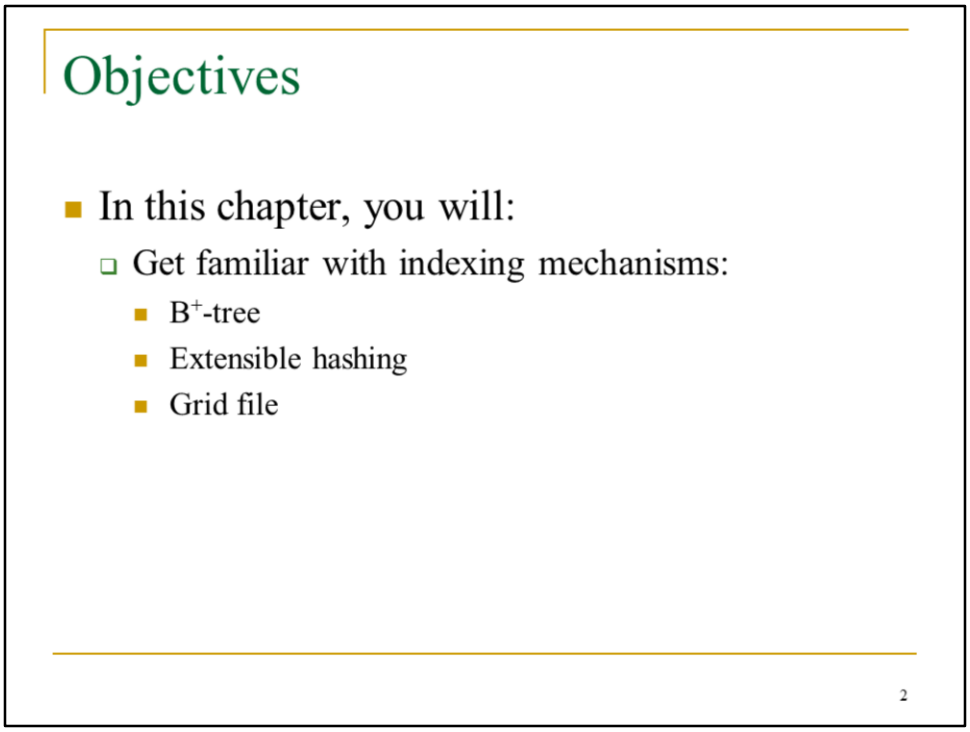
Big Data Analytics

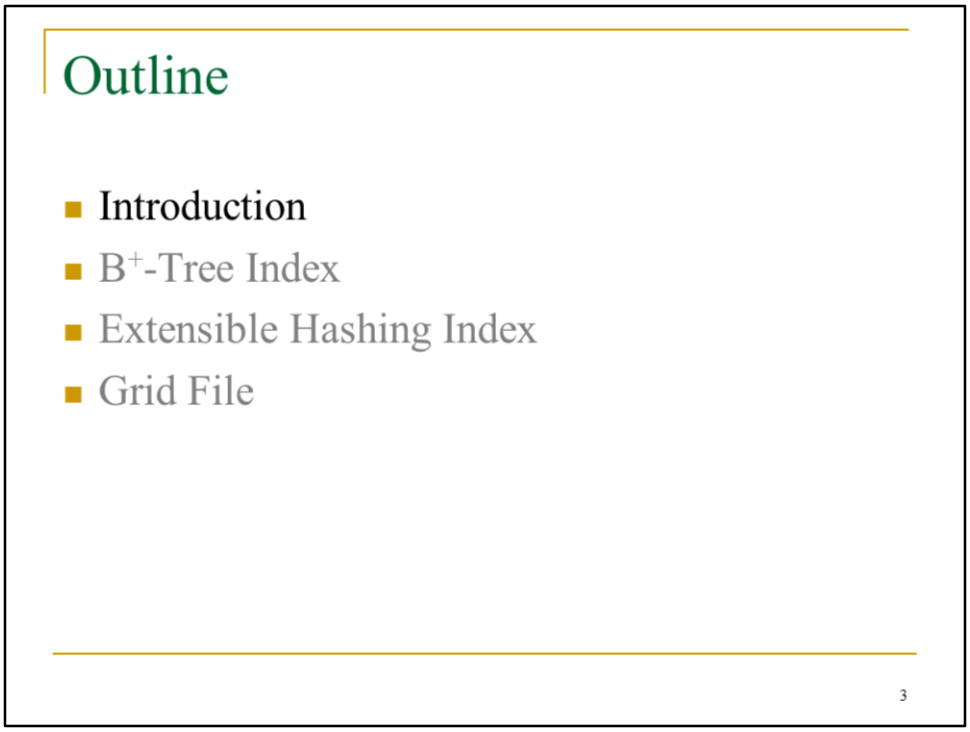
Chapter 3: Indexing Big Data
(Part 1)

1

Chapter 3: Indexing Big Data (Part 1)



In this chapter, we start to discuss the indexing over big data. We will first talk about the indexes such as B+-tree and extensible hashing over classical relational databases. Then, we will discuss the grid file over multidimensional data.



Here is the outline of this chapter. We first introduce some background about indexing, including the reason for using the index. Then, we will talk about the B+-tree and extensible hashing on relational tables. Finally, we discuss grid files for multidimensional data.

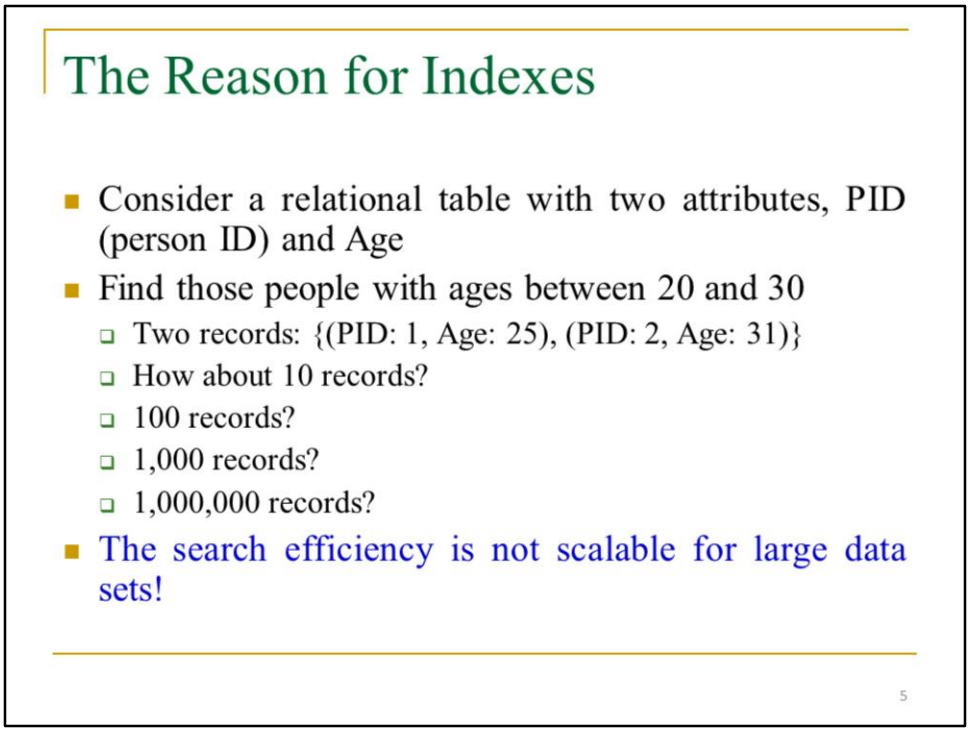
Introduction

- In real applications, we usually collect data of large scale
  - Relational tables with millions of records (tuples)
  - Sensor networks with thousands of sensor nodes, each collecting data such as temperature, humidity, etc. over time
  - In mobile computing or location-based services, millions of mobile users move around in the city
  - …
- Due to the large scale of data, it is very necessary to build indexes for accelerating the search over large data sets

4

In many real applications, we may encounter data of large size. For example, in relational tables, we often have millions of records (or tuples). In real applications of sensor networks, thousands of sensor nodes are usually deployed to different sites, each collecting data attributes such as temperature, humidity, and so on every second, every minute, or every hour. In mobile computing or location-based services, millions of mobile users move around in the city, and their trajectory data can be of large scale.

In order to efficiently process such large-scale application data, we need to build indexes to facilitate the search over large data sets.

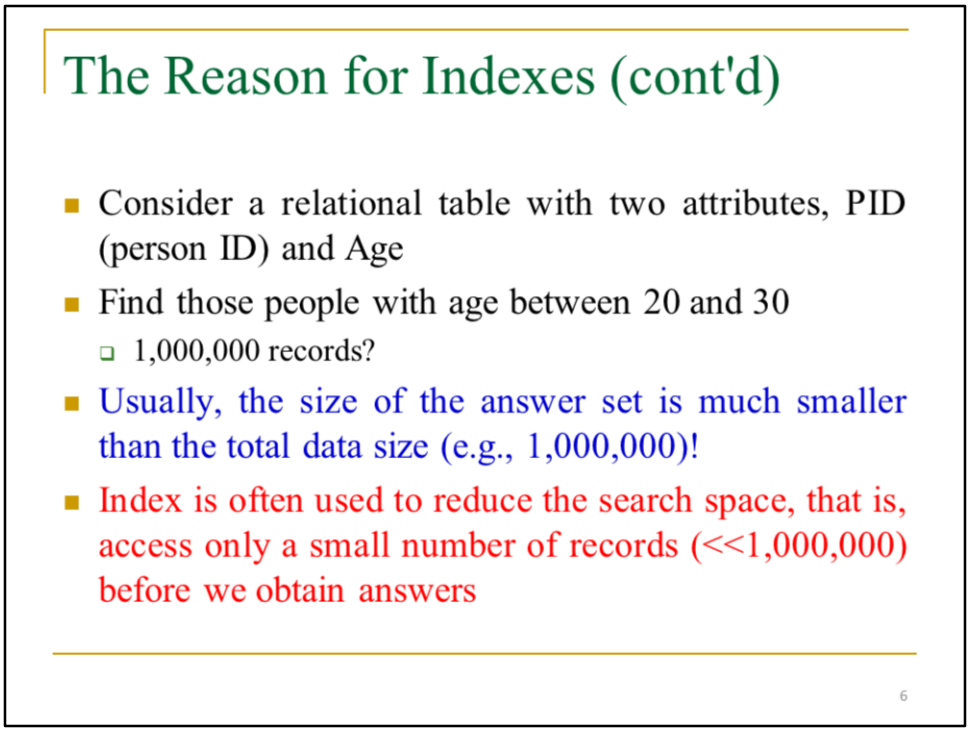


First, let us explain why we need to construct indexes over large data sets. We consider an example of a relational table with two attributes, PID (short for person ID) and Age. Assume that we have a query on this relational table, which finds those people from this table with ages between 20 and 30.

In the case that the table only contains two records: {(PID: 1, Age: 25), (PID: 2, Age: 31)}, we can simply scan these two records, check the query predicate (with respect to Age attribute) for each record, and return the first record as the query answer (since Age 25 is within the range from 20 to 30). If we only scan two records, this is very fast.

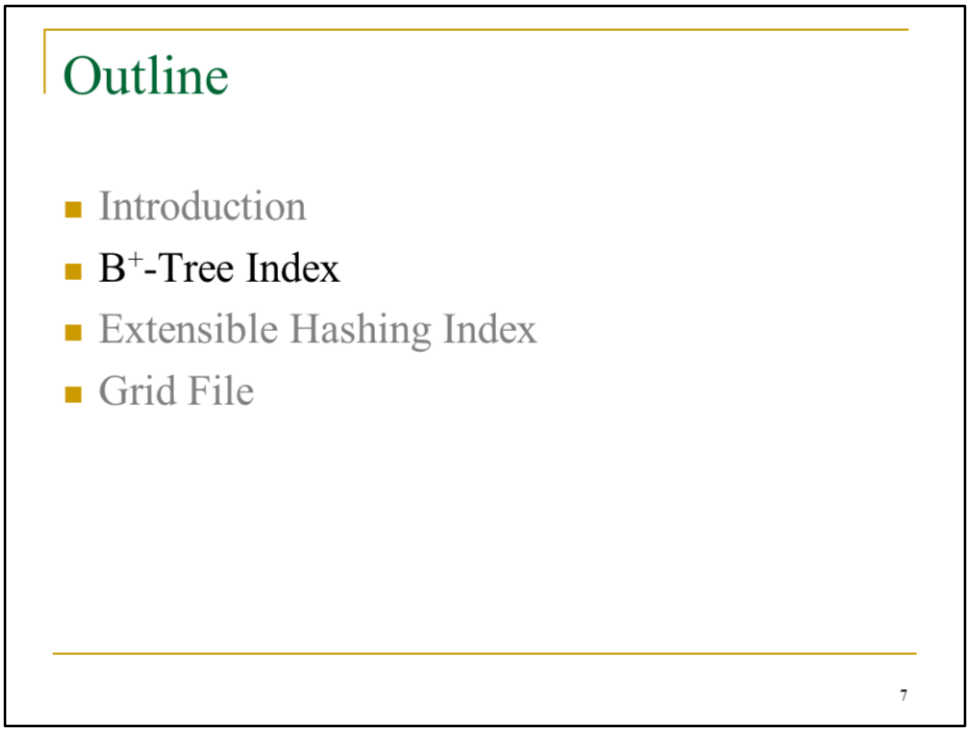
However, what if we have 10 records? 100 records? 1,000 records? Or even 1,000,000 records? Is that efficient to scan so many records now? The answer is NO. For larger and larger tables, the cost of sequentially searching all records in the table will significantly increase.

