# Designing a Robust OCR-Driven Chain-of-Title Database

## Introduction and Problem Overview

Constructing a **chain of title** from OCR-extracted land records is a complex two-phase task: first, tracing ownership backward in time to a valid root (≥50 years old), then moving forward to gather all documents during each owner’s tenure. The key challenge is **imperfect OCR data** – scanned historical documents often contain recognition errors, ambiguities, and gaps. Our system must handle these uncertainties gracefully, never halting the chain due to OCR errors or missing links. This calls for a **specialized data architecture and indexing strategy** that supports fuzzy matching, temporal queries, confidence tracking, and iterative data improvement. The solution will combine elements of graph traversal and relational querying in a hybrid approach, leveraging the strengths of each for performance and accuracy. In the following sections, we compare graph vs relational databases for this use case, outline an OCR processing pipeline, design a schema with temporal ownership periods and OCR confidence, propose fuzzy indexing techniques, and describe algorithms to build the chain-of-title despite data imperfections. We also address common OCR error patterns (e.g. name variations, date ambiguities, illegible text) and how to flag or resolve them, and provide example queries and Python snippets for key functions. The goal is a **practical, scalable design** (capable of county-wide deployment with thousands of records) that ensures a complete chain-of-title can be built to 50+ years in the past with high confidence, while highlighting any uncertainties for human review.

**Key Objectives:**  
- **Reliability:** Never halt chain construction due to OCR errors; always attempt to continue via alternate paths or fuzzy matches.  
- **Accuracy & Confidence:** Use OCR confidence scores and logical checks to rank link validity and flag low-confidence elements[[1]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=If%20you%20need%20to%20obtain,fields%20required%20by%20the%20business)[[2]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=Why%20Confidence%20Scores%20Matter). Distinguish OCR uncertainty from true gaps in records.  
- **Comprehensiveness:** Capture *all* relevant documents per owner (deeds, mortgages, liens, releases, easements, etc.), and determine which encumbrances remain active vs satisfied.  
- **Performance at Scale:** Optimize for fast traversals and searches through potentially millions of records, using appropriate indexes and data structures (cloud-based Postgres preferred, with extensions or complementary search engines as needed).  
- **Auditability:** Maintain an audit trail from raw OCR text to cleaned data and final decisions, allowing inspection of original text and confidence levels. Support iterative improvements (e.g. re-OCR or manual correction on flagged items).  
- **Flexibility:** Handle a variety of document qualities (pristine digital PDFs to 1970s smudged microfilm) and content variations (handwritten notes, stamps, name changes, etc.) via extensible data models and processing rules.

Below we dive into the detailed research and recommendations addressing each aspect of this problem.

## Graph vs. Relational vs. Hybrid Database Approach

One fundamental decision is how to model the highly interrelated title chain data: a **graph database** or a traditional **relational database**, or a combination. Each has advantages for certain aspects:

* **Graph Databases:** Naturally model entities (people, properties) and relationships (transactions) as nodes and edges. Graphs excel at recursive traversals and multi-hop queries, maintaining near-constant performance even as relationship depth increases[[3]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=data)[[4]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient). In a chain-of-title, each deed links a seller (grantor) to a buyer (grantee); representing this as a graph edge allows efficient backward/forward navigation. Graph queries (e.g. Cypher in Neo4j) are optimized for *pattern matching* and exploring connections without expensive multi-table joins[[5]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=The%20relational%20model%20processes%20queries,other%20techniques%20for%20joining%20data)[[6]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=without%20needing%20to%20dynamically%20calculate,make%20graph%20databases%20very%20efficient). For example, finding a path from current owner back to a 50-year-old root is a multi-hop traversal where graph DBs shine – Neo4j has shown order-of-magnitude speedups (100x+) over SQL when querying deep relationships[[7][8]](https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database#:~:text=,). A graph can also easily store **relationship properties** like timestamps or confidence weights on edges (useful for OCR confidence or match strength). However, off-the-shelf graph databases lack built-in fuzzy text search on attributes – we would likely integrate a search index or apply approximate matching in the query logic.
* **Relational Databases (SQL):** Relational systems like PostgreSQL are robust, familiar, and excel at structured queries and transactions. A well-indexed relational schema can handle chain-of-title queries, but complexity grows with each join/recursive step[[9]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=Relational%20databases%20excel%20at%20operations,operation%20regardless%20of%20optimization%20strategies)[[10]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Alternatively%2C%20relational%20databases%20use%20index,to%20retrieve%20the%20required%20data). Traversing ownership history in SQL requires recursive joins or Common Table Expressions (CTEs). While SQL can manage a few joins well, performance degrades as relationships chain deeper or queries involve many conditions[[11]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=%2A%20Finding%20friends,recommendation%20paths%20through%20purchase%20history)[[12]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Performance%20Relational%20databases%20face%20complex,querying%20relationships%20between%20connected%20data). In practice, a 50+ year chain might involve, say, 5–15 ownership hops; SQL can handle this depth, especially if indexes are used for each step, but complex branching (ambiguous matches) could become cumbersome. On the plus side, PostgreSQL offers powerful extensions for text search and fuzzy matching (pg\_trgm, fuzzystrmatch) and supports advanced indexing and filtering (e.g. GIN/GiST indexes, full-text search) which we can leverage for OCR text. It also supports **window functions and rich SQL** for temporal queries (like finding all documents in a date range) and can ensure **ACID** consistency (important for not losing any transactions). Relational storage may also be more straightforward for ensuring data integrity (e.g. all documents have required fields, enforce certain formats).
* **Hybrid Approach:** For this project, a hybrid strategy is likely optimal. We recommend using **PostgreSQL as the primary datastore** (storing structured data, enforcing schema, and using its extensions for fuzzy text search), augmented by **graph techniques** either via an additional graph engine or by modeling graph-like relationships in SQL. A pragmatic approach is to store the data in Postgres (which the team is comfortable with) and use a graph querying library or GraphQL layer to traverse relationships as needed. Postgres itself can perform recursive queries (WITH RECURSIVE CTEs) to walk the ownership chain. Another hybrid idea is to use an **external search index** (Elasticsearch or Solr) for heavy fuzzy searching of OCR text, while the core relational DB handles structured links and transactions. This offloads the fuzzy text matching to a specialized engine and then feeds results (possible matches) back into the relational logic. Given the team’s familiarity with GraphQL, an option is to design a GraphQL API on top of Postgres that can expose nodes (owners, properties, documents) and connections, and internally use optimized SQL or call out to a search service for fuzzy name matching. This retains a unified interface while utilizing the best tool for each sub-problem.

**Recommendation:** Start with **PostgreSQL** as the foundation, utilizing its JSON capabilities (to store raw OCR text and metadata) and extensions (for full-text and trigram indexes). Design the schema in a way that naturally represents a graph of relationships: e.g., a table of Documents where each record has a grantor\_id and grantee\_id (linking to a Party table of owners/entities), effectively creating an adjacency list. This can be traversed recursively via SQL. If performance tests show SQL struggling with deep or fuzzy queries, consider introducing a graph database like Neo4j for the relationship-heavy operations. Neo4j could ingest the same data (it can be kept in sync via periodic updates), enabling lightning-fast traversals and pathfinding when needed. As AWS notes, **graph databases outperform relational DBs for multi-hop queries** and complex relationship patterns[[13]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient)[[12]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Performance%20Relational%20databases%20face%20complex,querying%20relationships%20between%20connected%20data). However, given the moderate depth of title chains and the need for tight integration with fuzzy text logic, Postgres alone (with careful indexing) may suffice for thousands of records. We will proceed with a design centered on Postgres, noting where graph techniques or hybrid additions could enhance it.

## OCR Data Processing and Quality Management

The reliability of our chain-of-title depends heavily on how we process and manage OCR data from scanned documents. OCR outputs can be noisy: names misspelled, dates misread, text layout lost. We must implement a pipeline that not only extracts text, but also **captures confidence scores, identifies potential errors, and normalizes the data** for indexing – all while retaining the original raw text for auditability.

**1. OCR Pipeline Overview:**  
We will use Google Document AI for OCR (as currently chosen), which provides structured outputs and per-field confidence. The pipeline steps:

* **Image Preprocessing:** Before OCR, apply image enhancements to improve accuracy. This includes de-skewing crooked scans, increasing contrast, removing noise, and perhaps binarization. For older documents (e.g. blurry 1970s microfilm or handwritten notes), specialized preprocessing like adaptive thresholding or even AI-based enhancement (e.g. upscaling or handwriting-specific models) can improve OCR results. *Example:* For a faint marginal notation (satisfaction stamp), image filters might make the text more legible to OCR.
* **OCR Extraction:** Run the images through Document AI (or alternative OCR engines if needed for comparison). Collect not just the recognized text, but **structured fields** if the OCR model provides them (e.g. it might detect “Grantor: John Doe”, “Grantee: Jane Smith”, “Date: Jan 3, 2007” etc.). Importantly, capture the **confidence scores** for each word/field that the OCR engine provides. Modern OCR APIs typically give a confidence (0–100 or 0–1.0) per text region. We will store these for later use in the database.
* **Multi-Engine & Voting (Optional):** If resources permit, using two different OCR engines and comparing results can boost accuracy. For example, run Google Doc AI and another OCR (like Tesseract or ABBYY FineReader) on the same document. If one engine has low confidence or the outputs differ significantly, flag for review or combine results (e.g. choose the text with higher overall confidence or even merge by taking high-confidence words from each). Research shows that no single OCR guarantees correctness solely from confidence scores[[14]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=confidence%20scores%20are%20used%20to,that%20the%20answer%20is%20correct), so cross-checking can help.
* **Parsing & Field Extraction:** Take the raw OCR text (often a blob of text) and **parse it into structured data**: parties, dates, legal description, document type, references to other documents (book/page numbers), etc. This might involve regular expressions and heuristics, or an NLP approach:
* Use known document patterns: deeds often contain keywords like “grantee”/“grantor”, “dated”, “recorded on”, etc. A simple rules engine can locate these and extract the following text. For example, find “grantee:” in text and capture the name after it. If formal headers are absent, use position (e.g. first names in the document might be parties).
* **Natural Language Processing (NLP):** We can employ an NLP model or custom entity recognizer to detect person names, organization names, dates, addresses within the OCR text. Google’s Document AI might already classify fields; if not, a library like spaCy can be trained to identify names and dates in deeds.
* The output of this step is a **structured record** for each document: e.g. {DocumentID, GrantorName="Robeat Jolinson", GranteeName="Bobby Johnson Jr.", Date="Jan 3?, 200?", DocType="Warranty Deed", PropertyDesc="…", RelatedDocRef="Book 123 Pg 45", RawText=..., FieldConfidences={Grantor: 0.6, Grantee: 0.5, Date: 0.7, ...}}. We include the raw text or a reference to it (for later display or re-parsing if needed).
* **Data Cleaning & Normalization:** After extraction, apply cleaning rules to **normalize values**:
* **Names:** Normalize case (“JOHNSON, ROBERT” -> “Robert Johnson”), expand common abbreviations (“Co.” -> “Company”), and remove extraneous punctuation. For personal names, consider splitting into first, middle, last for consistency. Also, handle known OCR letter confusions (this can be context-aware): e.g. if a last name “Snuth” was probably “Smith” (common error of “m” vs “rn”), or “Jolin” might be “John” (OCR misreading ‘h’ as ‘li’). We might maintain a list of common proper nouns in the locality (like frequent last names, or corporate entities in the county) to correct to if OCR output is close. **Store multiple variants** if unsure: e.g., if “Robeat Jolinson” came with low confidence, we can store an alternate “Robert Johnson” guess (perhaps from a fuzzy match against a name dictionary) along with a lower confidence for that guess. This way, the chain-building algorithm can consider both.
* **Dates:** Convert OCR’d date strings to a standard DATE format if possible. If the OCR output is partial (e.g. “Jan 3?, 200?”), use placeholders or ranges. For example, “Jan 3?, 200?” we might interpret as between Jan 30, 2000 and Jan 39, 2009 (clearly invalid dates, so instead perhaps Jan 30, 2000 to Jan 31, 2009). More realistically, if a day or year digit is unclear, we can store a range: maybe “Jan 3x, 200y” becomes a date range covering all possibilities (Jan 30–39 of 2000–2009) or simply flag it as uncertain date around January 200?. We also use context: recording books are chronological, so if this document was in Book 1500 and the previous page had a known date Jan 30, 2007, likely this one is Jan 31, 2007. The system could infer missing date parts from neighboring records. Regardless, store what was read and possibly an estimated actual date with a confidence.
* **Document Type:** OCR might misread titles or form names (“Warranty Deed” could come out as “Warrany Deed”). We can have a controlled vocabulary of document types and match/correct OCR output to the nearest valid type (e.g. using fuzzy match to known types list: deed, mortgage, release, lien, etc.). If confidence is low, mark the doc type as “unknown” or “needs verification” rather than misclassify.
* **Legal Description:** These are tricky (many numbers, letters, and symbols for degrees, minutes, etc.). Normalize common units (convert all "O" to "0" if appearing in a numeric context, as in **N 89°15'30"** vs OCR “N 89°l5'3O” where we detect that in a metes-and-bounds pattern, “l” should be “1” and “O” should be “0”[[15]](https://nexval.com/15-technology-components-you-need-for-title-automation/#:~:text=nexval,massive%20public%20and%20third)). Similarly, ensure consistently formatted output (like add missing degree symbols, etc.). Each correction should only be done if confidence is low and pattern strongly suggests it – we don’t want to accidentally alter valid data. It might be best to store the **raw legal description text alongside a cleaned version** so that any automated corrections can be double-checked.
* **References (Book/Page, etc.):** If OCR captured a reference number (like “Book 5123 Page 487”), verify its format and existence. E.g., if it read “Book S123 Page 487” (where ‘S’ was misread instead of ‘5’), that reference won’t match anything. A strategy is to attempt to look up the referenced document in our database; if not found, try common substitutions (S->5, O->0, I->1, etc.) to find a match. Store both the read value and any matched actual document ID, with a match confidence. This will help us later link releases to the original instrument.
* **Confidence Scoring & Thresholds:** For each field and document, compute an overall quality score. The OCR engine’s per-character or per-word confidences can be aggregated: e.g., average confidence of all characters in the name “Robeat Jolinson” might be 85%, but critical characters like “b” vs “v” could be wrong. It’s known that a page-level 98% accuracy can translate to far lower field accuracy if the few errors fall in key fields[[1]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=If%20you%20need%20to%20obtain,fields%20required%20by%20the%20business). We will therefore rely on field-level confidence. If a field’s confidence is below a threshold (say <70 on a 0–100 scale), flag that field as “suspect”. These flags can later trigger manual review or additional processing (like re-check against a dictionary). We also establish a **confidence threshold for automation**: e.g., if grantor name confidence ≥90, trust it; if 50–89, use it but mark it fuzzy; if <50, treat the link as very unreliable and allow alternate possibilities in chain linking. The TDWI guide recommends determining thresholds empirically by analyzing errors vs scores[[2]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=Why%20Confidence%20Scores%20Matter)[[14]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=confidence%20scores%20are%20used%20to,that%20the%20answer%20is%20correct) – we can adjust these as we get data.
* **Storing Raw vs Clean Data:** It is crucial to **store the raw OCR text** (or at least the raw field values) alongside the cleaned/normalized version in the database. This maintains an audit trail and allows humans or future algorithms to see exactly what OCR saw. For example, in a PartyName field we might have raw\_name = "Robeat Jolinson" and normalized\_name = "Robert Johnson" with a name\_confidence = 0.60. The system might use the normalized name for matching and indexing but always be able to display the raw OCR text (perhaps with a strikethrough or highlight in the UI if it was corrected). This also helps when OCR tech improves – we could re-run OCR and compare results to the stored raw text.
* **Multiple Interpretations:** When OCR confidence is very low or ambiguous, our data model can store **multiple interpretations** for a field. For instance, if the OCR engine was unsure of a handwritten name, it might output top 2 candidates (some OCR APIs do provide alternate text suggestions). If not, we can generate them using our own algorithms (e.g. all close matches from a name dictionary). Store these alternatives with their scores. E.g., for a smudged signature, store alt\_name\_1 = "Jonathan Smith" (score 0.4), alt\_name\_2 = "John A. Smith" (score 0.3). The chain-building will consider these alternates when trying to match subsequent records. Essentially, we treat uncertain fields as *nodes with multiple possible values* until disambiguated.

