Databases	Lecture 6
ENGR 3599	Feb 28, 2019

Storage and Indexing

Up until now, we've focused on *logical modeling* — The Entity-Relationship model to model data, and the Relational model to model entities and relationships using relations — and the kind of querying that these models support.

Today, we worry about implementation aspects of the Relational model, sometimes called the physical modeling. We're going to first look at how we can model data in memory, and then move to disk-based storage.

We're working in Python, which is not a great language to do this kind of reasoning: it hides too much from you, and does a lot of work behind the scenes. A language like C or C++ would be more appropriate.

Basic implementations:

1. Unordered array of tuples
2. Ordered array of tuples
3. Hash table

An unordered array of tuples is the most straightforward representation: inserts are easy (just append at the end of the array) but lookups are expensive – in the worst case, you have to scan the whole array. (For instance, if the attribute you're searching for is not a primary key, then you have to scan the whole array because even finding the entry you're looking for does not guarantee that there isn't another.)

Ordered arrays and hash tables require you to make a decision: deciding which attribute (or attributes) to order/hash the entries by. We call the ordering attributes the *search key*. Queries that select on the search key will be faster.

An ordered array of tuples (ordered by the value of the search key) allow for a pretty fast lookup when searching for a given value of the search key by using binary search: start in the middle of the array, if the search key value of the tuple you find there is greater than what you're looking for, recursively binary search in the first half of the array, otherwise, recursively binary search in the second half of the array. Obviously, you stop when you find what you're looking for (or when you find nothing).

The downside of ordered arrays of tuples is that insertions are costly: you have to find where to insert the tuple (you can use a binary search to do that) but then you have to shift every tuple after that position one position to the right to make room for the tuple you're inserting. That's expensive.

Hash tables are arrays in which the position of a tuple in the array can be computed directly from the tuple via a *hash function*. A hash function is a function that takes a value in some space (here, the domain of the search key) and produces an index into the array. Insertion of a tuple in a hash table is easy: apply the hash function to the value of the search key for that tuple to get an index, and put the tuple at that index in the hash table. To lookup a tuple with a given search key value, apply the hash function to the value, and get the tuple at that index.

The difficulty with hash tables is collision: two tuples that hash to the same index. The two approaches in the literature to handle collision are open addressing and closed addressing.

- Open addressing means that at every cell of the array, there is a link to an ordered array that hold all the the tuples that hash at that particular index. Insertion means first using the hash function to get the index into the hash table, then inserting into the ordered array that lives at that index. Lookup is similar: using the has function to get the index into the hash table, then looking up the tuple in the array there using binary search.
- Closed addressing means that everything in contained in the hash table, without relying on additional structures like arrays linked to each cell of the hash table. To insert by closed addressing, you use the hash table to find the index where you should put the tuple. If there is already a tuple there, you look for another index from that index, for example, the next cell. If that is taken too, the next cell, and so on until you find a free cell. To look up, do the same: compute the hash to get the index, and if you get the tuple you were searching for, great, otherwise, you look in the next cell that insertion would have put it in, until you get to a tuple with a different hash value. Different ways to find the next index exist: the above is called linear addressing, where you go to an index that is a fixed number of cells before or after the one you're at. Other schemes exist with better "spread" properties.

In practice, ordered arrays and hash tables are commonly used. The advantage of ordered arrays is when you need to do range queries over the search key, queries that essentially say: look at all tuples where the search key is between A and B. Then you can do a binary search on A and a binary search on B, and you know that all the tuples you want are between the index you found for A and the index you found for B. Things are not so easy for hash tables, which would require you to do a lookup for every value in the range, which is not always easy (think when the search key is a timestamp).

What if you want to support efficient queries for more than just the search key? Then you can create what's called a *secondary index* on another attribute A. A secondary index is basically an ordered array (or a hashtable) with A as a search key, but instead of holding tuples, it's holding indices into the structure containing the tuples. So if you're doing queries against that attribute A, you can use the secondary index to help find the tuples faster.

