Working with Document Metadata and Hubrid Search

I'll add a section that shows how to set custom metadata for documents, and how to use this metadata effectively in hybrid search scenarios with Azure AI Search.

Working with Document Metadata in Azure AI Search for Hybrid Queries

1. Designing an Index with Rich Metadata

First, let's create an enhanced index schema that includes various metadata fields for hybrid search:

```
from azure.search.documents.indexes.models import (
    SearchIndex,
    SearchField,
    SearchFieldDataType,
    SimpleField,
    SearchableField,
    ComplexField,
    VectorSearch,
    VectorSearchProfile,
    VectorSearchAlgorithmConfiguration,
    HnswAlgorithmConfiguration
)

def create_advanced_index(index_client, index_name):
    """Create an index with rich metadata support and hybrid search
capabilities"""

    # Define the fields, including various metadata fields
    fields = [
        # Key field (required)
        SimpleField(name="id", type=SearchFieldDataType.String, key=True),

        # Core searchable content
        SearchableField(name="title", type=SearchFieldDataType.String,
                        analyzer_name="en.microsoft"),
        SearchableField(name="content", type=SearchFieldDataType.String,
                        analyzer_name="en.microsoft"),

        # Metadata fields - all filterable, facetable, and sortable
        SimpleField(name="created_date",
type=SearchFieldDataType.DateTimeOffset,
                    filterable=True, sortable=True),
        SimpleField(name="last_updated",
type=SearchFieldDataType.DateTimeOffset,
                    filterable=True, sortable=True),
        SimpleField(name="author", type=SearchFieldDataType.String,
                    filterable=True, facetable=True),
        SimpleField(name="document_type", type=SearchFieldDataType.String,
                    filterable=True, facetable=True),
```

```
        SimpleField(name="department", type=SearchFieldDataType.String,
                    filterable=True, facetable=True),
        SimpleField(name="priority", type=SearchFieldDataType.Int32,
                    filterable=True, sortable=True),
        SimpleField(name="status", type=SearchFieldDataType.String,
                    filterable=True, facetable=True),

        # Collection/array fields
        SimpleField(name="tags",
type=SearchFieldDataType.Collection(SearchFieldDataType.String),
                    filterable=True, facetable=True),

        # Geographic location data
        SimpleField(name="geo_location",
type=SearchFieldDataType.GeographyPoint,
                    filterable=True),

        # Numerical data for filtering and boosting
        SimpleField(name="view_count", type=SearchFieldDataType.Int32,
                    filterable=True, sortable=True),
        SimpleField(name="relevance_score",
type=SearchFieldDataType.Double,
                    filterable=True, sortable=True)
    ]

    # Create the index
    index = SearchIndex(name=index_name, fields=fields)

    try:
        result = index_client.create_or_update_index(index)
        print(f"Advanced index '{index_name}' created successfully with
metadata fields.")
        return result
    except Exception as e:
        print(f"Error creating index: {str(e)}")
        raise
```

2. Uploading Documents with Rich Metadata

Now, let's upload documents with detailed metadata:

```
def upload_documents_with_metadata(search_client):
    """Upload sample documents with rich metadata"""

    # Current time for timestamps
    from datetime import datetime, timedelta
    current_time = datetime.utcnow().isoformat()
    yesterday = (datetime.utcnow() - timedelta(days=1)).isoformat()

    # Sample documents with metadata
    documents = [
```

```
        {
            "id": "doc-001",
            "title": "Azure AI Search Implementation Guide",
            "content": "This comprehensive guide covers advanced
implementation techniques for Azure AI Search, including hybrid search
models, vector search, and semantic ranking.",
            "created_date": yesterday,
            "last_updated": current_time,
            "author": "Alice Johnson",
            "document_type": "technical_guide",
            "department": "Cloud Services",
            "priority": 1,
            "status": "published",
            "tags": ["azure", "ai", "search", "hybrid", "implementation"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-122.12, 47.67]  # Seattle coordinates
            },
            "view_count": 1200,
            "relevance_score": 0.95
        },
        {
            "id": "doc-002",
            "title": "Python Tools for Azure Integration",
            "content": "Learn how to use Python SDKs and libraries to
integrate with Azure services, including Azure AI Search, Cognitive
Services, and Azure Machine Learning.",
            "created_date": yesterday,
            "last_updated": current_time,
            "author": "Bob Smith",
            "document_type": "tutorial",
            "department": "Development",
            "priority": 2,
            "status": "published",
            "tags": ["python", "azure", "integration", "sdk"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-74.01, 40.71]  # New York coordinates
            },
            "view_count": 850,
            "relevance_score": 0.88
        },
        {
            "id": "doc-003",
            "title": "Natural Language Processing with Azure",
            "content": "This document explores how to implement NLP
solutions using Azure's AI services, focusing on text analysis, sentiment
detection, and knowledge mining capabilities.",
            "created_date": current_time,
            "last_updated": current_time,
            "author": "Carol Davis",
            "document_type": "whitepaper",
            "department": "AI Research",
            "priority": 1,
```

```
            "status": "draft",
            "tags": ["nlp", "azure", "ai", "text-analysis", "sentiment"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-118.24, 34.05]  # Los Angeles coordinates
            },
            "view_count": 320,
            "relevance_score": 0.92
        }
    ]

    try:
        result = search_client.upload_documents(documents=documents)
        print(f"Uploaded {len(result)} documents with rich metadata")
        return True
    except Exception as e:
        print(f"Error uploading documents: {str(e)}")
        return False
```

