HW2

1. Timing

Timing	Aquery	PostgreSQL
(a) Weighted avg price	0m10.635s	0m82.919s
(b) 10 moving avg	0m16.975s	0m139.05s
(c) 10 moving weighted avg	0m17.550s	0m151.16s
(d) Max positive price difference	0m16.261s	0m107.07s

Comparison:
Aquery did much better at time series query such as moving average because of its arrable data model. One of the reason that kdb+ is so fast that it does the operation in memory and its table is also loaded in memory. So when the q process dies, all data is gone. As for the postgresql, it loads data on disk and every time we read data from disk and write to disk while it also has cache mechanism for the performance.

2. We use trade.csv in question 1 as data source to do the tuning experiment with PostgreSQL and MySQL. In trade.csv, column stock is fractally distributed and column price is uniformly distributed. The tuning environment is Courant cims machine.

Rule of thumb 1: Covering index performs better than clustering index when first attributes of index are in the where clause and last attributes in the select. When attributes are not in order then performance is much worse.

PostgreSQL

According to the documentation of PostgreSQL, all indexes in it are secondary indexes, meaning that each index is stored separately from the table's main data area (which is called the table's heap in PostgreSQL terminology). This means that in an ordinary index scan, each row retrieval requires fetching data from both the index and the heap. Furthermore, while the index entries that match a given indexable WHERE condition are usually close together in the index, the table rows they reference might be anywhere in the heap. The heap-access portion of an index scan thus involves a lot of random access into the heap, which can be slow. To solve the performance problem, PostgreSQL support index-only scans, which can answer queries from an index alone without any heap access. To make effective use of the index-only scan feature, you might choose to create indexes in which only the leading columns are meant to match WHERE clauses, while the trailing columns hold "payload" data to be returned by a query. For example, if you commonly run queries like

```
SELECT y FROM tab WHERE x = 'key';
```

the traditional approach to speeding up such queries would be to create an index on x only. However, an index on (x, y) would offer the possibility of implementing this query as an index-only scan. Such an index would be larger and hence more expensive than an index on x alone, so this is attractive only if the table is known to be mostly static. It's important that the index be declared on (x, y) not (y, x), as for most index types (particularly B-trees) searches that do not constrain the leading index columns are not very efficient.

We experiment this tuning. In trade table we have columns(stock, date, quantity, price)

We execute the query:

```
SELECT stock FROM trade WHERE price = 300;  Time: 1769.172 ms
```

In traditional approach, we create an index on price only

```
CREATE INDEX idx_price ON trade(price);
SELECT stock FROM trade WHERE price = 300;  Time: 450.325 ms
DROP INDEX idx_price;
```

According to index-only scan feature, an index on (price, stock) would offer the possibility of implementing the query as an fast index-only scan.

So we create create index on (price, stock)

```
CREATE INDEX idx_price_stock ON trade(price, stock);
SELECT stock FROM trade WHERE price = 300;  Time: 407.012 ms
DROP INDEX idx_price_stock;
```

According to the note that index has to be declared on (price, stock) not (stock, price). We try this by

```
CREATE INDEX idx_stock_price ON trade(stock, price);
SELECT stock FROM trade WHERE price = 300;  Time: 1799.265 ms
DROP INDEX idx_stock_price;
```

Then we experiment on data in fractal distribution:

```
SELECT price FROM trade WHERE stock = 5353;  Time: 1656.320  ms

CREATE INDEX idx_stock ON trade(stock);
SELECT price FROM trade WHERE stock = 5353;  Time: 4.576 ms
DROP INDEX idx_stock;

CREATE INDEX idx_stock_price ON trade(stock, price);
SELECT price FROM trade WHERE stock = 5353;  Time: 6.720 ms
DROP INDEX idx_stock_price;

CREATE INDEX idx_price_stock ON trade(price, stock);
SELECT price FROM trade WHERE stock = 5353;  Time: 1666.992 ms
DROP INDEX idx_price_stock;
```

MySQL

According to MySQL documentation: In some cases, a query can be optimized to retrieve values without consulting the data rows. (An index that provides all the necessary results for a query is called a covering index.) If a query uses from a table only columns that are included in some index, the selected values can be retrieved from the index tree for greater speed:
So we experiment the tuning using similar queries as we use in PostgreSQL:

```
CREATE INDEX idx_stock ON trade(stock);
ALTER TABLE trade DROP INDEX idx_stock;
```