**2. OCR Quality Metadata:**  
We maintain metadata about each document’s OCR process: - Scanning info: DPI, whether it was from microfilm, any image enhancement applied. - OCR engine version and settings used, and date processed. - Overall OCR confidence distribution (e.g. what percent of characters were above 90% confidence, etc.) – this can help spot truly poor scans. - Flags like “handwritten sections present” or “seal or stamp detected” (non-text elements that might need attention). - If manual corrections were made, record who/when and what was changed (for audit and to avoid overwriting human fixes later).

This metadata can be stored in a JSONB column in Postgres (since it’s semi-structured and varies per doc). It provides context if we later need to reprocess (e.g. “Document X had very low confidence because it was cursive – maybe send for manual transcription”).

**3. Post-OCR Validation Rules:**  
Implement rules to catch obviously implausible OCR results: - Names with non-alphabetic characters or unlikely sequences (e.g. “J0hn D##” clearly bad OCR). - Dates that are out of range (e.g. a deed dated 2107 instead of 2017). - Book/page references that don’t conform to known numbering schemes. - Legal descriptions that fail basic checks (e.g. missing the words “Section” or “Lot” if those are expected for that locale; or impossible bearings like degrees > 360). Any such anomalies can be auto-flagged. Some could be auto-corrected (like a clearly wrong century in a date might be a single digit error), but generally flag for human review if high impact.

**4. Iterative Improvement:**  
The OCR pipeline should allow re-running on demand (for example, if a user flags a document as unreadable, we might attempt a different OCR engine or settings). Also, as new documents come in or as manual fixes accumulate, we can **expand our dictionaries** for names and other fields to catch recurring OCR mistakes. For instance, if we notice “Johnson” is often read as “Jolinson” for a certain typed font, we add that pattern to auto-corrections. Over time, this feedback loop will improve the automated accuracy. All manual interventions are logged so we can measure how many documents required help and possibly retrain OCR or parsing on those cases.

By carefully processing OCR data and capturing quality metrics, we ensure that the data fed into our indexing and chain algorithms is as clean as possible, while still acknowledging uncertainty through confidence scores and alternate values. This builds a solid foundation for the indexing strategy next.

## Bidirectional Fuzzy Indexing Strategy

To traverse ownership history backward and forward efficiently, we need **fast lookups by party name and date**, but with tolerance for OCR errors and name variations. Traditional title plants use separate **Grantor and Grantee indexes** – essentially, lists of transactions ordered by party name and date[[16]](https://www.law.cornell.edu/wex/grantor-grantee_index#:~:text=Institute%20www.law.cornell.edu%20%20Grantor,the%20transferring%20of%20property%20ownership). We will replicate this concept in our database, but enhanced with **fuzzy matching capabilities** to handle misspellings from OCR.

**Key Indexing Requirements:** - Quickly find all documents where a given person or entity is the **grantor** (for backward tracing to find who sold to current owner’s predecessor) or the **grantee** (for forward tracing or for backward link when matching a previous sale). - Support looking up a name even if it’s spelled differently (or incorrectly) in some records – e.g., a search for “Robert Johnson” should retrieve docs indexed under “Robeat Jolinson” or “R. Johnson Jr.” if those are likely the same. - Allow queries by date range on a property or owner’s tenure. For example, once we establish John Doe owned the property from 1990 to 2000, we need to find all documents *between 1990–2000* involving John Doe (or the property) to list encumbrances during that period. - Support composite searches like “all mortgages on this property from 1990–2000 that are unreleased”. - Efficiently handle **partial matches** when some fields are uncertain (e.g. if we only know part of a name or a date is fuzzy, the index should still help narrow candidates).

**1. Name Indexes with Phonetic & Fuzzy Matching:**  
We propose maintaining a **Name index** table keyed by normalized party name (perhaps last name, first name, etc.). Each document will insert two entries: one for the grantor name, one for the grantee name (pointing to the document ID, role, and date). To handle OCR errors: - **Phonetic Codes:** Compute a Soundex or Metaphone code for each name and index that as well. Soundex is an old but quick algorithm that maps similar-sounding names to the same code (e.g. “Smith” and “Smyth” share a code)[[17]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=Soundex%20works%20well%20with%20western,It%27s%20intended%20for%20phonetic%20comparison). Metaphone (particularly Double Metaphone) is more modern and handles a broader range of name variations[[18]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=I%20would%20suggest%20using%20Metaphone%2C,and%20spelling%20phonetically). These help catch cases where OCR might substitute similar-sounding letters or when nicknames are used. For example, “Robert” vs “Robbert” or “Johnson” vs “Jonson” might share a phonetic code. We create an index on these codes so that a search by name can retrieve candidates that *sound* like the query. - **Trigram/Levenshtein Index:** For direct OCR spelling differences that aren’t just phonetic (like “Jolinson” vs “Johnson”), we use **trigram similarity** indexing. PostgreSQL’s pg\_trgm extension tokenizes strings into three-letter fragments and can efficiently find similar strings by measuring overlap[[19]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=PostgreSQL%20Fuzzy%20Text%20Searching). We will create a GIN index on the party name column with the pg\_trgm operator, enabling queries like WHERE name % 'Robert Johnson' to find close matches within a threshold distance. This fuzzy operator % uses trigram similarity under the hood and is very effective for OCR errors or typos[[20]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=This%20is%20where%20fuzzy%20text,database%20something%20like%20the%20above)[[21]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=PostgreSQL%20Fuzzy%20Text%20Searching). We can set a similarity threshold (like 0.3 or 0.4 out of 1) to broaden or narrow matches as needed. For example, a search for “Jolinson” might find “Johnson” with high similarity because they share many trigrams (assuming the OCR error is minor). - **Combined Approach:** In practice, using both phonetic and trigram methods together yields best results[[22]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=Levenshtein%20distance%20looks%20at%20two,for%20missing%20or%20substituted%20letters)[[18]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=I%20would%20suggest%20using%20Metaphone%2C,and%20spelling%20phonetically). Our strategy is: - Standardize the query name (remove punctuation, case, maybe first name abbreviations vs full). - First query the Name index for exact and phonetic matches (cheap), then use a trigram similarity search for names above a certain similarity. This can be done via SQL or using a dedicated search engine if needed. - Rank the results by a composite score: e.g., a result that exactly matches phonetic code and has high trigram similarity gets top rank, whereas one that only phonetic matches but low trigram might be lower. - Only consider those above a certain score as likely matches to follow in the chain.

**Index structure:** In Postgres, one could create a functional index on soundex(name) or metaphone(name,4) (requires fuzzystrmatch extension) and another on the name text using pg\_trgm. Alternatively, maintain a separate table PartyNameIndex(name, name\_phonetic, doc\_id, role, date) with indexes on name\_phonetic and on name using trigram. The separate table makes it easy to query by name without scanning the main documents table.

Additionally, consider an **n-gram index** approach: storing character-level n-grams of names can help catch specific OCR substitutions (like “rn” vs “m”). However, pg\_trgm essentially does that internally by indexing trigrams, which covers common OCR errors of a few characters difference.

* **Handling Name Variants and Aliases:** Sometimes the same person appears under different names intentionally (e.g. “Robert Johnson Jr.” vs “Bobby Johnson” vs “Bob Johnson” could be the same individual, or a trust name vs individual name). OCR might also read “Jr.” as “Jr,” or miss it. We can’t fully resolve aliases automatically, but we can:
* Treat known suffixes (“Jr”, “Sr”, “III”) as separate tokens so that mismatch there doesn’t prevent a match (e.g. match ignoring suffixes first).
* Possibly maintain an **alias table** if during review we confirm two names are same entity (like link “Bobby” to “Robert”). Initially, this may not be populated, but over time user input or external data could add alias mappings, which our search can consult (i.e. if searching for Robert Johnson, also search for Bob Johnson).
* Use partial matching: if the OCR drops a first name initial or misreads it, we might match on last name + property context. For example, if the property legal description is the same and the last name matches, two names might refer to same owner (husband/wife scenarios aside). However, that’s more advanced – primary approach is focusing on fuzzy text of names themselves.

**2. Property/Legal Description Index (Tract Index):**  
While names are the main linkage, property identity is another way to chain title. Ideally, each document refers to the same parcel identifier (like a lot number or tax parcel ID). In older records, this can be inconsistent. If a stable **Property ID** exists or can be derived (say, from legal description normalization), we should index by that too: - Create a Property table with a unique ID (if the county has parcel numbers or if we assign our own). Link documents to Property IDs when possible. - If no unique ID is given, we could derive a hash of the legal description text and use that to group documents that likely describe the same parcel (though careful: legal descriptions change format over time or if land is split). - Maintain an index to find all docs for a property. This greatly helps the forward phase: once you know which property you’re dealing with, pulling all records by property ID in a date range is straightforward. It also can catch cases where the owner name changed spelling but the property is the same, thus likely the chain continues.

**Note:** In practice, chain-of-title search often relies on a *Tract Index* (index by property) if available[[23]](https://www.gsccca.org/docs/deed-lien-plat-documents/gsccca_indexing_standards_01_01_2018.pdf?sfvrsn=a850151f_2#:~:text=,Clerk), since that avoids name ambiguity. If the data allows (e.g. geo-coordinates or lot numbers can be parsed), incorporating this would be highly beneficial. Given the scope here, we’ll assume names are primary, but we should not ignore property indexing if feasible. We might at least normalize addresses (if mentioned) or lot/block identifiers and index those.

**3. Temporal Indexes:**  
Time is critical: We often query “find all documents in X date range for Y owner or property”. To optimize: - **Recording Date Index:** Index documents by their recording date (or effective date). A simple b-tree index on the date field allows range scans (e.g. WHERE date BETWEEN 1990 and 2000). This is useful when listing encumbrances during an ownership period. - **Ownership Period Query:** If we model ownership periods as separate records (more on data model later), we could index those by their period (start\_date, end\_date). Postgres supports **range types** (e.g. daterange) with GiST indexes to efficiently query overlapping ranges. So if we want to find which owner had the property on a certain date, or conversely, given an owner’s period, find docs overlapping that period, these indexes help. For example, to get all mortgages active during John’s ownership, we would find all mortgage documents with dates within John’s start–end range. A normal index on date combined with conditions works, but a range index could handle open-ended or uncertain ranges elegantly. - Considering OCR date uncertainty: If a date is fuzzy (e.g. uncertain day), we might store it as a small range (e.g. Jan 30–31, 2007 if unsure between 30th or 31st). Then using a range index, a query for documents in February 2007 would still correctly include or exclude that doc depending on overlap. In our implementation, we could store both a date\_text (raw OCR string) and a date\_start and date\_end (inferred earliest and latest possible date for that doc). Usually these will be the same exact date, but if uncertainty exists, date\_start/end capture it. Index on date\_start for sorting and range filtering. - **Composite Indexes:** Some queries might use multiple criteria, e.g., “find all releases for mortgages that are still unsatisfied.” That might involve joining a mortgage and release table on some ID. For speed, we might pre-index by status or by related doc ID. Another example: a composite index on (owner\_name, date) could speed up finding the next link in backward chain (since we often query by exact grantee name and find the earliest deed where they were grantee). But given we will mostly use name->doc or property->doc lookups, simple indexes on name and date suffice and then filter in memory for confidence etc.

**4. Fuzzy Matching in Queries:**  
We need to integrate the fuzzy logic into actual queries. Two approaches: - **SQL with extensions:** Using Postgres’s fuzzy search features as mentioned. For instance, to get possible predecessors of owner “Robert Johnson”, we could run:

SELECT doc\_id, grantor\_name, date, similarity(grantor\_name, 'Robert Johnson') AS sim  
FROM Documents   
WHERE grantor\_name %% 'Robert Johnson' -- uses pg\_trgm index for similarity search  
 AND date < :some\_cutoff\_date  
ORDER BY sim DESC;

The %% operator (or % depending on version) finds names with trigram similarity above a threshold[[24]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Performance%20Considerations). We could also incorporate Soundex: e.g. AND soundex(grantor\_name) = soundex('Robert Johnson') to ensure phonetic match as a coarse filter (but Soundex might be too broad on its own). Postgres can combine indexes if we have one on trigram and one on soundex (though function indexes usage might vary). Alternatively, do two queries: one exact phonetic match, one similar text, then union results. Another tool is the levenshtein() function (from fuzzystrmatch), which can directly compute edit distance between strings. But that won’t use an index unless we apply it after initial filtering (like find all names where first letter matches, then compute Levenshtein distance). We should prefer trigram indexing for performance, as it can use an index to shortlist candidates by similarity. - **External Search Engine:** For very large datasets or more complex fuzzy logic, an external search engine (Elasticsearch, Solr, or Manticore Search) can index the names and provide fuzzy query (with n-gram or Levenshtein automata). For example, Elastic can do a “fuzzy” query on a field with a specified edit distance. This might simplify handling multi-word names and scoring. The trade-off is adding another component and syncing data to it. Given a county-scale system (potentially millions of records), using an external search might be justified if performance of DB fuzzy search is insufficient[[24]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Performance%20Considerations). However, as one dev notes, Postgres pg\_trgm is perfectly suitable for small to mid-sized data and avoids maintaining another system[[25]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Using%20the%20built%20in%20,Solr%2C%20Elastic%20Search). We can start with Postgres fuzzy search and monitor performance.

