Database Management Systems
Homework #4

File Storage and Organization

Sequential Access vs Random Access
These access patterns refer to the order which data can be accessed off of a disk. When data is sequentially accessed, the data is stored on the disk in the same order as its being accessed. This has desirable effects on access time because all the data is stored within a small proximity. This causes seek time and rotational latency time to drastically decrease after locating the first piece of data. Random access implies that there is no orderly constraint governing how the files are stored. This leads to a high average seek time because the arm may move from disk to disk, reading just a bit from each.

Variable Length Record Blocks
Free lists are only useful when every record is stored using the same number of bytes. In such a case, the empty record locations could be represented as a 1D list because the number of bytes per record is constant; multiplying the index by the constant always results in the number of bytes between the header and the record. This strategy doesn't work with variable list records because each record may be composed of a different number of bytes. As an alternative, the slotted page structure can be used. In this block model, all free space is in the middle of the block, between the header and the records. Instead of a free list, the header contains an array of the location and size of each record, a count of the records in the block, and the size of the free space available.

Multitable Clustering
Storing multiple relations in a single block is rarely a good idea. It increases the space between the records of any single relation (because other relations records are spliced between). This slows down the access time when selecting from just a single relation. In the case of a join though, clustering can produce significant performance gains. As opposed to having to visit different block for different relations, clustering puts them all in one spot.

Column Storage vs Row Storage
There are two obvious possible formats to store records. For one, each record gets its own row. This makes sense because deleting or inserting a record pertains to nothing but its respective row. Data analysis queries become a difficulty when it comes to this format. For example,

counting how many values are null in a given attribute in a relation requires a entry and an exit into each row for each record. Alternatively, columns can be used to represent all of the values per attribute in a relation. This makes the given example simpler. To count the null values just one column would have to accessed, as opposed to going in and out or rows.

Indexing

Sparse Indices

When an index structure is sparse, there is not an index for every record. This is done so to reduce how long it takes to iterate through the list. The only way to still be able to find every search-key would be to order the records. Accordingly, indices not stored in the index are stored near the indices that do exist. And since the ordering scheme is known, navigating to a search-key's nearest index then traversing in a directed manner is effective.

Walkthrough

(sorted on dept_name)

addr	emp_id	dept_name	sal
0	0	accounting	100000
1	2	actuarial	120000
2	3	engineering	125000
3	1	engineering	132000

Comparing the ways different indexes would build and operate is best done by creating models and analyzing them. Since optimizing access time is the goal, we could assess efficacy by considering the expected worst case navigation required for an index. Consider the examples below displaying the subtle difference between dense and sparse indexing on dept_name: (index → addr/s)

Dense-index on dept_name:	Sparse index on dept_name:
accounting → 0	accounting → 0
actuarial → 1	actuarial → 1
engineering → 2, 3	engineering → 2 (with a pointer to 3 at 2)

In this case, the expected access time for the dense index is:
(0,1 + 1,1 +2,1 + 3,2)/4 = 1.5, 1.2   where a,b = worst case index iterations, pointer jumps
For sparse index it is:
(0,1 + 1,1 + 2,1 + 2,1,1)/4 = 1.2, 1, .25   where _, _,c = jumps to next record

It is preferable to use a dense index when E[a,b] < E[a,b,c] where E is the expected worst case access time.

Hashing

<u>Static vs. Dynamic</u>
When values are hashed to a bucket, there is a possibility that the bucket is full. This can be dealt with two ways, dynamically an statically. In the static case, the initial structure of buckets remains. An overflow bucket is created and chained to the bucket the value was supposed to go to in linked list. Alternatively, the value can be added to an existing bucket dynamically found to have available space. This makes deletion tough though since the locations they are added to depend on the system's state. Deletion would have to follow the same dynamic path.

<u>Why Dynamic?</u>
Drawbacks from static hashing become clear when there is a lot of growth beyond the original file system size. Overflow buckets link together into long chains on some buckets, but not others. This detracts from the efficiency gains that come with random distribution. Dynamic hashing faces no such problems with growth.

<u>Extendible Hashing Depth</u>
The depth in extendible hashing signifies how many bits are being used. When exponentiated, it relates to how many buckets there can be. It's important to keep track of this quantity because it verifies whether or not a bucket overflow necessitates a bucket split.

<u>Linear Hashing Overflow</u>
In the case of overflow, new buckets are added to accommodate more space. This is done by splitting a bucket and rehashing its values into either the old or new bucket in the split. In linear hashing there is a pointer to which bucket is to be split, and it moves sequentially in linear order. Accordingly, the bucket that splits isn't necessarily the one that overflows. If there does happen to be an insertion into a bucket that is full, an overflow bucket is chained to the hash bucket it was supposed to join. This doesn't happen often though, because of the random distribution of elements-it is unlikely that values will get inserted into a bucket enough times to accumulate and necessitate an overflow bucket before being split.

