Source: admin.md

Administration

DremioFrame provides tools for managing your Dremio environment, including sources, folders, users, grants, and security policies.

Catalog Management

Access via `client.catalog`.

Sources

```
# Create a new S3 source
config = {
    "accessKey": "...",
    "accessSecret": "...",
    "bucketName": "my-bucket"
}
client.catalog.create_source("my_s3", "S3", config)

# Delete a source
source_id = client.catalog.get_entity_by_path("my_s3")['id']
client.catalog.delete_catalog_item(source_id)
```

Folders

```
# Create a folder
client.catalog.create_folder(["space_name", "folder_name"])
```

User & Security Management

Access via `client.admin`.

Users

```
# Create a user
client.admin.create_user("new_user", "password123")

# Change password
client.admin.alter_user_password("new_user", "new_password456")

# Delete user
client.admin.drop_user("new_user")
```

Grants (RBAC)

Manage permissions for users and roles.

```
# Grant SELECT on a table to a role
client.admin.grant("SELECT", "TABLE marketing.sales", to_role="DATA_ANALYST")

# Grant all privileges on a space to a user
client.admin.grant("ALL PRIVILEGES", "SPACE marketing", to_user="john.doe")

# Revoke privileges
client.admin.revoke("SELECT", "TABLE marketing.sales", from_role="DATA_ANALYST")
```

Masking Policies

Column masking allows you to obscure sensitive data based on user roles.

1. **Create a Policy Function** (UDF)

```
# Create a function that masks SSN for non-admins
client.admin.create_policy_function(
    name="mask_ssn",
    args="ssn VARCHAR",
    return_type="VARCHAR",
    body="CASE WHEN is_member('admin') THEN ssn ELSE '***-**-****' END"
)
```

2. **Apply Policy**

```
# Apply to the 'ssn' column of 'employees' table
client.admin.apply_masking_policy("employees", "ssn", "mask_ssn(ssn)")
```

3. **Drop Policy**

```
client.admin.drop_masking_policy("employees", "ssn")
```

Source Management

Manage data sources (S3, Nessie, Postgres, etc.).

List Sources

```
sources = client.admin.list_sources()
```

Get Source

```
source = client.admin.get_source("my_source")
```

Create Source

```
# Generic
client.admin.create_source(
    name="my_postgres",
    type="POSTGRES",
    config={"hostname": "...", "username": "..."}
)

# S3 Helper
client.admin.create_source_s3(
    name="my_datalake",
    bucket_name="my-bucket",
    access_key="...",
    secret_key="..."
)
```

Delete Source

```
client.admin.delete_source("source_id")
```

Reflection Management

You can manage Dremio reflections (Raw and Aggregation) using the `admin` interface.

```
# List all reflections
reflections = client.admin.list_reflections()

# Create a Raw Reflection
client.admin.create_reflection(
    dataset_id="dataset-uuid",
    name="my_raw_reflection",
    type="RAW",
    display_fields=["col1", "col2"],
    distribution_fields=["col1"],
    partition_fields=["col2"],
    sort_fields=["col1"]
)

# Create an Aggregation Reflection
client.admin.create_reflection(
    dataset_id="dataset-uuid",
    name="my_agg_reflection",
    type="AGGREGATION",
    dimension_fields=["dim1", "dim2"],
    measure_fields=["measure1"],
```

```
    distribution_fields=["dim1"]
)

# Enable/Disable Reflection
client.admin.enable_reflection("reflection-id")
client.admin.disable_reflection("reflection-id")

# Delete Reflection
client.admin.delete_reflection("reflection-id")
```

Row Access Policies

Row access policies filter rows based on user roles.

