CPS352 Lecture - Indexing

Last revised 3/18/2021

Objectives:

1. To explain motivations and conflicting goals for indexing
2. To explain different types of indexes (ordered versus hashed; clustering versus non-clustering; dense versus sparse; multilevel indexes)
3. To show how a B-Tree can be used for an index or whole file
4. To show how hashing can be used for an index or whole file
5. To introduce SQL `create index`

Materials:

1. Projectables of example of sparse index.
2. Projectable of summary of index types
3. Projectables of example B+ Tree; search example; insertion example that splits a leaf (2); B+ Tree with links to support sequential traversal added;
4. Projectable of Figure 14.9 from book
5. Projectable of calculation of minimum nodes in a B+ Tree with m = 200 and room for 50 rows in leaf nodes and 3 levels of index
6. Projectable of example of a Hashtable
7. Projectable of syntax for `create index` in db2, mysql

I. **Introduction**

A. Database systems are often used to manage very large databases, with a high volume of transactions. In such cases, system performance (as measured by the average time it takes to complete a transaction) can be crucial not only to the success of the system, but also to the success of the organization using it.

B. One of the most common operations a DBMS has to perform is searching for the row(s) of some table containing some specified value(s) in some column(s).

1. The select operation in relational algebra entails just such a search.

   Example: suppose the IRS had a taxpayer table with the SSN as the primary key, and also holding the taxpayer's name, and suppose it contained information on 100 million tax returns, kept in order of primary key. To find the row corresponding to a given name might entail scanning all 100 million records. If the 100 records fit in a disk block, that could mean as as 1 million disk accesses, which could take as much as 10,000 seconds or over 2 hours!

2. In relational systems, the natural join operation is very common. This may entail scanning through one of the two relations row by row, searching the second relation for a match on the join field(s) for the current row in the first relation.

3. Because searches are so common, major improvements in system performance can be realized by storing special index structures with the database to facilitate searching. The purpose of an index is to enable us to find the row(s) we are interested in without having to look through the entire table - c.f. the relative ease of finding a topic in a book by using the index, rather than by reading through the book from start to finish!

4. Two important considerations in the design of indexing systems are:

   a) The tradeoff between the computational gain from using the index and the computational cost of maintaining it when the table is altered by an operation such as insert, update, or delete.

   b) The extra disk space that the index structure requires, together with extra time needed to maintain the index when modifying the data.

C. A key term used in discussing indexes is the term <u>search key</u>.

   1. This is a separate and distinct concept from the concept of superkey, candidate key, primary key, etc. Search keys can by superkeys, but need not be. [One of those unfortunate cases where the same word is used in two ways.].

2. A search key is the attribute (or sometimes set of attributes) on which an index is based. For example, if a book contains an index by topic, then topic is the search key. If it contains an index by author cited, then author is the search key, etc.

3. It is possible, of course, for one table to have multiple indexes - each based on a different search key.

   Example: a student registration system will have a Student table, for which it is likely that there will be at least two indexes: one based on the student-id, and another based on the last-name.

   Because each index needs to be maintained when the table is modified by an insert, delete, or update - and because index structures consume additional disk space - an index is created only when there is reason to believe that the performance gain from having it outweighs the cost of storing/maintaining it.

4. In general, indexes are much smaller than the table they index - so where possible they can be stored in a faster kind of storage to expedite processing of queries.

   a. If the system has both SSD and disk secondary storage, the indexes for a database may be kept in SSD storage while the data resides on disk.

   b. It may be expedient for the DBMS to copy all or the top level of an index into primary storage when a connection is established to a database - to be discarded when the connection is terminated.

D. In considering index structures, one has to consider issues like:

1. The purpose for which the index will be used.

a) All index structures support finding the row(s) in which some specific value of the search key occurs (e.g. SQL `where searchkey = somevalue).`

b) Some index structures also facilitate range queries (e.g. SQL `where searchkey between somevalue and someothervalue`).

c) Some index structures also facilitate accessing all the rows in the table in order of the search key (e.g SQL `order by searchkey`)

2. How frequently the data in the table in question is updated. If the table is modified frequently, the index structure needs to be one that facilitates fast modification. If the table is read-mostly (or read-only), the index structure might be one that makes modification more expensive but provides easier access to data.

