Lecture 3: Access Methods

Recap

Storage Management

- Database systems have a layered architecture.
- Design of database system components affected by hardware properties.
- Database is physically organized as a collection of pages on disk.
- The units of database space allocation are disk blocks, extents, and segments
- The DBMS can manage that sweet, sweet memory better than the OS.
- Leverage the semantics about the query plan to make better decisions.
- It is important to choose the right storage model for the target workload

Storage Models

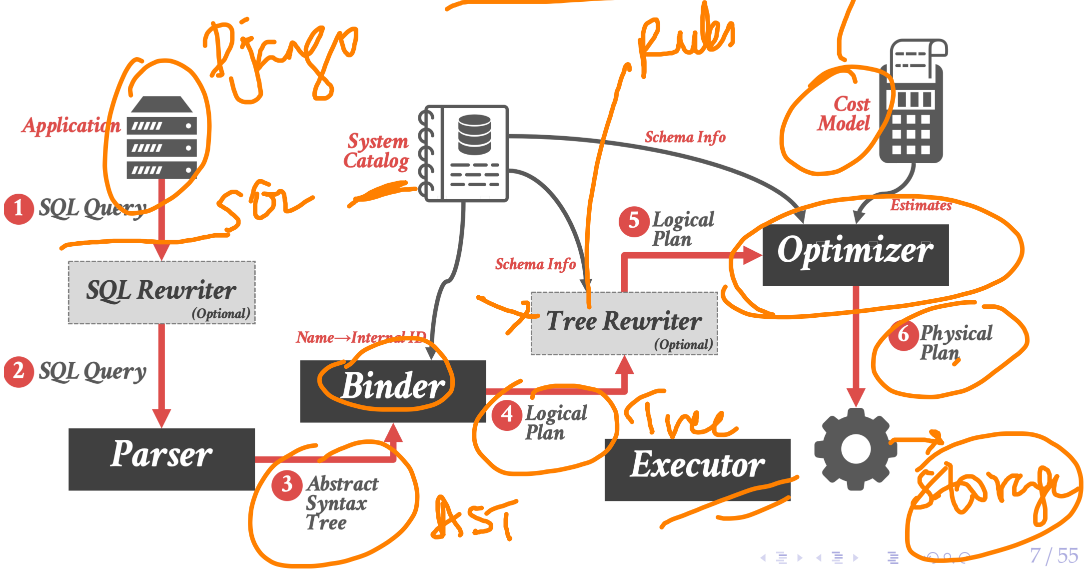
- It is important to choose the right storage model for the target workload
  - OLTP ⟶ Row-Store
  - OLAP ⟶ Column-Store

Today's Agenda

- Access Methods
- Hash Table
- B+Tree
- Index Concurrency Control

Access Methods

Anatomy of a Database System [Monologue]



Anatomy of a Database System [Monologue]

- Process Manager
  - Manages client connections
- Query Processor
  - Parse, plan and execute queries on top of storage manager
- Transactional Storage Manager
  - Knits together buffer management, concurrency control, logging and recovery
- Shared Utilities
  - Manage hardware resources across threads

Txn

Anatomy of a Database System [Monologue]

- Process Manager
  - Connection Manager + Admission Control
- Query Processor
  - Query Parser
  - Query Optimizer (*a.k.a.*, Query Planner)
  - Query Executor
- Transactional Storage Manager
  - Lock Manager
  - Access Methods (*a.k.a.*, Indexes)
  - Buffer Pool Manager
  - Log Manager
- Shared Utilities
  - Memory, Disk, and Networking Manager

