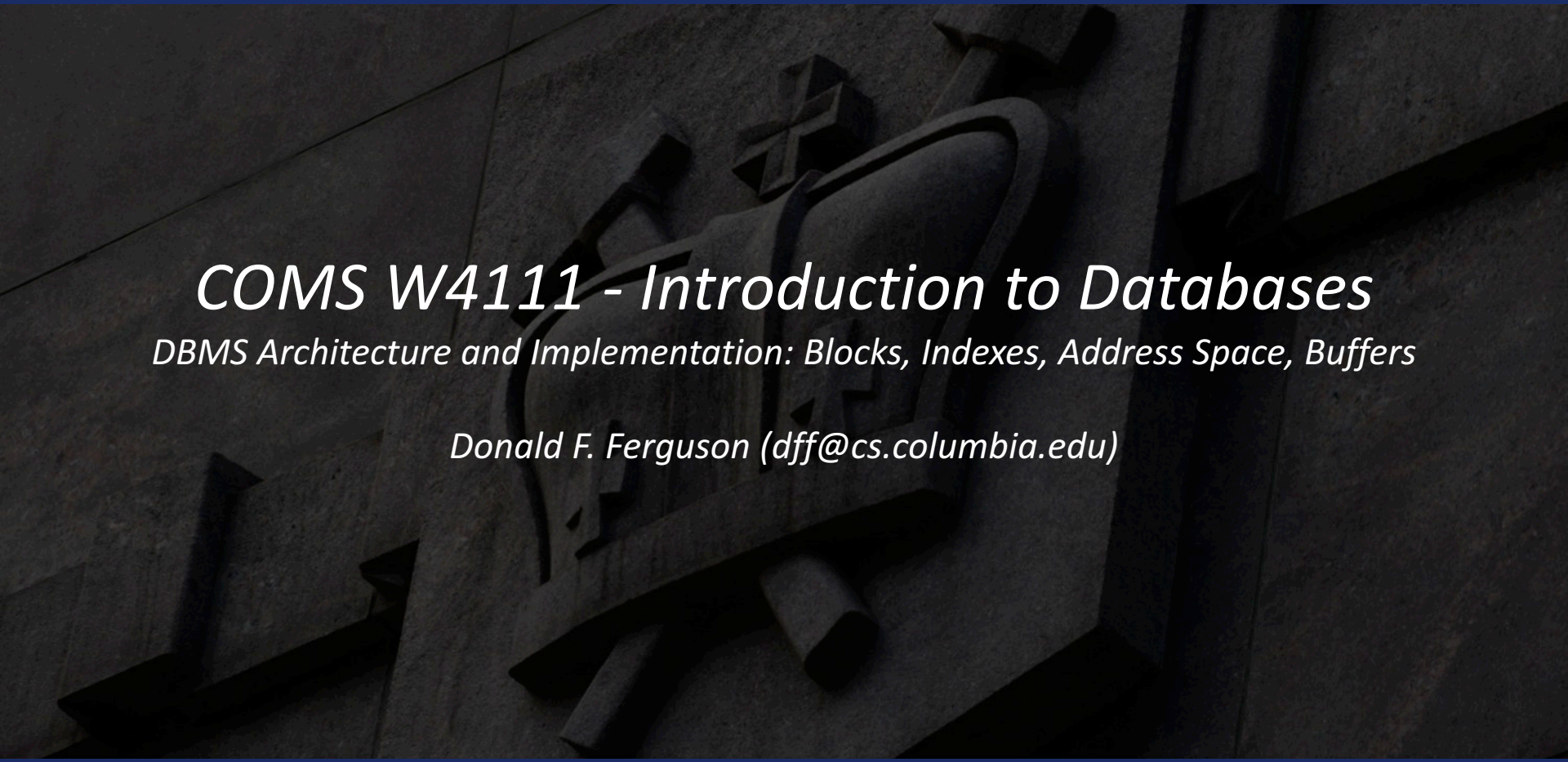


COMS W4111 - Introduction to Databases

DBMS Architecture and Implementation: Blocks, Indexes, Address Space, Buffers

Donald F. Ferguson (dff@cs.columbia.edu)

Module II – DBMS Architecture and Implementation

Reminder

Database Management System Reminder

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

Covered for the relational model.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Database Management System Reminder

2. Give users the ability to *query* the data (a "query" is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.

3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.

4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.

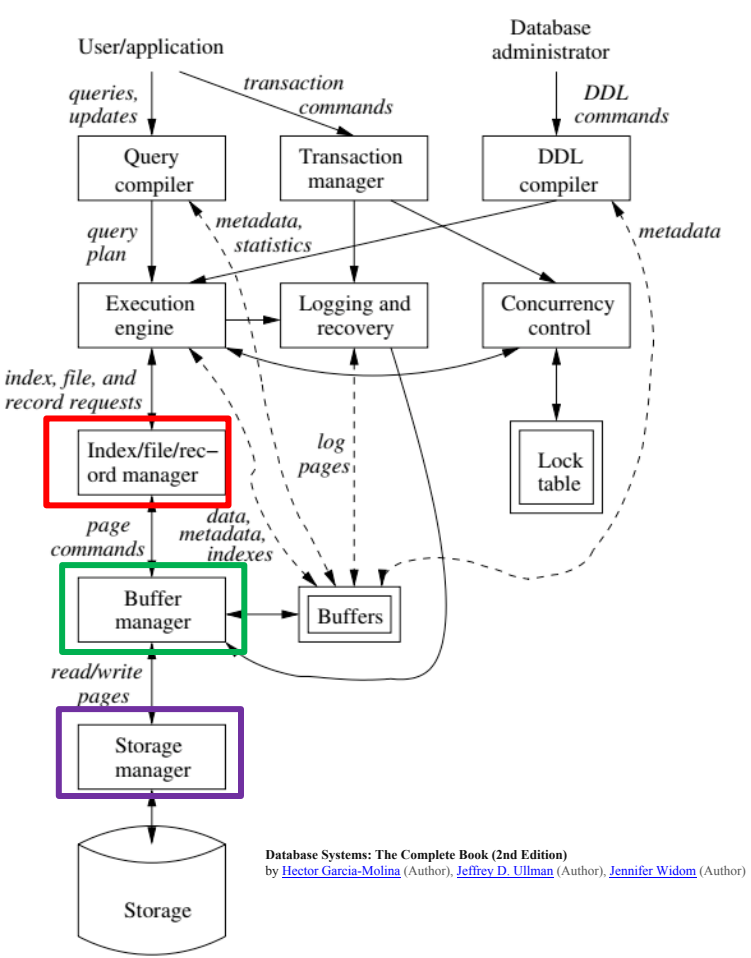
5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

Focus for next part of course.

Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Data Management

- Find things quickly.
- Access things quickly.
- Load/save things quickly.



Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Database Address Space

Data Address Space (Overly Simplified)

- The block/storage system represents block addresses with
  - A logical ID
  - That is simply a string, e.g. UUID or has some structure.
- There is a logical address to physical address mapping table that maps a logical address to a physical address
  - Block/storage engine ID.
  - An opaque address string that *the specific engine understands and manages*.
- For example, in a storage area network, the physical address might be
  - IP address of storage server.
  - ID of disk attached to the server.
  - {Cylinder, Head Number, Sector}

Any reference to a block, e.g. in an index node, uses the logical address.

Figuring out where, when and why to put a block in a specific engine/domain is an important optimization function in DMBS.

Da[illegible]Overly Simply)

Straightforward:
- s-sever11.nas.myco.com
- /var/dev1

- [illegible]ents block addresses with
  - [illegible]ply a string, e.g. UUID or has some structure
- There i[illegible]ical address to physical addr[illegible]
  logical a[illegible]s to a physical address
  - Block/s[illegible]ge engine ID.
  - An opaq[illegible]address string that *the specific e[illegible]*
- For example, [illegible]n a storage area network, the [illegible]ess might be
  - IP address of storage server.
  - ID of disk attached to the server.
  - {Cylinder, Head Number, Sector}

Most devices surface *Logical Block Addressing.*

Any reference to a block, e.g. in an index node, uses the logical address.

Figuring out where, when and why to put a block in a specific engine/domain is an important optimization function in DMBS.

