



# Architecture and Approach for GEDCOM 5.5.5 Data Processing

## Overview and Goals

Handling GEDCOM 5.5.5 files involves **parsing genealogical data from a structured text format, validating it, storing it in a database**, and then performing data **standardization, normalization, enrichment, and reconstruction** back to GEDCOM. GEDCOM is a widely adopted but surprisingly complex flat-file format – it looks simple, yet its tags can combine in many ways to cover myriad genealogy needs 1. The primary goals of the project are:

- **Parse** one or more GEDCOM files into an internal data structure without loss of information, enforcing GEDCOM 5.5.5 rules (e.g. required fields, character encoding).
- **Validate** the file structure and cross-references (ensuring the database representation exactly matches the original data content).
- **Store** the data in a suitable database (relational **SQL** or a **graph database**) for persistence and potential querying, while preserving the genealogical relationships.
- **Process** the data to standardize and normalize key fields (e.g. personal names, dates, places, and their sub-components) using known standards and reference datasets.
- **Enrich** the data by incorporating external information (such as standardized place names, coordinates, or additional attributes) from provided JSON/reference datasets.
- **Reconstruct** and export the cleansed data back into a valid GEDCOM 5.5.5 **.ged** file, preserving all original content (plus enhancements) so that the output GEDCOM is semantically equivalent to the input (or improved) and passes validation.

In essence, the system needs to **round-trip** the data: GEDCOM file → internal representation/DB → processed data → GEDCOM file, with fidelity and improvements. Below, I outline a comprehensive architecture and step-by-step approach, including design decisions for parsing, database modeling, data cleaning, and output generation.

## Key Challenges and Considerations

- **Strict Format Compliance:** GEDCOM 5.5.5 demands strict adherence to the spec for both readers and writers. Files must be Unicode (UTF-8 or UTF-16) and start with a Byte Order Mark (BOM) 2. The parser must be robust against format errors and enforce required structures. For example, GEDCOM 5.5.5 no longer tolerates certain sloppy practices tolerated in 5.5.1; a 5.5.5 reader **must not** accept invalid constructs and should error out on serious format violations 3 4. This means validation is critical at import time.
- **Hierarchical Data with Cross-References:** GEDCOM data is hierarchical (multi-level records) with cross-reference pointers linking records (individuals to families, sources, etc.). The parser must capture this structure. The format of each line is **Level [ @XREF\_ID@ ] TAG [ Value ]** (plus possibly a pointer) 5. For example, **0 @I1@ INDI** denotes an Individual record with ID I1, and

lines below it (level 1, 2, etc.) are that individual's details. Other records refer to `@I1@` as a pointer. We must ensure every pointer (e.g. a family link like `1 FAMS @F1@`) references a real record in the data <sup>6</sup>. Preserving and validating these relationships in the database is crucial so that the re-exported GEDCOM matches the original links exactly.

- **Complexity of Data Fields:** Certain fields (NAME, DATE, PLAC, etc.) have their own internal structure and variations. GEDCOM 5.5.5 requires that personal name pieces (GIVN, SURN, etc.) be explicitly present for each name <sup>7</sup>, and that they are consistent with the full name value <sup>4</sup>. Dates may include prefixes like ABT/BEF (about/before), date ranges, alternate calendars, etc., and places can include multiple jurisdiction levels and optional phonetic/romanized variants and map coordinates <sup>8</sup>. The design must account for standardizing these sub-structures.
- **Multiple Files and Merging:** Eventually, multiple GEDCOM files might be ingested. The system should be designed to handle one file at a time initially (for simplicity), but with an eye toward scaling up. This could include merging data or comparing across files in the database (which influences whether a relational or graph model is more suitable for relationships).
- **Choice of Storage:** We need to decide between a relational model (e.g. PostgreSQL) or a graph model (e.g. Neo4j), or even a hybrid approach (storing JSON in SQL). A relational database can enforce schema (helpful for data integrity) and allow SQL queries for analysis. A graph database naturally represents family trees (nodes for people and relationships for family links), aligning with the genealogical structure (family trees are indeed often better handled as graphs than text or tables <sup>9</sup>). We must design a schema or graph model that can store all required data (individuals, families, their attributes, and links) and still allow reconstruction of the original hierarchical GEDCOM.
- **Data Fidelity vs. Usability:** We want the database representation to be as **faithful** as possible to the GEDCOM (so nothing is lost), but also convenient for processing. Pure fidelity might mean storing every GEDCOM line and its hierarchy (even unknown/custom tags) exactly, which suggests a **JSON or line-by-line storage**. But for processing and standardization, we may prefer breaking the data into meaningful entities (Person, Family, etc.). The solution likely will involve **parsing into structured objects** and optionally storing those in the DB (possibly as normalized tables or JSON). We must ensure that any transformation is reversible or at least does not drop content so the final export can include everything (including seldom-used fields or user-defined tags).

With these in mind, I will describe how I would approach each phase of the project: parsing, database design, processing (standardization/enrichment), and export. I'll also discuss the choice of technology (Python and libraries) and outline the first steps to implement.

## High-Level Architecture Pipeline

1. Parse GEDCOM file → 2. Validate & build internal data structure → 3. Insert into database → 4. Retrieve/convert to processing format → 5. Standardize/normalize/enrich data → 6. Save processed data (DB or memory) → 7. Reconstruct and export GEDCOM.

To summarize:

- **Input:** GEDCOM 5.5.5 text file (UTF-8/16 with BOM, containing genealogical records).
- **Output:** Cleaned and enriched GEDCOM 5.5.5 text file (preserving all original info, plus any added standardizations).
- **Intermediate:** An internal representation (e.g. Python objects, dictionaries, or JSON) of the GEDCOM data and a persistent store in a database for multi-file management or complex queries.

Below, each stage is elaborated.

## Phase 1: Parsing GEDCOM Files into an Internal Structure

**Approach:** I would write a **GEDCOM parser** (likely in Python) to read the file line-by-line and build an in-memory representation of the hierarchical structure. Python is a good choice here due to its text-processing strength and available libraries; indeed, a library like `python-gedcom` could parse and give structured elements (Individual, Family, etc.) <sup>10</sup>, but writing a custom parser ensures we deeply understand and control the process (which is valuable for a project like this). As one genealogy developer recommended, it can be beneficial to build it from scratch as a learning experience, adding features incrementally <sup>11</sup>.

**File reading and encoding:** First, the parser will open the `.ged` file in binary mode to detect the BOM and encoding. GEDCOM 5.5.5 files **must** have a BOM and use UTF-8 or UTF-16 encoding <sup>2</sup>. The code should verify this (e.g. if no BOM found, or an unsupported encoding is detected, raise an error as "not a valid GEDCOM 5.5.5 file"). After determining encoding, re-open or decode the file accordingly.

**Line grammar:** According to the spec (and common practice), each line of a GEDCOM file has the form:

```
<level> <optional @XREF_ID@> <tag> <optional value>
```

Optionally, if the value is a pointer to another record, it appears in the form `@XREF@` as the value. For example: - `0 @I123@ INDI` – level 0 (top-level) Individual record with cross-reference ID I123, no direct value. - `1 NAME John /Doe/` – level 1 subrecord (of the individual) with tag NAME and value "John /Doe/". - `2 DATE 12 OCT 1890` – level 2 subrecord (child of an event, perhaps) with tag DATE and a value. - `1 FAMS @F45@` – level 1 subrecord indicating a Family-spouse link pointing to family record `@F45@` (pointer value).

I will parse each line by splitting on the first space to get the level, then checking if the second token starts with `@` and ends with `@` to detect an XREF ID vs. a normal tag. A simple regex could be used to capture these parts. This yields a structured "GEDCOM line" object, e.g. `{level: 1, xref: None, tag: "NAME", value: "John /Doe/"}`.

**Building the hierarchy (tree structure):** Once I have a sequence of parsed line objects, I will nest them according to their level numbers. The algorithm is essentially a **depth-controlled stack**:

- Initialize an empty list `records` to hold top-level records, and an empty stack `current_stack` to track the hierarchy of records as we parse.
- Iterate through each parsed line in order:
  - If `level == 0`: this is a new top-level record. Create a new record object (e.g. a dictionary or custom class) with its tag (e.g. "INDI") and XREF ID (if present), and no parent. Append it to `records`. Reset the stack and push this record as the current top.
  - If `level > 0`: this line is a subrecord. Compare its level to the length of the stack (which represents current depth):
    - If its level is exactly one more than the top of stack's level, then this line is a direct child of the last line. Create a subrecord object and attach it as a child of the record on top of the stack.

