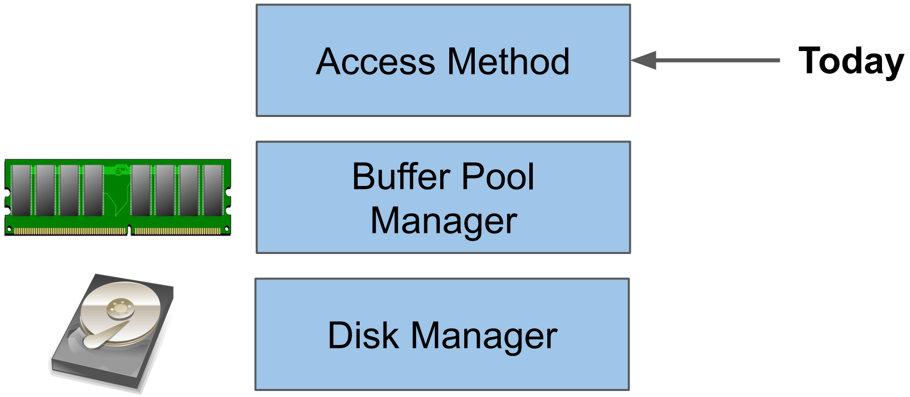
Databases and Big Data

Index

Recap

- Database stores data in files
- Disk Manager decides page layout on disk
- Buffer Manager moves pages in and out of memory
- OS is **not** your friend



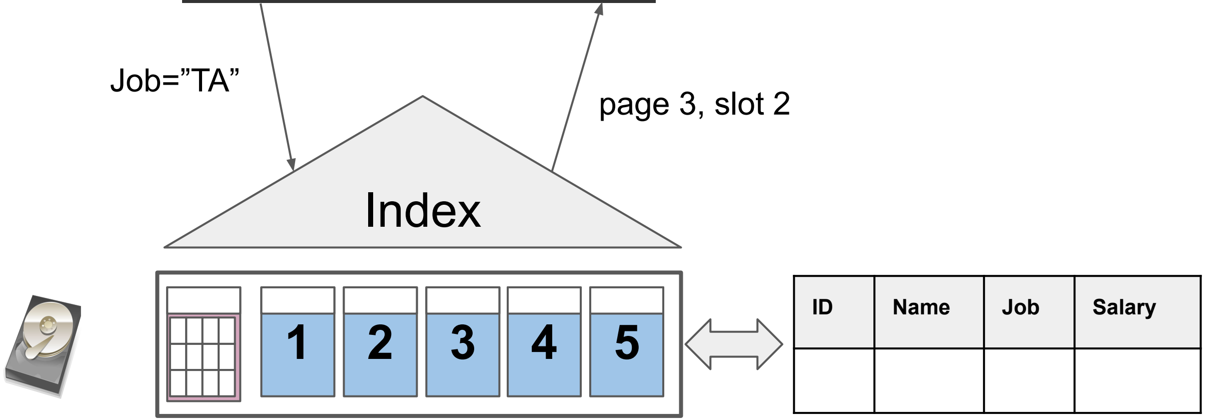
Access Method

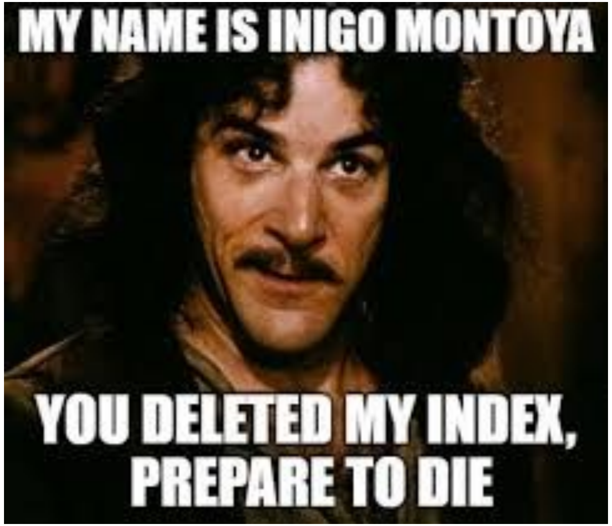
- Access methods:
  - Data structures and algorithms to access data with minimum I/O cost
- Index
  - Additional data structure for efficient data access
  - Most popular access method. Almost synonymous.

Index

- Use index whenever one is available

```
select * from Payroll where
Job="TA";
```

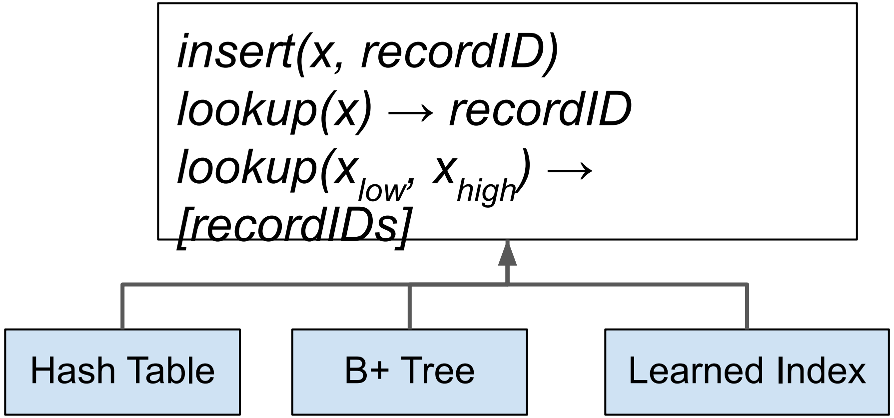




Index

Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and a pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it.

- Definition:
  - $f(x) \rightarrow location$
- APIs



WARNING: one-size-fit-all index doesn't exist.

Designing Access Methods: The RUM Conjecture

Manos Athanassoulis* Michael S. Kester* Lukas M. Maas* Radu Stoica†
Stratos Idreos* Anastasia Ailamaki‡ Mark Callaghan◇
*Harvard University †IBM Research, Zurich ‡EPFL, Lausanne ◇Facebook, Inc.

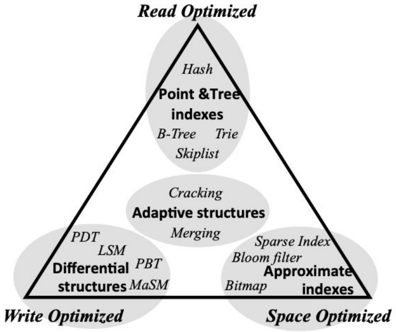


Figure 1: Popular data structures in the RUM space.

