4707 Lab4
Xi Chen 4626350
Shajiah Amin 4352468

SQL and Test Results
Part 1

1.no index

select * from student where age>20;
00:00:02.368006
select * from student where year=3;
00:00:01.103825
select * from student where gpa<3;
00:00:01.261218

Analysis: They are all using Seq Scan on student. Each one of these queries are range and equality searches. With range queries, you don't know how many rows the query will return because you don't know how many rows are within a lower or upper bound. The runtimes for these queries are correlated to how many records are returned that fulfill the WHERE clause. All these queries have to do a full table scan in order to find the rows that meet the WHERE clause requirement. If there was no WHERE clause, then the run times for all these queries would be exactly the same.

2.B tree

CREATE INDEX index_age ON student (age);
CREATE INDEX index_year ON student (year);
CREATE INDEX index_gpa ON student (gpa);
select * from student where age>20;
00:00:02.025662

use Seq Scan on student

select * from student where year=3;
00:00:01.206796

Bitmap Heap Scan

select * from student where gpa<3;
00:00:01.267591

Bitmap Heap Scan

Analysis: The run times with the B+ tree index compared to the run times without the B+ tree index are almost equivalent. The run time when we create the B+ tree index for each of the attributes from step 1 are almost exactly the same as when we don't have the index because by default, the dbms creates a B+ tree index on that attribute anyway, and this is especially true for range searches.

3. hash index

DROP INDEX index_age;
DROP INDEX index_year;
DROP INDEX index_gpa;
CREATE INDEX index_age ON student USING hash (age);
CREATE INDEX index_year ON student USING hash (year);
CREATE INDEX index_gpa ON student USING hash (gpa);
select * from student where age>20;
00:00:02.07742
use Seq Scan on student
select * from student where year=3;
00:00:02.253106
Bitmap Heap Scan
select * from student where gpa<3;
00:00:01.244387
use Seq Scan on student

Analysis: The result for these queries are almost the same as the original, which isn't that surprising since range searches don't benefit from hash indexes. However, it's surprising that the second query "select * from student where year=3;" is actually slower than it was using a B+ tree index since hash indexes are good for equality searches. With a B+ index, the run time was 01.206796 but with a hash index, it's 02.253106. Either way, it's a full table scan and doing sequential scans on the table, which means it does the same amount of work over the same number of rows..

4.

<u>Hash index</u>

select * from student where age>1;
00:00:02.109129
seq scan
select * from student where age>10;
00:00:02.43245
seq scan
select * from student where age>50;
00:00:00.99624
seq scan

B-tree index

select * from student where age>1;

00:00:02.102177

seq scan

select * from student where age>10;

00:00:02.420852

seq scan

select * from student where age>50;

00:00:00.954785

bitmap heap scan

Analysis: Since hash indexes aren't optimized for range searches, the run time is the same as it would be if there was no index. Also, the range of the WHERE clause definitely impacted the run time. This is just intuitive because by using common sense, all students would be older than 1, meaning that the first query would result in the greatest number of rows out of the three queries. However, not that many students are older than 50 so for the third query, not as many rows will be in the result. The results for the B+ tree index was surprising. The run times were really similar to the run times when using the hash index. Since it's a range search, I expected it to be a lot faster. For the first two queries, the run time results when using either indexes were almost identical, but for the third query, it was marginally faster by approximately 0.0414. Again, this is intuitive because the number of students older than 50 is a lot less than the number of students older than 1.

5.

hash index

select * from student where age=20;

00:00:00.002077

bitmap heap scan 0.

select * from student where age>10 AND age <20;

00:00:00.00177

seq scan

B-tree index

select * from student where age=20;

00:00:00.0034

bitmap heap scan

select * from student where age>10 AND age <20;

0.001653

bitmap heap scan

Analysis: As expected, the run time on the equality query with the hash index is faster than the one with the B+ tree index. It's not a huge improvement in run time, however, it's still faster. Also expected, the run time of the range search query is faster with a B+ tree index than a range search query with a hash index. Again, it's not much faster than the hash index, but at least it is faster. So I'm happy with that.

6.

select * from student where age>15 AND sex='male';

a. both attributes are non-indexed
   seq scan 0.006423

b. one of them is only indexed with either Btree or hash index
   age index with B tree
   seq scan 0. 002368

c. both of them indexed with similar indexes
   both indexed with B tree
   index scan using index_sex 0.008202
   both indexed with hash
   index scan using index_sex 0.007259

d. both of them indexed with different indexes
   age index with B tree,sex index with hash
   index scan using index_sex 0.002146
   age index with hash, sex index with B tree
   index scan using index_sex 0.002074

Analysis:

a.neither age or sex is indexed, so database did sequential scan.
for part b c and d . They are mixing among hash index, B tree index and non-index. B tree index is good for ranging query while hash index is good for equality index. Database did seq scan if there is no index on that element. And if there are two different index on sex and age, then database still gonna use index scan which is fast but not as fast as indexed with similar indexes.

Part2

1.

select * from student where age<30;
seq scan 00:00:01.989386
select DISTINCT * from student where age<30;
seq scan 00:00:02.069034

Analysis

The DISTINCT clause will sort all the rows in the result result and delete duplicates which takes longer to get result. This is because it needs to scan the result again to eliminate duplicates.

2.