Therefore, the search efficiency is not scalable for large data sets!

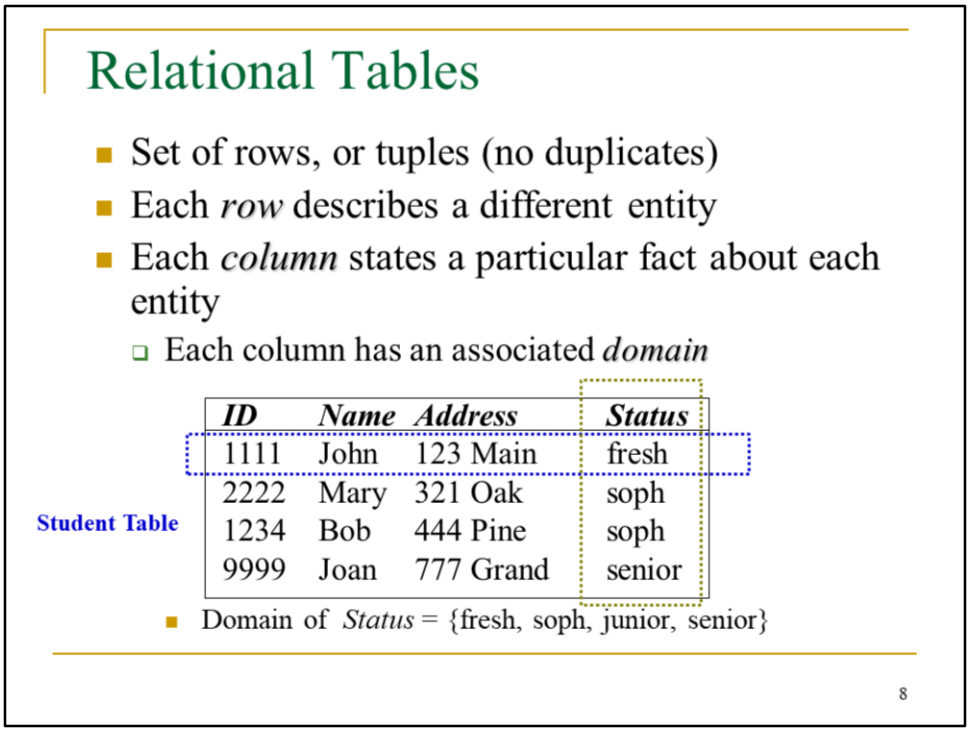


In fact, for normal queries, the size of the query answer set is usually much smaller than the total number of records in the table. In the example we mentioned just now, if the table has 1 million records, the answer set size is usually much smaller than 1M, for example, maybe around 10-100 people with ages from 20 to 30 in the table. Therefore, if we scan the entire table, we are actually wasting a lot of time cost to scan records that are not our query answers.

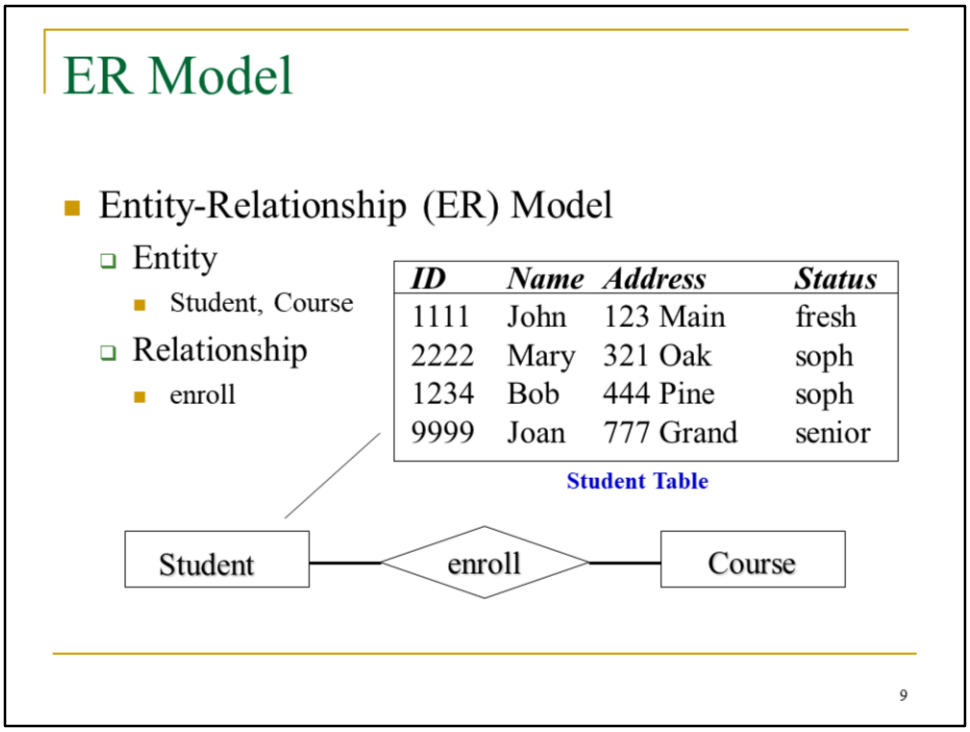
The index is exactly to help reduce the search space within the table. That is, we do not have to scan all records in the table, but with the help of the index, we can directly find some candidate records that might be our query answers. Here, the number of candidate records is much smaller than the table size, for example, 100-200 candidate records. This way, the index can greatly facilitate the query processing with low computation cost.



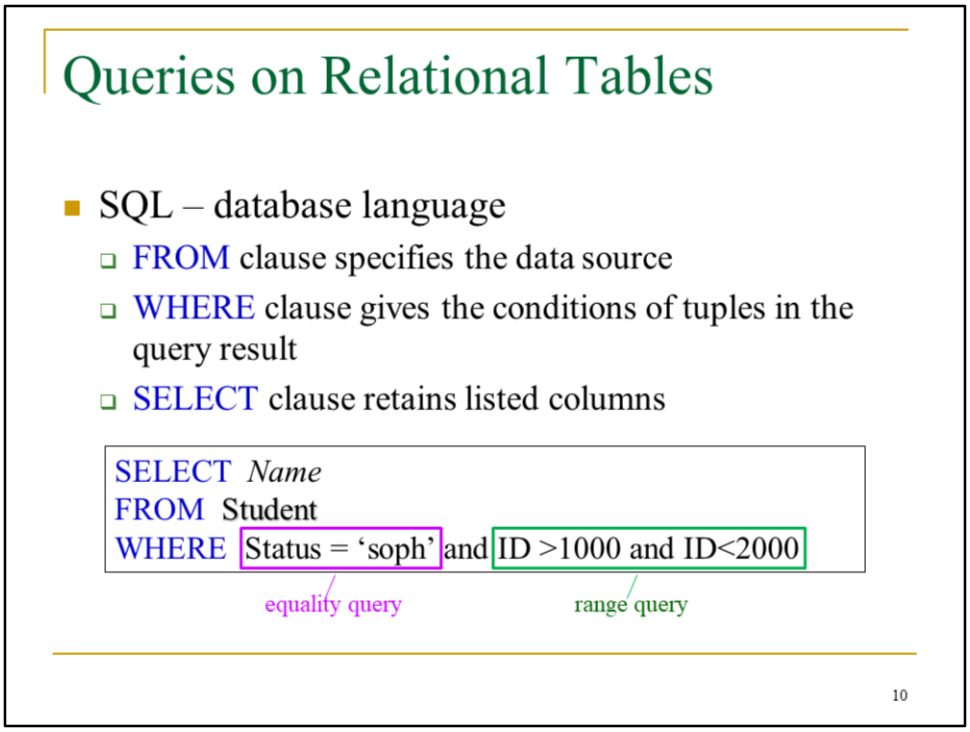
Next, we will talk about indexes on relational tables.



A relational table consists of a set of records (or called rows, or tuples). Each row represents a real-world entity (for example, a student record), and each column represents an attribute (for example, student ID, name, address, or status of a student record). Each attribute has its value domain, for example, student status can be freshman, sophomore, junior, or senior.

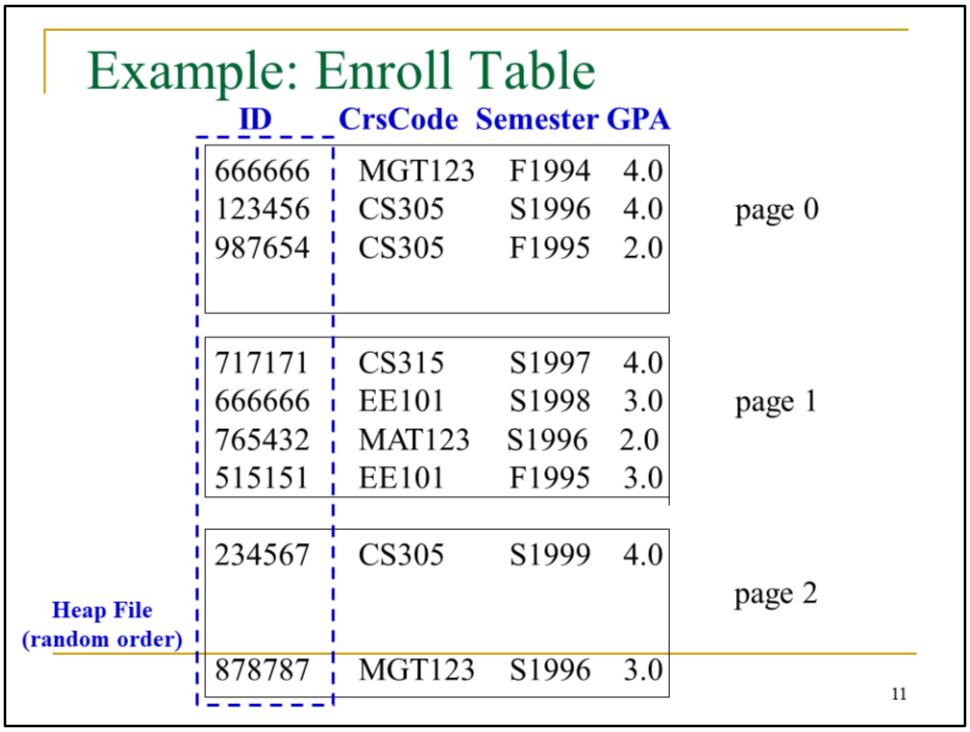


In relational databases, we usually model the real-world entities and their relationships by entity-relationship model (or ER model). Each entity or relationship between two entities can be exactly represented by a relational table, discussed in our previous slide. For example, two entities can be student and course, and their relationship is “enrollment”. In this case, entity Student corresponds to a relational table, and relationship “enrollment” also corresponds to a table.

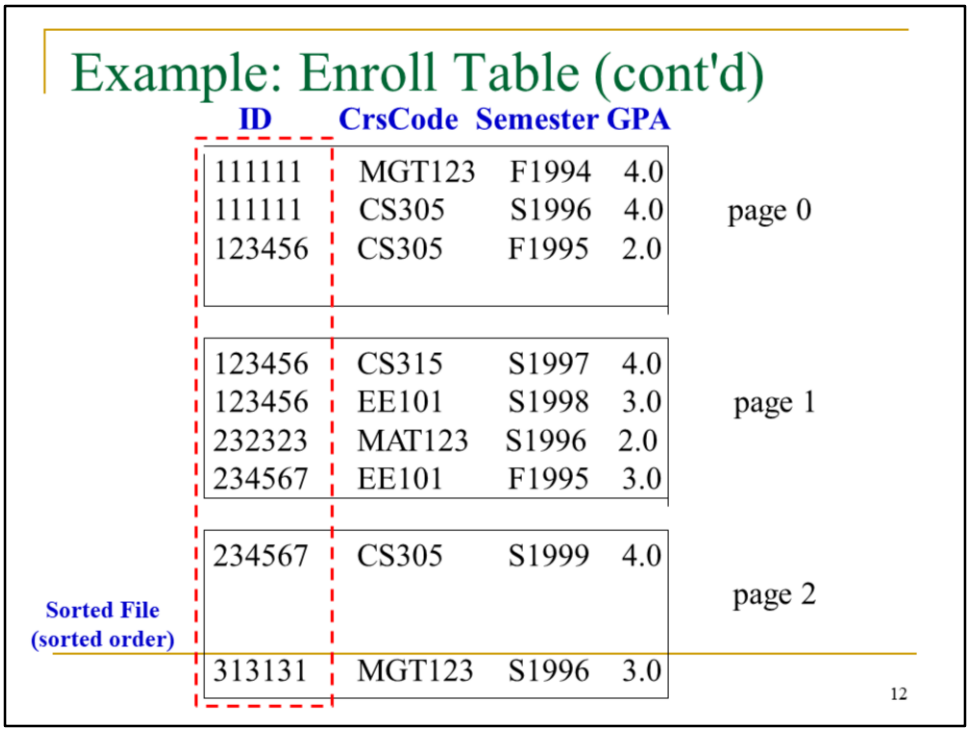


To query relational tables, we usually use SQL queries (that is, structured query language) in relational databases. We have 3 basic clauses, FROM, WHERE, and SELECT clauses. The FROM clause specifies the data sources (that is, table names), the WHERE clause specifies the condition of records we want to return in our query results, and the SELECT clause specifies the projected attributes we want to return in our query result.