Index

- Most DBMSes create index implicitly
  - On primary key
  - On UNIQUE constraints
- User creates index explicitly

```
create index user on Payroll(UserID);
```

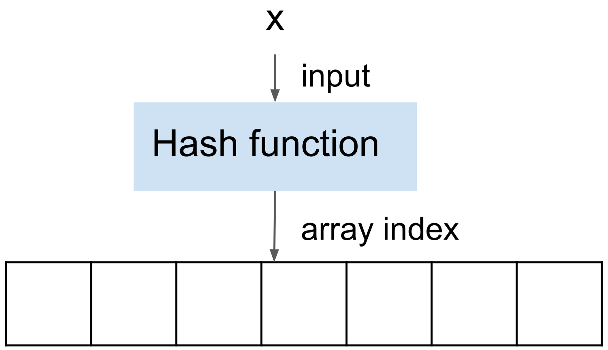
```
create table Payroll (UserID integer primary key,
                      Name varchar(100),
                      Job varchar(100),
                      Phone varchar(100) unique,
                      Salary integer);
```

```
create index phone on Payroll(Phone);
```

Hash Table

- You already know how to use it
  - Python's dictionary
- Implement associative array abstraction:
  - Map key to value
- Use a hash function H
  - Input: key
  - Output: offset in the array

```
x = {}
x['a'] = 1234
'B' in x
...
```

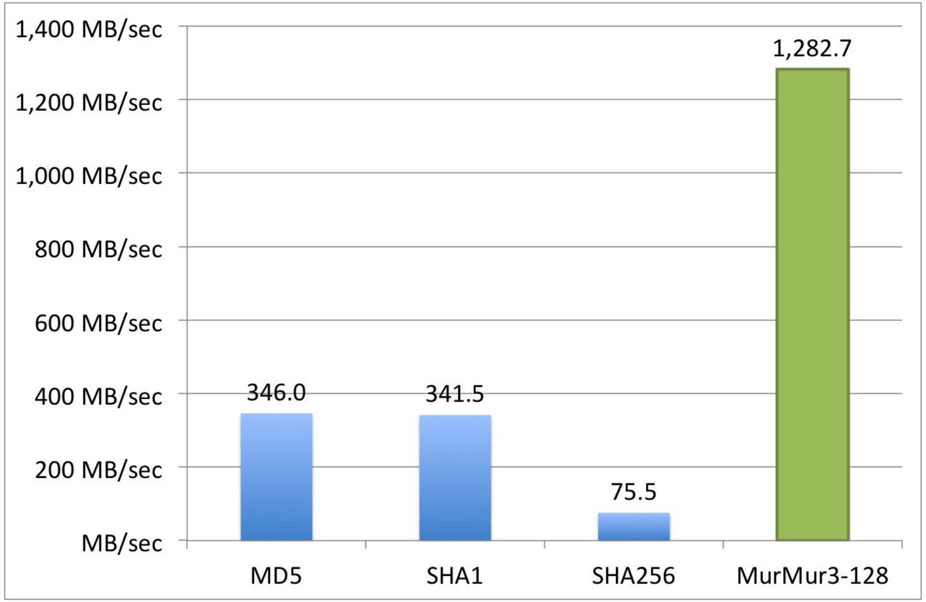


Hash Table

- Problem 1: what hash function to use?
  - Desired property: fast + low collision rate
  - Collision: $H(x_1) = H(x_2)$ when $x_1 \ne x_2$
- Bad hash function:
  - $H(x) = x \% 2$
- Good, but slow:
  - Cryptographic hash functions: SHA-1, SHA-2

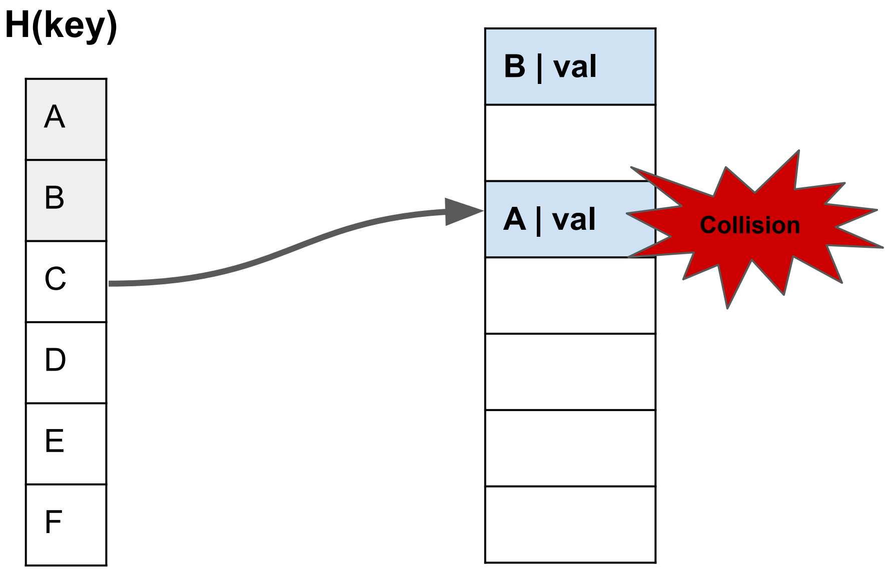
Hash Table

- Hash function in practice
  - MurmurHash
  - GoogleCityHash
  - GoogleFarmHash
  - CLHash
- Trade low-collision rate for speed



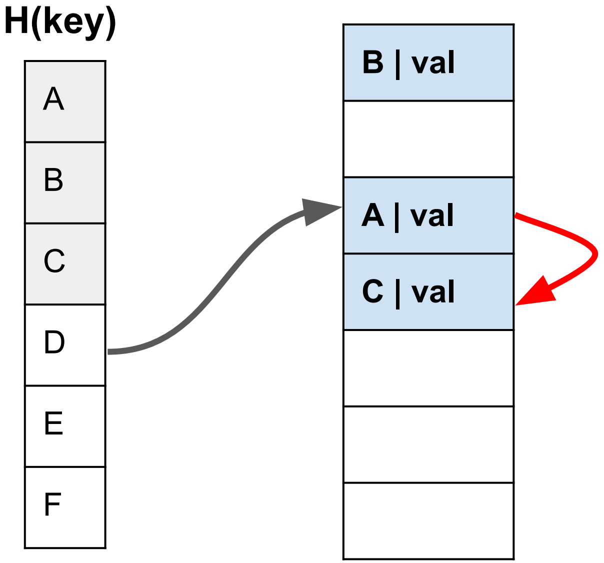
Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing



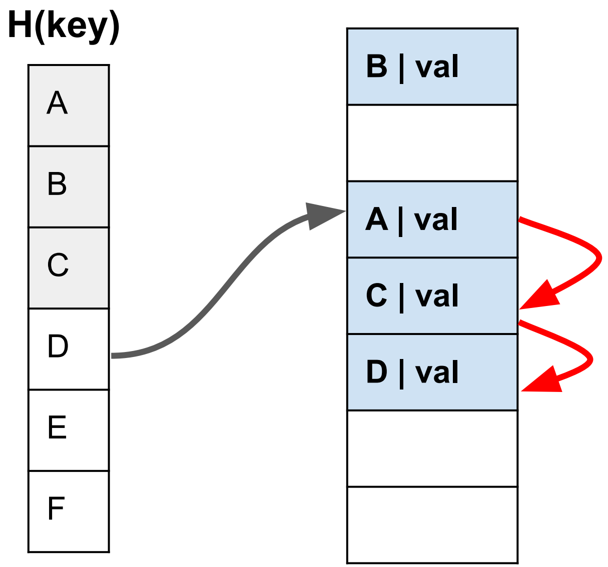
Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Linear Probing
  - Search for the next available slot



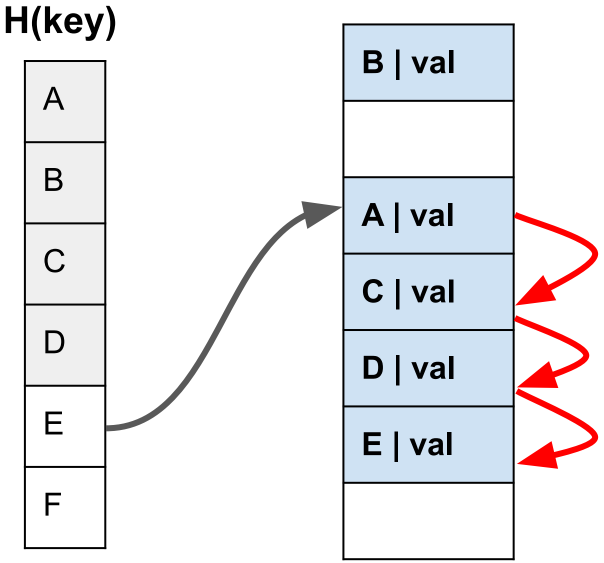
Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Linear Probing
  - Search for the next available slot



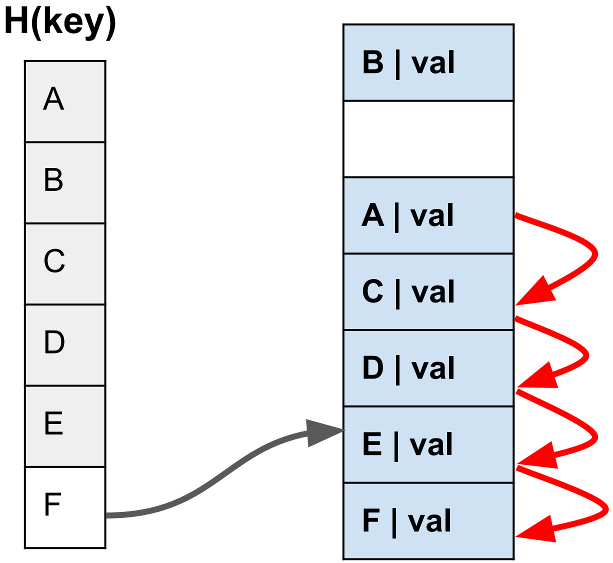
Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Linear Probing
  - Search for the next available slot



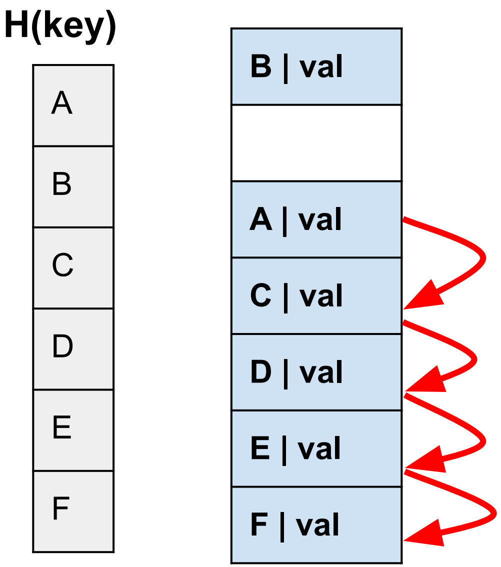
Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Linear Probing
  - Search for the next available slot



Hash Table

- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Linear Probing
  - Search for the next available slot
  - Worst case: O(N) lookup
  - In practice: very good when 50% full



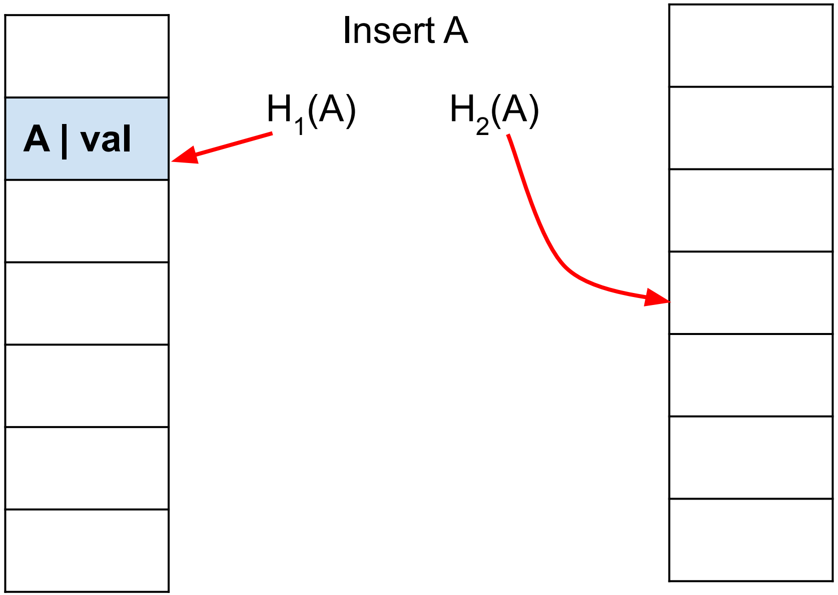
https://s3.amazonaws.com/learneroo/visual-algorithms/ClosedHash.html

