RDBMS and SQL
Physical View and Indexing

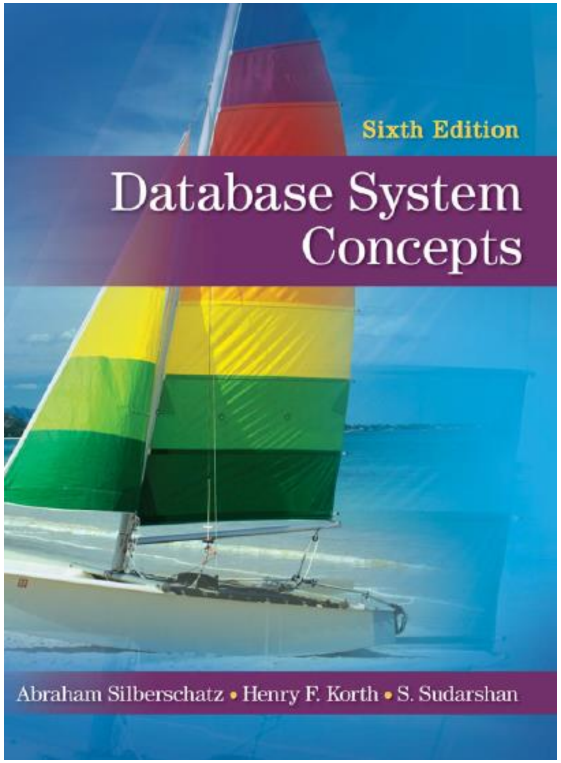
Venkatesh Vinayakarao
venkateshv@cmi.ac.in
http://vvtesh.co.in

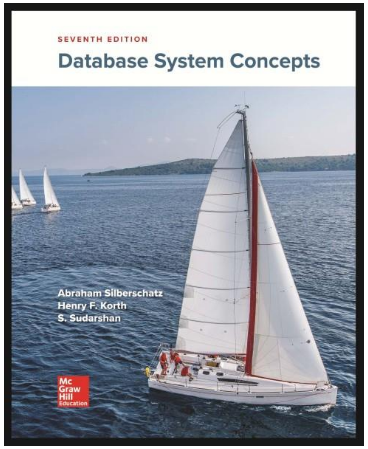
Chennai Mathematical Institute

Some slide contents are borrowed from the course text. For the authors' original version of slides, visit: https://www.db-book.com/db6/slide-dir/index.html.

Course Text

We will follow the…





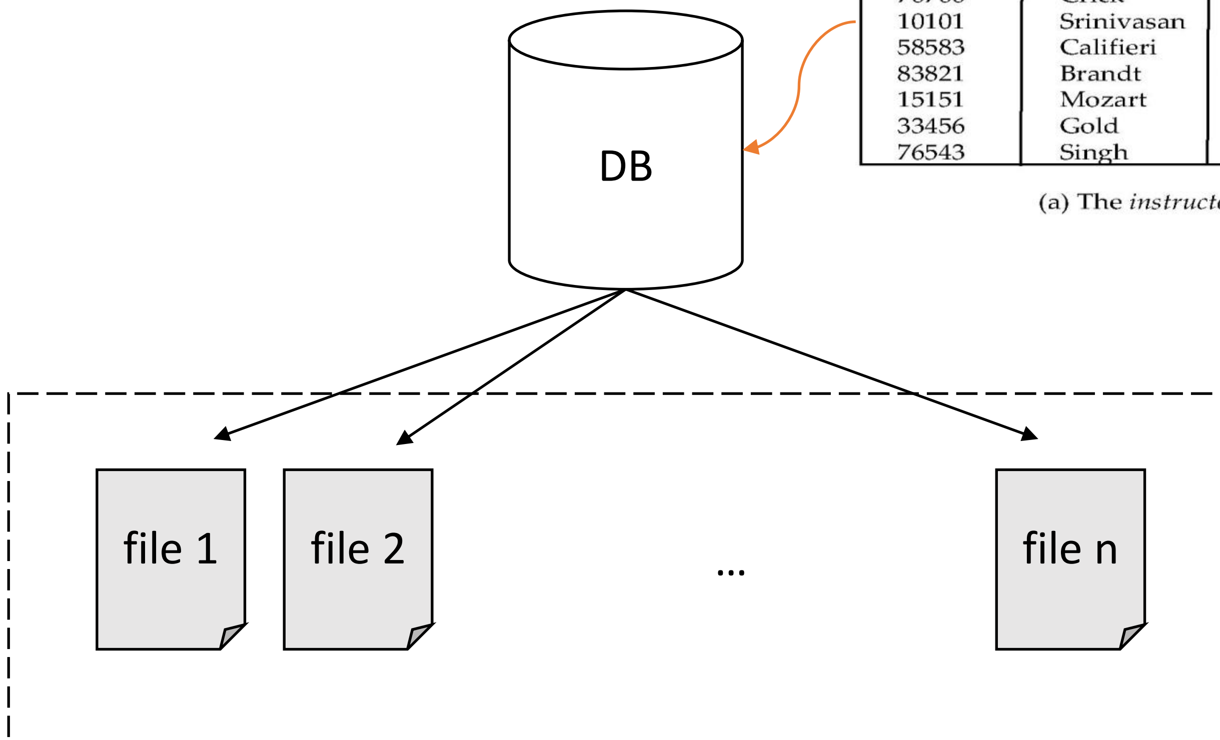
7th Edition Covers
Big Data, Block Chain,
Distributed Comptuing…

https://www.db-book.com/db6/index.html

File Organization

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

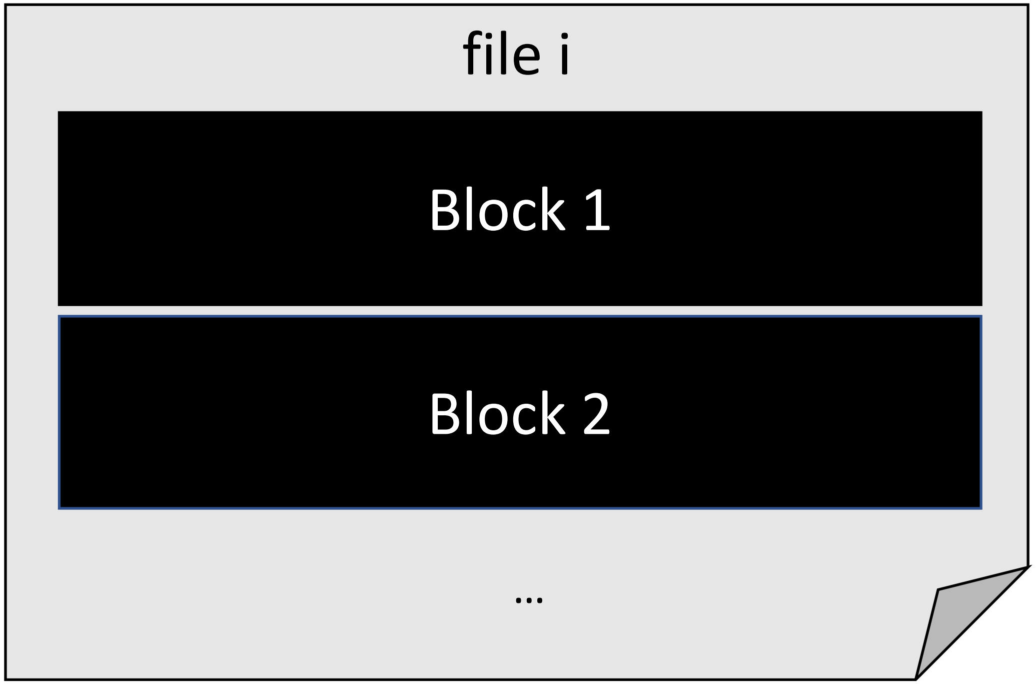
(a) The *instructor* table



Data stored as files.
Files are managed by the underlying OS.

