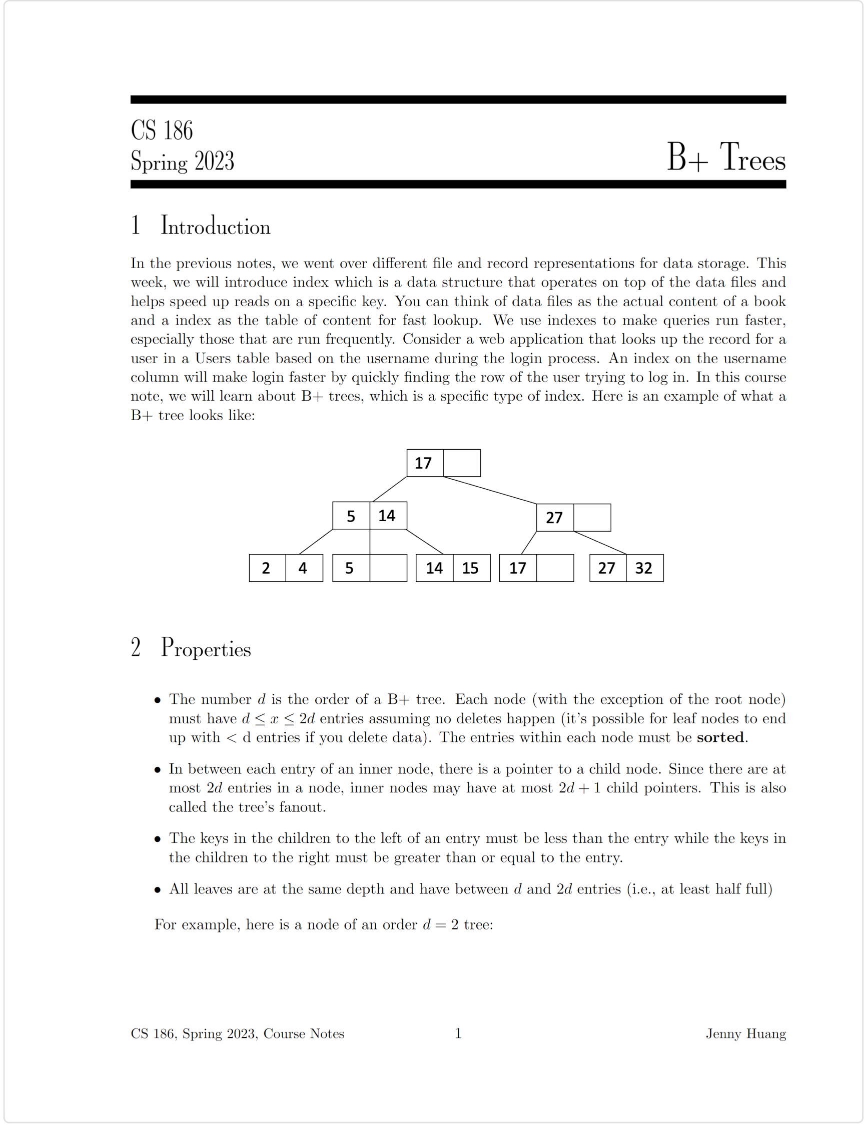
Query Evaluation

B+ Trees

Most common index type in Relational Systems.



CS 186
Spring 2023

B+ Trees

1 Introduction

In the previous notes, we went over different file and record representations for data storage. This week, we will introduce index which is a data structure that operates on top of the data files and helps speed up reads on a specific key. You can think of data files as the actual content of a book and a index as the table of content for fast lookup. We use indexes to make queries run faster, especially those that are run frequently. Consider a web application that looks up the record for a user in a Users table based on the username during the login process. An index on the username column will make login faster by quickly finding the row of the user trying to log in. In this course note, we will learn about B+ trees, which is a specific type of index. Here is an example of what a B+ tree looks like:

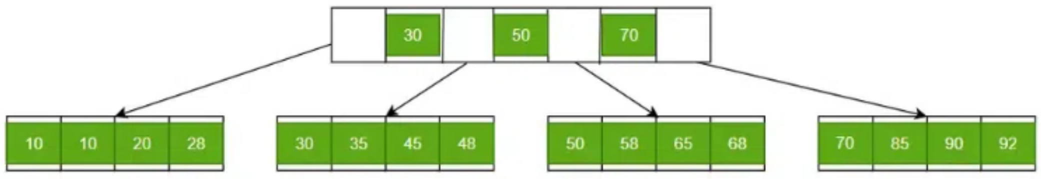
2 Properties

- The number d is the order of a B+ tree. Each node (with the exception of the root node) must have $d \leq x \leq 2d$ entries assuming no deletes happen (it's possible for leaf nodes to end up with < d entries if you delete data). The entries within each node must be **sorted**.
- In between each entry of an inner node, there is a pointer to a child node. Since there are at most $2d$ entries in a node, inner nodes may have at most $2d + 1$ child pointers. This is also called the tree's fanout.
- The keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.
- All leaves are at the same depth and have between d and $2d$ entries (i.e., at least half full)

For example, here is a node of an order $d = 2$ tree:

CS 186, Spring 2023, Course Notes 1 Jenny Huang

- A **B+ Tree** is a self-balancing, **multi-level index data structure** used in databases and file systems.
- It's an enhancement over B Trees with better search efficiency for large datasets.
- All **values/keys are stored at the leaf level**, with **internal nodes only storing keys** to maintain order and support quick traversal.



