Onyx Metadata Handling Guide

Table of Contents

Introduction & Overview

What is Metadata in Onyx?

Metadata in Onyx is additional information about documents that helps improve search quality and filtering capabilities. Think of it as "data about data" - it includes things like:

- **Document properties**: Author, creation date, department, document type
- **Tags**: Categories, topics, keywords assigned to documents
- **Source information**: Where the document came from (Confluence, Slack, etc.)
- **Custom fields**: Any additional information specific to your organization

Why is Metadata Important?

Metadata serves several crucial purposes in Onyx:

1. **Enhanced Search Quality**: Metadata is included in vector embeddings, helping the AI understand context better
2. **Filtering**: Users can filter search results by metadata fields (e.g., "show only documents by John Doe")
3. **Access Control**: Metadata helps determine who can see which documents
4. **Organization**: Documents can be categorized and organized using metadata

High-Level Metadata Flow

Here's how metadata flows through the Onyx system:

```
Document with Metadata → Chunking Process → Vector Database → Search
Results
       ↓                         ↓                  ↓              ↓
  {"author": "John",       Metadata gets      Stored in 3      Used for
   "tags": ["AI"]}        processed into     different        filtering &
                         2 formats          formats          ranking
```

The process involves:

1. **Document Ingestion**: Documents arrive with metadata from various sources
2. **Metadata Processing**: Metadata is cleaned and formatted for different uses
3. **Chunking**: Metadata is attached to document chunks in multiple formats
4. **Vector Database Storage**: Metadata is stored in Vespa for efficient search and filtering
5. **Search Integration**: Metadata is used for both vector search and filtering

Metadata Processing During Chunking

The Core Metadata Processing Function

The heart of metadata processing in Onyx is the `_get_metadata_suffix_for_document_index` function. This function takes raw metadata and creates two different representations:

Function Purpose: Convert metadata dictionary into search-friendly formats

Input Example:

```
metadata = {
    "author": "Jane Smith",
    "tags": ["python", "tutorial", "AI"],
    "department": "Engineering"
}
```

Output: Two different string representations:

1. **Semantic Version** (for AI understanding):

```
Metadata:
    author - Jane Smith
    tags - python, tutorial, AI
    department - Engineering
```

2. **Keyword Version** (for keyword search):

```
Jane Smith python tutorial AI Engineering
```

How Metadata Processing Works

Step 1: Filtering

- Some metadata keys are ignored (like "ignore_for_qa")
- Only relevant metadata is processed

Step 2: Format Conversion

- List values are converted to comma-separated strings

- All values are cleaned and normalized

Step 3: Two-Format Generation

- **Semantic format**: Natural language with keys and values
- **Keyword format**: Just the values concatenated

Configuration Options

SKIP_METADATA_IN_CHUNK: Environment variable that controls metadata inclusion

- `true`: Metadata is NOT included in chunks
- `false` (default): Metadata IS included in chunks

Token Limits: Metadata is subject to size limits

- Maximum 25% of chunk token limit
- If metadata is too large, it gets discarded to preserve content

Safety Measures

Content Preservation: Onyx ensures metadata doesn't overwhelm actual content

- Minimum 256 tokens reserved for actual content
- If metadata is too large, it's progressively removed
- Content always takes priority over metadata

Example of Size Limiting:

```python
# If metadata takes more than 25% of chunk size
if metadata_tokens >= chunk_limit * 0.25:
    # Metadata is discarded
    metadata_suffix_semantic = ""
    metadata_tokens = 0
```

Chunk Models & Metadata Storage

DocAwareChunk Model

Each chunk in Onyx contains several metadata-related fields:

```python
class DocAwareChunk:
    # Main content fields
    content: str
    blurb: str

    # Metadata fields
    metadata_suffix_semantic: str    # For vector embeddings
    metadata_suffix_keyword: str     # For keyword search
    # Document info
    source_document: Document
    title_prefix: str
```

Two Types of Metadata Suffixes

1. Semantic Metadata Suffix

- **Purpose**: Added to content before vector embedding
- **Format**: Natural language with keys and values
- **Example**: "Metadata:\n\tauthor - John Doe\n\ttags - python, AI"

2. Keyword Metadata Suffix

- **Purpose**: Added to content for keyword search
- **Format**: Space-separated values only
- **Example**: "John Doe python AI"

Metadata Inheritance

Regular Chunks: Get metadata from source document **Large Chunks**: Inherit metadata from constituent regular chunks **Mini Chunks**: Use same metadata as parent chunk

All chunks from the same document get identical metadata suffixes for consistency.