3. Whether the search key is, in fact, a superkey. In this case, an exact value search will find at most one row of the table. If it is not a superkey, an exact value search might find multiple rows. (Contrast searching a student database using student-id as a search key versus using last-name)

   a) If the search key is a superkey, then each entry in the index will refert to exactly one row in the table.

   b) But if the search key is not a superkey, the index structure must allow each entry to refer to multiple rows - calling for a more complex structure.

E. Recall from our discussion of physical file systems that the time needed to access information on disk is orders of magnitude greater than the time needed to access information in main memory. Hence, the goal of an index structure is to facilitate finding the disk block containing the desired table row with minimal disk accesses. We are willing to do a fair amount of searching (even a sequential search) in the block once it has been read from disk to actually locate the desired data, as long as we can find the block quickly.

F. Index structures are part of the physical level of database design, and so are usually created and maintained by the DBA.

1. One of the benefits of physical data independence as an attribute of a DBMS is that it allows the DBA to add or alter index structures at any time without affecting user queries (except, hopefully, for improving their speed.)

2. Setting up such structures, along with the file design issues we discussed in the previous lecture, constitutes the task of TUNING the database for optimal performance.

G. The DBMS itself may create indexes for certain purposes.

1. If an attribute of a table is declared to be the primary key or unique, the DBMS may create an index on that key which is used to enforce the constraint. Any time a row is being inserted into the table, the DBMS can first check the index to see if the value of a primary key or unique attribute occurs there - if so, the new insertion must be treated as a constraint violation.

2. Sometimes the DBMS will create a temporary index just for the processing of a single query. We will see examples of this when we look at query processing strategies.

   If it's small enough, this temporary index may be kept in primary storage rather than being kept on disk. In any case, it is discarded when processing the query is done.

II. **Types of Indexes**

A. Because the requirements for indexes vary, there are a number of different types of index structures. These can be considered in terms of various alternatives.

B. In an <u>ordered</u> index, the index entries are kept in order of the search key. In a <u>hashed</u> index, they are not - a hashing function (as discussed in CPS222) controls the order of the entries.

Example: Indexes in books are always ordered indexes - a hashed index would be close to useless to people.

1. An ordered index facilitates range and prefix queries and accessing all the rows in search key order. A hashed index does not.

2. But a hashed index often leads to more efficient access and maintenance, for reasons we will see.

C. In a <u>clustering</u> index, the actual data is stored in the order dictated by the index. In a <u>non-clustering</u> index, it is not.

1. Frequently, the term "clustering" is only used for ordered indexes, but technically a hashed index could be used to determined the placement of the rows, in which case - of course - they would not be kept in any sorted order.

2. Obviously, a given file can have at most one clustering index

3. A clustering index has both advantages and disadvantages:

   a) If the index is ordered, clustering makes range queries much easier, because the index need only be used to locate the first row in the range, and then subsequent rows are found by following the physical order of the data.

   Example (a bit contrived). If you wanted to read the descriptions of all the CS courses offered at Gordon, you might use an index to locate the first page of the catalog containing course descriptions for CS, and then continue from there to read all the descriptions without needing to refer to the index again.

b) If the search key is not a superkey, It also makes accessing multiple rows having the same search key value easier, because the index need only be used to locate the first row having the desired value, and subsequent rows are sequentially adjacent.

c) However, a clustering index complicates maintenance, because when a row is inserted, it must be inserted at the right physical space; when a row is deleted, the vacated space must be somehow“closed up”, and if a row is updated and the search key value changes, then it must be moved. (We will look at a structure - the B+ tree - that makes this relatively easy, though.)

4. Sometimes a clustering index, if one is present, is called a primary index. Non-clustering indexes are sometimes called secondary indexes. Note that, while a primary index might be based on the primary key of the file, it need not be. (Another place where the same word can have two different meanings!)