**5. Handling OCR-Specific Errors in Matching:**  
We plan to use **query expansion and custom similarity scoring** to address systematic OCR errors: - **Common OCR Substitutions:** Expand a search query to multiple variants that cover typical OCR confusions. For example, if looking for “Alexander”, also search for “AIexander” (where I is misread as l), or for “Alexandr” (missing e). We can programmatically generate these variants using known confusion pairs (0/O, 1/l/I, 2/Z, 5/S, 8/B, c/e, rn/m, vv/w, etc.). A simple way is to create regex patterns. E.g., a pattern for “Johnson” could be J(ohnson|olinson) to allow that specific error seen (“hn” vs “li”). This might be too specific; instead, at runtime, we could take the query string and substitute confusable characters in various combinations and search for those as well. Since this can explode combinatorially, we limit to most likely one-off errors. Another trick: use **lowered similarity threshold** to inherently allow those differences (the trigram similarity will catch many of them without explicit expansion). - **Ranking by Confidence:** When retrieving matches, incorporate not just textual similarity, but the OCR confidence of the stored name. For example, if “Robeat Jolinson” came from OCR with 60% confidence, and “Robert Johnson” from another doc is 95% confident, and both show up as matches to query “Robert Johnson”, we might rank the high-confidence exact match higher. In practice, our chain-building algorithm (next section) will handle confidence weighting of edges, but even in the query UI or initial search, we could sort by something like score = text\_similarity \* (0.5 + 0.5 \* (field\_confidence)) – so higher OCR certainty boosts the result. - **Aggregating multiple clues:** Sometimes one low-confidence match alone is iffy, but if *several* documents independently point to the same name, it reinforces the likelihood. For example, suppose “Robeat Jolinson” appears as grantee in one deed, and “Robt Johnson” (OCR dropped some letters) appears in a mortgage, both around the same year. Individually, each is fuzzy match to “Robert Johnson”. If our search can group by actual person, we’d see multiple hits. Without prior knowledge they’re the same person, we might miss that. One approach: retrieve all fuzzy matches for “Johnson” last name, then see if any cluster around similar first names or addresses. This might be too advanced for the index itself; rather, the chain algorithm will naturally encounter that if multiple docs link to the same subsequent owner. - **Edit distance tuning:** In matching algorithms like Levenshtein, we can give lower cost to likely OCR errors. For instance, treat ('rn' ↔ 'm') substitution as cost 0.5 instead of 1, treat digit-letter confusions as smaller penalty, etc. There’s research on OCR-specific edit distance weighting. Implementing this in SQL is tough; better done in application logic or perhaps using a fuzzy library in Python. We can retrieve a pool of candidate names via index, then use a custom Python function to compute a more nuanced similarity score for final selection.

**6. Additional Indexes for Encumbrance Tracking:**  
To identify active vs released encumbrances, we likely have tables for Mortgages and Releases (or a generic Document table with type). We should index the linking field that connects them. For example, if a release references a mortgage by book/page or document number, have an index on that reference in the mortgage table so we can quickly find the mortgage record given the reference. We will also index the **status** of an encumbrance (active/satisfied). Possibly maintain an “active liens” index per property or owner, updated when a lien is released. This can make querying “which liens remain active as of now on this chain” very fast (just check the flag instead of computing difference each time). However, maintaining that flag reliably through OCR uncertainty is tricky – it might require human confirmation that a particular release matches a mortgage. So, in initial implementation, we might instead do it on the fly: for each mortgage in chain, search for a matching release. Good indexing on references and names will assist here.

In summary, our indexing strategy in PostgreSQL involves: - **Name Index with Fuzzy capabilities:** using trigram and phonetic indexes to allow error-tolerant name lookup. - **Date Index (and possibly range index):** for chronological queries. - **Property/Tract Index:** if possible, to group docs by property. - **Composite indexes** where appropriate for frequent multi-column queries (e.g., maybe (grantee\_name, recording\_date) for backward search of previous owner quickly by name sorted by date).

All indexes will be designed to account for the large volume (county-wide data). Postgres can handle large indexes, but we’ll monitor performance. We expect the **trigram index** to be the heaviest but still efficient for thousands to millions of rows, as it turns text into sets of trigrams and uses GIN to query – many production systems use it for fuzzy search on big tables[[25]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Using%20the%20built%20in%20,Solr%2C%20Elastic%20Search). Should performance degrade, we will consider external search or caching of frequent searches.

## Data Model for Temporal Ownership with Uncertainty

A well-designed schema is central to representing the chain-of-title, especially to accommodate **temporal ownership periods**, OCR-derived data with confidences, and ambiguous relationships. We outline a schema that separates core entities (Party, Property, Document) and introduces join tables or link tables to model ownership periods and uncertain connections.

**Key Entities:**

* **Party (Owner) Table:** Represents an entity that can own property. This could be a person, a couple, a company, or a trust. Fields:
* party\_id (PK)
* name\_normalized (e.g. "Robert Johnson Jr.")
* name\_raw (e.g. "Robeat Jolinson")
* name\_confidence (numeric score)
* alt\_names (maybe a JSON array of alternate spellings or aliases seen, with confidences)
* Possibly type (individual, corporation, etc.) if determinable (e.g. if name contains “Inc” or “LLC”, categorize as organization).

We will populate Party from documents’ grantor/grantee names. Initially, each distinct name string might get its own Party entry (since we don’t know if two similar names are same person). Over time, if we identify that two entries are actually the same entity (through manual merge or algorithmic match), we can merge them or link them via an alias relationship. The Party table essentially allows grouping multiple docs under a presumed same owner identity. It’s crucial for phase 2 (forward traversal), where an owner might have multiple docs in their period.

* **Property Table:** Represents a property (parcel/tract). Fields:
* property\_id (PK)
* description\_normalized (some representative legal description or parcel ID if available)
* description\_raw (the OCR text of legal description from perhaps the root deed or current deed)
* We might also store location info if available (subdivision, lot, etc., to correlate with external data).
* One property can have many documents. If multiple descriptions appear (due to minor variations in wording), they all map to the same property\_id ideally.

In practice, establishing a unique property\_id is hard purely from OCR text; it might require manual seeding (like using a GIS or tax database). But we can create property groups by linking documents that have high similarity in legal description text. For now, let’s assume a property can be identified (maybe via the starting point of search or known parcel identifier).

* **Document Table:** Each recorded document (deed, mortgage, lien, release, etc.). Fields:
* doc\_id (PK, could be an internal ID; also store Book/Page or Instrument Number as available)
* type (enumeration or code: deed, mortgage, release, etc. Possibly a subtype like Warranty Deed, Quitclaim Deed, etc.)
* recording\_date (date when recorded, as parsed. Use date\_start and date\_end if uncertain.)
* effective\_date (if different, often same as recording date for our purposes, but some documents use execution date vs recording date.)
* grantor\_id (FK to Party who is transferring interest)
* grantee\_id (FK to Party receiving interest)
* property\_id (FK to Property that the document pertains to)
* ocr\_confidence (overall doc OCR quality score, e.g. average or lowest field confidence)
* raw\_text (the full OCR text, stored maybe in a separate table or a text column with TOAST compression, because it could be large; or store a reference to the file)
* parsed\_fields (JSON storing field-level raw vs cleaned values and confidences, e.g. {"grantor\_name\_raw": "...", "grantor\_conf": 0.72, "book": 123, "page":456, "book\_conf":0.9, ...})
* Additional fields for certain doc types: e.g., related\_doc\_id (FK to another document if this one references a prior instrument – used by releases referencing a mortgage; this can be populated by linking logic when possible).
* active (Boolean, for encumbrances like mortgages: true if not released, false if released. This could be updated when a release doc is linked.)

The Document table is central. Note that grantor\_id and grantee\_id link to Party – these represent *our interpretation* of who the parties are. If OCR was uncertain, we may initially create Party entries that are semi-duplicate. It might be worth linking Document to Party with a confidence as well. Alternatively, have a join table DocumentPartyLink with doc\_id, party\_id, role (Grantor/Grantee), confidence. This way, if a document’s party is uncertain between two names, we could link both with different confidence scores. However, that complicates queries. A simpler approach: assign the most likely party, but mark in the document’s parsed\_fields any alternate names considered. If later we find it was wrong, we can adjust.

* **Ownership (Chain) Table:** This table explicitly represents an ownership period (from one transaction to the next):
* owner\_id (FK to Party)
* property\_id (FK to Property)
* ownership\_start (date – typically the recording date of the deed where this owner acquired title)
* ownership\_end (date – the date of transfer out, or null if current owner; could use an approximate date if OCR uncertain, or an official root-of-title cutoff if going back 50+ years)
* source\_deed\_id (the document by which this owner acquired title)
* transfer\_out\_deed\_id (the document by which they conveyed to next owner, null if current or if unknown/off-record)
* Possibly a confidence or status flag if this link is inferred vs confirmed.

This Ownership table is essentially a *calculated* result of the chain-building, rather than directly entered from OCR. We can construct it once we have the chain, or dynamically derive it on the fly. But having it stored is useful for quick queries: it becomes a ledger of owners for each property with date ranges. It also helps detect gaps (if ownership\_end of one owner is in 1990 but next ownership\_start is 1992, we have a 2-year gap to investigate). Each property should ideally have continuous ownership periods from the root to current with no breaks.

We will generate these records after doing the backward phase, then verify continuity and then use them in forward phase queries.

* **AmbiguousLink/Unresolved Table:** This is optional, but could store instances where a link was *ambiguous* and we didn’t auto-resolve:
* e.g., if a deed’s grantee didn’t clearly match any prior owner’s name, or there were two possible prior deeds for a given owner name, etc. This table could list property, the problematic step, and why (e.g. “Grantor name OCR unclear, could not find matching grantee in previous 20 years”). These records prompt manual review.
* Similarly, any “orphaned document” (one that mentions the property or parties but couldn’t be inserted neatly into the chain) can be listed here.
* **OCR\_Revision Table:** To maintain audit trail for data corrections, have a table to log any manual or automated corrections after initial OCR:
* Fields: doc\_id, field, old\_value, new\_value, correction\_type (manual/auto), timestamp, user\_or\_process.
* This way, if someone fixes "Jolinson" to "Johnson", it’s recorded. The system can periodically apply these corrections to improve the confidence (perhaps even feeding back into OCR training if applicable).

**Fuzzy/Probabilistic Connections Modeling:**  
In the schema above, a Document has fixed grantor\_id and grantee\_id. How do we represent uncertainty? A few options: - Use a **probabilistic approach** in which a document can have multiple possible grantor parties. For example, we could initially link “Robeat Jolinson” document to Party A (Robert Johnson) with confidence 0.6, but also link it to Party B (some other similar name) with confidence 0.4 if applicable. This requires a join table as mentioned (Document–Party many-to-many with confidence). This is more like a graph where from one node (document) you have weighted edges to multiple party nodes. - Alternatively, keep one link but mark low confidence, and rely on the chain algorithm to handle it by exploring alternate search paths. For instance, if we’re going backward from current owner and hit a deed where grantor name was low confidence, the algorithm can search for any similar name in earlier grantee index. We might not need to explicitly store multiple grantors on the doc, we can just search on the raw name as well if needed. - As a simpler compromise, store in Document parsed\_fields something like grantor\_name\_alt and grantee\_name\_alt for alternate interpretations. This keeps schema simpler while preserving alternates.

**Handling Multiple Owners / Joint Ownership:**  
Often deeds have multiple grantees (e.g. husband and wife, or multiple grantors if an estate). We should support that: - Possibly a join table DocumentParty(role) for multiple parties per role. But for chain-of-title, usually if two people jointly take title, they also jointly convey it later. We can treat them as a combined “Party” for chain purposes or handle each but realize the property is under both names. A practical approach: for joint owners, create a Party entry that concatenates the names (“John & Mary Smith”) as one unit for chain tracing. However, that might complicate searching later docs if only one of them signs something (like a mortgage could be in both or one name). - Perhaps better: allow Ownership records to have a set of party\_ids for co-owners. This complicates queries. Another approach: just record each deed as separate entries for each grantee party in the index, but link them as co-owners by property. For forward encumbrances, we’d find docs under either name. - For simplicity, assume one primary name for chain (maybe choose one or handle one at a time, understanding that a chain could split if co-owners split ownership). Given complexity, we mention it but may not fully solve it here – as a design, at least not block multiple parties.

**Confidence in Schema:**  
We integrate confidence at several levels: - Party: name\_confidence indicates how sure we are of the spelling/identity for that party entry. - Document: ocr\_confidence for general quality, and in parsed\_fields each critical field has a confidence. - Ownership: could have a confidence if, say, the link was inferred (like if we couldn’t find a deed out but we assume an off-record transfer – that would be low confidence or “unverified”). - Relationship edges (if we implemented them explicitly) would have weights. Instead, we incorporate this in the algorithms rather than storing weight on a separate edge table.

**Storage of OCR text and variations:**  
We will **store raw OCR text** likely outside the main Document table for efficiency. Perhaps a DocumentText table with doc\_id and the full text (to not clog main table). Alternatively, compress in a bytea or text column. Because we need quick search in them sometimes (for e.g. full-text search maybe on legal description keywords), consider adding a full-text index on it (Postgres to\_tsvector). This could allow searching the OCR text for clues (like searching for the property address or an unusual name mention). However, this is secondary to our main approach; primary indexes cover structured fields.

**Example of Schema Usage:**  
Take the scenario: - OCR of a 1990 deed yields grantee\_name\_raw="Robeat Jolinson", grantee\_conf=0.6 -> We create Party P1 with name "Robeat Jolinson" (normalized to "Robert Johnson") and conf 0.6. - That deed’s grantor was "Alice Smith" (confidently read) -> Party P2 "Alice Smith". - Document D1 (deed) has grantor\_id=P2, grantee\_id=P1, date 1990. - Now, when tracing back from Robert Johnson (P1), we look at grantor Alice Smith (P2) in D1. The backward chain algorithm will see P2 as previous owner. - For forward, P1’s ownership period 1990 to whatever. We then find that in 2005, a deed shows grantor "Robert Johnson Jr." selling to someone else. OCR read it as "Robt Johnson Jr." with high confidence 0.9, which we normalized to "Robert Johnson Jr." and realized it likely corresponds to P1 (the same person, maybe plus Jr). The system could either match that to existing Party P1 via fuzzy logic or create a new Party P3 if it didn’t realize. Ideally, a fuzzy match links it to P1 (maybe updating P1’s name to include Jr variant). Document D2 (2005 deed) has grantor\_id=P1, grantee\_id=NewOwner, date 2005. - Now P1’s ownership\_end becomes 2005, linking D2 as transfer\_out. - P1 in parsed\_fields might have alt\_name "Robt Johnson Jr." recognized with conf 0.9, which now actually increases our confidence that P1’s true name is Robert Johnson Jr. We might update P1’s name\_normalized to that, or note alias. - Meanwhile, any mortgages in P1’s period (1990–2005) – say a 1995 mortgage listing "Robert Johnston" (typo OCR) – we would create a Party maybe P4 if unmatched, but ideally fuzzy match to P1 as well. If not, chain algorithm might pick it up by property link or partial name match.

This illustrates that some entity resolution might occur on the fly (merging parties). The data model should not prevent multiple entries that later unify; it should allow updating links when we determine two Party entries are same (which could be done by merging IDs and updating foreign keys, or by linking them with an alias table and adjusting queries to treat them as one set).

**Summary:** The schema is designed to **store raw data and interpretations side by side**. It models the temporal aspect via an Ownership table linking Party–Property with date ranges. It supports “fuzzy” data by allowing alternate values and confidence flags in fields. This foundation enables the chain construction algorithm to reason about ownership succession even when data is imperfect.

## Two-Phase Chain Construction Algorithm (with OCR Error Tolerance)

With data and indexes in place, we now describe the algorithms for building the chain-of-title: **Phase 1 (Backward traversal)** to find the root of title, and **Phase 2 (Forward traversal)** to gather all documents per owner’s tenure. Throughout, the algorithm uses *probabilistic matching* and heuristics to handle OCR errors and ambiguities, ensuring it does not break even if some links are uncertain. It will produce a ranked chain (or multiple possible chains) with confidence scores, rather than a single deterministic chain that might be wrong.