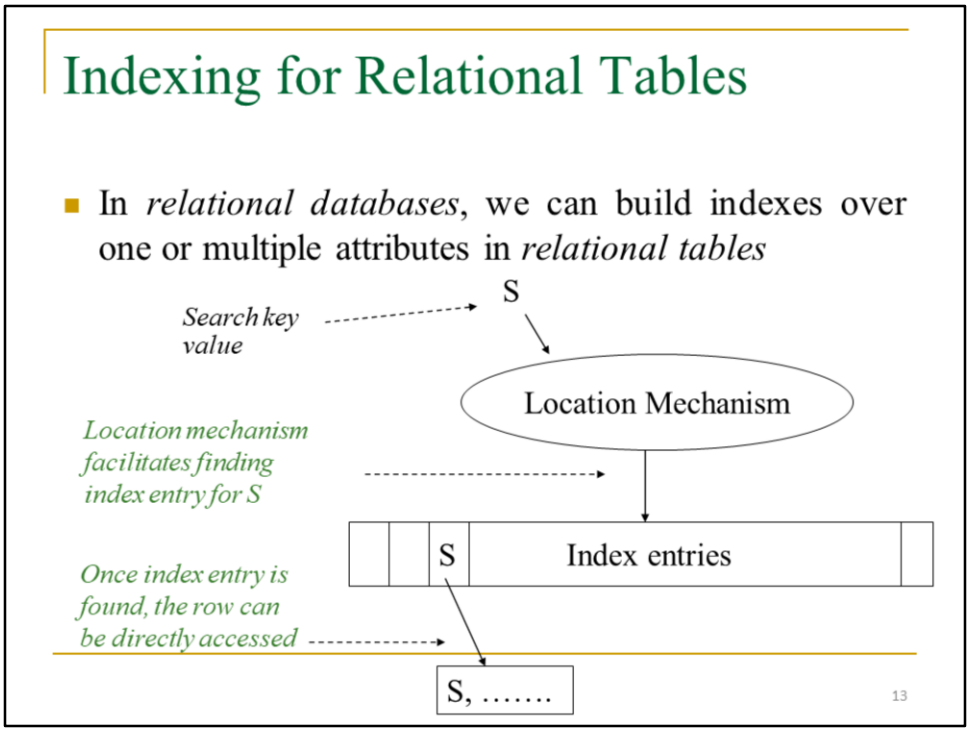
In the example of the slide, we want to find the student name from the Student table, such that the students' status is sophomore and the student ID is between 1000 and 2000 (exclusive).



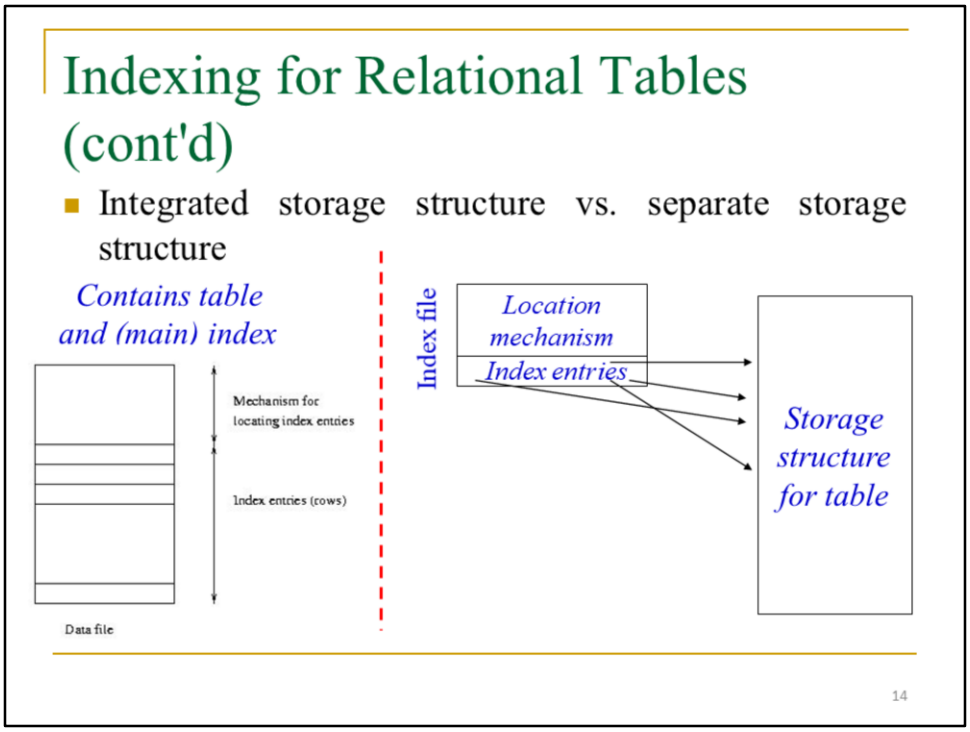
In order to answer SQL queries, one naïve way is to store all the records in the table sequentially on disk pages. This is called heap file (without any indexes). In this example, the enroll table contains records in random order on 3 disk pages. To answer an SQL query, we have to sequentially scan all records on 3 pages in this heap file.



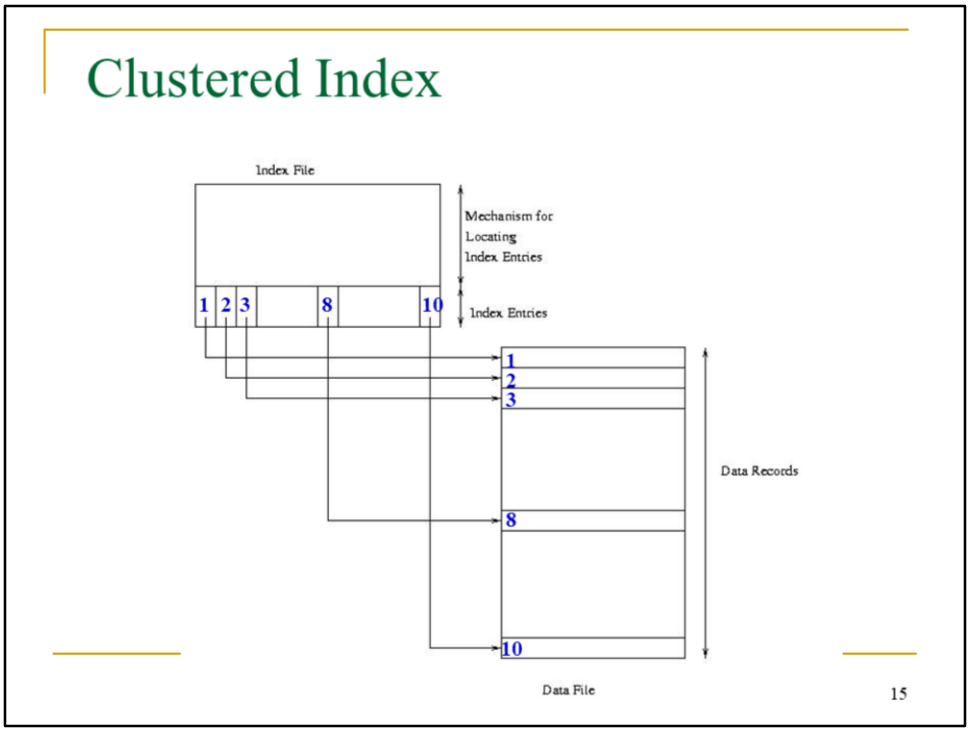
On the other hand, we can also organize records in the enroll table as a sorted file, for example, sorted in ascending order of the student ID. In this case, with a sorted file (still without indexes), we can conduct a binary search, with fewer I/Os, that is, log(n) I/Os, where n is the total number of disk pages in the sorted file. However, when the number of disk pages n is large, log(n) search cost is still very high.



The basic idea of using indexes for relational tables is to enable a quick access or a shortcut to the candidate answer in the relational table. For example, given a search key value, with some location mechanisms, we can quickly find index entries that may contain query answers, and then directly access records pointed by those index entries. In this way, we do not have to scan the entire table or data set.

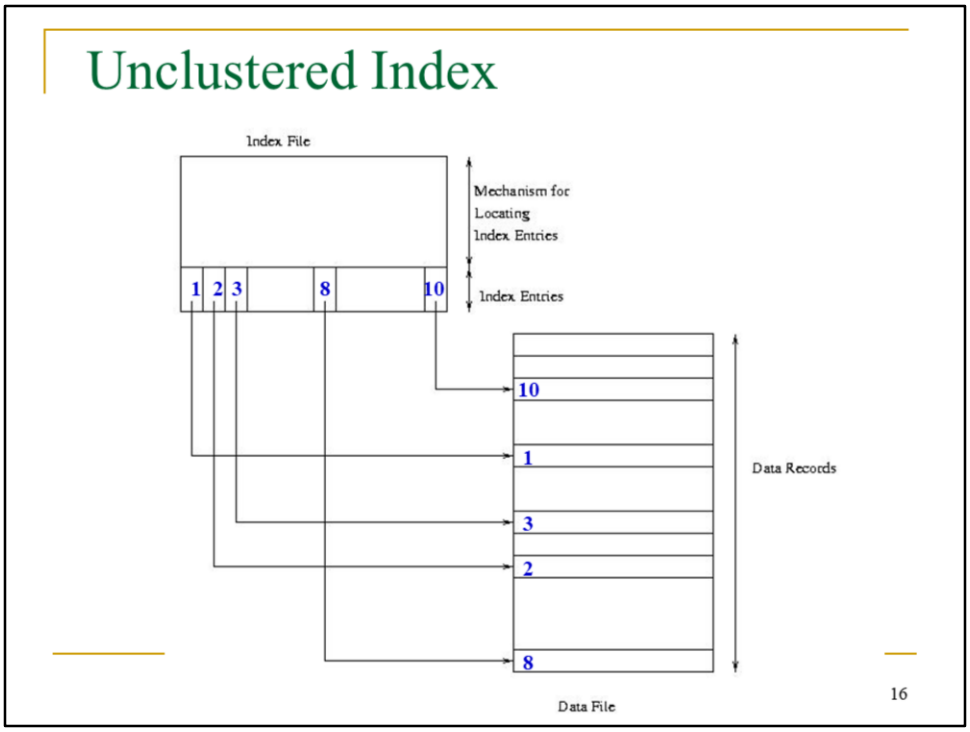


We can classify the indexes according to different criteria. If we consider the storage locations of index and data file, we can classify the indexes into 2 categories, integrated storage structure and separate storage structure. For the integrated storage structure (as shown on the left figure), both index and data file are stored in a single file. On the other hand, as shown on the right figure, the index and data file (that is, the table) are stored separately in two different files, which is called separate storage structure.