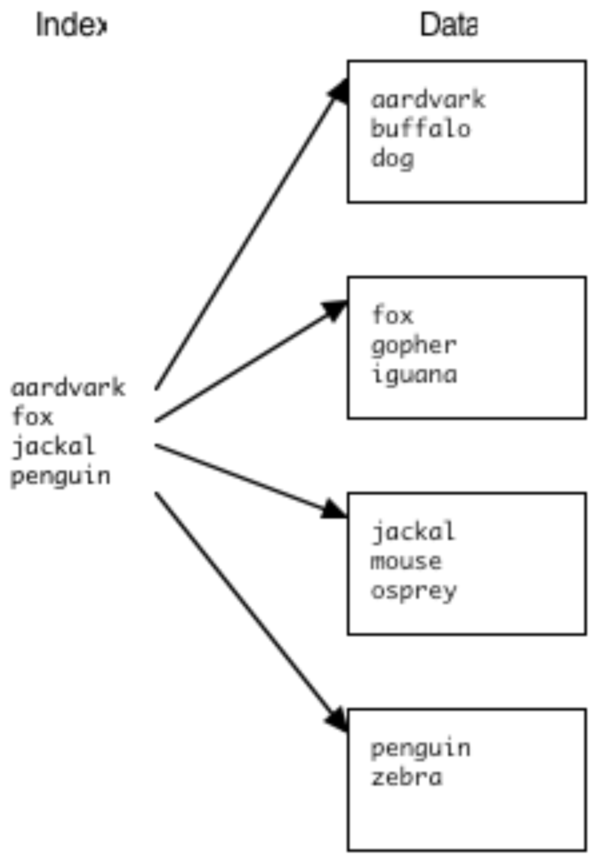
   Example: in a registration system, a Student table (with primary key student_id) might be set up with a clustering index based on name (rather than student_id) because often things like enrollment lists are sorted by name. In this case, the primary index is not based on the primary key.

D. In a dense index, there is one entry in the index for each distinct value of the search key; in a sparse index, only some values occur.

1. Only an ordered, clustering index can be sparse. The idea is that the index is used to locate a starting point for a search of the physical data (using the largest index entry that is <= the desired value). Then rows are searched in order until the right row is found.

2. Recall that we said earlier that data is physically stored on disk in blocks that typically contain several rows of a table. If there is a sparse index for this data, the index typically contains one entry per block - the smallest search key value occurring in that block.

Example - assume we have a library borrowers table in which 3 rows are stored per block (unrealistically low to keep example simple). Then a table of 11 borrowers with a clustering index based on last name might look like this:



PROJECT

E. We've covered a lot of terminology! To summarize:

ORDERED INDEX
- Facilitates exact match search
- Facilitates range or prefix query
- Facilitates order by chosen search key

HASHED INDEX
- Only facilitates exact match search
- Search and maintenance generally faster

DENSE INDEX

Has one entry for each value of the search key

SPARSE INDEX

Typically has only one entry for each block of data. (Therefore only possible if the index is ordered and clustering)

PRIMARY INDEX

at most one per table

clustering

usually ordered - technically can be a hash index, but
    term primary usually taken as implying ordered)

search key based on may or may not be primary key -
    but often primary key is the one used.
    - if search key is a super key, then any entry leads to only one row
    - if search key is not a super key, then any entry may lead to multiple rows. (In this case, must be ordered)

can be dense or sparse

SECONDARY INDEX

any number per table
    (but must consider tradeoff between benefits and space/maintenance cost when deciding whether/how many to have)

never clustering

can be either ordered or hashed

can be based on any search key

must be dense

PROJECT

III. **B-Tree Indexes**

A. The most common data structure used for ordered indexes is the B-Tree. Such a structure can be used for both clustering and non-clustering indexes. In the clustering case, the whole table may be stored using a variant of the basic B-Tree called the B+ Tree.

B. BTrees are discussed in depth in CPS222. We will just discuss them briefly here. We will develop the specific application to a clustering index using the B+ Tree variant to store the whole table here. **Note that this is not the same as the example in the book.** In this example, the entire table is stored in the tree. (This is appropriate for a primary index, but not for a secondary index.) [The book discusses this option later when it talks about B+ tree file organization]

1. A B+ Tree is a tree composed of two types of nodes:

    a) Leaf nodes contain the actual rows of a table

    b) Internal nodes contain index information - pairs consisting of a search key value and a pointer to a node where that value occurs.

