Database Systems Project 1 Q3

Keshav Singh, 281922

March 25, 2019

1 Specifications

The following specifications were followed for generating this report

1. Timing information was taken by using builtin junit timing
2. Everything was run using eclipse in Windows 10 OS running on i7 3.7 GHz processor with 16 GB RAM
3. There were slight deviations in every run, and bizarre anomalies due to Garbage collector, so the result of multiple runs were averaged.

2 Queries

2.1 Selection

Q1 = SELECT L_ORDERKEY L_PARTKEY FROM LINEITEM WHERE L_EXTENDEDPRICE> 30000

The result of executing Q1 on LINEITEM is shown in Table 1 below. As one can expect DSM is fastest since only the required columns are fetched for late execution and there is no overhead of calling next. Vector at a time is also close when the vector size is 100 . NSM is the slowest because of every column being fetched and overhead of next. PAX is slightly faster due to better layout and contiguous access from memory internally.

2.2 Aggregation

Q2 = SELECT AVG(L_EXTENDEDPRICE) FROM LINEITEM WHERE L_EXTENDEDPRICE> 30000

Data Model	NSM	PAX	DSM	DSM
Execution Type	Tuple-at-a-time	Tuple-at-a-time	Column-at-a-time	Vector-at-a-time
Time(seconds)	13	11.9	4.7	4.9

Table 1: Execution time for Q1 Select Query on different storage and execution techniques

Data Model	NSM	PAX	DSM	DSM
Execution Type	Tuple-at-a-time	Tuple-at-a-time	Column-at-a-time	Vector-at-a-time
Time(seconds)	12	9.6	3.2	4.1

Table 2: Execution time for Q2 aggregation Query on different storage and execution techniques

Data Model	NSM	PAX	DSM	DSM
Execution Type	Tuple-at-a-time	Tuple-at-a-time	Column-at-a-time	Vector-at-a-time
Time(seconds)	12.4	17.9	7.2	8.1

Table 3: Execution time for Q3 Join Query on different storage and execution techniques

When aggregate is added, one can see in TABLE 2 same proportion persists between different types of storage as when Select and Project was done. Aggregate makes PAX slightly faster, this could be also because 1 less column is selected. Also Column at a time DSM is faster than Vector at a time , which makes sense since no overhead because of next.

2.3 Join

Q3 = SELECT COUNT(*) FROM LINEITEM JOIN ORDERS ON ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY JOIN ORDERS AS O2 ON O2.O_ORDERKEY=ORDERS.O_ORDERKEY In this query, there were 2 joins, this was very interesting to observe and it was expected to see that the best performer would be DSM since late materialization that only the columns useful for join are retrieved until the very end of the query when aggregation is done, and then too, just the Column useful for aggregation is used. As expeced DSM with columnar operation works best here too ! Vector operation comes close, which makes sense that this join has been implemented in a hybrid manner ensuring complexity is O(M+N) in number of rows and columns. The row join has complexity O(N) but iterating over DBTuple has O(M) complexity for joining the tuples, thus overall complexity is O(MN) and emitted and the next overhead makes the execution slow. Thus PAX and NSM are much slower.

Size	1	200	400	800
Time(seconds)	12.5	7.6	8	8.1

Table 4: Execution time for different Vector Sizes

3 Conclusion

3.1 Volcano : NSM vs PAX

NSM and PAX have similar performance, but distribution of tuples across different pages makes PAX slower.

3.2 NSM Vs DSM (Column at a time)

NSM is amost always slower than DSM. The reasons are quite obvious, by having lazy approach to materializing data in DSM, our complexity is around O(N) where n is number of rows whereas NSM is around O(MN) where m is the number of columns, since we materialize the whole table at the end of every operation.

3.3 DSM : Column at a time Vs Vector at a time

Since we can fit all the data into the memory, the benefits of Vector at a time are not so visible right now, and by executing whole column and not losing time because of overhead caused by next, we can see that DSM with Column at a time is actually faster. In 4 we can see how having a small vector size , especially like 1 , which makes it closer to NSM can decrease performance by a lot.

4 Notes Regarding Late Materialization

The late materialization is implemented so that all the necessary columns are materialized only when aggregate is called or at the last operator before the data is emitted to the user. This was accomplished by passing the ids of row indices and column indices as output of every operator. Select and projection led to shrinking this Set of indices. In case of Hash join , additional lists were appended corresponding to every table. The scan operator was also necessary to pass down the pipeline in this architecture so that every operator can fetch just the column it needs. To account for joins, multiple scans were sent .