Contents

1 Distributed CouchDB Processing with PySpark

1.1 Overview

The SKOL Classifier uses **distributed UDFs** (User-Defined Functions) to process CouchDB documents in parallel across a Spark cluster. This approach:

1. **Avoids loading all data on the driver** - Only document metadata is collected centrally
2. **Distributes I/O operations** - Each Spark worker connects to CouchDB independently
3. **Scales horizontally** - Processing speed increases with cluster size
4. **Handles large datasets** - Can process millions of documents efficiently

1.2 Architecture

1.2.1 Traditional Approach (Non-Distributed)

```
Driver Node:
  1. Connect to CouchDB
  2. Download ALL attachments to driver
  3. Create DataFrame from collected data
  4. Distribute data to workers

Problem: Driver becomes bottleneck and may run out of memory
```

1.2.2 Distributed Approach (Current Implementation)

```
Driver Node:
  1. Connect to CouchDB
  2. Get list of (doc_id, attachment_name) pairs (lightweight metadata only)
  3. Create DataFrame with one row per attachment
     - If doc has multiple .txt files, creates multiple rows
  4. Distribute rows to workers

Worker Nodes (parallel):
  1. Each worker connects to CouchDB
  2. Each fetches only its assigned attachments
  3. Processing happens in parallel
  4. Results written back in parallel (one .ann per .txt)

Benefit: Work distributed across cluster, no bottleneck
```

1.3 How It Works

1.3.1 Reading from CouchDB

```
from skol_classifier import SkolClassifier

classifier = SkolClassifier()

# This uses distributed UDFs under the hood
df = classifier.load_from_couchdb(
    couchdb_url="http://localhost:5984",
    database="documents",
    username="admin",
    password="password"
)

# df is a Spark DataFrame where:
# - (doc_id, attachment_name) pairs were collected on driver
# - If a document has multiple .txt attachments, it has multiple rows
# - Content is fetched lazily by workers when needed
# - Each worker connects to CouchDB independently
```

What happens: 1. Driver iterates through all documents in CouchDB 2. For each document, loops through ALL attachments 3. Creates one row per attachment matching pattern (e.g., “*.txt”) 4. Driver creates DataFrame with (doc_id, attachment_name) columns 5. When the DataFrame is processed: - Spark distributes rows to workers - Each worker partition connects to CouchDB ONCE - Worker fetches all its assigned attachments using same connection - Processing happens in parallel

1.3.2 Writing to CouchDB

```
# predictions is a Spark DataFrame with results
results = classifier.save_to_couchdb(
    predictions,
    couchdb_url="http://localhost:5984",
    database="documents",
    username="admin",
    password="password"
)
```

What happens: 1. DataFrame is distributed across workers 2. Each worker runs a UDF that: - Connects to CouchDB - Saves its assigned documents - Returns success/failure status 3. Results are collected on driver

1.4 UDF Functions

1.4.1 Fetch Attachment UDF

```
from skol_classifier.couchdb_io import create_fetch_attachment_udf

# Create a UDF that runs on workers
fetch_udf = create_fetch_attachment_udf(
    couchdb_url="http://localhost:5984",
    database="documents",
    username="admin",
    password="password"
)

# Apply to DataFrame
df_with_content = doc_ids_df.withColumn(
    "value",
    fetch_udf(col("doc_id"), col("attachment_name"))
)

# When this DataFrame is evaluated:
# - Each worker connects to CouchDB
# - Fetches content for its assigned rows
# - All in parallel
```

1.4.2 Save Attachment UDF

```
from skol_classifier.couchdb_io import create_save_attachment_udf

# Create a UDF that saves on workers
save_udf = create_save_attachment_udf(
    couchdb_url="http://localhost:5984",
    database="documents",
    username="admin",
    password="password",
    suffix=".ann"
)

# Apply to DataFrame
result_df = content_df.withColumn(
    "success",
    save_udf(col("doc_id"), col("attachment_name"), col("content"))
)

# When evaluated:
# - Each worker saves its assigned documents
```

```
# - All writes happen in parallel
```

1.5 Multiple Attachments Per Document

The system is designed to handle documents with multiple text attachments:

1.5.1 Example Scenario

```
{
  "_id": "article_001",
  "_attachments": {
    "abstract.txt": {...},
    "methods.txt": {...},
    "results.txt": {...}
  }
}
```

1.5.2 How It's Processed

1. **Driver Phase**: Creates 3 rows in DataFrame:

   ```
   (article_001, abstract.txt)
   (article_001, methods.txt)
   (article_001, results.txt)
   ```

2. **Worker Phase**: Each attachment processed independently:
   - Fetches content for each .txt file
   - Classifies each independently
   - Saves as separate .ann files

3. **Result**: Document has 6 attachments:

   ```
   {
     "_id": "article_001",
     "_attachments": {
       "abstract.txt": {...},
       "abstract.txt.ann": {...},    // NEW
       "methods.txt": {...},
       "methods.txt.ann": {...},     // NEW
       "results.txt": {...},
       "results.txt.ann": {...}      // NEW
     }
   }
   ```

1.5.3 Benefits

- **Independent processing**: Each text file classified separately
- **Parallel processing**: Different .txt files can be on different workers
- **Scalability**: Documents with many attachments automatically parallelized
- **Flexibility**: Pattern matching allows selective processing

1.6 Performance Considerations

1.6.1 Parallelism

The number of parallel operations depends on: - **Number of Spark partitions**: More partitions = more parallelism - **Number of worker cores**: Each core can run one task - **DataFrame size**: Need enough rows to distribute

```
# Check partitions
print(f"Partitions: {df.rdd.getNumPartitions()}")

# Repartition for better parallelism
df = df.repartition(100)  # 100 parallel tasks
```

1.6.2 CouchDB Connection Pooling

Each worker creates its own CouchDB connection. For better performance:

1. **Use connection pooling** in CouchDB
2. **Increase CouchDB max connections**
3. **Use CouchDB cluster** for distributed database

```
# CouchDB configuration
[httpd]
max_connections = 1000

[cluster]
n = 3  # 3-node cluster
```

1.6.3 Memory Usage

With distributed UDFs: - **Driver memory**: Only stores document IDs (small) - **Worker memory**: Each stores only its partition's data - **Total capacity**: Sum of all worker memory

1.6.4 Network Considerations

- **Locality**: Place Spark workers close to CouchDB servers
- **Bandwidth**: Ensure sufficient network bandwidth
- **Latency**: Low latency between workers and CouchDB is crucial

1.7 Scaling Examples

1.7.1 Small Dataset (< 1000 documents)

```
# Simple approach works fine
classifier = SkolClassifier()
predictions = classifier.predict_from_couchdb(
    couchdb_url="http://localhost:5984",
    database="small_db"
)
```

1.7.2 Medium Dataset (1,000 - 100,000 documents)

```
# Increase parallelism
classifier = SkolClassifier()

# Load data
df = classifier.load_from_couchdb(
    couchdb_url="http://localhost:5984",
    database="medium_db"
)

# Repartition for better parallelism
df = df.repartition(50)

# Process
# ... (processing steps)
```

1.7.3 Large Dataset (> 100,000 documents)

```
# Use Spark cluster with multiple workers
classifier = SkolClassifier()

# Load data
df = classifier.load_from_couchdb(
    couchdb_url="http://couchdb-cluster:5984",
    database="large_db"
)

# Optimize partitioning
```

```python
# Rule of thumb: 2-4x number of cores
num_cores = 100  # total cores in cluster
df = df.repartition(num_cores * 3)

# Enable caching if processing multiple times
df.cache()

# Process in batches if needed
batch_size = 10000
total_docs = df.count()

for offset in range(0, total_docs, batch_size):
    batch_df = df.limit(batch_size).offset(offset)
    # Process batch...
```

1.8 Monitoring

1.8.1 Track Progress

```python
from pyspark import TaskContext

def fetch_with_progress(doc_id, attachment_name):
    context = TaskContext.get()
    if context:
        partition_id = context.partitionId()
        print(f"Partition {partition_id}: Processing {doc_id}")

    # ... fetch logic ...
```

1.8.2 Monitor Spark UI

The Spark UI (http://localhost:4040) shows: - Number of active tasks - Task duration - Data shuffle - Failed tasks