### Phase 1: Backward Ownership Tracing

**Goal:** Starting from the known current owner (and property), find the sequence of prior owners back ≥50 years or to original grant, whichever comes first.

**Inputs:** Current owner’s name (as given or as OCRed from current deed) and optionally the property identifier or current deed reference.

**Process:**

1. **Initialize Current Owner:** We assume we have the current owner’s name and possibly the deed by which they got title. Often, a title search starts from the last recorded deed into the current owner. If we have that document in the system, we identify: current\_owner\_party (Party entry for them) and get current\_owner\_acquire\_doc (the deed they received, with its date). If we don’t have the deed, we start with the name and property and will search for it.
2. **Backward Loop:** While the current\_owner’s acquisition date is > 50 years ago (or if unknown, while we haven’t hit an older boundary):
3. Let target\_name = current\_owner\_party.name\_normalized (plus possibly alt names if needed) as the name to search for **as a grantee** in older records (because to find how they got the property, we look for a deed where someone conveyed to them).
4. **Search Grantor/Grantee index:** Query the Document index for any document where grantee\_name matches target\_name (fuzzy). Use the **fuzzy matching strategy** to allow OCR variations. Also, constrain by date if possible: we might know the current owner acquired the property on a certain date from their deed; in that case, that deed is our link. If not known, we search backward broadly.
   * If property\_id is known, also search for documents with that property\_id and a recording date earlier than current\_owner’s acquisition (e.g. if current deed was in 2010, search < 2010).
5. **Identify Candidate Prior Deeds:** Likely we’ll find one or more candidate documents:
   * If exactly one deed shows grantee ≈ current\_owner (with high similarity) – we likely found the correct prior transaction. For example, a 1990 deed where grantee "Robeat Jolinson" (similar to "Robert Johnson") with date 1990.
   * If multiple hits (maybe common if name is common, e.g. “John Smith”), we need to filter by context: ideally by **property**. Check each candidate’s property description – does it refer to the same property? We compare either property\_id or if none, compare legal description text similarity. A correct link should involve the same parcel. If one stands out as matching the property, choose it. If property info is insufficient, use dates or other clues: perhaps the one that is chronologically appropriate (just before current owner’s tenure). E.g., if current owner’s deed was 2010, a prior deed in 1990 to them likely is the one, not another in 1985 to a same-named person (which might be a different property or a different person).
   * If still ambiguous (two deeds in similar date range to someone of that name), we might have to pursue both paths in parallel (creating a **branch** in chain). This is where our algorithm becomes a graph search exploring different possibilities. We can carry both chains forward until further evidence disambiguates or present both to user.
   * If none found (no deed with matching grantee), possibilities:
   * OCR error made the name completely unrecognizable (e.g. “Johnson” spelled so poorly it didn’t match). We then try broader search: perhaps search by property (all deeds on that property around or before that time). If we have property id or legal description, find any deed transferring that property in a plausible period.
   * Off-record transfer (like inheritance or court order): meaning no deed exists for that owner’s acquisition. In this case, we might consider we reached a gap. The system should flag this: e.g. “No prior deed found for X – possibly acquired via will or unrecorded transaction around [year].”
   * Or the chain went beyond 50-year limit: if we already are past 50 years, maybe we stop and call the last found owner the root (or if legally required, search to original grant if records available).
6. **Add to Chain:** Once a prior deed is identified, add that owner to the chain:
   * Determine prev\_owner\_party = deed.grantor (the person who sold to our current owner).
   * Record the ownership period: prev\_owner owned from deed.date of that transaction (or possibly from earlier if we find when they got it) until they sold in that deed (the date in question).
   * Set current\_owner = prev\_owner and repeat loop to go further back.
   * Also, store the deed in the chain list along with a confidence for this link. The confidence could be a combination of:
   * Name match confidence (similarity of names, and OCR confidence of that name in the deed).
   * Property match confidence (did the legal description clearly match? if yes, high confidence; if we’re not sure we might lower it).
   * Document type check (it should be a deed of transfer; if the only thing we found was, say, a quitclaim or a foreclosure, ensure it indeed transfers title).
   * For example, Link: “1990: Alice Smith → [Robert Johnson] (via Warranty Deed Doc#12345)” with confidence 0.9 if the name matched closely and property matched.
7. **Heuristics for Tough Matches:** The algorithm should employ some smart tricks:
   * If grantor name in candidate deed is somewhat different but other clues match, consider it. E.g., maybe the deed shows grantor as “Estate of John Doe” to Robert Johnson. Then our prev\_owner might not be a single person but an estate of someone (or John Doe died and his estate conveyed). We should still capture that link (with a note of the special case). Or if it’s “Mary Johnson et al.” as grantor, meaning multiple owners selling – for chain, we can pick the primary name and note multiple.
   * **Time gap heuristic:** If we find a deed from much earlier than expected (e.g. current deed 2010, next found previous deed 1950, with nothing in between), that’s a red flag that we missed something in between. Possibly intermediate records’ OCR were too bad to match the name. In such a case, consider broad search by property for any owners in between. Also possibly consider if the property was held by the same family (maybe Robert Johnson inherited from someone without a deed). The system can’t fully resolve that, but it should note the gap (e.g. “No deed found transferring between 1950 and 2010; title gap of 60 years – likely off-record transfer or long-term single ownership”).
   * **Alternate path search:** If the straightforward name search fails, try searching using slightly different info:
   * The **grantor index**: If we know the current owner’s deed, it lists the grantor (who sold to them). We should use that directly if available (that is exactly the previous owner!). If the user started with the actual current deed, we already have previous owner in hand. If not, we may inadvertently do an extra step: e.g., searching grantee index for current owner to find that deed which has a grantor. It’s worth clarifying: often the “backward” search is essentially following the chain of deeds: current owner’s deed gives previous owner; then find previous owner’s acquisition deed, etc. If we always have the deed, we might not need to search by fuzzy name every time – we could follow references (like many deeds reference the previous deed by book/page). However, not all do, and relying on references might be tricky if OCR misread them.
   * **Reference linking:** Check if the deed has a reference to a prior instrument (some deeds say “being the same property conveyed to grantor by deed dated X recorded in Book Y Page Z”). If so, we can jump directly to that referenced deed (if our DB is indexed by book/page). Use fuzzy match on those numbers if needed. This could speed the backward search and be more accurate (less reliance on names). We will attempt to parse such references in documents and, if found, navigate by them. If the reference is OCR’d incorrectly, we can try nearby numbers.
   * If we have the reference and find that document, verify it indeed makes sense (the grantee of that referenced deed should match our grantor, ideally).
8. **Stop Criteria:** Stop when either:
   * We reach a deed older than 50 years (e.g. from 1970 if current year ~2025) that can serve as root of title. Often, title standards require going back at least 50 years or to a clearly valid root (like a government patent or a subdivision creation deed).
   * Or when we reach the original source (like a land grant or first record for that land).
   * Or if the chain breaks (we encountered an unresolvable gap) – in which case stop and mark the last known point as the provisional root, with a note of uncertainty beyond that.
9. The result of Phase 1 is a list of owners from oldest (root) up to the current owner, each with their link deed and relevant info.

**Phase 1 Example:** (Following test scenario specifics) - Start: Current owner is **Robert Johnson Jr.** (say he acquired in 2005 from a deed). - That deed says grantor was **Bobby Johnson Sr.** (just a hypothetical scenario). Actually, scenario says “Robert Johnson (1990) OCR’d as Robeat Jolinson”, later “Bobby Johnson Jr., Mary Johnson illegible” – it sounds like Robert Johnson Jr. might have gotten it in 1990, and then later some Bobby Jr, Mary? Possibly Robert Sr. to Robert Jr & Mary? Let’s assume Robert Johnson Sr. (maybe listed as just Robert Johnson) owned in 1990 and conveyed to Robert Jr. and Mary (his children?) at some later date. Hard to parse scenario text, but we’ll illustrate generally. - Backward step 1: Search for deed where grantee ~ “Robert Johnson Jr.”. Suppose we find none directly because maybe he got it via inheritance. But maybe the current deed (if we had one) is the transfer to him. If not, look for “Robert Johnson” (without Jr) or “Bobby Johnson” variants. We find a 1990 deed where grantee “Robeat Jolinson” (which we recognize as likely “Robert Johnson”). That deed’s grantor is someone (maybe an unrelated seller). - Confirm property matches and date (1990 fits a plausible acquisition for Robert Jr., although Jr suggests there was Sr alive at time). We accept that as the deed into Robert (perhaps Jr). - That deed’s grantor is, say, **Alan Thompson**. Now current\_owner becomes Alan Thompson for next loop. - Next, search for deed where grantee ~ Alan Thompson (back in time). Find e.g. a 1970 deed to Alan Thompson from ABC Corp. - Now current\_owner = ABC Corp (the previous owner). - ABC Corp, search backward: we may find “A8C Gorp” in 1955 (OCR of ABC Corp). Recognize it via fuzzy search as ABC Corp. Take that deed, etc., until maybe root.

This backward chain has navigated OCR errors by fuzzy search at each step, and at “ABC Corp” we saw OCR misread it as “A8C Gorp” in an older doc but still caught it. The key was Soundex wouldn’t help for “8” vs “B”, but trigram would catch that “A8C” vs “ABC” is very similar (two out of three chars match if we treat 8 ~ B in shape). We might specifically allow digit-letter similarity in matching or have listed “8” and “B” as a pair to consider.

### Phase 2: Forward Chronological Traversal for Each Owner

Once we have the chain of owners (with their ownership periods roughly determined from the deeds found), we perform a forward search to collect **all documents affecting the property or the owner during each ownership period**. This includes mortgages, liens, releases, easements, etc., and also confirms which encumbrances were cleared.

**Process:**

1. We have a list of Ownership periods: e.g. Owner A: 1955–1970, Owner B: 1970–1990, Owner C: 1990–2005, Owner D: 2005–present (from Phase 1). Each has associated deed in and deed out (except current who has deed in only).
2. For each ownership period (for each owner in chain):
3. Determine the **time window**: from the date they acquired (inclusive) to the date they conveyed (exclusive, perhaps, since after conveying they no longer own). If the end date is unknown (like current owner), use today’s date or mark open-ended.
4. Also determine possible **property identifier** for that period. Ideally, we have one property\_id constant across chain, but if not, use the legal description from their deed as the reference.
5. **Query all documents in that date range** that involve that owner *or that property*. We will use a combination of:
   * Owner’s name query in all docs between those dates (the name could appear as either grantor or grantee in deeds, or as a party in liens, etc.). Use fuzzy matching because the name might be spelled differently in different docs, and include variations (if we have alt names for that party, use them too).
   * Property-based query: all docs with matching property legal description or ID in that date range. For example, search the full text for the lot/block or other key identifiers of that property during those years. This can catch things like an easement that might mention the property’s lot but perhaps only the developer or neighbor’s name.
   * The intersection of these would ideally cover everything, but to be safe, we take the union (any doc that either mentions the owner’s name or the property).
6. Filter the documents found:
   * Exclude the deed that conveyed the property to the next owner (that one is the boundary, not within the period).
   * Focus on relevant document types: mortgages, releases, assignments, liens, judgments, easements, leases, etc. (We may have everything, but some might be unrelated if the name is common. E.g. John Doe might have another property’s mortgage in that period – to avoid confusion, we check if the document’s property matches or likely pertains to our property. If we can’t tell property from the doc, might need manual check. Usually, mortgages and liens will describe the property too, so property matching helps eliminate those on other properties of the same owner.)
7. Each identified document is then associated with that ownership period in the output.
8. For each document, determine its **effect on title**:
   * Mortgage: mark as an encumbrance starting at its recording date. Check if we find a release (satisfaction) for it during the same or later period. If a release is found (with correct reference or at least matching mortgage amount/parties) then mark the mortgage as released (so not an active encumbrance beyond that point). If no release found up to current, then it might still be outstanding – flag as potential issue (though often old unreleased mortgages may be assumed paid but not recorded; still, title exam notes them).
   * Lien/Judgment: similar approach – see if a release or cancellation is recorded.
   * Easement: usually permanent (doesn’t get released), so it remains affecting the property.
   * If a document’s OCR is poor (e.g. marginal satisfaction not captured), the system might not find a release. We’ll list the mortgage as active, *but* possibly flag “original mortgage from 1980, no release found – check image for marginal note” if we suspect OCR missed it. The heuristic could be: if a mortgage is old (decades ago) and likely paid off (loan term passed), but no release in system, suspect a missing or mis-indexed release.
   * Assignment of mortgage: track if a mortgage was assigned to another party – not directly needed for chain-of-title, but good to note if trying to find who could release it.
   * Any conveyance during the period that doesn’t change the owner? Possibly none, but e.g. a quitclaim from a co-owner to the owner might happen (clearing spouse interest, etc.). The algorithm should catch that as it involves the name and property. In output, we note it, but it doesn’t change the owner, just the vesting.
9. **Confidence and Ambiguity:** Use OCR confidence to qualify each document:
   * If we include a document because of a fuzzy name match and low confidence, mark it as such (e.g. “(OCR guess)”).
   * If multiple uncertain docs refer to something similar, that can raise confidence (like two low-confidence liens both mention a similar name spelling—together they reinforce the owner’s name).
   * If we cannot confirm a document’s connection (like a lien with a very generic name and no clear property reference), maybe omit it or list under a “possible additional docs” section.
10. **Sort documents chronologically** for that owner. This will list, say, all encumbrances during their ownership in order of date.
11. Summarize Encumbrance Status: After listing all docs for each period, mark clearly which encumbrances are still active at the end of that period. This requires checking if for each mortgage or lien in Owner A’s period, was a release found *either in Owner A’s period or perhaps in Owner B’s period* (because sometimes an encumbrance is paid off after property transfer, though usually resolved before transfer). If not found at all, it might still affect current title – that’s crucial to flag.
12. We might propagate unresolved encumbrances forward: e.g. if Owner A’s mortgage from 1980 was never released, it technically burdens the property even under Owner B, C, etc., unless time-barred. We should carry it until we find a release or satisfaction, or it’s beyond statutory limits (perhaps beyond scope to determine automatically, but we can note it).
13. Similarly, easements carry forward through all owners.
14. **Output Structure:** The result will be something like:
15. Owner A (1955–1970): acquired by Deed X from Y in 1955. During ownership: Mortgage in 1955 (released 1965 by Doc Y), Second mortgage 1960 (no release found – STILL ACTIVE?), Easement 1962 (permanent), Sold to Owner B by Deed Z in 1970.
16. Owner B (1970–1990): etc… listing all docs in that period.
17. … up to current owner.
18. Each item with source references and confidence indicators.

**Algorithm Resilience:** This forward search is designed not to miss documents even if OCR is imperfect: - We search by multiple keys (name and property and date) to catch docs that might be missed if one field is wrong. - We allow fuzzy name matches so an OCR error in a lien doesn’t cause us to miss it (e.g. lien against “Robert Johnsen” would still be found). - We consider property match which helps if the name was completely off but the legal description can tie it. - We do not stop if something doesn’t match exactly; we gather broad and then filter logically, erring on the side of including possible relevant docs (with flags if unsure) rather than missing them.

**Handling Continuation of Ambiguities:** If an ownership was ambiguous (like we had two possible previous owners in Phase 1), Phase 2 can sometimes resolve it. For instance, if chain split into two candidate prior owners A and A’ (we didn’t know which was correct), then in forward search for each, perhaps one path yields documents that cleanly lead to the next known owner, and the other yields nothing sensible. The one that aligns better with what we know about subsequent ownership likely is correct. We can then choose that and drop the other. If both yield plausible results, we might present both chain alternatives to the user to decide.