2. B+ Tree indexes are typically multilevel, with all the leaf nodes on the same level.

3. The structure is controlled by a numeric value - called the <u>order</u> of the tree (n).

C. This value is determined by the size of a node and the size of a key-value pair, such that an internal node can hold at <u>most</u> n-1 keys plus n pointers to other nodes.

    a) The algorithm will guarantee that each internal node - except the root - will have at <u>least</u> ceiling(n/2) children. The root will have at <u>least</u> two children.

b) An internal node which contains k keys will have k + 1 children. The keys serve to separate the children in multiway search tree fashion.

c) Of course, leaf nodes will typically hold fewer entries (as determined by the size of the node and the size of a table row). The algorithm will ensure, however, that each leaf node holds no less than ceiling(maximum possible number of values / 2).

2. We will develop this example for a B+-Tree of order 7. (Much higher orders - often in the 100's - are typical - but we use a small value to keep the example simple). We will assume that each leaf node can hold a maximum of 3 table rows.

   PROJECT

   a) Note that this is a sparse index - possible because it is a clustering one.

   b) To search for a key:

```
start at the root.
while at an internal node:

If the value being sought is less than the smallest key stored in the node,
 go to the leftmost child
else
 go to the child corresponding to the largest stored key
 that is <= the desired value (where the second child
 corresponds to the first key ...)

when we get to a leaf node, the desired table row will be in it if it occurs
at all [ so a simple sequential search of this node can be used to find it ]
```

   (1)Examples:

   Search for horse: < penguin, so go left; > gopher but < jackal, so go to third leaf; leaf and found by searching leaf.

   Search for hippo: follow same path, but at leaf - not there so not in table

c) To add a new row:

Use search procedure to find node where it would be if it were present.

```
if there is room,
 put it there.
else
 divide the keys in two
 create a new right block to contain half the keys
 "promote" the first key in the right block.  Insert this key, plus a
         pointer to the new right block, in the parent
```

i. This may actually cause the parent to split as well, in which case we create a new internal node and promote the "split key" to the parent.

   This process could ultimately lead to the root having to split in two, in which case we create a new root that has the two halves of the original root as its children

ii. Examples:

   Insert terrance tortoise. Procedure finds node currently containing semantha snake and tommy turtle as the one where it belongs, and there is room - so put it there

   Insert donald duck.

   PROJECT steps

iii. This procedure is guaranteed to preserve the B+ Tree properties

   (a) If a leaf node is split, it must have been because the row to be added would have resulted in one more than the maximum number of rows needing to be stored - in which case each half is guaranteed to have at least ceiling(maximum number/2).

(b) If an internal node is split, there are m+1 children in all involved - in which case each half has at least ceiling(m/2) of them.

(c) If the root is split, the new root has at least two children.

(d) All leaves are always at the same level. The only way the height of the tree grows is by splitting the root - in which case the level of all existing nodes increases by 1

3. A refinement is to add links from each leaf node to the next, to facilitate range queries that span multiple leaves

   PROJECT

D. The examples in the book show how B+ Trees can be used for non-clustering indexes. Basically, in this case, the leaf nodes hold pointers to the disk blocks that contain the actual data.

   PROJECT Figure 14.9 in book

   Note: pointers can be block numbers, with a search of the block used to locate the actual record

E. For large tables, it will be necessary to use multiple levels in a B-Tree index. The example we just did actually entails a root block leading to one of several blocks at the next level which in turn lead to the blocks containing the data. Thus it might seem that three disk accesses would be required for each lookup - one to the root block, one to the block at the next level, and one to the block containing the data.

1. A simple - and very low storage cost - way to reduce the cost from 3 accesses to 2 is to keep the root block in primary memory at all times that we are connected to the database.

2. It may even be possible to keep the blocks at the next level down in primary memory as well. This would reduce the cost for a lookup to the bare minimum possible - 1 disk access.

3. For very large tables, it may be necessary to add a third level of index blocks, which may or may not be possible to store in primary memory.

   With a reasonable branching factor, it would take a large table indeed to require this many levels of index, though. Consider the following calculation using the assumptions shown:

Assume:	index block can have maximum of 200 children so minimum is 2 at root; 100 in rest
	leaf block can hold maximum of 50 rows so minimum is 25Level 0:

   Number of blocks at root: 1
   Level 1: Minimum number of blocks: 2 - each having 100 at least children
   Level 2: Minimum number of blocks: 200 - each having at least 100 children
   Level 3: Minimum number of blocks: 20,000 - each at least holding 25 rows
   So minimum number of rows is 500,000

   So using these parameters, any table with less than 500,000 rows would only need 2 levels of index!