Logical Block Addressing (https://gerardnico.com/wiki/data_storage/lba)

3 - The LBA scheme

LBA	C	H	S
0	0	0	0
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
5	0	0	5
6	0	0	6
7	0	0	7
8	0	0	8
9	0	0	9
10	0	1	0
11	0	1	1
12	0	1	2
13	0	1	3
14	0	1	4
15	0	1	5
16	0	1	6
17	0	1	7
18	0	1	8
19	0	1	9
Cylinder 0			

LBA	C	H	S
20	1	0	0
21	1	0	1
22	1	0	2
23	1	0	3
24	1	0	4
25	1	0	5
26	1	0	6
27	1	0	7
28	1	0	8
29	1	0	9
30	1	1	0
31	1	1	1
32	1	1	2
33	1	1	3
34	1	1	4
35	1	1	5
36	1	1	6
37	1	1	7
38	1	1	8
39	1	1	9
Cylinder 1			

- CHS addresses can be converted to LBA addresses using the following formula:

$$LBA = ((C \times HPC) + H) \times SPT + S - 1$$

where,

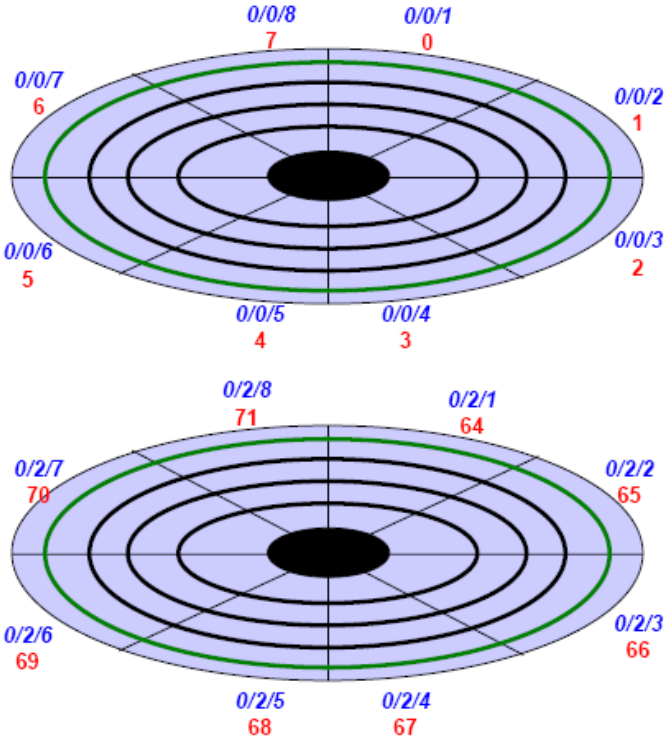
- C, H and S are the cylinder number, the head number, and the sector number
- LBA is the logical block address
- HPC is the number of heads per cylinder
- SPT is the number of sectors per track

Devices have configuration and metadata APIs that allow storage manager to

- Map between LBA and CHS
- To optimize block placement
- Based on access patterns, statistics, data schema, etc.

Another View

(http://www.c-jump.com/CIS24/Slides/DiskDrives/D01_0150_chs_vs_lba.htm)



Simple example: 4 sides, 8 sectors/side, 4 tracks/side (cylinders) yields 128 total sectors...

The green track is cylinder 0. On the first side (side 0), sectors are numbered 0/0/1 thru 0/0/8 in CHS format, or 0 thru 7 in LBA.

On cylinder 1 of the first side, sectors are numbered 1/0/1 thru 1/0/8 (8 thru 15).

The last sector on side 0 is numbered 4/0/8 (31).

Second side (side 1) starts:	*0/1/1*	*32*
Second side ends:	*4/1/8*	*63*
Third side (side 2) starts:	*0/2/1*	*64*
Third side ends:	*4/2/8*	*95*
Fourth side (side 3) starts:	*0/3/1*	*96*
Fourth side ends:	*4/3/8*	*127*

Summary

- There is a broad category of storage devices
  - "Just a disk"
  - RAID-0, RAID-1, RAID-5, RAID-X/Y
  - Network/cloud addressable devices
  - SSD
- A device has configuration information, often via APIs
  - Device geometry
  - Reliability metrics
  - IOPs, and sub-elements (seek, rotate, etc).
- Storage management consists of using statistics and query (optimization patterns) to
  - Arrange records into blocks for storage and IO efficiency.
  - Place blocks on devices at LBAs to meet performance and IO objectives.

Summary

- Almost all of this complexity and functionality is hidden from
  - Database developers.
  - Database designers and administrators.
- Data center, cloud and large scale infrastructure architects
  - Evaluate and design disk technology and architecture
  - In combination with other critical technology, e.g.
    - Networking
    - Process/compute nodes.
- Some customer, optimized, advanced data solutions are a combination of
  - Custom, optimized database implementation.
  - Co-designed with underlying infrastructure design.
- The disk arrays and controllers do a lot of the low-level functions, but …
  - The database admin using DB analytics understands the access patterns.
  - And uses DB tools to choose, optimize and configure storage options.

Record and Block Management

Terminology

- A tuple in a relation maps to a *record*. Records may be
  - *Fixed length*
  - *Variable length*
  - *Variable format* (which we will see in Neo4J, DynamoDB, etc).
- A *block*
  - Is the unit of transfer between disks and memory (buffer pools).
  - Contains multiple records, usually but not always from the same relation.
- The *database address space* contains
  - All of the blocks and records that the database manages
  - Including blocks/records containing data
  - And blocks/records containing free space.

