````
# Memory SDK Options - Complete Reference

This document provides a comprehensive reference for all available options when
working with the Memory SDK in LangPy.

## Table of Contents

1. [Memory Creation](#memory-creation)
2. [MemorySettings Configuration](#memorysettings-configuration)
3. [Memory Operations](#memory-operations)
4. [Filter System](#filter-system)
5. [Backend Options](#backend-options)
6. [Embedding Models](#embedding-models)
7. [Examples](#examples)

## Memory Creation

### Factory Method (Recommended)

```python
from sdk import memory

# Create a memory factory
mem_factory = memory()

# Create memory with options
mem_instance = mem_factory.create(
    name="my_memory",                                  # Memory name (required)
    backend="faiss",                                   # Vector store backend
    dsn="postgresql://user:pass@localhost:5432/db",    # PostgreSQL DSN (for
pgvector)
    embedding_model="openai:text-embedding-3-large",   # Embedding model
    api_key="sk-...",                                  # API key for embedding
model
    **kwargs                                           # Additional settings
)
```

### Direct Creation Methods

```python
# Using create_memory function directly
from memory import create_memory
mem = create_memory(
    name="default",
````
````

```
    backend="faiss",
    dsn=None,
    embed_model="openai:text-embedding-3-large",
    chunk_max_length=10000,
    chunk_overlap=256,
    **kwargs
)

# Using MemoryInterface directly
from sdk.memory_interface import MemoryInterface
from memory import MemorySettings

settings = MemorySettings(
    name="my_memory",
    store_backend="faiss",
    embed_model="openai:text-embedding-3-large"
)

mem = MemoryInterface(
    async_backend=None,
    sync_backend=None,
    settings=settings
)
```

## MemorySettings Configuration

### Storage Backend Options

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `store_backend` | `str` | `"faiss"` | Vector store backend: `"faiss"`, `"pgvector"`, or `"docling"` |
| `store_uri` | `str` | `None` | URI for vector store (file path for FAISS, DSN for pgvector) |

### Parser Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `parser_enable_ocr` | `bool` | `True` | Enable OCR for images |
| `parser_ocr_languages` | `List[str]` | `["eng"]` | OCR languages to use |
| `parser_max_file_size` | `int` | `52428800` | Maximum file size for parsing (50MB) |

```
### Chunker Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `chunk_max_length` | `int` | `10000` | Maximum length of each chunk |
| `chunk_overlap` | `int` | `256` | Overlap between consecutive chunks |

### Embedding Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `embed_model` | `str` | `"openai:text-embedding-3-large"` | Embedding model to
use |

### Reranking Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `enable_reranking` | `bool` | `True` | Enable cross-encoder reranking for
better results |
| `reranker_model` | `str` | `"BAAI/bge-reranker-large"` | Cross-encoder model
for reranking |
| `rerank_top_k` | `int` | `20` | Number of candidates to rerank |

### Hybrid Search Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `enable_hybrid_search` | `bool` | `True` | Enable hybrid search (ANN + BM25) |
| `hybrid_weight` | `float` | `0.7` | Weight for vector similarity vs keyword
matching (0-1) |

### Search Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `default_k` | `int` | `5` | Number of results to return by default |
| `similarity_threshold` | `float` | `0.7` | Similarity threshold for search |

### Metadata Configuration

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `include_source` | `bool` | `True` | Include source in metadata |
| `include_timestamp` | `bool` | `True` | Include timestamp in metadata |
```

| `include_tokens` | `bool` | `True` | Include token count in metadata |

### Memory Identification

| Option | Type | Default | Description |
|--------|------|---------|-------------|
| `name` | `str` | `"default"` | Memory name for identification |

## Memory Operations

### Upload Operation

```python
job_id = await mem.upload(
    content="text, Path, or bytes",           # Document content (required)
    source="optional_source_id",              # Source identifier (optional)
    custom_metadata={"key": "value"},         # Custom metadata (optional)
    progress_callback=callback_function,      # Progress callback (optional)
    background=True                           # Background processing (optional)
)
```

**Parameters:**
- `content`: Document content as string, bytes, or Path object
- `source`: Optional source identifier for filtering
- `custom_metadata`: Additional metadata to attach to the document
- `progress_callback`: Function to receive progress updates
- `background`: Whether to process in background (True) or wait for completion (False)

### Query Operation

```python
results = await mem.query(
    query="search text",                      # Search query (required)
    k=5,                                      # Number of results (optional)
    source="filter_source",                   # Filter by source (optional)
    min_score=0.7                             # Minimum similarity score (optional)
)
```

**Parameters:**
- `query`: Search query text
- `k`: Number of results to return (overrides default_k)
- `source`: Filter results by source

```
- `min_score`: Minimum similarity score threshold

### Metadata Operations

#### Get by Metadata
```python
results = await mem.get_by_metadata(
    metadata_filter={"key": "value"},       # Metadata filter (required)
    k=10                                    # Number of results (optional)
)
```

#### Update Metadata
```python
updated_count = await mem.update_metadata(
    updates={"new_key": "new_value"},       # Metadata updates (required)
    source="source_filter",                 # Source filter (optional)
    metadata_filter={"key": "value"}        # Additional filters (optional)
)
```

#### Delete by Filter
```python
deleted_count = await mem.delete_by_filter(
    source="source_filter",                 # Source filter (optional)
    metadata_filter={"key": "value"}        # Metadata filter (optional)
)
```

### Utility Operations

#### Get Job Status
```python
status = await mem.get_job_status("job_id")
```

#### Get Memory Statistics
```python
stats = await mem.get_stats()
```

## Filter System

### Simple Metadata Filter
```python
```
```

```
# Simple key-value filter
metadata_filter = {"category": "important", "type": "document"}
```

### FilterExpression
```python
from memory import FilterExpression

# Create filter expression
filter_expr = FilterExpression(
    field="category",
    operator="eq",
    value="important"
)
```

**Available Operators:**
- `"eq"`: Equal to
- `"neq"`: Not equal to
- `"in"`: In list
- `"nin"`: Not in list
- `"gt"`: Greater than
- `"gte"`: Greater than or equal
- `"lt"`: Less than
- `"lte"`: Less than or equal
- `"contains"`: Contains substring

### CompoundFilter
```python
from memory import CompoundFilter, FilterExpression

# AND filter
and_filter = CompoundFilter(
    and_conditions=[
        FilterExpression(field="category", operator="eq", value="important"),
        FilterExpression(field="type", operator="eq", value="document")
    ]
)

# OR filter
or_filter = CompoundFilter(
    or_conditions=[
        FilterExpression(field="category", operator="eq", value="important"),
        FilterExpression(field="category", operator="eq", value="critical")
    ]
```

```
)
```

## Backend Options

### FAISS (Default)
```python
mem = memory().create(
    name="my_memory",
    backend="faiss"
)
```

**Features:**
- Local vector storage
- Fast similarity search
- No external dependencies
- Automatic index management

### PGVector
```python
mem = memory().create(
    name="my_memory",
    backend="pgvector",
    dsn="postgresql://user:pass@localhost:5432/db"
)
```

**Features:**
- PostgreSQL with pgvector extension
- Persistent storage
- ACID compliance
- Scalable for large datasets

### Docling
```python
mem = memory().create(
    name="my_memory",
    backend="docling"
)
```

**Features:**
- Document-based storage
- Specialized for document processing

````
- Advanced parsing capabilities

## Embedding Models

### OpenAI Models
```python
# Latest and most capable
embed_model="openai:text-embedding-3-large"

# Smaller, faster
embed_model="openai:text-embedding-3-small"

# Legacy model
embed_model="openai:text-embedding-ada-002"
```

### HuggingFace Models
```python
# Any HuggingFace embedding model
embed_model="sentence-transformers/all-MiniLM-L6-v2"
embed_model="sentence-transformers/all-mpnet-base-v2"
```

### Custom Models
```python
# Custom embedding provider
embed_model="custom:my-embedding-model"
```

## Examples

### Basic Usage
```python
from sdk import memory

# Create memory
mem = memory().create("my_notes")

# Upload document
job_id = await mem.upload("path/to/document.pdf")

# Query memory
results = await mem.query("search query", k=10)
```
````

### Advanced Configuration
```python
from sdk import memory
from memory import MemorySettings

# Create with custom settings
settings = MemorySettings(
    name="advanced_memory",
    store_backend="pgvector",
    store_uri="postgresql://user:pass@localhost:5432/db",
    embed_model="openai:text-embedding-3-large",
    chunk_max_length=8000,
    chunk_overlap=512,
    enable_reranking=True,
    reranker_model="BAAI/bge-reranker-large",
    enable_hybrid_search=True,
    hybrid_weight=0.8,
    similarity_threshold=0.75
)

mem = memory().create(
    name="advanced_memory",
    backend="pgvector",
    dsn="postgresql://user:pass@localhost:5432/db",
    chunk_max_length=8000,
    chunk_overlap=512,
    enable_reranking=True,
    enable_hybrid_search=True,
    hybrid_weight=0.8
)
```

### Complete Workflow
```python
from sdk import memory
import asyncio

async def main():
    # Create memory
    mem = memory().create(
        name="document_storage",
        backend="faiss",
        embedding_model="openai:text-embedding-3-large"
    )
```

```
    # Upload documents
    job_id1 = await mem.upload("document1.pdf", source="reports")
    job_id2 = await mem.upload("document2.txt", source="notes")

    # Wait for processing
    while True:
        status = await mem.get_job_status(job_id1)
        if status and status["status"] == "completed":
            break
        await asyncio.sleep(1)

    # Query memory
    results = await mem.query("important findings", k=5)

    # Filter by source
    report_results = await mem.query("analysis", source="reports")

    # Update metadata
    await mem.update_metadata(
        updates={"reviewed": True},
        source="reports"
    )

    # Get statistics
    stats = await mem.get_stats()
    print(f"Total documents: {stats['total_documents']}")
    print(f"Total chunks: {stats['total_chunks']}")

asyncio.run(main())
```

### Sync Operations
```python
from sdk import memory

# All operations have sync versions
mem = memory().create("sync_memory")

# Sync upload
job_id = mem.upload_sync("document.pdf")

# Sync query
results = mem.query_sync("search query")

# Sync metadata operations
```

```
metadata_results = mem.get_by_metadata_sync({"type": "document"})
updated_count = mem.update_metadata_sync({"status": "processed"})
deleted_count = mem.delete_by_filter_sync(source="old_data")

# Sync utilities
status = mem.get_job_status_sync(job_id)
stats = mem.get_stats_sync()
```

## Environment Variables

| Variable | Description | Default |
|---|---|---|
| `LANGPY_PG_DSN` | PostgreSQL DSN for pgvector backend | None |
| `OPENAI_API_KEY` | OpenAI API key for embedding models | None |
| `ANTHROPIC_API_KEY` | Anthropic API key (if using Anthropic models) | None |

## Best Practices

1. **Choose the right backend**: Use FAISS for local development and small datasets, PGVector for production and large datasets
2. **Optimize chunk size**: Adjust `chunk_max_length` and `chunk_overlap` based on your document types
3. **Enable reranking**: Keep `enable_reranking=True` for better search quality
4. **Use hybrid search**: Keep `enable_hybrid_search=True` for comprehensive results
5. **Monitor performance**: Use `get_stats()` to monitor memory usage and performance
6. **Filter effectively**: Use metadata filters to narrow down search results
7. **Handle errors**: Always check job status when using background processing

## Troubleshooting

### Common Issues

1. **PostgreSQL connection failed**: Check DSN format and database availability
2. **Embedding API errors**: Verify API keys and model availability
3. **Chunk size too large**: Reduce `chunk_max_length` if hitting token limits
4. **Search quality poor**: Enable reranking and adjust similarity threshold
5. **Memory usage high**: Consider using smaller embedding models or reducing chunk overlap