Hash Table

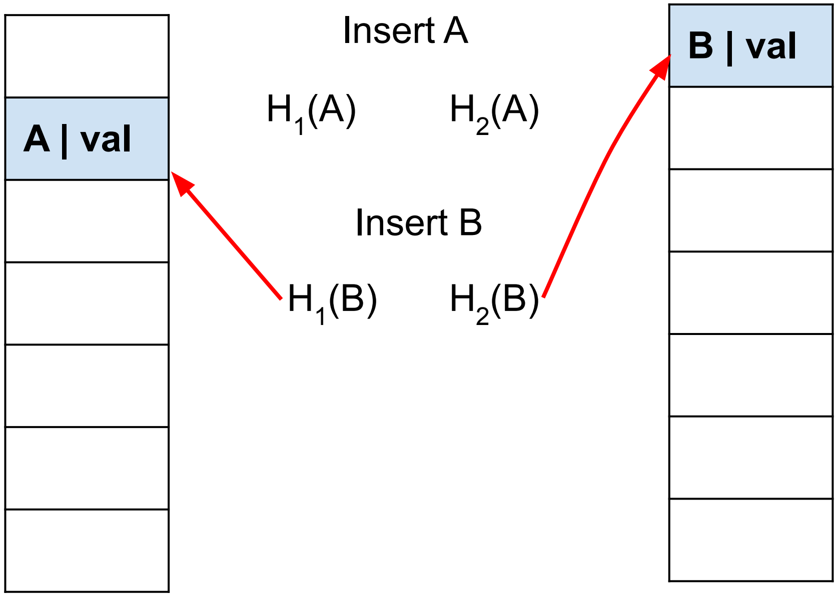
- Problem 2: How to deal with collision
  - Assumption: we know maximum size of the table
  - Closed Hashing
- Cuckoo Hashing:
  - Use multiple tables, and multiple hash functions
  - Insert: check every table, if there's a free slot, insert.
  - If no free slot:
    - Pick one key and rehash with different hash function to find new position
    - Insert to the recently vacated one

Cuckoo!