**Machine Learning Opportunities:** We can incorporate ML to improve chain construction: - A model could be trained on known correct chains to predict if a given fuzzy match is the right link. It could take features like name similarity, doc types, date gaps, etc., and output a confidence that this deed is the correct predecessor. This might help in automated disambiguation beyond simple thresholds. - ML for OCR correction: using a language model or a custom trained model to guess correct names from garbled text (especially for names, one could train a model on common names in region to correct OCR errors). - These are advanced enhancements; initially, rules and heuristics can do a lot.

**Ensuring Continuity:** After building both phases, we double-check continuity: - Each owner’s end date should equal the next owner’s start date (within reason – maybe same day or next day recording). If a gap, flag. - Check that the root of title owner we ended on indeed doesn’t have a prior recorded owner (or beyond 50 years is fine). - If any issues, present them clearly (gaps, unresolved liens, etc.).

By using this two-phase algorithm, we ensure the chain-of-title is constructed as completely as possible. The algorithm’s use of fuzzy matching and fallback strategies prevents OCR issues from causing dead-ends. Instead of stopping, the chain will branch or mark uncertainty and continue. Every link and document comes with a confidence, allowing a title examiner to focus on low-confidence parts for verification.

## Handling Common OCR Errors and Ambiguity Patterns

Real estate documents pose many recurring challenges, especially when processed via OCR. We outline specific patterns mentioned and strategies to handle each, ensuring our system remains robust in these scenarios:

* **OCR Name Variations:**
* *Example:* “John Smith” appears as “Jolin Snuth” in one OCR, later as “J.R. Smith” or in a trust name.
* *Strategy:* Use the fuzzy indexing and alias mechanisms as discussed. “Jolin Snuth” vs “John Smith” – trigram similarity will be moderate (both have "Jo", "hn" vs "li", "Snuth" vs "Smith" – not great, but phonetic Soundex might catch SNT vs SMTH? Probably not, but a couple letters are off). In such cases, the property context is vital. Our chain algorithm, upon failing to match “Jolin Snuth” to a known name, will search by property and likely find that a previous or subsequent record had “John Smith”. Once suspicion arises they are same, we link them and effectively treat as one party. We also flag that OCR likely mis-read it. For “J.R. Smith”, our fuzzy search should match it to "John Smith" if J.R. could stand for John (though algorithmically, “J R Smith” vs “John Smith” have low textual similarity). We might need a rule: if initials match the name’s initials and last name matches exactly, consider it a potential match (especially if context supports it). Similarly “Smith Family Trust” vs “John Smith” is tricky: if a person conveyed into their own trust, the chain still continues but the owner name changed form. Our system might list it as a new owner (the trust) since technically it’s a different legal entity, but for title purposes, it’s often treated continuously if John was behind the trust. We could note it as “John Smith, as trustee of Smith Family Trust”. This may require manual note unless we detect keywords like “Trust” or “LLC” indicating an entity change.
* In summary, for names: leverage all text matching tools plus custom logic for initials and common alternate forms (like Bob vs Robert – we could maintain a small dictionary of nicknames). Where high ambiguity remains, surface both possibilities (e.g. “John Smith (possibly OCR’d as Jolin Snuth in Doc #X)”).
* **Damaged Critical Data (Dates, Numbers):**
* *Example:* Recording date “Jan 3?, 200?” with unclear day/year digits.
* *Strategy:* As mentioned, represent uncertain dates as ranges or approximate. For query purposes, include the whole range. If an exact date is needed for output, mark it like “Jan 3?, 200?” in the report to indicate uncertainty. We can often infer missing pieces: if year “200?” and context suggests likely 2008 (because earlier and later docs around it are 2008), we can fill that in with a note. But we avoid false precision; better to mark uncertain. For day “3?”, likely 30 or 31 if two-digit with second missing. Perhaps use the max (31) for internal range but display “30 or 31”. In indexes, using 31 won’t cause big error except possibly including an extra day beyond actual – negligible.
* Other numbers like book/page references if partially illegible, we attempt variations as previously described. If we can’t resolve, we might store the reference as “Book 12?3” and search by parts (like find any book 1200-1299 page something referencing similar parties).
* If an amount (like mortgage amount) is misread (e.g. $100,000 as $100,0O0 with letter O), it’s less crucial for chain-of-title but could be for matching releases (some releases mention amount). We can normalize digits similarly, treating O as 0, etc., if numeric context.
* **Missing Links + OCR Errors:**
* *Scenario:* We have a deed into A (clear), but next we find deeds out from C, D, E – none from A because maybe A’s name was OCR’d poorly or A transferred off-record.
* *Strategy:* This indicates a break in the chain. If A was supposed to convey to someone but we never found A as grantor, instead we see others. Possibly A died and multiple heirs C, D, E got pieces (like partitioned or each inherited some interest, and then sold). Or OCR missed A’s name when searching. In such cases, our algorithm would have searched for A as grantor and not found, then maybe searched by property to see if any deed out of property by someone unknown. If we find that C, D, E sold it later, we deduce that A must have transferred to them (perhaps via inheritance or unrecorded gift). We should then annotate: “No deed from A found. However, in [year], C, D, E (possibly heirs of A) conveyed the property. This suggests A’s interest passed to them off-record (e.g., death).” We then continue chain from C, D, E as needed (if they conveyed to next known owner). Essentially, treat the multiple owners as a joint ownership period after A, with ambiguity. This is complex but not uncommon in title.
* OCR’s role: It might have been that the deed from A to C,D,E existed but OCR didn’t match A’s name. Our property-based search in forward phase might catch that deed in A’s period anyway (since property and date range match). If so, we wouldn’t have a gap, we’d have found that deed albeit with A’s name misread. So either way, something will surface. The system’s job is to highlight that A’s departure wasn’t clean: either missing or unusual (multiple grantees).
* For multiple new owners (like siblings inheriting), our chain might branch or we might treat them collectively as co-owners depending on records. Ideally, we follow each of their conveyances if they split the land. That can complicate the concept of a single chain-of-title since the property might have been subdivided. For our scope, maybe assume property remained whole.
* **Old Document Challenges (Handwritten, etc.):**
* *Example:* deeds from 1970s or earlier might be handwritten or typewriter-faded, causing OCR to struggle.
* *Strategy:* These likely come through with very low confidence scores. Our pipeline will flag them for manual review (confidence below threshold). We might consider sending such docs for manual transcription or double-checking. In chain algorithm, treat these with caution: if a critical link relies on a badly OCR’d deed, perhaps cross-verify by other means (like if we know the book/page, maybe retrieve a better scan or see if a summary exists in a later document). If not, proceed with whatever data we got but mark that “1950 deed – OCR quality poor, data unverified.” Ensure the chain doesn’t break: even if names are illegible, maybe the continuity of property from prior to next known owner is still inferable (like if book/page sequence and date sequence).
* We should store multiple interpretations for handwritten text – sometimes one OCR engine might output one guess, another engine outputs another. Both can be stored as possibilities.
* Possibly reach out to external sources: e.g., some counties have older indexes typed out – if accessible, could use that to corroborate. That might be outside our system though.
* **Corporate/Entity Names and OCR:**
* *Example:* “ABC Corp” OCR’d as “A8C Corp” or “ABC Gorp” or fully spelled “ABC Corporation”.
* *Strategy:* Our fuzzy matching on names will handle minor letter/number differences. We can also include rules: single-digit in middle of letters likely is OCR of a letter (8 <-> B, 0 <-> O, 1 <-> I). We can correct “A8C” to “ABC” in normalization if confidence low. Also, our search should consider “Corp” vs “Corporation” by stemming or a dictionary of common legal suffixes (Corp, Co, Ltd, LLC, etc.). If OCR produces “Gorp” from “Corp”, trigram similarity still works (Gorp vs Corp share orp – likely similar enough). We might add a specific list of legal terms to look out for and correct (e.g. any word like Gorp -> Corp, Cornp -> Corp if 'rn' misread etc.). For indexing, consider normalizing all entity suffixes to a standard form or remove them for matching (so “ABC Corp” and “ABC Corporation” both index under “ABC” plus maybe type marker).
* Ensure Soundex/Metaphone consider those as well (though those algorithms are more for person names, but e.g. Metaphone would treat B and 8 differently? 8 is not alphabetic, but if it sneaks in raw text, our normalization would remove digits from names before applying phonetic coding, likely).
* **Legal Descriptions OCR errors:**
* *Example:* “N 89°15’30” W” read as “N 89°l5’3O” W (l instead of 1, O instead of 0).
* *Strategy:* Recognize the pattern of degrees ° and minutes ' and seconds ". We know they should be numbers. Implement a small post-OCR parser for bearings:
  + After OCR, find patterns like [NSEW]\s+(\d+)[°\-](\d+)['\-](\d+) (with possible O or l characters). Validate numeric ranges: degrees should be 0-360 (or 0-89 for metes and bounds usually as bearing), minutes and seconds 0-59. If we see a letter where a digit expected (like l5), replace with likely digit (l->1, O->0) if that yields valid number. In “l5’3O” both 'l' and 'O' are in digit positions, so correct to "15'30".
  + We store the corrected version in normalized description, but keep raw too.
* If a legal description is extremely garbled, it might affect property matching. But even partial correct info can help (like section/township or lot number). We could attempt to parse known keywords (Lot 5, Block 3, etc.) and their numbers, correcting obvious OCR issues (Lot S might be 5, Block O might be 0, etc.).
* These corrections improve our ability to identify the same parcel across docs. Also, if needed for output, we can present the cleaned version for readability but footnote that OCR errors were corrected.
* **Document References (Book/Page) errors:**
* *Example:* Book 150 Page 123 misread as Book 15O Page I23 (O instead of 0, I instead of 1).
* *Strategy:* We parse references as separate number fields. Use numeric context: book/page are typically all digits. If non-digits present, convert letters that resemble digits. We can even try to validate by looking up the reference:
  + Query our database for an existing document with that reference (book 150 page 123). If not found, try replacing O/I and search 150/123 vs 15O/I23 etc.
  + If a match is found (i.e., we have that doc image in system), we can confirm and then store the correct reference.
  + If still not found, it could be referencing a document outside our dataset (maybe older than we have), or the OCR is too far off. We then mark the reference as unverified.
* Our chain algorithm uses references whenever possible to directly jump to docs, so handling these errors increases the success of that.
* **Signature Blocks and Notary Text:**
* Handwritten signatures often confuse OCR (it might output gibberish or nothing). But often the typed names are somewhere else, so we may not rely on signatures. However, sometimes the only mention of a party is in a signature line (like on old documents, the grantor’s name is only in cursive at the bottom). Document AI might or might not capture that. If it doesn’t, that’s a big problem – no grantor name extracted. We may need to have a secondary OCR pass focusing on signatures with a model specialized for handwriting, or at least leaving a placeholder. This scenario might be rare in printed deeds, but could occur in e.g. wills or affidavits.
* We should allow for the fact that a document might have an **unknown party** due to unreadable signature. The chain algorithm in backward search, if it had the grantee (the known part) but couldn’t read grantor, might have to skip identifying that earlier owner properly. We would then rely on the next document which presumably lists that owner as grantee from someone else. Worst case, that link remains a gap.
* We flag it: “Grantor’s name on 1945 deed illegible (OCR couldn’t read). Possibly the prior owner’s identity is unknown without manual review of document image.”
* **Marginalia and Stamps:**
* *Example:* A mortgage document has a stamped “Satisfied” note in the margin when paid off, but that might not be separately indexed or OCR might not capture it due to orientation/handwriting.
* *Strategy:* This is tough for automation. Some modern systems capture marginal notations if rescanned, but often they are missed. We cannot fully rely on OCR to catch these. So:
  + When a mortgage is found with no separate release document, the system should not assume it’s active; instead, flag it: “No release document found – check original mortgage for possible marginal satisfaction.” This tells the user to manually confirm.
  + If we have high-quality images, one could attempt image analysis to detect stamps like “PAID” or “RELEASED” (maybe by looking for specific shapes or keywords at certain positions). That’s advanced; likely easier to have a human check flagged ones.
  + Our database can have a field in Document for “has\_marginal\_note” if identified (perhaps a user could mark it after seeing it).
  + Over time, perhaps a machine learning model could be trained on sample marginal notes to OCR them (maybe treat them as separate text regions rotated 90 degrees).
  + For now, acknowledging this limitation and ensuring such cases are not ignored is key.
* **Document Type Misidentification:**
* *Example:* The OCR misreads “DEED” in the header as “GEED” or something, leading to wrong classification.
* *Strategy:* We classify document type not just by a single OCR word, but by multiple clues:
  + Titles or keywords: e.g. “Indenture” or “Mortgage” often appear in the text. Or the presence of certain phrases (“secured by a note”, “in consideration of $[amount]” indicates a deed, etc.). We can run simple keyword searches in OCR text to verify type. If OCR title is low confidence, and content contains “Mortgagee” or “promissory note”, then it’s likely a Mortgage even if header was misread.
  + Could use a ML classifier on the text content to label document type (train on a set of known deeds vs mortgages).
  + In the database, allow type to be updated or multiple tags if unsure (like mark as “Deed?”).
* If misidentified type slips through, it might cause us to treat a mortgage as a deed or vice versa. The chain algorithm could get confused if a mortgage looked like a deed. To mitigate, we ensure we differentiate by context:
  + Deeds have grantor AND grantee transferring ownership.
  + Mortgages have borrower (grantor) and lender (grantee) but do not transfer ownership. If our backward search accidentally picks a mortgage thinking it’s a deed, we’d realize the “grantee” is a bank and likely the next link doesn’t make sense (the bank didn’t become owner in chain-of-title, except in foreclosure scenarios, which would be explicitly a deed later).
  + So we likely wouldn’t mistake it because the chain logic expects an owner-to-owner link. If a doc’s grantee is a bank (and type was mis-set as deed), we might catch that because the next deed we found might have a different grantor etc.
  + Nevertheless, improving type detection reduces such confusion in the forward listing of documents (ensuring we categorize them correctly for display under encumbrances vs transfers).

By addressing each of these patterns with targeted strategies, our system becomes robust against them. Importantly, we **don’t discard data due to OCR faults**: even a partially recognized name or date is used as far as possible (with fuzziness) to continue linking, and uncertainties are flagged rather than causing failure. Over time, by correcting these common errors (through code rules or manual fixes that get fed back), the accuracy of the chain construction will improve for future records.

