Analysis of Different Pdf Processing Techniques

1. Time (s) - Execution Time for Extraction

- **Output Alignment**: The times reflect the computational overhead of each method.
  - **PyMuPDF (0.36s)**: Fastest (tied with PyPDF2). Uses fitz for efficient block-based text extraction, table finding, and link parsing—no heavy conversions or API calls.
  - **pdfplumber (1.06s)**: Slightly slower due to detailed table extraction (extract_tables()) and hyperlink processing, which iterates over pages and formats rows.
  - **PyPDF2 (0.35s)**: Very fast; simple extract_text() with basic regex-like table formatting (splitting on spaces). Minimal overhead.
  - **Tesseract (5.37s)**: Moderately slow, as expected—converts PDF to images (pdf2image), preprocesses with OpenCV (thresholding, dilation), detects tables via contours, and runs OCR (pytesseract). This is I/O and compute-intensive for a 4-page PDF.
  - **PageIndex (14.26s)**: Slowest, because extract_text_from_pdf() (implied in the truncated code) likely feeds into _build_tree_structure(), which makes multiple OpenAI API calls (e.g., to identify sections, generate titles/summaries). API latency dominates.
  - **Docling (2.24s)**: Moderate; uses unstructured.partition_pdf() for element partitioning (text, tables, images), which involves layout analysis but no external APIs during extraction.
- **Accuracy Assessment**: These times are realistic for a small PDF like "Test.pdf". Tesseract and PageIndex's higher times match their code's complexity (OCR/CV vs. LLM calls). If your run included network latency for OpenAI (in PageIndex), it explains the 14s.

2. Memory (MB) and Peak Mem (MB) - Resource Consumption

- **Output Alignment**: Memory usage correlates with data structures and dependencies.
  - **PyMuPDF (12.62MB / 3.90MB)**: Low; fitz loads PDF in memory but processes pages iteratively, with minimal buffers for blocks/tables.
  - **pdfplumber (25.62MB / 12.95MB)**: Higher due to storing extracted tables as lists of rows and processing hyperlinks/words per page.
  - **PyPDF2 (2.09MB / 2.63MB)**: Lowest; lightweight PdfReader with string-based processing—no complex objects.
  - **Tesseract (38.18MB / 128.76MB)**: High peak due to loading images (NumPy arrays from OpenCV), contours, and masks during table detection/preprocessing. Peak spikes during CV operations.

    - **PageIndex (9.09MB / 9.39MB)**: Low; mostly text strings and tree nodes (PageNode dataclass), but API calls don't add much memory (responses are small).
    - **Docling (130.12MB / 95.55MB)**: Highest; unstructured loads ML models (e.g., for layout detection) and partitions into elements (Text/Table/Image), which can consume significant RAM even for small PDFs.
- **Accuracy Assessment**: Spot-on. Docling's high memory is a known issue with unstructured (it uses heavy deps like Detectron2 internally). Tesseract's peak aligns with image handling. Low-memory libs like PyPDF2 perform as expected.

3. CPU % - Processor Utilization

- **Output Alignment**: Varies based on compute vs. I/O bound operations.
    - **PyMuPDF (46.30%)**: Moderate; efficient C-based parsing.
    - **pdfplumber (50.15%)**: Similar, with more CPU for table formatting.
    - **PyPDF2 (47.40%)**: Balanced; simple string ops.
    - **Tesseract (5.50%)**: Low; mostly I/O-bound (image conversion, OCR calls), with CV ops not maxing CPU on a multi-core system.
    - **PageIndex (9.49%)**: Low; API calls are network-bound, not CPU-intensive.
    - **Docling (49.20%)**: Moderate-high; ML partitioning in unstructured uses CPU for inference.
- **Accuracy Assessment**: Reasonable. Low CPU for Tesseract/PageIndex makes sense if waits dominate. Your ASUS VivoBook likely has moderate cores, explaining no 100% spikes.

4. Accuracy % - RAG Retrieval Quality

- **Output Alignment**: This is the averaged score from evaluate_pipeline_accuracy ((accuracy + completeness + relevance) / 3 per query, averaged over tests). Queries focus on tables/links, evaluated via retrieved chunks → GPT-generated answer → LLM comparison to ground_truth.
    - **PyMuPDF (53.33%)**: Highest; good table detection (find_tables()) and formatting (pipe-separated rows) likely retrieves bullet metrics as "tables," leading to better matches despite ground_truth mismatch.
    - **pdfplumber (35.00%)**: Decent; strong table extraction, but perhaps over-formats lists, reducing relevance.
    - **PyPDF2 (36.67%)**: Similar to pdfplumber; basic space-based table formatting captures bullets okay.
    - **Tesseract (7.50%)**: Lowest; OCR can introduce errors (e.g., misread bullets as text), and table detection via contours might fail on non-grid structures, leading to poor retrieval/generation.
    - **PageIndex (25.83%)**: Low-moderate; tree-based search (_search_tree with GPT relevance scoring) is innovative but may not prioritize "table" chunks well, especially without explicit table handling.

    - **Docling (14.17%)**: Low; unstructured partitions bullets as Text (not Table), so queries about tables retrieve generic text, scoring poorly against ground_truth.
- **Accuracy Assessment**: Credible, given the PDF's bullet-heavy structure (no true tables). Higher scores for native parsers (PyMuPDF) vs. OCR (Tesseract) or structured (Docling) make sense. The overall low scores (~7-53%) suggest the ground_truth mismatch inflated penalties—if aligned (e.g., to RAG metrics), scores could rise 20-30%, but relative order would stay similar.

Overall Analysis

- **Best Library**: **PyMuPDF** wins overall—balanced fast/low-resource with highest accuracy for this text/link-heavy PDF. We can use it for production unless we need OCR (Tesseract for scanned PDFs) or advanced structuring (Docling for complex layouts).