1. **RDBMS**

   a) The database design is poor for the following reasons:
   
   - customer names are repeated many times (degeneracy/redundancy)
   - having an attribute for each item is a bad design choice since it is wasteful (often 0 are ordered), it makes it hard to design queries, and means that queries must be altered when new items are added to the database
   
   The following additional attributes are required for processing orders:
   
   - customer addresses for delivery of the order
   - a customer identification number as a unique reference number (and primary key)
   - value the cost of the order
   
   The following ER diagram shows a redesign that addresses the problems mentioned above and includes the missing attributes:

   ![ER Diagram](image)

   b) There are five tables in the updated database:
c) Orders can be fulfilled in three steps

pack: SELECT ORDER-ITEM.OrderID, ITEM.Name, ORDER-ITEM.Quantity
     FROM ORDER-ITEM, ITEM WHERE ORDER-ITEM.ItemID = ITEM.ItemID
     AND OrderID = "123"

dispatch: SELECT ORDER-CUSTOMER.OrderID, CUSTOMER.Name, CUSTOMER.Address FROM ORDER-CUSTOMER, CUSTOMER WHERE ORDER-CUSTOMER.CustomerID = CUSTOMER.CustomerID AND OrderID = "123"

charge: SELECT ORDER-CUSTOMER.OrderID, CUSTOMER.Name, ORDER.Value FROM ORDER-CUSTOMER JOIN ORDER ON ORDER-CUSTOMER.OrderID = ORDER.OrderID JOIN CUSTOMER ON ORDER-CUSTOMER.CustomerID = CUSTOMER.CustomerID WHERE ORDER.OrderID = "123"

d) The following extension can be used to store the tester’s name and the items which were tested:

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>Attribute - primary key</th>
<th>Attribute* - foreign key</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeID</td>
<td>Name</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORDER-TESTER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderID*</td>
<td>EmployeeID*</td>
</tr>
</tbody>
</table>

e) The following query will enable the company to see who (if anyone) inspected order 123:

SELECT ORDER-TESTER.OrderID, EMPLOYEE.Name FROM ORDER-TESTER, EMPLOYEE WHERE ORDER-TESTER.EmployeeID = EMPLOYEE.EmployeeID AND ORDER-TESTER.OrderID = "123"
a) The ER diagram is:

```
allows for the possibility
that a film has several directors

person
Name ID
acts in
film
M
directs
FilmName FilmID
M
M
M

\( \text{Attribute} = \text{primary key} \)
\( \text{Attribute}^* = \text{foreign key} \)
```

b) Four relations are required:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Attributes</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSON</td>
<td>ID, Name</td>
<td></td>
</tr>
<tr>
<td>FILM</td>
<td>FilmID, FilmName</td>
<td></td>
</tr>
<tr>
<td>ACTOR-FILM</td>
<td>ActorID*, FilmID*</td>
<td>(from &quot;acts in&quot; relationship)</td>
</tr>
<tr>
<td>DIRECTOR-FILM</td>
<td>DirectorID*, FilmID*</td>
<td>(from &quot;directs&quot; relationship)</td>
</tr>
</tbody>
</table>

c) Relational algebra for the query:

\[
\prod_{\text{Name, Name'}} \text{ACTOR-FILM} \bowtie_{\text{FilmID} = \text{F}} \rho_{\text{AF}(\text{A,F})}(\text{ACTOR-FILM}) \bowtie_{\text{ActorID} = \text{ID}} \rho_{\text{P}(\text{I}, \text{Name'})}(\text{PERSON})
\]

SQL code for the query:

```sql
SELECT DISTINCT C.Name, D.Name FROM ACTOR FILM AS A JOIN ACTOR FILM AS B ON A.FilmID = B.FilmID JOIN PERSON AS C ON A.ActorID = C.ID JOIN PERSON AS D ON B.ActorID;
```

d) Relational algebra for the query:

\[
\prod_{\text{J}, \text{L}} \rho_{\text{U}(\text{A,B})}(\text{ACTOR-FILM}) \bowtie_{\text{B} = \text{G}} \rho_{\text{V}(\text{C,D})}(\text{ACTOR-FILM}) \bowtie_{\text{C} = \text{E}} \rho_{\text{W}(\text{E,F})}(\text{ACTOR-FILM}) \bowtie_{\text{F} = \text{H}} \rho_{\text{X}(\text{G,H})}(\text{ACTOR-FILM}) \bowtie_{\text{A} = \text{I}} \rho_{\text{Y}(\text{I,J})}(\text{PERSON}) \bowtie_{\text{H} = \text{G}} \rho_{\text{Z}(\text{K,L})}(\text{PERSON})
\]

