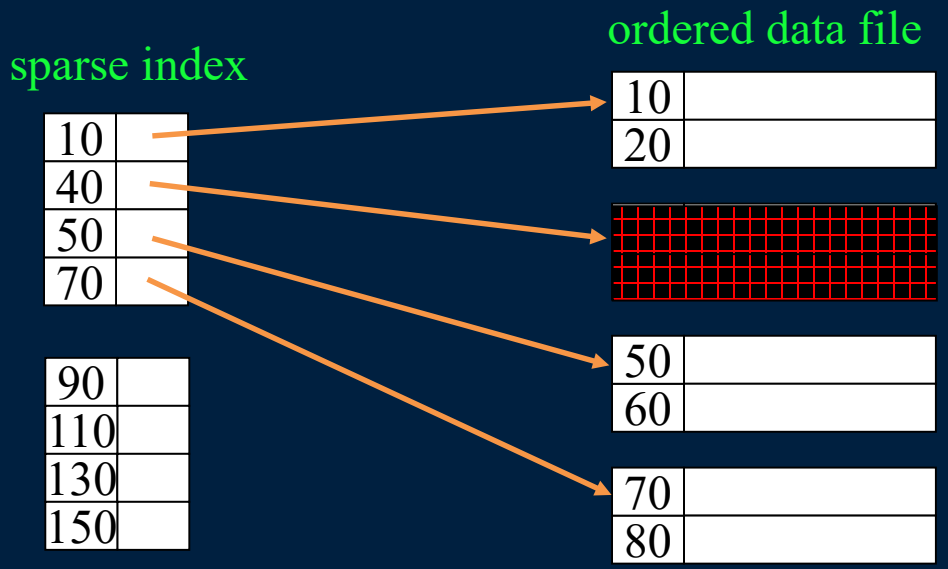
Record 30 deleted.
Shift record up in data block.
Update index entry to 40.

Index Maintenance
Record Deletion with a Sparse Index (5)



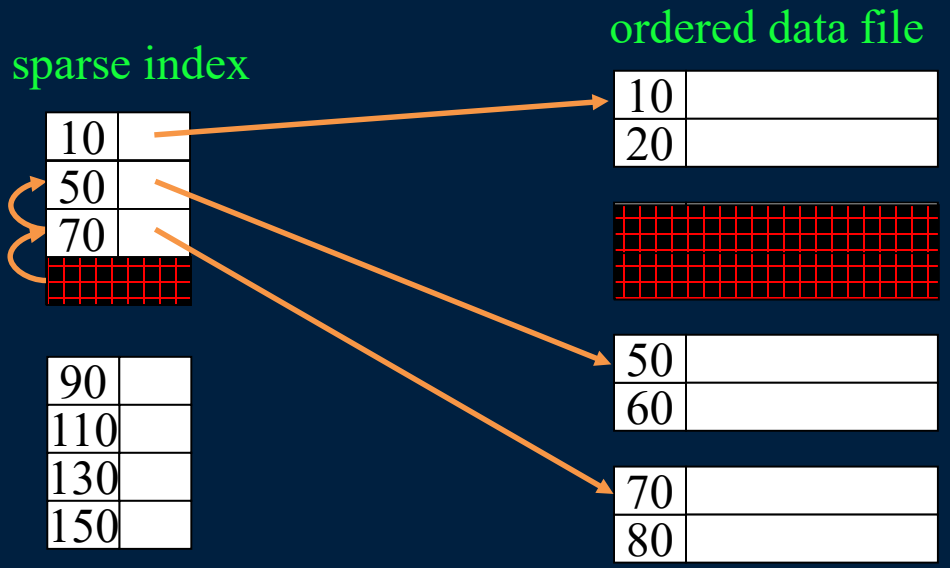
Delete record with key 40.

