CS150A Quiz 2 Solutions

Assume that each page in our system can hold 64 KB (1 KB = 1024 bytes), integers are 32-bits wide, and bytes are 8-bits wide.

Consider the following relation:

```
CREATE TABLE Submissions (
  record_id integer UNIQUE,
  assignment_id integer,
  student_id integer,
  time_submitted integer,
  grade_received byte,

  PRIMARY KEY(assignment_id, student_id)
);
```

Assume the column record_id corresponds to the row's actual record ID.

Q1: How large (in bytes) is a record?

We simply add up the sizes of each field in a record. We have 4 integer fields and 1 byte field, which is 4*4 = 17 bytes.

Q2: Suppose we begin each page with a 24-byte header plus a bitmap. At most, how many records can fit in an unpacked page?

Let’s convert everything into bits. First, a page holds 1,024 * 64 * 8 = 524,288 bits while a record holds 17 * 8 = 136 bits. Now, remember that in an unpacked page, each record needs an additional bit to represent whether or not it is valid (e.g. has been deleted). This means we need 1 more bit per record, so each record in fact requires 137 bits. Finally, we have an extra 24 bytes (24 * 8 = 192 bits) reserved for the page header. This gives us a maximum of (524,288 - 192) / 137 = 3,825 records.

We add two variable-length fields to our table schema. Now our table looks like this:

```
CREATE TABLE Submissions (
  record_id integer UNIQUE,
  assignment_id integer,
  student_id integer,
  time_submitted integer,
  grade_received byte,

  comment text,
  regrade_request text,

  PRIMARY KEY(assignment_id, student_id)
);
```

We decide to use slotted pages to store the variable length records. Each page begins with a 24-byte header plus a slot directory. (Assume this header contains information such as the number of valid records in the page.) Each pointer inside the slot directory consumes 20 bits/record, while the record header storing field offsets is 32 bits wide.

Q3: What is the maximum number of records that can fit in our slotted pages?

Again, we have 524,288 bits per page, the page header consumes 192 bits, and a record is 136 bits wide. Now instead of a bitmap, where each record takes 1 extra bits, we have a slot directory of pointers, so each record requires 20 extra bits. Thus each record costs us 136 + 20 = 156 bits. We need to store field offsets for the variable-length text fields, which is an additional 32 bits per record, bringing us to a total of 156 + 32 = 188 bits per record. Finally, note that we get the maximum possible number of records when all comment and regrade_request fields are NULL; i.e. they both take up 0 bytes. Then the smallest possible memory footprint per record is 188 bits/record, so we get (524,288 - 192) / 188 = 2,787 records.

Q4: We decide to squash the two text fields together into one field using a semicolon separator character (;), which allows us to shrink the record header from 32 bits to 16 bits at the cost of 8 bits (for the semicolon). For example, the columns ("Submitted late", "Dog ate my homework") get compressed into "Submitted late;Dog ate my homework". Which of the following are true with this new scheme?

Since we're using the semicolon as a field separator, we can't enter comments with semicolons. In our example, the comment "Fantastic work; good job!" would be truncated to "Fantastic work" and " good job!" would be misinterpreted as the regrade_request field. However, since our records become 8 bits shorter, we'll be able to fit more records per page. As a result, our table will become smaller, and table scans will speed up accordingly (depending on how many pages we save on storage).

Suppose we have an alternative 2 unclustered index on (assignment_id, student_id) with a depth of 3 (one must traverse 3 index pages to reach any leaf page). Here's the schema:

```
CREATE TABLE Submissions (
  record_id integer UNIQUE,
  assignment_id integer,
  student_id integer,
  time_submitted integer,
  grade_received byte,

  comment text,
  regrade_request text,

  PRIMARY KEY(assignment_id, student_id)
);

CREATE INDEX SubmissionLookupIndex
ON Submissions (assignment_id, student_id);
```

Assume the table takes up 12 MB on disk (1 MB = 1024 KB). (This includes extra space allocated for future insertions.)

Q5: We want to scan all the records in Submissions. How many I/Os will this operation take?

To do a full table scan, we read each page into memory once. There are 12 * 1024/64 = 192 pages in the table, so that's 192 I/Os (all page reads).

Q6: UPDATE Students SET grade_received=85 WHERE assignment_id=20 AND student_id=12345: How many I/Os will this operation take?

First, we need to read the page into memory. We can use the index to perform the lookup, which takes (3 page reads for the index + 1 page read for the data page). We write the modification in memory and then flush the page back to disk -- since we already knew where the page was, we can immediately do the write without the corresponding lookup (+1 page write for the data page). This costs us a total of 5 disk I/Os.

Q7: In the best case, how many I/Os does it take to perform an equality search on grade_received?

Remember that in the best case, any record can match the grade_received predicate. Therefore, we must check every record of the table. This is equivalent to performing a table scan, and we would read every page, requiring 192 page reads.

Many Piazza posts were confused as to whether we meant “worst case” instead of “best case”. This question is in fact worded correctly - the best case is identical to the worst! Because there are no restrictions on how many times a single grade_received value shows up, we must always read every record to complete this query correctly.

Q8: We want to speed up the process of looking up students' grades by student_id, so we will add an index to our current schema. Which of the following indices will help us the most if each student submits many assignments?

- An unclustered index on (record_id, student_id) would not really help us because student_id would be lexicographically ordered second, and our query doesn’t concern record_id.
- A clustered index on grade_received might help us in a few select cases (e.g. we knew that the student either got a 0 or a 100 on every assignment) but wouldn’t help us otherwise.

This leaves us with an unclustered index on student_id and a clustered index on (student_id, time_submitted). Even though our query doesn’t concern time_submitted, student_id comes lexicographically first, so the index remains useful to us. Since each student submits many assignments, an unclustered index would require us to potentially perform many disk reads per matching record, while a clustered index would reduce this number by a factor of (page size / record size). Therefore, we would prefer the clustered index on (student_id, time_submitted).