Scalability
Availability
Recoverability
Storage

 Access Methods

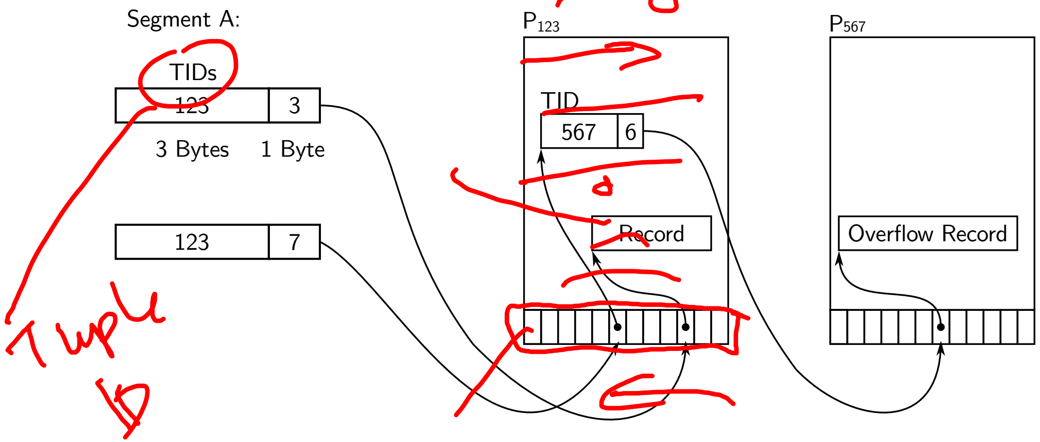
Access Methods

Access methods are alternative **ways for retrieving specific tuples** from a relation.

- Typically, there is more than one way to retrieve tuples.
- Depends on the availability of **indexes** and the conditions specified in the query for selecting the tuples
- Includes **sequential scan** method of **unordered table heap**
- Includes **index scan** of different types of **index structures**

We will look at these methods in more detail.

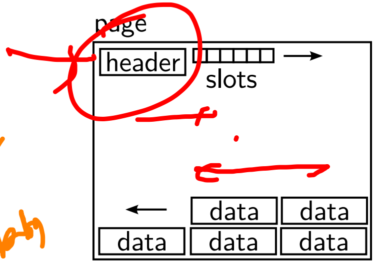
Slotted Pages

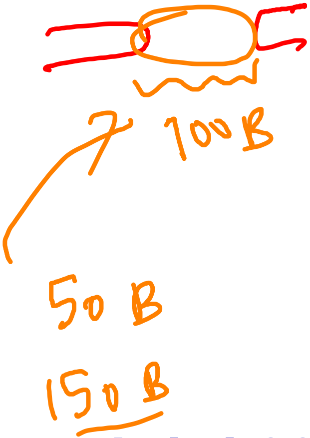


(TID size varies, but will most likely be at least 8 bytes on modern systems)

Slotted Pages (2)

Tuples are stored in slotted pages





- data grows from one side, slots from the other
- the page is full when both meet
- updates/deletes complicate issues, though
- might require garbage collection/compactification

Slotted Pages (3)

Header:

LSN	for recovery
slotCount	number of used slots
firstFreeSlot	to speed up locating free slots
dataStart	lower end of the data
freeSpace	space that would be available after compactification

Note: a slotted page can contain hundreds of entries!
Requires some care to get good performance.

Hash Table

Table Indexes

- A **table index** is a replica of a subset of a table's attributes that are organized and/or sorted for efficient access based a subset of those attributes.
- Example: {**Employee Id**, **Dept Id**} ⟶ Employee Tuple Pointer
- The DBMS ensures that the contents of **the table** and **the indices** are in sync.

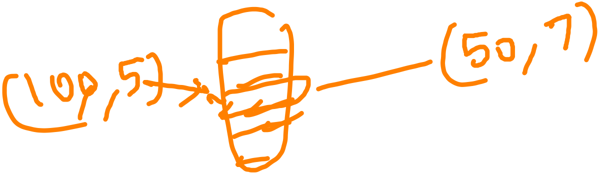


Table Indexes

- It is the DBMS's job to figure out the best index(es) to use to execute each query.
- There is a trade-off on the number of indexes to create per database.
  - Storage Overhead
  - Maintenance Overhead

Table Indexes

Data is often indexed

- speeds up lookup
- de-facto mandatory for primary keys
- useful for selective queries

Two important access classes:

- point queries
  find all tuples with a given value (might be a compound)
- range queries
  find all tuples within a given value range

Support for more complex predicates is rare.

Hash Tables

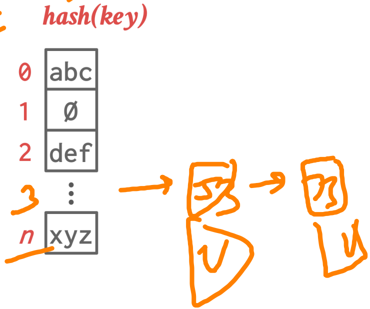
- A hash table implements an **unordered associative array** that maps keys to values.
  - mymap.insert('a', 50);
  - mymap['b']=100;
  - mymap.find('a')
  - mymap['a']
- It uses a **hash function** to compute an offset into the array for a given key, from which the desired value can be found.