1.8.3 CouchDB Metrics

Monitor CouchDB:

```bash
# Check active connections
curl http://admin:password@localhost:5984/_active_tasks

# Monitor database stats
curl http://admin:password@localhost:5984/database/_info
```

1.9 Error Handling

1.9.1 Handling Failed Fetches

```
# The UDF filters out errors automatically
df = classifier.load_from_couchdb(...)

# Check for errors in logs
# Failed fetches are logged but don't stop processing
```

1.9.2 Retry Failed Saves

```
results = classifier.save_to_couchdb(predictions, ...)

# Find failed saves
failed = [r for r in results if not r['success']]

if failed:
    print(f"Retrying {len(failed)} failed saves...")
    # Retry logic...
```

1.9.3 Handling CouchDB Downtime

```
def fetch_with_retry(doc_id, attachment_name, max_retries=3):
    for attempt in range(max_retries):
        try:
            # Connect and fetch...
            return content
        except Exception as e:
            if attempt == max_retries - 1:
                return f"ERROR: {str(e)}"
            time.sleep(2 ** attempt)  # Exponential backoff
```

1.10 Advanced Patterns

1.10.1 Process and Save in One UDF

Avoid passing large data through Spark by combining operations:

```
from skol_classifier.couchdb_io import create_process_and_save_udf

# This UDF reads, processes, and saves all on the worker
process_save_udf = create_process_and_save_udf(
    couchdb_url="http://localhost:5984",
    database="documents",
    username="admin",
```

```
    password="password"
)

# Apply to document IDs only
doc_ids_df.withColumn(
    "success",
    process_save_udf(col("doc_id"), col("attachment_name"), col("processed"))
)
```

1.10.2 Batch Processing

```
# Process in smaller batches to control memory
batch_size = 1000

# Get total count
total = df.count()

for i in range(0, total, batch_size):
    batch = df.offset(i).limit(batch_size)

    # Process batch
    predictions = classifier.predict_from_couchdb_df(batch)

    # Save batch
    classifier.save_to_couchdb(predictions, ...)
```

1.10.3 Distributed Cache

Share expensive objects across UDF calls:

```
from pyspark import SparkContext

def get_couchdb_connection(url, username, password):
    # Use broadcast variable to share connection config
    if not hasattr(get_couchdb_connection, 'server'):
        get_couchdb_connection.server = couchdb.Server(url)
        if username:
            get_couchdb_connection.server.resource.credentials = (username, pass

    return get_couchdb_connection.server
```

1.11 Best Practices

1. **Partition Wisely**: Use 2-4x the number of cores
2. **Monitor Resource Usage**: Watch memory and network

3. **Handle Errors Gracefully**: Don't fail entire job on one document
4. **Use CouchDB Views**: Create views for faster document queries
5. **Enable Spark Caching**: Cache DataFrames used multiple times
6. **Test Scaling**: Start small and scale up gradually
7. **Log Appropriately**: Log errors but avoid excessive logging

1.12 Troubleshooting

1.12.1 Issue: Slow Processing

Symptoms: Long execution time, low CPU usage

Solutions: - Increase number of partitions: `df.repartition(N)` - Check Spark UI for task distribution - Verify CouchDB isn't bottlenecked

1.12.2 Issue: Out of Memory

Symptoms: Workers crash, OOM errors

Solutions: - Reduce partition size: `df.repartition(more_partitions)` - Process in batches - Increase worker memory: `--executor-memory 8g`

1.12.3 Issue: Connection Timeouts

Symptoms: Many failed fetches, timeout errors

Solutions: - Increase CouchDB timeout settings - Add retry logic to UDFs - Check network connectivity - Use CouchDB connection pooling

1.12.4 Issue: Uneven Processing

Symptoms: Some workers idle while others work

Solutions: - Repartition DataFrame: `df.repartition("doc_id")` - Check data skew (some docs much larger) - Use `coalesce()` to reduce partitions if needed

1.13 See Also

- COUCHDB_INTEGRATION.md - Basic CouchDB usage
- examples/couchdb_usage.py - Code examples
- Apache Spark UDF Documentation
- CouchDB Performance Guide