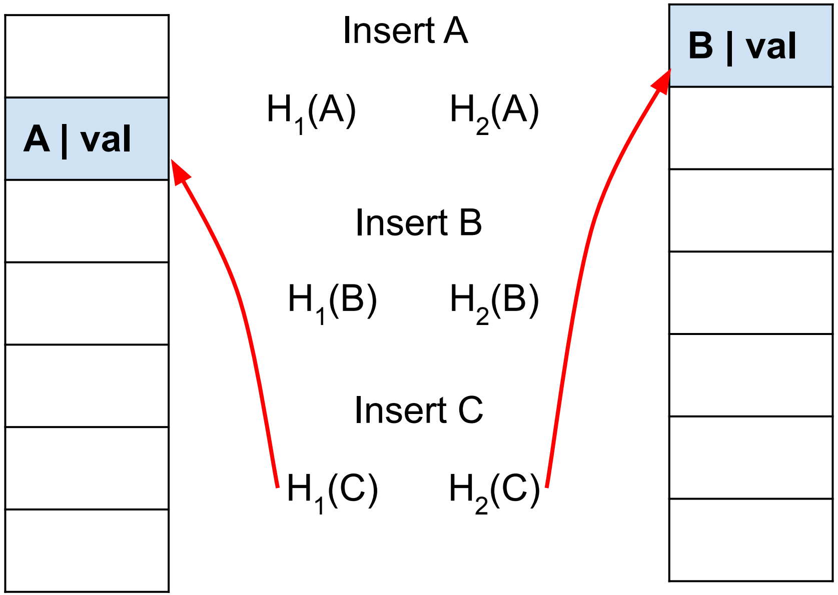
Cuckoo Hashing



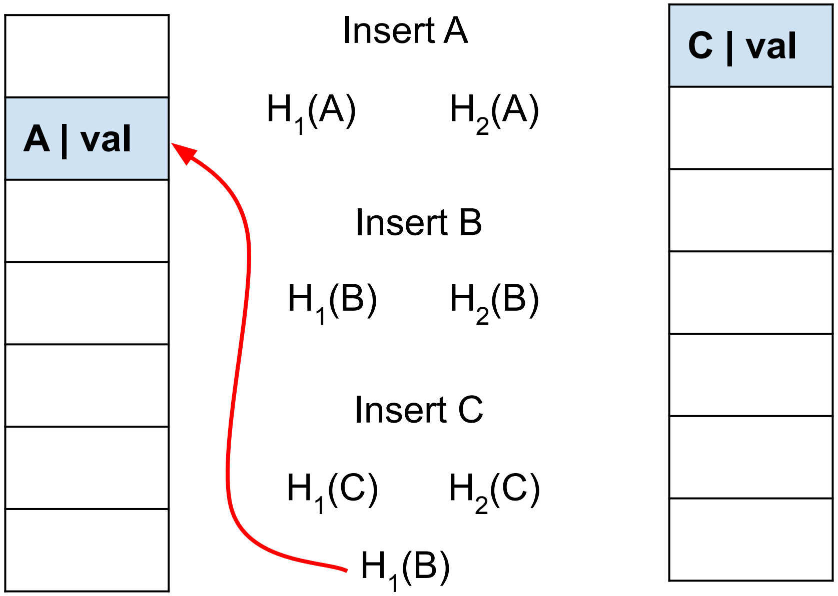
Cuckoo Hashing



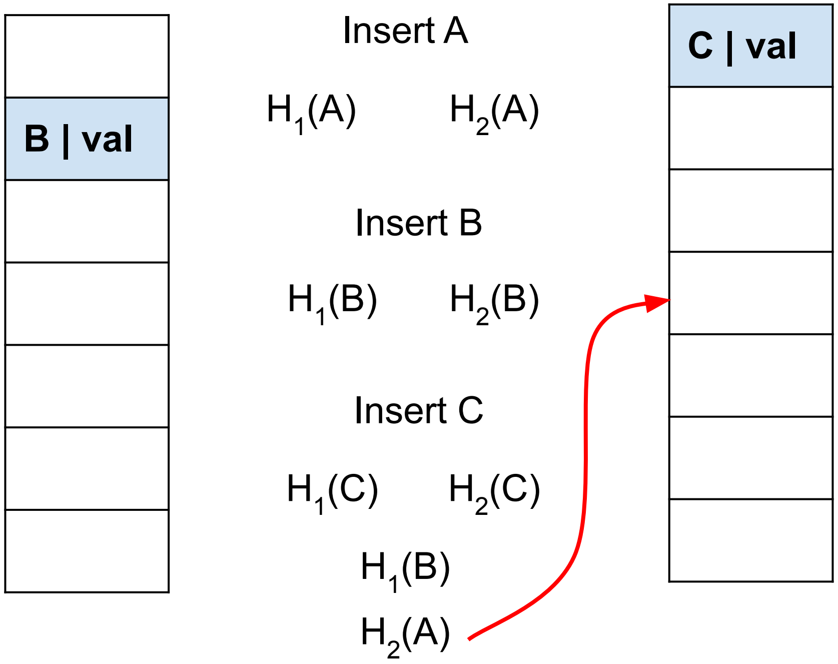
Cuckoo Hashing



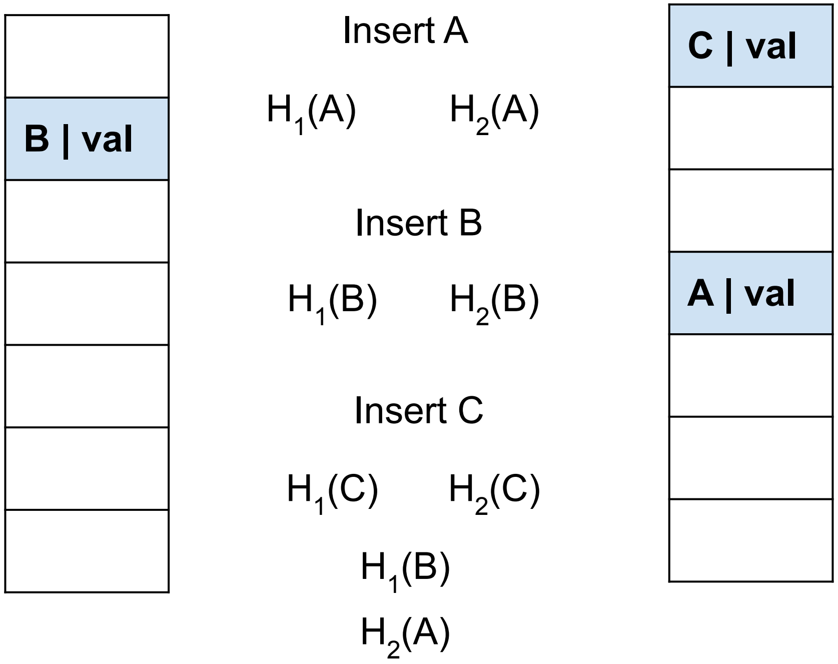
Cuckoo Hashing



Cuckoo Hashing



Cuckoo Hashing

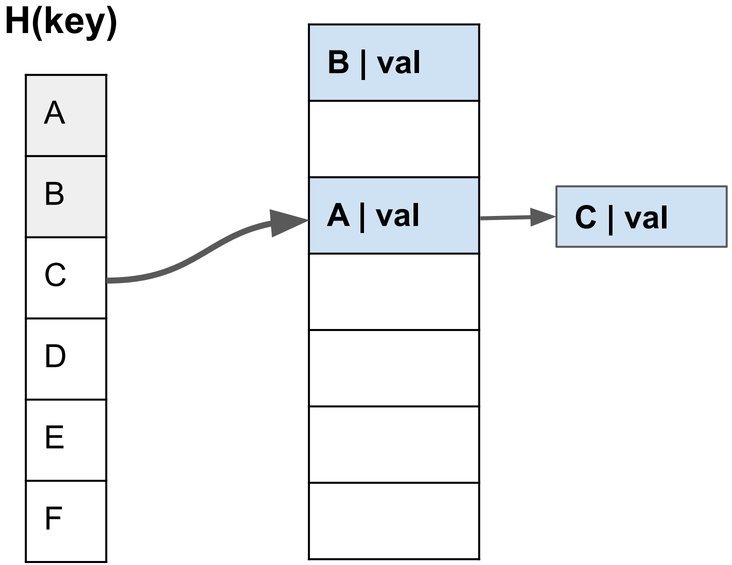


Cuckoo Hashing

- Look up is always O(1)
- Insert may bounce values
- May stuck in an infinite loop
    - If find cycle, rebuild with new hash functions
    - Or add more tables!

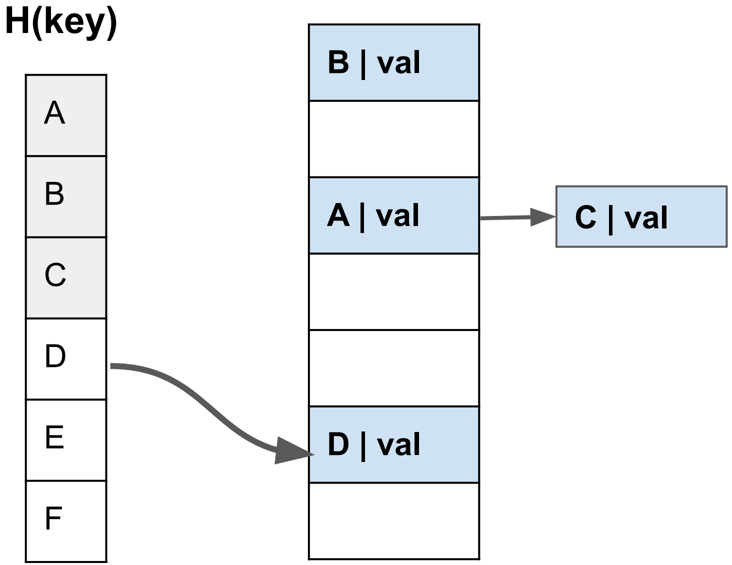
Hash Table

- Problem 2: How to deal with collision
  - Now we don't know # elements
  - Open hashing
- Chained Hashing
  - Table store pointers to linked list
  - Table element = bucket