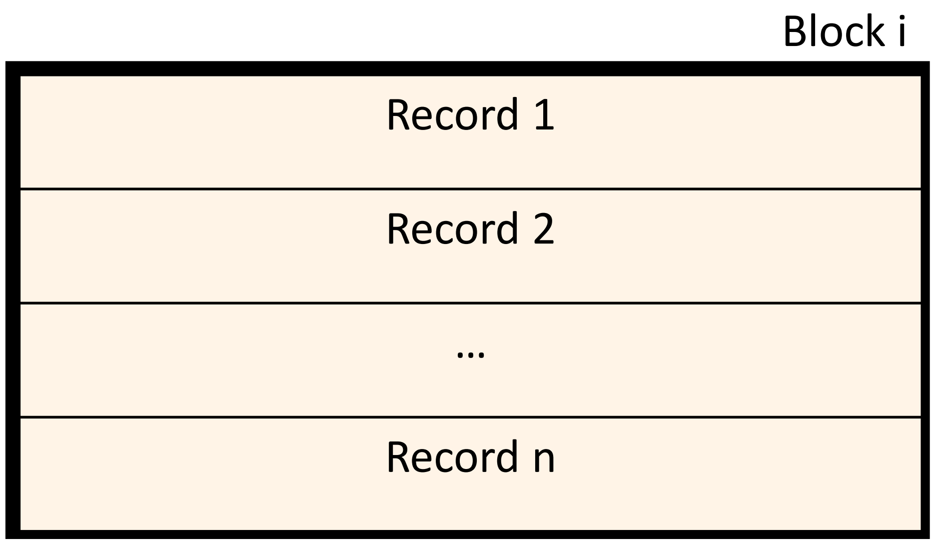
Files

- A **file** is a sequence of **blocks**.
- **Blocks** are fixed-length units of both storage allocation and data transfer.

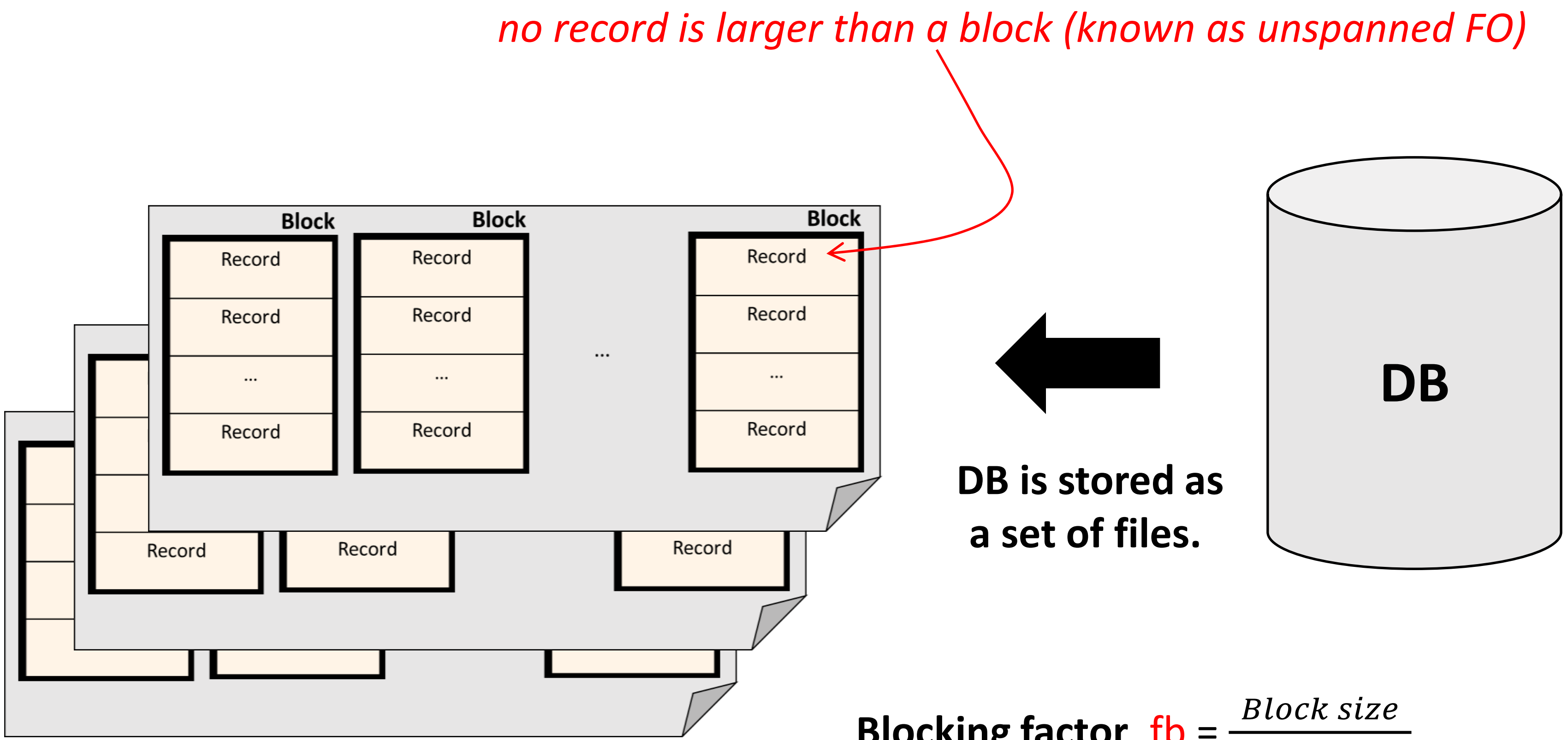


Records

- A **block** may contain several **records.**
- Each **record** is entirely contained in a single **block**.



File Organization (FO)



DB is stored as a set of files.

Blocking factor, fb = $\frac{Block\ size}{Record\ size}$

We have fb records per block.

For "SELECT" Queries: How to retrieve the blocks containing our data records with minimal disk access?

For "INSERT/DELETE" Queries: Can we ensure disk space is wisely used?

And more …

Sequential (Unspanned) File Organization with Fixed Length Records

A Simple Approach

Fixed Length Records =
Each record is made up of its fields whose length is fixed.

type *instructor* = record
  ID varchar (5);
  name varchar(20);
  dept_name varchar (20);
  salary numeric (8,2);
end

Sequential file organization =
Records are stored in sequential order (of key).

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Quiz

Assume each

- char takes 1 byte and
- numeric(8,2) type take 8 bytes

of physical storage. Say, block size in our file system is 1 KB. If there are 20 records in our relation, how many block accesses will we need to retrieve all of them?

type *instructor* = **record**
ID **varchar** (5);
name **varchar**(20);
dept_name **varchar** (20);
salary **numeric** (8,2);
end

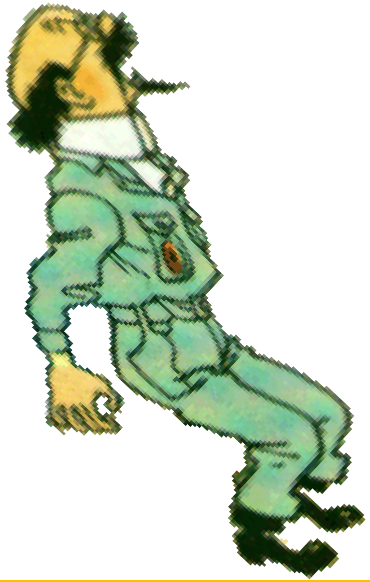
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

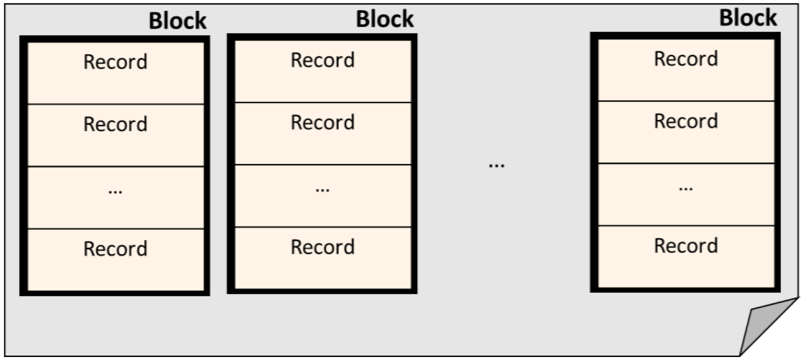
Quiz

- Assume each char takes 1 byte and numeric(8,2) type take 8 bytes of physical storage. Say, block size in our file system is 1 KB. If there are 20 records in our relation, how many block accesses will we need to retrieve all of them?
  - Record length = 53 bytes
  - Total no. of records = 20
  - Space required = 53 * 20 = 1060 bytes
  - Block size = 1024 bytes.
  - We need two block accesses to retrieve all records.