Index Maintenance
Record Deletion with a Sparse Index (6)



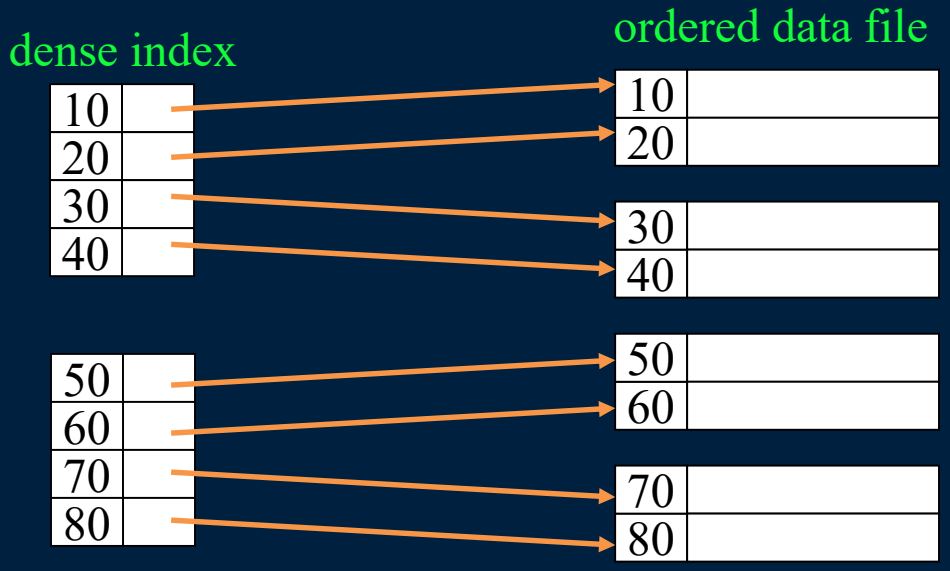
Delete record. Delete block.

Index Maintenance
Record Deletion with a Sparse Index (7)



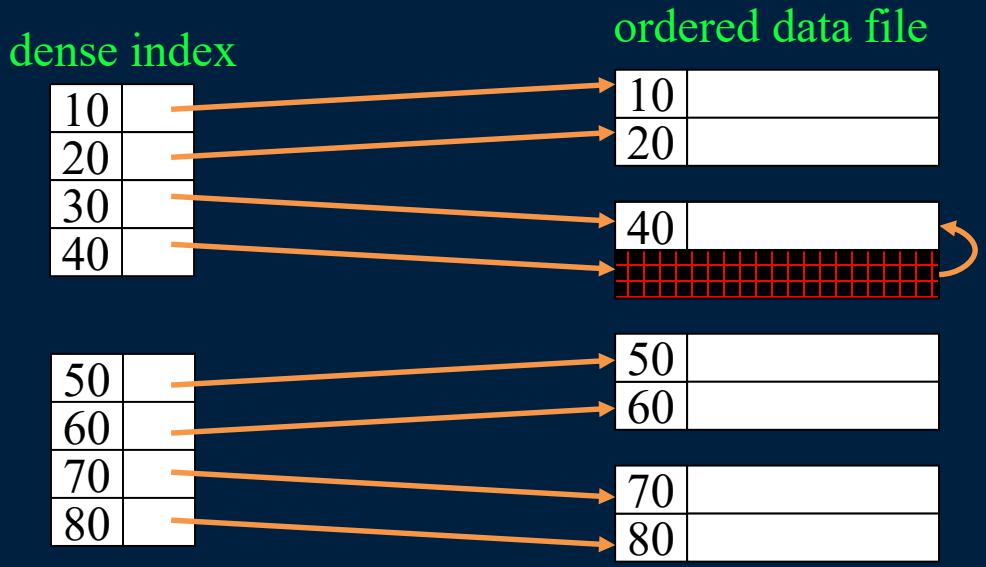
Delete index entry.
Shift index entries in block up.