- If its level is deeper by more than one (e.g. jump from level 1 to level 3 without a level 2 in between), that violates GEDCOM grammar (we should raise a validation error). Normally GEDCOM levels increment by 1 at most.
- If its level is equal or higher than one of the stack entries, we pop from the stack until the stack’s length equals the current line’s level (this finds the correct parent). Then attach the subrecord to that parent.
- Push the new subrecord onto the stack.

Using this approach, *each record will recursively contain a list of its sub-records*. By the end of the file, we have a tree (or forest) where the top level entries are HEAD, INDIS, FAMs, etc., each containing all their nested data. This methodology is well-known: one implementation describes reading all lines into objects, then structuring them by level into nested chunks [5](#) [12](#). The top-level chunks will correspond to individual and family records, and their children are the nested details. I would likely create data classes for major entity types (Individual, Family, Source, etc.), but initially a nested dictionary or JSON-like structure is fine for flexibility.

**Validation during parsing:** As we construct the structure, I would check for certain structural validity: - The first record *must* be a `HEAD` record and last record a `TRLR` (trailer) record, per GEDCOM format. If not, the file is invalid. - Required substructures in HEAD: e.g. HEAD must contain GEDC version info, a CHAR (character encoding) line, etc. We can verify `HEAD.GEDC.VERS == "5.5.5"` and `HEAD.GEDC.FORM.VERS` also present (5.5.5 introduced `HEAD.GEDC.FORM.VERS` to specify lineage form version) [13](#). If any mandatory HEAD field is missing, flag it. - Check that each cross-reference ID is unique and conforms to the GEDCOM ID syntax (e.g. starts with `@` and ends with `@`, and typically a letter indicating type like I for individual, F for family, etc., though not strictly required by the grammar except that certain records often use specific prefixes by convention). - When a pointer is encountered (like `FAMS @F45@`), record that as a link: e.g. in the Individual record structure, we might store something like `families_as_spouse = ["F45"]`. We will later ensure that an actual Family record with `@F45@` exists. If not, that’s a broken reference. - Enforce GEDCOM 5.5.5 rules for required fields: e.g. each Individual record **must** have at least one NAME subrecord (the spec’s grammar indicates NAME is {1:1} in an Individual) [14](#). We should also enforce the 5.5.5 requirement that each NAME record includes the full set of personal name pieces if applicable (NPFX, GIVN, SURN, etc.) [7](#). If, for instance, an Individual has a NAME "John /Doe/" but no separate GIVN or SURN subtags, that violates 5.5.5 (it might be an older 5.5.1-style file). We could decide to tolerate it (with a warning) and later fix it during standardization, but at least log it as an inconsistency. According to the spec, a 5.5.5 reader "*must check that each PERSONAL\_NAME\_PIECES is self-consistent, and must report a fatal error and abort import when it is not*" [4](#). In practice, we might handle it more gracefully by flagging and attempting to normalize it. - The parser should also accumulate any unknown or unsupported tags. GEDCOM allows custom tags (beginning with `_`). We must preserve these in the structure (even if we don’t fully process them) so that they can round-trip back out. For example, if we see an unfamiliar tag like `_XYZ`, we attach it as-is under its parent.

**Output of parsing:** The result of this phase is an **in-memory representation** of the GEDCOM file, for example as nested Python objects or dictionaries. For instance, we might end up with a data structure like:

```
records = {
    "HEAD": { ...header info... },
    "SUBM": { "@U1@": { NAME: "Reldon Poulson", ... } },
    "INDI": { }
```

```

    "@I1@": {
        NAME: [{"value": "Robert Eugene /Williams/", "GIVN": "Robert Eugene",
        "SURN": "Williams"}],
        SEX: "M",
        BIRT: { DATE: "2 Oct 1822", PLAC:
        "Weston, Madison, Connecticut, United States of America", SOUR: "@S1@" },
        DEAT: { DATE: "14 Apr 1905", PLAC: "Stamford, Fairfield, Connecticut,
        United States of America" },
        FAMS: ["@F1@", "@F2@"],
        FAMC: [], # if none
        ... other details ...
    },
    "@I2@": { ... },
    ...
},
"FAM": {
    "@F1@": {
        HUSB: "@I1@",
        WIFE: "@I2@",
        CHIL: ["@I3@"],
        MARR: { DATE: "Dec 1859", PLAC: "Rapid City, Pennington, South Dakota,
        USA" }
    },
    "@F2@": { HUSB: "@I1@", WIFE: null, CHIL: ["@I3@"], ... },
    ...
},
"SOUR": { "@S1@": { ...source details... } },
"REPO": { "@R1@": { ...repository details... } },
"NOTE": { ... if any ... },
"TRLR": True # just to mark trailer
}

```

(This is a conceptual sketch; actual structure might differ). Essentially, we will have collections of each record type keyed by their XREF IDs for quick access. We will also maintain cross-reference indices, e.g., a dictionary mapping `xref_id -> record` for all records, to easily look up target records when needed.

**First module to code - `gedcom_parser.py`**: As per the question, *the first thing I would code is the GEDCOM parsing logic*. The initial Python file/module would contain something like:

- A function or class `GedcomParser` with a method `parse(file_path)` that returns the internal structure (or perhaps an object model with attributes for individuals, families, etc.).
- Pseudocode for parsing could look like:

```

function parse_gedcom(file_path):
    data = open file_path in binary
    detect BOM and encoding, ensure it's UTF-8/16 with BOM (else error)

```

```

text = read file as Unicode text
lines = text.splitlines() # preserve order without newline chars
records = []
stack = [] # will hold (level, object) for current hierarchy

for each raw_line in lines:
    # Skip empty lines (shouldn't occur except maybe end of file)
    parse the line into (level, xref, tag, value):
        level = int(first token)
        next_token = second token
        if next_token starts with '@' and ends with '@' and level == 0:
            xref = next_token (e.g. @I1@)
            tag = third token
            value = remaining tokens joined (or None if no more tokens)
        else:
            xref = None
            tag = second token (if second token wasn't an XREF)
            value = remaining tokens (if any)
        create a node = { "level": level, "tag": tag, "id": xref, "value": value, "children": [] }

        if level == 0:
            records.append(node)
            stack = [node]
        else:
            # find parent: ensure stack length > level (otherwise structure error)
            # pop until parent with level == level-1 is at top
            while stack and stack[-1]["level"] >= level:
                stack.pop()
            if stack is empty after popping or stack[-1]["level"] != level-1:
                error "Invalid GEDCOM structure at line X"
            parent = stack[-1]
            parent["children"].append(node)
            stack.append(node)
    end for
return records # list of top-level records (including HEAD, etc.)

```

We would then likely post-process `records` into more convenient dictionaries keyed by XREF IDs (for INDIs, FAMs, etc.) as illustrated earlier.

This parser would also incorporate the validations discussed (e.g., after building `records`, verify that HEAD exists, all cross-references resolve, etc.). Any discrepancies could be logged or handled (depending on whether we choose to strictly abort on errors or try to correct them).

*By parsing and validating upfront, we ensure the data is ready for the next steps of storage and processing, with the structure and content as expected.* If any critical mismatches are found (e.g., a family pointer to a non-

existent family), I would decide how to handle it (perhaps create a placeholder record or drop the pointer with a warning). The ideal is to have an **in-memory representation that exactly represents the GEDCOM content** (including even custom tags or unusual constructs) so that we can eventually regenerate the same file (improved) from it.