3. Performing Hybrid Searches Using Metadata

Now, let's implement some advanced hybrid search scenarios that leverage the metadata:

```
def perform_hybrid_search(search_client, query_text, options=None):
    """
    Perform a hybrid search that combines full-text search with metadata
filtering
    and boosting for optimal results.
    """
    if options is None:
        options = {}

    # Default search options for hybrid search
    search_options = {
        "include_total_count": True,
        "highlight_fields": "content",
        "highlight_pre_tag": "<b>",
        "highlight_post_tag": "</b>",
        # Return all fields
        "select": "*"
    }

    # Update with any user-provided options
    search_options.update(options)

    try:
        # Execute the search
        results = search_client.search(query_text, **search_options)

        # Process the results
        print(f"\nHybrid search results for: '{query_text}'")
```

```
        print(f"Using options: {search_options}")

        count = 0
        for result in results:
            count += 1
            print(f"\nDocument ID: {result['id']}")
            print(f"Title: {result['title']}")

            # Display metadata
            print(f"Author: {result['author']}")
            print(f"Document Type: {result['document_type']}")
            print(f"Department: {result['department']}")
            print(f"Status: {result['status']}")
            print(f"Tags: {', '.join(result['tags'])}")

            # Display dates in a readable format
            from datetime import datetime
            created =
datetime.fromisoformat(result['created_date'].replace('Z', '+00:00'))
            updated =
datetime.fromisoformat(result['last_updated'].replace('Z', '+00:00'))
            print(f"Created: {created.strftime('%Y-%m-%d %H:%M:%S')}")
            print(f"Last Updated: {updated.strftime('%Y-%m-%d
%H:%M:%S')}")

            # Display relevance metrics
            print(f"View Count: {result['view_count']}")
            print(f"Relevance Score: {result['relevance_score']}")
            print(f"Search Score: {result['@search.score']}")

            # Display highlighted content if available
            if 'content' in result.get('@search.highlights', {}):
                print("Content highlights:")
                for highlight in result['@search.highlights']['content']:
                    print(f"  ...{highlight}...")

        print(f"\nFound {count} documents")

    except Exception as e:
        print(f"Search error: {str(e)}")
```

4. Example Hybrid Search Scenarios

Let's demonstrate various hybrid search scenarios that leverage metadata:

```
def demonstrate_hybrid_search_scenarios(search_client):
    """Show various hybrid search scenarios using metadata"""

    print("\n=== HYBRID SEARCH SCENARIOS ===\n")

    # Scenario 1: Basic hybrid search with filtering
```

```
    print("\n--- SCENARIO 1: Content search with department filter ---")
    perform_hybrid_search(
        search_client,
        "azure implementation",
        {
            "filter": "department eq 'Cloud Services'",
            "orderby": "relevance_score desc"
        }
    )

    # Scenario 2: Tag-based filtering with boost on priority
    print("\n--- SCENARIO 2: Tag-based search with priority boost ---")
    perform_hybrid_search(
        search_client,
        "azure",
        {
            "filter": "tags/any(t: t eq 'ai')",
            "search_fields": "title,content,tags",
            "scoring_profile": "priorityBooster"  # Assumes this scoring
profile exists
        }
    )

    # Scenario 3: Filtering by document type and status
    print("\n--- SCENARIO 3: Filtering by document metadata ---")
    perform_hybrid_search(
        search_client,
        "python",
        {
            "filter": "document_type eq 'tutorial' and status eq
'published'",
            "orderby": "view_count desc"
        }
    )

    # Scenario 4: Geo-filtered search within 100km of San Francisco
    print("\n--- SCENARIO 4: Geo-filtered search ---")
    perform_hybrid_search(
        search_client,
        "azure",
        {
            "filter": "geo.distance(geo_location,
geography'POINT(-122.4194 37.7749)') lt 1000",
            "orderby": "geo.distance(geo_location,
geography'POINT(-122.4194 37.7749)') asc"
        }
    )

    # Scenario 5: Time-based filtering for recent documents
    print("\n--- SCENARIO 5: Recent documents only ---")
    from datetime import datetime, timedelta
    two_days_ago = (datetime.utcnow() - timedelta(days=2)).isoformat()
    perform_hybrid_search(
        search_client,
```

```
        "azure",
        {
            "filter": f"created_date gt {two_days_ago}"
        }
    )
```

