Lecture 4 - Adaptive Data Structures

Data structures -> fundamental areas of computer science; they form the core of data systems, compilers, operating systems, human-computer interaction systems, networks, and machine learning.

The role of data structures: A data structure defines how data is physically stored. For all algorithms that deal with data, their design starts by defining a data structure that minimizes computation and data movement.

For example, we can only utilize an optimized sorted search algorithm if the data is sorted and if we can maintain it efficiently in such a state.

There Is No Perfect Data Structure Design

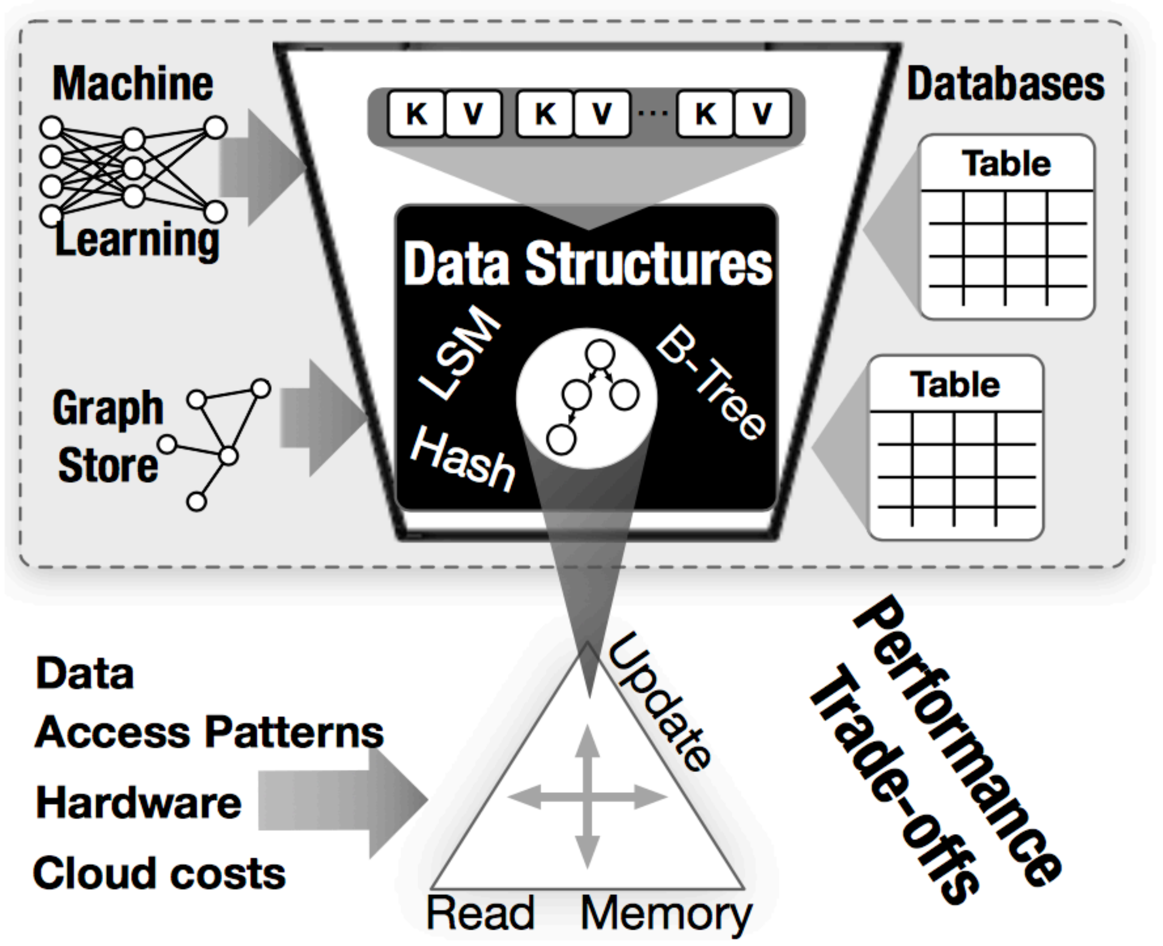


Figure 1: Design trade-offs.