## Phase 2: Inserting Parsed Data into a Database

Once parsed, we have a choice: use a **relational SQL database** (like PostgreSQL) or a **graph database** (like Neo4j), or even a combination (e.g., storing JSON in Postgres). I'll discuss how I'd approach each, and which I'd lean towards for this project:

### Option A: Relational (SQL) Database Design

If using PostgreSQL/MySQL, we need a schema that can accommodate genealogical data. A straightforward approach is to create tables corresponding to each top-level record type (Individual, Family, Source, Note, Repository, Submitter, etc.). For example:

- **Individuals table (INDI):** Columns for individual attributes:
  - `id` (primary key, could be an internal serial ID or we could use the GEDCOM XREF like "I1" as an ID string).
  - `xref` (the GEDCOM identifier, e.g., I1, I2, etc., if not used as PK).
  - `name_full` (the full name string with slashes, e.g. "Robert Eugene /Williams/").
  - `name_given`, `name_surname`, `name_prefix`, `name_suffix`, `name_nickname` (columns for name pieces, since 5.5.5 mandates these pieces <sup>7</sup> – this allows direct querying of surnames etc.).
  - `sex` (M, F, or other if 5.5.5 extended it – 5.5.5 introduced an optional `U` for unknown sex as default <sup>15</sup> ).
  - Birth, death, etc. – We have a design choice here: We could have specific columns for common events like `BIRT_date`, `BIRT_place`, `DEAT_date`, `DEAT_place`, etc. However, an individual can have many types of events (christening, graduation, etc.) and attributes (occupation, etc.), and possibly multiple occurrences (multiple occupation entries). It might be better to have a separate **Events/Facts table**.
  - If we go the separate table route: e.g., an **Events table** with fields (`event_id`, `indi_id`, `type`, `date`, `place`, `place_lat`, `place_long`, `age`, `agency`, etc., depending on type). Each row represents one event or attribute linked to an individual (or family). E.g., one row for birth of I1, another for death of I1, etc. We can also have a **FamilyEvents** table for events tied to a family (like marriage).
- **Family table (FAM):** Columns for family attributes:
  - `id` (PK), `xref` (like F1, F2).
  - `husband_id` (FK to Individuals), `wife_id` (FK to Individuals) – note: if the concept of husband/wife is extended (same-sex marriage), the tags might still be HUSB/WIFE in GEDCOM 5.5.5 but with possibly same gender individuals; in any case these fields can be null if unknown.
  - We would need a junction table or array for children: One family can have multiple children. Options: a separate **Child table** with (`family_id`, `child_id`) pairs for each child link, or a text array of child IDs (less ideal for complex queries).
  - Family also can have events (like marriage, divorce). So similarly a **FamilyEvents** table or columns for a single marriage event if we assume one marriage per family. However, GEDCOM allows multiple marriage events in a family (e.g., multiple marriage dates if a couple

married twice or multiple ceremonies) <sup>16</sup> <sup>17</sup>. A separate events table for family events is more flexible.

- **Source table** (SOUR): with columns for title, publication data, etc., plus maybe pointer to a repository.
- **Citation table** for linking sources to events or facts, since an event can have a citation linking to a source (with PAGE, etc.).
- **Notes table** (NOTE): for shared note records (if present).
- **Repositories table** (REPO) and **Submitter table** (SUBM).
- **Media table** (OBJE) if multimedia links are used.

Designing a fully normalized schema is doable but quite involved due to the richness of GEDCOM. There's a risk of ending up with many tables and complicated joins. One has to carefully map GEDCOM's hierarchical structures (like the multiple levels under an event: e.g., BIRT -> DATE, PLAC -> MAP -> LATI/LONG, -> NOTE, -> SOUR) into a relational form. We could flatten some of it (e.g., store BIRT\_date, BIRT\_place in Individuals as mentioned), but that only works for a few key events and ignores the flexibility of GEDCOM (there are many possible event types, custom events, etc.). A more generic design is to have an **Entity-Attribute-Value (EAV)** style table or a JSON column:

- **EAV approach:** A table `IndividualFacts` with columns: `indi_id`, `fact_type` (ENUM of allowed tags under INDI like BIRT, DEAT, OCCU, etc.), `date`, `place`, `value`, `other` (maybe JSON for any substructure like age, agency, etc.), and a separate table `FactSources` linking a fact to source records, etc.
- **JSON approach:** Simply have a JSON or JSONB column in the Individuals table that stores all the detail of that individual as a nested JSON object (including all events, names, etc.). PostgreSQL's JSONB is queryable, so one could still search within it, albeit less directly than normalized columns. This is in line with the approach of converting GEDCOM to JSON for ease of insertion and processing – it treats the GEDCOM structure as a JSON document. In our case, since the parsing already effectively gives us a nested structure, we could insert that JSON straight into a column. The advantage: **simplicity and fidelity** (the JSON structure can mirror GEDCOM exactly, so no info is lost). The downside: more difficult to join or query specific pieces (though one can index JSON). Given that the end goal is to export back to GEDCOM and possibly do some data cleaning rather than heavy analytical queries, storing as JSON might be perfectly acceptable.

Considering the user's note that converting to JSON was suggested due to difficulties in relational insertion, I acknowledge that **storing the data in JSON format inside SQL is a viable solution**. We could, for example: - Create a table `GedcomFiles` with columns `id`, `filename`, `data_json` where `data_json` holds the entire GEDCOM content as a JSON structure (all individuals, families, etc.). Or, - Create separate tables for major record types with a JSON column for details: e.g., `Individuals(id SERIAL, xref TEXT, data JSONB)`, where `data` contains the sub-structure of that individual (names, events, etc.). We might then have a simpler `Families(id, xref, data JSONB)`, etc. This way we can easily fetch an individual's data, and maybe also have top-level searchable fields (like name or birth date) as separate indexed columns for convenience.

**Data insertion logic:** Whichever relational design chosen, I would next write a module to **traverse the parsed structure and insert records** into the DB. For normalized tables, that means inserting each Individual, capturing its new DB primary key, inserting its events into an events table, linking them, etc. For JSON storage, it might be as simple as inserting a row with the JSON. Given the complexity of full normalization and the fact that you want an exact round-trip, I'm inclined toward a **JSON-based storage** or

a light normalization. This also aligns with the idea that we can manipulate the data as JSON (or Python dicts) for standardization and then dump back to GEDCOM.

Alternatively, a compromise is to store the data in a *graph database*.

## Option B: Graph Database (Neo4j) Design

Using Neo4j (or similar) could be very powerful for genealogical data, especially if we plan to query family relationships (e.g., find ancestors, detect loops, etc.) or integrate multiple GEDCOMs. The graph model typically would be:

- **Person nodes** for each Individual. Each Person node can have properties like `name` (full name or separate first/last name properties), `sex`, birth date, death date, etc., as well as an internal `id` property corresponding to the GEDCOM XREF (for reference). In Neo4j, you might label these nodes as `:Person`.
- **Family nodes** for each Family record, or we can choose not to explicitly use Family nodes. There are two modeling approaches:
  - **Including Family nodes:** Create a node for each Family (perhaps label `:Family`), with relationships from that Family to the spouse individuals and child individuals. For example, `(:Family)-[:HUSBAND]->(:Person)` and `(:Family)-[:WIFE]->(:Person)`, and `(:Family)-[:CHILD]->(:Person)` for each child. The Family node itself could have properties for marriage date/place, etc., or connect to an event node for marriage. This aligns closely with GEDCOM's representation of a family as a record that links people.
  - **No Family nodes (direct relationships):** Instead, use relationships directly between Person nodes: e.g. `(:Person)-[:SPOUSE_OF]->(:Person)` for marriages and `(:Person)-[:CHILD_OF]->(:Person)` for parent-child. This is simpler for traversal (you can traverse parent/child links without an intermediate node), but it loses the concept of the family as a grouping. In GEDCOM, a family record can have multiple children and exactly two parents (although one or both may be missing if unknown). Without a family node, representing a situation like an unmarried couple having a child or a single parent family is still possible (you'd just have one parent relationship), but tracking multiple children of the same couple requires something like a common family ID. Many genealogy graph implementations do forgo explicit family nodes and encode parent relationships directly, possibly duplicating a marriage node or property to group children. However, since GEDCOM explicitly groups children in a family record, I lean towards including Family nodes for clarity and to store family events like marriage.

Given Neo4j's flexibility, either approach works. The Neo4j Aura blog example you cited took the route of *no family nodes*, instead creating `:Person` nodes and using relationships MOTHER and FATHER to link to parent persons <sup>18</sup>. They only stored limited info (birth year, death year, etc.) as node properties and ignored the intermediate Family records from GEDCOM for simplicity. That model is fine for analysis, but if round-tripping to GEDCOM is needed, we might want to retain the Family concept. So I would design:

- **Person (Individual) nodes** with properties: `xref` (I1, I2, ...), `first_name`, `last_name` (or `name_full`), `sex`, maybe birth/death years or dates as properties for quick filtering, and any other frequently used attributes. We can omit storing everything as properties; some details (like an occupation or a specific event) can be separate nodes or just kept in a parallel data structure. Neo4j

can also store JSON-like structures or arrays in properties, but it's often better to use separate nodes for things like events if we want to query them.

