1. MCQ

1. B,C
2. A,E
3. B,D
4. C,E
5. A,E
6. C,E
7. B,C,E
8. 1 day's Order -> spread across a max 100 pages. We have 3000/300 = 10 records per page. At 100 orders per day, we will create 100/10 = 10 pages per day. We will reach 100 pages in 100/10 = 10 days after which the index will start to be an efficient approach. ANS B

2. TRUE/FALSE

9. FALSE
10. TRUE
11. FALSE
12. TRUE
13. TRUE
14. TRUE
15. FALSE
16. FALSE
17. TRUE
18. TRUE

3. Modelling

1. Add a many to many relationship from customer to bouquets with rank as an attribute of the relationship. We cannot enforce the values for rank in this approach. Alternatively, introduce a weak entityset with (**rank**) with relationships to both customer (parent) and bouquets. This can enforce that (email, rank) be unique in relational translation, but rank might still not be 1, 2. ..

2.

(A) add stockqty to Flowers

(B) UPDATE Flowers SET stockQty = stockQty+100 WHERE genericFname = ‘daisy pom’ and colour = ‘pink’

(C) We need to update stockqty whenever a bouquet is made which uses these flowers by decrementing it with the number of flowers required for the bouquet.

3.

```
SELECT bName FROM bouquets WHERE bName NOT IN
(SELECT bName FROM orderDetails WHERE orderId in
        (SELECT orderId FROM orders WHERE orderDate >= ‘2016-01-01’))
```

4.

```
SELECT bName
FROM orderDetails
WHERE bName <> ‘Spring Maiden’ AND orderId IN
        (SELECT orderId FROM orderDetails WHERE bName = ‘Spring Maiden’)
GROUP BY bName
ORDER BY COUNT(*) DESC
```

5.

```
SELECT genericFname, colour, SUM(flowerQty*orderQty)
FROM Orders O, OrderDetails D, bouquetContains B
WHERE O.orderid = D.orderID AND D.bName = B.bName
          AND O.deliveryDate = ‘2016-04-31’
GROUP BY genericFname, colour
```

6.

```
orders4day = FILTER Orders BY deliveryDate = ‘2016-04-31’;
od = JOIN orders4day BY (orderId), OrderDetails BY (orderId);
bq = JOIN od BY (bName), bouquetContains BY (bName);
fls    = FOREACH bq GENERATE genericFName,colour, flowerQty*orderQty as fQty
grp = GROUP fls BY genericFName,colour ;
gen = FOREACH grp GENERATE group, SUM(fQty);
```

4. Query Evaluation

1. For person, we have 200 occupations, giving about 100,000/200 = 500 persons per occupation, which is less than the total number of data pages (3,000). Hence even a regular index on occupation would be beneficial (1 leaf + 500 data page IO) whereas a clustered index on it could make it more effective, resulting in
(1 leaf + (RF=500/100,000) * (3,000 data pages)) = 16 IO.

Project only required attributes (name, login) from Person.
Size of this tuple will be (20 (name) + 4(INT, sender)) = 24,
which takes 24*500 = 12,000 Bytes
This can be stored in 3 Buffer frames. (But in reality pipeline to the join operator)

There are 2,000,000/100,000 = 20 messages per person.
In our case we need to retrieve, 20 * 500 = 10,000 messages in total.
Even with a regular index, this will only be 500*(1 leaf + 20 data page) IO, cheaper than a full table scan.

But we can also use an index clustered on sender, that will result only in 500 (1 leaf + 1 data page) IO.

Therefore we have a case for a nested index join where the Message is the inner relation and the Person is the outer relation, which itself can be read from using a clustered index.

Performing the join and pipelining to projecting only required field from Person and Messages will take (20 + 4 + 4) = 28 bytes, and can be stored in 28 * 10,000 / 4,000 = 70 buffer frames.

This can then be sorted in-memory based on an in-memory sorting algorithm on m.sender, p.name, m.receiver

The group by operator can scan over the sorted contents and produce the final output in a pipelined fashion.