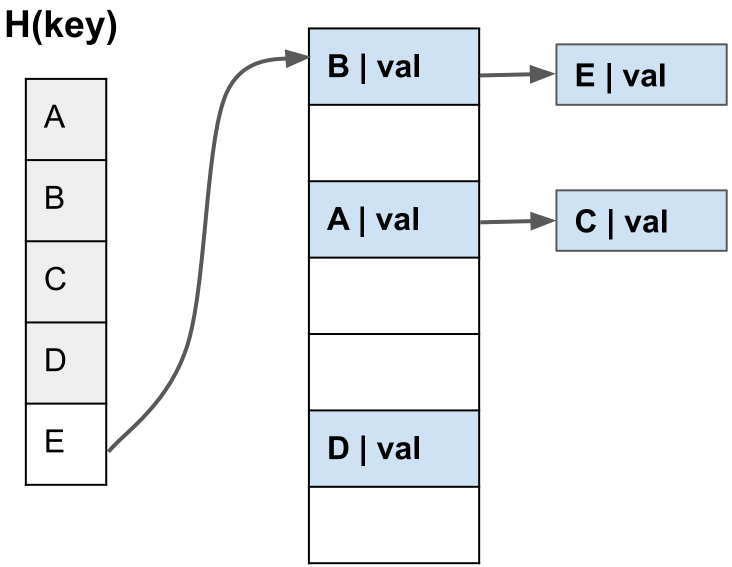
Hash Table

- Problem 2: How to deal with collision
  - Now we don't know # elements
  - Open hashing
- Chained Hashing
  - Table store pointers to linked list
  - Table element = bucket



Hash Table

- Problem 2: How to deal with collision
  - Now we don’t know # elements
  - Open hashing
- Chained Hashing
  - Table store pointers to linked list
  - Table element = *bucket*
  - Bad hash function → long list



https://s3.amazonaws.com/learneroo/visual-algorithms/OpenHash.html

Why B+ Tree

- Disadvantages of Hash Table
  - Worst case is really bad: O(N) lookup for chain-hashing
  - Support only ***point queries***

Point query

```
select * from Payroll where
Job="TA";
```

Range query

```
select * from Payroll where
Salary > 55000;
```

B+ Tree

- You know binary search trees (BST)!
    - They’re called 2-way search tree
    - REFRESH your memory.
- B+ tree:
    - Multi-way search tree
    - Perfectly balanced
    - All inner nodes, except root, is at least half full

The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B⁺-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B⁺-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

might be labeled with the employees’ last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part

“What, if anything, the B stands for has never been established” - Wikipedia

Stores data pointers only at the leaf nodes of tree. Leaf nodes are linked to provide ordered access to record.

B+ Tree

- B+ tree:
  - Multi-way search tree
  - Perfectly balanced
  - All inner nodes, except root, is at least half full
  - **Each node is a page**

Efficient lookup

Good worst case performance

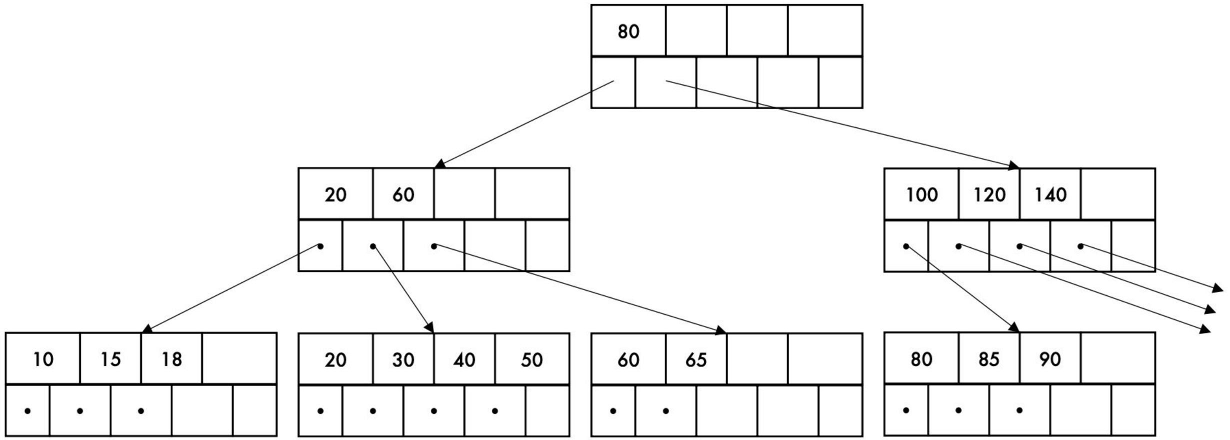
Easy to rebalance

All operations are in O(logN)

B+ Tree

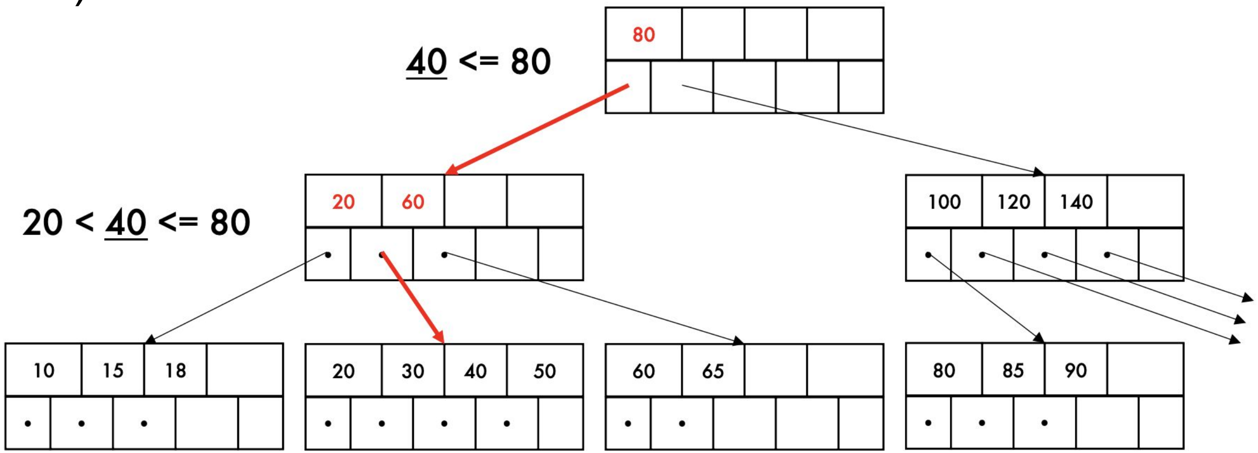
similar to BST except root node has multiple values & multiple pointers