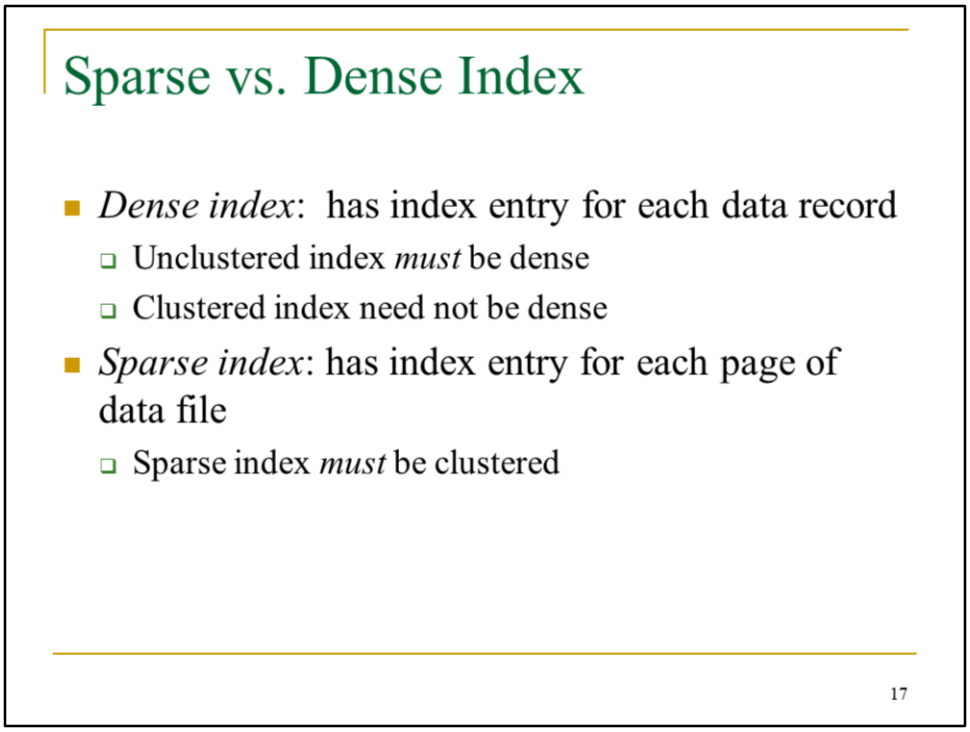
If we consider the orders of index entries and data records, then we can classify the indexes into clustered and unclustered indexes. In this figure, you can see the order of index entry is the same as that of the data records. We call such an index clustered index.

The clustered index is I/O efficient for range query. For example, if we want to search for data records with attribute between 2 and 8, then we only need to go to index entry 2, find the data record pointed by index entry 2, and do a sequential scan in the data file.



In contrast, in this figure, the order of index entries are different from that of data records in the data file. We call such an index unclustered index.

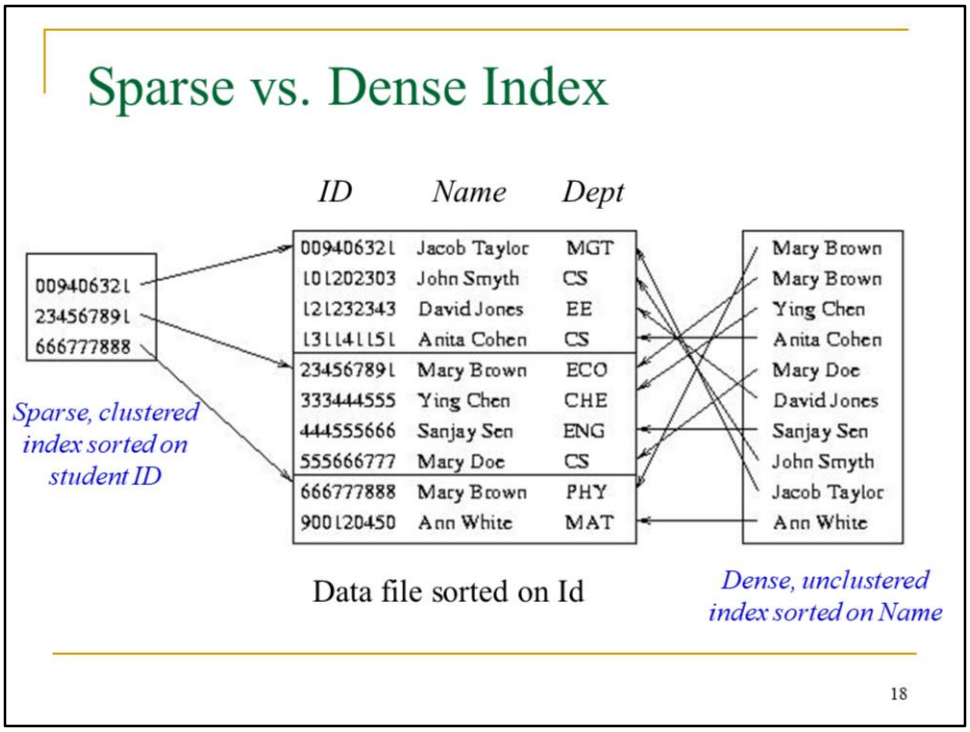
Due to different orders between index entries and data records, if we want to search for records with attribute from 2 to 8, we have to use 7 pointers in index entries and find the data records pointed by these pointers. These records might be located at different disk pages, which incurs higher I/Os compared with the clustered index.



According to the sparseness of index entries, we can also classify the indexes into dense and sparse indexes. For dense index, every data record (or records with the same attribute value) is usually pointed by an index entry. For sparse index, each index entry is usually pointing to the first record in a disk page of the data file.

Unclustered index must be a dense index, since index entries and data records have different orders. But clustered index is not necessary to be a dense index.

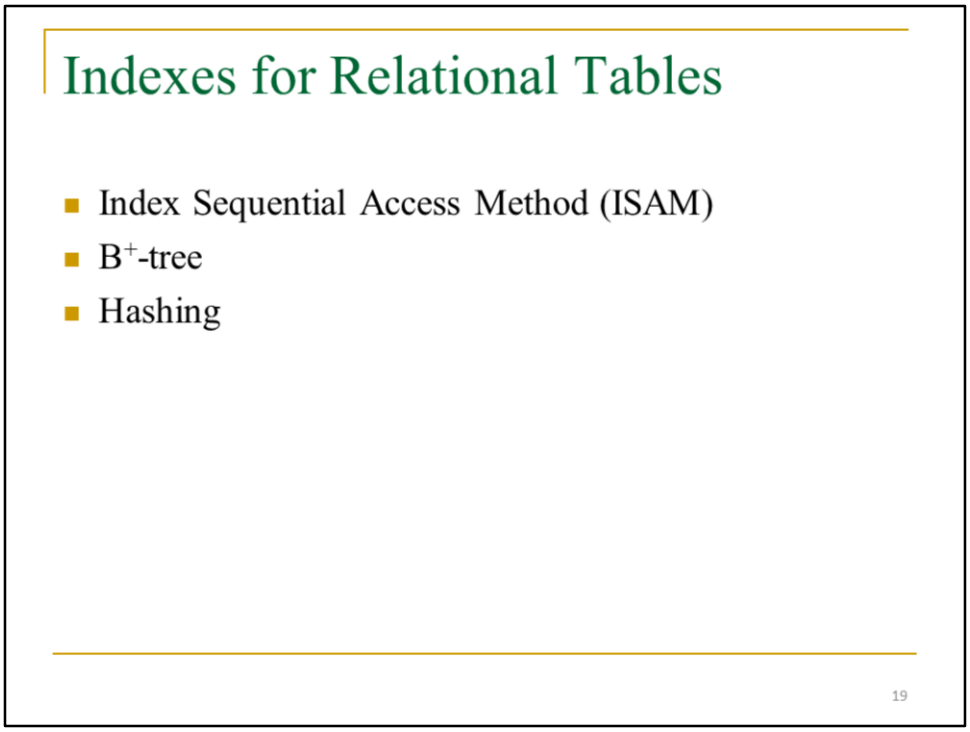
Similarly, sparse index must be a clustered index.



As an example, the table in the middle contains records of students in a data file, which are stored on 3 disk pages. You can see all the records are sorted on student ID.

The left table is an example of sparse index on student ID, which contains 3 index entries, each pointing to the first record on a disk page. This sparse index is also a clustered index, since the order of index entries is the same as that of data records.

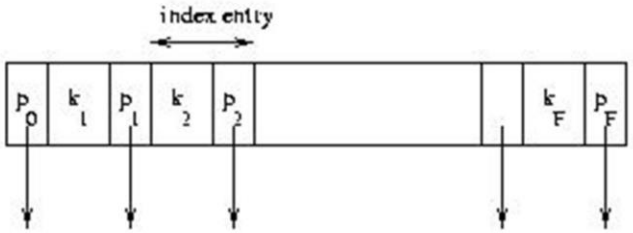
On the other hand, the right table is a dense index on attribute “student name”, where each student record is pointed by an index entry. Since the orders of index entries and student names in the data file are different, this index is also an unclustered index.



Next, we will talk about classical indexes on relational tables in relational databases, including index sequential access method (or ISAM for short), B^+-tree index, and extensible hashing.

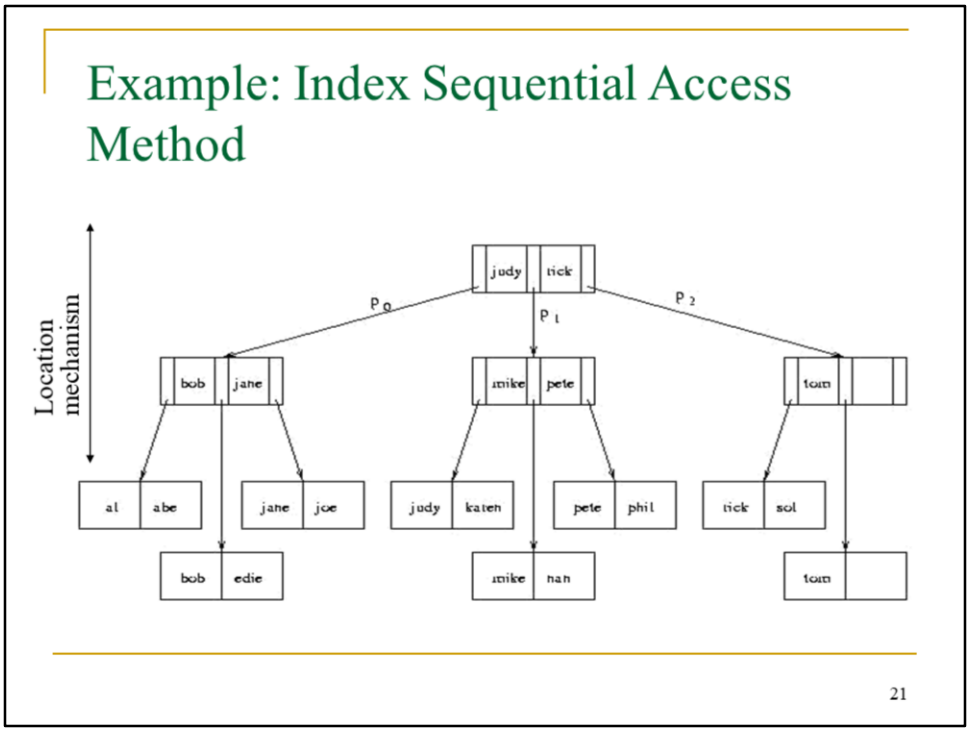
Index Sequential Access Method (ISAM)

- Generally an integrated storage structure
  - Clustered, index entries contain rows
- Separator entry = (k_i , p_i); k_i is a search key value; p_i is a pointer to a lower level page
- k_i separates set of search key values in the two subtrees pointed at by p_{i-1} and p_i.



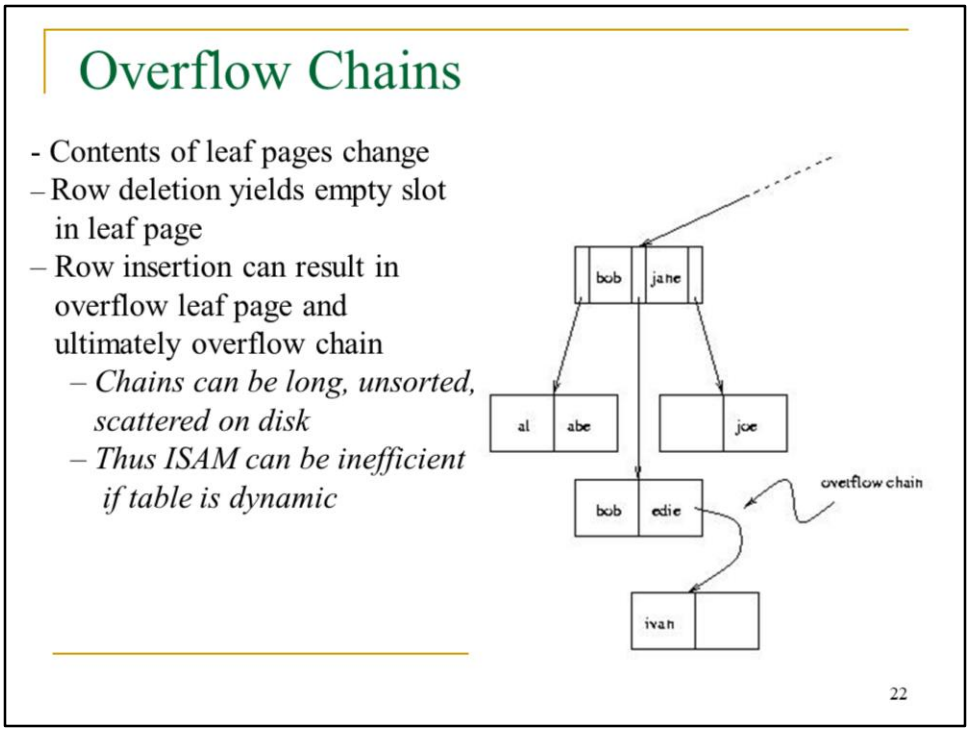
20

For the index sequential access method, we usually have index entries in the form (k_i, p_i), where k_i is a search key and p_i is a pointer pointing to a lower level index page or the location of the record. In this figure, we have F search keys and (F+1) pointers.