InnoDB

<u>Choosing Between B-tree and Hash Indices</u>
Advantages that B-tree brings comes with the fact that it's ordered and balanced. Performance gains are realized when operating on these order bound indices. When commonly operating on ranges, such as with >,<,≥,≤,≠,=, where the query usually returns multiple records, B-trees should be used. Hash indices work well because there's a function that maps each value to a single bucket—there is no traversing of paths or multiple pointers (usually). Accordingly, associated performance gains are best realized for single value look-ups. This is because as the number of required look ups increase, the computation cost of hashing increases. MySQL also

utilizes full table scans. This is the most efficient access strategy when a very large percentage of rows in a table are being called.

Page Structure

In class we discussed a slotted block structure that had a directory followed by free space followed by the records. This differs from InnoDB's structure in both the order and variety of its components. There are seven components to an InnoDB block. In order they are: fil header, page header, maximum and minimum records (for ordered records), user records, free space, page directory, and fil trailer. InnoDB puts record before free space and the directory after the two of those. Additionally, it has components which were not discussed in class: the fil head and the fil trailer, which include data regarding the file system program; the page header, which stores page related data; and supremum and minimum records, which store the high and low ordered value contained in the block.

Clustered Indices Only

Relations with a unique attribute allow records to be ordered according to a scheme and efficiently accessed sequentially—this is a clustered index. InnoDB enforces a policy that all records must contain a clustered index, usually the primary key. This is done for the performance gains associated with sequential access. In the case that a user prefers to represent a relation using a secondary index, the primary key and specified attributes for each row is stored in the order associated with the secondary index. This information can then be used to look up the rows in the clustered index.

In-Memory Buffer Pool Strategy

In the LRU algorithm there is a list kept with the most recently accessed record in front. Since the list has a fixed size, records accessed from disk to be moved to memory evict a record from the back of the list. InnoDB implements a variation of this algorithm. The main difference is that in InnoDB's model, when a record is brought into memory, the record is added at the midpoint of the list. This is effectively a partition, creating new and old sublists. There is also an additional mechanism which brings records to the front at the head of the new sublist, which occurs when a record is accessed from memory. Such a design prevents data which has been recently accessed from memory (presumably in the new sublist) from getting evicted by large operations. E.g. consider a record which must be accessed every second. In the case of full table scan, it would get evicted according to the LRU algorithm (assuming the full table is larger than the memory size). According to InnoDB's algorithm the whole old list will get evicted, but none of the records in the new list. They aren't evicted until as many unique records are read from memory as their index in the new list.

PostgreSQL

Page Structure

Blocks are organized into 5 components—in order: PageHeaderData, ItemIdData, Free space, Items, special space. PageHeaderData includes information about the page, such as pointers delimiting the start and end of free space. ItemIdData acts as the directory, including the offset and length of each item in the block. The free space is next, placed between the directory and the items, which makes it easy to simultaneously allocate space for both the index in the directory and the item itself. Items stores the records. Special space is an area where data regarding special index instructions for special indexing methods can be stored.

Free Space Maps

It's important to be able to efficiently find where in a page there is enough free space for the addition of a new record. This is done by Postgres using a binary tree for each page which maps free space. Each node's value is the max available space of one of its children, and leaves are indices and the space available at them. Finding a compatible index is as simple as navigating down the tree where the nodes value is greater than the record size until a leaf. Instead of signifying the actual free space sizes, free spaces values are mapped onto a 0-255 scale so that the value can be stored as a byte. This creates a 1/256th of a page granularity at which free space can be assessed, since the exact value is not stored (to get the approximate free space, we can scale the 0-255 value by block_size). Pages' free space maps (FSM) are organized into another binary tree stored on specialized FSM pages. This tree operates similarly, except that leaf nodes now represent pages the respective max space available on them. When looking for free space, we start at the root on these specialized FSM pages and navigate similarly to the above mentioned fashion until a target page with enough space is found. There is a potential problem that arises when descending the higher level tree and that is concurrent edits. As the higher level FSM tree is being navigated, one of the values of the leaf nodes may change if it is edited concurrently. In the case this turns out to be true, the value at the leaf in the higher level tree is edited to reflect the updated amount of space. This change must also bubble up the tree so to correctly reflect the trees current state. The process then begins again from the beginning with hopes no concurrent edits on the target page again!