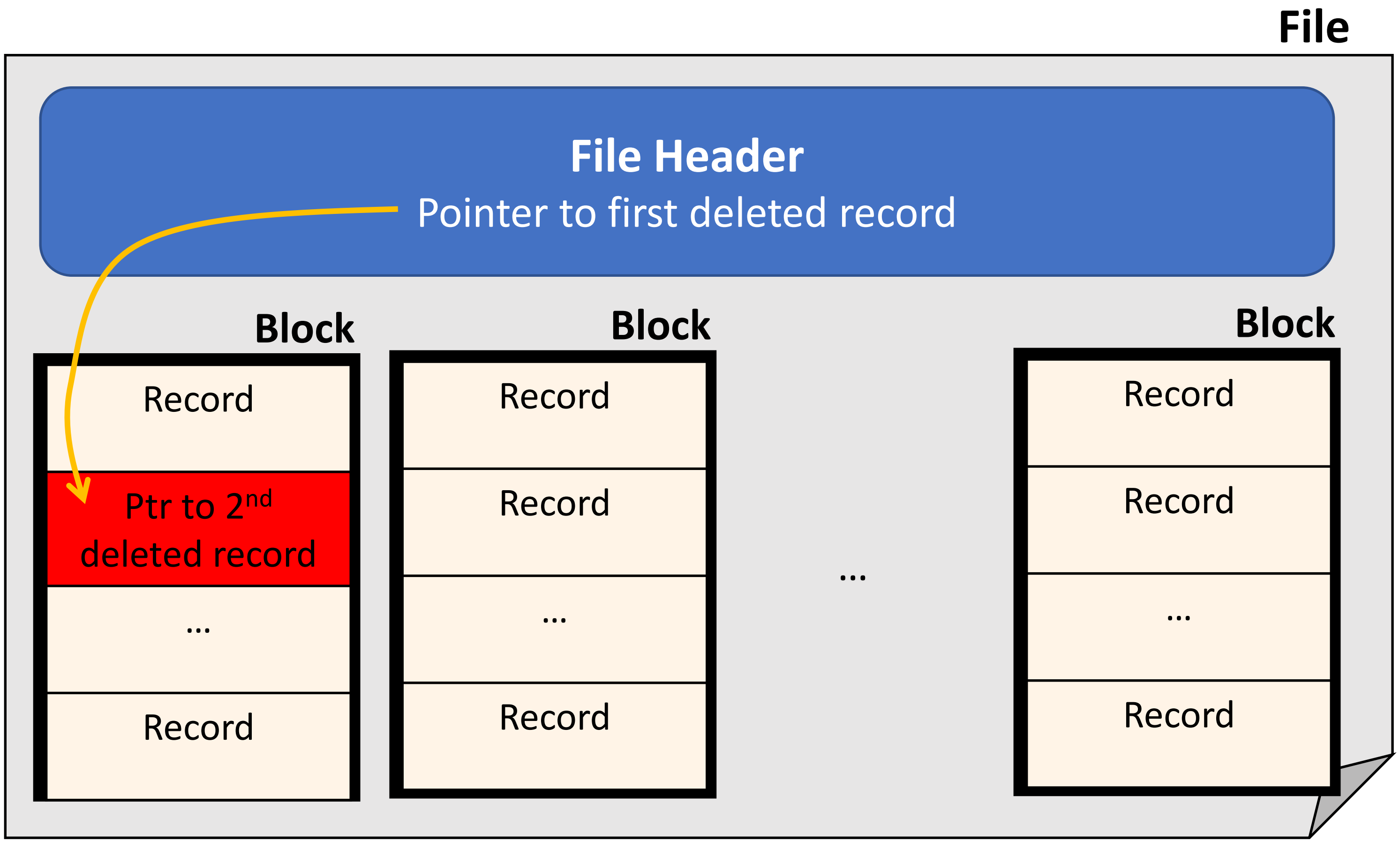
Issues

- Deletion
  - Causes gaps inside blocks.
- Space optimization
  - block size may not be a multiple of record length
    - space wasted in blocks.





Space Usage



Deleted records form a linked list called the "free list".

Free List

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Free list

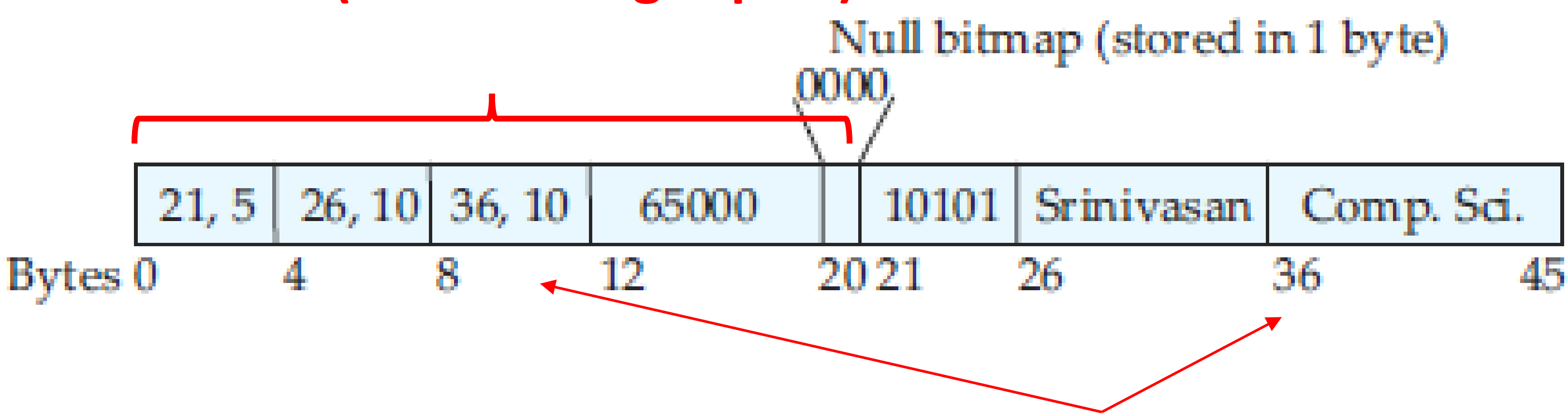
1 → 4 → 6

We now have a way to handle deletion.

How to optimize for space (block size not being a multiple of record length?

Variable Length Record

Metadata about the variable length data is stored (in fixed length part)



Read 10 bytes from 36th byte for this field
Null bitmap shows which fields are null.

Does the order in which records are stored matter?

Yes, we usually search with the primary key!

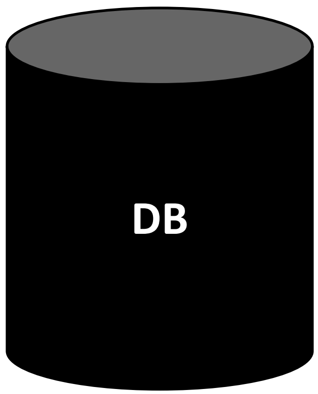
So, save records in the order of primary key.

Storage Organization of Records

- **Heap file organization**
  - Place any record anywhere in the file.
- **Sequential file organization**
  - Records are stored in sequential order (of key).
- **Hashing file organization**
  - Hash (some attribute of) records to blocks.

Motivation

- We usually access only a small part of the DB.

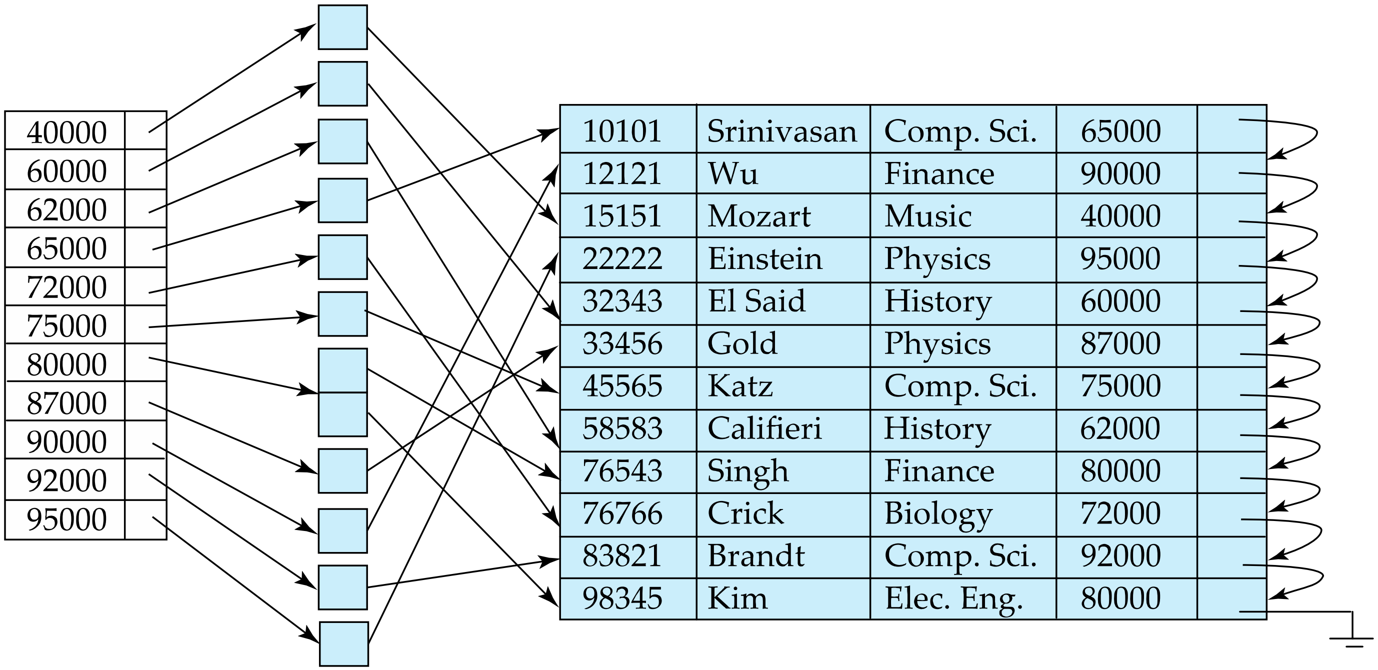


Find the instructors in the physics department

Need additional structures to access data efficiently

An Index to Speed Up SELECT queries

- Indices different from the order in which the records are stored in the disk are called Secondary indices.