- **Family nodes** with property `xref` (F1, F2, ...) and perhaps a property `marriage_date` or so. We then create relationships: `(:Person)-[:HUSBAND]-(:Family)-[:WIFE]->(:Person)` to link spouses, and `(:Family)-[:CHILD]->(:Person)` for children. (We could also do spouse relationships directly between persons and still have a Family node just to group children, but I'll keep it simple).
- **Event nodes (optional):** We could create nodes for significant events (Birth, Marriage, Death, etc.) and connect them to persons or families. E.g., a Birth event node connected to the person and maybe carrying date/place as properties. However, for our purpose of data cleaning and re-output, it might be overkill to fully model events as nodes unless we need to query across them (like find all birth locations etc., which could also be done via properties or relational DB). If focusing on cleaning, we might not store events as separate nodes at all; instead, we might store event info as properties or attached JSON on Person/Family nodes just for persistence, then handle details in Python code.
- **Source, Note, Repository nodes:** Similar approach if needed – each could be a node with appropriate relationships (e.g., a Citation relationship from an event or person to a Source node).
- **Pros of graph:** Once data is in Neo4j, finding and enriching relationships (like detecting duplicate individuals across GEDCOMs or computing kinship) becomes easier with Cypher queries. Neo4j can easily handle multi-file data merged into one graph, where separate GEDCOMs might be separate subgraphs that you could later connect if you find matches (just a forward-looking consideration).

**Storing vs. Processing:** We have to decide how much processing to do inside the database vs in code. If using a graph DB, likely we would still *parse in Python*, then use a Neo4j driver to create nodes and relationships. The *standardization and enrichment* (like fixing names/places) could either be done *before insertion* (i.e., clean the data in Python then insert clean data into Neo4j), or *after insertion* by querying and updating the graph (using Cypher or Python drivers). For a relational DB, similarly, we could run SQL UPDATE statements or manipulate JSON columns in SQL, but Python might be simpler for complex text transformations.

Given the project emphasis on having standardized, normalized data and using provided reference datasets (likely loaded in Python or as JSON files), I would perform the **data standardization/enrichment in Python code** after pulling data from the DB. The database in this scenario is mainly for storage and possibly for integrity constraints or multi-file integration, not for heavy transformation logic (which is easier to express in Python).

**Conclusion on choice:** Both Postgres and Neo4j can work. If I want to leverage existing JSON datasets and Python libraries (for example, for date parsing or geographic coordinates), using Python to manipulate the data and Postgres as a store may be simplest. If the goal was more on exploring family connections, Neo4j would shine. Since the end goal is to export a GEDCOM, not to maintain a live interactive application, a relational/JSON approach is perfectly fine.

Given the user is open to either and specifically mentions PostgreSQL, I will proceed assuming a **PostgreSQL database with JSON columns** for ease of implementation. This means after parsing, I would insert each top-level record's data structure into the DB. For example: - `INSERT INTO individuals(xref, data) VALUES ('I1', <JSON of individual I1>)` - Do similarly for families, sources, etc. Alternatively, one big JSON for the whole file could be stored, but per-record gives more flexibility if we want to update one person at a time.

This way, the database contains all data, and we can run checks or queries as needed (Postgres can even enforce relationships – e.g., we could ensure that every FAMS in an Individual's JSON points to a valid family xref present in the families table, either via triggers or by application-level checks).

**Ensuring fidelity in DB:** We must ensure that whatever goes into the database can reproduce the original file. Using JSON largely ensures that since we can store even unknown tags easily. If we normalized into strict columns, we might inadvertently drop data that doesn't fit our schema. So, I would either include a catch-all JSON for extra subrecords or just go full JSON as described. This strategy addresses the earlier difficulty the user had: by using a JSON structure that mirrors GEDCOM, insertion becomes easier (no complex join logic needed) and nothing gets lost, and upon export we can reconstruct exactly (just traverse the JSON).

## Phase 3: Retrieving Data & Preparing for Processing

After storage, the next step is to retrieve the data from the database into a form suitable for processing and standardization. If we stored data as JSON in Postgres, we could simply fetch those JSON objects into Python. If we fully normalized, we would run queries to assemble all relevant pieces (which is more complicated – e.g., joining name parts, events, etc., to get a composite view of each individual).

**Likely approach:** Query the database to get all individuals (and their details), all families, etc., and recreate the internal structure or use it directly. For example, if each individual's data was stored as JSON, I could load each into a Python dict. If using Neo4j, I would query all Person nodes and their relationships to gather the equivalent data (like for each person, get their names, events, etc.). In code, I'll end up with a similar structure as I had after parsing, but perhaps augmented with database IDs or cross-file info if relevant.

It might even be that we do not need to store-and-retrieve for a single file processing – we could parse and directly process in memory without ever persisting. However, persisting early can be helpful for debugging and ensuring data integrity (and necessary if we plan to handle multiple files or a large file that might not comfortably fit in memory, although most GEDCOM files are only a few MBs to maybe tens of MBs for very large trees).

For now, assume we've inserted and now loaded the data back into Python as structured objects (perhaps the same as the parse output, or we continue using that parse output without hitting the DB in between, which is possible if immediate processing is the goal).

## Phase 4: Data Standardization and Normalization

Now comes the core processing where we apply rules and reference data to **clean and standardize** the genealogical information. We'll focus on the key fields mentioned: **NAME**, **DATE**, **PLAC** (and their sub-tags), as well as other child tags like those under NAME and PLAC, etc.

I would likely create a separate module or set of functions for each type of data standardization, for clarity. For example, a `standardize_name(person)` function, `standardize_date(date_str)` function, `standardize_place(place_str)` function, etc., using internally available reference data.

Let's break down the tasks:

## 4.1 Personal Name Standardization (INDI.NAME and sub-tags)

Each Individual can have one or more NAME records (multiple name entries if the person had aliases, married names, etc.). GEDCOM 5.5.5 requires that for each name, the parts (prefix, given, surname prefix, surname, suffix, nickname) appear as subrecords if those parts exist <sup>7</sup>. Our goals for name standardization:

- **Ensure the presence of all relevant name piece tags** for each name. If the original file was missing some (e.g., many GEDCOM 5.5.1 files might have just `1 NAME John /Doe/` without explicitly listing `2 GIVN John` and `2 SURN Doe`), we will add them. We can derive these by parsing the `NAME` value:
  - The portion between the slashes `/ . . . /` is the surname. Text before the first slash is the given name(s) and prefix, and text after the surname slash (if any) is suffix. E.g. "Lt. Cmndr. Joseph /Allen/ Jr." – prefix "Lt. Cmndr.", given "Joseph", surname "Allen", suffix "Jr." <sup>19</sup>. We would split accordingly. (We should be careful to handle cases where surname is empty – e.g., "Mary // indicating an unknown surname – the spec allows the surname to be empty between slashes).
  - Populate the tags: `NPFX`, `GIVN`, `SURN`, `NSFX` as needed. If any part is truly not present (like no prefix or suffix), we don't create that subrecord. (Note: per 5.5.5, these subrecords are mandatory only if the name part exists; they are not to be present with empty values.)
- **Normalize name casing and spacing:** For consistency, we might choose to capitalize surnames or not depending on preferred style (GEDCOM itself often uses mixed case as entered). We might ensure the surname is in title case, given names in title case, etc., unless there's a reason to preserve case (like "van der Berg" surname which has lowercase parts). Since the user has datasets of standardized data, perhaps they have a list of correct spellings for surnames or common nicknames. We could use those:
  - For example, if the given name is "Robt.", we might expand it to "Robert" using a name dictionary. If a nickname is in quotes or parentheses in the name line (some data entry conventions put nicknames in quotes), we could move that to the NICK tag.
  - Remove any unnecessary punctuation. Ensure prefix abbreviations are consistent (e.g., "Dr." vs "Dr" – probably keep the period).
- **Validate consistency:** After splitting, we can recombine the parts to double-check we get the original full name string (except differences in spacing/punctuation we intentionally standardized). The spec says the reader must not arbitrarily split differently; we follow what the file says <sup>4</sup>. In our case, if we derived parts, we are essentially upgrading a 5.5.1 style name to 5.5.5 standard. We should be sure that the *concatenation of NPFX + GIVN + SPFX + SURN + NSFX* (with appropriate spaces and the surname in slashes) matches the `NAME` line value. If not, there's inconsistency (maybe the original had something odd like a middle name outside of GIVN in some older file). We'll resolve those as needed (maybe by adjusting the parts or leaving the full name in NAME and using parts only as hints).
- **Multiple names:** If an individual has multiple NAME entries (e.g., maiden name, AKA, etc.), ensure each is handled similarly. We might also use the `NAME.TYPE` subtag to indicate the type of name (e.g., married name, AKA) if the data or user's preference dictates. Standardization might involve tagging one name as primary and others with TYPE like "alias" or "maiden" etc.
- **Enrichment:** using external data – possibly a list of known **name variants**. E.g., if the person's given name is "Bob", we might add a note or an alias name "Robert" if we know that's the formal name. Or if they have a nickname not given, we could populate the NICK tag if our dataset tells us a known nickname for that formal name. This gets into user preference territory, but it might be something

they intend (since they mentioned enrichment, maybe they have a DB of known nicknames or common name expansions).

- **Output normalization:** Ensure that after processing, each Individual's name structure is consistent and follows best practices. For example, *every name part present must not be empty or placeholder*. If an individual had no surname originally (just "Mary /[blank]"/"), we could leave surname blank but it might be more consistent to put something like "Unknown" or at least keep the slashes with nothing between them to indicate an unknown surname (which is the GEDCOM way).

## 4.2 Date Standardization (All DATE tags)

Dates in GEDCOM can be tricky. GEDCOM 5.5.5 supports the same date formats as 5.5.1, which include: - Exact dates: e.g. 12 OCT 1890 (day, month, year). - Partial dates: OCT 1890 (month year), or 1890 (year only). - Date ranges/periods: BET 1900 AND 1910 , FROM 1900 TO 1905 . - Phrases: in rare cases, a date can be ABT 1900 (about 1900), BEF 1880 , AFT 1700 . - Dual dates (for Julian/Gregorian overlap), e.g., 15 FEB 1691/2 (meaning 1691 in one calendar, 1692 in another). - Calendar escape sequences: e.g., @#DJULIAN@ 1 APR 1700 to explicitly use Julian calendar. (5.5.5 still allows these escapes, like @#DFRENCH R@ for French Republican, etc., but also notes they complicate processing <sup>20</sup> ). - Text in dates (shouldn't generally happen except in a phrase context or in older GEDCOM where people put things like "circa 1900" which ideally should be ABT 1900 ).

For standardization: - **Normalize formats:** We should convert all dates to a consistent format. GEDCOM expects the day as a one- or two-digit number (with no leading zero on day < 10, I believe), month as three-letter English abbreviation in uppercase (JAN, FEB, MAR, etc.), and year as four-digit (or a range if needed). We should ensure months are in the proper format. If we find a date like "1 January 1900", we convert to 1 JAN 1900 . If month or day names were in lowercase or mixed case, uppercase them (since GEDCOM typically uses uppercase month names). - **Standardize modifiers:** If someone wrote "circa 1900" in a note or as a date value, we should convert that to the official GEDCOM term ABT 1900 . Similarly, "bef. 1880" -> BEF 1880 . Using our reference data or logic, we can identify such cases. - **Validate date values:** Check that the day is within the correct range for the month, and month is a valid month. If we have an approximate date (ABT), that's fine. If we have a range (BET/AND or FROM/TO), ensure both dates are valid and the range logic makes sense (the first should be <= second). If any date is clearly invalid (e.g. "32 JUN 2000"), we might correct it if we have external knowledge or mark it for manual review. - **Calendar escapes:** It's unlikely we have to change these, but if our goal is standardization, one could choose to convert dates from alternate calendars to the Gregorian equivalent or simply ensure the escapes are properly used (the spec says if an application doesn't support a calendar and a user enters a date with an escape, the @#...@ should remain and not be doubled or anything <sup>21</sup> ). If the data has no such escapes, no issue. If it does and we have the capability, we might convert a Julian date to Gregorian (enrichment), but that might be beyond scope. Probably we leave alternate calendars as-is but ensure they are well-formed. - **Leading zeros:** Ensure day numbers have no leading zero (the spec's examples typically show e.g. 2 OCT 1822 not 02 OCT 1822 ). Actually GEDCOM might allow 02 but generally one-digit day is fine. Consistency is key. - **Standardizing date phrasing:** If the user's data sets include, say, common phrases or alternate date formats, we could use that. For example, if some GEDCOM had a date like Jun 1880 (3-letter month not in uppercase or missing day), we correct to JUN 1880 . If they have quarters or seasons (some genealogists write "Spring 1850" which isn't GEDCOM standard), we might translate "Spring 1850" to, say, BET MAR 1850 AND MAY 1850 or leave it as a phrase in a note because GEDCOM doesn't have a direct notion of seasons. - **Enrichment:** Possibly add calculated dates. For instance, if only an age at an event is given and date missing, we might compute an approximate date (this is advanced and usually not done

automatically without context). Or if we have an external database of historical dates (not likely relevant here). More plausible enrichment: if a death date is present but no burial event, maybe we could add a burial event with an estimated date (some people do that, but it's speculative - probably out of scope unless provided). - **Output:** All date values should end up in a valid GEDCOM format string. We might also choose to uniformly use full upper-case for the month and prefixes (ABT/BEF etc. are traditionally uppercase in GEDCOM).

We should test after processing that any date string still matches the allowed GEDCOM date grammar. A good practice is to incorporate a date parser (there are libraries or we can write a small one) to verify that each date value can be interpreted.

### 4.3 Place Standardization (PLAC tags and substructures)

Place names often benefit the most from external reference data (for normalization and enrichment). GEDCOM place values are free-form text, typically a comma-separated list from smallest locality to largest (e.g., "City, County, State/Province, Country"). There's no enforced format except what genealogists conventionally use. GEDCOM 5.5.5 keeps places as simple text but allows optional sub-tags for phonetic variation, romanized variation, and a map coordinate under the PLAC structure <sup>8</sup>.

Standardizing places involves: - **Consistent formatting of place text:** We should decide on a standard order and naming convention. Usually it's already "Town, County, State, Country". If some entries are incomplete or use abbreviations, we normalize those. For example, ensure "USA" is spelled out "United States of America" (or vice versa if abbreviations are preferred, but full names are often better for clarity). Ensure each level is separated by a comma+space, and no trailing commas. Trim whitespace around each part. If we have entries where county or state is missing, we might fill them if known (enrichment). - **Use reference datasets:** The user mentioned having JSON datasets for standard place names. Possibly a list mapping variant names or abbreviations to standard names (e.g., "Mass." -> "Massachusetts"). I would integrate those by doing lookups on each place part: - Split the place by commas into parts [place parts already appear to be comma-separated in examples]. - Trim each part and try to match it to a standardized name from our dataset for that level (maybe the dataset is organized by country/state names, etc.). - For example, "Pennington" might be recognized as a county name and we ensure it has the correct spelling. Or "Stamford, CT" might be expanded to "Stamford, Fairfield, Connecticut, United States of America". - This could also involve adding missing higher-level jurisdictions: If the GEDCOM place was "Stamford, Connecticut", we might append ", United States of America" if we know that (assuming context that Connecticut is in USA). - **Coordinate enrichment:** GEDCOM 5.5.1 introduced the ability to include latitude (LATI) and longitude (LONG) under a PLAC->MAP substructure. 5.5.5 keeps that. The spec notes a drawback: if the same place is used in many events, the coordinates get repeated every time <sup>22</sup>, which bloats the file. We can't change that about GEDCOM, but **in our database model** we could avoid duplication by having a separate Place reference table. However, when exporting back, we either have to include coordinates for each occurrence or choose to include them only on one occurrence. The standard implies if you have coords, you include them every time. To manage this: - We could use our place dictionary to find coordinates for each unique place name. Perhaps the user's datasets include latitude/longitude for places. - As part of enrichment, for each place name, attach a MAP/LATI/LONG subrecord if not already present. For example, if "Honolulu, Hawaii, USA" is in our data, we might add MAP\n+1 LATI N21.3069\n+1 LONG W157.8583 (like in the example for Barack Obama's birth place <sup>23</sup> <sup>24</sup>). - If a place already has coordinates but our data indicates different ones (or more precise ones), we might update them or at least flag discrepancy. - Because repeating coordinates is redundant, an

alternative is to create a **global place list**: we assign an internal ID to each unique place and store it once with coordinates in a separate structure. But since GEDCOM doesn't support referencing a place record (there is no place entity with an XREF in 5.5.5), we would still have to embed the coordinates in each PLAC instance for full compliance. So likely we do embed them if enrichment is desired. - **Phonetic/Romanized name variants**: If the dataset or requirement includes supporting these (for non-Latin scripts, etc.), we could add **FONE** (phonetic) and **ROMN** sub-tags. For instance, if we have a place in Cyrillic and we want a Latin transliteration, **ROMN** can be used. For standard U.S. data this is probably not needed. - **Normalize abbreviations**: e.g., "St." to "Saint" for place names if desired (Saint could also be standardized to "St" consistently if using abbreviation, but typically one picks one style). - **Case normalization**: Perhaps ensure each comma-separated part uses Title Case (Each Important Word Capitalized) except certain conjunctions. Many place names are already title case, but if not, we fix that (e.g. "den Haag" -> "Den Haag"). - **Trailing data**: If any place field had extraneous info (like "Boston (now part of XYZ)", maybe move such commentary to a NOTE instead of inside the PLAC field, if aiming for clean data). - **Validation**: After standardization, the place string should remain a plain string (commas allowed). GEDCOM doesn't impose a specific format beyond it being text. But we should ensure no forbidden characters (it should not contain newline or stuff unless broken into multiple PLAC lines via CONC, which our parser would have handled anyway). - **Potential multi-language issues**: Not heavily mentioned, but if the file has place names in local language and the user wants them standardized to English, that could be part of enrichment (translating names of countries, etc.). Using provided datasets, we could map "Deutschland" to "Germany", etc., as needed.

In addition to NAME, DATE, and PLAC, the user said "*and all child GEDCOM tags*". This likely refers to subordinate tags that we need to handle in the process. We've already touched on many: - For NAME: child tags are the name pieces (NPFX, GIVN, etc.) and possibly FONE/ROMN if present for alternate scripts of the name. We would standardize those by ensuring they match the main name and are correctly cased. - For PLAC: child tags include FONE/ROMN (alternate spellings) and MAP/LATI/LONG. We covered adding/updating coordinates and possibly alternate names if needed (but unless we have alternate language names for the place, we might leave FONE/ROMN empty). - For DATE: not exactly child tags, but sometimes DATE has a TYPE in event structures (GEDCOM allows specifying a TYPE of event, but that's separate from DATE). - For events (like BIRT, MARR, etc.): They can have a <<NOTE\_STRUCTURE>> or <<SOURCE\_CITATION>> as children. Standardizing those might mean: - Ensuring any notes are properly concatenated (GEDCOM notes can be spread across multiple CONC/CONT lines – our parser likely already merged them, but when output, we need to split long lines per GEDCOM line length limits, which we will handle in export). - For source citations: ensure consistency in formatting of PAGE, etc., or enrich with a repository link if needed. We might not have much to change unless we have data to fill missing citations or standardize source titles. - For addresses (if present under e.g. a SUBMitter or REPO or even an individual's address): Standardize abbreviations (e.g., "St." -> "Street"), ensure each address part (ADR1, CITY, STAE, POST, CTRY) is filled or use proper country names etc. This could be part of enrichment if we have address data.

Essentially, the approach is: for each record type and each tag, apply the relevant cleaning: - Individual: NAME (and sub-tags), SEX (maybe enforce M/F/U or standard values; 5.5.5 introduced possibly new sex values like "X" for non-binary <sup>25</sup> – if we encounter non-standard like "U" or blank, we could set it to U for unknown as default <sup>15</sup> ), Events (dates/places as above), Family links (just ensure pointers valid; no cleaning needed except maybe ensure each child listed in family reciprocally has that family in their FAMC). - Family: ensure consistency of HUSB/WIFE vs the individuals' SEX (the spec notes HUSB should point to a male individual, etc.; but also allows one-parent families) <sup>26</sup> – we might check that and possibly flip if someone had them swapped (fixing data errors). - Also ensure children ordering maybe (spec encourages ordering

children by birth date <sup>27</sup>). We could sort the children list by birthdate as an optional normalization (and note to preserve original order if needed, but usually sorting is considered an improvement unless the user had a deliberate custom order). - Source/Notes: not much to standardize unless we have external databases to standardize source titles or repositories. If so, apply those (e.g., ensure repository names or addresses match a standard form). - Remove truly duplicate entries (if any redundant data exists) – though GEDCOM rarely has exact duplicate lines except maybe multiple NAME or OCCU entries on purpose.

The **first coding tasks for processing** would be to implement these transformations. Likely, I'd write small functions or methods within a `GedcomProcessor` class: - `normalize_name(person)` that goes through each name entry in a person's data. - `normalize_dates(record)` that scans any DATE in the record (individual events, family events, source data events, etc.) and normalizes the text. - `normalize_place(record)` similarly for any PLAC tags found in the record (individual's events or family events). - Possibly a generic traversal that finds all instances of a certain tag in the nested structure.

One approach is to do a tree traversal of the entire data structure and handle tags by name:

```
function normalize_record(node):
    for each child in node.children:
        switch(child.tag):
            case "NAME": normalize_name_child(child)
            case "DATE": child.value = normalize_date_string(child.value)
            case "PLAC": normalize_place_child(child)
            case ... (recurse into child for deeper substructures)
```

But since we also have our data grouped by type (like individuals dict, families dict), we might just loop through individuals and handle their known sub-structures (names, events, etc.), then families etc. Either way works.

## Phase 5: Data Enrichment

Enrichment often overlaps with standardization, but specifically it means **augmenting the data with new information** that wasn't present in the original, using external sources. Based on the question, enrichment might include:

- **Filling missing pieces:** If an Individual has a known missing detail that we have in an external source, add it. For example, if our dataset tells us the middle name or maiden name of a person which wasn't in the GEDCOM, we could add an alternate NAME entry or add to the existing name.
- **Adding standardized forms:** For places, as discussed, adding latitude/longitude to places is enrichment. Also adding country names if they were omitted.
- **Geocoding or IDs:** Perhaps assigning unique IDs to places or linking them to an authority (some genealogists link to GeoNames IDs or FamilySearch place IDs, but GEDCOM 5.5.5 doesn't directly support that except via maybe a Note or user tag).
- **Historical context:** Unlikely needed, but sometimes enrichment could mean adding tags like an estimate of age at death (GEDCOM has AGE tag under Death event to record age). If not present and

if we have birth and death, we can calculate age and put it in the AGE field for the Death event.

Similarly, if marriage date and birth dates are there, maybe add AGE for marriage event.

- **External IDs:** If the user has JSON datasets, maybe they include mapping of person to an external database. We could add a REFN (user reference number) or even a custom tag like \_FSID for a FamilySearch ID if that was relevant. But this depends on their datasets.

- **Sources enrichment:** If some facts are unsourced, and the user has a database of sources, enrichment might mean adding a source citation for a birth or death from a known database. This is a complex task and usually not automated without a match, so perhaps out of scope unless the user specifically has something like "when standardizing, also link to source X if place is Y" – probably not automatic.

- **Quality flags:** GEDCOM has a concept of **PEDI** for pedigree (e.g., an adopted child might have PEDI adopted, as in the example for Joe Williams above <sup>28</sup>). If the data lacked that and the user knows adoption vs birth relationships, they could enrich by adding **PEDI** tag to the FAMC link.

Similarly **FAMC** links can have a NOTE or a **AGE** at adoption, etc. These are minor details but part of completeness.

Given we have specific mention of JSON datasets for standard forms, I suspect the enrichment is primarily about **places** (like a gazetteer) and **names** (like a lexicon of name spellings, nicknames, etc.). Possibly also for **dates** (maybe a calendar conversion table or list of holidays but less likely).