Read amplification is how much excess data an algorithm is forced to read; due to hardware properties such as page-based storage, even when reading a single data item, an algorithm has to load all items of the respective page.

Write amplification is how much excess data an algorithm has to write; maintaining the structure of the data during updates typically causes additional writes.

Memory amplification is how much excess data we have to store; any indexing that helps navigate the raw data is auxiliary data.

For example, a Log-Structured Merge-tree (LSM-tree) relies on batching and logging data without enforcing a global order.

While this helps with writes, it hurts reads since now a single read query (might) have to search all LSM-tree levels.

Similarly, **a sorted array enforces an organization in the data which favors reads but hurts writes**, e.g., inserting a new item in a sorted (dense) array requires rewriting half the array on average. In this way, there is no universally good design.

How can one think of adaptive data structure design?

Data structures such as indexes are **models**

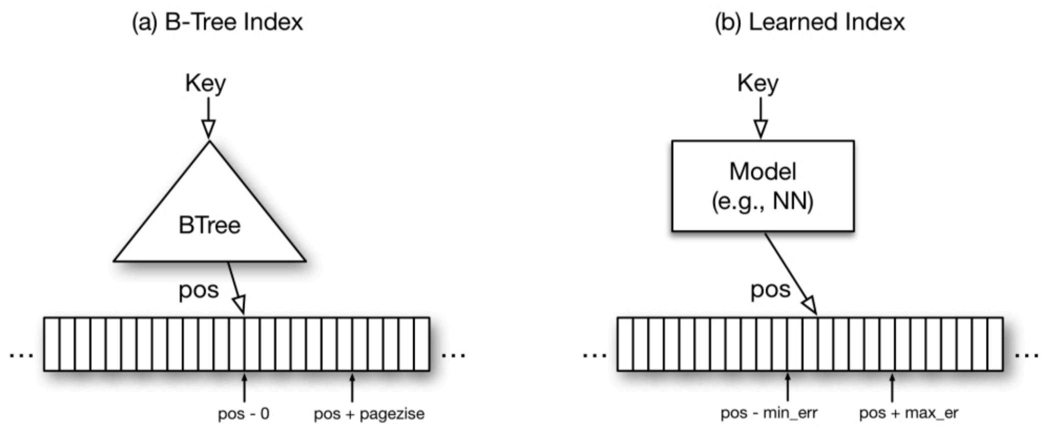


Figure 1: Why B-Trees are models

You give them a Key and they map it to a position in memory.

Example consider a sorted array: A model that predicts the position of a key within a sorted array is effectively approximating the cumulative distribution function (CDF):