Token Budget Management

When creating chunks, Onyx calculates available space:

```
Available Tokens = Chunk Limit - Title Tokens - Metadata Tokens - Context Tokens
```

If space is insufficient (< 256 tokens), Onyx removes elements in this order:

1. Contextual RAG context
2. Title and metadata
3. Uses full chunk limit for content

Vector Database Indexing (Vespa)

Three Types of Metadata Storage

When chunks are indexed in Vespa, metadata is stored in three different ways:

1. `metadata` Field (JSON Storage)

- **Purpose**: Complete metadata preservation
- **Format**: JSON string
- **Example**: `{"author": "John Doe", "tags": ["python", "AI"]}`

- **Usage**: Reconstruction and display

2. `metadata_list` Field (Filterable Attributes)

- **Purpose**: Fast filtering and search
- **Format**: Array of key-value pairs
- **Example**: `["author|John Doe", "tags|python", "tags|AI"]`
- **Usage**: Efficient filtering in queries

3. `metadata_suffix` Field (Search Text)

- **Purpose**: Keyword search integration
- **Format**: Space-separated values
- **Example**: `"John Doe python AI"`
- **Usage**: Included in BM25 keyword search

Vespa Schema Configuration

```
field metadata_list type array<string> {
    indexing: summary | attribute
    rank: filter
    attribute: fast-search
}

field metadata type string {
    indexing: summary | attribute
}

field metadata_suffix type string {
    indexing: summary | attribute
}
```

Metadata Processing for Vespa

Step 1: Cleaning

- Invalid Unicode characters are removed
- Metadata is sanitized for Vespa compatibility

Step 2: Transformation

- JSON metadata is converted to filterable attributes
- Key-value pairs are joined with INDEX_SEPARATOR ("|")

Step 3: Storage

- All three metadata formats are stored in Vespa document

Search & Retrieval Integration

Vector Search Integration

Semantic Embedding: When creating vector embeddings, content includes semantic metadata:

```
embedding_content = chunk.content + chunk.metadata_suffix_semantic
```

This helps the AI understand context better. For example, knowing a document is about "python programming" helps match it with queries about "Python development".

Keyword Search Integration

BM25 Search: Keyword search includes metadata values:

```
search_content = chunk.content + chunk.metadata_suffix_keyword
```

This allows finding documents by metadata values even if they're not in the main content.

Filtering Capabilities

Metadata-based Filtering: Users can filter results using metadata:

```
# Filter by author
filter_query = 'metadata_list contains "author|John Doe"'

# Filter by multiple tags
filter_query = '(metadata_list contains "tags|python") and (metadata_list
contains "tags|AI")'
```

Performance Optimization

Fast-Search Attributes: Vespa's `fast-search` attribute enables O(1) filtering **Rank Filtering**: Filters are applied before ranking for better performance

Configuration & Scenarios

Configuration Options

Environment Variables:

- `SKIP_METADATA_IN_CHUNK`: Controls metadata inclusion (default: false)
- `BLURB_SIZE`: Size of chunk summaries (default: 128)
- `MINI_CHUNK_SIZE`: Size of mini chunks (default: 150)

Constants:

- `MAX_METADATA_PERCENTAGE`: Maximum metadata size (25% of chunk)
- `CHUNK_MIN_CONTENT`: Minimum content size (256 tokens)

Scenario 1: Normal Operation (Metadata Enabled)

Configuration: `SKIP_METADATA_IN_CHUNK=false` (default)

Behavior:

- Metadata is processed and attached to chunks
- Both semantic and keyword versions are created
- Metadata is counted toward token limits
- All three Vespa fields are populated

Example:

```
# Input document
document.metadata = {"author": "Jane", "tags": ["python"]}

# Chunk gets metadata
chunk.metadata_suffix_semantic = "Metadata:\n\tauthor - Jane\n\ttags -
python"
chunk.metadata_suffix_keyword = "Jane python"

# Vespa storage
vespa_doc.metadata = '{"author": "Jane", "tags": ["python"]}'
vespa_doc.metadata_list = ["author|Jane", "tags|python"]
vespa_doc.metadata_suffix = "Jane python"
```

Scenario 2: Metadata Disabled

Configuration: `SKIP_METADATA_IN_CHUNK=true`

Behavior:

- No metadata processing occurs
- Metadata suffix fields remain empty
- Only JSON metadata is stored in Vespa
- Chunks contain only content and title

Example:

```
# Even with document metadata
document.metadata = {"author": "Jane", "tags": ["python"]}

# Chunk has no metadata suffixes
chunk.metadata_suffix_semantic = ""
chunk.metadata_suffix_keyword = ""

# Only JSON storage in Vespa
vespa_doc.metadata = '{"author": "Jane", "tags": ["python"]}'
vespa_doc.metadata_list = ["author|Jane", "tags|python"]
vespa_doc.metadata_suffix = ""
```