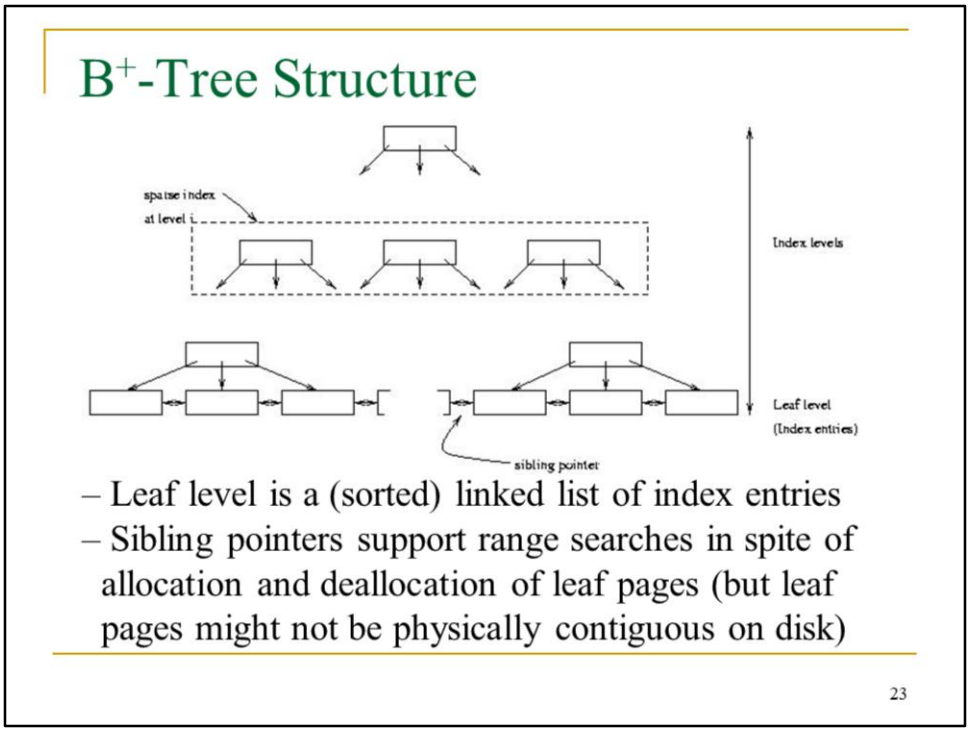


Here is an example of the index sequential access method. The index is a hierarchical tree structure, where leaf nodes store actual data records. Each index page stores 2 search keys and 3 pointers. In this example, the search keys are strings, therefore, keys in each index page are sorted in alphabetical order.

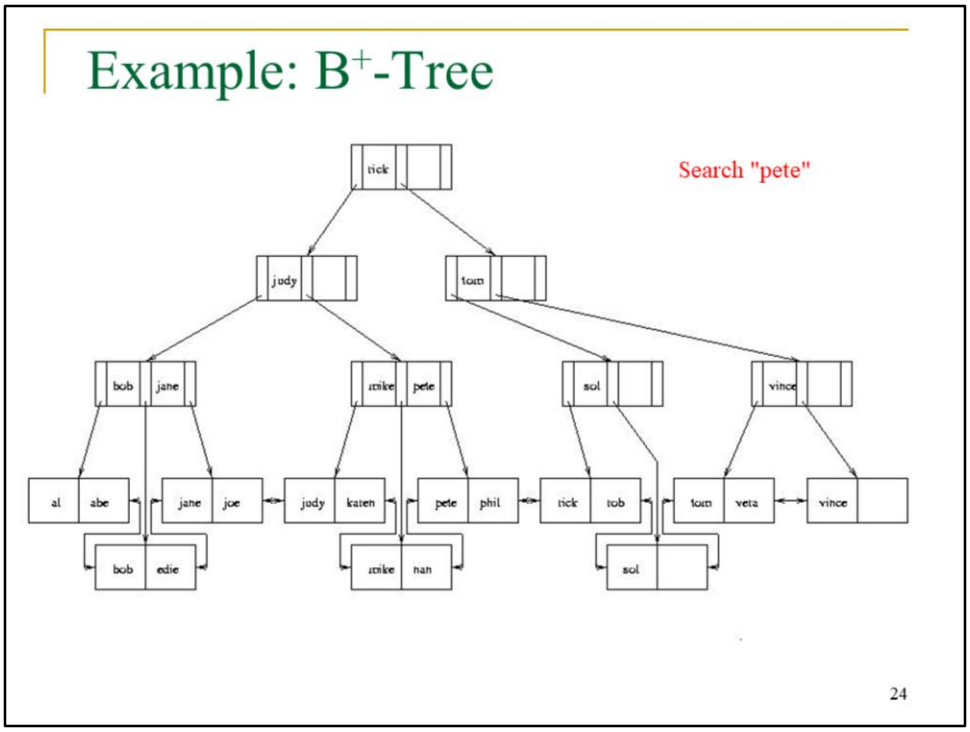
For example, if we want to search for “pete”, we first start from the root of the tree and compare “pete” with search keys “judy” and “rick” in the root. Then, we found that “pete” should be after “judy” and before “rick”, and we descend to the next index level through pointer p_1. Next, we found that “pete” is the same as the second search key, and we select the right branch of the search key “pete”, which points to a leaf node containing the data record “pete”. This way, we can use ISAM to enable fast search of particular record, where the search cost is the height of the ISAM tree.



However, ISAM is a static index structure. It has the disadvantage that it cannot well support dynamic updates of data records. For example, if we keep inserting more and more records into a leaf node. One possible remedy is to add a chain of overflow pages when the disk page of the leaf node is full. This potentially increases the height of the tree structure, which in turn increases the search cost.

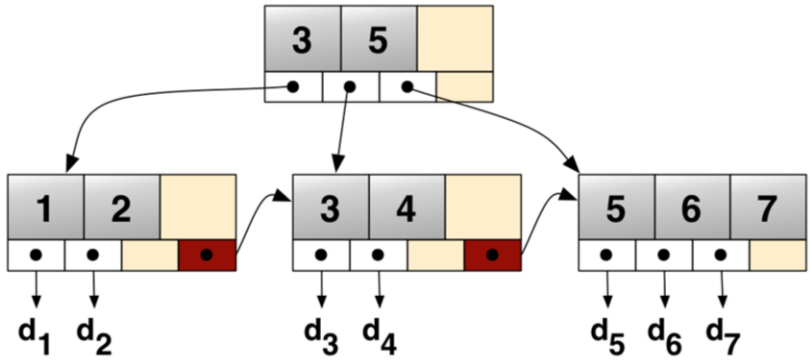


The main reason for the disadvantage of ISAM is that, it cannot support dynamic updates of the tree index to keep the height of the tree as small as possible. B+-tree is exactly an index that dynamically maintains the tree structure upon insertions and deletions, such that the heights from root to all leaf nodes are the same. Moreover, there is a double-linked list among leaf nodes which can support range searches.



Here is an example of a B+-tree. To search a keyword “pete”, we can start from the root and descend to a leaf node that may contain “pete”. First, “pete” is alphabetically before “rick” in the root, so we descend to the left branch of “rick”. Then, since “pete” is after “judy”, we descend to the right branch of “judy”. Next, we descend to the right branch of “pete”. Finally, we find “pete” in the leaf node. The search cost is proportional to the height of the B+-tree, that is, the number of nodes encountered from root to a leaf node.

Example: B$^+$-Tree (cont'd)



https://en.wikipedia.org/wiki/B%2B_tree

25

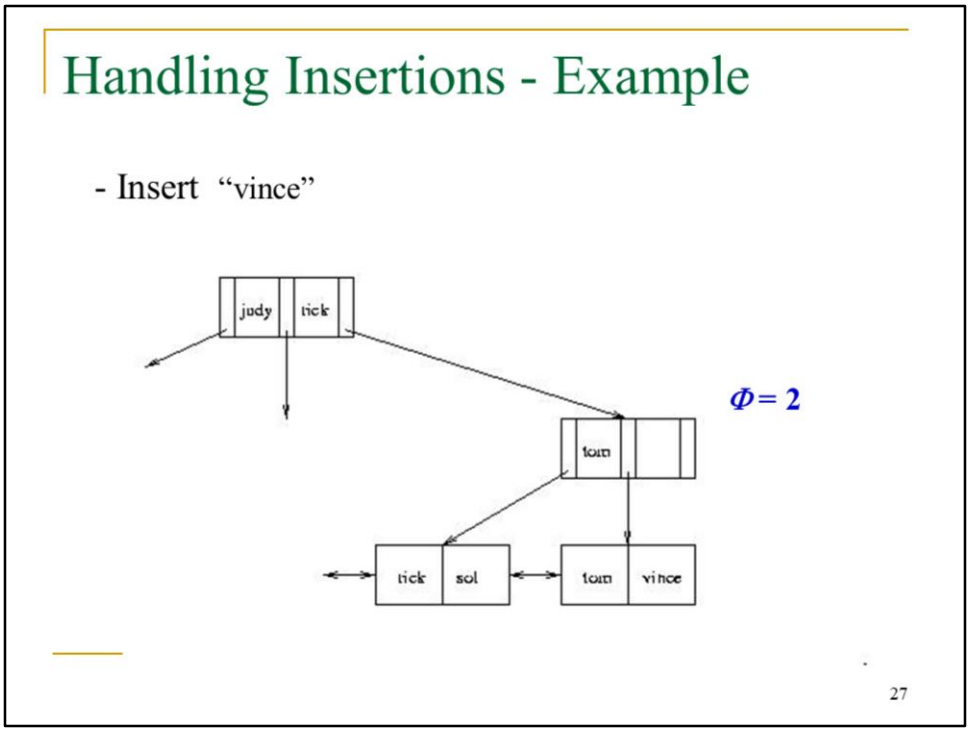
B+-tree can be built not only on string data, but also on numeric data. Here is another example of the B+-tree over a numeric attribute.

Insertion and Deletion in B^+ Tree

- Structure of tree changes to handle row insertion and deletion – *no* overflow chains
- Tree remains *balanced*: all paths from root to index entries have same length
- Algorithm guarantees that the number of separator entries in an index page is between $\Phi/2$ and Φ
  - Hence the maximum search cost is $log_{\Phi/2}Q + 1$ (with ISAM search cost depends on length of overflow chain)

26

B+-tree has the property that the tree is a balanced tree, that is, all paths from root to leaf nodes have the same length. Different from ISAM, there is no overflow page for leaf node. When the leaf node is full, we need to update the B+-tree (rather than adding overflow pages) in order to guarantee the balance of the tree. Also, the number of entries in each node of the B+-tree is kept between half full and full.

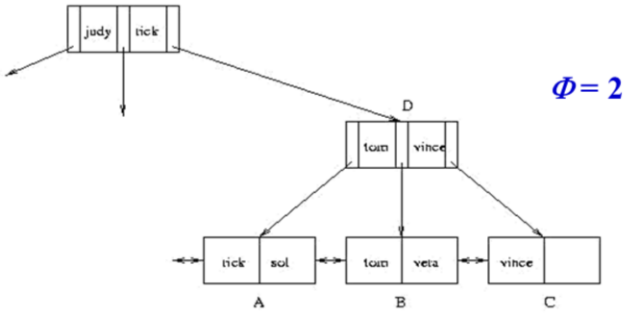


We will see how to update the B+-tree upon insertions first. In this example, we want to insert "vince". We will first find the location to insert this new record, in a way very similar to searching a record "vince" from the B+-tree. Then, if the leaf node is not full, we can directly insert this record "vince" into this leaf node, as shown in this figure.

Next, we will insert a record "vera" into this tree structure. We can see that we will insert this record into a full leaf node containing "tom" and "vince". In this case, B+-tree will dynamically split this full node into two nodes.

Handling Insertions (cont'd)

– Insert "vera": Since there is no room in leaf page:
  1. Create new leaf page, C
  2. Split index entries between B and C (but maintain sorted order)
  3. Add separator entry at parent level



$\Phi = 2$

28

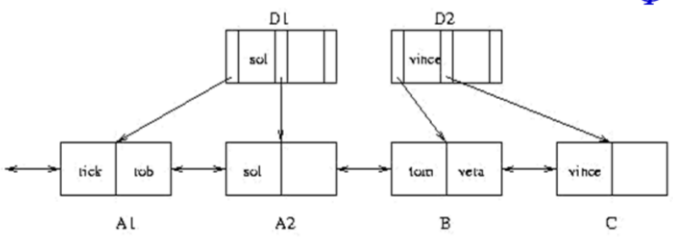
For example, split the full leaf node into nodes B and C, and at the same time maintain non-leaf nodes correspondingly.