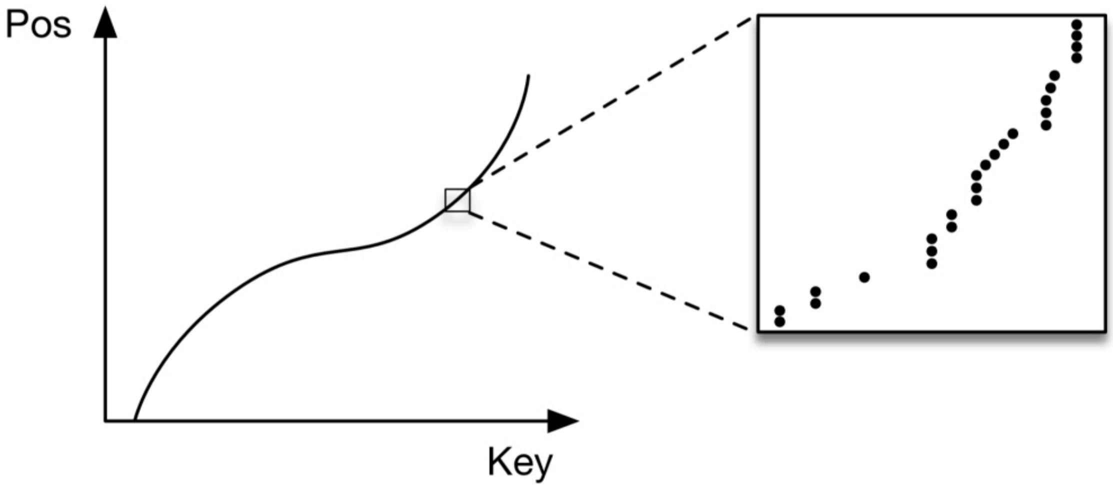
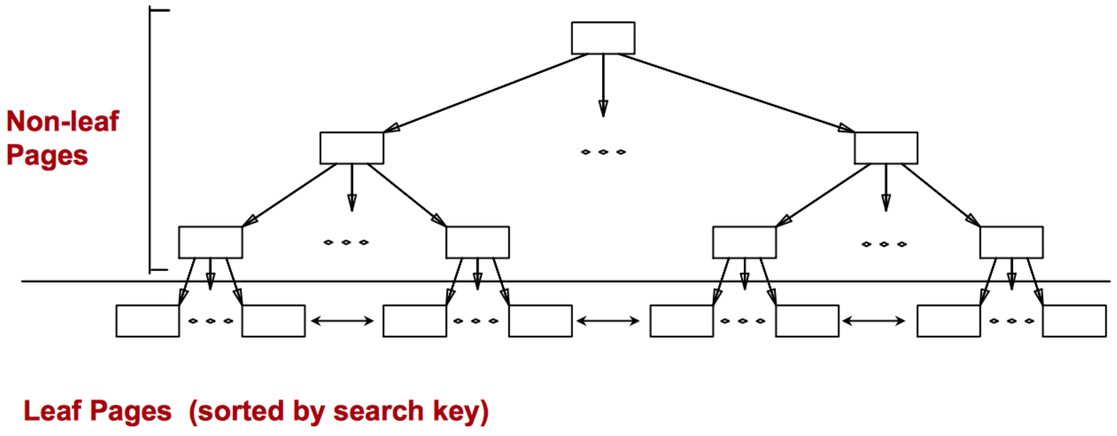


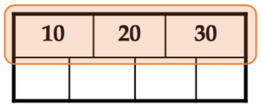
Figure 2: Indexes as CDFs

Approach 1
Fit the CDF such that we find a function (model) that minimizes the cost of lookup. Here, overfitting is good: we want to capture the precise nuances of our exact data as precisely as possible (to identify locations with minimum lookups.

Why does a B+-tree overfit?
Let's revisit a B+-tree and thing about this when reading the paper on Thursday