5. Creating Scoring Profiles for Hybrid Search

To fully leverage hybrid search, we should create scoring profiles that boost relevance based on metadata:

```
from azure.search.documents.indexes.models import (
    ScoringProfile,
    TextWeights,
    FreshnessScoringFunction,
    MagnitudeScoringFunction,
    DistanceScoringFunction,
    TagScoringFunction,
    ScoringFunctionAggregation,
    FreshnessScoringParameters,
    MagnitudeScoringParameters,
    DistanceScoringParameters,
    TagScoringParameters
)

def create_scoring_profiles(index_client, index_name):
    """Create scoring profiles for hybrid search scenarios"""

    try:
        # Get the existing index
        index = index_client.get_index(index_name)

        # Define scoring profiles
        scoring_profiles = [
            # Profile 1: Boost by priority and freshness
            ScoringProfile(
                name="priorityAndFreshness",
                text_weights=TextWeights(
                    weights={"title": 5, "content": 3, "tags": 2}
                ),
                functions=[
                    # Boost newer documents
                    FreshnessScoringFunction(
                        field_name="last_updated",
                        boost=2,
                        parameters=FreshnessScoringParameters(
                            boosting_duration_in_days=30
                        ),
                        interpolation="logarithmic"
                    ),
                    # Boost based on priority field (higher priority =
higher boost)
```

```
                MagnitudeScoringFunction(
                    field_name="priority",
                    boost=1.5,
                    parameters=MagnitudeScoringParameters(
                        boosting_range_start=3,
                        boosting_range_end=1
                    ),
                    interpolation="linear"
                ),
                # Boost based on view_count
                MagnitudeScoringFunction(
                    field_name="view_count",
                    boost=1.2,
                    parameters=MagnitudeScoringParameters(
                        boosting_range_start=0,
                        boosting_range_end=1000
                    ),
                    interpolation="logarithmic"
                )
            ],
            function_aggregation=ScoringFunctionAggregation.SUM
        ),

        # Profile 2: Boost by relevance_score and tag matching
        ScoringProfile(
            name="relevanceBooster",
            text_weights=TextWeights(
                weights={"title": 4, "content": 2}
            ),
            functions=[
                # Use the pre-computed relevance score
                MagnitudeScoringFunction(
                    field_name="relevance_score",
                    boost=3,
                    parameters=MagnitudeScoringParameters(
                        boosting_range_start=0,
                        boosting_range_end=1
                    ),
                    interpolation="linear"
                ),
                # Boost documents with specific tags
                TagScoringFunction(
                    field_name="tags",
                    boost=2,
                    parameters=TagScoringParameters(
                        tags_parameter="preferredTags"
                    ),
                    interpolation="linear"
                )
            ],
            function_aggregation=ScoringFunctionAggregation.SUM
        ),

        # Profile 3: Geographical proximity boosting
```

```
            ScoringProfile(
                name="geoProximity",
                functions=[
                    DistanceScoringFunction(
                        field_name="geo_location",
                        boost=2,
                        parameters=DistanceScoringParameters(
                            reference_point_parameter="currentLocation",
                            boosting_distance=100
                        ),
                        interpolation="linear"
                    )
                ],
                function_aggregation=ScoringFunctionAggregation.SUM
            )
        ]

        # Add scoring profiles to the index
        index.scoring_profiles = scoring_profiles

        # Update the index
        index_client.create_or_update_index(index)
        print(f"Added scoring profiles to index '{index_name}'")

    except Exception as e:
        print(f"Error creating scoring profiles: {str(e)}")
```

