Sample Solutions

Question 1

Let |R| and |S| denote the number of pages of R and S respectively.
Grace hash join:

Cost = read R + write k partitons + read k partitions +
read S + write k partitions + read k partitions
= 3 * (|R| + |S|)

Hybrid hash join:
Cost = read R + write k-1 partitions + cost to read k-1 partitions +
read S + write k-1 partitions + cost to read k-1 partitions
= (|R| + (k-1)/k*|R| + (k-1)/k*|R|) + (|S| + (k-1)/k * |S| + (k-1)/k * |S|)
= (3 – 2/k)(|R| + |S|)

Since (3 – 2/k) < 3, Hybrid hash join is always better (in terms of I/O) than Grace hash join. Hybrid hash join can also returns initial answers much earlier – GRACE has to wait for the partitioning phase to complete. Observe also that the savings is more with smaller k.

Note that in actual fact, it is possible for bucket zero to be larger/smaller than other partitions. It is also lightly that the number of buckets under GRACE and Hybrid be different.

The above discussion assumes data are uniformly distributed across the k partitions. Note that you need to check that the buffer space is enough. Straightly speaking, to partition R, we need 1 input buffer, (k-1) output buffer (for the k-1 partitions to be written out) and the remaining space to build the first partition in memory. Don't forget 1 output buffer for the join result when partitioning S. So, assuming we have B buffers, we have B-2-(k-1) buffers for the partition in memory.

Question 2

Example:
Relation R

rid	A	B
1	3	3
2	4	4
3	5	5
4	6	6
5	7	7

Relation S

sid	D	E
1	1	1
2	2	2
3	3	3

4 3 4
5 7 7

R JOIN S (R.A=S.D)

A	B	E	rid	sid
3	3	3	1	3
3	3	4	1	4
7	7	7	5	5

Join Index

rid	sid
1	3
1	4
5	5

Advantages: No need to perform join processing – simple table-lookup like an index. Initial response time is good. As you scan the index, you retrieve the corresponding records, and output the answers. Disadvantages: Updates may be costly to maintain. For example, insert (6, 3, 4) into R imply that we must also compute join of (6, 3, 4) with S to get the tuples that are in the join index. Same problem with deletions – some records in the join index must be removed. Therefore, join index is useful only for static databases.

It is also possible that the performance of join index is worse than processing the join from scratch if the result size is large.

Question 3
(A) Query. Find applicants that earn a salary greater than 60000 and have GMAT scores higher than the average score of applicants from USA.

This query is an uncorrelated nested query, i.e., the inner subquery is independent of the outer subquery.

SQL:

```
SELECT  al.pid, a1.income
FROM    applicant a1
WHERE al.income > 60000
AND    a1.gmat >
       (SELECT avg(a.gmat)
        FROM    applicant a, location l
        WHERE a.cityid = l.cityid
        AND    l.country = "USA");
```

Evaluation strategy:
a) Process the inner query completely to get a single average value
- do selection on relation location using the predicate l.country = "USA"
- execute a nested loops join (or any join algorithm) between relation applicant and the selected tuples
- find the average gmat of the resulting tuples from the join result

b) Process the outer query

- do selection on relation applicant using the predicates a1.income > 60000 and a1.gmat > average gmat
- project the attributes pid and income of the resulting tuples

(B) Query. Find applicants that earn a salary greater than 60000 and have GMAT score higher than the average score of applicants from the same US city.

This nested query is a correlated query: the inner query is "different" depending on the tuples of the outer query being processed.

SQL:

```
SELECT  al.pid, a1.income
FROM    applicant a1
WHERE al.income > 60000
AND    a1.gmat >
        (SELECT avg(a2.gmat)
         FROM    applicant a2, location l2
         WHERE a1.city = a2.city
        AND    a2.city = l2.city
        AND    l2.country = "USA");
```

(NOTE: If the inner query returns empty set, e.g., a non-USA city, then the semantics for avg is not defined. The answer will be NULL, and g1.gmat > NULL is FALSE)

Naïve evaluation strategy:
For each tuple of applicant a1,
a) Check whether a1.income is greater than 60000. If yes, for each tuple of applicant a2,
   - Check whether a1.city = a2.city
   - Do selection of location using l.country = "USA"
   - Check whether a2.cityid = l.cityid of the resultant tuples
b) Find average of the resultant tuples' gmat
c) Check whether a1.gmat > resultant average gmat
d) Project pid and income

However, this method is computationally expensive and the subquery has to be evaluated for each tuple of relation applicant. A more efficient evaluation of the query is to transform the previous query into the following:

Inner Query:

```
(SELECT cityid, avg(a.gmat)
FROM     applicant a, location l
WHERE a.cityid = l.cityid
AND    l.country = "USA"
GROUP BY  cityid);
```

Outer Query:

```
SELECT  a.pid, a.income
FROM     applicant a, tmpAgg t
WHERE a.income > 60000
```

```
AND    a.cityid = t.c1
AND    a.gmat > t.c2
```

Where tmpAgg is the temporary relation for the inner query and c1 and c2 are the first and second columns in tmpAgg respectively.

In SQL, this can be expressed in a compact form:

```
SELECT  a.pid, a.income
FROM     applicant a,
       (SELECT city, avg(a.gmat)
        FROM     applicant a, location l
       WHERE a.cityid = l.cityid
       AND    l.country = "USA"
       GROUP BY  cityid) as t;
       WHERE a.income > 60000
       AND    a.cityid = t.c1
       AND    a.gmat > t.c2
```