- Example



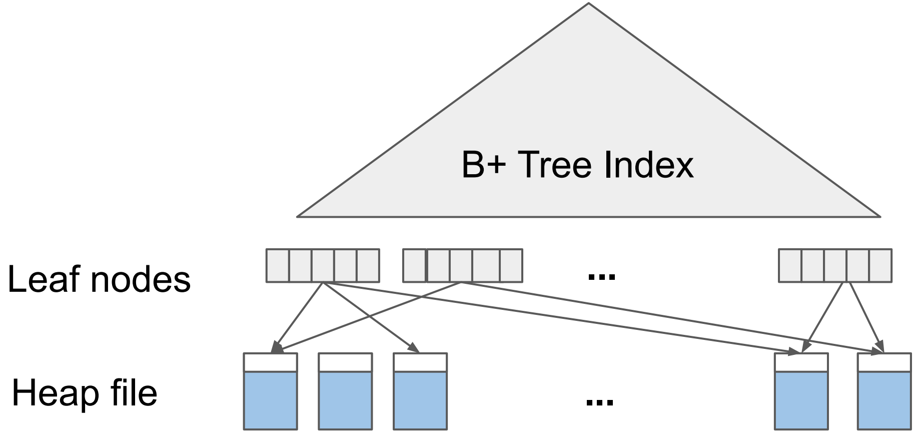
B+ Tree

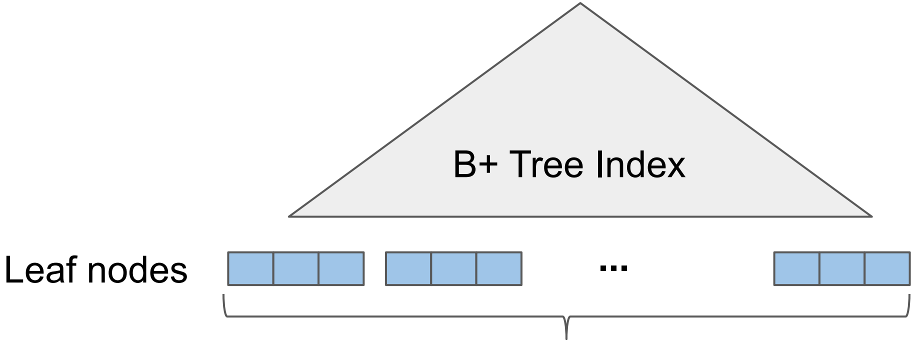
- Find value 40
- Point queries:
  - I/O cost: $(\log_M N + 1)$



B+ Tree

- What's in the leaf?
    - Record ID
    - Or data tuple



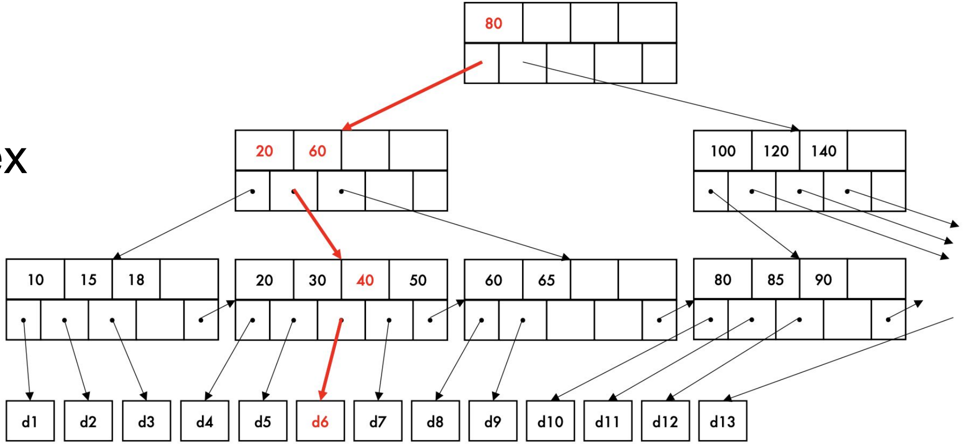


this is sorted

What can you say about these tuples?

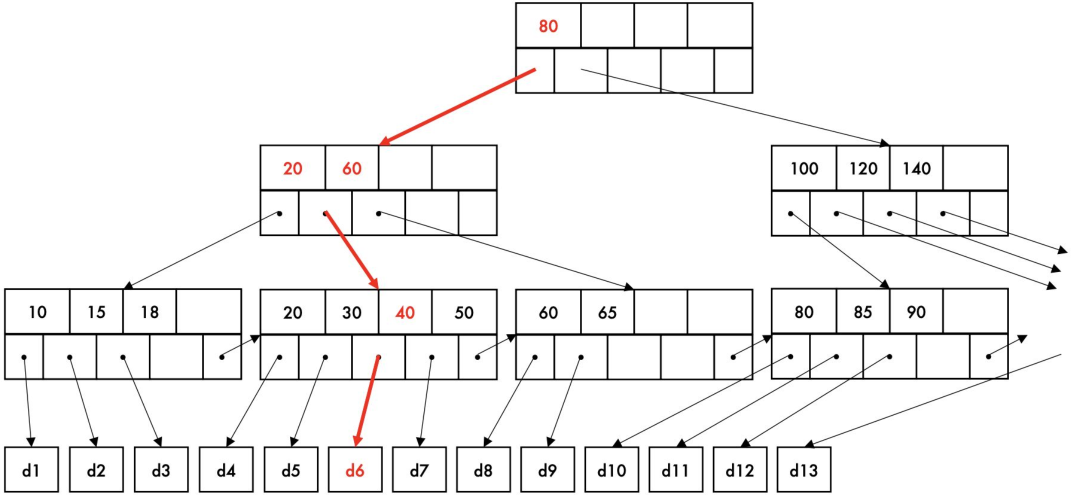
B+ Tree

- Clustered Index
  - Heap file is sorted on the index attribute
  - **Only one per table!**
    - Why?



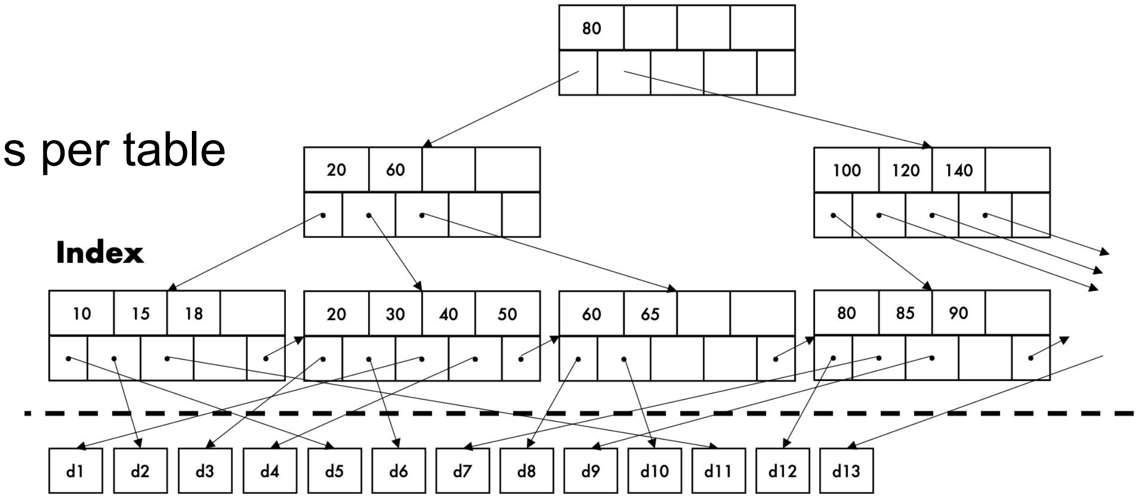
B+ Tree

- What's in the leaf?
  - Or tuple data
  - Called **clustered index**



B+ Tree

- Unclustered index
  - Heap file not sorted on the index attribute
  - Can have many of this per table



