Organización física de archivos e índices

CI-0127 Bases de Datos, Universidad de Costa Rica

Sivana Hamer

Importante: Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

1 Physical storage

Computers can store data in different mediums:

- **Primary storage:** Data stored in main memory, thus the data is stored only while the computer is on (*volatile storage*). This includes cache and RAM. This memory is faster, but is more expensive and has less storage capacity than secondary storage.
- **Secondary storage:** Data stored persistently in the database (*nonvolatile storage*). This includes flash memory (SSD) and magnetic disks (HDD). This storage is slower than primary storage, however, it is cheaper and has more storage capacity.
- **Tertiary storage:** Data stored in removable media that is offline from the main system. This includes DVDs, flash drives, and magnetic tapes.

Databases store *data persistently* as there are large amounts that must be stored for large amounts of time. Thus, they are usually stored in nonvolatile storage with magnetic disks as the main medium of storage for data files. Thus, DBMS tends to have a disk-oriented structure using the primary memory as storage. While the data is being used, it has to be loaded into volatile storage. Databases are also frequently backed up with magnetic tapes.

2 Physical database design

DBMS manages the structure in which the data is organized with the *physical database design*. All the information about the database is saved in disk blocks through the operative system (OS). But, the DBMS is in charge of managing the blocks to have more control over efficient access and support recovery.

The general physical database design of a *disk-oriented* is shown in Fig. 1. When a query is executed by the *execution engine* that asks for data that is stored in the database. This data is stored into files that are organized into pages. Therefore, the *storage manager* finds the files, pages and records related to the query from memory and disk. To get the data, first the relevant page directory is loaded from disk to memory. Based on the information of the page directory, the relevant page is found and loaded. Finally, the records are found within the page using the header and are given to the query as a result.

In the following sections, the building blocks of the physical memory are detailed: files, pages and records. Their relationship is shown in Fig. 2.

