Question2:

Kim, Tae Young (tyk252)
Fang, Minda (mf3308)

The two rules of thumb concern with clustered & non clustered index.

New data is created for the test on clustered & non clustered index. There are 10 million rows of trades in the table.

In this part, our table has four attributes just like the trade table in Question 1: *stocksymbol*, *stocktime*, *quantity* and *price*. The quantity is uniformly ranging from 100 to 10000. The price, however, is uniformly ranging from 200 to 204 (which is a five point interval). There is another different table created in the part of Clustered Index. We will see in that part.

Clustered Index:

In this part, we are exploring if it is always better to have a clustered index depending on its effect on queries on other attributes.

In this part, we will run the same query on two different tables. One table is just the original stock table. The other one is all the same except the ***stocktime*** attribute, which has a type of **varchar**(200) instead of **INT**. Thus, the main difference between the two tables is that this new table's ***stocktime*** attribute has much larger size.

Here is test done in MySQL:

```
// Now there is no  primary key in this table stock.
mysql> alter table stock drop primary key;
Query OK, 1000000 rows affected (20.40 sec)
Records: 1000000  Duplicates: 0  Warnings: 0

select * from stock where quantity > 8000;
201858 rows in set (0.72 sec)

// Now primary key is added in this table stock.
mysql> alter table stock add primary key(stocktime);
Query OK, 0 rows affected (15.89 sec)
Records: 0  Duplicates: 0  Warnings: 0

select * from stock where quantity > 8000;
201858 rows in set (0.45 sec)

// Now there is no  primary key in this table stockNew.
mysql> alter table stockNew drop primary key;
Query OK, 1000000 rows affected (21.08 sec)
Records: 1000000  Duplicates: 0  Warnings: 0
select * from stockNew where quantity > 8000;
201858 rows in set (0.82 sec)

// Now primary key is added in this table stockNew.
mysql> alter table stockNew add primary key(stocktime);
Query OK, 0 rows affected (18.15 sec)
Records: 0  Duplicates: 0  Warnings: 0

select * from stockNew where quantity > 8000;
201858 rows in set (2.54 sec)
```

Here is test done in PostgreSQL:

// Now there is no primary key in this table **stock**.

```
postgres=# ALTER TABLE stock DROP CONSTRAINT stock_pkey;
ALTER TABLE
Time: 49.289 ms

postgres=# select * from stock where quantity > 8000;
Time: 438.350 ms
```

// Now primary key is added in this table **stock**.

```
postgres=# ALTER TABLE stock ADD CONSTRAINT MyPrimaryKey PRIMARY KEY
(stocktime);
ALTER TABLE
Time: 2540.293 ms

postgres=# select * from stock where quantity > 8000;
Time: 417.840 ms
```

// Now there is no primary key in this table **stockNew**.

```
postgres=# ALTER TABLE stockNew DROP CONSTRAINT stockNew_pkey;
ALTER TABLE
Time: 30.888 ms

postgres=# select * from stockNew where quantity > 8000;
Time: 494.375 ms
```

// Now primary key is added in this table **stockNew**.

```
postgres=# ALTER TABLE stockNew ADD CONSTRAINT YourPrimaryKey PRIMARY
KEY (stocktime);
ALTER TABLE
Time: 13466.720 ms

postgres=# select * from stockNew where quantity > 8000;
Time: 547.269 ms
```

Analysis:

Adding the clustered index in the table **stock** made the select query faster (in MySQL from 0.72 sec to 0.45 sec and in PostgreSQL from 438.350 ms to 417.840 ms). However, these two test also indicate that adding the clustered index in the table ***stockNew*** made query on selecting ***quantity*** within a range slower (in MySQL from 0.82 sec to 2.54 sec and in PostgreSQL from 494.375 ms to 547.269 ms).

For table ***stock***, since the primary key ***stocktime*** has type of INT, so it doesn’t have much overhead. In the table ***stockNew***, the primary key ***stocktime*** has type of VARCHAR(200), which is very large (could be up to 200 bytes). Obviously it takes longer time to traverse the B-tree.