Secondary index on *salary* field of *instructor*

Creating an Index in MySQL

```
CREATE INDEX
name_of_the_index
ON customer (name(10));
```

- Creates an index using the first 10 characters of the name column.
- Column prefixes for indexes can make the index file much smaller.
- Speeds up INSERTs.

https://dev.mysql.com/doc/refman/8.0/en/create-index.html

For "SELECT" Queries: How to retrieve the blocks containing our data records with minimal disk access?

For "INSERT/DELETE" Queries: Can we ensure disk space is wisely used?

Now, we have answers!

1. Hashing, Sequential, Heap FO
2. Free Lists, Variable Length Records
3. Secondary Index

Can we improve indexing to save some space?

Sparse Index

- **Sparse Index**: contains index records for only some search-key values.

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Dense Index

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

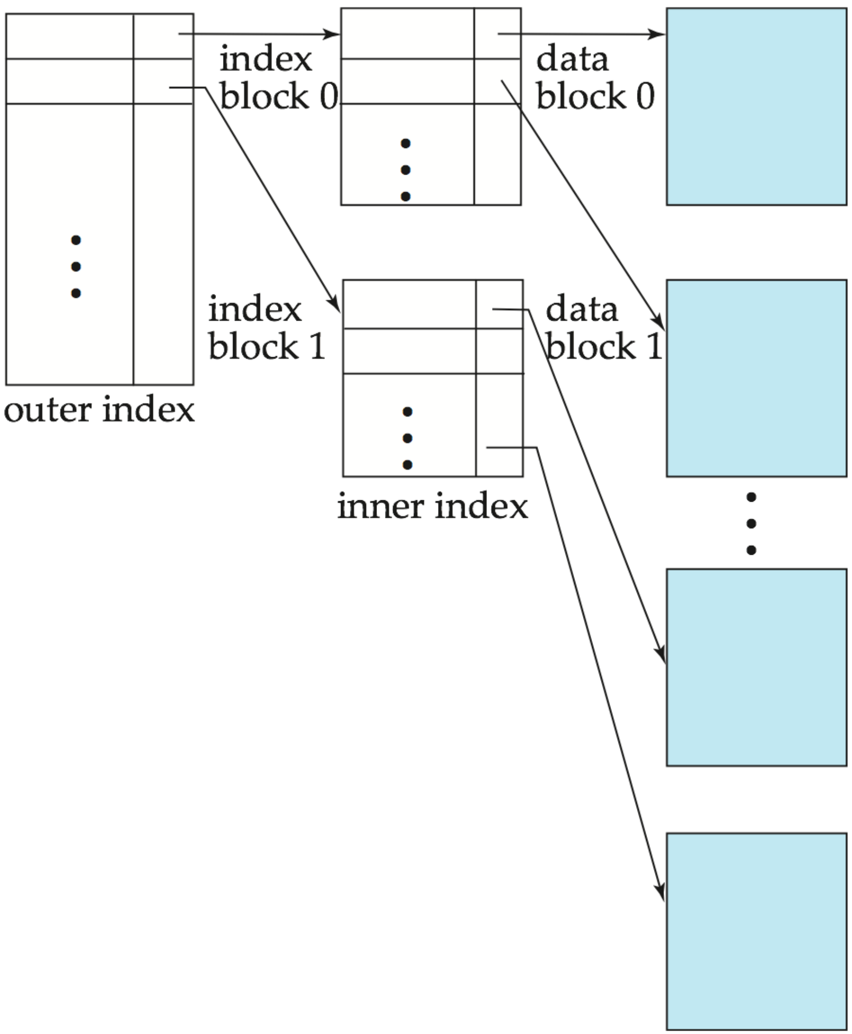
Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Can we improve indexing for faster search?

Multilevel Index

- Indexing the index!



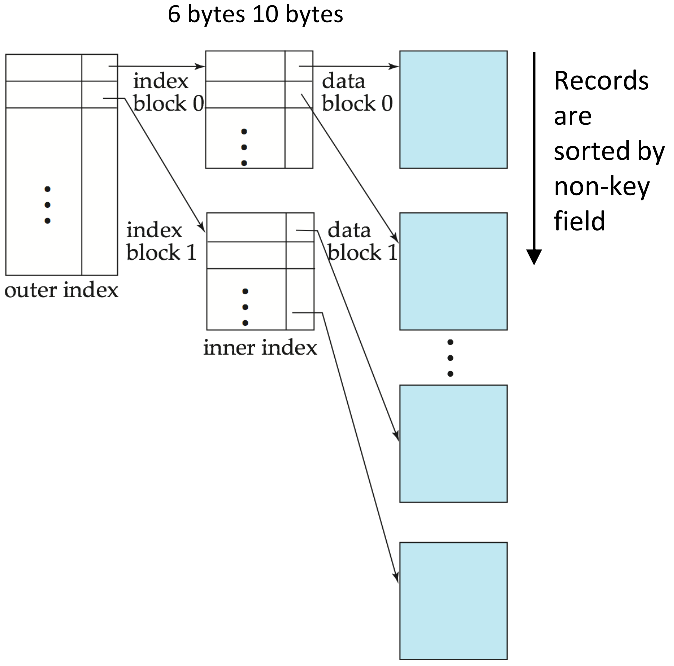
GATE CS 2008

- Consider a file of 16384 records. Each record is 32 bytes long and its key field is of size 6 bytes. The file is ordered on a non-key field, and the file organization is unspanned. The file is stored in a file system with block size 1024 bytes, and the size of a block pointer is 10 bytes. If the secondary index is built on the key field of the file, and a multi-level index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multi-level index are respectively _______ .

GATE CS 2008

- The file organization is unspanned.
- Number of Records = 16384
- Record Size = 32 bytes
- Block Size = 1024 bytes

Search Key 6 bytes	Pointer 10 bytes

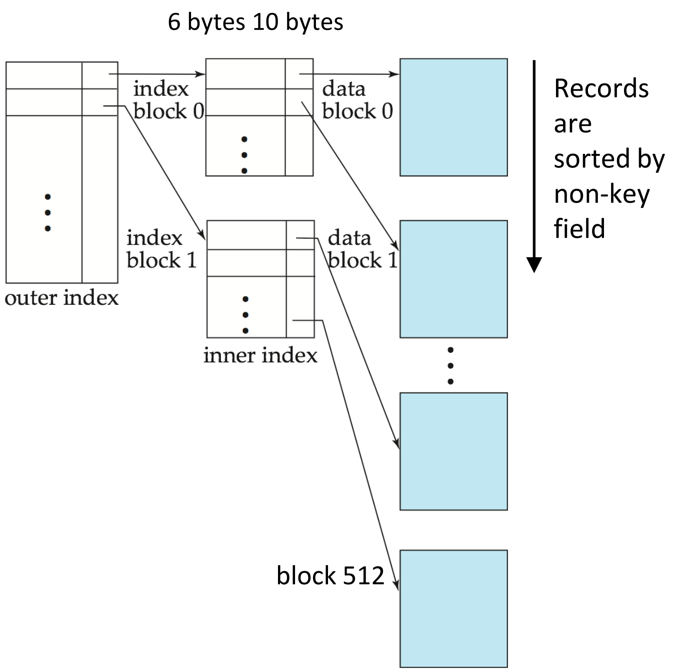


GATE CS 2008

- How many records per block?

1024/32 = 32 records per block

Block 1 (size = 1024 bytes)
Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)



GATE CS 2008

- How many data blocks do we need if we have 16384 records?