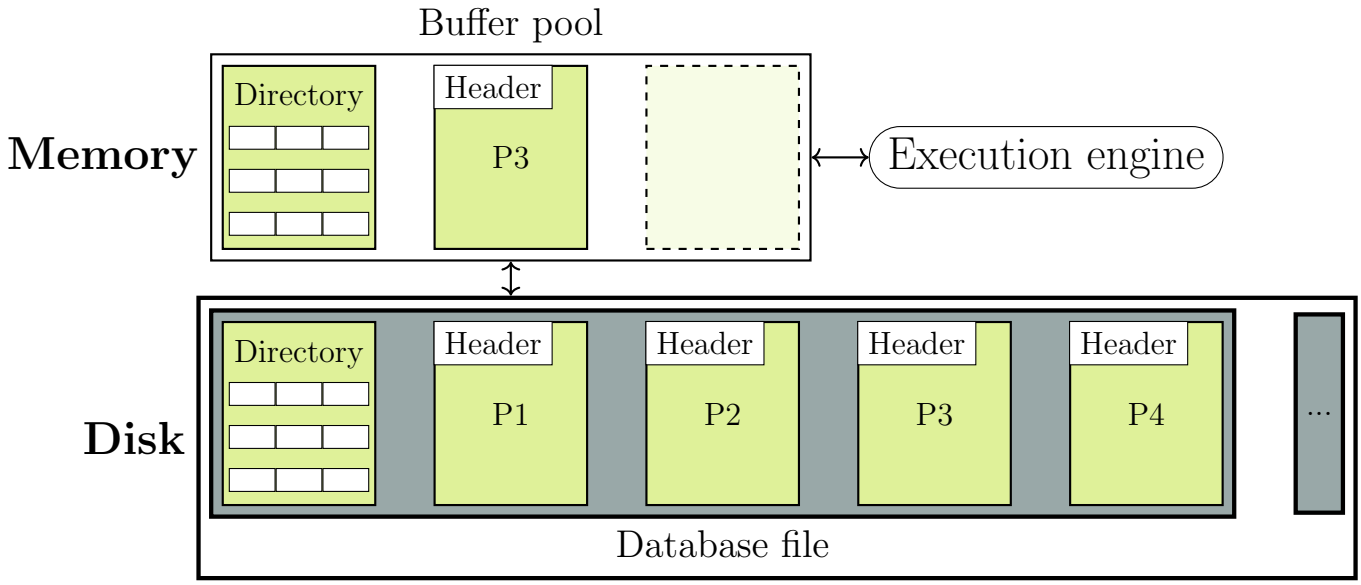


Figure 1: Disk-oriented physical database design

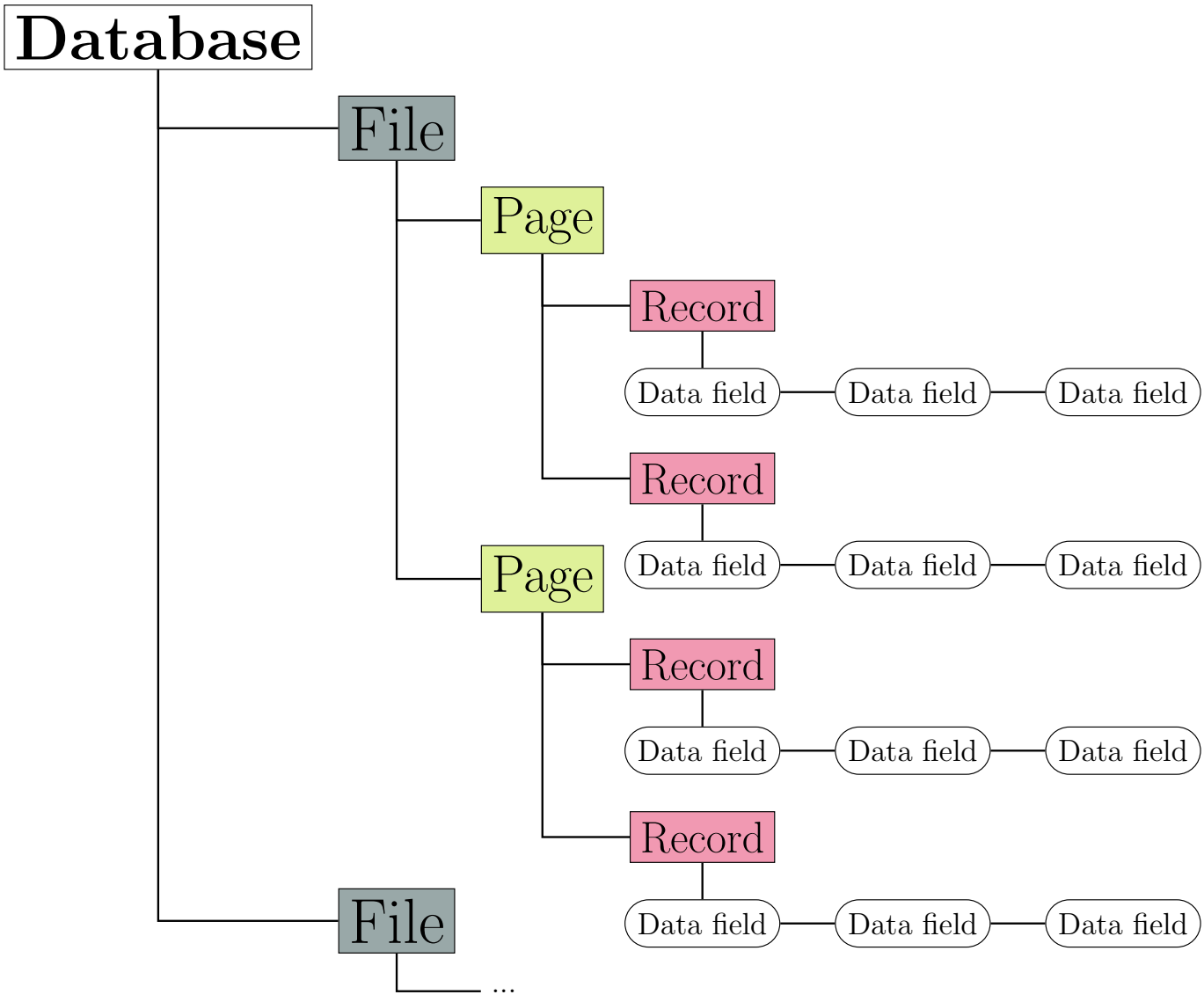


Figure 2: Relationship between databases, files, pages and records

3 Files

Databases store data in files. These files are saved on disk using the OS, however, only the DBMS knows how to decipher these files. Some DBMS store files in a hierarchy, while others store everything in one file.