You can have secondary indexes for as many attributes as you want. Keep in mind that each secondary index requires additional space (of the order the number of tuples in the relation) which can be costly in terms of space, and you're going to have to maintain those indexes, meaning changing them every time you do an insert in the case of ordered arrays — since inserting into an ordered array changes the position of tuples in the array (and therefore, the indices need to track those changes).

Data is usually stored and accessed on disk. Why?

- Data too large for memory
- Want data to be persistent
- Want more than 1 process to access the data

Data is stored on disk in files – for the sake of discussion, one relation per file, where each file is split into blocks. A block can contain a fixed number of tuples (how many depends on the size of the blocks and the size of the tuples — we assume all tuples in a relation have the same size). A block in a file can be accessed by position: disks support random-access files. To read from a file, you first pull a block into memory, and then can read data from that block. To write to a file, you first pull the block in memory, make the change, then push the block to disk. Pulling and pushing blocks is comparatively expensive. Algorithms for handling data on disk will try to minimize the number of blocks pulled and pushed.

All the concepts we saw for in-memory management carry over when talking about disk-based data.

When tuples are unordered, the file is called a heap file (confusing, I know). Insert are easy: pull the last block of the file, and add the tuple at the end of that block before pushing it. If the block is full, allocate a new block, write the tuple to it, and push that new block at the end of the file. Lookups are expensive: since the tuples are not ordered, the best you can do is pull the first block, see if the tuple you're looking for is there, if not pull the second block, and so on, sequentially, until you've found the tuple you're looking for. If you're looking up something based on an attribute that is not the primary key, then there may be multiple tuples of interest, and you need to keep on looking until the last block.

When tuples in the file are ordered, the file is called a sequential file. Again, we need to define a search key by which to order tuples. Lookups are reasonably fast when using binary

seach: pull the middle block, and determine if the block contains the tuple you're looking for (by looking at the smallest and largest tuple in the block). If the tuple should be there, do a binary search to find it. If the tuple should not be there, then recursively pull the middle block in the left half of the file, or the middle block in the right half of the file, depending on where the tuple should live.

Inserts are a lot more expensive in a sequential file, since a priori they require shifting every tuple to the right, which requires pulling all the blocks after the block where the insertion happens. A common approach is to simply accumulate all the inserts in an overflow heap file (which needs to be searched when doing a lookup, obviously) and to reconstruct the sequential file at regular intervals in a batch fashion, perhaps whenever the overflow file reaches a certain size.

Binary search on a sequential file is good, but we can do better – binary search still requires a fair number of block pulls. Better is to build an index, to obtain an indexed sequential file. An index is a block on the file that holds an entry per block of the file, giving the value of the search key of the first tuple in each block. The index is kept ordered by value of the search key. Then to find the block where a tuple should live when doing a lookup, you can simply do a binary search in the index on the value of the search key, and find the index of the corresponding block. When the index can all fit within one block, this is really fast, and requires two block pulls: one for the index, one for the block to retrieve the tuple. Inserts in an indexed sequential file are handled like inserts in a sequential file – placed in an overflow heap file, and the indexed sequential file is rebuild at regular intervals.

(When the search key is not a private key, the index needs to be a bit more complicated: a clustered index has an entry for each value of the search key present in the data, where for each value the index points to the first block containing tuples with that value for the search key. I will let you figure out how to work with that.)

The above works well when the index can fit in a single block, or perhaps a small number of blocks. When the index spreads over too many blocks, it is more efficient to build a multi-level index. Build the index as before, over multiple blocks, then build an level 2 index for the index, where each entry indicates which block contains the index entries for that value. If the level 2 index itself gets too large, you can create a level 3 index for the level 2 index, and so on.

Tree structures called B trees (and a variant called B+ trees) have been developped to manage multi-level indexes, basically to try to keep them balanced to minime the number of block pulls necessary to lookup values through the index. The textbooks contain good descriptions of such tree structures.

We can also add a secondary index for some attribute A to indexed sequential files. A secondary index for A is an index that for each value of A points to blocks holding tuples with that value for their A attribute. Since tuples with a given value of the A attribute can

be spread across arbitrary blocks (the sequential file is ordered by the search key which is not A), the index for a value of A should hold an array of block numbers.