Fixed Length Record

- Sample DDL

```
CREATE TABLE `products` (
  `product_id` char(8) NOT NULL,
  `product_name` varchar(16) NOT NULL,
  `product_description` char(8) NOT NULL,
  `product_brand` enum('IBM','HP','Acer','Lenovo','Some really long brand name') NOT NULL,
  PRIMARY KEY (`product_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- Why
  - Is this fixed length? product_brand and product_name are clearly variable length.
  - Isn't char(8) to short for a product description?

Hypothetical Explanations

"In computing, **internationalization and localization** are means of adapting computer software to different languages, regional differences and technical requirements of a target locale.[1] Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. Localization (which is potentially performed multiple times, for different locales) uses the infrastructure or flexibility provided by internationalization (which is ideally performed only once, or as an integral part of ongoing development)." (https://en.wikipedia.org/wiki/Internationalization_and_localization)

- Retrieving a product description may require two elements
  - Language code, e.g. (EN, DE, ES, …)
  - Description ID, e.g. 01105432
  - (EN, 01105432) and (DE, 01105432) are the same description in different languages (English and German)
  - The description ID is an attribute of the product.
  - The language is an attribute or property of the user, install location, etc.

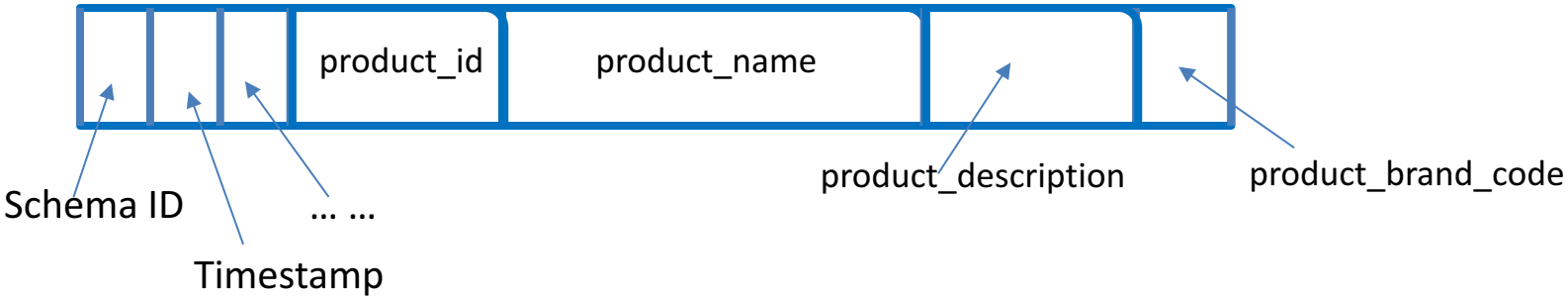
Hypothetical Explanations

- This concept is
  - An element of data modeling and design
  - Not DBMS implementation or data management.
- I introduce the concept because
  - Important, general data model topic we have not covered.
  - Pattern is a reason why many records we think would be variable are often fixed.
    - VARCHAR field in a tuple is often
    - A pointer to a value in some other record.

Hypothetical Explanation

- `product_name` ***varchar***(16) **NOT NULL**
  - (NB: Assumes that the product_name is not localized).
  - Length can be 0 .. 15 characters.
  - But,
    - If the value is almost always between 12 and 16 characters.
    - The database engine may decide to store in a fixed length area.
    - Because the relatively small savings available from treating like a variable length record
    - Does not justify the added complexity involved in variable size record management.
- `product_brand` enum(**'IBM'**,**'HP'**,**'Acer'**,**'Lenovo'**,**'Some really long brand name'**)
  - The DB engine would/could represent with a 1 byte code in a lookup table.
  - This enables simplified, fixed length record management
  - And saves space because
    - 1,000 products in the catalog from 'Some really long brand'
    - Requires 1,000 bytes instead of 27,000 bytes.
  - Repeated values:
    - The same value occurring repeatedly in multiple distinct tuples is common, even when not an enum, e.g.
      - Country names, City names, street names, last names, first names, …
      - Foreign keys in many-to-many associative entities.
    - DBMS storage management engine may detect the situation, and replace values with symbol lookups.

Fixed Length Record



The record has

- Database specific header fields, e.g.
  - Format information
  - Last modified
- Areas for each of the fixed length fields.

Packing Fixed Length Blocks

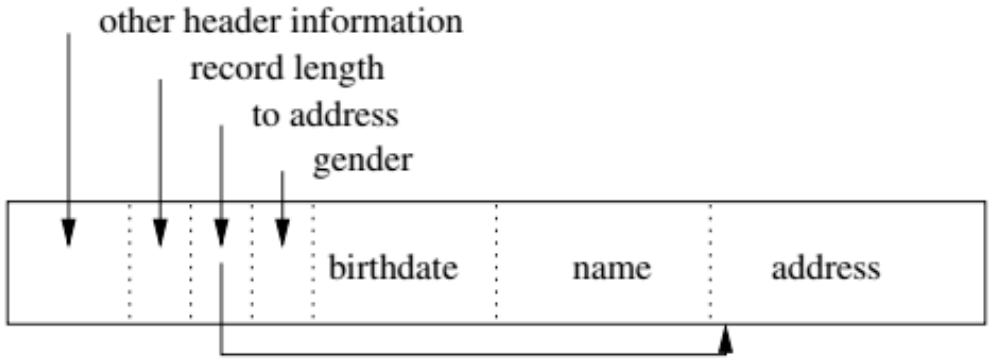
Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina and Jeffrey D. Ullman

Figure 13.17: A typical block holding records

In addition to the records, there is a *block header* holding information such as:

