Contributors:
Menelaos Kokolios

1. (a) What were the heuristics applied in System-R's optimiser and why were they applied? In particular, explain why the optimiser chose a specific type of plan. [7 marks]

   (b) Consider the following algorithm for join processing of integer keys.

   - Assume there are two inputs $R$ and $S$. Let $R$ be the smaller input. Let the join predicate be $R.x = S.y$.
   - Let the domain of the join key be $[0, D)$.
   - Generate $k$ partitions for $R$. To decide the partition $p$ of a record $r \in R$ apply the following function: $p = \lfloor \frac{r.x}{k} \rfloor$.
   - Generate $k$ partitions for $S$. To decide the partition $p$ of a record $s \in S$ apply the following function: $p = \lfloor \frac{s.y}{k} \rfloor$.
   - Process $R$ and $S$ partition by partition. For each of the $2k$ partitions ($k$ partitions for each input) $R_i, S_i, i \in [0, k)$:
     - load $R_i$ in memory,
     - build a hash table for it,
     - scan $S_i$ and probe $R_i$ for matches,
     - for all matching records, output them.

   i. Devise an I/O cost expression for this algorithm, explaining its various factors. What special properties does the output have in terms of the order of the values of the join output? Explain whether or not a System-R style query optimiser could take advantage of them. [5 marks]

   ii. When will this algorithm perform well and when will it behave badly? [5 marks]

   iii. How can you improve the performance of this algorithm? [8 marks]

(a)
Is this maybe to do with the whole "cheapest acces method overall" plus the cheapest of each access method in an interesting order? Not sure how to spin that out for 7 marks though...
*Actually, I believe this should be related to dynamic programming.*
I would have mentioned how it pushes selections and projections down, and tries to delay cartesian joins for as long as possible (ie, reduces the number of relations first, if it can). It's limited to nested loops and sort-merge and left-deep plans though.

It uses equivalence rules to push down selections and projections. It delays cartesian products. It does so in order to minimise the input cardinality and the memory requirements of the join. Finally the optimiser uses a cardinality based cost model to pick, not always the cheapest plan overall, but usually one of comparable cost.

(b)
(i)

In fact, this is the Grace hash join algorithm. Let the number of pages for R is Pr and the number of pages S is Ps. Then in the buliding phase, we should read and write R and s once respectively:  2Pr + 2Ps.  In the probe phase, we have to scan R and S once again. So the total cost is 3(Pr + Ps).  Basically, the special property we us have hash join is the tuples with the same values on the join attributes are together. Yes, A System R style query optimiser could take advantage of it. For example, if we need to project the join attributes (select distinct), due to the tuples with the same values on the join attributes are together, System R query optimiser know it can eliminate the duplicates easily if it uses hash join, rather than select nested loop join.

As the output is a list of partitions, it may be posible to use pipelining. Send partitions as generated instead of wait for the hole input to be processed.

(ii)
After the partion phase,
if the number of  values in all partions are uniform distributed, it will have a good performance.

On contrast, the the case of skew, it behaves badly

I agree with the skew-data answer, but I would add up to that that we need at least m pages on the buffer pool (m is the number of partitions) and that B > sqrt(f * Pr). In case we dont have that memory grace join will behave badly as well.

Assuming that we want the algorithm to produce m partitions and that each partition is of approximately equal size, the only way that this hash function could possibly do this is by having D = k^2 and by having r.x and s.y uniformly distributed in that range. This is because we have that:

p = floor(a/k) = 0
p = floor(b/k) = k
=> a = [0, k) and b = [k(k-1), k^2)

If D is small, for instance D = k, then this would mean that all the keys would fall into the 0th bucket, therefore, the problem wouldn't even be reduced. Also, if D is too large, i.e. D>k^2, this means that keys would fall into buckets that would be greater than k => which is not what we want either.

(iii) Choose an appropriate hash function to make each bucket have proper and well-distributed data.
(How can it be worth 8 marks?)
Use the hybrid hash join? So you keep partition R1 in memory at all times, slightly reducing number of writes necessary +1
But hybrid hash join requires extra memory.
I agree with hybrid to improve performance in the generic situation (no skew data), but I don't see how it solves the skew data problem.

2. (a) Give three methods to partition data in a parallel database system. Explain the advantages and disadvantages of each method. [*4 marks*]

   (b) Consider a parallel database system that can use any of the three methods you identified above. Assume that the system needs to sort a partitioned table on the partitioning attribute. For each method give a sorting algorithm which performs no more than two full passes over the data. [*6 marks*]

   (c) Consider external sorting with replacement selection, as it was discussed during the lectures. In contrast to standard external sorting, replacement selection will create runs of (potentially vastly) different sizes on disk. Assume that the number of buffer pool pages is much smaller than the number of runs, so multiple merging phases will be necessary. How would you optimise the order in which you merge the runs and why? [*8 marks*]

   (d) Consider two inputs $R$ and $S$ that will be joined using sort-merge-join. Assume that there are $B$ pages available for the computation and multiple merging phases will be necessary (*i.e.*, the sizes of both inputs is much greater than $B$ pages). During the lectures, it was mentioned that the final merging phase of external sorting can be combined with the merge join phase of sort-merge-join. Describe such an approach. [*7 marks*]

(a)
Range partition
Range partition is good for equi-join and range-queries. as well as aggregation. But range partition is problematic with skew.
Hash partition
Similarly, Hash partition is good for equi-join as well.
It is also problematic with skew.
Round-robin partition
Round-robin partition is good for situations where you need to retrieve all the tuples of a given relation.

