Part D – Scalability Challenge

Scenario

Our boss wants the *Campus Event Scheduling System* to scale up to 1 million events. We therefore analyse the memory usage for array-list and linked-list, suggest any structural optimizations, and sketch a design for parallel conflict detection using multiple cores.

1. Memory Usage Estimation

Each *Event* object stores an ID, title, date, time, and location. Let's assume:

Each object roughly stores:

- an ID (i.e. int type): 4 bytes
- title, date, time, location: ~100 bytes total (i.e. strings + references)
- Object overhead in Python: ~60 bytes (it is the extra memory that Python uses to store information about our data - not the data itself, ex: type of data etc.)

Therefore, that is approx. **170 bytes per Event** (which is a rough estimate).

Array-List (Python List):

- Python lists store pointers to objects and not the objects themselves.
- Each pointer (or) reference takes ~ 8 bytes.
- So, **total memory estimation** = 1,000,000 events * (170 bytes per Event + 8 bytes per reference) ~ **178 MB** total (approx.)

Linked List:

Each node stores:

- Reference to Event (~ 8 bytes)
- Reference to the next node (~ 8 bytes)
- Python object overhead (~ 60 bytes) i.e. ~ 76 bytes per node.
- So, **total memory estimation** = 1,000,000 events * (170 bytes per Event + 76 bytes per node) ~ **246 MB** total (approx.)

Structure	Memory Estimation	Approx. Memory for 1 M Events
Array-List OR Python list	170 bytes + 8 bytes per reference	~ 178 MB
LinkedList	170 bytes + 76 bytes per node	~ 246 MB

Extra Memory Requirement = (246 - 178) / 178 x 100 = 38.2%

Thus, the linked-list implementation requires ~38% more memory than the array-list for 1M events.

Conclusion:

- Array List uses less memory (~ 178 MB).
- Linked List uses more memory (~ 246 MB). That is due to extra node objects and pointers.
- Also, Array List has better cache locality and faster sequential access.

2. Suggested Optimizations for Large-Scale Data

i) Indexing by Date or Event ID:

We can maintain an *Index Dictionary* for quick lookup:

index = {event.id: event}

This avoids using the linear or binary search and also reduces the search time to *O(1)*.

ii) Hybrid Structure:

We can use an *Array-List (or Python List)* for the main storage of events (good for sorting of events) and a *HashMap* (or Dictionary) for direct access or lookups of the events.

- Array List: It keeps a chronological order and stores the items in order - for ex., sorted by date or time. It is great for sorting the elements because they are stored contiguously in memory, making the sorting algorithms faster. But searching for a specific event (ex - event with event_ID = 12345) would take *O(n)* time if we go one by one.
- HashMap: It helps with a quick ID lookup. In Python, this is basically a *Dictionary* which stores data as key-value pairs - for ex., *{event_id: event_object}*. We can instantly access any event if we know its ID in *O(1)* average time. But HashMaps are unordered, so we cannot easily sort or iterate it in order.

Putting them together (i.e. the hybrid structure idea), we can use both the structures at the same time:

array_list = [] # keeps events sorted by date/time

event_map = {} # quick lookup by the event ID

When we add a new event:

- We append it to the array_list (and later sort it if needed)
- And then we add it to the event_map with its ID as the key

This way:

- We can sort and display the events easily (using the Array)
- And search or update the events instantly (using the HashMap)

In short:

- Array-List = keeps the things *in order* (good for sorting).
- HashMap = gives *instant access* by using the key (good for searching).
- Hybrid = best of both the worlds – i.e. fast searches AND easy sorting.

iii) Grouped Conflict Checks:

Normally, if we want to find which events overlap in time, we would have to check *every event against every other event.* This means if we have 1,000 events, we end up performing 1,000 x 1,000 = 1,000,000 checks. That is what leads to $O(n^2)$ time – which is very slow when the number of events gets bigger and bigger.

Therefore, instead of checking all events with all other events, we can group them first - for example:

- Put all events that happen on the **same date** in one group
- or even better, in the **same hour block** (like 9–10 AM, 10–11 AM, etc.), in one group

Then, we only check for conflicts within each of those small groups, since events on different days or time blocks cannot possibly overlap with each other.

Let's consider we have these 6 events:

Event	Date	Time
A	Oct 15	10-11 AM
B	Oct 15	10:30-11:30 AM
C	Oct 15	2-3 PM
D	Oct 16	9-10 AM
E	Oct 16	2-3 PM
F	Oct 17	10-11 AM

If we check all pairs, 6x6 = 36 checks.
But if we **group by date** first:

- Oct 15 group: A, B, C
- Oct 16 group: D, E
- Oct 17 group: F

Now we only check inside each group:

- Oct 15: 3x3 = 9 checks
- Oct 16: 2x2 = 4 checks
- Oct 17: 1x1 = 1 check

Total = 9+4+1 = 14 checks instead of 36. That is much faster!

iv) Use of External Storage:

For 1M+ events, we can either use a database or a file-backed structure (for ex., maybe a .csv, .json etc.)

- Example: SQLite or Pandas Data Frame.
- And load only relevant events into memory when needed.

3. Parallel Conflict Detection (Multi-Core Idea)

Conflict detection means to check if any events overlap in time. Instead of checking all the pairs of events (which is slow), we can parallelize the process of checking:

The main idea for this: Divide the Events by Date

1. Split events by date into smaller chunks. For ex., events from Jan-Mar go to Core 1, Apr-June go to Core 2, etc.
2. Each core runs *conflict detection* independently within its subset.
3. Then we combine the results at the end.