4 Pages

Each file is logically partitioned into fixed-length storage units used for data allocation and transfer called *pages*. A page is sometimes called a *block*. Pages may contain several records. Every page has a unique identifier.

There are several ways a DBMS can locate a page within a file including database heaps, sorted files, and hashes. DBMS commonly save unordered pages that are tracked using a *page directory* (Fig. 3). This directory saves the position for every page, retrieving the specific pages with the identifier. This page directory is the first page within the file, thus it is easily found.

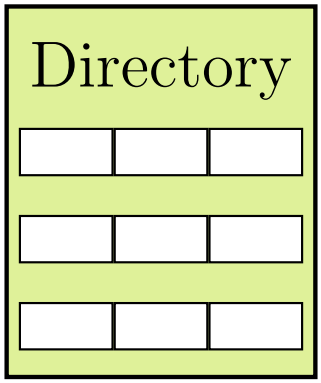


Figure 3: Page directory

To store a variable length of records on a page, we can use a *slotted-page structure* or log-based structure. DBMS commonly uses the slotted-based structure, hence we shall further describe it. The slotted-page structure is shown in Fig. 4. There is a header at the begging of each page containing the number of records within the page, offset to the last used record, and an array that details the size and location of each record (slot array). Records are stored beside each other in the block, starting from the end. Data is inserted at the end of the free space. Deleted records may be reused or ignored, depending on the DBMS.

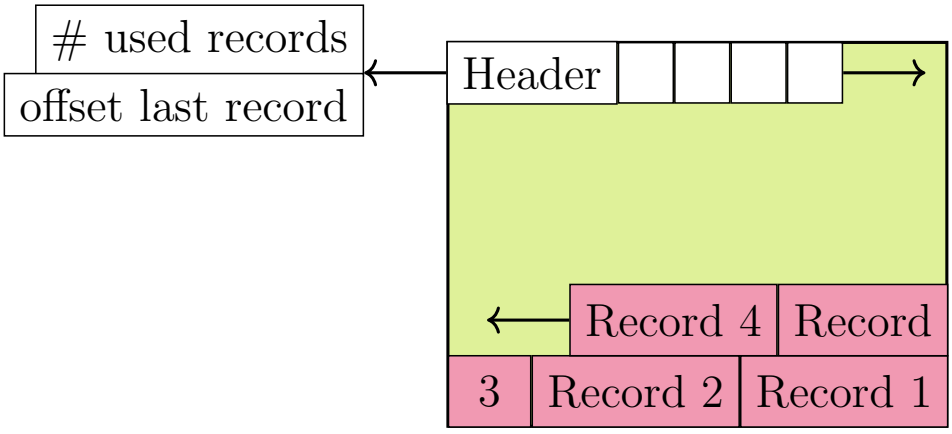


Figure 4: Slotted-page structure for a page

5 Records

Records are a collection of data items for different fields. They are what we know as tuples or rows in the relational model and SQL, respectively. Each record is uniquely identifiable with an identifier (primary key of the table).

Usually, in a database records of different relations have different sizes. Thus, there are several approaches used to manage these differences:

- **Fixed length.** We assign a number of bytes n for each record.
- **Variable length.** To represent every record with variable length (Fig. 5). Each record will have a fixed-length header at the begining of the record with details the offeset (os_x) to the data item and the size of the data item (l_x). Therefore, every data item has the tuple $(offset_x, length_x)$ Followed by this header, the information of the data items are stored (df_x).