Similarly, if we want to insert another record "rob", we first find the location, node A, to insert the new record "rob". Since A is full, we again split this node into two, A1 and A2, and update their ascendants such as node D.

Handling Insertions (con't)

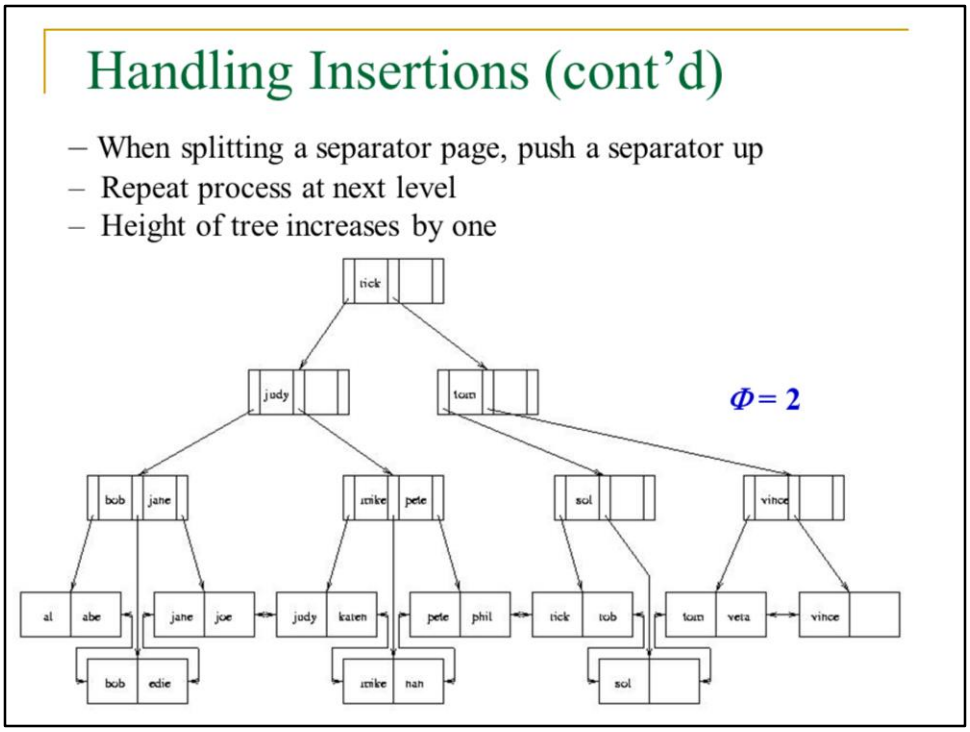
– Insert "rob". Since there is no room in leaf page A:
  1. Split A into A1 and A2 and divide index entries between the two (but maintain sorted order)
  2. Split D into D1 and D2 to make room for additional pointer
  3. Three separators are needed: "sol", "tom" and "vince"

$\Phi = 2$



29

As shown in the figure, we also split non-leaf node D into D_1 and D_2. This way, we will propagate the updates upwards.



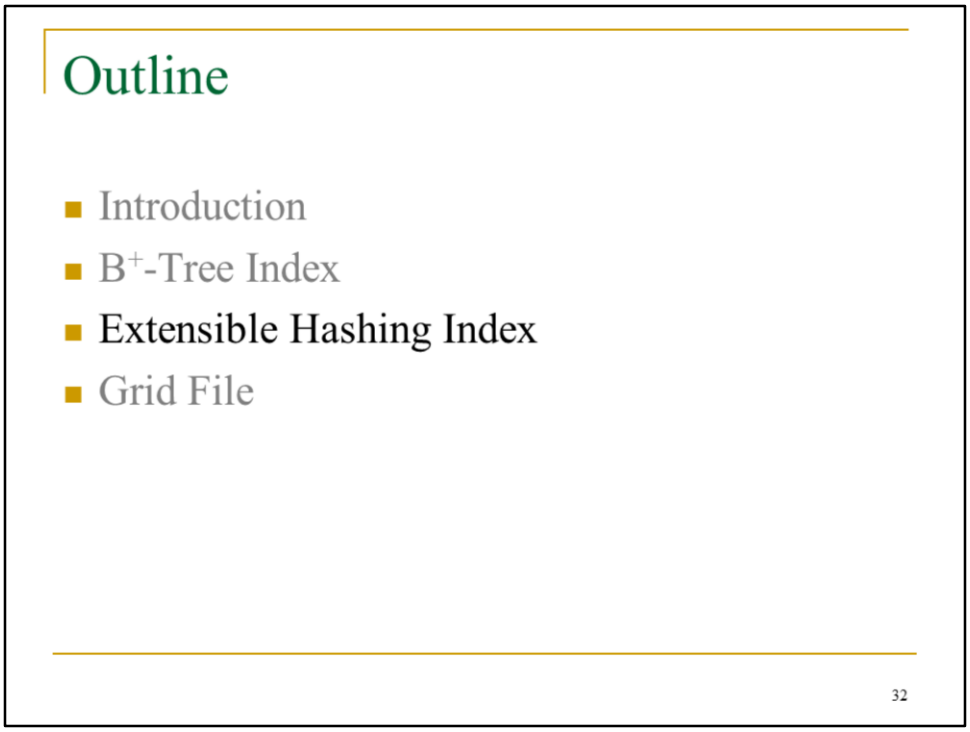
In the worst case, we will split the full root of the B+-tree into 2 nodes, and create a new root with two pointers pointing to two split nodes (from old root), which increase the height of the tree by 1.

Handling Deletions

- Deletion can cause page to have fewer than $\Phi/2$ entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
  - See book
- In practice, tables generally grow, and merge algorithm is often not implemented
  - *Reconstruct tree to compact it*

31

The deletion of data records may cause the B+-tree to have the underflow problem. In this case, B+-tree will merge the underflowed node with its sibling. Here, we will not talk about B+-tree update details with deletions.



Recall that the search cost of B+-tree is proportional to the height of the B+-tree. Next, we will talk about extensible hashing index with O(1) search cost.

Hash Index

- Index entries partitioned into buckets in accordance with a *hash function*, $h(v)$, where v ranges over search key values
- Each bucket is identified by an address, a
- Bucket at address a contains all index entries with search key v such that $h(v) = a$
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

33

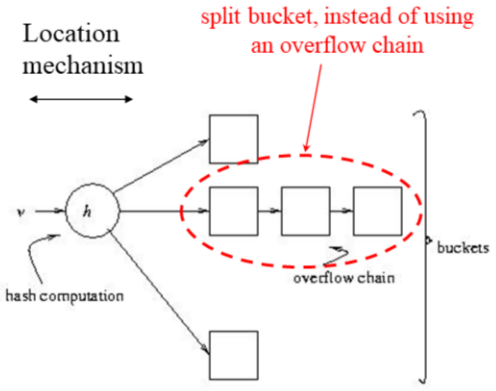
In particular, a hash index uses a hash function h(v) to hash the key, v, of each data record to a bucket address a. Each bucket corresponds to a disk page, storing data records hashed to this bucket. Note that, if a bucket is full, we need to add a chain of overflow pages.

Equality Search with Hash Index

Given v:

1. Compute $h(v)$
2. Fetch bucket at $h(v)$
3. Search bucket

Cost = number of pages in bucket (cheaper than B^+ tree, if no overflow chains)



34

Once we build a hash index over multiple data records , to search for a particular record with key v (i.e., equality search), we can use the hash function h(v) to find the location, h(v), of the bucket, and then sequentially scan all records in the bucket (including its overflow chain).

This hash index is good for equality search, but not for range search like B+-tree. Usually, the cost of equality search via hash index is O(1.2). This estimated O(1.2) is due to the overflow chain of the bucket.

Next, we will discuss one particular hash index, extensible hashing index, which does not require the overflow pages.

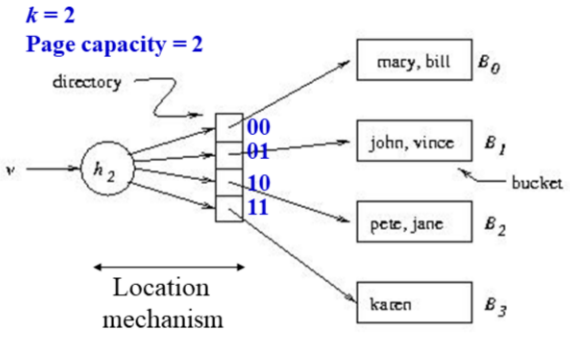
Extensible Hashing

- Example:
  - family of hash functions based on h:
    - $h_k(v) = h(v) \mod 2^k$ (use the last k bits of $h(v)$)
    - At any given time a unique hash, h_k, is used depending on the number of times buckets have been split

35

Assume that we have a random hashing function h_k(v), which is given by h(v) modulus 2^k. Here, 2^k is the number of buckets. Initially, we can set k to 1 or 2 (i.e., 2 or 4 buckets, respectively). Later on, if the bucket is full, instead of adding overflow pages to the full bucket, we will increase/extend the number of buckets to 2^{k+1}. This is why the hashing index is “extensible”.

Example: Extendable Hashing



v	$h(v)$
pete	11010
mary	00000
jane	11110
bill	00000
john	01001
vince	10101
karen	10111

Extendable hashing uses a directory (level of indirection) to accommodate family of hash functions

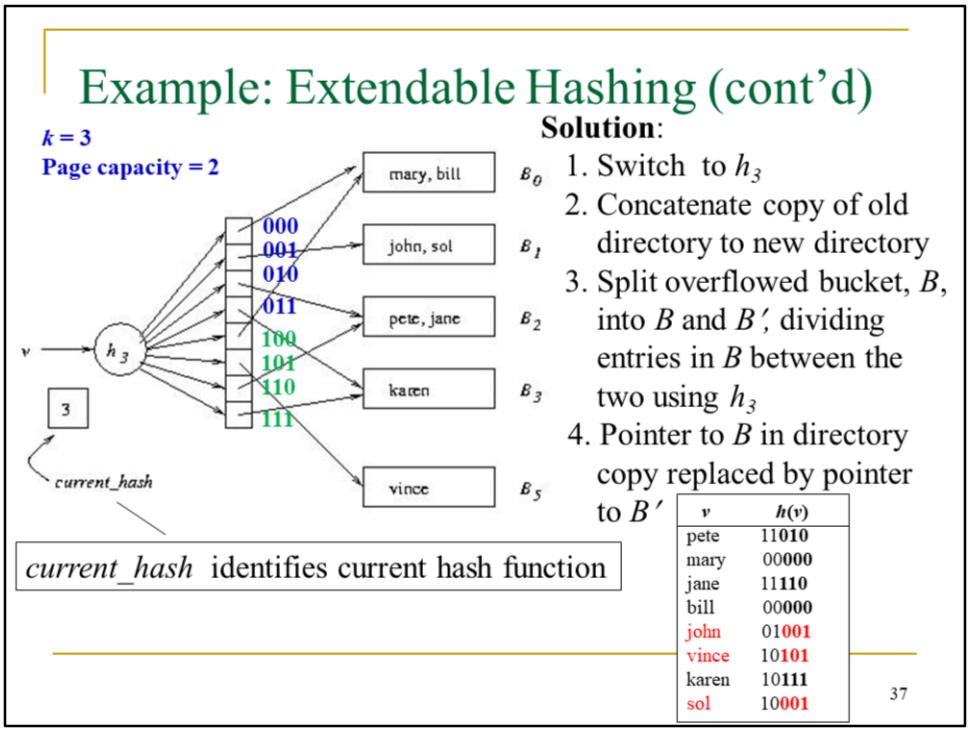
Suppose next action is to insert sol, where $h(sol) = 10001$.