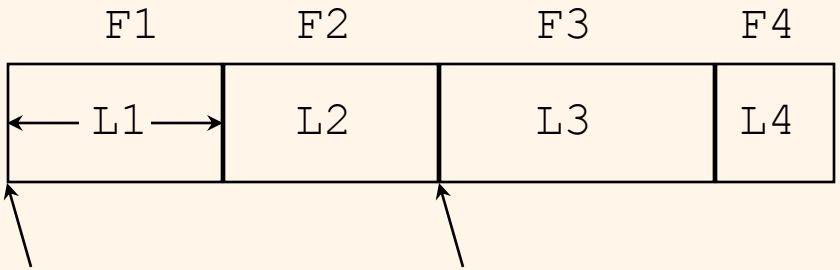
1. Links to one or more other blocks that are part of a network of blocks such as those that will be described in Chapter 14 for creating indexes to the tuples of a relation.
2. Information about the role played by this block in such a network.
3. Information about which relation the tuples of this block belong to.
4. A "directory" giving the offset of each record in the block.
5. Timestamp(s) indicating the time of the block's last modification and/or access.

Variable Length Record



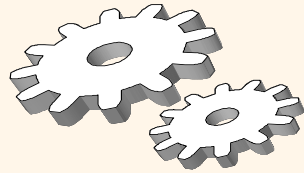
- Same DBMS specific header information as fixed-length, plus
  - Record length.
  - Offset into record for each of the fields.
- Variable sized areas for each of the variable sized fields.

Record Formats: Fixed Length

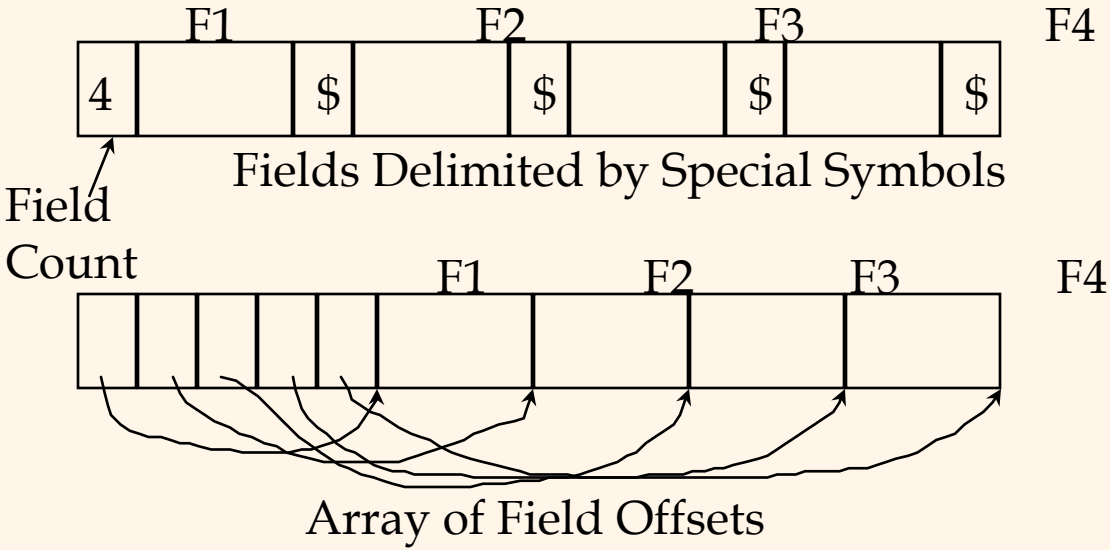


- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field does not require scan of record.

Record Formats: Variable Length



- Two alternative formats (# fields is fixed):



☛ Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

INSERT, DELETE, UPDATE

Fixed Length Records

Variable Length Records

INSERT, UPDATE and DELETE

- Is pretty straightforward for fixed length records
- But can get pretty interesting for variable length records, e.g.
  - What if INSERTed R5 is bigger than E1 or E2?
  - How do I expand R2?
  - Deleting R2 leaves a small gap → Fragmentation?

One (Simple) Solution is …

The block does not look like this in memory.

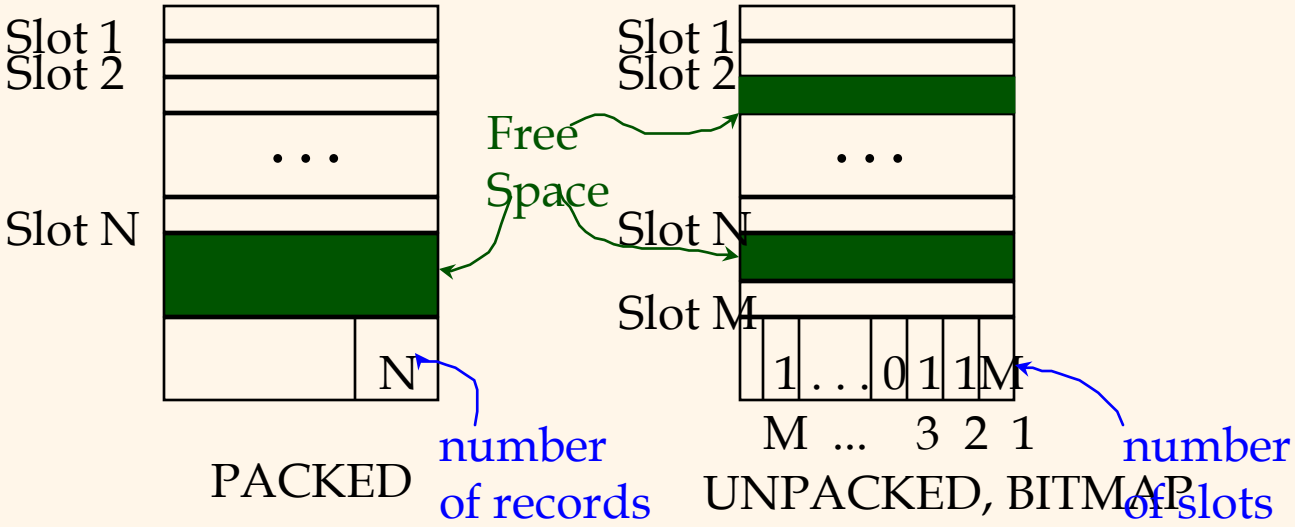
Variable Length Records

It always looks like this in memory …

Variable Length Records

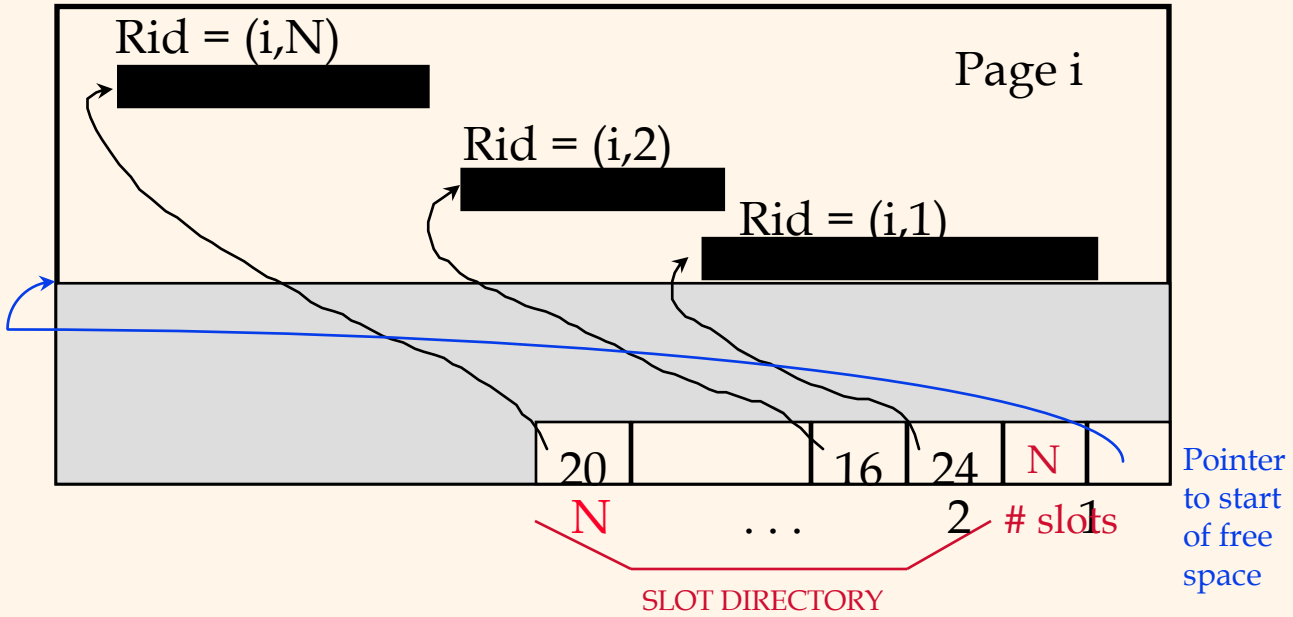
- INSERT always goes into the start of E1.
- DELETE or UPDATE R4:
  - Move (R2,R3) to end of R1
  - If R4 updated, move to end of new position of R3.

Page Formats: Fixed Length Records



- *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records



- *Can move records on page without changing rid; so, attractive for fixed-length records too.*

Block and Record Addresses

Blocks and records have addresses

- Index tables map an index entry to {block, record}
- SELECT scan on a table needs a logical linked list of the blocks holding records, e.g. “next block.”
- Index elements are themselves blocks, and need to reference other nodes in the index.

- But, …
  - The blocks can be all over the place, and
  - Each “place” has a different address format and access protocol.

Cloud 2

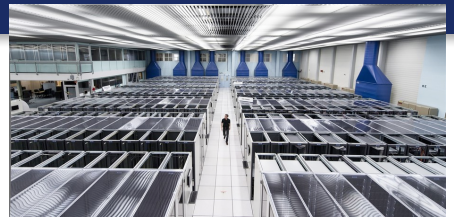
RACKSPACE® CLOUD BLOCK STORAGE

Cloud 1

AWS Storage Gateway Service

amazon web services EBS

Disk Room 1



Disk Room 2

Tape Library

Data Address Space (Overly Simply)

- The block/storage system represents block addresses with
  - A logical ID
  - That is simply a string, e.g. UUID or has some structure.
- There is a logical address to physical address mapping table that maps a logical address to a physical address
  - Block/storage engine ID.
  - An opaque address string that *the specific engine understands and manages*.
- For example, in a storage area network, the physical address might be
  - IP address of storage server.
  - ID of disk attached to the server.
  - {Cylinder, Track, Sector}

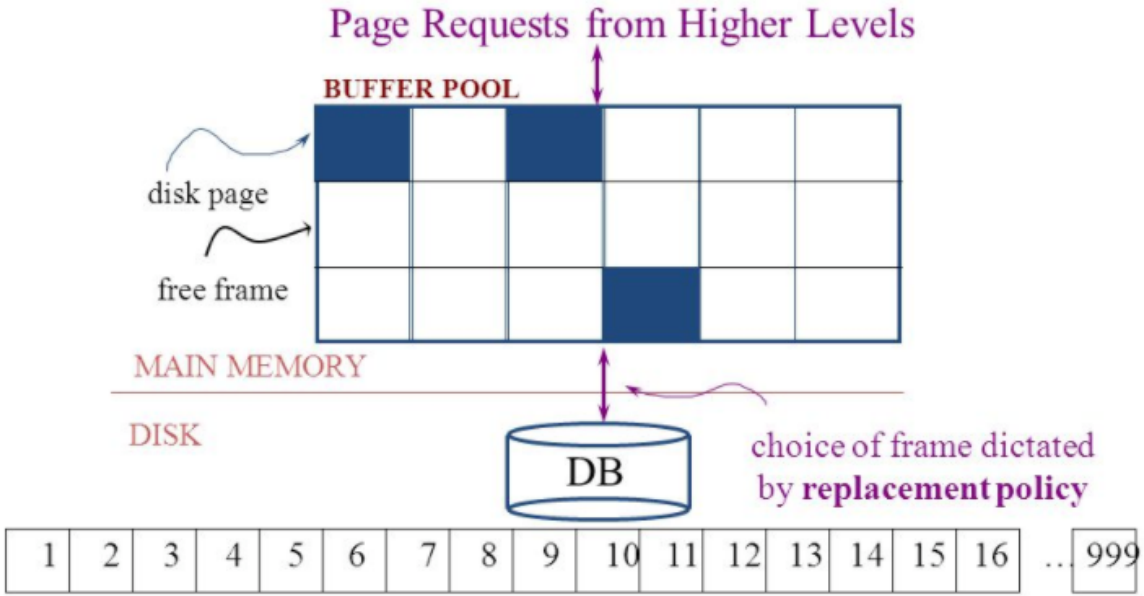
Any reference to a block, e.g. in an index node, uses the logical address.

Figuring out where, when and why to put a block in a specific engine/domain is an important optimization function in DMBS.

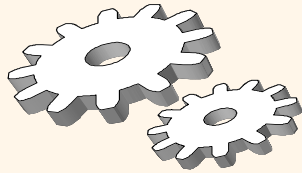
Buffer Pools

The Logical Concept

- The DBMS and queries can only manipulate in-memory blocks and records.
- A very, very, very small fraction of all blocks fit in memory.