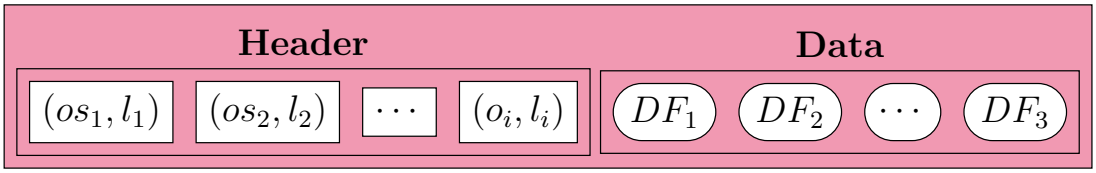
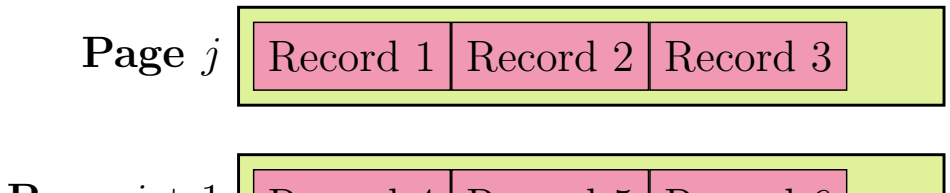


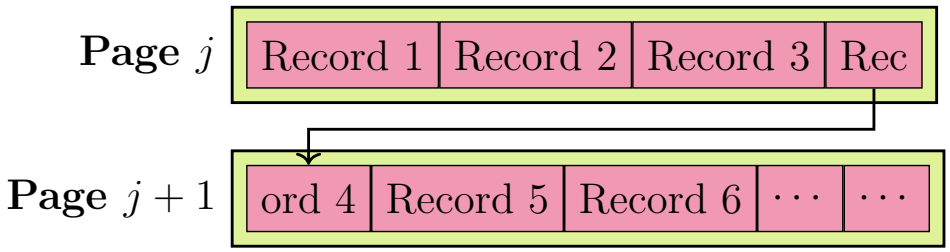
Figure 5: Variable length record structure

As we can have more records in a relation than space available in a record, we can store records in several pages. DBMS tend to save only records for the same relation in a table. There are two ways we can save records across multiple pages (Fig. 6):

- **Unspanned.** If each record is not allowed to be saved in multiple pages.
- **Spanned.** If the record can be divided within several pages. This uses all the space within the page. Saves more space but everytime we need a record that is saved in multiple pages, we will need to load several pages.



(a) Unspanned



(b) Spanned

Figure 6: Saving records across multiple pages

We can calculate the number of records that fit in a page with the blocking factor (bfr).

$$bfr = \lfloor \frac{B}{R} \rfloor$$

- B is the number of bytes for a page. This is fixed length (i.e., all pages have the same size).
- R is the number of bytes for a record.
- bfr is the blocking factor. This details the number of records that fit within a page.

We can also calculate the number of pages needed to save a number of records.

$$b = \lceil \frac{r}{bfr} \rceil$$

- r is the number of records to save.
- bfr is the blocking factor. bfr for variable-length records represents the average number of records per page.
- b is the number of pages needed to save all the records.

With regards to searching:

- If the records are ordered by attributes and we are searching for them, we can need to retrieve $O(logb)$ pages (binary search).
- If not, we must iterate over every page, in worst case, to find the record. On average, we will have to iterate over $O(n/2)$ pages.

We can see the file, page and slot (i.e., record) information in SQL Server with the following command for a table T.

```
SELECT sys.fn_PhysLocFormatter(%%physloc%%) AS [File:Page:Slot], *
FROM T;
```

6 Indexes

Though files and pages are stored in a certain order physically, databases use an auxiliary data structure called an *index* to provide a faster way to find the *secondary access paths* of the data within a database without altering the physical structure. Based on an *indexing field* or *indexing attribute*, the index is created to enable fast access on those fields. The DBMS ensures that the tables and indexes are synchronized. There is a trade-off between speed and additional resources required (additional space and more maintenance effort for synchronization). There are two levels of indexes: ordered indexes and multi-level indexes.

7 Ordered indexes

Ordered indexes are auxiliary hash tables that use the sorted order of the records and a search key to find the associated records. This is similar to a library, where books are sorted in some order (e.g., genre, author name, DOI) and we have a key with which we can search for the books based on the sorting. Thus, every record has an *index record* that saves both the search key value and the pointers to the records.

