Recap

Sort Data by 'Age'
- Makes it possible to filter for 'age' = _ / age > _ / age < _ / _ < age < _
  - Binary search to first record, scan to last record

Store Data in Chunk Pages
- Makes it easier to jump to records if you don't have fixed-size records
- Works well with Cache Lines / SSD Pages / HDD Pages
- We discussed a few layout strategies

Summaries
- Index Pages allow you to load fewer pages when doing a binary search
- Still need to binary search within a page
- Quick analysis: How many pages will get loaded in a binary search?
  - Binary Search: log_2 N
  - w/ Index Pages: log_k N (where k is the number of "keys" on a page)

Primary vs Secondary Indexes

Challenges
- Can't handle multiple attributes?

(Nonve) Idea 1: Store multiple copies of the index

Pros
- Can support multiple attributes
- "Easy" to implement

Cons
- Tons of space wasted
- Updates: Have to keep multiple pages in sync

Idea 2: Sort on Tuples of attributes (e.g., <Age, Rank> or <Rank, Age>)

Pros
- Can support some queries for multiple attributes
- Can simultaneously filter on multiple attributes

Cons
- Can only support some queries for multiple attributes

Idea 3: Add a layer of indirection
- Instead of <key, rest of record> pages, store <key, page # with full record>

Pros
- Supports multiple attributes with relatively few caveats
- Minimal space overhead
• Minimal update overheads... but...

▼ Cons
• Still need to binary search through the target page
• Makes it hard to do reorganization... the target record is locked to that one page

▼ Idea 4: Primary Keys
▼ Instead, use: <search key, record key> (call it a Primary Key)
• Typically called a “Secondary” Index
▼ Have a separate index that maps record key to record
• Typically called a "Primary" or "Clustered" Index

▼ Pros
• Supports multiple attributes with almost no caveats
• Minimal space overhead
• Virtually no update overhead

▼ Cons
▼ Adds a log_k(N) lookup factor (Multiply cost by log_k(N))
• If we’re clever we can often cut this down to just a flat additional log_k(N) cost (Add log_k(N) to cost). How?
• This trick also helps us make accesses sequential (good for HDDs)

▼ Observation (Time Permitting): Multiple Secondary Keys
• The same trick can be used to help us satisfy multiple queries on secondary keys

▼ Handling Changing Data

▼ Challenges
• Can’t insert into the middle of a sorted file
• Can’t insert into a packed (sorted) summary page

▼ Idea 1: Out-of-order pages (B+Tree-Ish Indexes)
▼ Treat pages as atomic blobs of storage (rather than a single contiguous region)
• Bonus: Don’t need fixed-size records
• Leave empty space on each data page and each summary (tree) page
▼ What to do when a page “fills up” or “empties out”?
• Shift records to/from other pages at the same level (pivot)
• Merge two pages together
• Create a new level / flatten a level

▼ Degenerate case:
• Super-tall structure

▼ Idea 2: As above, but maintain size invariant (B+Tree)
• Invariant 1: Uniform Tree Depth
• Invariant 2: 50% ≤ fill ≤ 100% (for all except root page)
▼ When page drops below 50% fill, merge with adjacent page
  • Recur higher if necessary
▼ When page exceeds 100% fill, split into 2 pages
  • Recur higher if necessary
• When root drops to 1 pointer, reduce depth by 1
• When root exceeds capacity, increase depth by 1
• Optimization: Borrow/Loan records/[key+pointer]s from/to adjacent pages
▼ Analysis:
  • Balanced, so worst case == common case
  • Fill = at worst 1/2, so the tree is half-unused (i.e., we have space for 2N records, but are only using N)
▼ log_k(2N) vs log_k(N) best case
  • log_k(2N) = log_k(2) + log_k(N) ~= at worst 1 more level of tree than we really need
▼ Worst case behavior
▼ Alternating Insertions / Deletions occurring on a 50%/100% boundary:
  • Every insert triggers a split
  • Every delete triggers a merge
  • Doesn’t happen very often...
  • Borrow/Loan help prevent this
  • Other ideas: Background task to continuously rebalance tree away from dangerous split/merge thresholds