Data acquisition, extraction, and storage

Pierre Senellart

12 December 2025

This exam lasts two hours and a half. The only documents allowed are ten A4 sheets (both sides), with the content of your choice. Communicating devices are strictly forbidden. When writing code, imprecision in the syntax of the languages will be tolerated. The exam is formed of a single problem, graded out of 20 points.

Context

You work on building a unified system to collect, process, and query environmental sensor network data. Sensors expose JSON APIs; geospatial land parcels come in a JSON format called GeoJSON. Your system must acquire, normalize, store, process, and serve this heterogeneous data.

1. Incremental Crawler (3 points)

Each station exposes all measurements taken after a given point of time with:

`GET https://<station>/api/v2/measurements?after=<ISO8601-timestamp>`

(ISO 8601 is the standard for date and time representation, using formats such as `2025-03-18T14:20:00Z`.)

Write pseudocode for a function `crawl_available(stations, state)` that:

- loads last-seen timestamps from `state` (a mapping for each station to its last timestamp of crawl),
- sends requests to stations from `stations`, ensuring no two requests are sent to the same station within 200 ms,
- add the measurements to the local filesystem,
- updates the stored timestamp for each station.

Your pseudocode must explicitly show control flow and error handling.

2. Data Volume (1 point)

Each station produces one 7 kB JSON record every minute. There are 10 000 stations. Give an order-of-magnitude computation of the total size (in GB) for one full year. Show your calculation.

3. JSON/GeoJSON Selectors (3 points)

Given the GeoJSON FeatureCollection:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "parcel_id": "FR-093-18729",
        "owner": { "name": "Dupont", "type": "private" },
        "area_m2": 1280
      },
```

```
      "geometry": { ... }
    }
  ]
}
```

Provide *three concrete selectors* (in JSONPath, JMESPath, `jq`, or similar) to extract:
(a) all parcel IDs,
(b) all owner names,
(c) all areas between 1000 and 2000 m^2 (use filtering if available).

4. JSON Transformation Pipeline (3 points)

You receive sensor measurements as JSON Lines files (`*.jsonl`), where each line is a JSON document with the structure:

```
{
  "station_id": "ST-91",
  "timestamp": "2025-03-18T14:20:00Z",
  "temperature": 17.3,
  "ozone": 82.1,
  "battery": 92
}
```

All files must be converted to a *normalized* JSON Lines format containing only:
- `station_id`,
- `timestamp`,
- a `payload` object storing all remaining fields.

You must also store it in directories of the form:

```
normalized/year=YYYY/month=MM/day=DD/
```

Write Python-like pseudocode that:
(a) iterates through all `*.jsonl` files,
(b) parses and validates each JSON line,
(c) builds normalized records,
(d) writes them to the correct output directory.

5. Relational Schema + SQL Query (2 points)

Measurements are thus normalized into JSON objects of the form:

```
{
  "station_id": "ST-91",
  "timestamp": "2025-03-18T14:20:00Z",
  "payload": {
    "temperature": 17.3,
    "ozone": 82.1,
    "battery": 92
  }
}
```

(a) (1 point) Propose SQL `CREATE TABLE` statements for a relational schema storing:
- stations (with an ID and optional metadata),
- measurements (one tuple per measurement).

Your schema must specify `PRIMARY KEY` and appropriate types.
(b) (1 point) Write a SQL query returning the IDs of all stations whose *average ozone level over the last 24 hours exceeds 100*. You may assume the current time is `NOW()`.

6. Distributed Processing with MapReduce (2 points)

You must compute, for each station and for each calendar date, the *maximum ozone value* from a dataset of more than 400 million measurement records. The dataset is stored in the normalized JSON Lines format presented before.

(a) Write *MapReduce pseudocode* (both Mapper and Reducer). The job should output lines of the form:
`station_id  date  max_ozone`

(b) In 2–3 sentences, explain why MapReduce is appropriate for the task and what its main limitations are compared to Spark.

7. Data Quality Rules (4 points)

Before storing measurements, you must apply a small set of *data-quality validation rules*:

- R1: temperature must be in the range $[-40, 60]$ (but may be missing),
- R2: ozone must be non-negative (but may be missing),
- R3: timestamps must be strictly increasing for each station.

(a) (2 points) Write Python-like pseudocode that reads a stream of normalized measurements and rejects records violating any rule. For R3, you may store per-station state in a dictionary.

(b) (2 points) Suppose that records have already been stored in a database as in Question 5. Which constraints can be imposed in the schema? Show how to do it. When the constraint cannot be imposed in this way, write an SQL query returning all measurements that violate the constraint.

8. Provenance Using Semirings (2 points)

Each measurement is annotated with provenance following the *provenance semiring* model. We use tokens `a`, `b`, `c`, ... for things such as:

- acquisition sources,
- data-quality rules that were applied (R1, R2, R3),
- transformations (e.g., unit conversion).

For a record x, its provenance annotation may look like:

$$p_x = a \otimes b \oplus c.$$

(a) (1 point) Explain briefly (2–3 sentences) what the expression $a \otimes b \oplus c$ means in the provenance semiring.

(b) (1 point) Suppose we query:

```
SELECT DISTINCT station_id FROM measurements WHERE ozone > 80
```

Provide the formula for the provenance of an output tuple in terms of input provenance polynomials.