Problem Description:

Design and Describe a system to process a file/stream of JSON content in a non-serial manner, persist the records to a transactional data store, and log a “Finished” event upon persistence of the final record. The goal is to minimize processing time and actively recognize processing completion, without losing data. Communicate your design using any mediums you deem necessary or effective. (eg. Descriptive English language, Illustrations, Pseudocode, etc.)

1. Assume you have control of the file/stream.
2. Assume that the JSON total size is on the order of multiple gigabytes or larger, with variable record schema and size,
3. Assume that the Transactional persistence mechanism may be intermittently unavailable and also when available has the potential to reject individual records.
4. Individual records can fail persistence. If they do fail, the process should continue and the failure facts should not be lost, in order to attempt remediation.
5. Please include a description of how the remediation process could work.
6. Be sure to describe stack component expectations and feel free to make specific component recommendations that meet your needs. eg. "This component should be an in-memory store. Redis would be a good choice."

```
[
   {
      “id”: 1,
      “awesome_attribute”: “awesome attribute value”,
      “dynamic_attribute”: “is lucky to be here”
   },
   {
      “id”: 2,
      “awesome_attribute”: “stellar attribute value”,
      “tubular_attribute”: {
         “has_tubes”: “Totally”,
         “which_tubes”: [3,5,7,4]
      }
   },
...
]
```

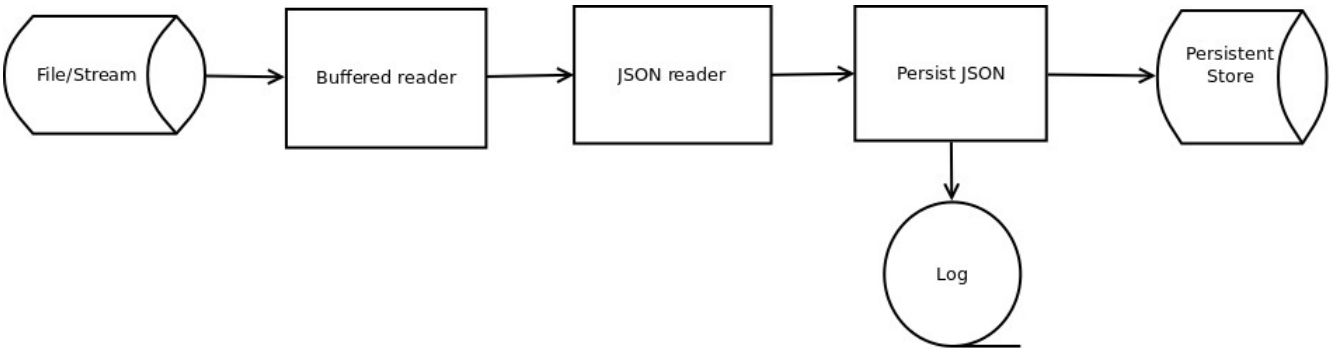
(Record content specifics are given only as examples of structural potential in JSON. Aside from the “id” attribute they have no semantic bearing
on the problem other than that they are variable.)

Assuming that the file/stream is essentially of unbounded size, I will assume that each record within the file is of a manageable size.

The problem breaks down into these parts:

1. Ingestion of the file/stream.
2. Identification of individual records.
3. Validation.
4. Persistence.
5. Event logging (errors, end of run).

The first two parts could easily be done by pipelining the data through file-like objects.



The “buffered reader” would read directly from the data source. The amount of data read (size of the buffer) could be a runtime configurable option. On instantiation of this object, the external file would be opened, and any errors handled (ENOENT, EPERM, etc.)

The “JSON reader” would be instantiated with the “buffered reader” as an input argument. The “JSON reader” would then read from the “buffered reader”, one character at a time, skipping all input until the first opening brace is found. It would then continue to read, saving the input into it's own buffer, until the closing brace is found. Errors here would be logged, and processing continuing with the next record, if possible. In the case of bad input data, it is possible that the closing brace is never found – an “End of File” condition (or ENOMEM if the input file is exceptionally large) would be an indication of bad data. In either case, the event would be logged and processing terminated. (Note: in the case of ENOMEM, only a small portion of the beginning of the record should be logged, enough to identify the bad record.)

At this point, it is assumed that a JSON record has been read, and it is passed on to the “Persist JSON” object. This object would validate the data as well-formed JSON, and the “id” attribute extracted. If validation succeeds and the “id” attribute is found, the record would then be persisted, using the “id” attribute value as the key in the persistence data store. If validation fails, or the “id” attribute is not found, the event would be logged and processing would continue with the next record.

It is possible that the persistence mechanism might not be available. Several strategies could be considered for dealing with this situation:

1. Block until the mechanism is once again available. Probably not a good solution.
2. Log the event, and continue with the next record. Also probably not a good solution.
3. Wait for timeout on the operation. Retry the operation after some configurable delay, and continue to retry for some configurable number of retries. In the case that all retries are exhausted before the operation succeeds, it is probably safe to assume that there is a serious problem. The event should be logged, and the process terminated. This is probably the best solution.

It is also possible that the record is rejected by the persistence mechanism for some reason, such as duplicate record. Rejected records should be logged, and processing continuing with the next record.

When end of file is detected on the file/stream, the number of records successfully persisted and the number of records in error should be reported, the input file/stream closed, the connection to the persistence mechanism closed, and the process terminate normally.

Error remediation can be done using the error log. When a record is rejected or otherwise fails to be persisted, that record should be written to the error log. This log could then be examined for errors, and records fixed in some way (manually, or perhaps some automated process), and the previously failed records resubmitted for processing.

In the case of a timeout causing the process to fail, the record being processed at the time of the failure could be used as a checkpoint record, allowing for the original input file/stream to be used as input again, but skipping all records up to the point of the previous failure, where processing would continue again as normal.