## OCR-Aware Query Optimization and Ambiguity Resolution

Building on the indexing and algorithms, we consider how queries will be executed and results combined to resolve ambiguities. We want our queries to be both **efficient** (fast response for interactive use, e.g. in a title search app) and **effective** (returning the needed records despite OCR imperfections). Here are techniques and examples:

* **Fuzzy Search Algorithms:**  
  We incorporate fuzzy matching at query time using Levenshtein distance and Jaro-Winkler for short strings. For example, for searching names, Python’s difflib or a library like rapidfuzz could rank candidate names by similarity. In SQL, we saw how pg\_trgm does it. If using an external search (ElasticSearch):
* We can use a **fuzzy query** with a specified edit distance. E.g., {"query": {"fuzzy": {"grantor\_name": {"value": "Robert Johnson", "fuzziness": "AUTO"}}}} which would match “Robeat Jolinson” within 2 edits.
* Alternatively, index with **ngrams** so that even if OCR splits or merges words, the tokens still overlap for matching.
* Jaro-Winkler is good for detecting transpositions or slightly misspelled names with consideration to prefix similarity (it gives extra weight to the beginning of the string, useful because OCR often screws up middle letters more than first few)[[26]](https://arxiv.org/html/2304.03464v3#:~:text=Record%20Linkage%20with%20Multimodal%20Contrastive,because%20there%20are%20visually). We might use Jaro-Winkler to boost scores of candidates where the first few letters match (which often indicates same name).
* **Example Implementation:** In Python, using rapidfuzz.fuzz.QRatio (or partial ratio) to compare “Robeat Jolinson” vs “Robert Johnson” might yield a high score ~ 88%. We could integrate that in our chain assembly when multiple options are present, to choose the best.
* **Query Expansion for OCR variants:**  
  As noted, before performing a search, expand the query to include common OCR confusions. This is especially helpful if using an exact search engine. We maintain a map of confusable characters/groups. For instance:
* OCR\_CONFUSIONS = {  
   '0': ['O','Q','D'],  
   '1': ['I','l','|'],  
   'B': ['8'],  
   '8': ['B'],  
   '5': ['S'],  
   'S': ['5'],  
   'rn': ['m'], # these are multi-char confusions  
   'm': ['rn'],  
   # etc.  
  }
* A function can generate variants of a string by substituting one confusion at a time. For example, generate variants for "Robeat Jolinson":
* replace 'o' with 'a' or 'Robcat'? Not a common one, skip.
* 'beat' vs 'bert'? If 'e' and 'r' confusion? Actually 'Robeat' vs 'Robert': here the confusion is missing 'r'. Perhaps treat missing small letters as a possibility (OCR dropping letters). This is where a direct expansion might not catch, but Levenshtein would.
* 'Jolinson' vs 'Johnson': 'li' vs 'hn'. Could add rule that 'li' <-> 'h' (common in certain fonts). Yes, letter 'h' can look like 'li'.
* We could include that as a pattern: 'li': ['h'], 'h': ['li'].
* Using these, we’d create query strings "Robert Johnson" from "Robeat Jolinson" and search that too (the user might input the correct name though, so we expand the other way to catch OCR in DB). If using SQL trigram, explicit expansion not needed as it handles one-off differences. If using exact matching (like on aliases table), expansions help.
* **Confidence-Weighted Ranking:**  
  When presenting search results or choosing among multiple chain paths, we rank by a combined confidence:
* **Document relevance score:** e.g. text similarity score from trigram or search engine.
* **OCR field confidence:** if a match relies on an OCR with low confidence, downrank it. For instance, a deed with grantee name confidence 30% matching the query might be less reliable than another deed where that name was 90% confident.
* **Contextual logic:** e.g., if a deed’s date is way out of expected range, downrank. Or if multiple fields match (name and property), uprank.

Implementation: For each candidate document or link, calculate a numeric score. For example:

score = (name\_similarity \* 0.7) + (name\_confidence \* 0.3)

or more complex: score = log(similarity \* name\_conf \* property\_match\_confidence \* 100) etc., ensuring high similarity & confidence yields top. We then pick the top for auto chain or consider top 2 if close.

In our output, we can even show a “chain confidence” for the whole assembled chain. Perhaps the product of link confidences or minimum of them. That gives a quick sense if any link is shaky.

* **Combining Multiple Low-Confidence Matches:**  
  Sometimes no single document clearly matches, but several partial matches together indicate a trend:
* E.g., Owner X’s name is very unique, but OCR got different parts wrong in different docs. One mortgage has “X” mostly correct, another lien has a different part correct. If we see 3 documents from 1980s all referencing something like X (with variations), it’s almost certain Owner X had those encumbrances. The system can aggregate by grouping docs that share the same property and have names that all fuzzy-match to X. Even if each individually was below confidence threshold, collectively they make sense that X was the owner and those are his documents.
* We can implement this by after initial retrieval of docs, clustering them by property and approximate name. If in one cluster we have multiple docs, we raise the confidence that cluster belongs to the owner. We could then include all docs in cluster.
* Practically, in forward phase, we already filter by property and date, which inherently clusters relevant docs. So if an OCR name was weird in one doc, as long as property matches, we’ll include it anyway.
* **Pattern Recognition for OCR Errors:**  
  We integrate knowledge of common patterns directly:
* The list of confusions from earlier.
* Recognize when OCR likely split or merged words: e.g., if we see an unexpected space in a name (“Mc Donald” vs “McDonald”) – join them for matching.
* Or if letters swapped (common in OCR if a line break or scanner artifact).
* Some patterns can be auto-corrected, others just flagged.
* It might be beneficial to train an error-detection model that given an OCR’d word and a confidence, predicts if it’s likely incorrect. The paper on confidence-aware error detection (ConfBERT) suggests approaches[[27]](https://arxiv.org/html/2409.04117v1#:~:text=Confidence,develop%20ConfBERT%2C%20a%20BERT). For our scope, simpler rules suffice: any name with low confidence or containing weird char combos is suspect.
* **Using Document Structure Templates:**  
  Many deeds and mortgages follow templates (especially from 1990s onward, often standardized forms). If we know a county’s deed format, we could use template-specific parsing to improve accuracy. For instance:
* If “This Warranty Deed made this \_\_\_ day of \_\_\_, 20\_\_ between [Grantor] and [Grantee]” is the typical text, we can search the OCR text for keywords like “between” to find grantor and grantee positions even if OCR of the names was not perfect around the edges.
* Or the legal description often follows a line starting with “Legal Description:” or just after a known phrase.
* By aligning OCR text to an expected pattern, we can sometimes fill gaps.
* This is more relevant to extraction accuracy than query, but it ensures the data we index is correct. If OCR missed a field but template suggests where it should be, we might salvage it.
* **Redundant Information Leverage:**  
  Redundancy is our friend:
* Names often appear multiple times: in the body, in signatures, in notary acknowledgments. If one occurrence was misread and another is clear, we should capture the clear one. Our parser should look at the entire text for a recognizable name pattern. If two versions differ (e.g., one part says “Robeat Jolinson” and another says “Robert Johnson” in a printed notary section maybe), then we know the correct from the notary (if that part OCR’d better).
* Dates similarly appear multiple places (document date vs recording stamp date). If one is illegible, the other might be fine.
* We should cross-verify within a doc: for example, a mortgage might list the names on page 1 and again in the notary block. If OCR outputs two different strings for what should be the same name, we can choose the one with higher confidence or even combine them (maybe one got the first name right, the other got the last name right).
* Implement: when building the Document’s Party fields, if we find multiple candidate names inside the text that could be the party (perhaps by searching for known last name or using capital letter patterns), use that to confirm or correct.
* **External Cross-Reference:**  
  Though not always available, we could cross-check external data:
* For example, property tax records might list owner names – could verify our current owner name.
* If we had access to an address or parcel database, we could confirm that a certain person owned that parcel in certain years (some counties have ownership history databases).
* Or use public records (like prior title commitments, if digitized).
* These are beyond the self-contained system, but mentioning that integration can further validate OCR interpretations (e.g., confirming a name is a real person in that county, or an address exists).
* Possibly even simple web search for a unique name might confirm spelling.

**Example Queries:**

To illustrate, here are some example SQL/Python queries that demonstrate the system’s capabilities:

* **Backward chain query (simplified):** Find previous deed for owner X:
* WITH cte AS (  
   SELECT d2.\*  
   FROM Documents d1   
   JOIN Documents d2 ON d2.grantee\_id = d1.grantor\_id  
   WHERE d1.doc\_id = :current\_deed\_id   
   AND similarity(d2.grantee\_name, d1.grantor\_name) > 0.4  
   ORDER BY similarity(d2.grantee\_name, d1.grantor\_name) DESC  
   LIMIT 5  
  )  
  SELECT \* FROM cte;
* *Explanation:* Here we assume we have current deed d1, and we look for any d2 where d2.grantee = d1.grantor (the previous owner). We use similarity on names as a safety if, say, we didn’t properly link Party IDs. In a normalized schema, we’d do by party\_id equality rather than similarity since grantor\_id and grantee\_id would match if same party. But if we haven’t merged parties, similarity is backup. We limit to top 5 similar, expecting the correct deed to be among them.
* **Forward encumbrance query:** Find all docs for Owner X during 1990–2005:
* SELECT doc\_id, type, recording\_date, grantor\_name, grantee\_name  
  FROM Documents   
  WHERE recording\_date >= '1990-01-01' AND recording\_date < '2005-01-01'  
   AND (  
   -- name matches as party  
   (similarity(grantor\_name, 'Robert Johnson') > 0.3 OR similarity(grantee\_name, 'Robert Johnson') > 0.3)  
   -- plus any known alt spellings:  
   OR (grantor\_name ILIKE '%Johnson%' AND grantor\_name ILIKE '%Rob%')   
   -- the above line is a crude filter to catch partial if similarity fails  
   OR property\_id = :prop\_id  
   )  
  ORDER BY recording\_date;
* This would use indexes on date and either trigram index on names or fall back to ILIKE if needed (the ILIKE example is one way to catch "Johnson" where maybe first name is off; but trigram does that better). We include property\_id match separately which might catch docs with completely different names (like an easement might list the grantee as the power company, not the owner’s name, but property\_id links it).
* **Using Python for fuzzy matching on candidates:**  
  Suppose we have a list of names from the index that came up for "Robert Johnson". We can refine in Python:
* from rapidfuzz import fuzz, process  
  name\_query = "Robert Johnson"  
  candidates = ["Robeat Jolinson", "Robert J Johnson", "Albert Johnson", "Robt. Johnson Jr."]  
  for cand in candidates:  
   score = fuzz.token\_sort\_ratio(name\_query, cand)  
   print(cand, score)
* This might print:
* Robeat Jolinson 86  
  Robert J Johnson 90  
  Albert Johnson 77  
  Robt. Johnson Jr. 95
* We see "Robt. Johnson Jr." has 95 (very close, just abbreviation differences), "Robeat Jolinson" 86 (pretty close, likely should be considered same person), "Albert Johnson" 77 (some similarity because last name matches, but first is different enough to likely exclude if others exist). We would set a threshold maybe ~80 for candidate acceptance, but if none above 80, we might take the top anyway.
* **Confidence combination example:**  
  If "Robeat Jolinson" had OCR confidence 60, and "Robt. Johnson Jr." 90, we might create an aggregated confidence for the link that "Robert Johnson Jr is the same as Robeat Jolinson" maybe ~ (0.86 similarity \* average(0.60,0.90)) = 0.64. But since we have multiple references (the Jr one is strong), we’d lean towards confirming the identity.
* **UI Query for manual review flags:**  
  Possibly a query to list all places where confidence < X or gap found, so a user can quickly review those. E.g.:
* SELECT \* FROM Ownership WHERE confidence < 0.5 OR gap\_flag = true;

In essence, the querying layer is built to **embrace fuzziness**: using specialized indexes and algorithms to ensure that slight mismatches do not exclude the correct records. By weighting results and combining evidence, the system surfaces the most likely chain. Ambiguities are resolved either by picking the highest-confidence path or by clearly presenting alternatives to a human (with context as to why each is plausible). Throughout, performance is maintained by leveraging indexes (trigram indexes for text, b-tree for dates, etc.) so that even fuzzy queries avoid full table scans. In testing, we would benchmark, for instance, a trigram name search on a million-record table – Postgres can handle that using the index (it breaks each name into trigrams and does a union of postings, which is quite fast, though we have to tune threshold to not return too many). If performance becomes an issue, we’d adjust (like require at least first letter match to reduce search space, which is a typical trick).

Finally, as OCR technology improves or more data is corrected, our queries inherently become more accurate (because the underlying data gets better). The system is designed to continuously integrate those improvements (e.g., if we manually fix a name, next search will directly hit exact match instead of fuzzy).

## Recommended Technology Stack and Schema Summary

Based on the above analysis, we recommend the following tech stack and provide a summary of the schema design to implement the solution:

* **Database:** **PostgreSQL** (with PostGIS if spatial needed for properties). Use extensions: pg\_trgm for fuzzy text search, fuzzystrmatch for Soundex/Metaphone, possibly unaccent if dealing with diacritics. PostgreSQL provides reliability, and the ability to store JSON for OCR details, plus powerful indexing. It also can scale to the data sizes (thousands to millions of records) on cloud infrastructure (e.g. AWS RDS or Cloud SQL). If needed, consider **GraphQL API** on top for ease of querying in application – this can hide the complexity of fuzzy queries behind GraphQL resolvers that invoke SQL and logic.
* **OCR & Text Processing:** **Google Document AI** for initial OCR extraction (since already in use). Augment with **Python** scripts for post-processing (using libraries like regex for cleaning, rapidfuzz for advanced fuzzy logic, spaCy for NER if needed). Possibly incorporate **Tesseract** for a second pass on difficult sections (especially if Document AI output confidence is low in some region, crop and run Tesseract with a different setting).
* **Backend Application:** A Python-based backend (e.g. using Django or Flask) to orchestrate the pipeline:
* Ingest OCR outputs, clean & store in DB.
* Run chain construction algorithm (could be on-the-fly when user requests or precomputed and stored).
* Provide endpoints or interfaces for retrieving chain-of-title reports, with confidence info.
* **Frontend/UI:** A web interface for title examiners:
* Allow searching by property or owner to trigger chain building.
* Display the chain in an organized way (perhaps a timeline or list of owners with expandable sections for documents).
* Highlight text that is OCR uncertain (e.g. red underline for letters we weren’t sure of).
* Provide links to view document images for verification, especially on flagged items.
* Possibly allow users to submit corrections (which feed back into the system).
* **Schema Summary (Main Tables):**

\*\*Party\*\* (represents people/orgs)  
- party\_id (PK)  
- name\_normalized (text)  
- name\_raw (text)  
- name\_confidence (real)  
- alt\_names (jsonb) – e.g. [{"name": "...", "conf": 0.4}, ...]  
- type (enum: Individual/Company/Trust etc, if derivable)  
  
\*\*Property\*\*   
- property\_id (PK)  
- parcel\_id (if available from external data)  
- legal\_desc\_normalized (text)  
- legal\_desc\_raw (text)  
- location (geom or address fields, optional)  
- notes (text, e.g. "derived from deed book X page Y")  
  
\*\*Document\*\*   
- doc\_id (PK)  
- doc\_type (enum: Deed, Mortgage, Lien, Release, Easement, etc)  
- recording\_date (date)  
- recording\_date\_confidence (real, if OCR uncertain)  
- book (int, nullable if not applicable)  
- page (int, or instrument\_number if applicable)  
- grantor\_id (FK to Party, nullable if unknown)  
- grantee\_id (FK to Party, nullable if unknown)  
- property\_id (FK to Property, nullable if not determined)  
- ocr\_confidence (real, average or lowest field conf)  
- parsed\_fields (jsonb) – stores key fields raw vs cleaned, e.g.   
 {  
 "grantor\_name\_raw": "...", "grantor\_name\_conf": 0.85,  
 "grantee\_name\_raw": "...", "grantee\_name\_conf": 0.60,  
 "amount": 100000, "amount\_conf": 0.90,  
 "related\_doc": {"book":123,"page":45,"conf":0.75}  
 }  
- has\_active\_encumbrance (bool, for encumbrance docs like mortgage; true if no release found)  
- status (text, e.g. "released", "open", "assigned", for mortgages/liens)  
- comments (text, for any manual notes e.g. "illegible signature")  
  
\*\*OwnershipPeriod\*\* (each record is one owner's tenure on one property)  
- ownership\_id (PK)  
- property\_id (FK)  
- owner\_id (FK to Party)  
- start\_date (date)  
- end\_date (date or null if current)  
- start\_doc\_id (FK to Document that conveyed title to this owner)  
- end\_doc\_id (FK to Document by which owner conveyed away title, null if current or unknown)  
- confidence (real) – confidence that this period is correctly identified  
- flags (json or text) – e.g. "gap\_after": true if no direct link to next owner  
  
\*\*NameIndex\*\* (could be a materialized view or table for quick search, or just use Party/Document directly with trigram indexes)  
- Fields: maybe store combinations of last name, first name, soundex, etc for advanced queries.

