01 - Introduction & Getting Started

Discusses the syllabus and requirements to succeed in the course.

02 - Foundations

Searching

- Searching is the most common operation performed by a database system.
- In SQL, the SELECT statement is arguably the most versatile and complex operation, used to retrieve specific data based on conditions.

Baseline Search Algorithm: Linear Search

- The simplest and most basic search technique.
- Starts at the beginning of a list and proceeds element by element until:
    - You find what you're looking for.
    - You reach the last element and haven't found it.
- Time Complexity: O(n), where n is the number of records.

Fundamental Database Concepts

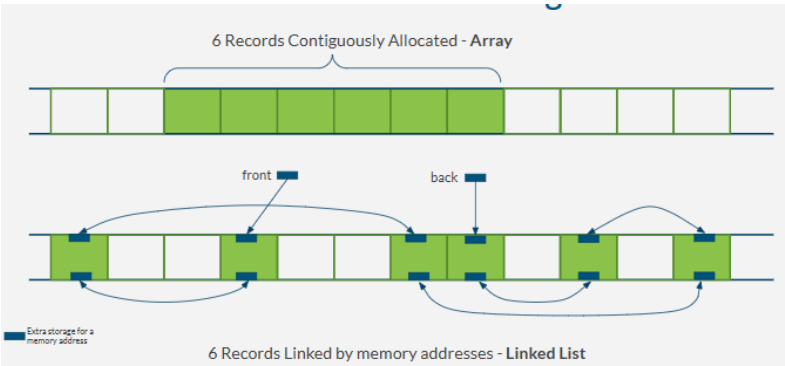
- Record: A collection of values for attributes of a single entity instance; a row of a table.
- Collection: A set of records of the same entity type; a table.
- Trivially stored in some sequential order, like a list.
- Search Key: A value for an attribute from the entity type used to locate records.
    - Can consist of one or more attributes (e.g., composite keys).

Optimized Search Techniques

- Indexing: Improves search performance by creating a sorted data structure (e.g., B-trees, hash indexes).
- Binary Search: Efficient for sorted data, reducing search complexity to O(log n).
- Hashing: Uses a hash function to directly map keys to storage locations, achieving near O(1) search time.
- Full-Text Search: Used for searching textual data efficiently, often with inverted indexes.
- Range Queries: Searching for values within a certain range (e.g., BETWEEN in SQL)
- Pattern Matching: Searching with wildcards (e.g., LIKE in SQL for partial string matches).

Lists of Records

- If each record takes up x bytes of memory, then for n records, we need n*x bytes of memory
- Contiguously (touching) Allocated List: All n*x bytes are allocated as a single "chunk" of memory
- Linked List
    - Each record needs x bytes + additional space for 1 or 2 memory addresses
    - Individual records are linked together in a type of chain using memory addresses



Pros and Cons

- Arrays are faster for random access but slow for inserting anywhere but the end
    - Imagine you have a long row of mailboxes, each with a number on it, 1, 2, 3, and so on. These mailboxes represent an array.
        - If you want to quickly find mailbox #5, you can just go straight to it. That's fast random access (finding things quickly).
        - But if you want to add a new mailbox between #2 and #3, you'd have to shift all the mailboxes after #2 over by one spot. That's slow insertion (hard to add things in the middle)
- Linked lists are faster for inserting anywhere in the list but slower for random access
    - Now, think of a linked list like a treasure hunt, where each clue (a mailbox) tells you where the next one is.
        - If you want to insert a new mailbox between #2 and #3, you just change the clue at #2 to point to the new mailbox and the new mailbox points to #3. That's fast insertion (easy to add things in the middle).
        - But if you want to find mailbox #5, you have to start at #1 and follow the clues one by one until you get there. That's slow random access (hard to find things quickly).
- Arrays: fast for random access, slow for random insertions
- Linked Lists: slow for random access, fast for random insertions

Binary Search

- Input: Array of values in sorted order, target value
- Output: The location (index) of where the target is located or some value indicated the target was not found

Time Complexity

- Linear Search
    - Best case: target is found at the first element; only 1 comparison
    - Worst case: target is not in the array; n comparisons
        - Therefore, in the worst case, linear search is O(n) time complexity
- Binary Search
    - Best case: target is found at mid; 1 comparison (inside the loop)
    - Worst case: target is not in the array; log base 2 n comparisons
        - Therefore, in the worst case, binary search is O(log base 2 n) time complexity

Back to Database Searching

- Assume data is stored on disk by column id's value
- Saerching for a specific id = fast
- But what if we want to search for a specific specialVal?
    - Only option is linear scan of that column
- Can't store data on disk sorted by both id and specialVal (at the same time)
    - Data would have to be duplicated → space inefficient
- We need an external data structure to support faster searching by specialVal than a linear scan

What do we have in our arsenal?

1. An array of tuples (specialVal, rowNumber) sorted by specialVal
    a. We could use binary search to quickly locate a particular specialVal and find its corresponding row in the table
    b. But, every insert into the table would be like inserting into a sorted array - slow…
2. A linked list of tuples (specialVal, rowNumber) sorted by specialVal
    a. Searching for a specialVal would be slow - linear scan required
    b. But inserting into the table would theoretically be quick to also add to the list

Something with Fast Insert and Fast Search?

- Binary Search Tree: a binary search tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent