Contents

1 CouchDB File Reading Module

1.1 Overview

The `couchdb_file.py` module provides a UDF-friendly alternative to `read_files()` for processing annotated text files stored as CouchDB attachments in PySpark partitions. It preserves database metadata throughout the extraction pipeline.

1.2 Key Features

- **Drop-in replacement** for `file.py` when working with CouchDB
- **Metadata preservation**: Tracks `doc_id`, `attachment_name`, and `db_name` (ingest_db_name)
- **PySpark optimized**: Designed for use with `mapPartitions()` for distributed processing

- **Compatible**: Works seamlessly with existing `parse_annotated()` and `group_paragraphs()` functions
- **Efficient**: Processes partitions in parallel with minimal overhead

1.3 Architecture

1.3.1 Class Hierarchy

```
FileObject (Abstract)
├── File (file.py)
│   └── read_line() → Line
└── CouchDBFile (couchdb_file.py)
    └── read_line() → Line (with CouchDB metadata)

Line (line.py)
├── Standard fields: filename, line_number, page_number, etc.
└── Optional CouchDB fields: doc_id, attachment_name, db_name
```

1.3.2 Data Flow

```
CouchDB
  ↓ (CouchDBConnection.load_distributed)
DataFrame[doc_id, attachment_name, value]
  ↓ (mapPartitions + read_couchdb_partition)
Line objects
  ↓ (parse_annotated)
Paragraph objects
  ↓ (group_paragraphs)
Taxon objects with CouchDB metadata
```

1.4 Classes

1.4.1 CouchDBFile

File-like object for CouchDB attachment content.

Constructor:

```
CouchDBFile(
    content: str,
    doc_id: str,
    attachment_name: str,
    db_name: str
)
```

Methods: - `read_line()`: Iterator yielding `Line` objects

Properties: - doc_id: CouchDB document ID - attachment_name: Attachment filename - db_name: Database name (ingest_db_name) - filename: Composite identifier “db_name/doc_id/attachment_name” - line_number: Current line number - page_number: Current page number - empirical_page_number: Printed page number from document

1.4.2 Line Class with CouchDB Support

The Line class now includes optional CouchDB metadata fields that are automatically populated when the Line is created from a CouchDBFile.

Standard Properties: - line: Text content - filename: File identifier (or composite “db_name/doc_id/attachment_name” for CouchDB) - line_number: Line number - page_number: Page number - empirical_page_number: Printed page number - contains_start(): Check for annotation start marker [@ - end_label(): Get label from annotation end marker #Label*]

Optional CouchDB Properties (populated when created from CouchDBFile): - doc_id: CouchDB document ID (Optional[str]) - attachment_name: Attachment filename (Optional[str]) - db_name: Database name - ingest_db_name (Optional[str])

1.5 Functions

1.5.1 read_couchdb_partition()

Process CouchDB rows in a PySpark partition.

```
def read_couchdb_partition(
    partition: Iterator[Row],
    db_name: str
) -> Iterator[Line]
```

Args: - partition: Iterator of PySpark Rows with columns: - doc_id: CouchDB document ID - attachment_name: Attachment filename - value: Text content - db_name: Database name for metadata tracking

Returns: - Iterator of Line objects with metadata

Usage in PySpark:

```
df.rdd.mapPartitions(lambda part: read_couchdb_partition(part, "mycobank"))
```

1.5.2 read_couchdb_rows()

Process a list of CouchDB rows (non-distributed).

```
def read_couchdb_rows(
    rows: List[Row],
    db_name: str
) -> Iterator[Line]
```

Usage:

```
rows = df.collect()
lines = read_couchdb_rows(rows, "mycobank")
paragraphs = parse_annotated(lines)
```

1.5.3 read_couchdb_files_from_connection()

Complete pipeline from CouchDBConnection to Line objects.

```
def read_couchdb_files_from_connection(
    conn: CouchDBConnection,
    spark: SparkSession,
    db_name: str,
    pattern: str = "*.txt.ann"
) -> Iterator[Line]
```

Usage:

```
from skol_classifier.couchdb_io import CouchDBConnection
from couchdb_file import read_couchdb_files_from_connection

conn = CouchDBConnection("http://localhost:5984", "mycobank")
lines = read_couchdb_files_from_connection(conn, spark, "mycobank", "*.txt.ann")
paragraphs = parse_annotated(lines)
taxa = list(group_paragraphs(paragraphs))
```