Hash Tables

- Operation Complexity:
  - Average: O(1)
  - Worst: O(n)
- Space Complexity: O(n)
- Constants matter in practice.
- **Reminder:** In theory, there is no difference between theory and practice. But in practice, there is.

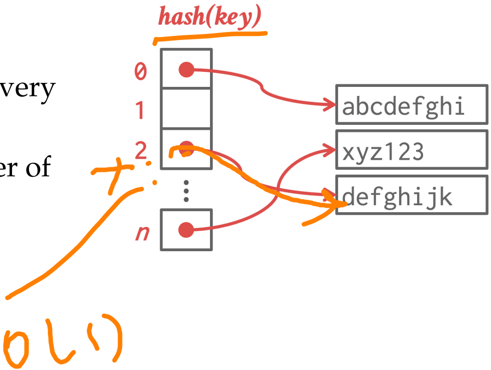
Naïve Hash Table

- Allocate a giant array that has one slot for every element you need to store.
- To find an entry, mod the key by the number of elements to find the offset in the array.



Naïve Hash Table

- Allocate a giant array that has one slot for every element you need to store.
- To find an entry, mod the key by the number of elements to find the offset in the array.



Assumptions

- You know the number of elements ahead of time.
- Each key is unique (*e.g.*, SSN ID ⟶ Name).
- Perfect hash function (no **<u>collision</u>**).
    - If key1 != key2, then hash(key1) != hash(key2)

Hash Table: Design Decisions

- Design Decision 1: **Hash Function**
  - How to map a large key space into a smaller domain of array offsets.
  - Trade-off between being fast vs. collision rate.
- Design Decision 2: **Hashing Scheme**
  - How to handle key collisions after hashing.
  - Trade-off between allocating a large hash table vs. additional steps to find/insert keys.

B+Tree

- point lookup

x = 50

- size | in-memory

O(n)

B-Tree

map | unordered_map

B-Trees (including variants) are the dominant data structure for external storage.

disk

Classical definition:

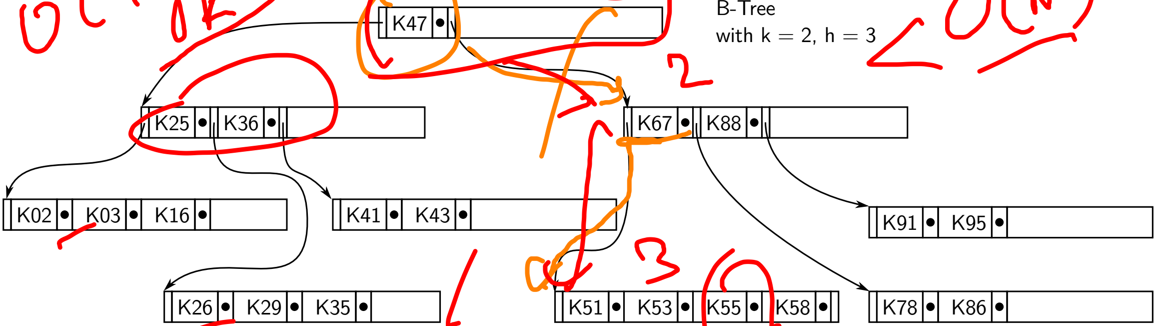
- a B-Tree has a degree k
- each node except the root has at least k entries
- each node has at most 2k entries
- all leaf nodes are at the same depth

Storage Amplification

C++: RBTree

B-Tree (2)

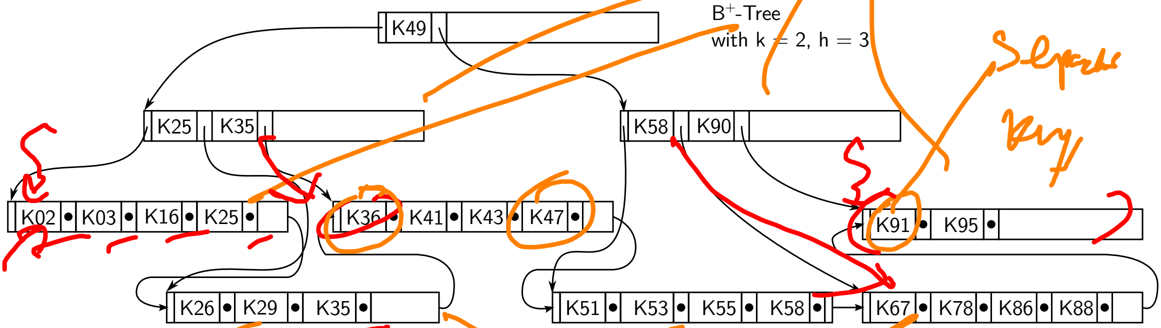
Example:



The • is the TID of the corresponding tuple.