There are several types of ordered indexes described in the following subsections. The relationship between these is shown in Table 1.

Depending on the type index field, we may have an entry for every search key value (*dense index*) or entries for some search key values (*sparce or non-dense index*). Primary and clustering indexes are nondense. Secondary indexes are dense for keys, but may be dense or nondense depending on the sorting order of the records for non-key attributes.

	Physically ordered by indexing key	**Physically not ordered by indexing key**
Index field is key	Primary index	Secondary index (key)
Index field is non-key	Clustering index	Secondary index(non-key)

Table 1: Different types of indexes based on the storage order and indexing field

Primary index

Records *are sorted with the index record*. The index record uses as a search key value the *key attribute* of a relation. There is a search key for the *first* record within a page (the *anchor record*). Thus, there is one key per number of pages in the ordered data files. The structure is shown in Fig. 7.

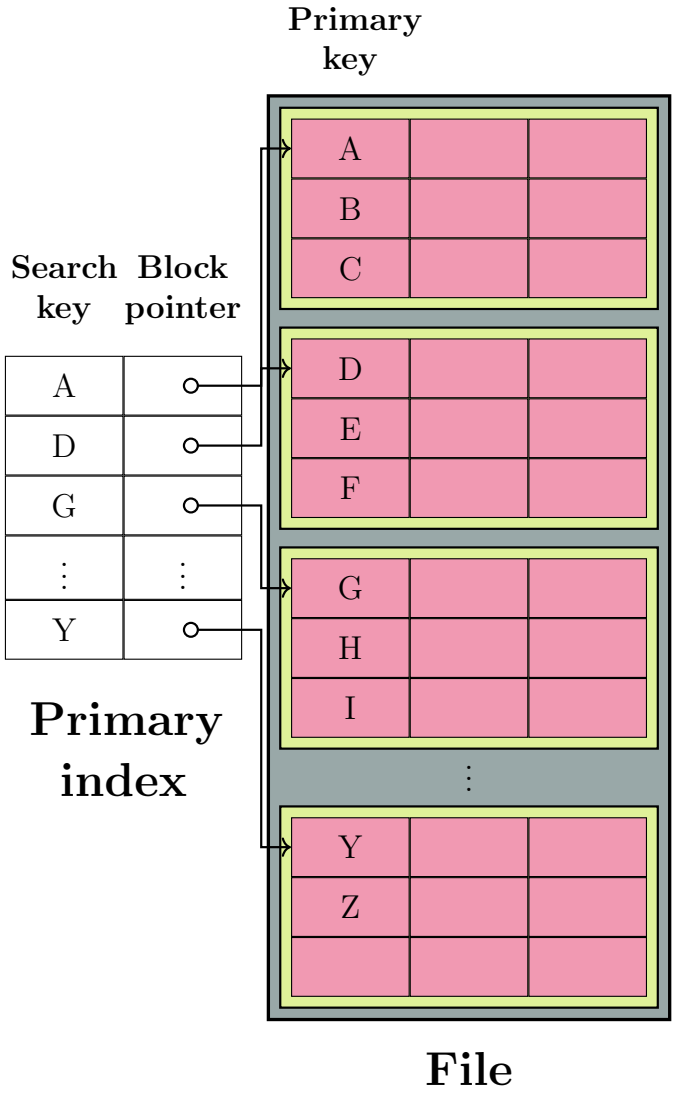


Figure 7: Primary index

Clustering index

Records *are sorted with the index record*. The index record uses as a search key value a *non-key attribute* of a relation. As there can be repeated values, the index saves an entry for *each distinct* value for the index record, pointing at the page where the first record is found. The structure is shown in Fig. 8.