Scenario 3: Oversized Metadata

Trigger: Metadata exceeds 25% of chunk token limit

Behavior:

- Semantic metadata is discarded
- Keyword metadata is preserved
- Warning may be logged
- Content is prioritized

Example:

```
# Large metadata that exceeds limit
large_metadata = {"description": "Very long description..." * 100}

# Result: semantic version discarded
chunk.metadata_suffix_semantic = ""
chunk.metadata_suffix_keyword = "Very long description..."  # Preserved
```

Scenario 4: Insufficient Space

Trigger: After title and metadata, less than 256 tokens remain for content

Behavior:

- System progressively removes elements
- Order: contextual RAG → title/metadata → full chunk
- Ensures minimum content threshold

Example:

```
# With large title and metadata
title_tokens = 200
metadata_tokens = 200
available_tokens = 512 - 200 - 200 = 112  # < 256 minimum

# Result: title and metadata removed
chunk.title_prefix = ""
chunk.metadata_suffix_semantic = ""
chunk.metadata_suffix_keyword = ""
```

Practical Examples

Example 1: Confluence Document Processing

Input Document:

```
confluence_doc = Document(
    title="Python Best Practices",
    metadata={
        "author": "John Smith",
        "space": "Engineering",
        "tags": ["python", "best-practices", "coding-standards"],
        "last_modified": "2024-01-15",
        "page_type": "documentation"
    },
    content="Python is a versatile programming language..."
)
```

Processing Steps:

1. **Metadata Extraction**:

```
# Raw metadata
{"author": "John Smith", "space": "Engineering", "tags": ["python", "best-
practices", "coding-standards"]}
```

2. **Metadata Transformation**:

```
# Semantic version
semantic = """Metadata:
    author - John Smith
    space - Engineering
    tags - python, best-practices, coding-standards
    last_modified - 2024-01-15
    page_type - documentation"""

# Keyword version
keyword = "John Smith Engineering python best-practices coding-standards
2024-01-15 documentation"
```

3. **Chunk Creation**:

```
chunk = DocAwareChunk(
    content="Python is a versatile programming language...",
    metadata_suffix_semantic=semantic,
    metadata_suffix_keyword=keyword,
    # ... other fields
)
```

4. **Vespa Storage**:

```
vespa_doc = {
    "content": "Python is a versatile programming language... " + keyword,
    "metadata": '{"author": "John Smith", "space": "Engineering", "tags":
["python", "best-practices", "coding-standards"], "last_modified": "2024-
01-15", "page_type": "documentation"}',
    "metadata_list": ["author|John Smith", "space|Engineering",
"tags|python", "tags|best-practices", "tags|coding-standards",
"last_modified|2024-01-15", "page_type|documentation"],
    "metadata_suffix": keyword
}
```

Example 2: Slack Message Processing

Input Message:

```
slack_msg = Document(
    title="Discussion about ML models",
    metadata={
        "channel": "#data-science",
        "author": "Alice Johnson",
        "timestamp": "2024-01-15T10:30:00",
        "thread_id": "1234567890",
        "message_type": "thread_reply"
    },
    content="I think we should consider using transformers for this
task..."
)
```

Processing Result:

```
# Metadata transformations
semantic = """Metadata:
    channel - #data-science
    author - Alice Johnson
    timestamp - 2024-01-15T10:30:00
    thread_id - 1234567890
    message_type - thread_reply"""

keyword = "#data-science Alice Johnson 2024-01-15T10:30:00 1234567890
thread_reply"

# Vespa filterable attributes
metadata_list = [
    "channel|#data-science",
    "author|Alice Johnson",
    "timestamp|2024-01-15T10:30:00",
    "thread_id|1234567890",
    "message_type|thread_reply"
]
```

Example 3: Search Query with Metadata Filtering

User Query: "Show me Python tutorials by John Smith"

Vespa Query Construction:

```
# Vector search with metadata context
search_content = query + " author: John Smith, tags: python"

# Metadata filtering
metadata_filter = '(metadata_list contains "author|John Smith") and
(metadata_list contains "tags|python")'

# Combined query
vespa_query = f"""
    select * from chunk where
    {metadata_filter} and
    ({vector_search_clause} or {keyword_search_clause})
"""
```

Search Results Include:

- Documents matching the query content
- Filtered by author "John Smith"
- Filtered by tag "python"
- Ranked using both content and metadata relevance

Example 4: Metadata Size Limiting

Large Metadata Document:

```
large_doc = Document(
    metadata={
        "description": "Very detailed description " * 200,  # Very long
        "tags": ["tag" + str(i) for i in range(100)],      # Many tags
        "author": "John Doe"
    },
    content="Short content"
)
```