B^+-Tree

Most DBMS use the B^+-Tree variant:



- key+TID only in leaf nodes
- inner nodes contain separators, might or might not occur in the data
- increases the fanout of inner nodes
- simplifies the B-Tree logic

Page Structure

Inner Node:

LSN	for recovery
upper	page of right-most child
count	number of entries
key/child	key/child-page pairs
...	...

Leaf Node:

LSN	for recovery
0	leaf node marker
next	next leaf node
count	number of entries
key/tid	key/TID pairs
...	...

Similar to slotted pages for variable keys.

Index Concurrency Control

Index Structures: Design Decisions

- **Meta-Data Organization**
  - How to organize meta-data on disk or in memory to support efficient access to specific tuples?
- **Concurrency**
  - How to allow multiple threads to access the derived data structure at the same time without causing problems?

Observation

- We assumed that all the data structures that we have discussed so far are single-threaded.
- But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.

Concurrency Control

- A **concurrency control protocol** is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.
- A protocol's correctness criteria can vary:
  - **Logical Correctness:** Am I reading the data that I am supposed to read?
  - **Physical Correctness:** Is the internal representation of the object sound?

Locks vs. Latches

- **Locks**
  - Protects the database's **logical contents** from other txns.
  - Held for the duration of the transaction.
  - Need to be able to rollback changes.
- **Latches**
  - Protects the critical sections of the DBMS's internal **physical data structures** from other threads.
  - Held for the duration of the operation.
  - Do not need to be able to rollback changes.

Locks vs. Latches

	Locks	**Latches**
Separate. . .	User transactions	Threads
Protect. . .	Database Contents	In-Memory Data Structures
During. . .	Entire Transactions	Critical Sections
Modes. . .	Shared, Exclusive, Update, Intention	Read, Write (*a.k.a.*, Shared, Exclusive)
Deadlock	Detection & Resolution	Avoidance
. . . by. . .	Waits-for, Timeout, Aborts	Coding Discipline
Kept in. . .	Lock Manager	Protected Data Structure

Reference

Latch Modes

- **Read Mode**
  - Multiple threads can read the same object at the same time.
  - A thread can acquire the read latch if another thread has it in read mode.
- **Write Mode**
  - Only one thread can access the object.
  - A thread cannot acquire a write latch if another thread holds the latch in any mode.

	Read	Write
Read	✓	X
Write	X	X

Latch Implementations

- Blocking OS Mutex
- Test-and-Set Spin Latch
- Reader-Writer Latch

Latch Implementations

- **Approach 1: Blocking OS Mutex**
  - Simple to use
  - Non-scalable (about 25 ns per lock/unlock invocation)
  - Example: std::mutex

```
std::mutex m;

m.lock();
// Do something special...
m.unlock();
```

Latch Implementations

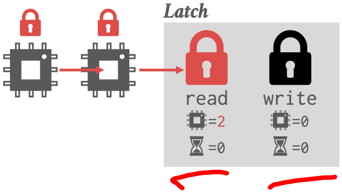
- **Approach 2: Test-and-Set Spin Latch (TAS)**
  - Very efficient (single instruction to latch/unlatch)
  - Non-scalable, not cache friendly
  - Example: std::atomic<T>
  - Unlike OS mutex, spin latches do **not** suspend thread execution
  - Atomic operations are faster if contention between threads is sufficiently **low**

```
std::atomic_flag latch; // atomic of boolean type (lock-free)

while (latch.test_and_set(...)) {
        // Retry? Yield? Abort?
}
```

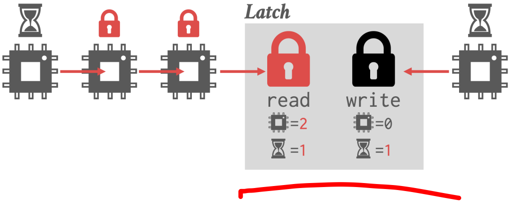
Latch Implementations

- **Approach 3: Reader-Writer Latch**
  - Allows for concurrent readers
  - Must manage read/write queues to avoid starvation
  - Can be implemented on top of spinlocks



Latch Implementations

- **Approach 3: Reader-Writer Latch**
  - Allows for concurrent readers
  - Must manage read/write queues to avoid starvation
  - Can be implemented on top of spinlocks



B+Tree Concurrency Control

- We want to allow multiple threads to read and update a B+Tree at the same time.
- We need to handle two types of problems:
  - Threads trying to modify the contents of **a node** at the same time.
  - One thread **traversing** the tree while another thread splits/merges nodes.