SQL code for the query:

```sql
3
SELECT DISTINCT E.Name, F.Name FROM ACTOR_FILM AS A JOIN ACTOR_FILM AS B ON A.FilmID = B.FilmID JOIN ACTOR_FILM AS C ON B.ActorID = C.ActorID JOIN ACTOR_FILM AS D ON C.FilmID = D.FilmID JOIN PERSON AS E ON D.ActorID = E.ID JOIN PERSON AS F ON A.ActorID = F.ID;

3  a) The query can be compactly expressed using relational algebra as:

\[ \Pi_{\text{Room}} \sigma_{\text{Name} = "A. Smith", \text{Day} = "Tuesday", \text{Hour} = "10"} \left( \text{CR} \bowtie \text{CDH} \bowtie \text{SNAP} \bowtie \text{CSG} \right) \]

b) The corresponding expression tree is:

```
  /--\Room
   \  /--\Name="A. Smith", Day="Tuesday", Hour="10:00"
     \  /--\CR
        \--\CDH
            \--\SNAP
                \--\CSG
```

c) The optimised expression tree is:

```
  /--\Room
   \  /--\CR
        \--\CDH
            \--\SNAP
                \--\CSG
```

d) Suppose we have a subexpression \( \pi_L(R) \) that is part of a larger expression, and let \( R \) be a single relation (rather than an expression involving one or more occurrences of operators). Suppose also that above this subexpression in the expression tree is another projection. To perform the projection on \( R \) now requires us to examine the entire relation, regardless of the existence of indexes. If we instead carry along the attributes of \( R \) not on the list \( L \), until the next opportunity to project out those attributes, we are frequently able to save a significant amount of time.
For instance, in part (c) the subexpression $\pi_{\text{CourseID, StudentID}}(\text{CSG})$ which has the effect of getting rid of grades. Since our entire expression eventually focuses on a few tuples of the CSG relation, we are much better off projecting out grades later; by so doing, we avoid ever examining the entire CSG relation. The resulting expression tree is:

![Expression Tree]

\textit{Transaction Processing}

4 \hspace{1cm} \textit{This question appeared in the 2013 Tripos exam.}

(a) Multiple transactions can have read locks on the same resource, but a write lock is exclusive. Since there are multiple locks on resources $A$, $B$, $C$ and $D$ these must be read locks. Since two locks remain, they must both be write locks: $T_6$ has a write lock on $E$, and $T_4$ has a write lock on $F$.

(b) \textbf{BOOK WORK:} A wait-for graph represents the waiting relationships between current transactions. The nodes represent transactions. A directed edge from node $T_1$ to node $T_2$ implies transaction $T_1$ is waiting for transaction $T_2$ to release a lock.

The wait-for graph for the scenario given in the question is:

![Wait-for Graph]
(c) The wait-for graph contains directed cycles and is therefore in deadlock. All of the directed cycles pass through $T_4$ and so aborting this single transaction resolves deadlock. The transactions then complete in this order: $T_6, T_2, T_1, T_3, T_5$.

5 (a) Pessimism and optimism are two extremes of transactional concurrency control policy. The main difference between optimism and pessimism with reference to the ACID properties is isolation. Pessimistic policies exhibit ‘strict isolation’ using locking systems. This means that no transaction can access an object that has had a write operation performed on it until the transaction that performed the write operation has committed. More optimistic policies allow other transactions to perform operations on objects before a previous invoker has committed. If a potential conflict is subsequently detected then all of the transactions involved are aborted. The difference is a tradeoff between efficiency and the likelihood of an abort. If simultaneous access to an object is unlikely, then optimistic strategies will be more efficient. Alternatively, if simultaneous access is frequent, then pessimistic strict locking will probably be better although note that dead-lock is more likely with pessimistic schemes. The locking mechanism itself carries an overhead, hence optimistic strategies are also preferred for cases where data is read frequently but updated rarely.

(b) i. A pessimistic policy would be more efficient since there can be a large number of potential concurrent accesses associated with a flight booking system (e.g. when a large group of people need to book on the same flight).

ii. An optimistic policy would lead to a more efficient implementation since it is likely that updates will be reasonably infrequent but many concurrent reads may occur when several interested parties review a case.

iii. An optimistic policy would be more efficient. The assumptions above about criminals hold for patients. Also, it is important to be able to get a potentially life-saving information quickly and optimistic methods do not delay at transaction startup since they never have to wait for locks.

6 (a) The table below shows in bold the time at which each transaction waits for a lock to be released (no subsequent actions will be processed). At time=13 deadlock occurs. (Note that if the fact that transaction 3 waits at time=6 is missed, then deadlock may be found at time=10). Transaction $T_2$ will complete.
The wait-for-graph when deadlock occurs is:

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \]

Suggestion: Once the students understand this question, you might like to point them to question 4 on the 2009 Tripos paper (http://to.eng.cam.ac.uk/teaching/past_papers/part2a/) which is of the same kind, but trickier.

The crib is a little terse for this question and so it might be useful to know that the possibly mysterious edge between T1 and T4 in the wait-for-graph occurs at t=7. T1 ends up waiting at t=6 (to acquire a write lock on B). At t=7, T4 can still obtain a read lock on B though (since only read locks are currently given out – it does not need to wait and it sneaks in, T1 must now wait for it too).

At deadlock the other transactions are waiting at the following points: T2 waits to obtain a write lock on D (t=10), T4 waits to obtain a write lock on D (t=11), T3 waits to obtain a read lock on A (t=12).