Index Maintenance
Record Deletion with a Dense Index



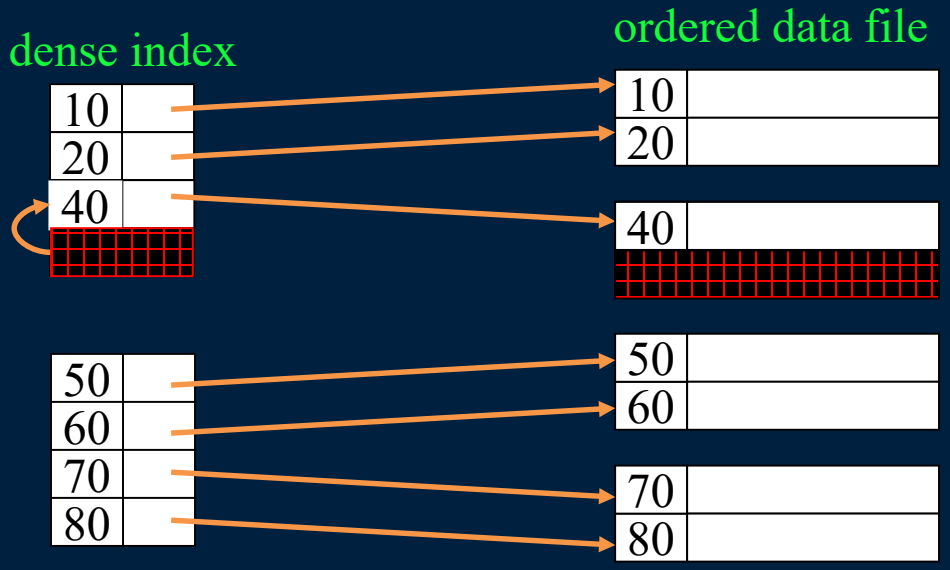
Delete record with key 30.

Index Maintenance
Record Deletion with a Dense Index (2)



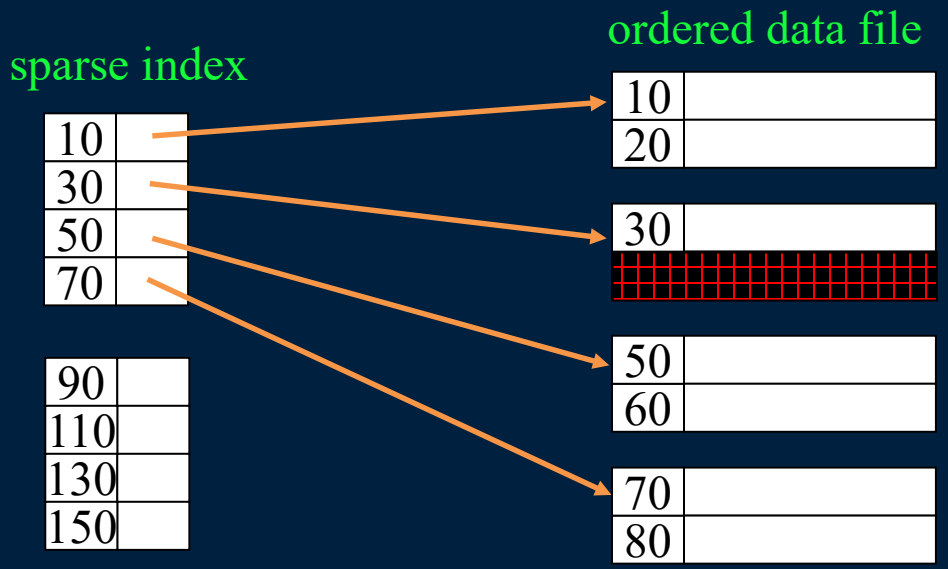
Delete record. Shift 40 up.