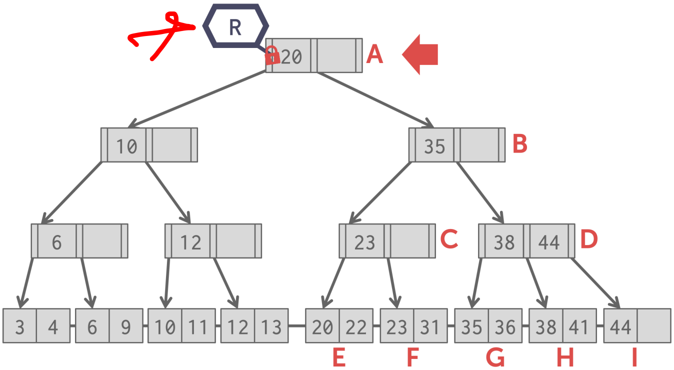
Latch Crabbing/Coupling

- Protocol to allow multiple threads to access/modify B+Tree at the same time.
- Basic Idea:
  - Get latch for parent.
  - Get latch for child
  - Release latch for parent if "safe".
- A **safe node** is one that will **not split or merge** when updated.
  - Not full (on insertion)
  - More than half-full (on deletion)

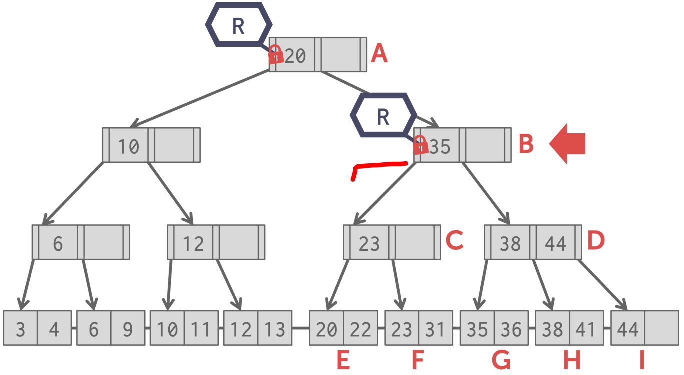
Latch Crabbing/Coupling

- **Find:** Start at root and go down; repeatedly,
  - Acquire **R** latch on child
  - Then unlatch parent
- **Insert/Delete:** Start at root and go down, obtaining **W** latches as needed. Once child is latched, check if it is safe:
  - If child is safe, release all latches on ancestors.