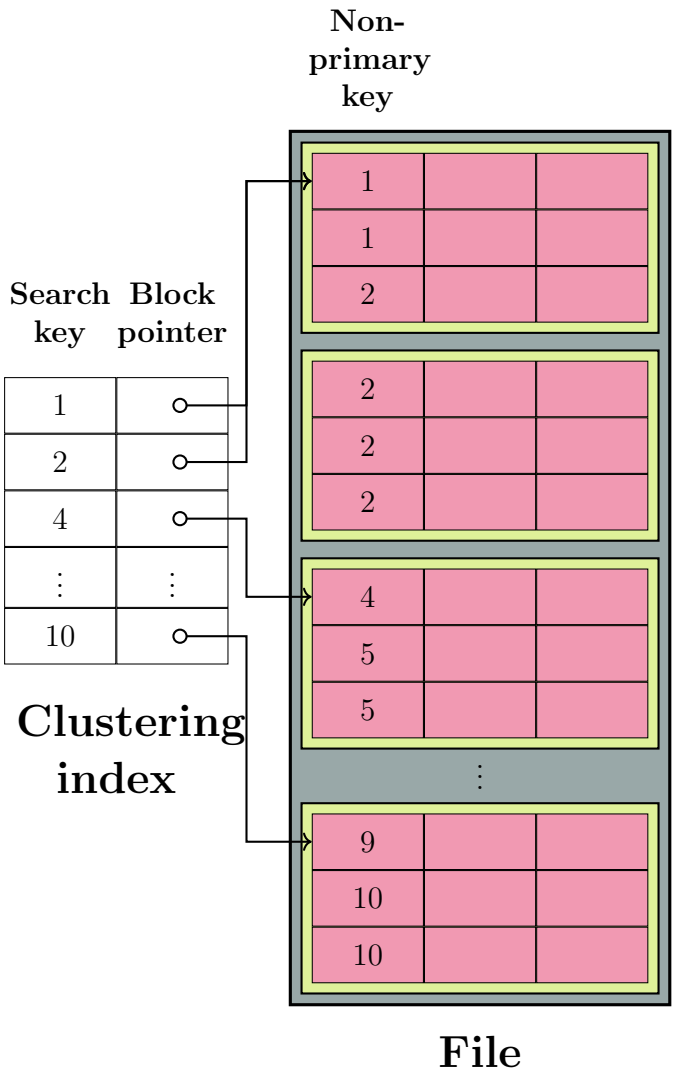


Figure 8: Clustering index

Secondary index

Records *are not sorted with the index record*. Either we can use as the search key value the key or non-key attributes of a relation. Therefore, there needs to be an index record for every value. The structure is shown in Fig. 9.