2. Although one could argue that there can be only about 100 or so possible values for birth year, therefore getting some benefit from using an index on it, in reality we are not given any information about the possible distribution of birthyear (and so database treats it like a regular integer field), hence a full table scan might be required on the Person table. Since we do not have any cardinality information on Person, we cannot predict how many records needs to be read from Messages. Therefore no potential of using index on it either.

Here our best bet would be to pipeline the selection on Person to the projection of required fields (name, sender,login) from person (28 bytes per tuple), fit it into 100 buffers (about 14,000 Person info) and perform a block nested join with Messages as the inner relation. So worst case we would end up reading Messages about 8 times. So the IO cost would be roughly the cost of reading all pages of Persons + 8*Number of Pages in Messages.

3.

Mapper:

For each tuple t_m in Messages, output t_m.mid, t_m

For each tuple t_t in MessageText, output t_t.mid, length(t_t.mtext) as t_t.mlen

Reducer(mid, (t_x1, t_x2))

t_m = find tuple from (t_x1, t_x2) where t_xi is of type Messages.

t_t = find tuple from (t_x1, t_x2) where t_xi is of type MessageText.

Output (mid, t_m, t_t.mlen)

4.

(a) (From bottom)

Number = 2,000,000 Size = 4+20+3+4 = 31

Number = 1,000 (because there are 2,000 zip codes, 1/2000 chance of it being the same zip code as the sender). Size = 4+20+4+20 = 48

(b) 50 persons per each of 2000 zip. 50x50 matches per zip. Total records = 2000*(50*50) = 5,000,000 , size (assuming same attributes as before 31)

5. Concurrency Control

1. T1 -> T2 , T2 -> T3, T2 -> T1, T3 -> T2, T4 -> T2, T2 -> T4. Not serializable due to cycles.

2.

(A) Lost Update t10 overwritten at t12

(B) Dirty Read t4 and t6 (T2’s write is not committed yet)

(C) Unrepeatable Read t1 unrepeatable at t6

3

A

	T1	T2	T3	T4	a	b	c
	s1(a)				S1		
	r1(a)				S1		
		X2(a)			S1, WL(X2)		
			S3(b)		S1, WL(X2)	S3	
			r3(b)		S1, WL(X2)	S3	
			S3(a)		S1, WL(X2, S3)	S3	
	r1(a)				S1, WL(X2, S3)	S3	
	c1				S1, WL(X2, S3)	S3	
	U1(a)				X2, WL(S3)	S3	
		w2(a)			X2, WL(S3)	S3	
				S4(c)	X2, WL(S3)	S3	S4
		X2(b)			X2, WL(S3)	S3,WL(X2)	S4
		w2-'(a)			X2, WL(S3)	S3,WL(X2)	S4
		Abort			X2, WL(S3)	S3,WL(X2)	S4
		U2(a)			S3	S3	S4
			r3(a)		S3	S3	S4
			c3		S3	S3	S4
			U3(a,b)				S4
				X4(c)			X4
				w4(c)			X4
				c3			X4
				U3(c)			

B. We ignore T2 which is aborted, so one of the possible serial schedule will be T1->T3->T4 . You could also do it T3->T1->T4

4.

	T1	T2	T3	T4	a	b	c
	S1(a)				S1		
	r1(a)				S1		
	U1(a)						
		X2(a)			X2		
		w2(a)			X2		
			S3(b)		X2	S3	
			r3(b)		X2	S3	
			U3(b)		X2		
			S3(a)		X2,WL:S3		
	S1(a)				X2,WL:S3,S1		
				S4(c)	X2,WL:S3,S1		S4
				r4(c)	X2,WL:S3,S1		S4
				U4(c)	X2,WL:S3,S1		S4
		X2(b)			X2,WL:S3,S1	X2	
		W2(b)			X2,WL:S3,S1	X2	
		X2(c)			X2,WL:S3,S1	X2	X2
		w2(c)			X2,WL:S3,S1	X2	X2
		c2			X2,WL:S3,S1	X2	X2
		U2(a,b,c)			S3,S1		
	r1(a)		r3(a)		S3,S1		
	U1(a)		U3(a)				
	c1		c3				
				X4(c)			X4
				W4(c)			X4
				c3			X4
				U4(c)			