To sum up, when applying clustered index to one attribute, we should not ignore the influence that the key size of the new clustered index brings. We'd better have clustered index on the keys which has relatively small size.

Non clustered Index:

There are two testing queries for the non clustered index testing:
1) select all trades info which has the same specific price;
2) select all trades info which has the same specific quantity;

Here is test done in MySQL:
// Now there is no non-clustered index on neither price nor quantity.
select * from stock where price = 203 limit 1000000;
→ 200431 rows in set (0.58 sec)

select * from stock where quantity = 500 limit 1000000;
→94 rows in set (0.45 sec)

// Now there is a non-clustered index on price.
create index price_index using btree on stock (price);
→Query OK, 0 rows affected (6.58 sec)
→Records: 0 Duplicates: 0 Warnings: 0

select * from stock where price = 203 limit 1000000;
→200431 rows in set (0.97 sec)

drop index price_index on stock;
→Query OK, 0 rows affected (0.21 sec)
→Records: 0 Duplicates: 0 Warnings: 0

// Now there is a non-clustered index on quantity.
create index quantity_index using btree on stock (quantity);
→Query OK, 0 rows affected (7.82 sec)
→Records: 0 Duplicates: 0 Warnings: 0

select * from stock where quantity = 500 limit 1000000;
94 rows in set (0.00 sec)

Here is the test done in PostgreSQL:
// PostgreSQL Part
// Now there is no non-clustered index on neither price nor quantity.
postgres=# select * from stock where price = 203 limit 1000000;
Time: 382.377 ms

postgres=# select * from stock where quantity = 500 limit 1000000;
Time: 236.736 ms

```
// Now there is a non-clustered index on price.
postgres=# create index price_index on stock (price);
CREATE INDEX
Time: 3256.831 ms

postgres=# select * from stock where price = 203 limit 1000000;
Time: 235.659 ms

postgres=# drop index price_index;
DROP INDEX
Time: 32.324 ms

// Now there is a non-clustered index on quantity.
postgres=# create index quantity_index on stock (quantity);
CREATE INDEX
Time: 2897.533 ms

postgres=# select * from stock where quantity = 500 limit 1000000;
Time: 1.691 ms
```

Analysis:

Indexes store their information in a balanced tree, referred to as a B-tree, structure, so the number of reads required to find a particular row is minimized.

From the result of testing of **MySQL**, we can see that having non clustered index on ***quantity*** fasten the query on selecting all trades info which has the same specific quantity (lowering the query time from 0.45 sec to almost 0.00 sec). However, having non-clustered index on ***price*** makes query on select all trades info which has the same specific price much slower than not having non-clustered index on ***price***(raising the query time from 0.58 sec to 0.97 sec).

From the result of testing of **PostgreSQL**, we can see that having non clustered index on ***quantity*** greatly fasten the query on selecting all trades info which has the same specific quantity (lowering the query time from 236.736 ms to 1.691 ms). Having non-clustered index on ***price*** makes query on select all trades info which has the same specific price has smaller improvement on the query time when compared to the improvement on the query of quantity (lowering the query time from 382.377 ms to 235.659 ms).

The non-clustered index on ***quantity*** works well because the distribution range of ***quantity*** is very large, and this query just selected a very small percentage of the whole record. The index added allows the query to hit fewer pages than it would by scanning the whole table.

The distribution range of ***price*** is very small and the query selecting all trades with the same price in the end selected about 1/5 of the whole record. The index, however, does not much lower the number of page accesses as compared to table scanning. Non-clustered index actually makes query performance on ***price*** improve not so much or even worse probably because the pages being accessed may be out of order and additional overhead of going through the B+tree greatly increases the query time.

This rule of thumb may be good when applying to an attribute which has a sparse distribution of data. And this rule works up against our expectation when applying to an attribute which has a rather dense distribution of data. This test clearly shows the benefit of non-clustered index and its side-effect as well.