6. Using Scoring Profiles and Parameters in Searches

Now, let's use these scoring profiles in our hybrid searches:

```
def perform_advanced_hybrid_searches(search_client):
    """Perform searches using scoring profiles and parameters"""

    print("\n=== ADVANCED HYBRID SEARCH WITH SCORING PROFILES ===\n")

    # Scenario 1: Using priorityAndFreshness scoring profile
    print("\n--- SCENARIO 1: Priority and freshness boosting ---")
    perform_hybrid_search(
        search_client,
        "azure",
        {
            "scoring_profile": "priorityAndFreshness"
        }
    )

    # Scenario 2: Using relevanceBooster with tag parameters
    print("\n--- SCENARIO 2: Relevance and tag boosting ---")
    perform_hybrid_search(
        search_client,
        "azure services",
```

```
        {
            "scoring_profile": "relevanceBooster",
            "scoring_parameters": ["preferredTags=ai,search,hybrid"]
        }
    )

    # Scenario 3: Using geographical proximity boosting
    print("\n--- SCENARIO 3: Geo-proximity boosting (from San Francisco) -
--")
    perform_hybrid_search(
        search_client,
        "azure",
        {
            "scoring_profile": "geoProximity",
            "scoring_parameters": ["currentLocation=-122.4194,37.7749"]  #
San Francisco coordinates
        }
    )

    # Scenario 4: Complex filtering with metadata
    print("\n--- SCENARIO 4: Complex metadata filtering ---")
    perform_hybrid_search(
        search_client,
        "azure",
        {
            "filter": "document_type eq 'technical_guide' or (tags/any(t:
t eq 'implementation') and priority lt 3)",
            "scoring_profile": "priorityAndFreshness"
        }
    )
```

7. Complete Implementation with Hybrid and Metadata-Enhanced Search

Here's a complete example that ties everything together:

```
def main():
    """Main function to demonstrate Azure AI Search with metadata for
hybrid search"""
    try:
        # Authenticate
        credential = AzureCliCredential()

        # Get subscription ID
        from azure.mgmt.resource import SubscriptionClient
        subscription_client = SubscriptionClient(credential)
        subscription = next(subscription_client.subscriptions.list())
        subscription_id = subscription.subscription_id

        print(f"Using subscription: {subscription.display_name}
({subscription_id})")
```

```
        # Get search service details
        resource_group_name = input("Enter your resource group name: ")
        service_name = input("Enter your search service name: ")

        # Create Search Management client
        search_mgmt_client = SearchManagementClient(credential,
subscription_id)

        # Get the admin key
        try:
            admin_key = search_mgmt_client.admin_keys.get(
                resource_group_name=resource_group_name,
                search_service_name=service_name
            ).primary_key

            print(f"Successfully connected to search service:
{service_name}")

            # Set up search clients
            service_endpoint =
f"https://{service_name}.search.windows.net"
            credential = AzureKeyCredential(admin_key)

            # Create index client
            index_client = SearchIndexClient(endpoint=service_endpoint,
credential=credential)

            # Create a new advanced index
            index_name = "advanced-metadata-index"
            create_advanced_index(index_client, index_name)

            # Create a search client
            search_client = SearchClient(endpoint=service_endpoint,
index_name=index_name, credential=credential)

            # Upload documents with rich metadata
            upload_documents_with_metadata(search_client)

            # Create scoring profiles
            create_scoring_profiles(index_client, index_name)

            # Demonstrate hybrid search scenarios
            demonstrate_hybrid_search_scenarios(search_client)

            # Demonstrate advanced hybrid searches with scoring profiles
            perform_advanced_hybrid_searches(search_client)

            # Interactive hybrid search
            interactive_hybrid_search(search_client)

        except ResourceNotFoundError:
            print(f"Error: Search service '{service_name}' not found in
resource group '{resource_group_name}'")
```

```
        except Exception as e:
            print(f"An error occurred: {str(e)}")

    def interactive_hybrid_search(search_client):
        """Interactive hybrid search with customizable options"""

        print("\n=== INTERACTIVE HYBRID SEARCH ===")
        print("You can search with various options. Enter 'exit' to quit.")

        while True:
            query = input("\nEnter search query: ")
            if query.lower() == 'exit':
                break

            # Ask for filter options
            print("\nFilter options (leave blank for no filter):")
            print("1. Department filter")
            print("2. Document type filter")
            print("3. Tag filter")
            print("4. Custom filter")
            filter_choice = input("Choose filter option (1-4) or press Enter
    for none: ")

            filter_expr = None
            if filter_choice == '1':
                department = input("Enter department name: ")
                filter_expr = f"department eq '{department}'"
            elif filter_choice == '2':
                doc_type = input("Enter document type: ")
                filter_expr = f"document_type eq '{doc_type}'"
            elif filter_choice == '3':
                tag = input("Enter tag to filter by: ")
                filter_expr = f"tags/any(t: t eq '{tag}')"
            elif filter_choice == '4':
                filter_expr = input("Enter custom OData filter expression: ")

            # Ask for scoring profile
            print("\nScoring profile options:")
            print("1. priorityAndFreshness - Boost by priority and recency")
            print("2. relevanceBooster - Boost by relevance score and tags")
            print("3. geoProximity - Boost by geographical proximity")
            print("4. None (default scoring)")

            scoring_choice = input("Choose scoring profile (1-4) or press
    Enter for default: ")

            scoring_profile = None
            scoring_parameters = None

            if scoring_choice == '1':
                scoring_profile = "priorityAndFreshness"
            elif scoring_choice == '2':
                scoring_profile = "relevanceBooster"
                tags = input("Enter preferred tags (comma-separated): ")
```

```
            if tags:
                scoring_parameters = [f"preferredTags={tags}"]
        elif scoring_choice == '3':
            scoring_profile = "geoProximity"
            location = input("Enter location (longitude,latitude) or press
Enter for San Francisco: ")
            if not location:
                location = "-122.4194,37.7749"  # San Francisco
            scoring_parameters = [f"currentLocation={location}"]

        # Build search options
        search_options = {
            "include_total_count": True,
            "highlight_fields": "content",
            "highlight_pre_tag": "<b>",
            "highlight_post_tag": "</b>",
            "select": "*"
        }

        if filter_expr:
            search_options["filter"] = filter_expr

        if scoring_profile:
            search_options["scoring_profile"] = scoring_profile

        if scoring_parameters:
            search_options["scoring_parameters"] = scoring_parameters

        # Perform search
        perform_hybrid_search(search_client, query, search_options)

if __name__ == "__main__":
    main()
```