Problem: This causes overflow in B_1

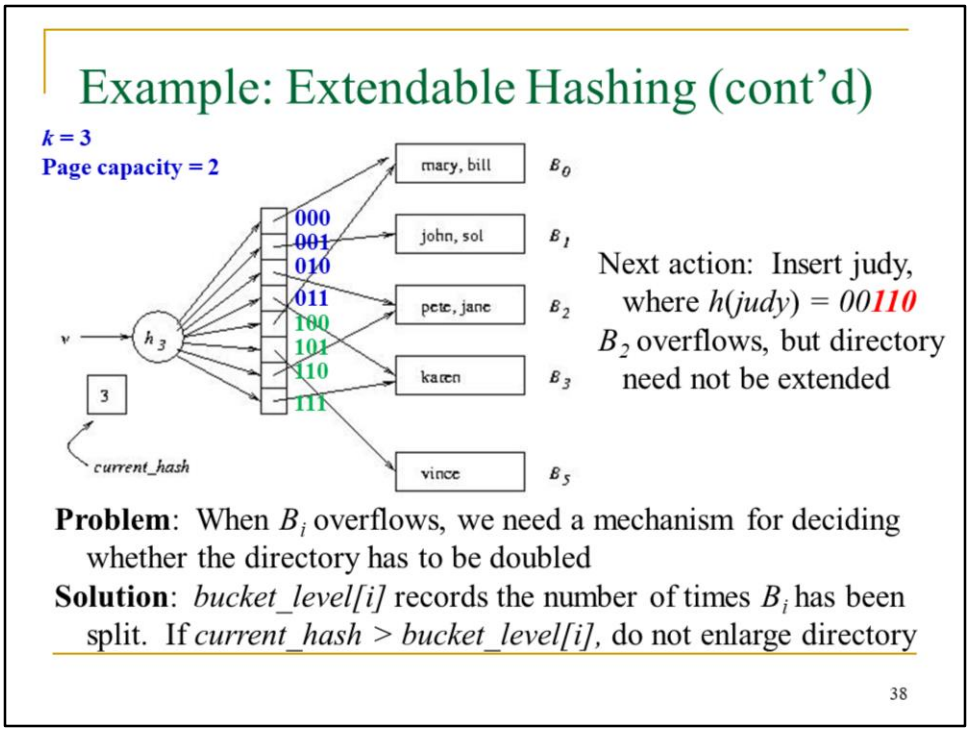
36

Here is an example of the extensible hashing, where k = 2 (i.e., 4 buckets) and page capacity is 2 (i.e., each bucket can store a maximum of two records). In this example, there are 4 buckets, with No. 00, 01, 10, and 11. The hashing function h(v) hashes each string value v into a sequence of 0's and 1's. The h_k(v) hashing function will only take the last two (or k) bits as the bucket No. For example, "pete" has the binary hashed value 11010, which is hashed to bucket 10 (the last two bits of the hashed value). Therefore, "pete" is hashed to bucket 10 (i.e., bucket B_2 in the figure).

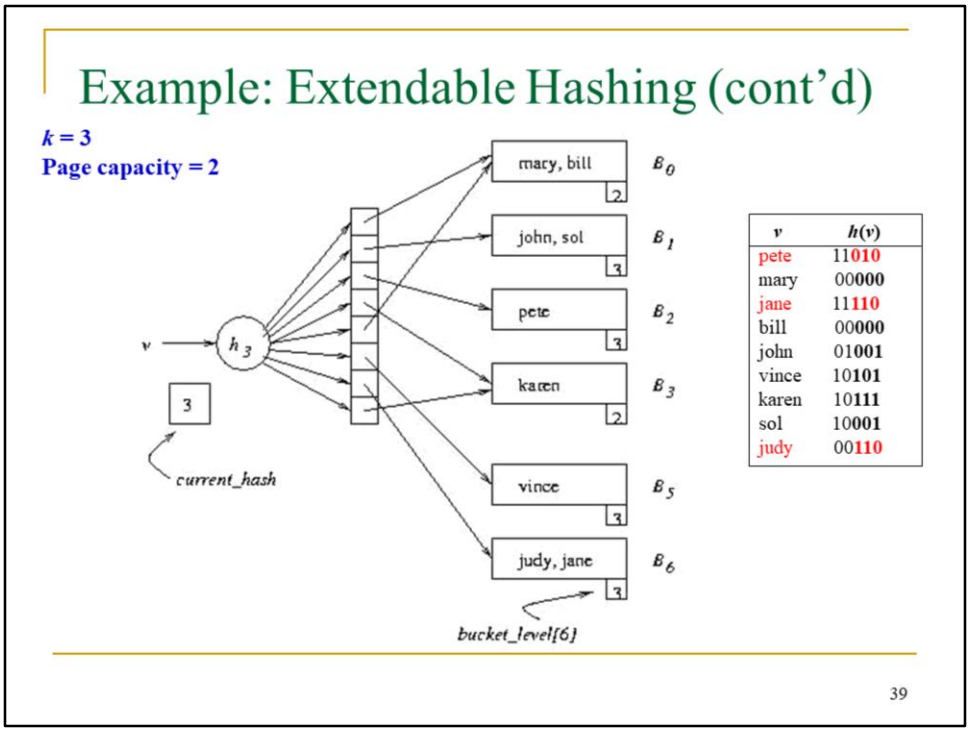
Now we consider inserting "sol" into the extensible hashing. Since the hashed bucket number is 01 (the last two bits of h(sol)), we aim to insert "sol" into bucket B_1. However, bucket B_1 is full and cannot store more records. In this case, we need to extend or double the hashing directory.



Specifically, we increase k by 1, and duplicate the hashing directory (including pointers as well). As shown in the figure, we double the directory to 8 buckets, with bucket No. from 000 to 111. Pointers in bucket No. 000 and 100 are pointing to the same bucket B_0. Due to the overflow of bucket B_1, we create one new bucket B_5, let pointers 001 and 101 point to two buckets B_1 and B_5, respectively, and redistribute records between these two buckets. Here, for the extended hashing index, we will consider the last 3 bits of the hashed values (instead of 2 bits before the expansion).



Similarly, if we want to insert "judy" into the hashing index, we hash it into bucket 110 (or bucket B_2). Since bucket B_2 is full but pointed by two directory entries 010 and 110, in this case, we do not need to double the hashing directory, and only need to create a new bucket B_6 and redistribute records between B_2 and B_6.



Here is the result after the insertion of "judy".

For the detailed implementation, to determine if there are two directory entries pointing to the same bucket, the extensible hashing index maintains a global variable, current_hash, which is equal to k, and a local variable for each bucket, which is also initially set to k. If a bucket is split, then the local variable of this bucket will be increased by one.

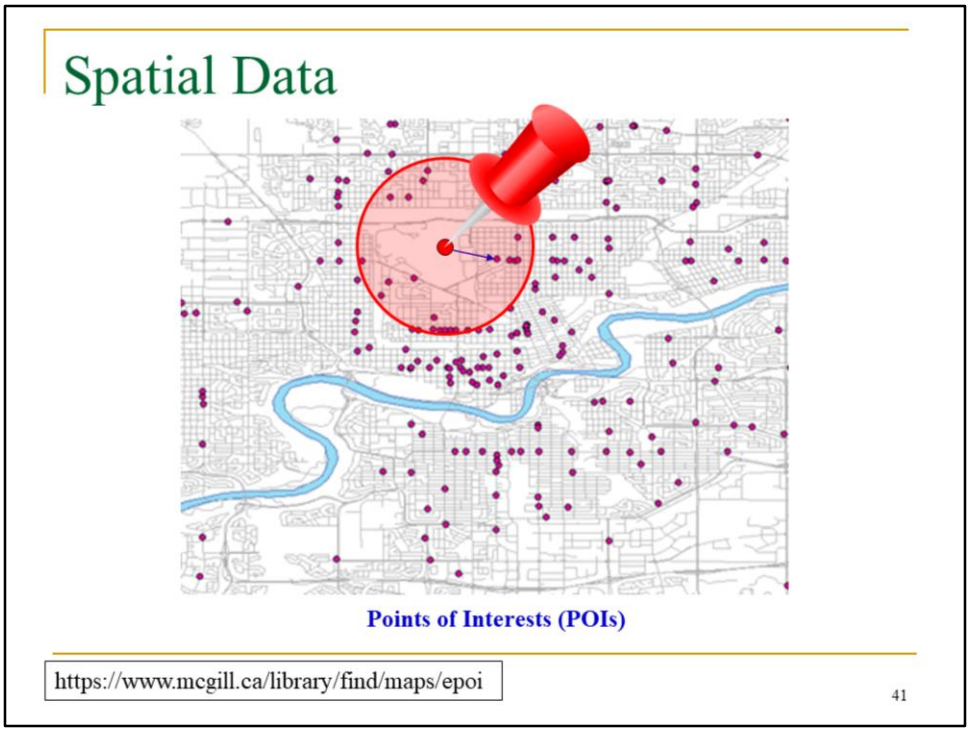
When the global variable is greater than the local variable of a bucket, we do not need to double the hashing directory (since two pointers are pointing to this bucket). When the global variable is equal to the local variable of a bucket, we have to double the hashing directory.

Outline

- Introduction
- B^+-Tree Index
- Extensible Hashing Index
- Grid File

40

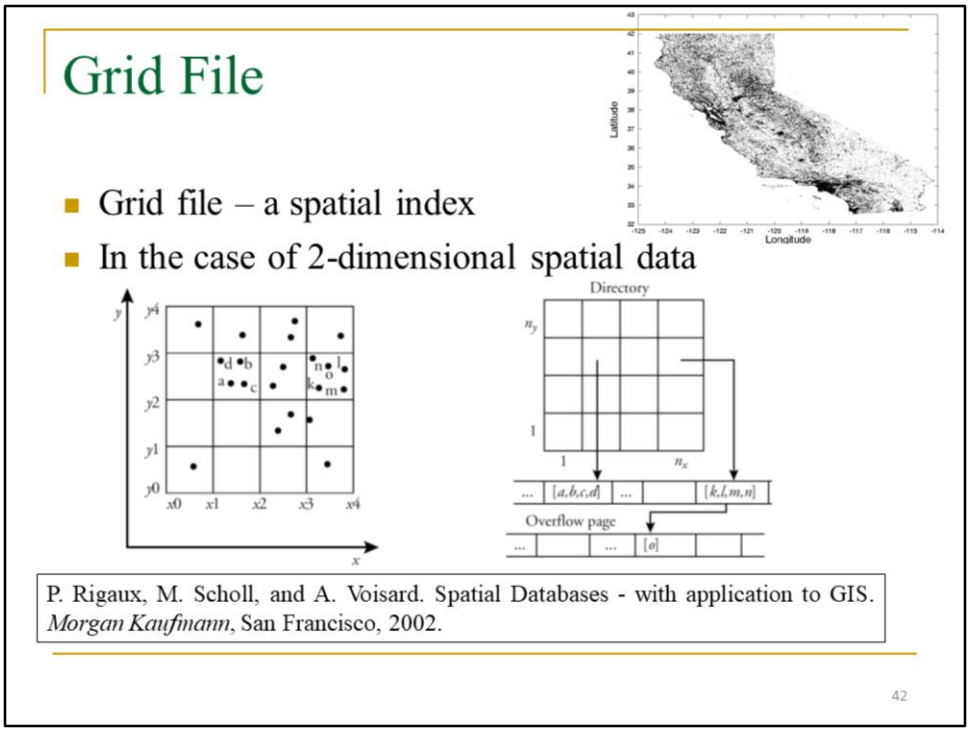
Next, we will discuss the grid file for multidimensional data.



https://www.mcgill.ca/library/find/maps/epoi

Consider a map of a city, in which there are many points of interest (POIs) such as restaurants, hotels, airports, and so on. Each POI is associated with its 2-dimensional spatial location (Latitude, Longitude). For spatial data, the dimensionality can be even higher such as 3D, 4D, or in general d-dimensional data.

With such spatial data, we can issue many classical queries such as the range query or nearest neighbor query. Given a query point (e.g., a query issuer‘s current location), we may want to find all restaurants within 10 miles (i.e., range query) or the restaurant closest to the query point (i.e., nearest neighbor query).



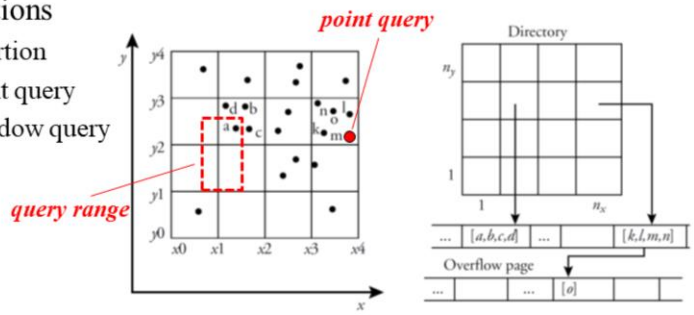
https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm

Rigaux, P., Scholl, M., and Voisard, A. 2002. Spatial Databases - with application to GIS. Morgan Kaufmann, San Francisco, 410pp.

The grid file is a spatial index. A typical example is to create a grid file over 2D spatial data points. In particular, as shown in the left figure, we divide the 2D data space into multiple cells of equal size. If a spatial data point falls into a cell, then we say that the cell contains this data point. As shown in the right figure, each cell has a pointer pointing to a list of data points falling into this cell such as [a, b, c, d]. When the list of data points exceeds the page size, we maintain a list of overflow pages.

