Disks

Disks - tracks, cylinders, sectors, blocks/pages
Rotational latency, seek times
Access time: seek time + rotational latency (averaged); seek time is the time to move the drive head in/out to the proper track. Average seek times for desktop drives is 9ms. Rotational latency (average) for a server drive may be as low as 2ms (depends on RPMs, which range from 5400 - 15000). Can do the math on that one: 1m / 15000 RPMs * 60s/1m = 0.004 s = 4ms.

So average access time 10ms. However, once found, a block may be read very quickly. A block is anywhere from 512 bytes to 16kb. Once in memory, processing of 16kb is *much* faster than reading from disk.

Other factors: buffering (lookahead). New media (SSDs, 3D XPoint memory)

File Organization

Consider how to store a relation on disk. Various schemes with pluses and minuses, but let’s take one tack: ordered storage by primary key.

- Cram records into blocks, maintain some kind of record of block order
- Reads can use binary search, so \( \log_2 \) performance on single-record lookups. Plus range scans are relatively inexpensive.
- Inserts may be expensive - need to move all records out of way, plus may need to allocate new blocks
- Deletions likewise expensive if closing up gaps; alternately keep space for later insertions, but that can waste space

Insight: binary log is great, but suppose we have 100M records? What is cost? Assume we can stuff 100 records into one block. Then we have \( 2^{20} \) (1M) blocks to search through \( \rightarrow 20 \) disk accesses @ 10ms = 200ms. Solution: create a secondary level “index” which simply tracks the keys at the beginnings of each block. Now we can stuff 100 (or more!) of these into a block and search them using binary search. Since we have 1/100th the number of keys to search, we reduce our binary search cost by about 6 - bringing us down to 140 + 10ms - we are still searching 10000 blocks (ie \( 2^{14} \)). Add another level: another \( x100 \) reduction, so now we search only 100 blocks - 70 + 10 + 10 ms. Do it again, final level all fits into one block. Now our cost is down to 30ms, the number of levels in our index hierarchy (this works because: blocks are ready quickly, and a block can be processed much, much faster once in memory than I/O).

E.g., performance is no longer \( \log_2 \) but more like \( \log_{100} \). Another way of looking at this is that instead of dividing search space by 2 each time, we divide search space by 100.

This basic idea dovetails with the data structure called the B-tree.

B-Trees

Balanced tree structure with high number of entries in each node. For order \( m \) B-tree, we have at most \( m \) children. We always have 1 more child pointer than nodes, search is performed exactly like for binary tree (ish).

Rules (Knuth):

1. Every node has at most \( m \) children
2. Every non-leaf node (except root) has at least \( \lceil m/2 \rceil \) children
3. The root has at least two children unless it is itself a leaf
4. A non-leaf node with \( k \) children has \( k - 1 \) keys.

5. All leaves appear on same level

The rules above work best for odd \( m \).

Best/worst case heights with \( n \) keys:

Best is \( \lceil \log_m(n + 1) \rceil \)

Worst is \( \lfloor \log_{m/2}(\frac{n+1}{2}) \rfloor \)

Algorithms:

- Searching: as for binary tree (can also use b-search within node)
- Insertion: insert at leaf level. If leaf has no room split leaf and add median value as separator value in parent. If parent has no room, recursively manage that situation by splitting, etc. If you get to root and root has no room, then split root and add a level to tree.
- Deletion: if value is in leaf, simply delete, then rebalance if underflow (if less than \( \lceil m/2 \rceil \) keys). If internal node, then it has children. Find nearest leaf child (max in left subtree or min in right subtree) to replace it with. Now we again have a value removed from a leaf - rebalance if underflow.
- Rebalancing on underflow: if right sibling exists, has more than minimum \# of elements, do a left rotation to borrow element. Otherwise, if left etc, do a right rotation. Else must merge two nodes (pick either right or left to combine with). Since we are under, we can take the separating value from parent and insert between values in two nodes (must do this). Now parent may be underfull, so have to recursively rebalance it. Root can be less than minimum - if empty, then can remove root (shorten tree, yay!).

“Secondary” Indexes

Only kind postgres has

Index stores pointers into files with records, which are not stored in any particular order w.r.t. index keys

If records are ordered w.r.t. PK, then secondary indexes used for non-PK attributes

Other considerations: non-unique keys, etc.

Note that PostgreSQL always creates indexes (B-tree) for unique and PK attributes

Can also create indexes if desired

Talk more about impact of indexes when discuss query planning and execution, performance tuning.