Block 1

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
...
...
Record 32 (32 bytes)

Block 2

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
...
...
Record 32 (32 bytes)

...

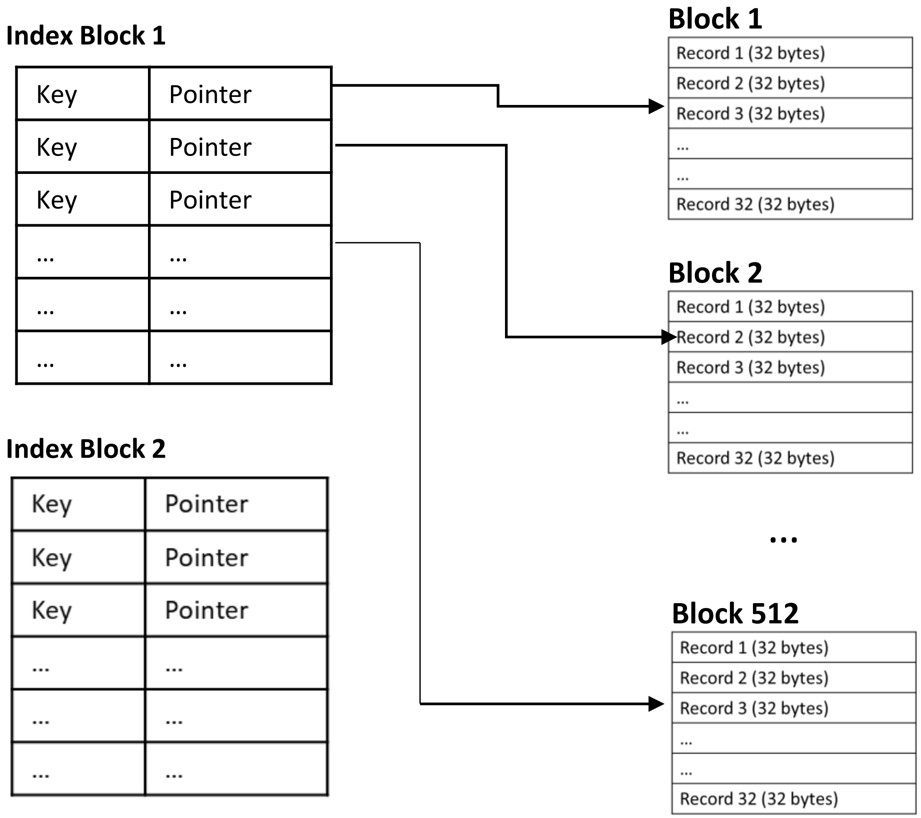
Block n

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
...
...
Record 32 (32 bytes)

16384/32
= **512 blocks**

GATE CS 2008

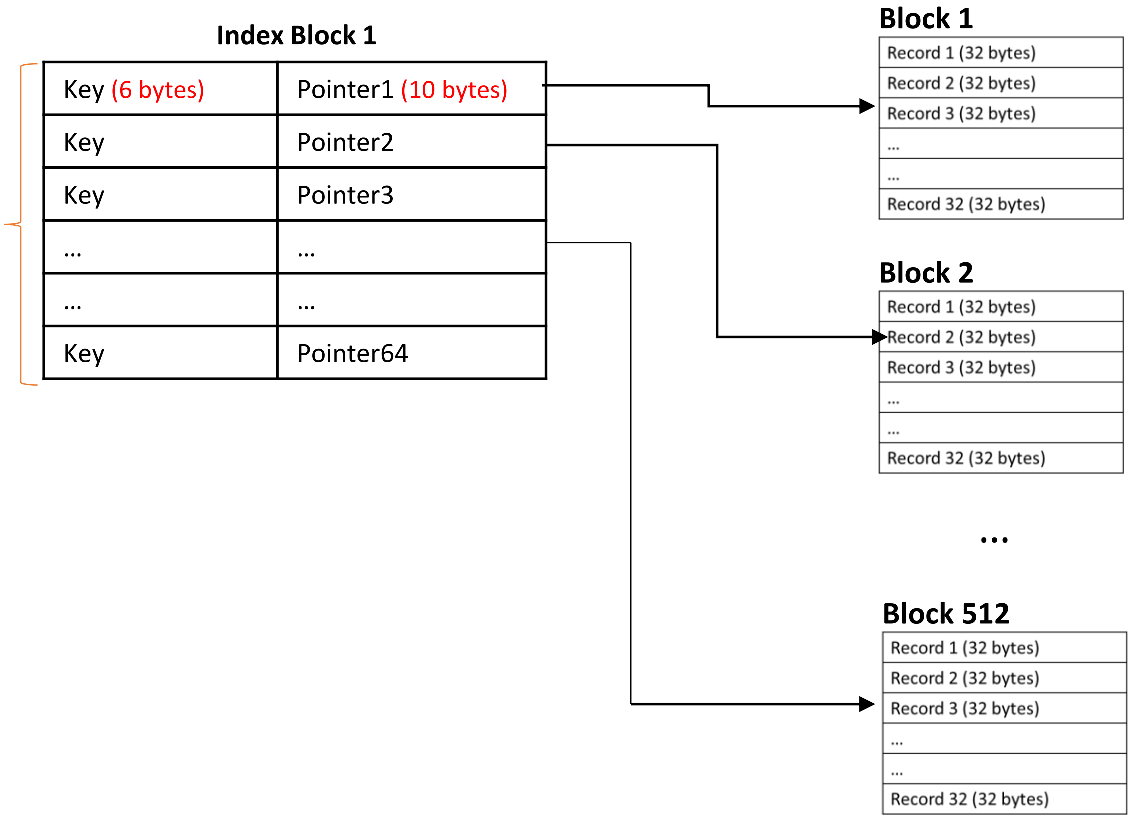
- How does the index block look like?



GATE CS 2008

- How many index blocks exist?

So, 1024/16
= 64 records



GATE CS 2008

- How many levels of index do we need?

Index Block 1

Key (6 bytes)	Pointer1 (10 bytes)
Key	Pointer2
Key	Pointer3
…	…
…	…
Key	Pointer64

Index Block 2

Key (6 bytes)	Pointer65 (10 bytes)
Key	Pointer66
Key	Pointer67
…	…
…	…
Key	Pointer128

Block 1

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

Block 2

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

…

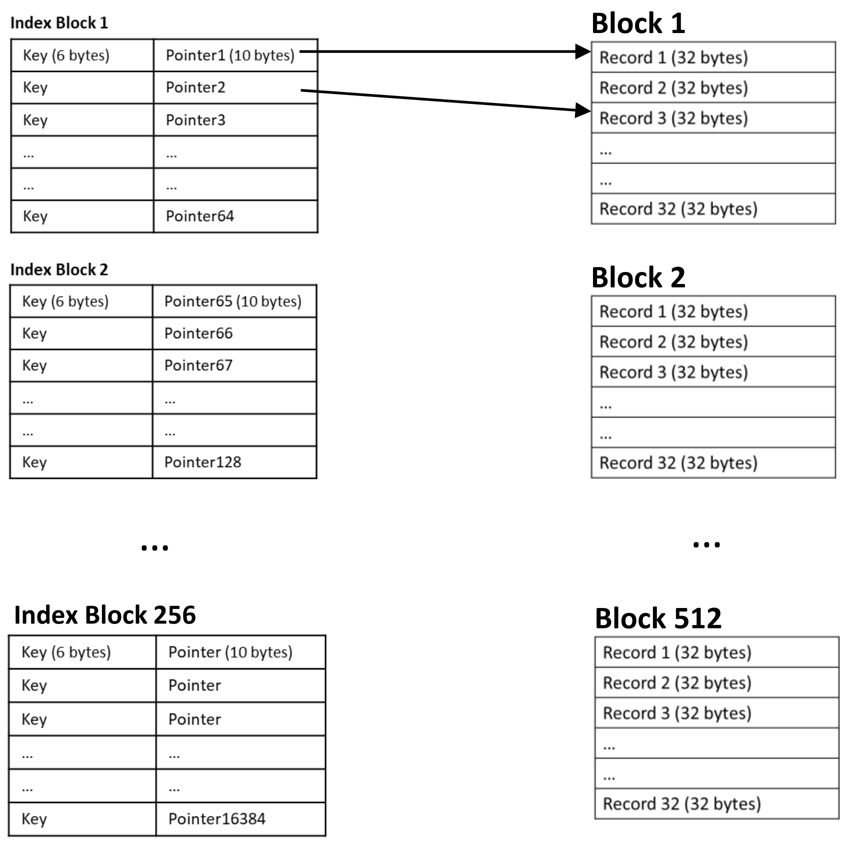
Block 512

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

Totally, 16384/64 = 256 index blocks exist.