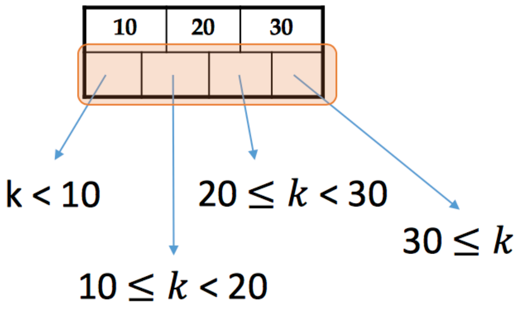
Parameter ***d*** = the order



Each *non-leaf ("interior")* ***node*** has d $\leq m \leq$ 2d ***entries***

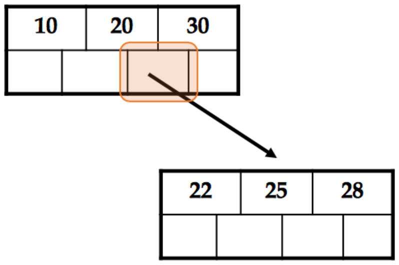
- ***Minimum 50% occupancy***

Root ***node*** has 1 $\leq m \leq$ 2d ***entries***

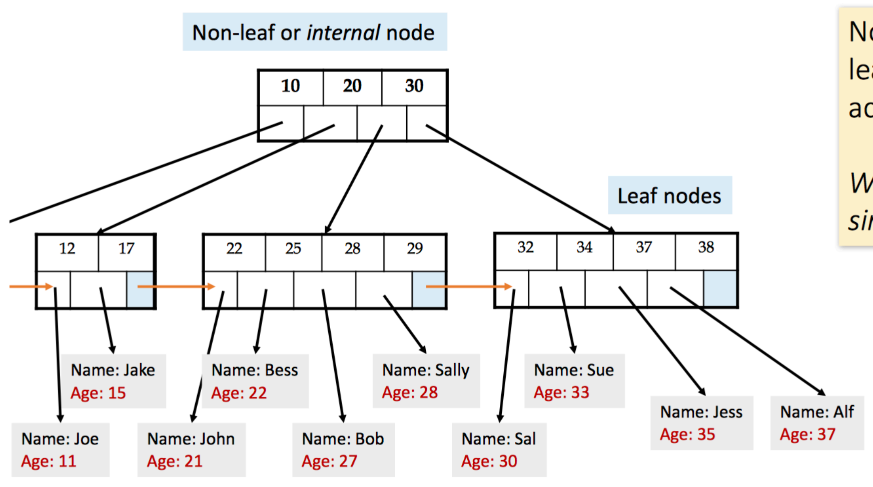


The *n* entries in a node define *n+1* ranges

Non-leaf or *internal* node



For each range, in a *non-leaf* node, there is a **pointer** to another node with entries in that range

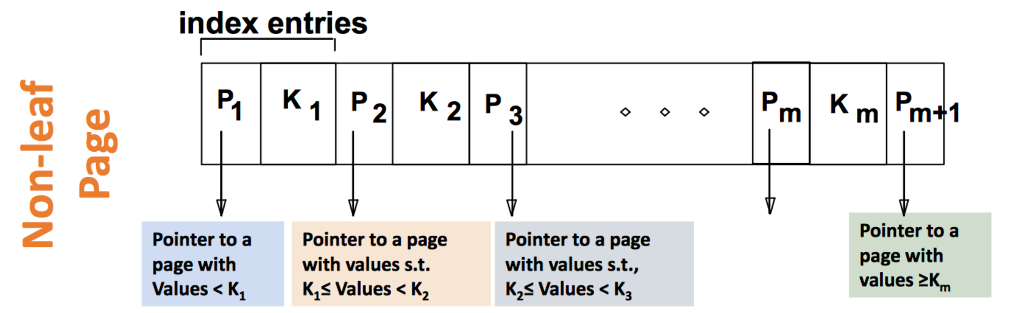


Note that the pointers at the leaf level will be to the actual data records (rows).

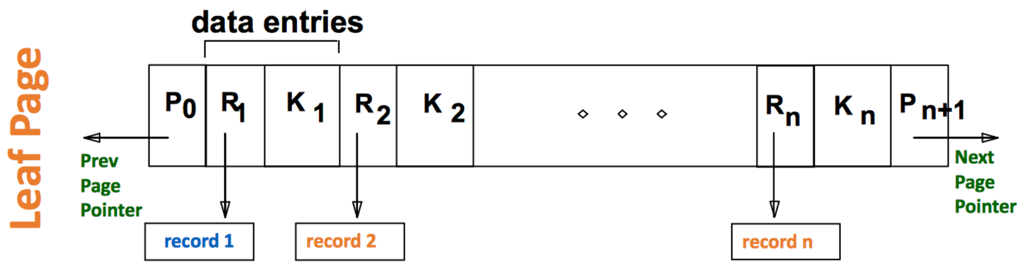
We might truncate these for simpler display (as before)…

Height = 1

B+ Tree Page Format



index entries
Non-leaf Page
P_1
K_1
P_2
K_2
P_3
P_m
K_m
P_{m+1}
Pointer to a page with Values < K_1
Pointer to a page with values s.t. $K_1 \leq$ Values < K_2
Pointer to a page with values s.t., $K_2 \leq$ Values < K_3
Pointer to a page with values $\geq K_m$