Index Maintenance
Record Deletion with a Dense Index (3)



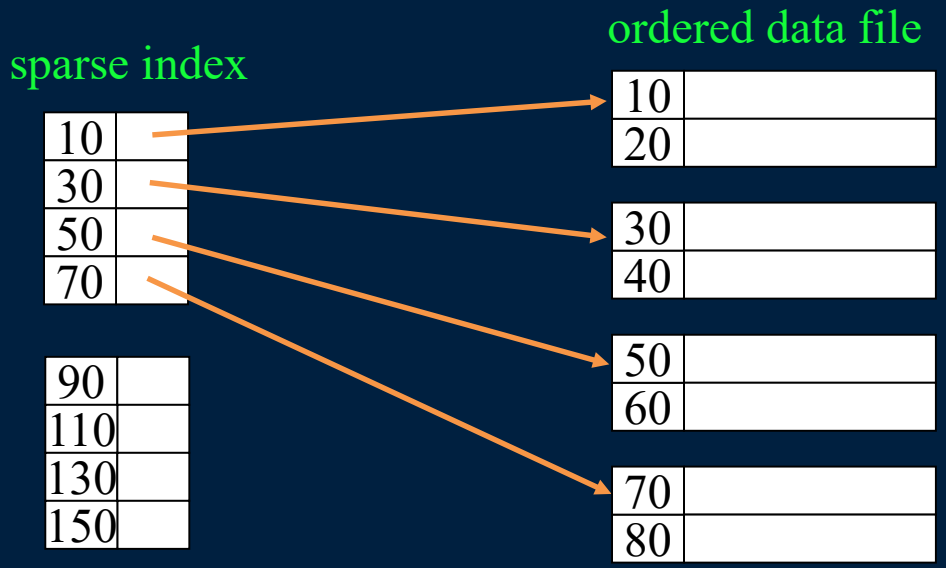
Delete index entry.
Shift index entry for 40 up.

Index Maintenance
Record Insertion with a Sparse Index



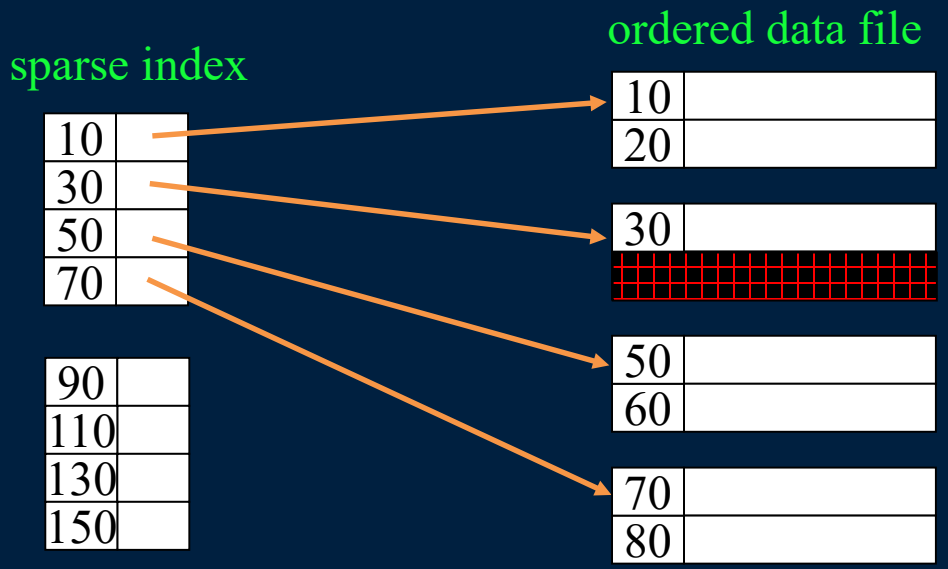
Insert record with key 40.