*(Note: We might not implement NameIndex if using Postgres functional indexes on Party or Document name fields directly.)*

* **Indexes:**
* On Party(name\_normalized text\_ops) – perhaps a trigram index on this for direct searching of parties.
* On Party(soundex(name\_normalized)) – btree for phonetic lookup.
* On Document(grantor\_name gin\_trgm\_ops) and (grantee\_name gin\_trgm\_ops) if we keep names in Document for quick search (even though we have Party, having the actual text can be convenient for search without join).
* On Document(recording\_date) and maybe (property\_id, recording\_date).
* On Document(property\_id) obviously.
* On Document(doc\_type) if frequently filtered by type (like retrieving all releases).
* On OwnershipPeriod(property\_id) and (owner\_id) for retrieving chain by property or by owner.
* On any reference fields like Document(related\_doc\_book, related\_doc\_page) if storing them for quick find of referenced doc.
* **Cloud Deployment:** Use a cloud service for Postgres (as desired by user, e.g. AWS RDS, Google Cloud SQL). Google Document AI already cloud. The processing can run on an app server or cloud function. The volume (thousands of county documents, which might actually be tens or hundreds of thousands of pages) suggests using cloud storage (S3 or GCS) for images and maybe a queue system (like Cloud Pub/Sub) for OCR tasks if doing in bulk.
* **Optional Graph Component:** If later needed, integrate Neo4j (cloud AuraDB or self-hosted) to store Party–Document–Party relationships. That can facilitate running a graph algorithm to find all paths from current owner node to any node 50 years back. But given complexity of merging OCR fuzzy nodes, it might not give a huge advantage unless we fully commit to an entity resolution graph. For now, the relational approach is sufficient and probably easier for the team to manage given familiarity with Postgres.

## Sample Python Code Snippets

Below are some illustrative Python snippets for critical components, demonstrating how one might implement them. (Note: These are simplified examples to convey approach; integration with the actual data structures would be required.)

**1. OCR Post-Processing & Error Correction:**

Let's implement a simple correction for common OCR errors in names and numbers:

import re  
  
OCR\_CONFUSIONS\_CHAR = {  
 '0': 'O',  
 'O': '0',  
 '1': 'I',  
 'I': '1',  
 'l': '1', # lowercase L to 1  
 '5': 'S',  
 'S': '5',  
 '8': 'B',  
 'B': '8'  
}  
OCR\_CONFUSIONS\_MULTI = {  
 'rn': 'm',  
 'm': 'rn',  
 'li': 'h',  
 'h': 'li'  
}  
  
def correct\_ocr\_text(text):  
 # Correct multi-char patterns first  
 for wrong, correct in OCR\_CONFUSIONS\_MULTI.items():  
 # Use word boundaries or regex if needed to avoid partial replacements  
 # Here simple global replace if pattern exists  
 if wrong in text:  
 # Only replace if the result makes a known word? (not implemented here for simplicity)  
 text = text.replace(wrong, correct)  
 # Now single-char replacements  
 result\_chars = []  
 for ch in text:  
 if ch in OCR\_CONFUSIONS\_CHAR:  
 # e.g. replace '0' with 'O'  
 result\_chars.append(OCR\_CONFUSIONS\_CHAR[ch])  
 else:  
 result\_chars.append(ch)  
 corrected = ''.join(result\_chars)  
 # Capitalize first letters of words if it's supposed to be a name  
 # (Assume input text is a name if it contains a space)  
 if ' ' in corrected:  
 corrected = ' '.join(w.capitalize() for w in corrected.split())  
 return corrected  
  
# Example usage:  
name\_raw = "Robeat Jolinson"  
print(name\_raw, "->", correct\_ocr\_text(name\_raw))  
date\_raw = "Jan 3?, 200?"  
print(date\_raw, "->", correct\_ocr\_text(date\_raw))

Expected output:

Robeat Jolinson -> Robert Johnson  
Jan 3?, 200? -> Jan 31, 2000? # Actually our function isn't smart enough to fix ?, so maybe it leaves it.

In reality, we'd have more context for date, but this shows how letter swaps can be fixed (it replaced 'ea' with 'er' maybe via our rules if 'ea' wasn't touched, hmm our function as written might not fix 'ea' -> 'er'. We might need a dictionary check or to specifically know that 'Robeat' is not a common name but 'Robert' is, which requires a dictionary of names.)

**2. Entity Extraction from OCR (using regex):**

For a deed text, suppose we want to extract parties:

import re  
  
text = """WARRANTY DEED  
  
THIS INDENTURE made this 30th day of January, 1990, between   
ALICE SMITH (Grantor) and ROBERT JOHNSON JR. (Grantee).  
  
Witnesseth: ... (legal description) ..."""  
  
# Simple regex to find "between X (Grantor) and Y (Grantee)"  
match = re.search(r'between\s+(.+?)\s\*\(Grantor\)\s+and\s+(.+?)\s\*\(Grantee\)', text, re.IGNORECASE)  
if match:  
 grantor\_name = match.group(1).title()  
 grantee\_name = match.group(2).title()  
 print("Grantor:", grantor\_name, "Grantee:", grantee\_name)

This would output:

Grantor: Alice Smith Grantee: Robert Johnson Jr.

This is a simplified approach; actual text might not literally have "(Grantor)" labels, but we often see "between X and Y" phrasing. We’d have multiple regex patterns for different document wording styles.

**3. Fuzzy Name Matching (Levenshtein via Python):**

Using rapidfuzz (which is like FuzzyWuzzy but faster):

from rapidfuzz import process, fuzz  
  
name\_query = "Robert Johnson"  
# Suppose we have a list of party names from the index:  
party\_names = ["Robeat Jolinson", "Robert Johnston", "Robt Johnson Jr", "Robert Jackson"]  
matches = process.extract(name\_query, party\_names, scorer=fuzz.token\_sort\_ratio, score\_cutoff=80)  
for match in matches:  
 print(match)

This might output:

('Robt Johnson Jr', 100)  
('Robeat Jolinson', 86)  
('Robert Johnston', 86)

So it found "Robt Johnson Jr" as best (100 score after token sort, because essentially "Robt" and "Jr" and "Johnson" vs "Robert Johnson" is almost a rearrangement difference), and it also found the others above 80. We can then consider these as possible matches. The score helps decide.

**4. Confidence Calculation Combining Factors:**

Suppose we want to compute a confidence for a link given OCR confidence and name similarity:

# Example values:  
ocr\_conf = 0.60 # 60% confidence in OCR of name  
sim\_score = 0.86 # 86% similarity between "Robeat Jolinson" and "Robert Johnson"  
# Let's incorporate property match too:  
property\_match = True # suppose we confirmed property matches, we give that full weight  
prop\_conf = 1.0 if property\_match else 0.0  
  
# We could do a weighted average:  
link\_confidence = (ocr\_conf\*0.4 + sim\_score\*0.5 + prop\_conf\*0.1)  
print(f"Link confidence = {link\_confidence:.2f}")

This yields link\_confidence ~ 0.74 (74%). If property\_match was false, it would drop to ~0.68. These weights are arbitrary; in practice we might give property match a higher weight because it's very indicative.

**5. Heuristic Correction for "rn" vs "m":**

To demonstrate a simple heuristic:

def ocr\_name\_similarity(name1, name2):  
 # custom similarity that treats rn and m as equivalent  
 norm1 = name1.replace('rn', 'm').replace(' ', '').lower()  
 norm2 = name2.replace('rn', 'm').replace(' ', '').lower()  
 # simple ratio of matching characters or use SequenceMatcher  
 from difflib import SequenceMatcher  
 return SequenceMatcher(None, norm1, norm2).ratio()  
  
print(ocr\_name\_similarity("John Somerville", "John Sornerville"))

If the difference is 'm' vs 'rn', our replacement normalizes them both to 'm': It would show a high similarity (likely 1.0 if that's the only difference treated as same).

**6. Chain Construction Outline (pseudo-code):**

def build\_chain(current\_owner\_name, current\_property\_id=None):  
 chain = []  
 visited\_names = set()  
 owner\_name = current\_owner\_name  
 while True:  
 if owner\_name in visited\_names:  
 # loop detected, break to avoid infinite (shouldn't happen in real title chain)  
 break  
 visited\_names.add(owner\_name)  
 # search for deed with grantee ~ owner\_name  
 candidates = search\_deeds\_by\_grantee(owner\_name, property\_id=current\_property\_id)  
 if not candidates:  
 break # no further back deeds found  
 # pick best candidate  
 best\_deed, best\_score = None, 0  
 for deed in candidates:  
 # assume candidates includes similarity score and property match flag  
 sim = deed['name\_similarity']; prop\_match = deed['property\_match']  
 conf = deed['grantee\_confidence']  
 score = sim \* 0.7 + conf \* 0.3 + (0.1 if prop\_match else 0)  
 if score > best\_score:  
 best\_score = score  
 best\_deed = deed  
 if not best\_deed:  
 break  
 # Add to chain  
 chain.append({  
 'owner': owner\_name,  
 'acquired\_from': best\_deed['grantor\_name'],  
 'doc\_id': best\_deed['doc\_id'],  
 'date': best\_deed['date'],  
 'confidence': best\_score  
 })  
 # set up for next iteration  
 owner\_name = best\_deed['grantor\_name']  
 if best\_deed['date'] < cutoff\_date: # 50-year cutoff  
 break  
 return chain

This pseudo-code demonstrates the backward loop logic with scoring. In reality, we would use party IDs and so forth, but conceptually similar.

These code snippets and pseudo-code pieces illustrate how various parts of the system could be implemented. The actual system would integrate them with the database and a larger application logic, handling edge cases and scalability (like using iterative database queries rather than loading all candidates into Python if too many, etc.).

## Performance Considerations and Scalability

Performance is a crucial aspect, given potentially **thousands of properties and documents** to process, and the need to run these searches quickly during a title examination. Here we address how the system scales and what resources it needs:

* **Bulk OCR processing:** Converting county records (potentially hundreds of thousands of pages) via OCR is heavy. Google Document AI is a cloud service that can scale horizontally (and one pays per page). We would batch process initial ingestion perhaps offline. Real-time OCR would be used only for new incoming documents or if re-scanning on demand. The OCR pipeline using 56 CPU cores achieved needed throughput for Kadaster’s project[[28]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=match%20at%20L528%20OCR%20and,into%20the%20spool%20directory%20for)[[29]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=match%20at%20L573%20of%20the,lasts%20for%20approximately%20four%20years) – similarly, we can parallelize OCR tasks in cloud. Storage of OCR text can be large (but text is smaller than images; even 1 million pages of text might be a few GB).
* **Database performance:**
* For queries, the **graph traversal** depth is moderate (maybe up to 10 hops back). Postgres can handle recursive CTE of that depth easily, especially if indexed. The heavy part is fuzzy matching: but pg\_trgm index lookups on a name field are sub-second even on millions of rows if the search term is selective[[25]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Using%20the%20built%20in%20,Solr%2C%20Elastic%20Search). We should ensure to use it properly (e.g. WHERE name % 'Bob' hits index).
* If a name is common like "John Smith", fuzzy search will return many results – but then we filter by property or date which should cut it down. Worst-case, it might fetch dozens or hundreds and then our Python logic filters them – still manageable.
* For forward search, pulling all docs in a date range for a known property is straightforward (range index on date, and property filter, very fast). If property not known, name search plus date is again using indexes on name (fuzzy) and date (range), which Postgres can combine (though combining GIN and B-tree might not be straightforward, we might do two-step: first find candidate docs by fuzzy name (small set) then filter by date).
* **Memory/Storage for variations:**
* Storing multiple OCR interpretations and confidence does increase storage. But text data is relatively cheap. Each document might carry maybe a few KB of JSON metadata and text. Even 1 million docs is a few GB in Postgres – okay on modern systems. Indexes on text (trgm) do bloat size (they can be ~ up to size of the text itself or more). But again, at our expected scale (maybe tens of millions of names entries worst-case?), it's still in tens of GBs which is fine for a cloud DB with proper instance size.
* If performance suffers, one might consider splitting heavy search tasks onto a search server (Elastic). But let's assume Postgres can handle initial requirements.
* **Concurrency:**
* Usually one chain search at a time is fine (one user looking up one property). If multiple users search concurrently, the queries are not too heavy to cause big contention except maybe on very large fuzzy matches. We can optimize common cases (like often the current deed is given, so then we follow direct references rather than wide search).
* We can also cache results of common searches (though each property is unique typically, caching less beneficial except maybe caching name index stuff in RAM).
* **Benchmarking Graph vs SQL:**
* If in the future, relationships become very complex (like if we wanted to quickly find all properties owned by a given person historically, etc.), a graph db might be considered. But for chain-of-title, which is mostly a single linked list per property, relational works well. Graph DB would shine if we had to explore multiple branches at once (like find any common owner between two properties, or find fraud patterns across graph).
* The research indicates graph DB *can* be much faster for recursive queries[[7]](https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database#:~:text=,), but given our manageable depth, Postgres can likely handle it within seconds. If we find any query taking too long (say over a couple seconds), we will tune (add indexes, refine conditions).
* **Quality Assurance Workflow & Improvement:**
* We expect initially many documents may be flagged for review. Over time, those get corrected and stored, so the fraction of flags drops.
* We will maintain metrics: e.g., “X% of chains had a gap that required manual fix” or “Y% of names were corrected manually”. This can guide if we need to improve OCR or add new heuristics.
* Possibly schedule periodic re-processing of older low-conf OCR with improved OCR models (Document AI updates, etc.). Because our system keeps raw images and text, we can re-run and diff results to update fields (keeping old for audit).
* **Data Volume and Cloud Resources:**
* Thousands of records countywide sounds like maybe small (if literally a few thousand). But possibly it could be hundreds of thousands or more (some counties have millions of record images). We design for up to millions to be safe.
* Use a server with ample RAM for Postgres (to hold indexes in memory). The fuzzy search may benefit from work\_mem setting if many similarities computed.
* Partitioning by county or year might help if data grows huge (e.g. partition Document table by year or decades for easier management).
* Backups and maintenance of this DB are straightforward with cloud services.