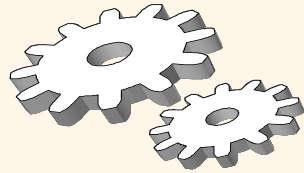
When a Page is Requested ...



- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

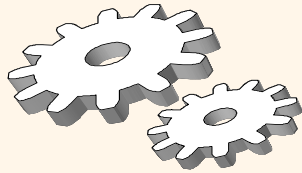
☛ *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!*

More on Buffer Management



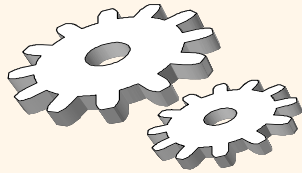
- Requestor of page must unpin it, and indicate whether page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later.)

Buffer Replacement Policy



- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern*.
- *Sequential flooding*: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

DBMS vs. OS File System



OS does disk space & buffer mgmt: why not let OS manage these tasks?

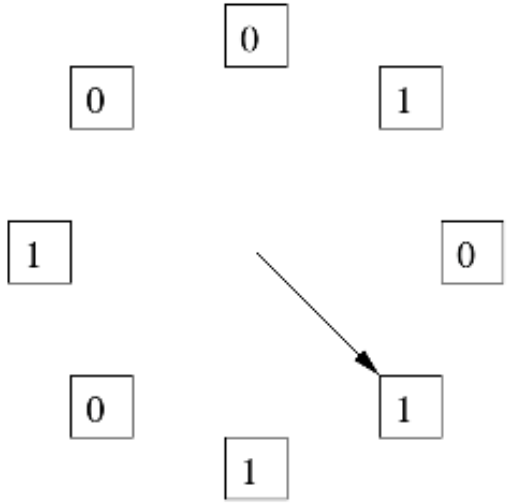
- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

Replacement Policy

- The *replacement policy* is one of the most important factors in database management system implementation and configuration.
- A very simple, introductory explanation is (https://en.wikipedia.org/wiki/Cache_replacement_policies).
  - There are a lot of possible policies.
  - The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Bélády's optimal algorithm/simply optimal replacement policy or the clairvoyant algorithm.
- All implementable policies are an attempt to approximate knowledge of the future based on knowledge of the past.
- Least Recently Used is based on the simplest assumption
  - The information that will not be needed for the longest time.
  - Is the information that has not been accessed for the longest time.

The “Clock Algorithm”

- LRU is (perceived to be) expensive
  - Maintain timestamp for each block.
  - Update and resort blocks on access.
- The “Clock Algorithm” is a less expensive approximation.
  - Arrange the frames (places blocks can go) into a logical circle like the seconds on a clock face.
  - Each frame is marked 0 or 1.
    - Set to 1 when block added to frame.
    - Or when application accesses a block in frame.
  - Replacement choice
    - Sweep second hand clockwise one frame at a time.
    - If bit is 0, choose for replacement.
    - If bit is 1, set bit to zero and go to next frame.
- The basic idea is. On a clock face
  - If the second hand is currently at 27 seconds.
  - The 28 second tick mark is “the least recently touched mark.”



Replacement Algorithm

The algorithms are more sophisticated in the real world, e.g.

- "Scans" are common, e.g. go through a large query result in order (will be more clear when discussing cursors).
  - The engine knows the current position in the result set.
  - Uses the sort order to determine which records will be accessed soon.
  - Tags those blocks as not replaceable.
  - (A form of clairvoyance).
- Not all users/applications are equally "important."
  - Classify users/applications into priority 1, 2 and 3.
  - Sub-allocate the buffer pool into pools P1, P2 and P3.
  - Apply LRU within pools and adjust pool sizes based on relative importance.
  - This prevents
    - A high access rate, low-priority application from taking up a lot of frames
    - Result in low access, high priority applications not getting buffer hits.

Summary

- Like all other aspects of DMBS implementation
  - Block management
  - Buffer management

  Are incredibly complex, heavily researched topics.

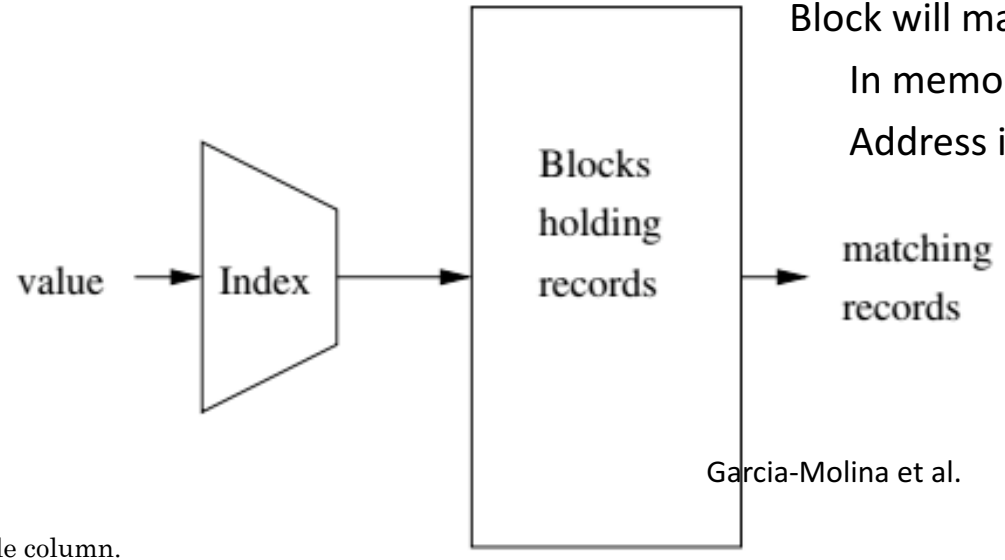
- This course section has just skimmed the surface
  - Awareness of concepts.
  - Foundation for future more advanced courses.
  - Understand concepts to consider, research, prototype, etc. if you have to implement your own data management engine.

Indexes

Index