1.6 Usage Examples

1.6.1 Example 1: Distributed Processing in PySpark

```
from pyspark.sql import SparkSession
from skol_classifier.couchdb_io import CouchDBConnection
from couchdb_file import read_couchdb_partition
from finder import parse_annotated, remove_interstitials
from taxon import group_paragraphs

def process_partition_to_taxa(partition, db_name):
    """Extract taxa from a partition of CouchDB rows."""
    lines = read_couchdb_partition(partition, db_name)
    paragraphs = parse_annotated(lines)
    filtered = remove_interstitials(paragraphs)
    taxa = group_paragraphs(filtered)
```

```
    for taxon in taxa:
        for para_dict in taxon.dictionaries():
            # Extract CouchDB metadata from composite filename
            parts = para_dict['filename'].split('/', 2)
            if len(parts) == 3:
                para_dict['db_name'] = parts[0]
                para_dict['doc_id'] = parts[1]
                para_dict['attachment_name'] = parts[2]
            yield para_dict

# Setup
spark = SparkSession.builder.appName("TaxonExtractor").getOrCreate()
conn = CouchDBConnection("http://localhost:5984", "mycobank", "user", "pass")

# Load from CouchDB
df = conn.load_distributed(spark, "*.txt.ann")

# Process in parallel
taxa_rdd = df.rdd.mapPartitions(
    lambda part: process_partition_to_taxa(part, "mycobank")
)

# Convert to DataFrame
from pyspark.sql.types import StructType, StructField, StringType

schema = StructType([
    StructField("serial_number", StringType(), False),
    StructField("db_name", StringType(), True),
    StructField("doc_id", StringType(), True),
    StructField("attachment_name", StringType(), True),
    StructField("label", StringType(), False),
    StructField("body", StringType(), False),
])

taxa_df = taxa_rdd.toDF(schema)
taxa_df.write.parquet("output/taxa.parquet")
```

1.6.2 Example 2: Local Processing (Testing)

```
from couchdb_file import read_couchdb_rows
from finder import parse_annotated
from taxon import group_paragraphs
from pyspark.sql import Row
```

```
# Simulate CouchDB data
rows = [
    Row(
        doc_id="doc123",
        attachment_name="article.txt.ann",
        value="[@Species nova Author 1999#Nomenclature*]\n[@Description text her
    )
]

# Process
lines = read_couchdb_rows(rows, "mycobank")
paragraphs = parse_annotated(lines)
taxa = list(group_paragraphs(paragraphs))

# Access metadata
for taxon in taxa:
    for para_dict in taxon.dictionaries():
        print(f"From: {para_dict['filename']}")
        print(f"Label: {para_dict['label']}")
        print(f"Text: {para_dict['body'][:100]}...")
```

1.6.3 Example 3: Integration with CouchDBConnection

```
from skol_classifier.couchdb_io import CouchDBConnection
from couchdb_file import read_couchdb_files_from_connection
from finder import parse_annotated
from taxon import group_paragraphs

# Connect
conn = CouchDBConnection(
    couchdb_url="http://localhost:5984",
    database="mycobank_docs",
    username="admin",
    password="secret"
)

# Load all annotated files
lines = read_couchdb_files_from_connection(
    conn=conn,
    spark=spark,
    db_name="mycobank",
    pattern="*.txt.ann"
)

# Extract taxa
```

```
paragraphs = parse_annotated(lines)
taxa = list(group_paragraphs(paragraphs))

print(f"Extracted {len(taxa)} taxa from CouchDB")
```

1.7 Metadata Tracking

1.7.1 Filename Format

CouchDB metadata is encoded in the `filename` property using the format:

`db_name/doc_id/attachment_name`

Example:

`mycobank/article_2023_001/fulltext.txt.ann`

Parsing:

```
parts = filename.split('/', 2)
db_name = parts[0]           # "mycobank"
doc_id = parts[1]            # "article_2023_001"
attachment_name = parts[2]   # "fulltext.txt.ann"
```

1.7.2 Metadata Flow Through Pipeline

```
# Step 1: Create Line
line = Line(...)
line.doc_id          # "doc123"
line.db_name         # "mycobank"
line.filename        # "mycobank/doc123/file.txt.ann"

# Step 2: Create Paragraph
paragraph = Paragraph(...)
paragraph.filename   # "mycobank/doc123/file.txt.ann"

# Step 3: Create Taxon
taxon = Taxon(...)
para_dict = taxon.dictionaries()[0]
para_dict['filename']  # "mycobank/doc123/file.txt.ann"
```

1.8 Testing

Run the test suite:

```
cd tests
python test_couchdb_file.py
```

Test coverage: - Basic CouchDBFile creation and reading - Annotated content parsing - Page number tracking - Metadata preservation - Partition reading - Integration with parse_annotated() - Full pipeline (Lines → Paragraphs → Taxa)

1.9 Comparison: file.py vs couchdb_file.py

Feature	file.py	couchdb_file.py
Input source	Local files	CouchDB attachments
Line class	`Line` (no CouchDB fields)	`Line` (with CouchDB fields populated)
Metadata	filename, line_number, page_number	+ doc_id, attachment_name, db_name
Filename format	Path string	"db_name/doc_id/attachment_name"
Use case	Local files, traditional pipeline	Distributed processing, database-backed
PySpark	Not optimized	Designed for mapPartitions

1.10 Best Practices

1. **Use distributed processing for large datasets**
   - Use `read_couchdb_partition()` with `mapPartitions()`
   - Avoid `collect()` on large DataFrames
2. **Preserve metadata throughout pipeline**
   - Parse composite filenames to extract CouchDB metadata
   - Include db_name, doc_id, attachment_name in output schema
3. **Efficient partition processing**
   - Process entire partitions in a single function
   - Avoid creating new connections per row
4. **Testing and debugging**
   - Use `read_couchdb_rows()` for local testing
   - Verify metadata preservation at each stage
5. **Schema design**
   - Include CouchDB metadata columns in output DataFrames
   - Use composite filenames for traceability

1.11 See Also

- EXTRACTING_TAXON_OBJECTS.md - Full extraction guide

- examples/extract_taxa_from_couchdb.py - Complete examples
- skol_classifier/couchdb_io.py - CouchDBConnection class
- file.py - Original file reading module
- line.py - Base Line class