Index Maintenance
Record Insertion with a Sparse Index (2)



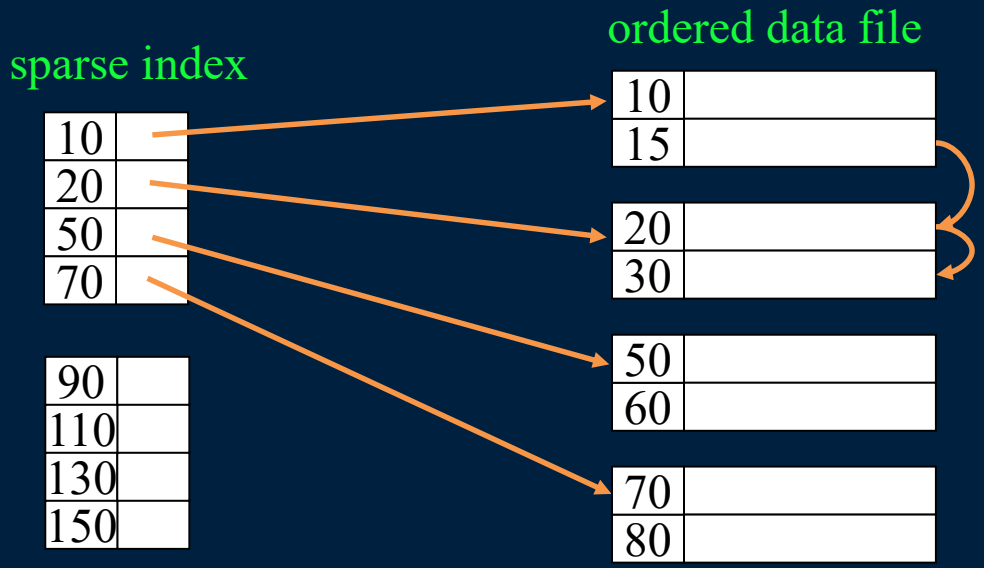
Record inserted in free space in second block. No updates to index.

Index Maintenance
Record Insertion with a Sparse Index (3)



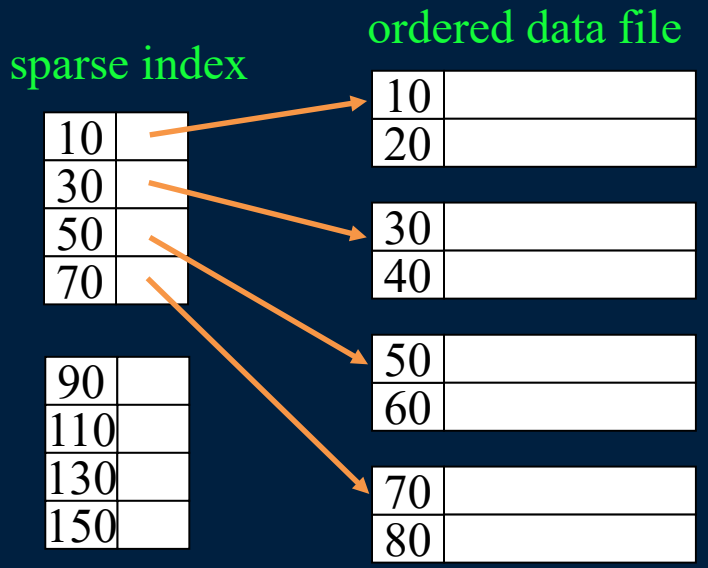
Insert record with key 15.
Use immediate re-organization.

Index Maintenance
Record Insertion with a Sparse Index (4)