Structure

Binary Search

1. **Root Node**: Contains pointers to child nodes (can be either leaf or internal nodes).
2. **Internal Nodes**: Store keys for routing; each key in an internal node acts as a separator for child nodes.
3. **Leaf Nodes**: Contain actual values and keys; form a **linked list**, enabling range-based traversal.

Properties

1. **Order (m)**: Determines the maximum number of children per node.

- An internal node with `m` children has up to `m-1` keys.

2. **Balanced Tree**: All leaf nodes are at the same depth.
3. **Sequential Access**: Leaf nodes are linked, making range queries efficient.
4. **Keys-only Internal Nodes**: Internal nodes only store routing keys, while leaf nodes hold all values.

Insertion

1. **Locate Leaf Node** for insertion using routing keys in internal nodes.
2. **Add Key-Value** pair in the correct position in the leaf node.
3. If a node **overflows** (exceeds `m-1` keys):

- Split the node, promote the median key to the parent, and adjust pointers.
- If the parent overflows, recursively split up to the root if necessary.

Deletion

1. **Locate Leaf Node** where the key exists.
2. **Remove Key** and handle underflow:

- If a node underflows (fewer than `[m/2]` keys), **borrow a key** from a sibling or **merge with a sibling**.
- Recursively adjust parent nodes as needed.

Searching

1. Start at the **root** and traverse down using routing keys until reaching the leaf level.
2. If key is present, **retrieve the associated value** in the leaf.

Range Query

1. **Find the starting key** in the leaf level.
2. **Traverse linked leaf nodes** to retrieve a sequence of values within the specified range.

Advantages

- **Efficient Range Queries**: Linked leaf nodes support fast range-based retrieval.
- **Disk-based Optimization**: Minimized disk I/O due to the balanced nature and linked structure.
- **Dynamic Updates**: Insertion and deletion operations maintain balance automatically.

Applications

- **Databases**: B+ Trees are used in database indexing (e.g., MySQL, PostgreSQL).
- **File Systems**: Common in file systems to organize and locate files.
- **Data Warehousing**: Supports quick retrieval of large, ordered datasets.

Comparison with B-Trees

- **B+ Trees**: All values are in the leaf nodes, and leaves are linked.
- **B-Trees**: Values and keys are stored at all levels, with no linked list in leaves.

Complexity

- **Insertion/Deletion/Search**: $O(\log N)$ due to tree balance.
- **Range Query**: $O(\log N)$ + $O(K)$ where K is the range length, leveraging linked leaf nodes.

Join Implementations

A Join query is one of the most time-consuming operations in a RDBMS.
Nested Loop Join: Good for small datasets or when one table has an index on the join key.
Merge Join: Efficient for pre-sorted data.
Hash Join: Ideal for larger datasets where no suitable indexes exist.

See: Principles of database management the practical guide to storing, managing and analyzing big and small data, page 906

Nested Loop Join

technique to physically implement a join in which one of the tables is denoted as the inner table and the other becomes the outer table. For every row in the outer table, all rows of the inner table are retrieved and compared to the current row of the outer table. If the join condition is satisfied, both rows are joined and put in an output buffer.

This approach is **mainly effective if the inner table is very small** or if the internal data model provides facilities for efficient access to the inner table, such as by means of a primary or clustered index over the join columns of the inner table. This approach is also more efficient if the filter factor of the other query predicates is very restrictive with respect to the rows that qualify in the inner table.
Suitable for:

- Small datasets.
- Non-equijoin conditions like `>` or `<` in queries. (See example below)

Example of a nested loop join as a **optimizer** for query:

```
SELECT *
FROM Sales S1
JOIN Sales S2 ON S1.ProductID > S2.ProductID
WHERE S1.ProductName = S2.ProductName;
```

Merge Join

With a sort-merge join, the tuples in both tables are **first sorted according to the attribute types involved in the join condition**. Both tables are then traversed in this order, with the rows that satisfy the join condition being combined and put in an output buffer.

In contrast to a Nested Loop Join, every table is traversed only once. This approach is appropriate if many rows in both tables satisfy the query predicates and/or if there exist no indexes over the join columns. The sorting algorithm that is applied in first stage may be quite time consuming, hence a sort-merge join will be more efficient if the tuples in R and/or S are already physically ordered in the stored tables according to the attribute types used in the join condition.
Suitable for:

- Equijoin (`=`) and range conditions, assuming both tables are pre-sorted.
- Large datasets where sorting is feasible.

Hash Join

Builds a hash table for one table based on the join key and probes it with the other table.

The performance of this approach depends on the size of the hash file. If, instead of a physical file, the hash structure can be maintained entirely in internal memory, a hash join can be very efficient.