//this is with the WHERE clause

```
SELECT S.year, S.age
FROM student S
WHERE year = 1 AND S.age = (SELECT MIN(S.age)
                            FROM student S
                            WHERE year = 1)
UNION
SELECT S.year, S.age
FROM student S
WHERE year = 2 AND S.age = (SELECT MIN(S.age)
                            FROM student S
                            WHERE year = 2)
UNION
SELECT S.year, S.age
FROM student S
WHERE year = 3 AND S.age = (SELECT MIN(S.age)
                            FROM student S
                            WHERE year = 3)
UNION
SELECT S.year, S.age
```

```
FROM student S
WHERE year = 4 AND S.age = (SELECT MIN(S.age)
                            FROM student S
                            WHERE year = 4)
UNION
SELECT S.year, S.age
FROM student S
WHERE year = 5 AND S.age = (SELECT MIN(S.age)
                            FROM student S
                            WHERE year = 5);
```

Hash Aggregate with seq scan on student
00:00:00.322177

//this is with the HAVING clause

```
SELECT S.year, MIN(S.age) AS Age
FROM student S
GROUP BY S.year
HAVING COUNT(*) > 0
ORDER BY S.year;
```

HashAggregate with seq scan on student
00:00:00.079231

Analysis: The HAVING clause is applied towards the end of query and there is almost no optimization taking place on the HAVING clause. The WHERE clause restricts how many rows are in the result before actually returning the result. However, the HAVING clause restricts how many rows are in the result *after* returning the result. Maybe this is why the query with the HAVING clause is faster

3.

//the first variant uses JOIN and WHERE

```
select DISTINCT dname from student
INNER JOIN major
ON student.sid=major.sid AND age>50;
HashAggregate  (cost=36057.53..36057.63 rows=10 width=3)
  Group Key: major.dname
  ->  Hash Join  (cost=4296.34..35660.90 rows=158654 width=3)
       Hash Cond: (major.sid = student.sid)
       ->  Seq Scan on major  (cost=0.00..17586.51 rows=1219151 width=7)
```

```
      ->  Hash  (cost=3971.00..3971.00 rows=26027 width=4)
            ->  Seq Scan on student  (cost=0.00..3971.00 rows=26027 width=4)
                  Filter: (age > 50)
 00:00:00.003343
```

//the second variant uses IN and a nested query.

```
select DISTINCT dept.dname from dep,major,student
where student.age>50 AND student.sid=major.sid AND
dept.dname=major.dname
AND student.sid in(
select student.sid from student,major
where student.sid=major.sid
group by student.sid
HAVING count(*)>0);
 HashAggregate  (cost=237170.57..237182.87 rows=1230 width=32)
   Group Key: department.dname
   ->  Hash Join  (cost=205310.19..236972.26 rows=79327 width=32)
         Hash Cond: (major.dname = department.dname)
         ->  Hash Join  (cost=205272.51..235843.83 rows=79327 width=3)
               Hash Cond: (major.sid = student.sid)
               ->  Seq Scan on major  (cost=0.00..17586.51 rows=1219151 width=7)
               ->  Hash  (cost=205109.84..205109.84 rows=13014 width=8)
                     ->  Merge Semi Join  (cost=157499.05..205109.84 rows=13014
width=8)
                           Merge Cond: (student.sid = student_1.sid)
                           ->  Index Scan using student_pkey on student
(cost=0.42..7178.42 rows=26027 width=4)
                                 Filter: (age > 50)
                           ->  Materialize  (cost=157498.63..197106.08 rows=200000
width=4)
                                 ->  GroupAggregate  (cost=157498.63..194606.08
rows=200000 width=4)
                                       Group Key: student_1.sid
                                       Filter: (count(*) > 0)
                                       ->  Merge Join  (cost=157498.63..186010.33
rows=1219151 width=4)
                                             Merge Cond: (student_1.sid = major_1.sid)
                                             ->  Index Only Scan using student_pkey on student
student_1  (cost=0.42..6678.42 rows=200000 width=4)
```

```
                                    ->  Materialize  (cost=157497.60..163593.36
rows=1219151 width=4)
                                          ->  Sort  (cost=157497.60..160545.48
rows=1219151 width=4)
                                                Sort Key: major_1.sid
                                                ->  Seq Scan on major major_1
(cost=0.00..17586.51 rows=1219151 width=4)
              ->  Hash  (cost=22.30..22.30 rows=1230 width=32)
                    ->  Seq Scan on department  (cost=0.00..22.30 rows=1230 width=32)
(25 rows)
 00:00:00.095988
```

Analysis: The run time for the query that used the JOIN and WHERE clauses was faster than the query that used IN clause as well as a nested query. It's clear that using idexes does make a difference on the speed and efficiency of the queries, although sometimes, it isn't that much of an improvement in speed. B+ tree index speeds up range searches and hash index speeds up equality searches.

Explanation:

B+ trees:

http://hackthology.com/lessons-learned-while-implementing-a-btree.html

It has optimal performance of disk reads for range queries because B+ trees perform well when executing range queries. Each node in the tree has N number of keys in each node and N is the degree of the tree. There are interior and exterior nodes. The interior nodes hold keys and pointers to nodes. The exterior nodes hold keys and their associated values. This indicates that the interior nodes have a different (usually higher) degree than the exterior nodes. B+ trees are optimized for disks. When you do disk reads, the disk returns the disk block where the bytes of info that you need are contained. B+ trees make their nodes the size of one disk block so this means there are fewer disk reads to find the value associated with the keys.

The leaves are all on the same level so the height of the tree is consistent. Also, all the leaves have a pointer to the next sibling, which is good for iterating through a sequential block of values and this is why range queries are really efficient. B trees are good for all comparison operators as well. And also the LIKE comparison.

Hash index:

https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html

They are good for equality operations like == or != but not comparison operators that find a range of values. Hash indexes are used because they are super efficient when looking up values. When queries need to compare values, a hash index created on that column being compared will be the key thats a pointer to the row that contains that value.