GATE CS 2008

- How many levels of index do we need?



GATE CS 2008

- How many levels of index do we need?

Outer Index Block 1

Key (6 bytes)	Pointer1 (10 bytes)
Key	Pointer2
Key	Pointer3
…	…
…	…
Key	Pointer64

Need 4 such blocks
to hold 256 entries

Index Block 1

Key (6 bytes)	Pointer1 (10 bytes)
Key	Pointer2
Key	Pointer3
…	…
…	…
Key	Pointer64

Index Block 2

Key (6 bytes)	Pointer65 (10 bytes)
Key	Pointer66
Key	Pointer67
…	…
…	…
Key	Pointer128

…

Index Block 256

Key (6 bytes)	Pointer (10 bytes)
Key	Pointer
Key	Pointer
…	…
…	…
Key	Pointer16384

Block 1

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

Block 2

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

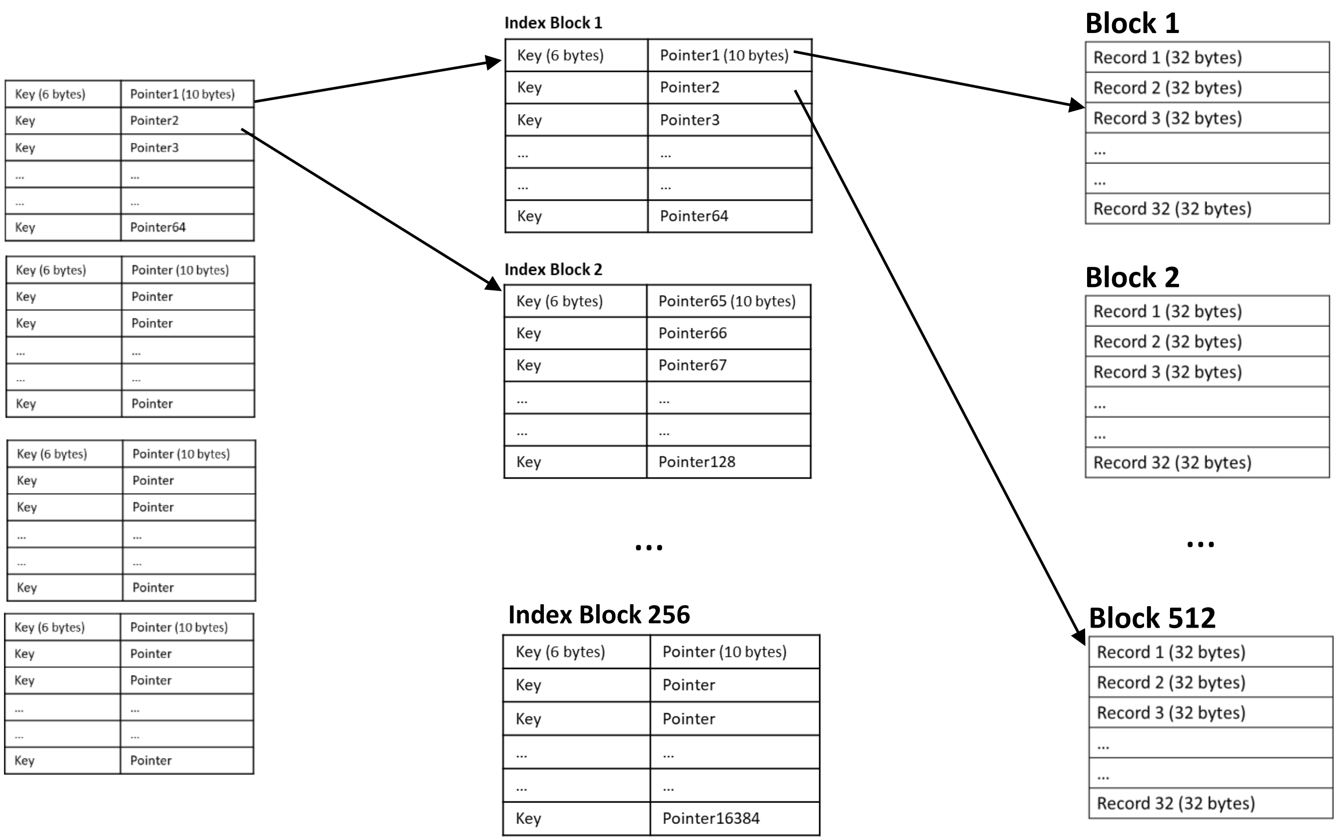
…

Block 512

Record 1 (32 bytes)
Record 2 (32 bytes)
Record 3 (32 bytes)
…
…
Record 32 (32 bytes)

GATE CS 2008

- How many levels of index do we need?



GATE CS 2008

- Consider a file of 16384 records. Each record is 32 bytes long and its key field is of size 6 bytes. The file is ordered on a non-key field, and the file organization is unspanned. The file is stored in a file system with block size 1024 bytes, and the size of a block pointer is 10 bytes. If the secondary index is built on the key field of the file, and a multi-level index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multi-level index are respectively _______
- **Answer: (256, 4)**

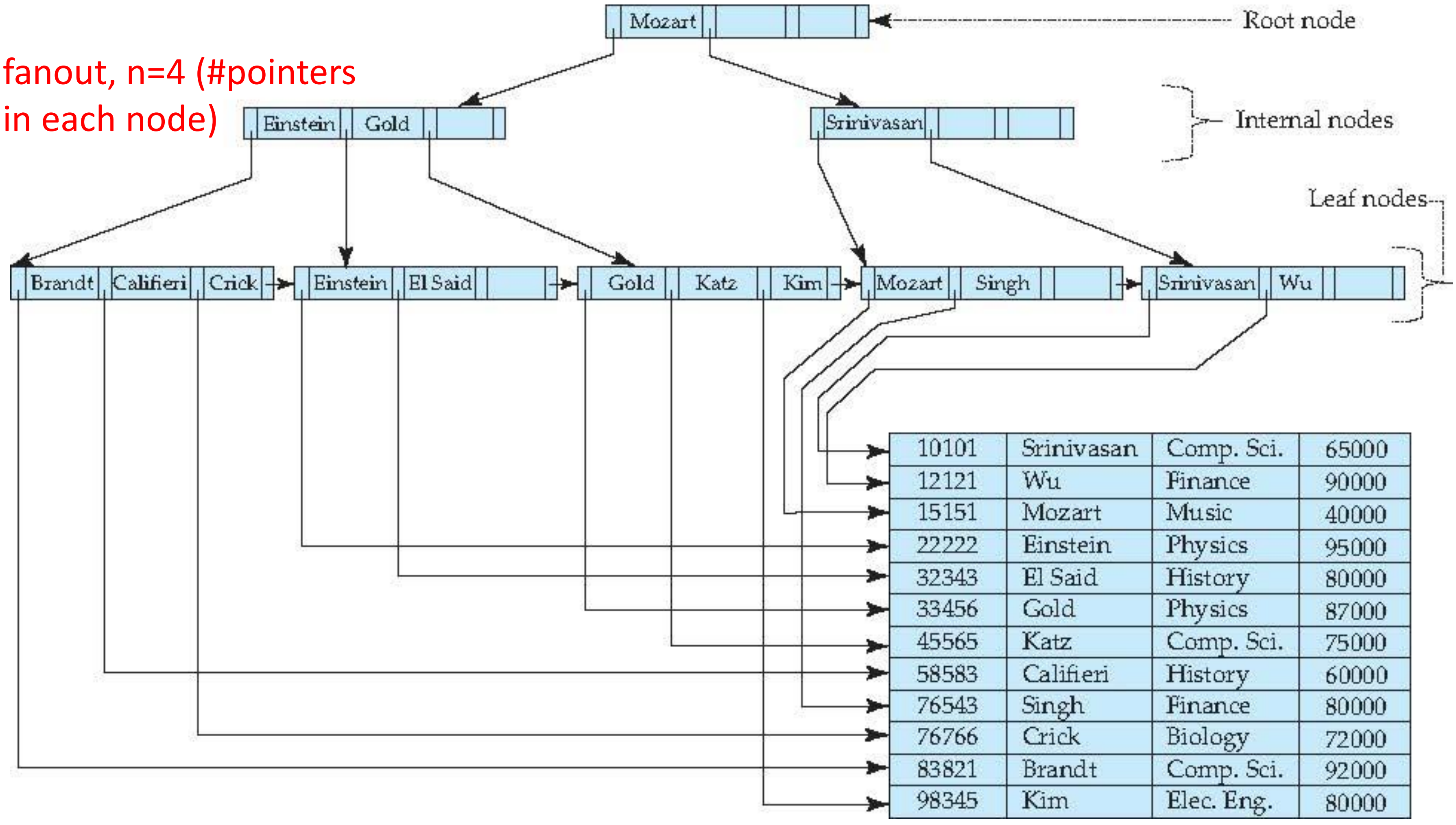
Inserting and deleting entries of the multi-level index is expensive. Can you improve it?