Processing with Size Limits:

```
# Token calculation
metadata_tokens = count_tokens(metadata_semantic)  # Let's say 400 tokens
```

```
chunk_limit = 512
max_metadata_tokens = 512 * 0.25 = 128  # 25% limit

# Result: metadata exceeds limit
if metadata_tokens > max_metadata_tokens:
    # Semantic metadata discarded
    chunk.metadata_suffix_semantic = ""

    # Keyword metadata preserved (shorter)
    chunk.metadata_suffix_keyword = "Very detailed description... John 
Doe"
```

Troubleshooting & Best Practices

Common Issues and Solutions

Issue 1: Metadata Not Appearing in Search Results

Symptoms:

- Metadata is present in documents but not helping search
- Filtering by metadata returns no results

Possible Causes:

- `SKIP_METADATA_IN_CHUNK=true` environment variable
- Metadata exceeds size limits
- Invalid characters in metadata values

Solutions:

```
# Check configuration
echo $SKIP_METADATA_IN_CHUNK  # Should be empty or false

# Check for invalid characters
# Metadata values should not contain control characters or invalid Unicode
```

Issue 2: Poor Search Quality

Symptoms:

- Relevant documents not being found
- Search results seem to ignore document context

Possible Causes:

- Metadata is too large, overwhelming content
- Important metadata keys are being ignored

Solutions:

```
# Check metadata size
metadata_size = len(json.dumps(document.metadata))
content_size = len(document.content)

# Metadata should be < 25% of content
if metadata_size > content_size * 0.25:
    # Reduce metadata or increase content
```

Issue 3: Slow Query Performance

Symptoms:

- Search queries taking too long
- Filtering operations are slow

Possible Causes:

- Not using fast-search attributes
- Complex metadata filtering

Solutions:

```
# Use simple metadata filters
# Good: metadata_list contains "author|John Doe"
# Bad: complex regex or substring matching

# Limit number of filter conditions
# Good: 2-3 metadata filters
# Bad: 10+ metadata filters
```

Best Practices

1. Metadata Design

Keep Keys Consistent:

```
# Good: consistent keys across documents
{"author": "John", "department": "Engineering"}
{"author": "Jane", "department": "Sales"}

# Bad: inconsistent keys
{"author": "John", "dept": "Engineering"}
{"writer": "Jane", "department": "Sales"}
```

Use Appropriate Data Types:

```
# Good: strings and lists
{"tags": ["python", "tutorial"], "author": "John"}

# Bad: nested objects (not supported)
{"author": {"name": "John", "email": "john@example.com"}}
```

2. Performance Optimization

Limit Metadata Size:

- Keep metadata under 25% of content size
- Use concise, meaningful values
- Avoid very long descriptions in metadata

Use Efficient Filtering:

```
# Efficient: direct attribute matching
metadata_list contains "author|John Doe"

# Inefficient: complex text matching
metadata contains "John"
```

3. Search Quality

Include Relevant Context:

- Add metadata that helps understand document purpose
- Include categorization information
- Add temporal information when relevant

Avoid Noise:

- Don't include system-generated IDs in searchable metadata
- Filter out technical fields that don't help search
- Use the `ignore_for_qa` key for non-searchable metadata

4. Maintenance

Monitor Metadata Quality:

```
# Check for common issues
def validate_metadata(metadata):
    # Check for empty values
    empty_values = [k for k, v in metadata.items() if not v]

    # Check for very long values
    long_values = [k for k, v in metadata.items() if len(str(v)) > 500]

    # Check for special characters
    special_chars = [k for k in metadata.keys() if not k.isalnum()]

    return {"empty": empty_values, "long": long_values, "special":
special_chars}
```

Regular Cleanup:

- Remove obsolete metadata fields
- Standardize metadata formats across connectors
- Update metadata processing logic as needed

Configuration Examples

Production Configuration:

```
# Enable metadata processing
SKIP_METADATA_IN_CHUNK=false

# Reasonable chunk sizes
BLURB_SIZE=128
MINI_CHUNK_SIZE=150
DOC_EMBEDDING_CONTEXT_SIZE=512

# Enable chunk summaries for better context
USE_CHUNK_SUMMARY=true
USE_DOCUMENT_SUMMARY=true
```

Development/Testing Configuration:

```
# Disable metadata for faster testing
SKIP_METADATA_IN_CHUNK=true

# Smaller chunks for testing
DOC_EMBEDDING_CONTEXT_SIZE=256
```

High-Volume Configuration:

```
# Standard metadata processing
SKIP_METADATA_IN_CHUNK=false

# Larger chunks for better performance
DOC_EMBEDDING_CONTEXT_SIZE=1024
BLURB_SIZE=256
```

This comprehensive guide covers everything you need to know about metadata handling in Onyx. The system is designed to balance search quality with performance, ensuring that metadata enhances rather than hinders the user experience.