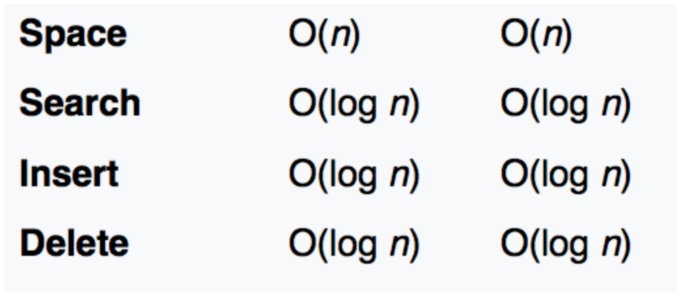
Questions to think of:

What is a B+-Tree good for?

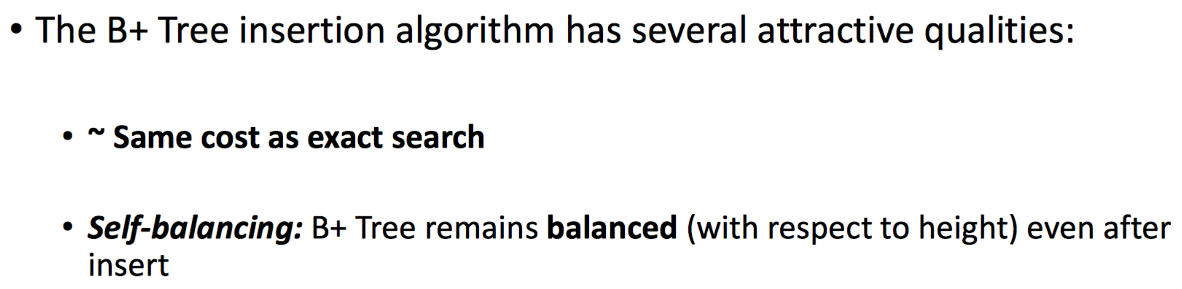
Cost of retrieval

Cost of insertion

n: data entries



Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)



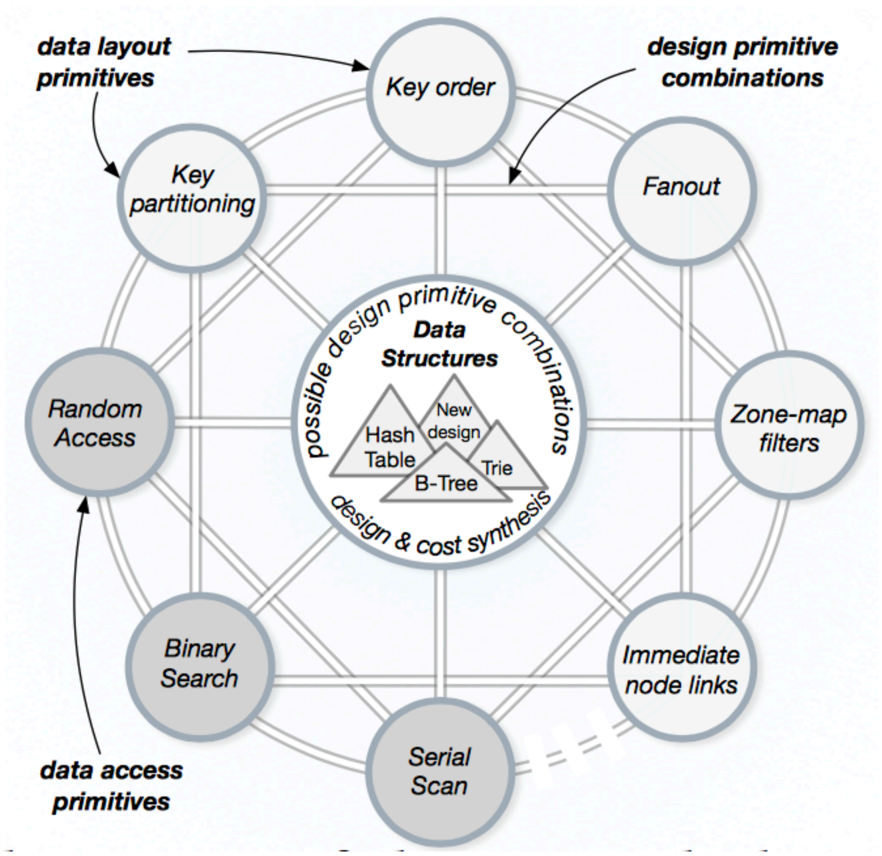
- The B+ Tree insertion algorithm has several attractive qualities:
    - **~ Same cost as exact search**
    - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