```
CREATE TABLE `Customers` (
  `CustomerID` varchar(5) NOT NULL,
  `CompanyName` varchar(40) NOT NULL,
  `ContactName` varchar(30) NOT NULL,
  `ContactTitle` varchar(30) DEFAULT NULL,
  `Address` varchar(60) DEFAULT NULL,
  `City` varchar(15) DEFAULT NULL,
  `Region` varchar(15) DEFAULT NULL,
  `PostalCode` varchar(10) DEFAULT NULL,
  `Country` varchar(15) DEFAULT NULL,
  `Phone` varchar(24) DEFAULT NULL,
  `Fax` varchar(24) DEFAULT NULL,
  PRIMARY KEY (`CustomerID`),
  KEY `City` (`City`),
  KEY `CompanyName` (`CompanyName`),
  KEY `PostalCode` (`PostalCode`)
```

Column values or set of column values.



Block will map to

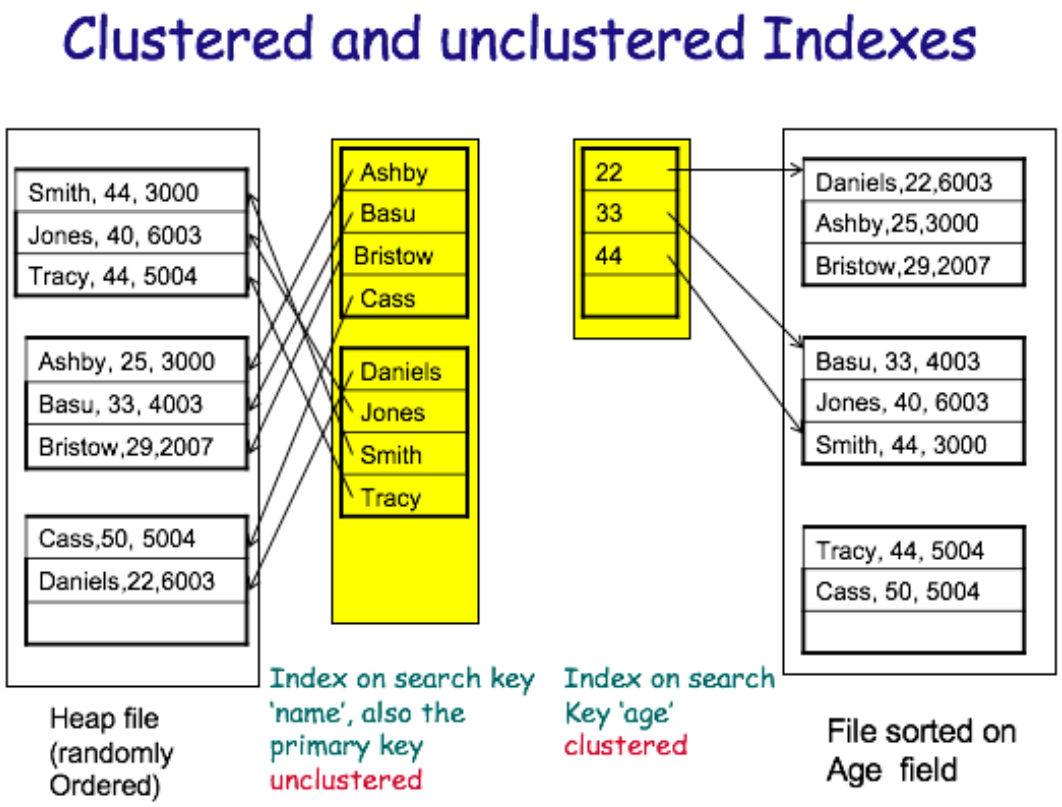
In memory block in frame or

Address in Data Address Space.

Garcia-Molina et al.

- Indexes are created on
  - Simple keys = e.g. a single column.
  - Composite keys = a set of columns, e.g. (last_name, first_name)
- Most common index implementations are: *Hash, B+ Tree*
- An *entry* in an index if of the form – (k, v):
  - k is the key value.
  - v is one of: 1) the data itself (tuple), 2) the address in data address space of tuple, 3) a list of addresses
- Index can be *unique/non-unique*, *dense/sparse, clustered/unclustered, ordered/hashed.*

Clustered vs Unclustered



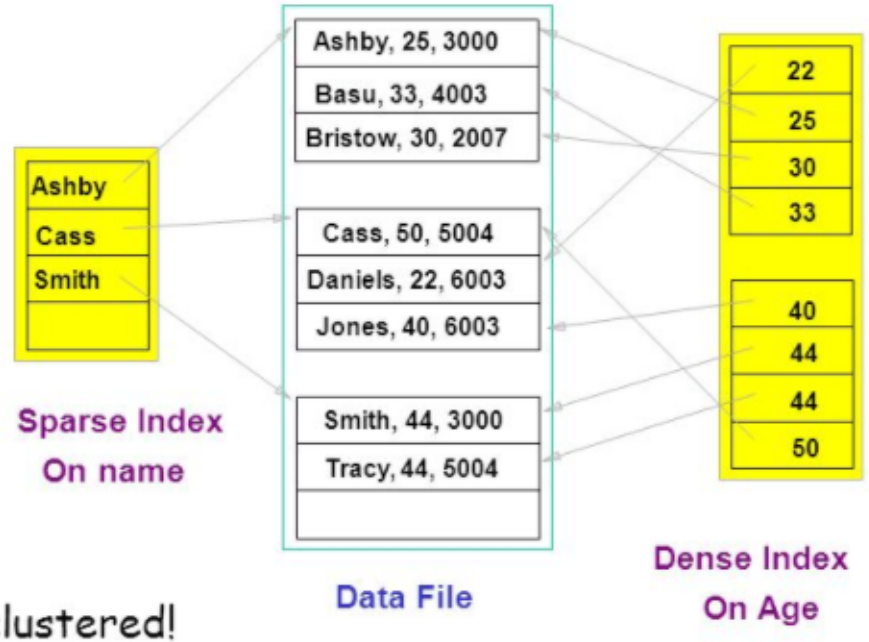
Order of keys in index
is approximately
the same as
order of records in file.

Dense versus Sparse Index

Dense vs Sparse Index

"Storage and File Structure. Architecture of a DBMS," Shona Holmes.
http://slideplayer.com/slide/6841281/

- Dense vs Sparse:
- If there is at least one index entry in the index per search key value, then dense.



Every sparse index is clustered!

B+ Tree

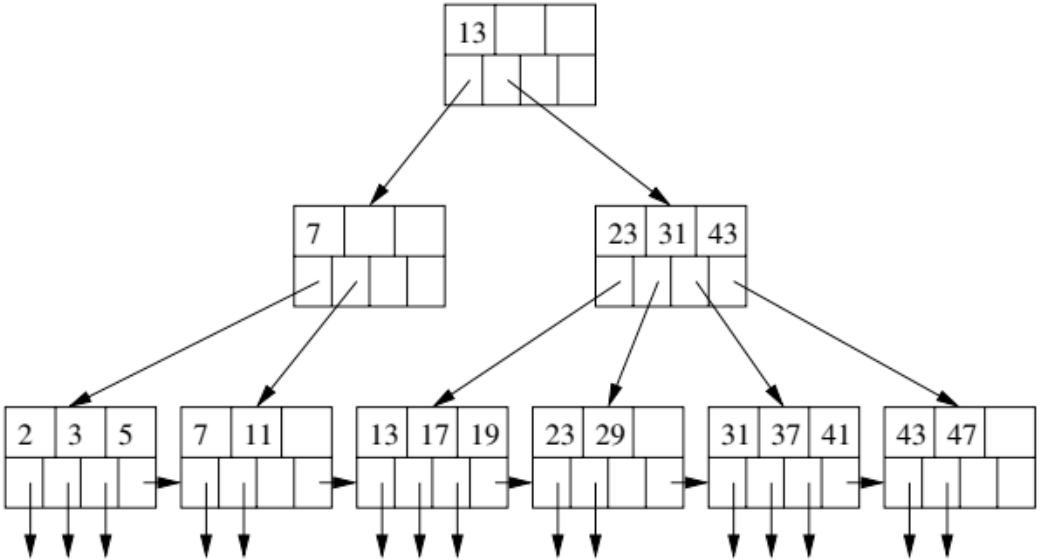


Figure 14.13: A B-tree

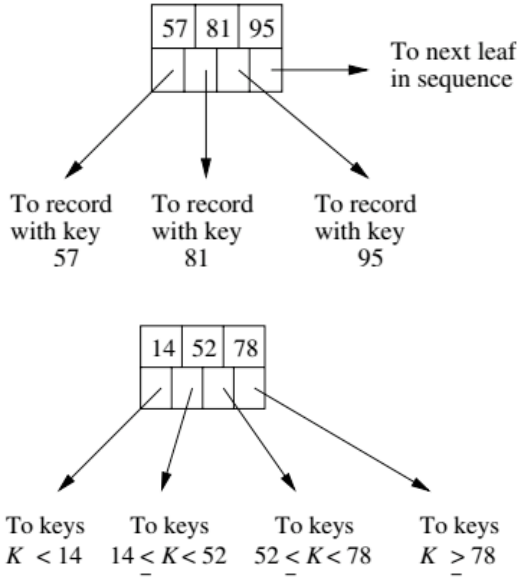
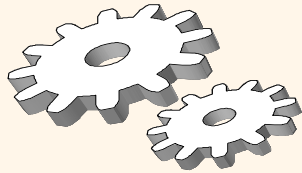


Figure 14.12: A typical interior node of a B-tree

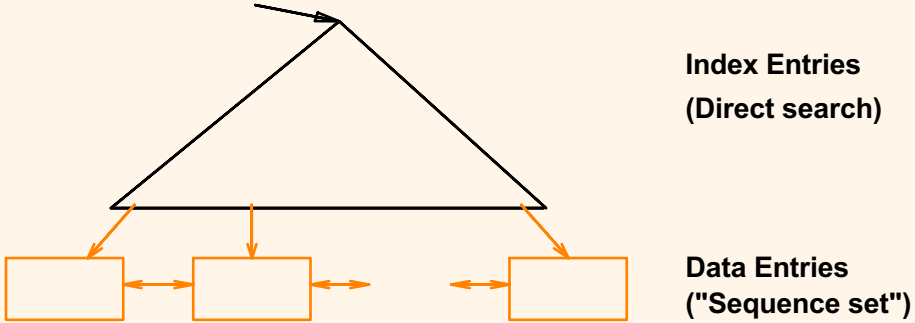
Based on concept of a *binary search tree*, but with two core modifications

1. Unit of transfer to/from memory is a *block*, which will hold many (key,record) maps.
2. Accessing records in order (WRT to index) is common → clustering is common.

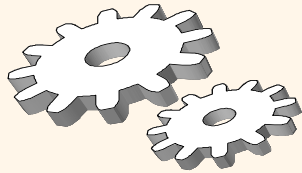
B+ Tree: Most Widely Used Index



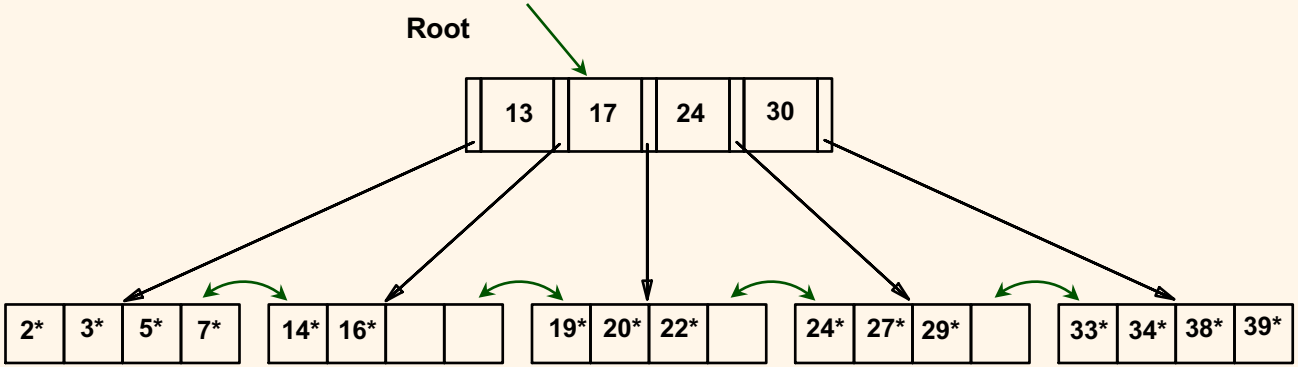
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

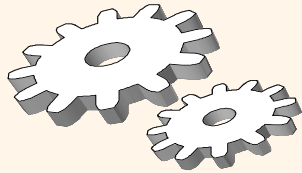


- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



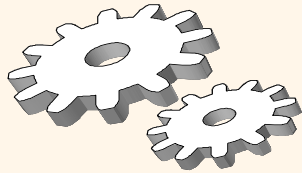
☛ *Based on the search for 15*, we know it is not in the tree!*

B+ Trees in Practice



- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

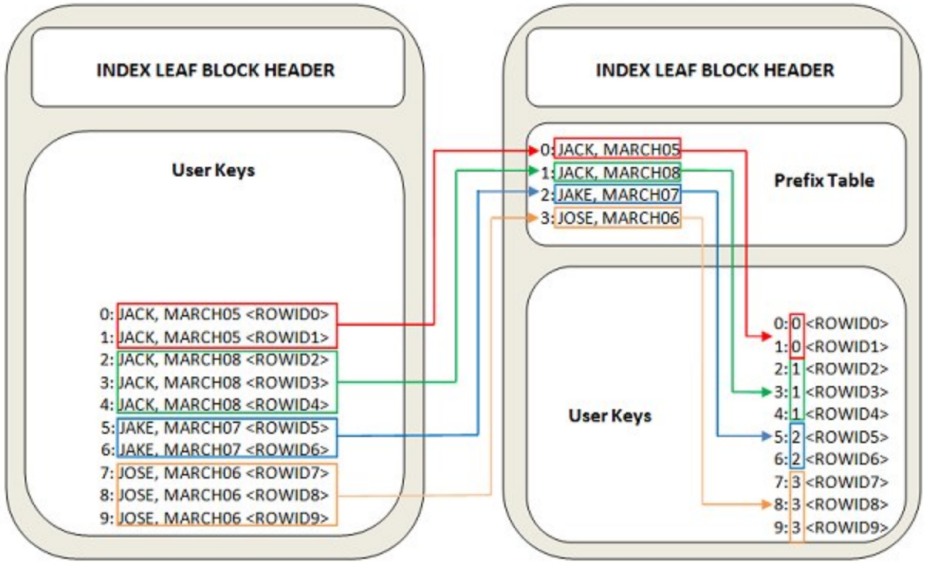
Prefix Key Compression



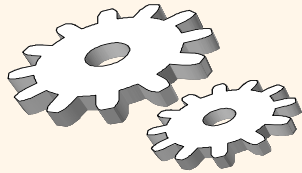
- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

Index Key Compression

Index Key Compression allows us to compress portions of the key values in an index segment (or Index Organized Table), by reducing the storage inefficiencies of storing repeating values. It compresses the data by splitting the index key into two parts; the leading group of columns, called the prefix entry (which are potentially shared across