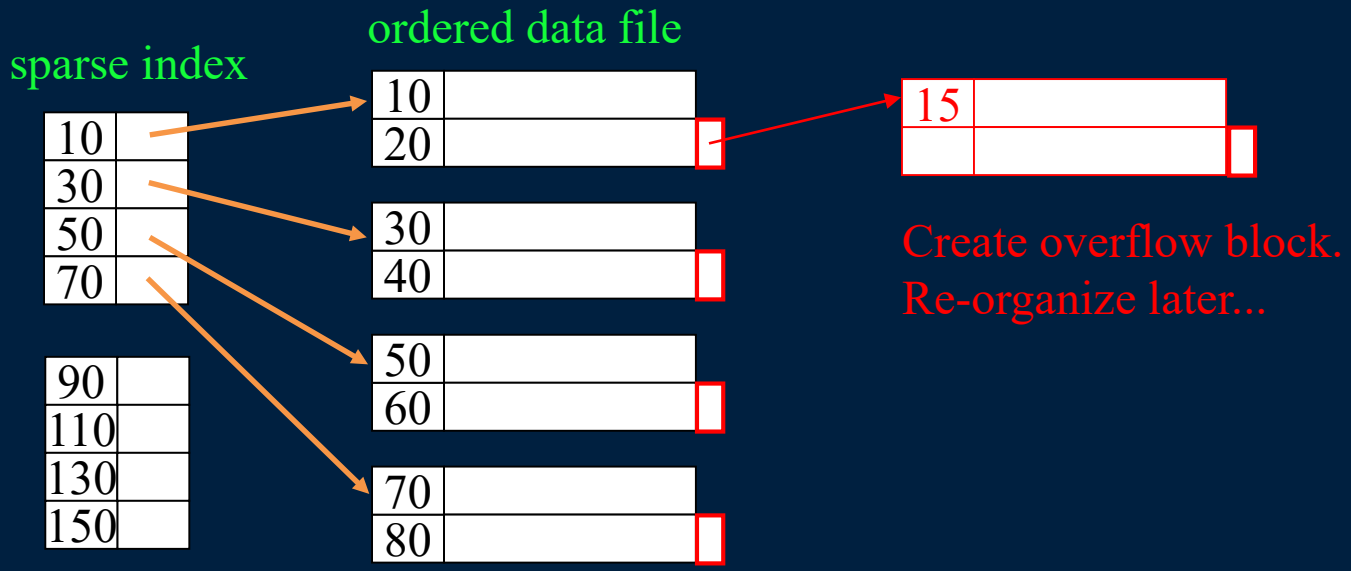
Shift records down to make room for 15.
Update index pointer for block 2.

Index Maintenance
Record Insertion with a Sparse Index (5)



Insert record with key 15.
Use overflow blocks.

Index Maintenance
Record Insertion with a Sparse Index (6)



Handling Data Evolution

Since it is common for both the data file and index file to evolve as the database is used, often blocks storing data records and index records are not filled completely.

By leaving a block 75% full when it is first created, then data evolution can occur without having to create overflow blocks or move records around.

The tradeoff is that with completely filled blocks the file occupies less space and is faster to process.

Conclusion

Indexes are lookup mechanisms to speed access to particular records in the data file.

- An ***index*** consists of an ordered sequence of index entries containing a search key and a pointer.
  - An index may be either ***dense*** (have one entry per record) or ***sparse*** (have one entry per block).
  - ***Primary*** indexes have the index search key as the same key that is used to physically order the file. ***Secondary*** indexes do not have an affect on the data file ordering.

An index is an ordered data file when inserting/deleting entries.

- When the data file is updated the index may be updated.

Major Objectives

The "One Things":

- Explain the types of indexes: ordered/unordered, sparse/dense, primary/secondary, single/multi-level
- Perform calculations on how fast it takes to retrieve one record or answer a query given a certain data file and index type.

Major Theme:

- Indexing results in a dramatic increase in the performance of many database queries by minimizing the number of blocks accessed. However, indexes must be maintained, so they should not be used indiscriminately.

Objectives

- Define: index file, data file, search key, index entry
- List the index evaluation metrics/criteria.
- Explain the difference between the difference types of indexes: ordered/unordered, dense/sparse, primary/secondary, single/multi level and be able to perform calculations.
- List the techniques for indexing with duplicate search keys.
- Discuss some of the issues in index maintenance.
- Compare/contrast single versus multi-level indexes.
- Explain the benefit of secondary indexes on query performance and be able to perform calculations.


THE UNIVERSITY OF BRITISH COLUMBIA