In implementation, enrichment would be integrated with the normalization functions: - For place: while normalizing "Stamford, CT", lookup "CT" in a states JSON to get "Connecticut", and possibly that it's in USA, then attach country. Also retrieve lat/long from a coordinates JSON and attach them under PLAC. - For name: while normalizing "Robt. /Williams/", recognize "Robt." from a name dictionary and change to "Robert". Or if we have "Robert /Williams/ Jr" and a dataset of suffixes, we confirm "Jr" is standard (maybe change "JR" to "Jr." with proper punctuation). - For dates: maybe use a library or data to convert a French Republican calendar date to Gregorian if encountered (this is a bit advanced; if not needed, skip). - Possibly use a **timeline dataset**: e.g., if someone's birth year is ~1820 and our dataset says "the 1820 census exists for X", maybe not directly relevant. Probably not this.

We would also incorporate any **existing JSON formats** the user has created during earlier processing attempts (the question implies they have JSON from prior steps). We might either reuse that or map it into our approach if it contains standardized values.

## Phase 6: Preparing the Output (Reconstruction of GEDCOM)

After standardization and enrichment, we should have a cleaned dataset in memory (and optionally we could update the database with these cleaned values too, if we want to store the standardized form). But the ultimate goal is to output a well-formed GEDCOM 5.5.5 file that reflects the new standardized data.

**Reconstruction steps:** 1. **Assign XREF IDs:** We must ensure each record retains a unique identifier. Ideally, we kept the original IDs (I1, I2, etc.) and did not change them. If we combined multiple files or created new records, we have to generate new IDs for those. For example, if we added a new individual or note during enrichment, we need a new **@I#@** or **@N#@** that doesn't collide. A strategy is to pick a range or prefix for new IDs (like if file had I1..I100, start new ones at I101 onward). For this single-file scenario, likely no new individuals, just enriched data on existing, so IDs remain the same. 2. **Ordering of records:** Typically, GEDCOM files start with the HEAD, then SUBM (submitter) record, then all individuals, then families, then

other records (sources, notes, repositories), and finally the TRLR. We should follow the same order as the original if possible. Many programs list individuals first, families next, etc., but the GEDCOM spec technically doesn't enforce a strict order of the different record types (aside from HEAD first, TRLR last). However, to make diffing easier (comparing new vs old file) and keep things organized, I would maintain the original ordering of records as much as possible. Our `records` list from parsing captured records in order encountered; we can use that ordering for output. This ensures, for instance, if the original file had individuals interwoven with families (some GEDCOMs might list in order of entry), we preserve that. But generally it's HEAD, then SUBM, then others.

**3. Constructing lines:** We'll need to convert our internal structured data back into GEDCOM lines with levels. This is essentially the inverse of parsing. A recursive traversal of each record's children can generate lines:

- Start with level 0 lines for each top-level record: e.g., `0 @I1@ INDI`.
- Then for each child of that record, output `1 TAG value` (or `1 TAG @XREF@` if it's a pointer).
- Then go deeper for sub-subrecords: `2 TAG value`, and so on.
- Use the same indentation levels as originally (the structure ensures we know the level).
- We must also be mindful of long values: GEDCOM line length is limited (about 255 characters per line historically; the spec 5.5.5 might allow a bit more but still advises splitting long text). For example, long NOTE texts or very long concatenated place names may need to be split using `CONC` and `CONT` lines. The rule is: if a value exceeds the line length, break it and use a `1 CONT` for a new line (if a natural break like a newline in text) or `1 CONC` if it's just a long continuous text. Our internal data likely still has the full text (with newlines preserved in a NOTE's value maybe). We should implement a function to split a long string into allowed line lengths at spaces if possible, adding `CONT/CONC` tags of the appropriate level.
- Since we probably merged `CONC/CONT` when parsing (for ease of editing the full text), we now need to re-split for output. We must do this carefully to not insert unwanted spaces or lose any.
- For example, if a note value has actual newline characters, those become separate `CONT` lines. If just long, use `CONC`. The spec requires not to introduce or remove whitespace around the breaks <sup>29</sup> (i.e., a reader should preserve spaces, and we as writer should split only where needed and exactly as rules say).

**4. Include all tags, even if unchanged:** We output everything, including any tags we didn't specifically process. Because we preserved unknown tags, we should output them as they were (or maybe standardized their position or order if needed, but generally keep as is).

**5. Updated HEAD:** In the HEAD record, we might update the `DATE` to the current date/time of export (many programs do this, putting the date of file creation). The HEAD.SOUR (source software) could be updated to reflect our application name or left as original. If we want to indicate the file has been standardized by our tool, we might change HEAD.SOUR to something custom (but it's optional). We also ensure HEAD.GEDC.VERS remains "5.5.5" and FORM "LINEAGE-LINKED" etc. If we changed character encoding (likely we keep UTF-8), ensure HEAD.CHAR is "UTF-8" and if we output in UTF-8, probably use `1 CHAR UTF-8`. - Also ensure the BOM is added at the very beginning of the file when writing.

**6. Final TRLR:** Output `0 TRLR` at end, as required.

**Validation of output:** After writing the GEDCOM text, I would ideally run it through a GEDCOM validation tool or at least re-parse it with our parser to ensure no structural errors. We should double-check that:

- The number of individuals, families, etc. matches the original (unless intentionally added).
- All cross-references still resolve. (For instance, if we added coordinates to every place, we didn't introduce new cross-references, so that's fine; but if we accidentally referenced a source that doesn't exist, that would be a bug).
- The file can be read by another GEDCOM reader without errors. Given we conformed to spec and improved data quality, it should be acceptable. GEDCOM 5.5.5 readers will be strict, but our changes (like adding missing name pieces, proper date formats, etc.) are making the file **more** compliant, not less <sup>30</sup>.

At this point, we have the final `.ged` file ready. If multiple GEDCOM files were processed, we'd do this for each or possibly merge them if that was a goal (merging is another complex topic though, not explicitly asked here).

## Technology Choice: Why Python (and Alternatives)

I would implement the above primarily in **Python**. Python is well-suited for text parsing and data manipulation, and has libraries for working with JSON, date parsing, and database connectivity (e.g., `psycopg2` for PostgreSQL, or `py2neo` for Neo4j). It also allows rapid development and easy debugging for this kind of data transformation task. The pseudo-code I provided can be translated to actual Python readily.

**Advantages of Python:** - Easy string handling and file I/O, which is crucial for reading/writing GEDCOM (essentially a text format). - Dynamic data structures (dicts, lists) to represent the GEDCOM tree intuitively. - Libraries for validation or format conversion if needed (although GEDCOM is niche, one could use `gedcom` or `anytree` library to assist with hierarchical parsing if not doing it manually). - Good support for JSON and interfacing with Postgres (which supports JSON fields directly). - Since performance is usually acceptable for files of a few MBs (even a 100MB GEDCOM can be handled in Python with optimized code), the convenience outweighs lower-level languages. If needed, performance bottlenecks (like parsing huge files) could be mitigated with optimized code or using PyPy or C extensions.

**Alternatives:** - **Java or C#**: Strong typing and existing libraries (e.g., `gedcom4j`) for Java, which parses GEDCOM) could be used. They might be more verbose to implement custom logic but are certainly capable. If this were to be a large-scale production system, those could be options. However, since we want flexibility with JSON and quick iteration, Python is ideal. - **Neo4j (Cypher)**: Some processing could be done with Cypher queries (like fixing data in the graph), but complex string manipulations (like standardizing names) are easier in a general-purpose language. - **SQL**: We could do some normalization in SQL (for example, using regexes or functions to replace text). For instance, Postgres could update all place names with a certain pattern. But writing those by hand is more error-prone and less maintainable than doing it in Python with clear logic and perhaps using regex or external dictionaries easily.

Given the user's project context, Python seems to be the recommended choice for the first implementations. We can always integrate with a SQL database or export CSV for Neo4j import as needed (like how the Neo4j blog used a Python script to produce CSV for import <sup>31</sup> <sup>32</sup> ).

**First Python file revisited:** So to clarify, the very first thing I'd code is the **parser** (e.g., `parse_gedcom.py` as described in Phase 1). Right after that, I might code a simple test to ensure that parsing a sample file (like the attached GEDCOM example) produces the expected internal structure. Then I would code a module for **database insertion** (maybe `store_gedcom.py`) if using a DB immediately, or skip directly to processing functions in a `process_gedcom.py` if doing in-memory.