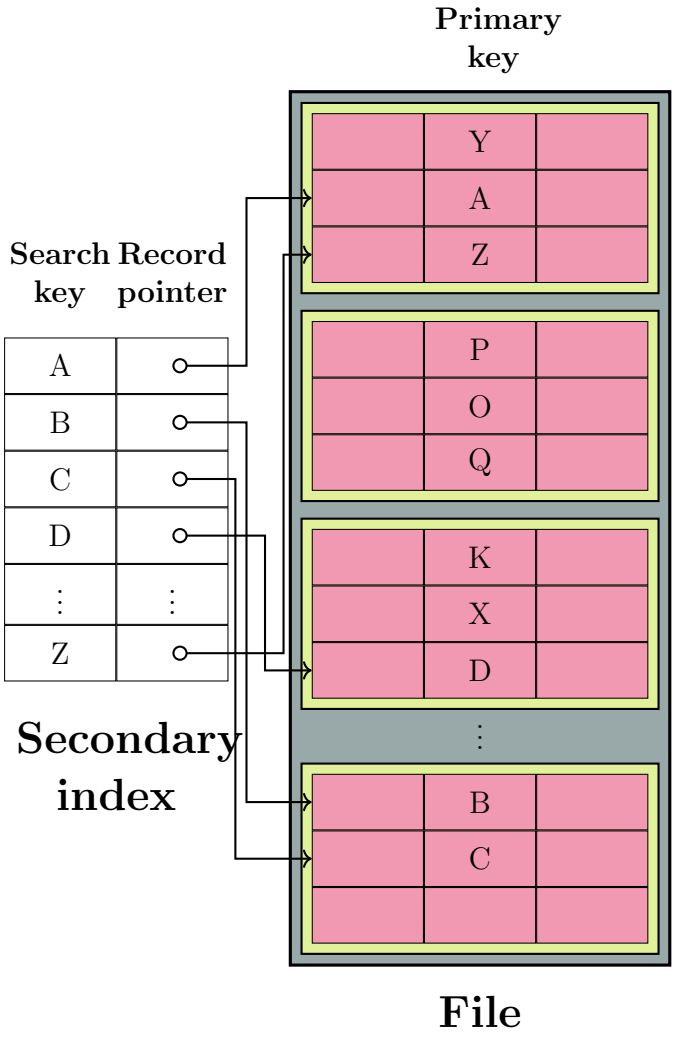


Figure 9: Secondary index for a primary key

8 Multi-level indexes

Ordered indexes disadvantage is that the performance of index lookups and sequential scans does not scale well when the file grows. Furthermore, frequent reorganization of records within the file are needed but are undesirable.

Thus, *balanced tree structures* are used to maintain efficiency in insertions and deletions. In a balanced tree of n children, every non-leaf node has between $\lceil \frac{n}{2} \rceil$ and n children. Therefore the distance from the root of the tree to any leaf is the same. Specifically, *B+ Trees* are the most used auxiliary data structures for databases. However, this data structure, compared to ordered indexes, has an additional overhead for insertions, deletions and space.

9 B+ Tree

A *B+Tree* is a self-balancing tree that allows insertions and deletions in $log_n(b)$ time. We can see an example in Fig. 10. This is a tree with $n = 2$ keys. Thus every node has $n + 1 = 3$ references.

Let us define that every node of the tree is a combination of references (rf) and values (v). In our example, it has the form $(rf_1, v_1, rf_2, v_2, rf_3)$.

There is only one inner node for this tree (in this example, the node with keys 5 and 9). Inner nodes are non-leaf nodes. Meanwhile, all the nodes in the lowest level are leaf nodes. There is a leaf node for every value of the search field with a pointer to either the page or record where the value of the node is stored. In this example, we are pointing to every record. The last pointer in a leaf node saves the direction where the following leaf node of the tree is.

The inner nodes have the following properties.

- Any value in the subtree in which rf_1 points to is denominated X_1. Thus, $X_1 \leq V_1$.
- Any value in the subtree in which rf_2 points to is denominated X_2. Thus, $V_1 < X_2 \leq V_2$.
- Any value in the subtree in which rf_2 points to is denominated X_3. Thus, $V_2 < X_3$.