In summary, our design should be performant for the anticipated load. Graph databases offer tempting performance on relationship queries, but given the team's preference and familiarity with SQL (Postgres) and the need to integrate fuzzy text logic, the relational approach is appropriate and likely sufficient[[13]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient)[[12]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Performance%20Relational%20databases%20face%20complex,querying%20relationships%20between%20connected%20data). We always have the option to augment with external search or a graph layer if needed, making it a flexible, extensible architecture.

## User Interface and UX Considerations

Finally, we consider how to present the results and interact with the system, as the ultimate output is a **chain-of-title report with OCR confidence info**. A user (title examiner) should be able to quickly grasp the chain and focus on uncertain areas. Key UI/UX features:

* **Structured Chain Presentation:** Display the chain as a chronological list of owners:
* Each owner’s period as a section with a heading (e.g. **Owner: Robert Johnson Jr. – Owned 1990 to 2005**).
* Under each, indent or list the documents during that period, ordered by date:
  + E.g. *1990: Warranty Deed from Alice Smith (Book 123 Pg 45) –* *Acquired Title.*
  + *1995: Mortgage to Bank of America (Book 130 Pg 77) – $100k – Released 2004.* (if released, maybe show as ~strikethrough or a green check mark indicating released).
  + *1998: Easement to Utility Co. (Book 140 Pg 10) – (Permanent easement).*
  + *2004: Release of Mortgage (Book 200 Pg 5) – releases 1995 mortgage.*
  + *2005: Warranty Deed to Next Owner (Book 210 Pg 33) –* *Conveyed Title.*
* This clearly separates phases and shows actions during each.
* **Confidence Indicators:** Use visual cues for OCR confidence:
* Highlight or underline any names or dates that were low confidence or corrected. Perhaps a tooltip on hover shows “OCR read: Robeat Jolinson (60% confidence), interpreted as Robert Johnson.”
* If a whole document’s inclusion is low confidence (like a fuzzy match), maybe an icon (e.g. a warning symbol) next to it. Clicking it could show why (e.g. “Grantee name matched approximately, please verify content”).
* Possibly color code text: e.g. red text for uncertain characters. (Similar to how NLM highlighted low confidence characters in red for operators[[30]](https://lhncbc.nlm.nih.gov/LHC-publications/PDF/pub2001010.pdf#:~:text=employed%20in%20MARS%20includes%20a,This%20is).)
* **Active/Released Encumbrances:** Make it easy to see which encumbrances are still outstanding:
* Use different icons or formatting: active liens might be in bold red “(ACTIVE)” label, whereas released ones in grey “(Released)”.
* Maybe have a summary at the end: “Active encumbrances as of today: None” or list ones not released.
* **Flags for Review:** If the chain has gaps or unresolved questions, show them prominently:
* E.g. an entry like: *“Gap:* *No deed found from Owner X to Owner Y (1970s). Possible unrecorded transfer or missing record. Further investigation needed.”* – maybe highlighted in yellow.
* The UI could allow the user to mark if they resolved it (e.g. they found an unindexed will, etc., and they can attach a note).
* **Document Viewing:** Each document entry should link to the image or text:
* A “View document” button that opens the scanned image (or text overlay) so the user can manually read it if needed. This is critical for trust: user must be able to verify the automated system’s work.
* Possibly highlight on the image where a particular field was read if we have that coordinate (some OCR APIs give bounding boxes).
* **Manual Edits:** The UI could permit the examiner to correct data inline:
* e.g., click an owner name to edit if it was wrong. The correction then updates the database (maybe flagged as manual override). This new value would be used in future chain builds.
* Or mark a document as “not related” if the system included a wrong one, so the logic can learn to exclude similar in future.
* Provide a way to upload a better scan if OCR failed.
* **Performance in UI:** Provide search capabilities:
* Search by current owner name or address to start building a chain (if not already triggered by user input).
* Possibly show intermediate search results if needed (like if multiple candidates for prior deed, maybe present a choice “Did the property come from Alice Smith or Allen Smythe?” if our confidence was tied and let user pick).
* However, the goal is to minimize such user decisions by making smart choices; only ask when system is truly unsure.
* **Audit Trail for Users:** Show the raw OCR text or a diff of raw vs cleaned in a collapsible section for those interested:
* For example, under each document, a toggle “show OCR details” which reveals something like:
* OCR Grantor: ALICE SMlTH (Confidence 92%) – interpreted as ALICE SMITH   
  OCR Grantee: ROBEAT JOLINSON (Confidence 60%) – interpreted as ROBERT JOHNSON JR.  
  OCR Date: Jan 30, 1990 (Confidence 85%) – used as Jan 30, 1990  
  ...
* This helps transparency.
* **Guidance and Next Steps:** At the end of the report, perhaps include suggestions:
* “The chain appears complete back to 1950, fulfilling a 50-year search requirement.” or
* “One unreleased lien from 1988 may require a release or affidavit to clear.”
* “OCR confidence low on 1940 deed – manual verification recommended.”
* **Mobile/Printing:** The output likely needs to be printable as a formal report. So the design should translate to a clean printed document (with footnotes or appendix for OCR stuff maybe).

By focusing on clear visualization of the chain and transparency of data quality, the UI will instill confidence in the results and make the examiner’s job easier: they can trust the system for the routine parts and quickly identify where their expertise is needed to resolve uncertainties.

In conclusion, our comprehensive design combines a robust **data architecture**, sophisticated **fuzzy indexing and search techniques**, careful **OCR data handling**, and a user-focused **interface** to tackle the challenges of building chains of title from OCR’d records. This solution will efficiently trace ownership history backwards to a 50-year root[[31]](https://en.wikipedia.org/wiki/Grantor%E2%80%93grantee_index#:~:text=Grantor%E2%80%93grantee%20index%20,name%20of%20the%20parties) and forward through each owner’s tenure, without breaking on OCR errors. Instead, it embraces uncertainty by scoring and flagging it, thus ensuring no stone is left unturned in the title search. The recommended PostgreSQL-based implementation with OCR confidence integration is both practical and scalable, leveraging modern text search capabilities and the team’s existing tech stack preferences. With this system in place, title professionals can dramatically accelerate their examinations, focusing their attention only where the system flags potential issues – a big leap toward **automated, AI-assisted title searching** in the real estate industry[[32]](https://www.afxllc.com/all-articles/revolutionizing-title-search-ai-title-search/#:~:text=5%20Ways%20AI%20Title%20Search,reports%20for%20real%20estate%20efficiency).

## Sources

* Neo4j, *Graph Database vs Relational Database* – Discusses relationship traversal efficiency (graph vs SQL joins)[[3]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=data)[[13]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient).
* AWS, *Graph vs Relational Databases* – Summary of when graph databases excel (multi-hop queries) vs relational[[12]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Performance%20Relational%20databases%20face%20complex,querying%20relationships%20between%20connected%20data)[[6]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=without%20needing%20to%20dynamically%20calculate,make%20graph%20databases%20very%20efficient).
* Randall Degges, *Easy Fuzzy Text Searching with PostgreSQL* – Introduction to pg\_trgm extension for fuzzy matching in Postgres[[19]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=PostgreSQL%20Fuzzy%20Text%20Searching)[[24]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Performance%20Considerations).
* Stack Overflow – Comparison of Soundex vs Levenshtein for name matching[[22]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=Levenshtein%20distance%20looks%20at%20two,for%20missing%20or%20substituted%20letters), recommending Metaphone and n-grams for better accuracy[[18]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=I%20would%20suggest%20using%20Metaphone%2C,and%20spelling%20phonetically).
* TDWI, *Using OCR: How Accurate is Your Data?* – Importance of field-level confidence and thresholds for automation[[1]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=If%20you%20need%20to%20obtain,fields%20required%20by%20the%20business)[[14]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=confidence%20scores%20are%20used%20to,that%20the%20answer%20is%20correct).
* FIG Congress 2010 paper by Wouters et al., *Extracting Information from Deeds by OCR* – Describes large-scale OCR of deed archives (15 million deeds) and mentions handling of “noisy text” from poor scans[[33]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=OCR%20and%20recognition%20engines,into%20the%20spool%20directory%20for)[[34]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=of%20the%20computer%20capacity%20is,lasts%20for%20approximately%20four%20years).
* LHN NLM, *Pattern matching techniques for correcting low confidence OCR* – Describes using dictionaries and approximate matching to correct OCR errors in fields, highlighting use of confidence scores to identify suspects[[30]](https://lhncbc.nlm.nih.gov/LHC-publications/PDF/pub2001010.pdf#:~:text=employed%20in%20MARS%20includes%20a,This%20is).
* Mashvisor, *Chain of Title Guide* – Basic definitions (for context on what chain of title is).
* CourthouseDirect blog, *Grantor/Grantee Indexes* – Explains traditional indexing methods in county records (grantor-grantee, tract index)[[16]](https://www.law.cornell.edu/wex/grantor-grantee_index#:~:text=Institute%20www.law.cornell.edu%20%20Grantor,the%20transferring%20of%20property%20ownership).
* First American DNA, *How ML/AI expedite real estate data* – Alludes to AI improving title searches (for context of industry trend)[[35]](https://dna.firstam.com/insights-blog/how-machine-learning-and-artificial-intelligence-have-expedited-real-estate-transactions#:~:text=How%20Machine%20Learning%20and%20Artificial,expedite%20the%20time%20it).

[[1]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=If%20you%20need%20to%20obtain,fields%20required%20by%20the%20business) [[2]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=Why%20Confidence%20Scores%20Matter) [[14]](https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx#:~:text=confidence%20scores%20are%20used%20to,that%20the%20answer%20is%20correct) Using OCR: How Accurate is Your Data? | TDWI

<https://tdwi.org/articles/2018/03/05/diq-all-how-accurate-is-your-data.aspx>

[[3]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=data) [[5]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=The%20relational%20model%20processes%20queries,other%20techniques%20for%20joining%20data) [[9]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=Relational%20databases%20excel%20at%20operations,operation%20regardless%20of%20optimization%20strategies) [[11]](https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/#:~:text=%2A%20Finding%20friends,recommendation%20paths%20through%20purchase%20history) Graph Database vs. Relational Database: What’s The Difference?

<https://neo4j.com/blog/graph-database/graph-database-vs-relational-database/>

[[4]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient) [[6]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=without%20needing%20to%20dynamically%20calculate,make%20graph%20databases%20very%20efficient) [[10]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Alternatively%2C%20relational%20databases%20use%20index,to%20retrieve%20the%20required%20data) [[12]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Performance%20Relational%20databases%20face%20complex,querying%20relationships%20between%20connected%20data) [[13]](https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/#:~:text=Graph%20databases%20store%20both%20objects,make%20graph%20databases%20very%20efficient) Graph vs Relational Databases - Difference Between Databases - AWS

<https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/>

[[7]](https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database#:~:text=,) [[8]](https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database#:~:text=,) (PDF) Performance Comparison of Graph Database and Relational Database

<https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database>

[[15]](https://nexval.com/15-technology-components-you-need-for-title-automation/#:~:text=nexval,massive%20public%20and%20third) 15 Technology Components You Need for Title Automation - nexval.ai

<https://nexval.com/15-technology-components-you-need-for-title-automation/>

[[16]](https://www.law.cornell.edu/wex/grantor-grantee_index#:~:text=Institute%20www.law.cornell.edu%20%20Grantor,the%20transferring%20of%20property%20ownership) grantor-grantee index | Wex | US Law | LII / Legal Information Institute

<https://www.law.cornell.edu/wex/grantor-grantee_index>

[[17]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=Soundex%20works%20well%20with%20western,It%27s%20intended%20for%20phonetic%20comparison) [[18]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=I%20would%20suggest%20using%20Metaphone%2C,and%20spelling%20phonetically) [[22]](https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex#:~:text=Levenshtein%20distance%20looks%20at%20two,for%20missing%20or%20substituted%20letters) fuzzy search - Levenshtein distance based methods Vs Soundex - Stack Overflow

<https://stackoverflow.com/questions/42013/levenshtein-distance-based-methods-vs-soundex>

[[19]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=PostgreSQL%20Fuzzy%20Text%20Searching) [[20]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=This%20is%20where%20fuzzy%20text,database%20something%20like%20the%20above) [[21]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=PostgreSQL%20Fuzzy%20Text%20Searching) [[24]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Performance%20Considerations) [[25]](https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/#:~:text=Using%20the%20built%20in%20,Solr%2C%20Elastic%20Search) Randall Degges - Easy Fuzzy Text Searching With PostgreSQL

<https://www.rdegges.com/2013/easy-fuzzy-text-searching-with-postgresql/>

[[23]](https://www.gsccca.org/docs/deed-lien-plat-documents/gsccca_indexing_standards_01_01_2018.pdf?sfvrsn=a850151f_2#:~:text=,Clerk) [PDF] Indexing Standards for Real and Personal Property Records for the ...

<https://www.gsccca.org/docs/deed-lien-plat-documents/gsccca_indexing_standards_01_01_2018.pdf?sfvrsn=a850151f_2>

[[26]](https://arxiv.org/html/2304.03464v3#:~:text=Record%20Linkage%20with%20Multimodal%20Contrastive,because%20there%20are%20visually) Record Linkage with Multimodal Contrastive Learning - arXiv

<https://arxiv.org/html/2304.03464v3>

[[27]](https://arxiv.org/html/2409.04117v1#:~:text=Confidence,develop%20ConfBERT%2C%20a%20BERT) Confidence-Aware Document OCR Error Detection - arXiv

<https://arxiv.org/html/2409.04117v1>

[[28]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=match%20at%20L528%20OCR%20and,into%20the%20spool%20directory%20for) [[29]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=match%20at%20L573%20of%20the,lasts%20for%20approximately%20four%20years) [[33]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=OCR%20and%20recognition%20engines,into%20the%20spool%20directory%20for) [[34]](https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf#:~:text=of%20the%20computer%20capacity%20is,lasts%20for%20approximately%20four%20years) Microsoft Word - ts02h\_wouters\_3841.doc

<https://fig.net/resources/proceedings/fig_proceedings/fig2010/papers/ts02h/ts02h_wouters_3841.pdf>

[[30]](https://lhncbc.nlm.nih.gov/LHC-publications/PDF/pub2001010.pdf#:~:text=employed%20in%20MARS%20includes%20a,This%20is) Pattern matching techniques for correcting low confidence OCR words in a known context

<https://lhncbc.nlm.nih.gov/LHC-publications/PDF/pub2001010.pdf>

[[31]](https://en.wikipedia.org/wiki/Grantor%E2%80%93grantee_index#:~:text=Grantor%E2%80%93grantee%20index%20,name%20of%20the%20parties) Grantor–grantee index - Wikipedia

<https://en.wikipedia.org/wiki/Grantor%E2%80%93grantee_index>

[[32]](https://www.afxllc.com/all-articles/revolutionizing-title-search-ai-title-search/#:~:text=5%20Ways%20AI%20Title%20Search,reports%20for%20real%20estate%20efficiency) 5 Ways AI Title Search Solutions Are Revolutionizing Real Estate

<https://www.afxllc.com/all-articles/revolutionizing-title-search-ai-title-search/>

[[35]](https://dna.firstam.com/insights-blog/how-machine-learning-and-artificial-intelligence-have-expedited-real-estate-transactions#:~:text=How%20Machine%20Learning%20and%20Artificial,expedite%20the%20time%20it) How Machine Learning and Artificial Intelligence Have Expedited ...

<https://dna.firstam.com/insights-blog/how-machine-learning-and-artificial-intelligence-have-expedited-real-estate-transactions>