Property Listing Platform (System Design)

Data Structure Design

Property Listings Data

Structure:

```
property_listings = {
    "property_id_1": {
        "user_id": "user_1",
        "details": {
            "location": "New York",
            "price": 500000,
            "type": "Apartment",
            "status": "available",
            "timestamp": "2025-01-01T12:00:00Z"
        }
    },
    "property_id_2": { ... }
}
```

- **Key:** `property_id` (unique identifier for each property)
- **Value:** Dictionary containing user ownership, property details, and status.

Justification:

- Fast O(1) lookup for property details.
- Easily scalable and extensible for additional attributes.

User Portfolios

Structure:

```
user_portfolios = {
    "user_id_1": ["property_id_1", "property_id_3"],
    "user_id_2": ["property_id_2"]
}
```

- **Key:** `user_id`
- **Value:** List of property IDs owned by the user.

Justification:

- Efficient mapping of users to their properties.
- Supports fast retrieval of all properties associated with a user.

Shortlisted Properties

Structure:

```
shortlisted_properties = {
    "user_id_1": {"property_id_2", "property_id_4"},
    "user_id_2": {"property_id_5"}
}
```

- **Key:** `user_id`

- **Value:** Set of property IDs shortlisted by the user.

Justification:

- Set ensures no duplicate shortlists for a user.
- Efficient for adding, removing, and checking if a property is shortlisted.

Search Indices

Structure:

- **Location Index:**

```
location_index = {
    "New York": {"property_id_1", "property_id_3"},
    "Los Angeles": {"property_id_2"}
}
```

- **Price Index:**

```
price_index = {
    (0, 100000): {"property_id_5"},
    (100001, 500000): {"property_id_1", "property_id_3"}
}
```

Justification:

- Location index allows O(1) lookup for properties by location.
- Price index with predefined ranges enables efficient filtering by price.
- Supports intersection and union of results for multiple criteria.

Property Status Updates

Approach:

- Locate the property in `property_listings` using `property_id`.
- Update the `status` field (e.g., "available" -> "sold").
- Reflect changes in relevant search indices (e.g., remove from `location_index` if no longer relevant).

Example:

```
property_listings["property_id_1"]["status"] = "sold"
location_index["New York"].remove("property_id_1")
```

Search/Sort Implementation Strategy

Price Range Filtering

Approach:

1. Identify all price ranges overlapping the query range.
2. Retrieve property IDs from matching ranges in `price_index`.
3. Return matching properties by intersecting with other criteria if applicable.

Example:

- Query: `min_price=100000, max_price=500000`

- Combine ranges `(100001, 200000)` and `(200001, 500000)`.

Code:

```
matching_ids = set()
for price_range, properties in price_index.items():
    if min_price <= price_range[1] and max_price >= price_range[0]:
        matching_ids.update(properties)
```

Location-Based Search

Approach:

1. Use `location_index` for O(1) retrieval of property IDs for a specific location.
2. Intersect results with other filters if provided.

Example:

- Query: `location="New York"`
- Retrieve: `{"property_id_1", "property_id_3"}`.

Code:

```
location_results = location_index.get("New York", set())
```

Multiple Criteria Sorting

Approach:

- Use Python's `sorted()` function with a custom key.
- Example sorting criteria: `price`, `timestamp`.

Code:

```
def sort_criteria(property):
    return (property.details["price"], property.details["timestamp"])

sorted_results = sorted(properties, key=sort_criteria)
```

Search Result Pagination

Approach:

1. Calculate start and end indices based on `page` and `limit`.
2. Slice the results accordingly.

Code:

```
start = (page - 1) * limit
end = start + limit
paginated_results = sorted_results[start:end]
```

Performance Considerations

- **Indexing:**
  - Precompute and store indices for frequent search fields (location, price).

  - Use in-memory data structures (e.g., dictionaries) for quick lookups.
- **Scalability:**
  - Partition indices for large datasets (e.g., by region or price range).
  - Use caching mechanisms (e.g., Redis) for frequently accessed queries.
- **Concurrency:**
  - Implement locking mechanisms or use atomic operations for concurrent updates to shared data.

Indexing Strategy

- Build indices for fields queried frequently (e.g., `location`, `price`).
- Store index entries as sets of property IDs for fast intersection operations.
- Periodically rebuild indices to handle updates and maintain consistency.

Example Rebuild Logic:

```
def rebuild_location_index():
    location_index.clear()
    for property_id, details in property_listings.items():
        location = details["location"]
        location_index.setdefault(location, set()).add(property_id)
```