(b)
For range partition, the partitioned table has been sorted on the partitioning attribute after partitioning. For Hash partition,
Range Partition; External merge-sort- range-partitioning sorts the relations, so only the merging phase left
Hash Function;

Range Partition: We just need to sort locally -> 1 pass over the data
Hash Partition: First we range partition and then we sort locally -> 2 passes over the data
Round Robin: The same as hash partition

(c)Loser tree?
In every pass merge the k biggest ( k max number of merges) until nothing left. Repeat all merged . Why? The bigger the runs the lower height of the merge tree

I would merge the runs going from the smallest to the biggest. This way I make sure I dont read the big runs many times. Example with B = 3 (2 to merge, 1 for output):
I have 4 runs
A: 8 pages, B: 8 pages, C: 2 pages, D: 2 pages

By merging the big first

I merge A and B => 32 I/Os
AB = 16 pages
I merge AB with C => 36 I/Os
ABC = 18 pages
I merge ABC with D => 40 I/Os
Total = 108 I/Os

Or by merging the small first:

I merge C with D => 8 I/Os
CD = 4 pages
I merge CD with B => 24 I/Os
BCD = 12 pages
I merge BCD with A => 40 I/Os
Total = 72 I/Os

I think that actually you should merge at every step the two that are most similar in size.

Example:

Imagine we have 1 file that is of size N and 8 of size n (where N >> n e.g. N=1000n). Assume we always only merge two files for simplicity but wlog.
Lets try three different ways:

Case A: Merge largest always:
Here, the amount of work done (just in terms of pages accessed not in terms of IOs) is:

N + n
(N + n) + n
…
(N+7n) + n

which gives a grand total of:
\sum_{i=1}^8 (N+in) = 8N+36n

Case B: Merge smallest first:

Amount of work done:

n+n
2n+n
…
7n+n
8n+N

Giving a total of:

N +\sum_{i=1}^7 n(i+1) = N + 28n + 7n = N + 35n

Case C: Always merge smallest of similar size
Here we try to always maintain the smallest of similar size to each other => We do:

4 times: n + n to get 4 2n files
2 times: 2n + 2n to get 2 4n files
1 times: 4n + 4n to get 1 8n file
then: 8n + N =>

This gives a total of: 8n + 8n + 8n + 8n + N = 32n + N

Which is the smallest.

(d)The join is evaluated in two phases
First, the two input relations are sorted on the join attribute
Then, they are merged and join results are propagated

3. (a) Give the definition of a phantom in transaction processing and an example of when phantoms arise. [5 marks]

   (b) Explain briefly the mechanism database systems employ to deal with phantoms. What transaction isolation level does this mechanism result in? [5 marks]

   (c) Consider a database server that only deals with transactions of the following kind: there are three tables $X$, $Y$, and $Z$ in the database and any two tables can act as an index to the third. Transactions will read two of these tables, say $S_1$ and $S_2$, to identify a record of the third one, say, $T$ that they will then modify; $(S_1, S_2, T)$ can be any permutation of $(X, Y, Z)$ and the system has multiple such transactions running concurrently (*i.e.*, transactions are not run serially, though the resulting schedule should be serialisable). The read tables, $S_1$ and $S_2$, are read in a random fashion and not necessarily completely. The record that will be modified in $T$ can be any record and the transaction will identify it after it is done accessing $S_1$ and $S_2$.

      i. Devise an efficient locking protocol for this type of system. Explain why your protocol is efficient. [8 marks]

      ii. Assume that the requirements change: both $S_1$ and $S_2$ are completely and sequentially scanned, while more than one record of $T$ will be modified, and at arbitrary points in time. Should your protocol change? If it should change, give the new protocol and explain why it is better than the original one. If it should not, explain why it still works. [7 marks]

(a)
The phantom problem is a situation where a transaction retrieving a collection of objects but get different result values, even though it does not modify any of these objects itself and follow the strict 2PL protocol. This problem arises  where a transaction cannot assume it has locked all objects of a given type.

for example:
All sailors with rating 1, new sailors of rating 1 can be added by a second transaction after one transaction has locked all of the original ones.

(b)
Index locking

Predicate locking

The mechanism databases use to deal with the phantom issue is predicate locking. Index locking is a special case of predicate locking for which an index suppotrs efficient implementation of the predicate locking.
Index locking produces serialisable schedules (transaction isolation level = serialisable) even though it doesn't conform with  2PL.

(c)

i) I believe that multiple granularity locking is the answer here and a good combination of IS, IX, SIX, S and X locks. I would require IS locks for the S tables since they are not necessarily read completely and upgrade to an S lock for the tuples I want to read. This way I achieve some interleaving even in cases that other transaction regard my S tables as T tables and want to update them.
After finished reading S, and without releasing any lock (to ensure strict 2PL), for my T table I would require IX locks all the way down to the tuple I want to modify where I would upgrade to an X lock.
The use of IS and IX locks in the top level of my hierarchy allow interleaving of transactions and therefore increase efficiency

ii) The fact that S tables need to be scanned on their entirety leads us to drop the IS locks solution and use S locks even for the highest level of our hierarchy. Our policy regarding T tables doesn't need to change (I think) since we can have IX locks and upgrade to X whenever we want to perform an update.
In terms of efficiency the new policy is worse, cause it uses S locks on the S tables, which doesn't allow other transactions to require IX locks in these tables (since a transaction S table migh be the T table of another transaction). The reverse problem also exists, since a transaction holding an IX lock on a T table will not allow another transaction to require an S lock on that table..