Problem 1

PostgreSQL and MySQL have different process models. PostgreSQL uses an OS process per session, while MySQL uses an OS thread per session.

Explain pros and cons of the two models.

Answer:

Process model

Pros:
 - OS provides memory address space isolation between sessions
 - One session can crash without necessarily taking down the rest of the DBMS

Cons:
 - More expensive context switching
 - Sharing/coordination between sessions is harder

Thread model:

Pros:
 - Cheaper context switching
 - Easy sharing beween sessions (same memory address space)

Cons:
 - Less isolation between sessions (one thread may bring down the whole DMBS)

Problem 2

In both the process per session and thread per session models it is possible ot use pooling (process pooling or thread pooling). What are the benefits of using pooling?

Answer:

 - It saves the expensive thread/process creation overhead by reusing threads/processes
 - Easy queueing/limiting of number of simultaneous sessions

```
Problem 3
=========
```

A DBMS with the same capabilities of System R receives this query:

```
SELECT *
FROM t1, t2, t3
WHERE t1.a = t2.a AND t2.b = t3.b
ORDER BY t1.c + 2 * t2.c + 0.5 * t3.c
LIMIT 10;
```

List all interesting orders and specify for each order why it is interesting.

Answer:

t1.a: Join condition
t2.a: Join condition
t2.b: Join condition
t3.b: Join condition

These can be used for a merge join.

The ordering expressoin is too complex to provide an interesting order in this system.

```
Problem 4
=========
```

The system in Problem 3 is extended with support for the rank join operation. Which new orders in the query have become interesting? Explain why.

Answer:

The new interesting orders are the entire ORDER BY expression (t1.c + 2 * t2.c + 0.5 * t3.c), and each column mentioned in the expression: t1.c, t2.c and t3.c.

The column orderings can be used by a rank-join operator to produce the result in the order of the ORDER BY clause, saving a sorting operation. Also, becauese of the LIMIT, the rank-join can stop before producing all rows of the join result (unless the data set is such that it can't be done). A normalt join + sort would have to produce all rows of the join result and then do a non-pipelining sort operation, even if the LIMIT is much less than the size of the complete result.

Problem 5

Explain the problem of empty join results with traditional sampling based cardinality estimation. What is it and why does it happen?

Answer:

The problem of empty join results is that joining samples may produce an empty join result, even if the join of the complete tables will produce a non-empty result.

It occurs because the random table samples contain only a few rows, and hence also fewer matches. After a number of joins, the join result may become very small or disappear completely just because the sampling algorithm was "unlucky" about it's selection of rows.

The classical solution is to create larger samples, which has a negative effect on performance.

Problem 6

How does index-based join sampling address the problem of empty join results?

Answer:

Index based join sampling, using complete indexes instead of samples, joins against the complete data set and therefore has a higher probability of producing a larger join result from which it can generate a sample of the join result. However, it may still be "unlucky" and pick rows with values that produce few or none join results.

Problem 7

What is pipelining in the context of query execution? Give an example of a non-pipelining operation and explain why it isn't pipelining.

Answer:

Assume a Volcano-like iterator model. Pipelining is that rows in general pass through the iterators in a way that doesn't reuire building large buffers of rows. It doesn't have to be that one output row is produced for each input row (e.g., join does not do that), but in general there is a constant flow of result rows as the inputs are being read.

Sorting is the prototype of a non-pipelining operator. In order to produce the first row of the sorted result, the sort operator needs to read and buffer all input rows, since the first output row may be the last row to be read.

Problem 8

How does interesting order tracking affect the use of pipelining operations?

Answer:

Interesting order tracking provides a way for the optimizer to understand which plans produce rows in which order. This allows it to understand when it can read the rows in the sorting order and skip the non-pipelinging sort operation.

Non-pipelining sorting may still be the best plan in some cases, but with interesting order tracking the optimizer is able to make a cost based decision about it.

Problem 9

Explain how delete node entries in LSM-trees work and how they affect lookups.