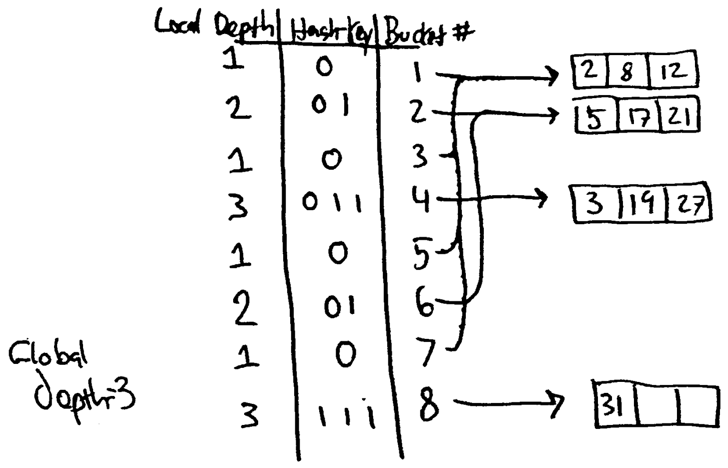
Shared Buffer

Postgres manages buffer organization using two values per page in the buffer: pins and content locks. Pins are used as a selection property, to reflect which page is being accessed in the buffer. Content locks are used to signify whether it's a read or write operation that being done with the block. This is important because in the read case, it's permissible for other backends to concurrently read the value, so shared locks are instantiated. In the case of a write operation, an exclusive lock is instantiated to mark that no other backends can instantiate a lock on it and even access it for that matter. When looking to evict from the buffer, the first page which is not pinned is evicted. The strategy differs in the case of a sequential table scan where a ring of buffers are reserved. They are used and reused to perform the scan, taking advantage of the fact that accessing a record during a scan does not necessarily make it a good candidate to be cached in memory.

Example Hash Tables

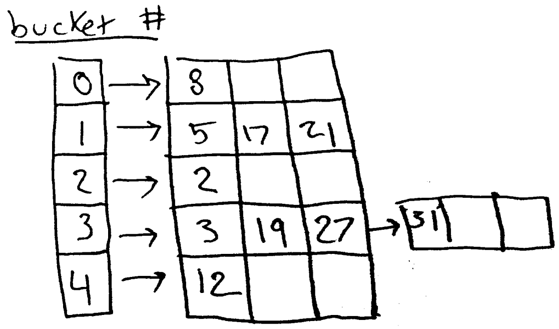
Extensible Hashing

X	h(x)=X%8	binary(h)
2	2	010
3	3	011
5	5	101
8	0	000
12	4	100
17	1	001
19	3	011
21	5	101
27	3	011
31	7	111



Linear Hashing

X	h(x)=X%8	$b_0(x)$ =h(x)%2	$b_1(x)$ =h(x)%4	$b_2(x)=\boxed{h}$ =h(x)%8 = $\boxed{h}$
2	2	0	2	
3	3	1	3	
5	5	1	1	
8	0	0	0	
12	4	0	0	
17	1	1	1	
19	3	1	3	
21	5	1	1	
27	3	1	3	
31	7	1	3	



Query Processing

Sort-based Division Algorithm
Sort the “numerator” by the denominators values. Instead of scanning the whole table, read values in the range of the denominators attributes. For each row, look at the non-denominator attributes, considered a candidate return value. I will use the term “on fire” and warmth to denote viable candidate return values (cold is bad and ready for eviction), and “section” to denote groups of rows which share the denominator attribute (the pizza post about this homework included relation o with sections 8 and 4). For initial section all rows get added to the hot list. At every following section, cool down the hot list initially and ignore rows not on the hot list. Heat up any candidate return value which is present in both the current section and the hot list. Repeat until arrive at irrelevant row in numerator. Return the hot values.

Hash-based Division Algorithm
Scan every row in the relation. Create a hash table using candidate return values as keys which stores the denominator attribute that cooccurs with each candidate return value. Iterate through the hash tables and return hash keys that point to tables in which all denominator attributes are present.

Block Nested-loop Join Cost Approximation
R1 has 800 blocks and R2 has 1500 blocks. With a memory size of M blocks, and saving one block for return values and one block for the inner relation, the estimated worst case cost is 800/(M-2)*1500+800 = 1,200,000/M - 599,200. This is because we have to iterate through all 1500 blocks in R2 for each block in R1 in addition to each block in the return set. It is divided by M-2 because multiple blocks can stored in memory and compared at a time so they can be accessed together.

Merge Join
Merge join requires that the relations are sorted so the cost to external merge join is the first and second cost, additionally there are the costs associated with going through all of the blocks the one time required.
Total cost = $2*1500(\log_{m-1}(1500/m)+1) + 2*800(\log_{m-1}(800/m)+1) + 1500 + 800$ transfers.

Hash Join
The hash join gets performance gains by using a hashing function to partition the operating relations. This allows for less comparisons since each tuple in a partition must only be compared to tuples in its respective partition. The estimated worst case cost is therefore:
$3(1500 + 1800) + 4*n_h$ where n_h is the number of partitions.

Binary Search Tree
There are only a few conditions: the value of the left child must be less than its parent node’s value, the value of the right child must be greater than its parent node’s value. Additionally, each child must be a binary search tree itself. Search time on a binary search tree is quite

optimal with worst case search time being n if its organized poorly and essentially a linked list. If the tree is balanced, the worst case performance becomes $\log_2 n$, where n is the number of nodes. This is because navigation to each child eliminates half the tree. There is a bottleneck on how efficient it is to use a binary search tree depending on the number of the files being structured and the size of the files being read. This is because iterating through a lengthy data structure could potentially be more costly than just reading the files in a scan, structure free. This bottleneck definitely occurs in index structures, where nothing but the index is stored. It is worth using in the FSM because each node's value represents the max space available on the page. Since the nodes point to the pages, as opposed to indices it's worth it.

Query Optimization

The two queries are the same if r2 lacks a C attribute. If func is min/max they would still be the same if the min/max is in r1.