We execute the query:

```
SELECT stock FROM trade WHERE price = 300; Time: 0.8 sec = 800 ms
```

In traditional approach, we create an index on price only

```
CREATE INDEX idx_price ON trade(price);
SELECT stock FROM trade WHERE price = 300; Time: 0.15 sec = 150 ms
ALTER TABLE trade DROP INDEX idx_price;
```

We try an index on (price, stock) would offer the possibility of implementing the query as an fast index-only scan.
So we create create index on (price, stock)

```
CREATE INDEX idx_price_stock ON trade(price, stock);
SELECT stock FROM trade WHERE price = 300;  Time: 0.03 sec = 30 ms
ALTER TABLE trade DROP INDEX idx_price_stock;
```

According to the note that index has to be declared on (price, stock) not (stock, price). We try this by

```
CREATE INDEX idx_stock_price ON trade(stock, price);
SELECT stock FROM trade WHERE price = 300; Time: 3.04 sec = 3040 ms
ALTER TABLE trade DROP INDEX idx_stock_price;
```

Then we change the target column to another form of distribution

```
SELECT price FROM trade WHERE stock = 5353;  Time: 1.02 sec = 1020 ms

CREATE INDEX idx_stock ON trade(stock);
SELECT price FROM trade WHERE stock = 5353;  Time:  0.01 sec = 10 ms
ALTER TABLE trade DROP INDEX idx_stock;

CREATE INDEX idx_stock_price ON trade(stock, price);
SELECT price FROM trade WHERE stock = 5353;   Time: 0.02 sec = 20 ms
ALTER TABLE trade DROP INDEX idx_stock_price;

CREATE INDEX idx_price_stock ON trade(price, stock);
SELECT price FROM trade WHERE stock = 5353;  Time:  3.01 sec = 3010 ms
ALTER TABLE trade DROP INDEX idx_price_stock;
```

Conclusion:
For stock, the timing order from longest to shortest is: not ordered covering > no index > traditional index > ordered covering.
For price, the timing order from longest to shortest is: not ordered covering > no index > ordered covering > traditional index.
We can see from the results that if a wrong covering index is used, the performance will be even worse than querying without using index.

Rule of thumb 2: Indexes may be better or worse than scans
PostgreSQL
If the application issues queries such as

```
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire test1 table, row by row, to find all matching entries. If there are many rows in test1 and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method. But if the system has been instructed to maintain an index on the id column, it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.
We examine this rule by create index with different queries

First we find most and least frequent stock and price
select stock, count(stock) from trade group by stock order by count; 5353 (count: 62)
select stock, count(stock) from trade group by stock order by count desc; 38851 (count: 1011)
select price, count(price) from trade group by price order by price; 50 (count: 11442)
select price, count(price) from trade group by price order by price desc;   500 (count: 11748)

```
CREATE INDEX idx_price ON tradeIndex(price);
CREATE INDEX idx_stock ON tradeIndex(stock);
```

Multipoint query:

```
SELECT stock FROM trade WHERE price = 300;
```
Time: 1769.172 ms

```
SELECT stock FROM tradeIndex WHERE price = 300;
```
Time: 450.325 ms

```
SELECT price FROM trade WHERE stock = 5353;
```
Time: 1656.320 ms

```
SELECT price FROM tradeIndex WHERE stock = 5353;
```
Time: 4.576 ms

```
SELECT price FROM trade WHERE price = 300;
```
Time: 1918.216 ms

```
SELECT price FROM tradeIndex WHERE price = 300;
```
Time: 643.689 ms

```
SELECT stock FROM trade WHERE stock = 5353;
```
Time: 1712.494 ms

```
SELECT stock FROM tradeIndex WHERE stock = 5353;
```
Time: 2.648 ms

Range query:

```
SELECT stock FROM trade WHERE price < 300;
```
Time: 5652.495 ms

```
SELECT stock FROM tradeIndex WHERE price < 300;
```
Time: 6336.941 ms

```
SELECT stock FROM trade WHERE price > 300;
```
Time: 5056.970 ms

```
SELECT stock FROM tradeIndex WHERE price > 300;
```
Time: 4829.190 ms

```
SELECT stock FROM trade WHERE stock > 10000;
```
Time: 6812.183 ms

```
SELECT stock FROM tradeIndex WHERE stock > 10000;
```
Time: 7043.024 ms

MySQL

According to the documentation of MySQL, Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially. Indexes are less important for queries on small tables, or big tables where report queries process most or all of the rows. When a query needs to access most of the rows, reading sequentially is faster than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query.