Answer:

When an row is deleted and no entry for that row is found in C_0, a delete node entry is inserted into C_0 with that key value. The delete node entry signals that this value has been deleted. During the rolling merge process, this entry is read and propagated to C_1, C_2, etc., and matching values are deleted during the merging.

While this is ongoing (i.e., while there are active delte node entries in the system), lookups must be filtered through the delete node entries so that deleted rows are not returned. Since delete node entries are located in the same place in the index where the rows with that key value would have been, they are encountered during a lookup in a component (C_x) earlier than where rows with this key value are stored (C_(x+n)).

Problem 10

What is the purpose of join indexes in C-Store? Why are join indexes not necessary in a row store?

Answer:

Join indexes are indexes that contain the mapping between two different orderings of data from the same table. Since ordering is the same for all rows in a projection, join indexes are mappings between different projections. They are used to reconstruct all the records of a table from its projections.

In row based storage, the full records are always stored together, so it's never necessary to reconstruct them from partial records.

Problem 11

Imagine a distributed DBMS specialized for storing calendar events (meetings, appointments, etc.). A single user can have several phones, tablets and computers. All devices have a complete copy of the user's calendar and are considered nodes in the same DBMS. In addition, there is a node running in a cloud service that also has a full copy. If a device is offline, the user can still read and update their calendar events on that device. The data is then synchronized with all other online nodes when the devices comes online and connects to the cloud service, which acts as the coordinator for synchronization.

Where is this system located along the dimensions of transaction location (where) and synchronization point (when)? Explain why.

Answer:

Update anywhere: Any node can handle updates.

Lazy: Synchronization is done after commit. Single nodes can commit transactions without even if disconnected from all other nodes.

Problem 12

Give an example of how an inconsistency may occur when devices in the system described in Problem 11 are doing operations while disconnected.

Answer:

Example:
There is an event for a physical in-person written re-sit exam from 09:00 to 13:00 on a day in August. This event is distributed to all nodes. Nodes/devices A and B are then disconnected and operating offline. While disconnected, device A modifies the event to change from a physical exam to a at home digital written exam. At the same time, while disconnected, device B modifies the event to change it to an at home oral exam from 11:00 to 11:30.

When devices A and B reconnect to the other nodes, both these updates must be distributed to the other nodes. But they are conflicting, and a conflict resolution mechanism is required to handle it and make sure the system as a whole reaches the same state for this event.

Problem 13
==========

Which of the three properties of the CAP theorem does the system in Problem 11 have? Explain why.

Answer:

Available: Reads and writes are always available, also when disconnected.

Partition tolerance: The system handles partitioning down to single node partitions.

It is not consistent: Inconsistencies may occur, as demonstrated.

Problem 14
==========

A DBMS receives this query for hotels to display on a map on a cell phone screen. The area of the map that is visible on the screen is the box called @map_viewport.

```
SELECT *
FROM hotel JOIN country ON hotel.country_id = country.id
WHERE
  country.population > 1000000
  AND
  ST_Within(hotel.position, @map_viewport)
  AND
  hotel.star_rating = 4
ORDER BY hotel.name;
```

The boolean expression ST_Within(a, b) returns true if a is within b.

Assume that there are indexes on all columns mentioned in the query and that the indexes support window queries, point queries, distance scan, equality lookups, index range scans and full index scan.

List all sargable predicates in the query and what type of index access would be used for that predicate.

Answer:

Note: The question asks for predicates, not just column names.

country.population > 1000000: index range scan on the index over country.population for the range (1000000, end of index]

ST_Within(hotel.position, @map_viewport): Window query on the hotel.position index for the window @map_viewport.

hotel.star_rating = 4: Index equality lookup for 4 on the index on hotel.position.

hotel.country_id = country.id: Index equality lookup on either the index on hotel.country_id or th eindex on country.id, depending on which is the inner table of the join. Both options will be evaluated.

The hotel.name column in ORDER BY is _not_ a sargable predicate. It's not even a predicate. It is an interesting order.