multiple key values), and the suffix columns (which is unique to every index key). As the prefixes are potentially shared across multiple keys in a block, these can be stored more optimally (that is, only once) and shared across multiple suffix entries, resulting in the index data being compressed.

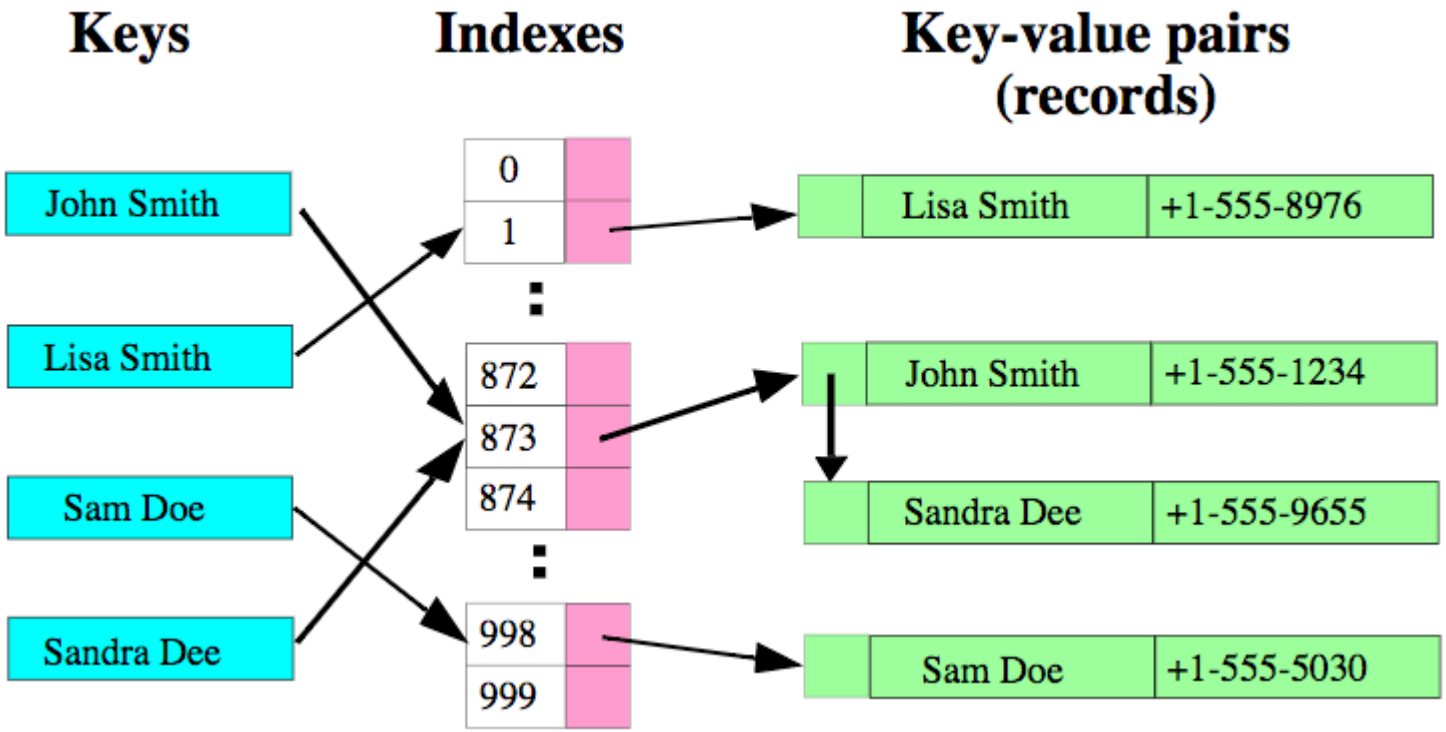


Static Hashing

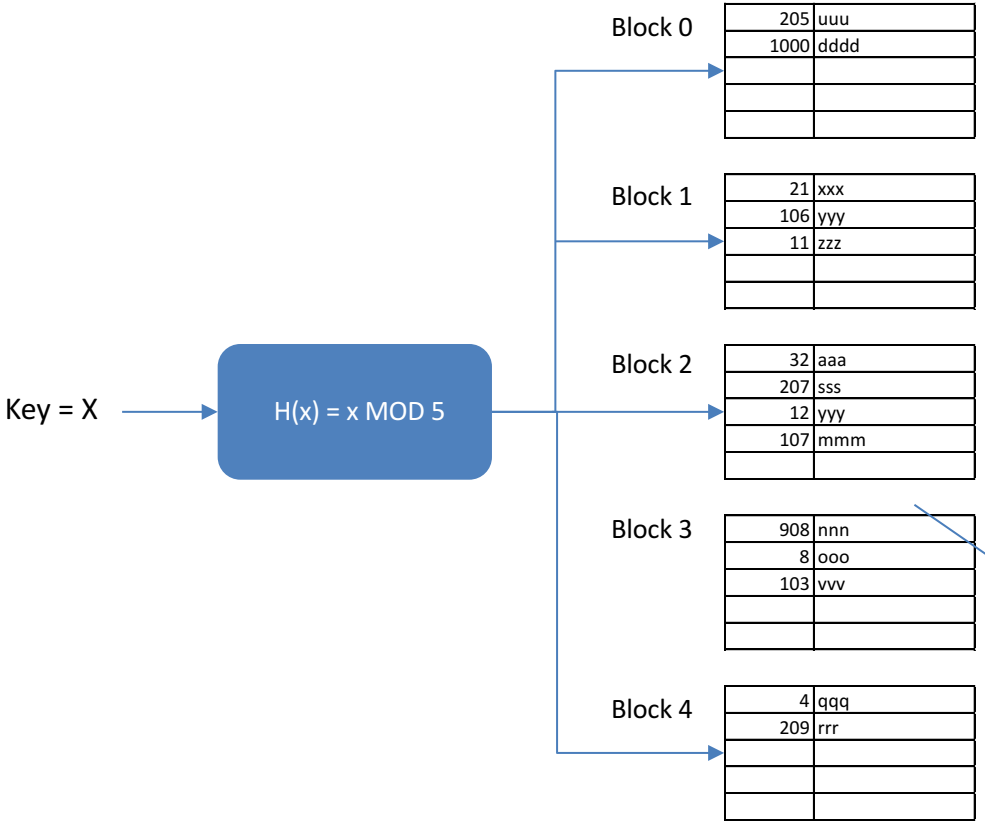


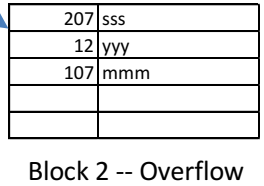
- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Hash Index – Basic Data Structure



Database Hash Index





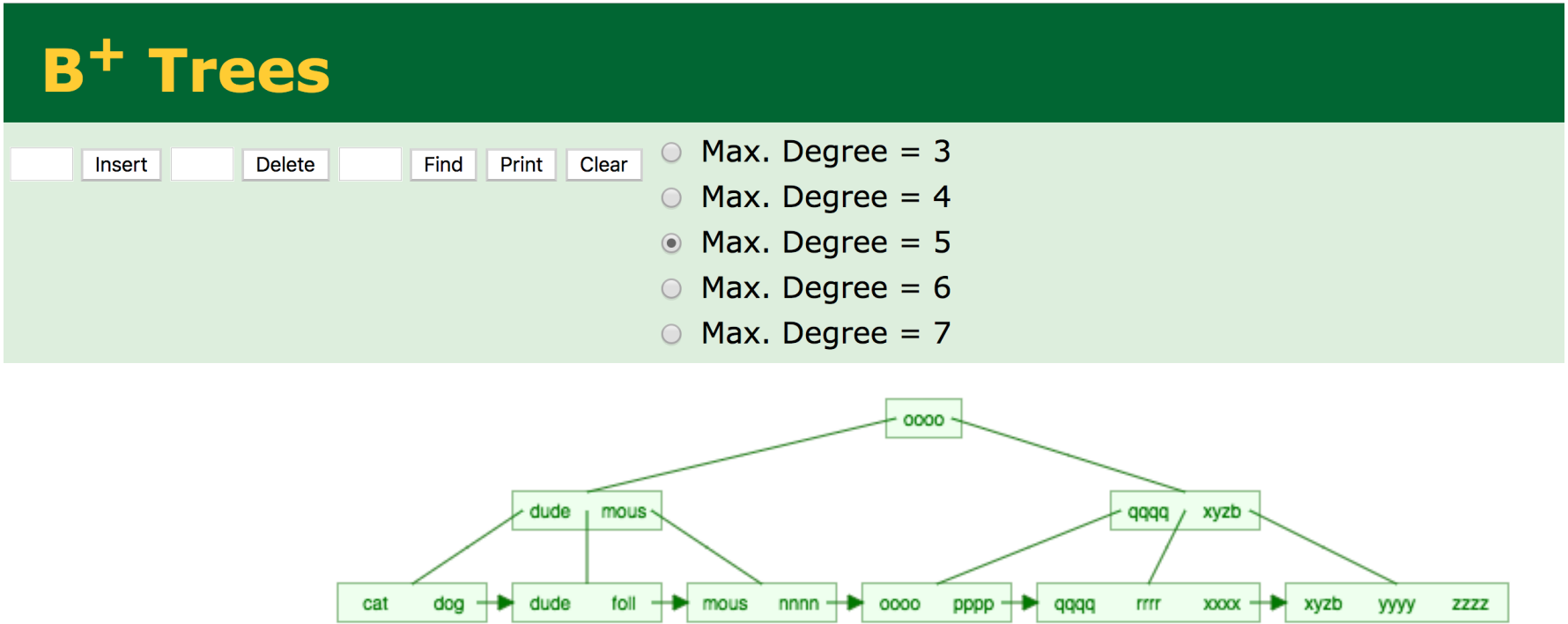
- Index blocks are disk blocks, just like data blocks.
- The format is
  - Key value
  - Data value
- The data value may be
  - The actual record, or
  - The logical address for the data block and record ID holding the tuple.

Some Comparisons

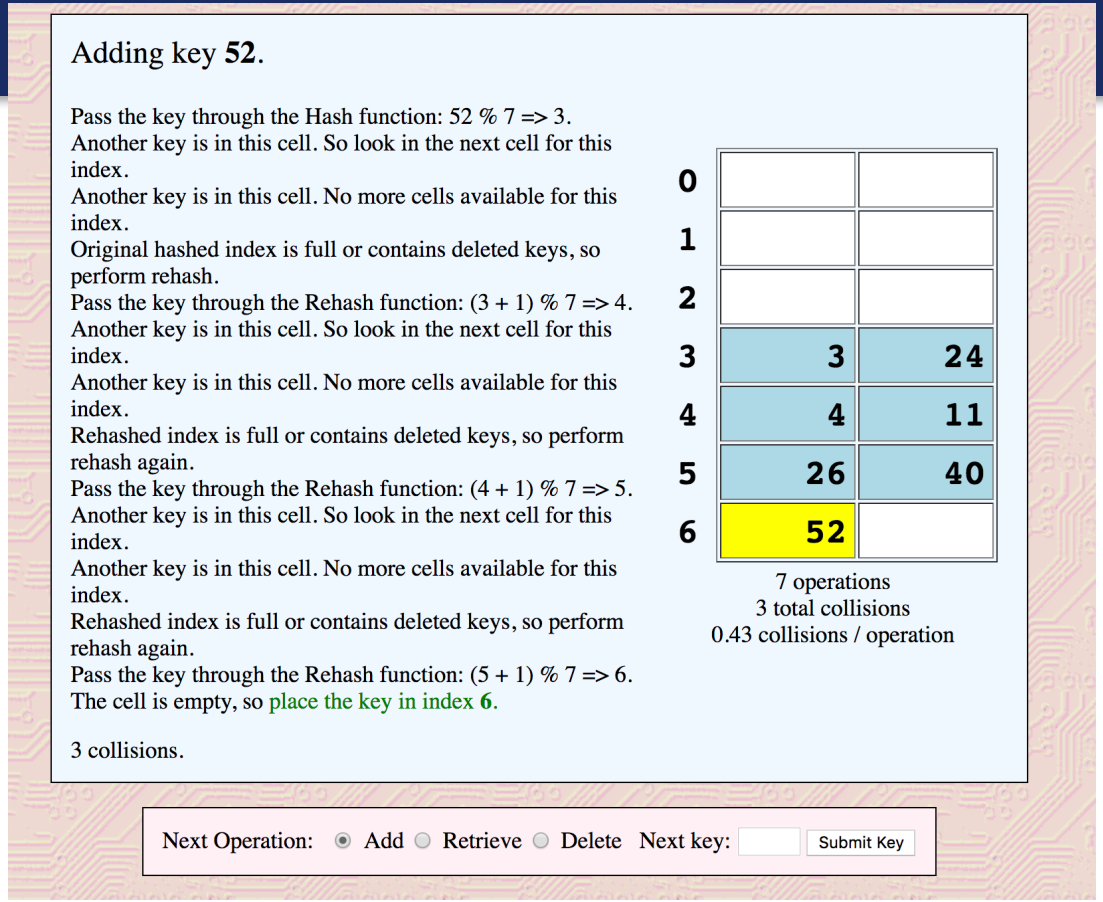
- B-Tree supports
  - WHERE operators =, >, >=, <, <=, BETWEEN and LIKE (for operands).
  - MIN(), MAX()
  - ORDER BY
  - WHERE clauses with partial keys and AND. A key on (c1, c2, c2) is actually a set of indexes of the form (c1), (c1,c2) and (c1,c2,c3).
  - Estimates of the number of records in a range (for query optimization).
- Hash Index
  - Extremely fast for the (key, value) data model.
  - Only for operators =, NOT =
  - Does not support partial keys.

Some Simulators

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



Some Simulators



- http://www.cse.unt.edu/~donr/courses/2050/HashSimulator/hashsim.cgi
- http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html