For a comparison between learned indexes and B+Trees read http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html

Approach 2 (Extension)
As we saw there are other considerations, retrieval is not the only concern, so how do we navigate the previous optimization space?

We can start synthesizing data structures from basic components (search problem over the space described above).

See: https://www.eecs.harvard.edu/~kester/files/datacalculator.pdf



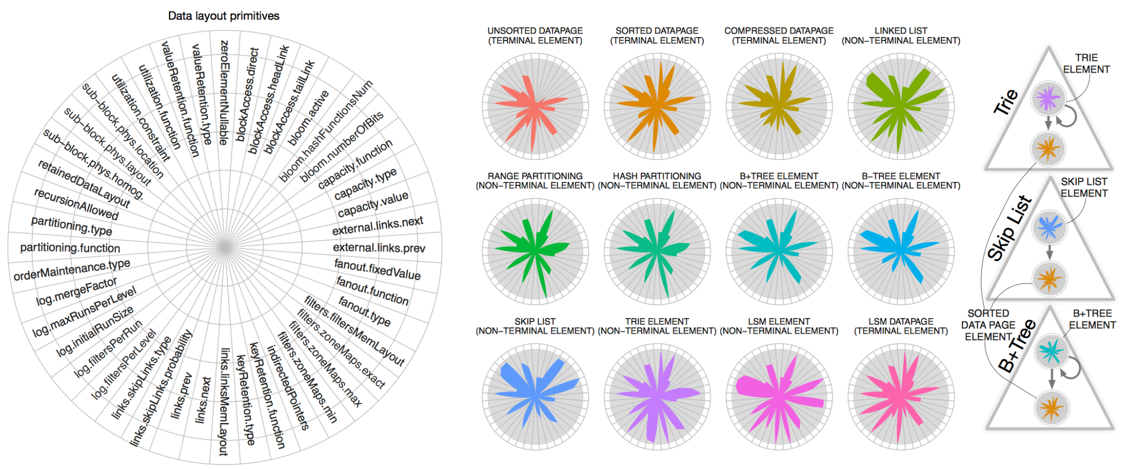


Figure 3: The data layout primitives and examples of synthesizing node layouts of state-of-the-art data structures.

Start synthesizing structures given the space of data layout primitives, data access primitives, constraints due to data and query workloads and hardware profile.

Use samples over random workloads and hardware to collect statistics and learn different models of how different design choices operate on different hardware platforms.

Tricks for efficient search from Idreos et al.

1. **Design Continuums** that allow us to search fast within pockets of the design space.
2. **Performance Constraints** that provide application, user, and hardware based bounds.
3. **Learned Shortcuts** to accelerate a new search by learning from past results.
4. **Practical Search Algorithms** that utilize continuums, constraints, and shortcuts.

We need search algorithms that navigate the possible design space to automatically design data structures which are close to the best option (if not the best) given a desired workload and hardware.

And the usual suspect appears: Reinforcement Learning for effective search. Why is reinforcement learning a good fit for systems? We have access to very accurate simulations.

The above is an instance of hyper-parameter optimization! We will revisit hyper-parameter optimization later in the class. For not let's focus on simple search strategies:

http://www.cs.cornell.edu/courses/cs4787/2019sp/notes/lecture14.pdf