Problem 1:

Our implementation applies to a Last-Write-Wins approach to dealing with conflicts. Assume we have sites *a* and *b*. Now, make both sites concurrently execute a *put* operation on the same key: *test*, where *a* executes

put test a

and *b* executes

put test b

After the execution, the clock matrix, key-value pair and log at *a* become

$$\begin{array}{cc} 1 & 0 \\ 0 & 0 \end{array}$$

{'test': ['a']}

{operation: 'insert', key: 'test', value: ['a'], timestamp: 1, site_id: a}.

The clock matrix, key-value pair and log at *b* become

$$\begin{array}{cc} 0 & 0 \\ 0 & 1 \end{array}$$

{'test': ['b']}

{operation: 'insert', key: 'test', value: ['b'], timestamp: 1, site_id: b}.

Now, let *a* and *b* execute

a: *send b*

b: *send a*

where *a* sends a message to *b* containing its id, T_a and e_a, and *b* sends a message to *a* containing its id, T_b and e_b. Upon receiving this event, *a* updates its kv to

{'test': ['b']}

However, *b* doesn't update its kv. Even though the counters of two insertion events are identical, the site id *b* is thought larger than *a*. That means

$$e_a \rightarrow e_b$$

As a result, insertion of 'b' is thought as the last one. Hence, it is still

{'test': ['b']}

This idea is based on the totally ordered Lamport clock, so the casual consistency is ensured.

Problem 2:

Regarding dictionary update, our implementation performs the following operations. Firstly, it unpacks the messages and updates the matrix clock. Since parsing uses Python dict access of O(1) time, this step totally takes $O(n^2)$ time where **n represents the number of sites in the system**. For each event, dict provides O(1) operation for comparison and update, so the total time complexity for processing |NP| events is O(|NP|). In summary, the dictionary update takes $\mathbf{O(n^2 + |NP|)}$ time.

Regarding log truncation, our implementation checks whether each event is known by all sites. This means for each event, it takes O(n) time for checking. Since there are $|PL_i|$ local events, the truncation step takes $\mathbf{O(n \times |PL_i|)}$ time in total.

The following is the pseudocode for dictionary update:

```
function handle_incoming_message(message, key_value_store, log, matrix_clock, site_indices, site_id):
    # Parse the received message
    msg = parse_json(message)
    sender_site_id = msg['site_id']
    sender_matrix_clock = msg['matrix_clock']
    events = msg['events']

    # Update local matrix clock
    update_matrix_clock(matrix_clock, sender_matrix_clock, site_indices)

    # Process each event
    for event in events:
        event_site_id = event['site_id']
        event_timestamp = event['timestamp']
        idx_i = site_indices[event_site_id]
        local_idx = site_indices[site_id]

        # Check if the event is new
        if event_timestamp > matrix_clock[idx_i][local_idx]:
            # Apply the event to the key-value store
            apply_event(event['operation'], event['key'], event['value'], event_timestamp, event_site_id, key_value_store)
            # Add event to the log
            log.append(event)
            # Update matrix clock to reflect this event has been processed
            matrix_clock[idx_i][local_idx] = event_timestamp
```

The following is the pseudocode for log truncation.

```
function truncate_log(log, matrix_clock, site_indices, site_ids):
    events_to_keep = []
    for event in log:
        event_site_id = event['site_id']
        event_timestamp = event['timestamp']
        idx_i = site_indices[event_site_id]

        # Check if all sites have acknowledged the event
        acknowledged_by_all = True
        for site in site_ids:
            idx_j = site_indices[site]
            if matrix_clock[idx_i][idx_j] < event_timestamp:
                acknowledged_by_all = False
                break
        if not acknowledged_by_all:
            events_to_keep.append(event)
    # Update the log if necessary
    if len(events_to_keep) != len(log):
        log.clear()
        log.extend(events_to_keep)
        return True    # Log was truncated
    return False    # No changes made
```

Problem 3:

NO. In our implementation, each site maintains its own local log and matrix clock to track dependencies. This ensures casual consistency among all sites. Before sending a message, a site only includes events and values that the receipt "doesn't know" based on its knowledge. Upon any receiving event, the log and clock make sure that all causally preceding operations have been applied before applying a new one, and the clock is correspondingly updated. Therefore, a global order is not required.

Problem 4:

In terms of compute:

Pros: Strategy 2 only considers local and neighbor information, potentially speeding up the events processing step. Also, it allows node to discard events more promptly, which can reduce the computational workload of managing large logs.

Cons: Additional computational logic must be implemented for the gateway nodes, since they need to act as representatives for events occurring outside their immediate area. This also applies to other nodes, because the computational steps required for each event can be increased to implement neighbor-specific storage and determine when to discard events based on neighbor acknowledgments.

In terms of storage:

Pros: Our implementation stores a truncated log and a matrix clock of O(n^2) space complexity, while strategy 2 only requires a subset of 2DTT and a partial log that discards events once all neighbors are informed. Therefore, strategy 2 requires less disk and memory space.

Cons: Strategy 2 may require additional storage for the gateway nodes. Besides, the failure of a neighbor may lead to redundant log contents.

In terms of network:

Pros: Strategy 2 requires less network bandwidth because it only sends the relevant rows to target neighbors. The packet needed to be sent is smaller. Moreover, the communication is localized, meaning that messages are passed among neighbors and the overall performance is improved.

Cons: Communication across boundaries may introduce additional hops and workloads on the gateway nodes. Besides, it is possible for a node to send more messages to cover all necessary neighbors, which could potentially increase the total number of packets on the network.

In terms of different types of workloads:

If most workloads are concentrated within specific neighborhoods or areas, strategy 2 has an efficient storage utilization and lower communication overhead due to its localized nature. Event processing and log truncation will be faster, improving the throughput and delay.

If most workloads involve interactions across different areas or require global consistency, strategy 2 may be less efficient because the gateway nodes can become bottlenecks.

If workloads are within a balance of localized and global operations, the performance of strategy 2 depends on the ratio of two kinds of workloads. There is a tradeoff between the benefits from reduced storage and localized communication and the overhead of managing cross-area events.