However, since the question asks "*what would your first Python file be and what would be all the direct steps to parse GEDCOM into a format for processing the data?*", I'll summarize that now:

- `gedcom_parser.py` : This module handles reading the GEDCOM file and converting it to a Python data structure (e.g., dictionaries/lists or custom objects). Steps inside:

- Open file, verify BOM and encoding [2](#).
- Read all lines; parse each line into components (level, optional id, tag, optional value) [5](#).
- Build the hierarchical structure by nesting lines according to their level (using a stack) [12](#).
- Validate structure and cross-references (e.g., ensure pointers match targets, required tags present, name pieces consistency) [4](#).
- Return the structured data (perhaps as a dict of records or similar).
- After parsing, we have the data “*in a format for processing*” – likely this means either as Python objects or as JSON that can be easily manipulated. We could dump it to JSON at this point if we wanted (for debugging or even to use as intermediate), but not necessary.

From here, subsequent files might be: - `gedcom_standardize.py` : Contains functions to go through the parsed data and apply standardization/enrichment (as discussed in Phase 4 and 5). It would likely import the parser, then operate on the data. - `gedcom_export.py` : Contains code to write out the GEDCOM file from the processed structure, dealing with formatting and line breaks.

If using a database: - `gedcom_to_sql.py` : to insert parsed data into SQL tables (if needed before processing). - `gedcom_from_sql.py` : to read it back or directly query from SQL for processing.

However, an agile approach might parse, process in memory, and only then store or output, depending on needs. The user did mention difficulties with SQL insertion which led to JSON, so perhaps the new approach might skip heavy normalization in SQL and use JSON or just use the DB as a backing store rather than a working area for transformations.

## Summary of Approach

In summary, **my approach** to this GEDCOM processing project is:

- **Parsing:** Write a custom GEDCOM 5.5.5 parser in Python to read the file into a hierarchical in-memory structure, validating format compliance. This ensures we can accurately capture every detail of the GEDCOM (including all tags and links) [5](#) [12](#). This is the foundational step and the first code to implement.
- **Database Design:** Choose a storage method that balances fidelity and usability. I favor using PostgreSQL with JSON columns for each record type (or even a single JSON for the whole file) to store the data as parsed. This way, insertion is straightforward (just dump the JSON) and nothing is lost in translation. If relational tables are used, carefully design them to accommodate multiple events and names per person, or use an EAV or link table approach. Alternatively, consider Neo4j for representing individuals and families as nodes with relationships, which mirrors the genealogy graph and simplifies relationship queries (family trees are naturally graphs [9](#)). In either case, ensure that all cross-references (person-family links, source citations, etc.) are preserved correctly in the schema.
- **Validation in DB:** After insertion, run checks (e.g., SQL queries or Python assertions) to verify that each link is intact (every FAMS has a corresponding family, every family's child link points to an individual, etc.). This double-checks that our parsing and storage didn't drop anything.
- **Processing (Standardize & Normalize):** Load the data into a processing module (in Python). Apply transformations:
- **Name fields:** Split and normalize personal name parts, add missing sub-tags, use reference data to correct abbreviations or add nicknames [7](#).

- **Dates:** Reformat all date strings to a consistent GEDCOM format (DD MMM YYYY), handle prefixes (ABT/BEF), ensure valid values.
- **Places:** Normalize place names (consistent commas, full region names), add country if missing, use external data to correct names and add coordinates via PLAC.FONE/ROMN/MAP sub-tags as appropriate <sup>8</sup> <sup>22</sup>.
- **Other tags:** Standardize address formats, event descriptions, notes (maybe remove excessive whitespace or ensure CONT lines split at sensible points), ensure uniform use of abbreviations (or none) across the file.
- **Enrichment:** Integrate external JSON datasets to enhance data:
  - For example, use a standardized list of place names to correct any misspellings and add latitude/longitude <sup>22</sup>.
  - Use a name dictionary to expand nicknames (e.g., "Liz" to "Elizabeth") or add missing name types (married name vs maiden name).
  - If provided, use historical data to fill in missing pieces (like adding an approximate birth date if only age in census was given – though that would be speculative).
  - Essentially, any additional data the user has collected as "standard" can be applied at this stage programmatically.
- **Updating DB (optional):** If we want the DB to hold the cleaned data, we could either update the JSON fields or add new columns for standardized values. This might not be necessary unless we need to persist the changes or do further SQL-based analysis. We could also skip updating the DB and just proceed with export using the in-memory cleaned data.
- **Re-export to GEDCOM:** Traverse the cleaned data structure to produce a new GEDCOM file:
  - Write the HEAD (with updated metadata, e.g., date) and ensure it reflects GEDCOM 5.5.5 format.
  - For each record (INDI, FAM, etc.), output the lines in proper order and levels, including all newly added or modified subrecords (like name parts, coordinates) and all original ones that remained.
  - Maintain original cross-reference IDs (unless there was a need to renumber due to merges/new additions).
  - Ensure the output conforms strictly to GEDCOM syntax (e.g., lines not exceeding length limits, use of CONC/CONT for long text, double-@@ escaping if needed for literal '@' in text, etc. – the spec has many details like handling @@ to represent @ in notes <sup>33</sup>, which our code must implement).
  - End with 0 TRLR trailer.
- **Verify output:** Optionally parse the output GEDCOM with the same parser or a GEDCOM validator to confirm it's an exact functional round-trip of the input. Differences should only be in the content we intended to change (e.g., standardized names or added fields), not in structure or lost data. The ideal is that if someone compared the original and new GEDCOM, they'd see the improvements but no arbitrary alterations or omissions.

Finally, regarding **multiple GEDCOM files**: The architecture above can scale to multiple files by either processing them sequentially or merging into one database. If merging (to find duplicates, etc.), a graph DB becomes useful (to connect individuals across files). But if just batch processing files independently, we can reuse the same pipeline for each file. It's wise to design the code modularly so it can handle a directory of GEDCOMs in a loop if needed.

---

This design provides a **comprehensive roadmap** for tackling the GEDCOM project. The very first coding step would be implementing the parser, as everything hinges on accurately reading the GEDCOM structure. From there, each component (DB schema, standardization functions, etc.) builds on the previous, ensuring

that by the end, we have a robust system that can input a raw GEDCOM and output a cleaned, standardized GEDCOM that matches the original in genealogical meaning.

Throughout, the focus is on maintaining data integrity (so that the database and the output GEDCOM are an exact match to the original information <sup>4</sup>) while enhancing quality and consistency. By following this plan, we leverage the strengths of Python for parsing and text processing, use databases where they add value (ensuring consistency and enabling future scalability), and adhere strictly to the GEDCOM 5.5.5 standard, resulting in a high-quality end product.

### Sources:

- The GEDCOM 5.5.5 specification (for rules on format, required tags, and examples) <sup>7</sup> <sup>4</sup> <sup>8</sup> <sup>22</sup>.
  - Developer insights on GEDCOM parsing and data modeling <sup>5</sup> <sup>12</sup>.
  - Community advice on building GEDCOM parsers incrementally <sup>11</sup>.
  - Neo4j genealogy import example (for graph modeling perspective) <sup>18</sup>.
  - General GEDCOM handling notes indicating complexity and need for strict compliance <sup>1</sup> <sup>9</sup>.
- 

<sup>1</sup> <sup>5</sup> <sup>12</sup> GitHub - jaklithn/GedcomParser: Parser of GEDCOM files

<https://github.com/jaklithn/GedcomParser>

<sup>2</sup> <sup>3</sup> <sup>4</sup> <sup>6</sup> <sup>7</sup> <sup>8</sup> <sup>13</sup> <sup>14</sup> <sup>15</sup> <sup>16</sup> <sup>17</sup> <sup>19</sup> <sup>20</sup> <sup>21</sup> <sup>22</sup> <sup>25</sup> <sup>26</sup> <sup>27</sup> <sup>28</sup> <sup>29</sup> <sup>30</sup> <sup>33</sup> GEDCOM 5.5.5.pdf

<file://file-Baym79ED7DerPX36VdJRkV>

<sup>9</sup> <sup>10</sup> <sup>18</sup> <sup>23</sup> <sup>24</sup> <sup>31</sup> <sup>32</sup> Discover AuraDB Free: Importing GEDCOM & Genealogy Data

<https://neo4j.com/blog/developer/discover-auradb-free-importing-gedcom-files-and-exploring-genealogy-ancestry-data-as-a-graph/>

<sup>11</sup> software recommendations - Writing GEDCOM Parser for Java 2024 - Genealogy & Family History Stack Exchange

<https://genealogy.stackexchange.com/questions/19317/writing-gedcom-parser-for-java-2024>