B^+-Tree Index Files

- B+-tree indices are an alternative to indexed-sequential files.
- Advantage of B+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
  - extra insertion and deletion overhead, space overhead.

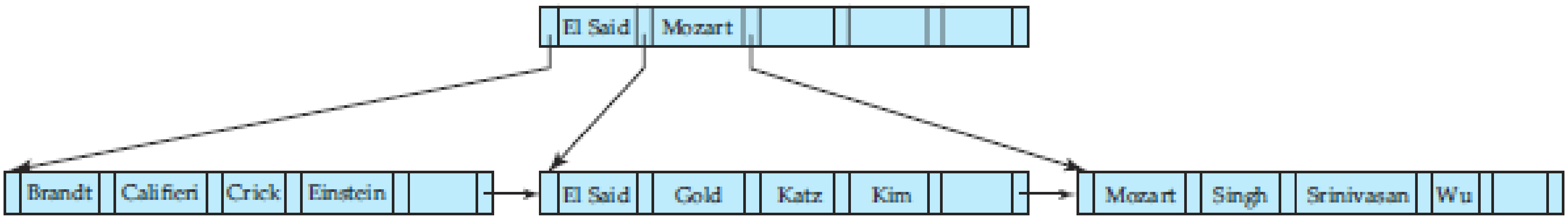
Example of B$^+$-Tree

fanout, n=4 (#pointers in each node)



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

n=6

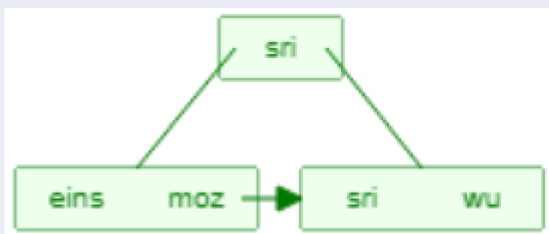
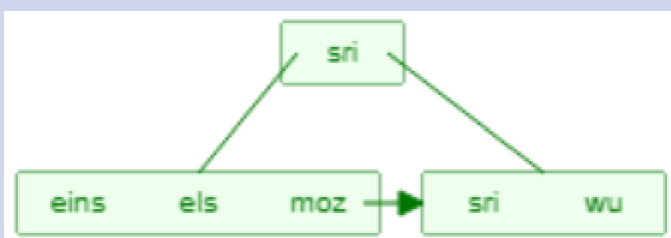


El Said
Mozart
Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinivasan
Wu

Rules

- Root node
  - can hold fewer than $n/2$ pointers.
  - must hold at least two pointers, unless the tree consists of only one node.
- Internal nodes
  - all pointers are pointers to tree nodes.
  - and *must* hold at least $\lceil n/2 \rceil$ pointers and up to n pointers.
- Leaf nodes
  - Can contain from as few as $\lceil (n-1)/2 \rceil$ values, up to n-1 values

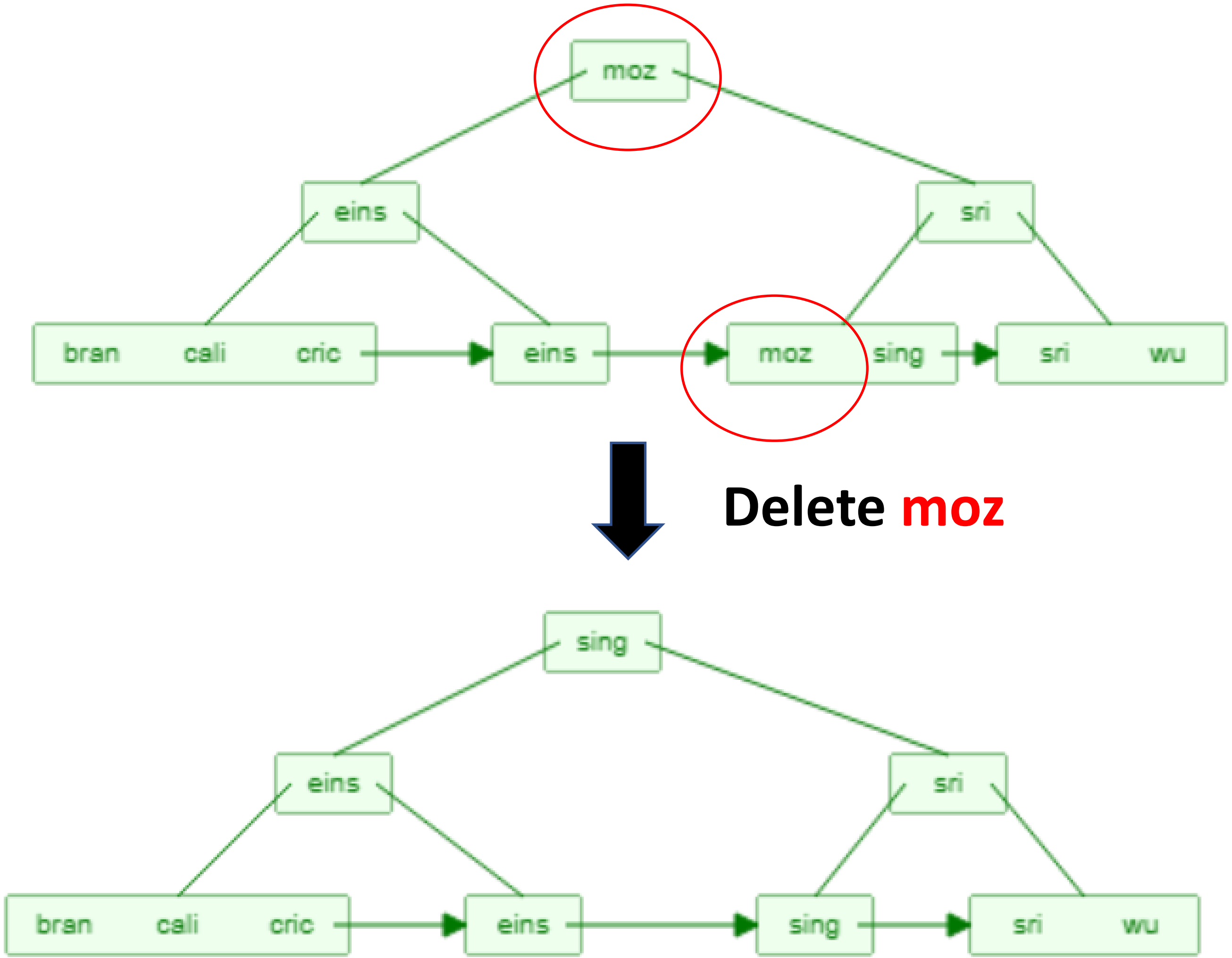
B+ Tree Construction

Insert	B+ Tree
sri	
wu	
moz	
ein	
els	

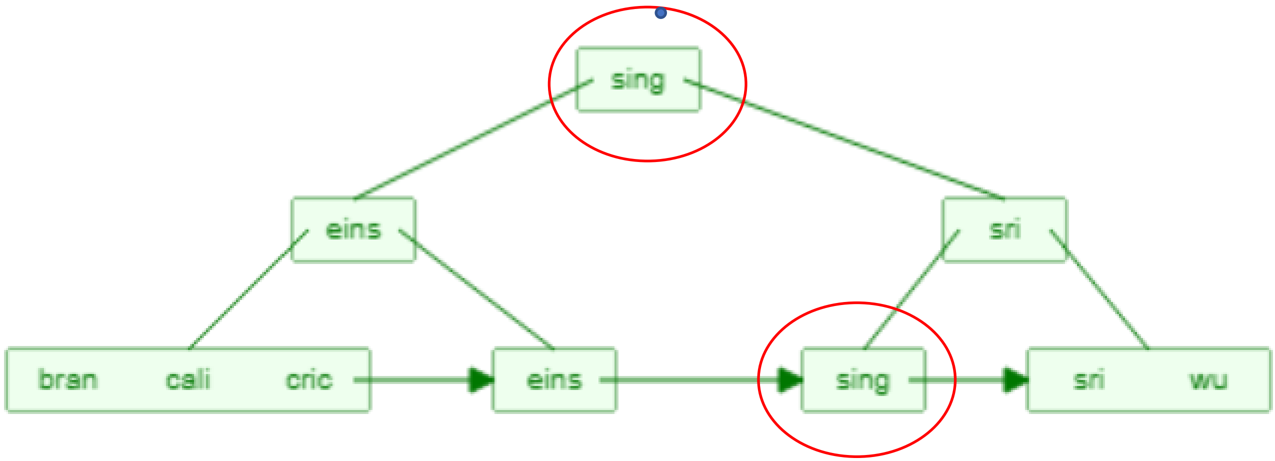
10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