Grid File (cont'd)

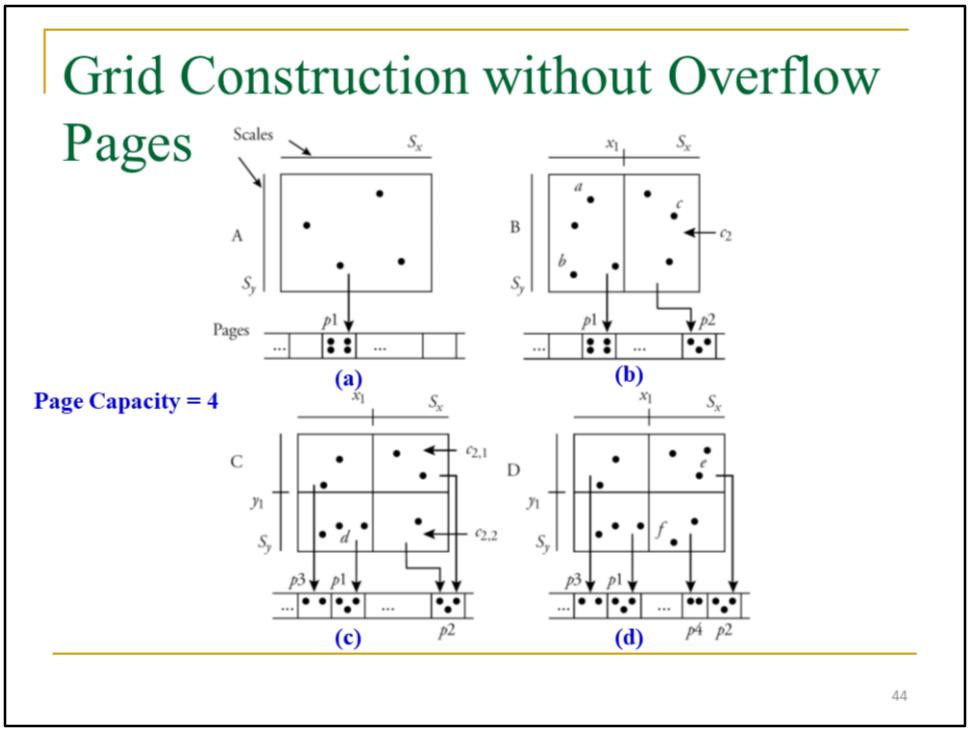
- Fixed grid
  - Partition a *d*-dimensional data space into cells of equal size
  - Operations
    - Insertion
    - Point query
    - Window query



43

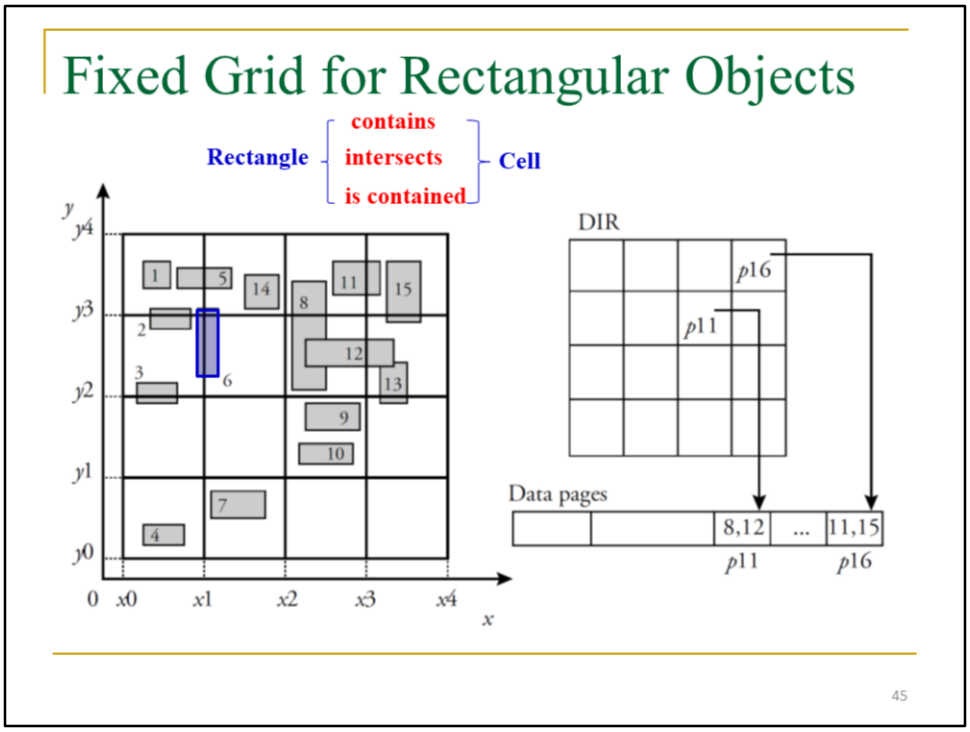
P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases - with application to GIS. *Morgan Kaufmann*, San Francisco, 2002.

Typical operations over the grid file include insertion, point query, and window query. The insertion is to insert a point into a cell of the grid file. The point query is to check the existence or non-existence of a point in the database. The window query is to retrieve all data points within a given query range.



P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases - with application to GIS. *Morgan Kaufmann*, San Francisco, 2002.

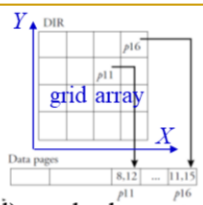
Here is an example of dynamically constructing a grid file. Initially, as shown in Figure (a), the entire data space is just one cell. When the cell contains data points more than the page capacity (i.e., 4 in this example), we will split the data space into 2 cells of equal size, and, as shown in Figure (b), allocate two lists of points pointed by these 2 cells, respectively. Similarly, in Figure (c), when one of the 2 lists exceeds the page size, we will further split the space into 4 cells, pointing to 4 disk pages (or 4 lists), respectively. Note that, here cells c_{2, 1} and c_{2, 2} are pointing to the same disk page p_2, since there is no overflow. In the case of overflow, as shown in Figure (d), if we add one more point to cell c_{2, 2}, we will allocate one new disk page p_4, and redistribute points in cells c_{2, 1} and c_{2, 2} to pages p_2 and p_4, respectively.



P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases - with application to GIS. *Morgan Kaufmann*, San Francisco, 2002.

The grid file can not only support data points, but also those objects with non-zero areas (e.g., rectangles). In this case, each object may span across multiple cells. One possible remedy to accommodate such objects is to let multiple cells record the same object for multiple times. As shown in the figure, the blue rectangular object intersects with 4 cells. Therefore, all these 4 cells will record this object for 4 times.

Grid File Implementation



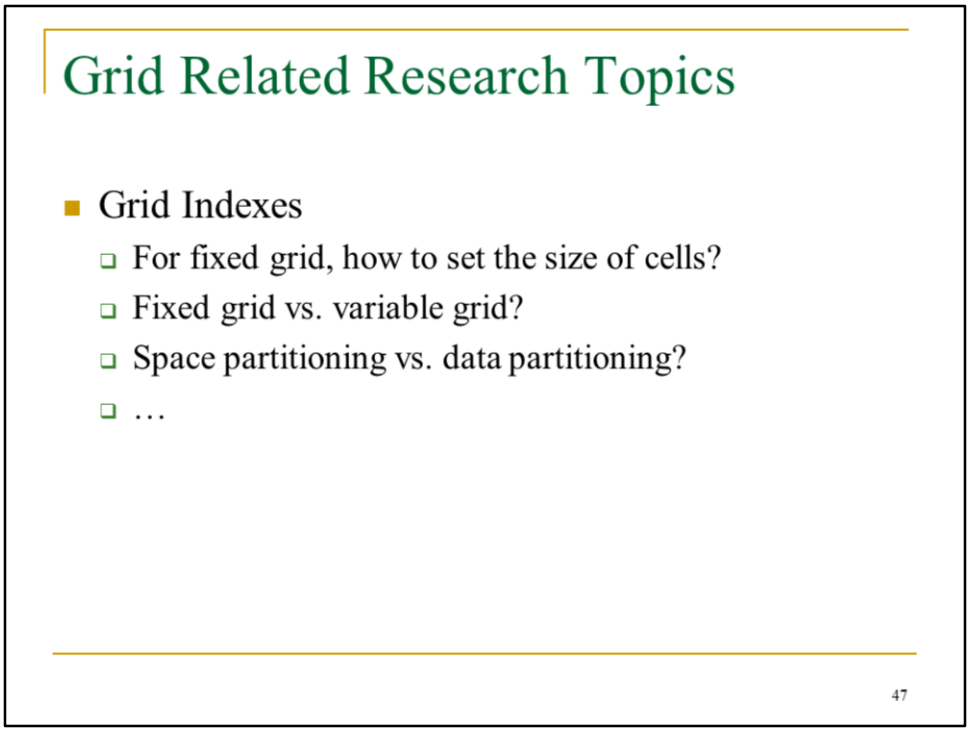
- 2-dimensional grid directory
  - Grid array: a 2-dimensional array (disk-based), each element with a pointer, p_i, pointing to a bucket
  - Linear scales: two vectors (memory-based 1-dimensional array), X and Y, indicating the range of cells on the two dimensions
- Example:
  - Age vector X = [0, 20, 40, 60, 80] and last name vector Y = [A, H, O, U, Z]
  - Search over grid with query (35, **L**ian)
  - Exact match: at most 2 I/Os
  - Range search: (1) use linear scales to obtain relevant cells; (2) access the grid array to obtain bucket addresses; and (3) retrieve data from buckets

46

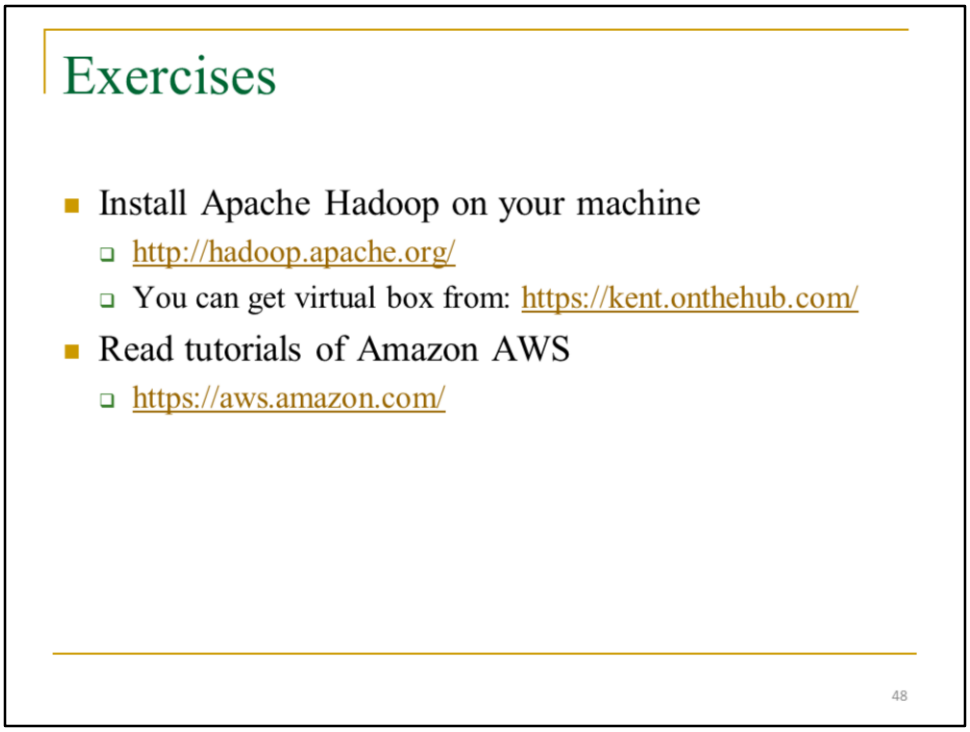
http://infolab.usc.edu/csci587/Fall2016/slides/session2-spatial-indexes-no-rtree.pdf

For the detailed implementation of the grid file, we can maintain a grid array which stores the content of each cell and linear scales, which record the boundaries of cells on each dimension. This implementation can also support grid files with cells of different sizes.

For example, we consider a two-dimensional grid file over objects with two attributes, Age and Last Name. The two linear scales are represented by vectors X and Y. That is, we split the age into age groups [0, 20), [20, 40), [40, 60), and [60, 80], and last name to name groups starting from A to H (exclusive), from H to O (exclusive), and so on. Then, if we want to search for a person with age 35, and last name “Lian”, according to linear scales, we can directly visit a cell with age range [20, 40) and letter range of the last name [H, O). Usually, it takes 2 I/Os, that is, 1 page access to the grid array to obtain the pointer in the cell and 1 page access to the actual list/bucket containing the record (35, Lian).



There are many other important and interesting issues with the grid file or grid index. For example, it is not trivial how to set an optimal size for cells that can achieve low query processing cost. It is also interesting to study variable grids with cells of different sizes. When constructing the grid file, it is not clear if we should directly partition the data space or partition the space according to the actual distribution of data points.



After the class, you can prepare the software for understanding/implementing the distributed computing, via MapReduce, discussed in the next module. You can install Apache Hadoop on your PC and read tutorials of Amazon Web Service (AWS). If you are using MacOS, you can install virtual box and operating systems such as Linux and Ubuntu.