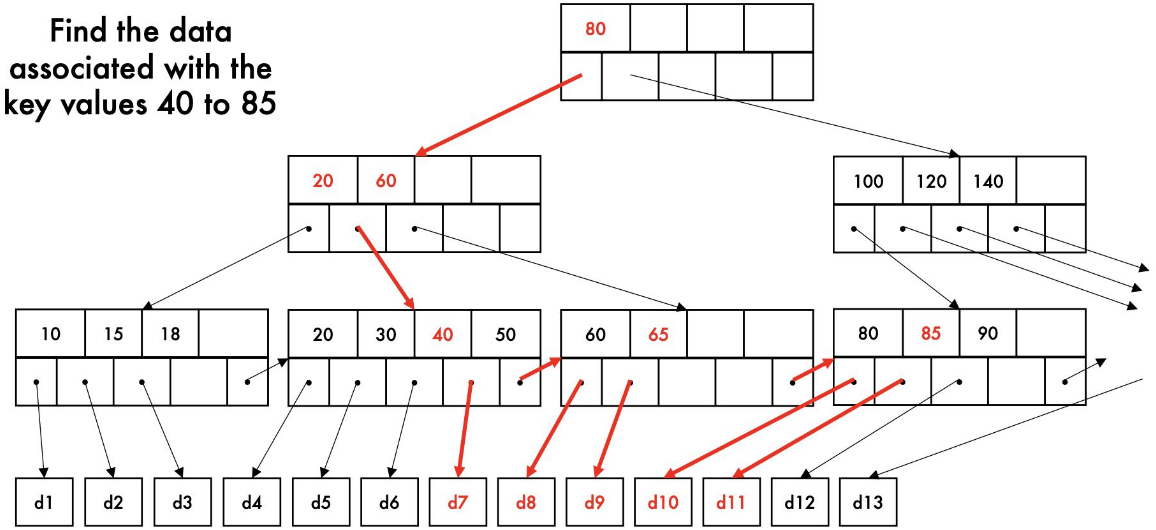
B+ Tree

- Range queries
  - I/O cost: $\log_M N$ + #tuples/(page size)

1 I/O to read each page
M = # values in a node (=5 here)
N = branching factor of the tree
logMN = height of tree

#tuples = what you return (in this case it is 5 (d7 to d11)
page size = # tuples in a page

so #tuples/page size = approx how many I/O to fetch these results



B+ Tree

if full,
- split evenly into two,
put in K in parent node
- do recursively up

- Look up: similar to binary search tree
- Insert/delete: more complex
  - Need to keep the tree balanced

https://cmudb.io/btree

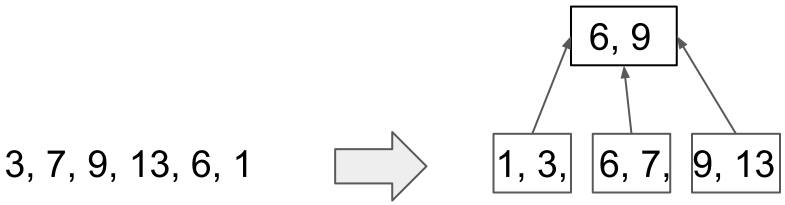
Insert

Find leaf L
Insert to L in sorted order. If L not full, done
If L is full
+ Split to L and L’ evenly at split key K
+ Insert K to the parent node of L (recursive)

B+ Tree

- How to build one
  - Approach 1: insert one by one
  - Approach 2: sort, then build
- You should know which approach is better

approach 2 is better



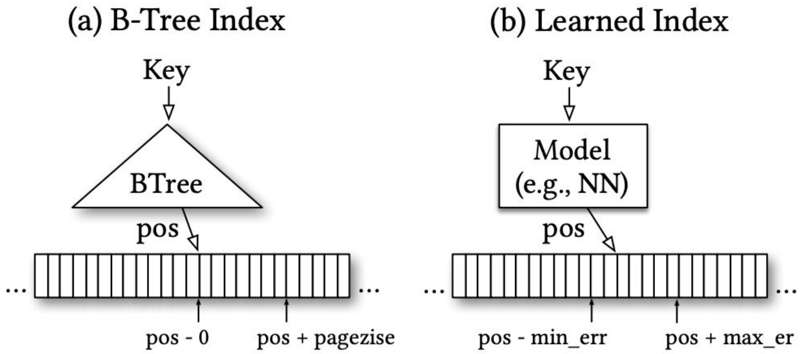
Learned Index

- Recall what an index does:
    - f(x) → recordID
- B+ Tree performance independent of data distribution
    - If we have this mapping
    - B+ Tree still need O(logN) time

x	f(x)
0	0
1	1
2	2
..	..

Learned Index

- Machine-learn function f(.)
  - Faster than B+ tree in **<u>some cases</u>**



The Case for Learned Index Structures

Tim Kraska*
MIT
Cambridge, MA
kraska@mit.edu

Alex Beutel
Google, Inc.
Mountain View, CA
alexbeutel@google.com

Ed H. Chi
Google, Inc.
Mountain View, CA
edchi@google.com

Jeffrey Dean
Google, Inc.
Mountain View, CA
jeff@google.com

Neoklis Polyzotis
Google, Inc.
Mountain View, CA
npolyzotis@google.com

Abstract

Indexes are models: a B-Tree-Index can be seen as a model to map a key to the position of a record within a sorted array, a Hash-Index as a model to map a key to a position of a record within an unsorted array, and a BitMap-Index as a model to indicate if a data record exists or not. In this exploratory research

Summary

- Access methods / index help access data faster
- Hash Tables good point queries,
- B+ Trees good for range queries and have good worst case performance
- Learned Index uses ML model to learn data distribution
- No one-size-fit-all indices