8. Best Practices for Metadata in Hybrid Search

Here are some best practices for using metadata effectively in hybrid search scenarios:

```
def print_metadata_best_practices():
    """Print best practices for using metadata in hybrid search"""

    print("\n=== BEST PRACTICES FOR METADATA IN HYBRID SEARCH ===\n")

    best_practices = [
        {
            "area": "Index Design",
            "practices": [
                "Define clear metadata fields that align with common
filtering and sorting needs",
                "Make fields filterable, facetable, and sortable only when
needed (impacts index size)",
```

```
                "Use appropriate data types (e.g., DateTimeOffset for
dates, not strings)",
                "Create collection fields (arrays) for multi-value
properties like tags"
            ]
        },
        {
            "area": "Document Processing",
            "practices": [
                "Normalize metadata values for consistency (e.g.,
lowercase tags, standard date formats)",
                "Add computed/derived metadata fields that aid searching
(e.g., relevance_score)",
                "Include both specific and general metadata to support
different query granularities",
                "Consider adding geo-coordinates where relevant for
location-based searching"
            ]
        },
        {
            "area": "Scoring Profiles",
            "practices": [
                "Create multiple scoring profiles for different search
scenarios",
                "Use freshness functions for time-sensitive content",
                "Use magnitude functions for numeric boosting (views,
priority, etc.)",
                "Use tag functions to boost matches on important
keywords",
                "Use distance functions for geographical relevance"
            ]
        },
        {
            "area": "Hybrid Search Implementation",
            "practices": [
                "Combine full-text search with metadata filtering",
                "Use relevance tuning through scoring profiles rather than
just filtering",
                "Pass user context as scoring parameters (location,
preferences, etc.)",
                "Consider user behavior data as metadata for personalized
results",
                "Use facets to help users navigate through metadata-based
categories"
            ]
        }
    ]

    for area in best_practices:
        print(f"\n{area['area']}:")
        for i, practice in enumerate(area['practices'], 1):
            print(f"  {i}. {practice}")
```

This comprehensive guide should give you a solid foundation for working with document metadata in Azure AI Search, especially for hybrid search scenarios. By properly designing your index schema with rich metadata fields and creating appropriate scoring profiles, you can create sophisticated search experiences that combine the best aspects of keyword search and filtering with intelligent relevance boosting.