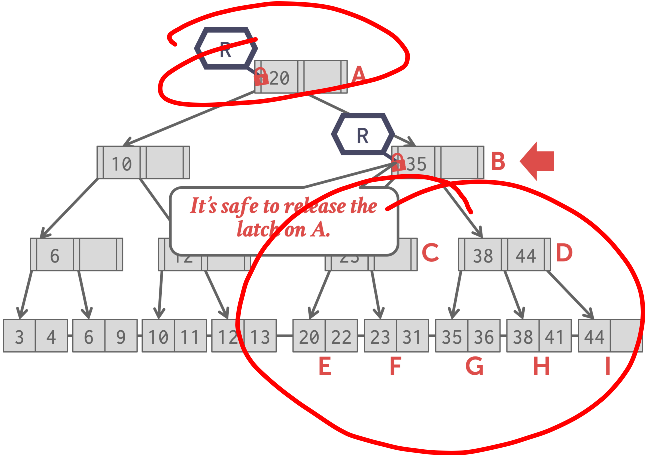
Example 1 - Find 38



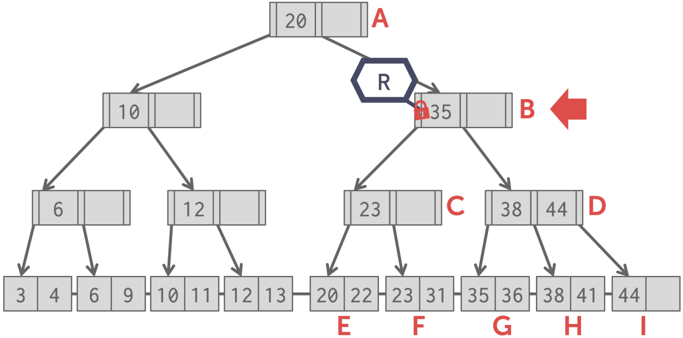
Example 1 - Find 38



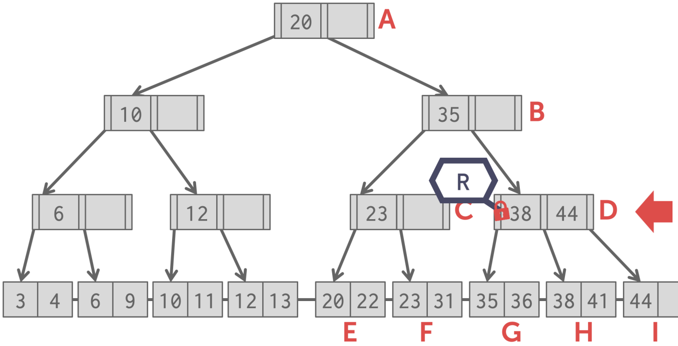
Example 1 - Find 38



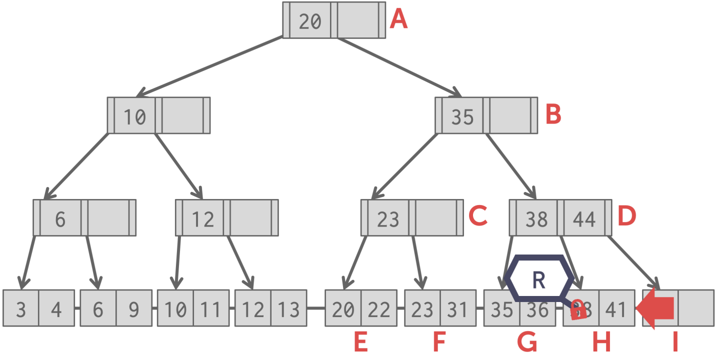
Example 1 - Find 38



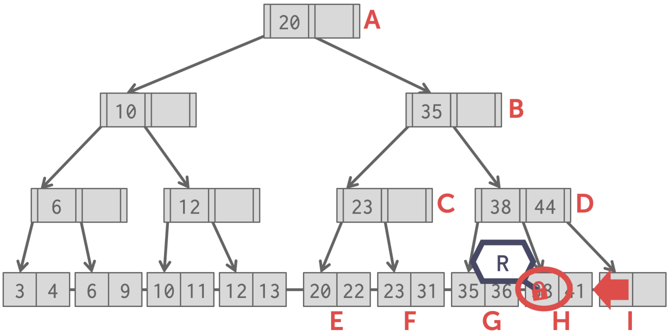
Example 1 - Find 38



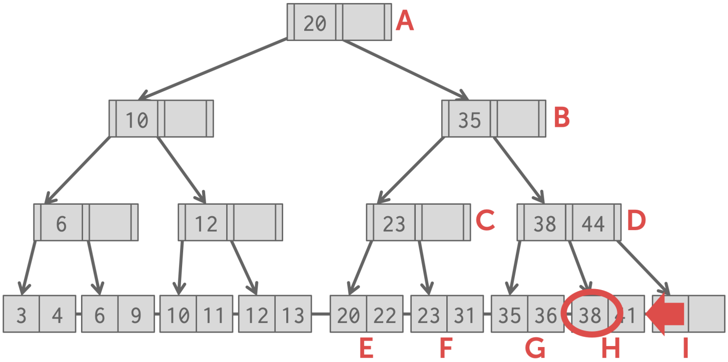
Example 1 - Find 38



Example 1 - Find 38



Example 1 - Find 38



Conclusion

Parting Thoughts

- Access methods are the alternative ways for retrieving specific tuples
- We covered two access methods: sequential scan and index scan
- Sequential scan is done over an unordered table heap
- Index scan is done over an ordered B-Tree or an unordered hash table
- Hash tables are fast data structures that support O(1) look-ups

Parting Thoughts

- Hash tables are usually **not** what you want to use for a indexing tables
  - Lack of ordering in widely-used hashing schemes
  - Lack of locality of reference $\longrightarrow$ more disk seeks
  - Persistent data structures are much more complex (logging and recovery)
  - Reference
- The venerable B+Tree is always a good choice for your DBMS.
- Making a data structure thread-safe is notoriously difficult in practice.
- We focused on B+Trees but the same high-level techniques are applicable to other data structures.

R-trees

Next Class

- Recap of query processing