```
CREATE INDEX idx_price ON tradeIndex(price);
CREATE INDEX idx_stock ON tradeIndex(stock);
```

Multipoint query:

```
SELECT price FROM trade WHERE price = 300;
```
Time: 0.75 sec = 750 ms

```
SELECT price FROM tradeIndex WHERE price = 300;
```
Time: 0.03 sec = 30 ms

```
SELECT stock FROM trade WHERE stock = 5353;
```
Time: 1.08 sec = 1080 ms

```
SELECT stock FROM tradeIndex WHERE stock = 5353;
```
Time: 0.01 sec = 10 ms

Range query:

```
SELECT stock FROM trade WHERE price > 300;
```
Time: 2.20 sec = 2200 ms

```
SELECT stock FROM tradeIndex WHERE price > 300;
```
Time: 1.99 sec = 1990 ms

```
SELECT stock FROM trade WHERE stock > 10000;
```
Time: 3.37 sec = 3370 ms

```
SELECT stock FROM tradeIndex WHERE stock > 10000;
```
Time: 4.76 sec = 4670ms

Conclusion: In the multipoint query, if we create indexes separately for two columns (fractally distributed data and uniformly distributed data), overall performance is better than scan while the improvement is much more impressive on the fractal distributed data.
In range query, if we create indexes separately for two columns fractally distributed data and uniformly distributed data), there is no improvement on the query performance and the performance is even worse.

Rule of thumb 3: Clustered index is twice as fast as non-clustered index and orders of magnitude faster than a scan.

PostgreSQL
When a table is clustered, it is physically reordered based on the index information.
We examine the tuning
Clustering index:

```
SELECT price FROM trade WHERE price = 300;
```
Time: 5158.014 ms
```
SELECT price FROM tradeIndex WHERE price = 300;
```
Time: 545.589 ms
```
SELECT price FROM tradeCluster WHERE price = 300;
```
Time: 39.335 ms

```
SELECT stock FROM trade WHERE stock = 40000;
```
Time: 1694.841 ms
```
SELECT stock FROM tradeIndex WHERE stock = 40000;
```
Time: 2.972 ms
```
SELECT stock FROM tradeCluster WHERE stock = 40000;
```
Time: 2.826 ms

MySQL
According to the documentation, when you define a PRIMARY KEY on your table, InnoDB uses it as the clustered index. Define a primary key for each table that you create. If there is no

logical unique and non-null column or set of columns, add a new auto-increment column, whose values are filled in automatically. If you do not define a PRIMARY KEY for your table, MySQL locates the first UNIQUE index where all the key columns are NOT NULL and InnoDB uses it as the clustered index. Accessing a row through the clustered index is fast because the index search leads directly to the page with all the row data. If a table is large, the clustered index architecture often saves a disk I/O operation when compared to storage organizations that store row data using a different page from the index record. All indexes other than the clustered index are known as secondary indexes.
We make a unique index on the column date (which is the only primary key in the table) to see the query performance on this column.

```
SELECT stock FROM trade WHERE date = 5000; Time: 0.73 sec = 730 ms

CREATE UNIQUE INDEX idx_date ON tradeCluster(date);


SELECT stock FROM tradeCluster WHERE date = 5000; Time: 0.02 sec = 20 ms
ALTER TABLE tradeCluster DROP INDEX idx_stock;
```

Conclusion: Generally, clustering index is faster than non clustering index.

3.

AQuery	Performance
Solution1: Query1	0m37.424s
Solution2: Query2	0m19.997s

Friends.csv has 750000 rows
Like.csv has 150000 rows
To circumvent superlinearity, we use additional dummy value column for a workaround.
In query1, we use friends relationship table to join friends relationship table and then remove the ones that are not satisfied.
In query2, we use friends relationship table to join like relationship table, and then remove the ones that are not satisfied.
Since the friends relationship table is much larger than like relationship table, query2 is faster than query1.