   PROJECT

IV. **Hashed Indexes**

A. An alternate approach to providing fast access to the record(s) containing a given key value is HASHING.

1. The index can be a clustering one. In this case, basic idea is this:

   a. The file is organized into m buckets, numbered 1..m.

   b. A function is devised that maps a value of the search key into an integer in the range 1..m. This function is called the HASH

FUNCTION. It should have the property that the probability of a key mapping to a given value is about 1/m - i.e. the distribution of values of the function should be uniform.

Example: A function that generally has this property is (key mod m) + 1 - where the key is first converted to an integer if necessary. This simple function actually works fairly well if m is a prime number, or at least has no small prime factors.

c. When a record is to be stored, the hash function of its key is used to decide which bucket it is to be stored in.

d. To make this work, the file must be designed so that the buckets never become completely full. (About 80% loading is a good rule of thumb.)

Nonetheless, there will always be cases where some one bucket runs out of room - in which case some sort of strategy is used to assign the new record to another bucket. (Overflow buckets; rehashing using a new hash function)

PROJECT Example - hash function used is (sum of ASCII codes of letters in first name) mod 11 + 1 - which is not terribly good.

Example: lookup tommy - calculate hash function as

116 + 111 + 109 + 109 + 121 = 566 mod 11 = 5 + 1 = 6
 - found in bucket 6

Example: lookup terrance - hash function also calculates as 6 - but not there, so not in table

2. Hashtables can also be used for non-clustering indexes. In this case, the bucket located by hashing holds key-value pairs, with the value being the number(s) of the bucket(s) where the desired row occurs.

B. Hashing compared with ordered indexes

1. Hashing can be much faster than use of an ordered index. A well-designed hash structure usually allows the desired record to be found using a single disk access (for a clustering index kept in memory), or two disk accesses (index block plus data in a non-clustering index) whereas an ordered index may require more accesses to work down through the levels of a tree structure. Also, there is less overhead involved in maintaining a hash structure, so insertions and deletions are less costly this way.

2. However, hashing does not support range queries or prefix match queries, only exact match queries. Neither does it support accessing the table in the order of its search key.

3. Finally, an ordered index can guarantee the number of disk accesses needed to locate a given item. With a hashtable, the worst case performance may require accessing all the records in the file - though this is very unlikely in practice!

C. Hashing is discussed more extensively in CPS222

V. **Specifying Index Structures in SQL**

A. The DBMS will sometimes automatically create indexes for a table - either permanent indexes or temporary ones used just during the processing of a single query. .

1. As an example, db2 automatically creates an index for any column (or set of columns that are declared to be the primary key of a table or are declared `unique` (hence a candidate key). Why do you think it does this?

   ASK

   If a column is declared to be a primary key or unique, the DBMS must enforce the rule that no two rows can have the same value for this column. This means that, when a row is inserted or updated, the DBMS must check to be sure the value does not already occur in the table. Absent an index, this would require scanning through the entire table - a very cumbersome process with a large table. With an index, the problem reduces to checking the index to see if an entry occurs. (Contrast checking some book to see if it discusses a certain topic by using the index, versus reading through the entire book looking for a certain word!)

2. We will talk about temporary indexes later, when we talk about query processing "optimization".

B. SQL includes a `create index` statement that can be used to manually create an index. The DBA can do this if it is known that a particular column will commonly be used as a search key for queries or as the join column in natural joins.

1. Though the basic syntax is the same, details of the options provided by this statement vary quite widely from system to system.

   Example: PROJECT `create index` syntax for db2, mysql.

2. For our library database, what indexes might be appropriate?

   Decomposed scheme:

   Borrower(<u>borrower_id</u>, last_name, first_name),
   Book_title(<u>call_number</u>, title),
   Book_author(call_number, author)
   Book(<u>call_number, copy_number</u>, accession_number,
   borrower_id, date_due)

   ASK

   Borrower: borrower_id for sure (natural joins); maybe last_name
   Book_title: call_number (natural joins)
   Book_author: call_number (natural joins)
   Book: call_number + copy_number (to facilitate checkout/return)

db2 would create all but one of these as a result of primary key constraints. Which would need to be created manually?

ASK

call_number index on Book_author

3. Demo: create Book_author table as above, then index.

```
create table book_author
   (call_number char(20), author char(20));
create index ba_call on book_author(call_number);
```