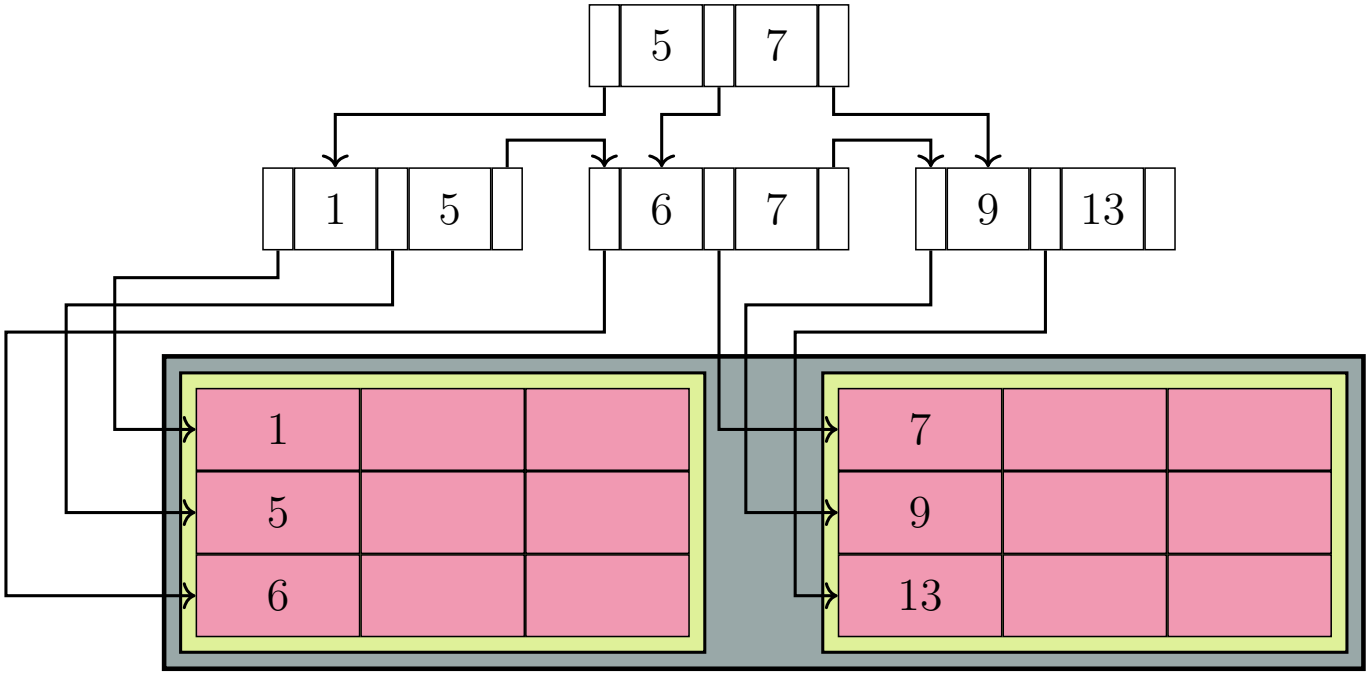


Figure 10: B+Tree example

Searching

We can take advantage of the structure of the tree to iterate through from the leaf to the root to find specific values. We can see the steps in Algorithm 1.

Inserting

We can also insert new values into the three. Insertions must ensure that the tree is balanced and still complies with all the conditions of the data structure. We can see the high-level steps in Algorithm 2.

Deleting

Deletions have the same restrictions as insertions. We can see the high-level steps in Algorithm 3.

Algorithm 1 Searching for records that has a search key value v of a B+ Tree of order p

```
function FIND(v)
    n = tree.getRootPage();
    n.readPage();
    p = n.getNumberPointers();
    while (!n.isLeafNode()) do
        l = n.getLarger(v);          ▷ The node n with i x_i values, get those that are v ≤ x_i
        xi, pi = l.min(); ▷ Finds the smallest x_i that is larger than v with the respective pointer p_i
        if (xi.isNull()) then                                                    ▷ v > x_i
            n = n.getLastPointer();                          ▷ Gets last non-null pointer of n
        else if (v == xi) then                                                   ▷ v = x_i
            n = n.getNext(pi);                               ▷ Gets the pointer following p_i
        else                                                                     ▷ v < x_i
            n = pi;                                          ▷ Gets the current pointer
        end if
        n.readPage();
    end while
    r = n.hasRecordWithKey(v)
    if (!r.isNull()) then                          ▷ We did found a record with the value
        return n;
    else                                       ▷ We did not found a record with the value
        return null;
    end if
end function
```

Algorithm 2 Inserting a record that has a search key value v of a B+ Tree of order p

```
function INSERT(v)
    l = tree.find(v)                    ▷ We use a variation of the find(v) method
    if (l.hasSpace()) then
        l.insertOrder(v);                              ▷ Also updates the references
    else                                                   ▷ We must split the node
        m = l.getMiddleKey();
        l, l2 = l.splitEvenly();
        l.parent.insertOrder(m, l2);
    end if
end function
```

Algorithm 3 Deleting a record that has a search key value v of a B+ Tree of order p

```
function DELETE(v)
    l = tree.find(v)                    ▷ We use a variation of the find(v) method
    if (l.hasRecordWithKey(v)) then
        l.remove(v)
        if l.sibling.canRedistribute() then   ▷ Siblings are adjacent nodes with the same parent
            l.redistributeSibling();
        else
            n, o = merge(l, l.siblings);      ▷ n is the new merged node, while o is the old node
            n.parent.removeReference(o);
        end if
    end if
end function
```

References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016, chapters 16 and 17.

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY: McGraw-Hill, 2020, chapter 12, 13 and 14.

[3] A. Crotty and L. Ma. Lecture #3, #4, #5,#6 and #7. [Online]. Available: https://15445.courses.cs.cmu.edu/fall2021/schedule.html

[4] Microsoft. Sql server guides. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-guides

[5] ——. Database files and filegroups. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-files-and-filegroups