See https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

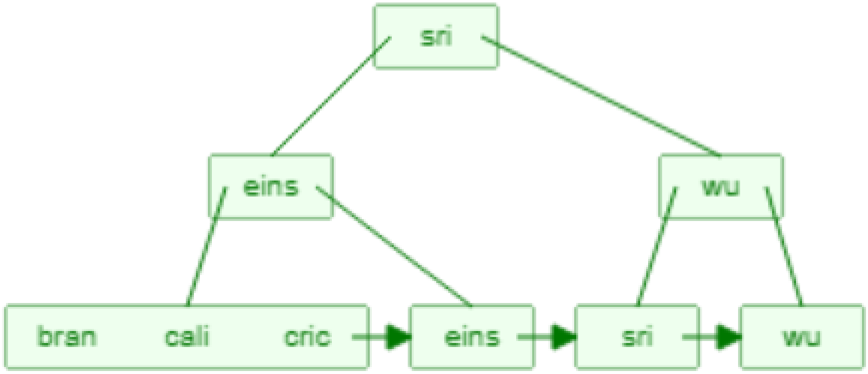
Deletion of a Key in a B+ Tree



Deletion of a Key in a B+ Tree



Delete sing



Readings

- Insert and Delete algorithms over B+Trees

procedure delete(*value* K, *pointer* P)
find the leaf node L that contains (K, P)
delete_entry(L, K, P)

procedure delete_entry(*node* N, *value* K, *pointer* P)
delete (K, P) from N
if (N is the root **and** N has only one remaining child)
then make the child of N the new root of the tree and delete N
else if (N has too few values/pointers) **then begin**
Let N' be the previous or next child of *parent*(N)
Let K' be the value between pointers N and N' in *parent*(N)
if (entries in N and N' can fit in a single node)
then begin /* Coalesce nodes */
if (N is a predecessor of N') **then** swap_variables(N, N')
if (N is not a leaf)
then append K' and all pointers and values in N to N'
else append all (K_i, P_i) pairs in N to N'; set $N'.P_n = N.P_n$
delete_entry(*parent*(N), K', N); delete node N
end
else begin /* Redistribution: borrow an entry from N' */
if (N' is a predecessor of N) **then begin**
if (N is a nonleaf node) **then begin**
let m be such that $N'.P_m$ is the last pointer in N'
remove $(N'.K_{m-1}, N'.P_m)$ from N'
insert $(N'.P_m, K')$ as the first pointer and value in N,
by shifting other pointers and values right
replace K' in *parent*(N) by $N'.K_{m-1}$
end
else begin
let m be such that $(N'.P_m, N'.K_m)$ is the last pointer/value

procedure insert(*value* K, *pointer* P)
if (tree is empty) create an empty leaf node L, which is also the r
else Find the leaf node L that should contain key value K
if (L has less than $n-1$ key values)
then insert_in_leaf (L, K, P)
else begin /* L has $n-1$ key values already, split it */
Create node L'
Copy $L.P_1 \ldots L.K_{n-1}$ to a block of memory T that can
hold n (pointer, key-value) pairs
insert_in_leaf (T, K, P)
Set $L'.P_n = L.P_n$; Set $L.P_n = L'$
Erase $L.P_1$ through $L.K_{n-1}$ from L
Copy $T.P_1$ through $T.K_{\lceil n/2 \rceil}$ from T into L starting at L.
Copy $T.P_{\lceil n/2 \rceil + 1}$ through $T.K_n$ from T into L' starting at
Let K' be the smallest key-value in L'
insert_in_parent(L, K', L')
end

ocedure insert_in_leaf (*node* L, *value* K, *pointer* P)
if ($K < L.K_1$)
then insert P, K into L just before $L.P_1$
else begin
Let K_i be the highest value in L that is less than K
Insert P, K into L just after $T.K_i$
end

ocedure insert_in_parent(*node* N, *value* K', *node* N')
if (N is the root of the tree)
then begin
Create a new node R containing N, K', N' /* N and N
Make R the root of the tree

For "SELECT" Queries: How to retrieve the blocks containing our data records with minimal disk access?

For "INSERT/DELETE" Queries: How to ensure disk space is wisely used?

Can we improve indexing for **faster search**, improved **space utilization**?

Now, we have some answers!

1. Hashing, Sequential, Heap FO
2. Free Lists, Variable Length Records
3. Secondary Index, Multi-level Index, Dense and Sparse Indices, B+ Trees

Research Directions

- We have gone a long way away from “Sequential File Organization with Fixed Length Records”.

File Searching Using Variable Length Keys

RENE DE LA BRIA

MANY computer applications require the storage of large amounts of information within the computer's memory where it will be readily available for reference and updating. Quite commonly, more storage space is required than is available in the computer's high-speed working memory. It is, therefore, a common practice to equip computers with magnetic tapes, disks, or drums, or a combination of these to

it just t
ing reco
ing the
then ta
record.
corresp
nique re
handled

1959

Programming Techniques
G. Manacher, S.L. Graham Editors

Pagination of B*-Trees with Variable-Length Records

Edward M. McCreight
Xerox Palo Alto Research Center

A strategy is presented for pagination of B*-trees with variable-length records. If records of each length are uniformly distributed within the file, and if a wide distribution of record lengths exists within the file, then this strategy results in shallow trees with fast access times. The performance of this strategy in an application is presented, compared with that of another strategy, and analyzed.

Key words and Phrases: B-tree, index, database, tree, storage structure, searching
CR Categories: 3.73, 4.33, 4.34

Introduction

B^*-trees have come into increasing use as index structures for large disk-stored databases. A number of characteristics recommend them for such applications. They preserve order among keys, so neighborhood searching is an inexpensive operation. They preserve reasonable density and good access times across any sequence of updating operations. They carry out re-organization locally and incrementally in response to updating operations, so massive global index re-organizations are never required.

For our purposes, a keyed record contains two fields: a key, which is a member of some totally-ordered domain (like the set of eight-digit decimal integers, for example), and a value, which might represent some property of the key, or might be a pointer to a record in a database associated with the key. A B^*-*tree of order m* [1, 2] is an ordered uniform-depth tree of fixed-length pages (or nodes) each containing an ordered sequence of fixed-length keyed records. Every page contains at most m-1 records, and every page except the root contains at least $(2m-4)/3$ records. The root page contains at least one record. In other words, every page except the root is somewhere between $\frac{2}{3}$ full and completely full, and the root page may not be empty.

A page represents an interval in the linear key space, and its son pages represent a partition of its interval into subintervals. Thus each nonleaf page has one more son page than the number of (interval boundary) records it contains. Insertion of a new record is

1977

Progressive Indexes: Indexing for Interactive Data Analysis

...eldt
...dam
...wi.nl

Stefan Manegold
CWI, Amsterdam
manegold@cwi.nl

Hannes Mühleisen
CWI, Amsterdam
hannes@cwi.nl

...asingly ...ting in- ...resents ..., as (1) ...ice, (2) ...the sys- ...queries. ... as tra- ...ence a

...a novel ... on au- ...sponse ...o have

and generate hypotheses. When dealing with small data sets, providing answers within this interactivity threshold is possible without utilizing indexes. However, exploratory data analysis is often performed on larger data sets as well. In these scenarios, indexes are required to speed up query response times.

Index creation is one of the major difficult decisions in database schema design [8]. Based on the expected workload, the database administrator (DBA) needs to decide whether creating a specific index is worth the overhead in creating and maintaining it. Creating indexes up-front is especially challenging in exploratory and interactive data analysis, where queries are not known in advance, workload patterns change frequently and interactive responses are required. In these scenarios, data scientists load their data

2019

Bigtable: A Distributed Storage System for Structured Data

FAY CHANG, JEFFREY DEAN, SANJAY GHEMAWAT, WILSON C. HSIEH, DEBORAH A. WALLACH, MIKE BURROWS, TUSHAR CHANDRA, ANDREW FIKES, and ROBERT E. GRUBER
Google, Inc.

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this article, we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed databases*

2008

Thank You

B+ Tree simulation available at
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html