Create a Policy Function

```
# Create a function that returns TRUE if user can see the row
client.admin.create_policy_function(
    name="region_filter",
    args="region VARCHAR",
    return_type="BOOLEAN",
    body="is_member('admin') OR is_member(region)"
)
```

Apply Policy

```
# Apply to 'sales' table
client.admin.apply_row_access_policy("sales", "region_filter(region)")
```

Drop Policy

```
client.admin.drop_row_access_policy("sales")
```

Source: batch_operations.md

Batch Operations

DremioFrame provides a `BatchManager` to perform bulk operations on the Dremio catalog efficiently using parallel API requests.

BatchManager

The `BatchManager` utilizes a thread pool to execute multiple API calls concurrently, significantly speeding up operations like creating many folders or deleting multiple datasets.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.batch import BatchManager

client = DremioClient()
# Initialize with 10 concurrent workers
manager = BatchManager(client, max_workers=10)
```

Creating Folders

Create multiple folders at once.

```
paths = [
    "space.folder1",
    "space.folder2",
    "space.folder3"
]

results = manager.create_folders(paths)

for path, result in results.items():
    if "error" in result:
        print(f"Failed to create {path}: {result['error']}")
    else:
        print(f"Created {path}")
```

Deleting Items

Delete multiple items (datasets, folders, spaces) by their ID.

```
ids_to_delete = ["id1", "id2", "id3"]

results = manager.delete_items(ids_to_delete)

for id, success in results.items():
    if success is True:
        print(f"Deleted {id}")
    else:
        print(f"Failed to delete {id}: {success['error']}")
```

Performance Considerations

Rate Limits: Be mindful of Dremio's API rate limits when using a high number of workers.

Error Handling: Batch operations return a dictionary of results where individual failures are captured without stopping the entire batch. Always check the results for

errors.

Source: catalog.md

Catalog & Admin

The `Catalog` class provides access to Dremio's catalog and administrative functions via the REST API.

Accessing the Catalog

You can access the catalog through the `DremioClient` instance:

```
from dremioframe.client import DremioClient

client = DremioClient()
catalog = client.catalog
```

Listing Items

To list the contents of the root catalog or a specific path:

```
# List root catalog
items = catalog.list_catalog()
for item in items:
    print(item['path'], item['type'])

# List contents of a source or folder
items = catalog.list_catalog("Samples")
```

Managing Sources

You can create and delete sources:

```
# Create a source (example for S3)
config = {
    "bucketName": "my-bucket",
    "authenticationType": "ACCESS_KEY",
    "accessKey": "...",
    "accessSecret": "..."
}
catalog.create_source("MyS3Source", "S3", config)

# Delete a source
catalog.delete_catalog_item("source-id-uuid")
```

Managing Views

You can create and update virtual datasets (views). The `sql` argument accepts either a raw SQL string or a `DremioBuilder` object (DataFrame).

```
# Create a view using SQL string
catalog.create_view(
    path=["Space", "MyView"],
    sql="SELECT * FROM source.table"
)

# Create a view using a DataFrame (Builder)
df = client.table("source.table").filter("id > 100")
catalog.create_view(
    path=["Space", "FilteredView"],
    sql=df
)
```

Update a view (fetches latest version tag automatically)

```
catalog.update_view(
    id="view-id-uuid",
    path=["Space", "MyView"],
    sql="SELECT * FROM source.table WHERE id > 100"
)
```

Collaboration (Wikis & Tags)

Manage documentation and tags for any catalog entity (dataset, source, space, folder).

Wikis

```
# Get Wiki
wiki = catalog.get_wiki("entity-id")
print(wiki.get("text"))

# Update Wiki (fetch version first to avoid conflict)
try:
    current_wiki = catalog.get_wiki("entity-id")
    version = current_wiki.get("version")
except:
    version = None

catalog.update_wiki(
    id="entity-id",
    content="# My Dataset\n\nThis is a documented dataset.",
    version=version
)
```

Tags

```
# Get Tags
tags = catalog.get_tags("entity-id")
print(tags)

# Set Tags (Overwrites existing tags)
catalog.set_tags("entity-id", ["production", "marketing"])
```

Source: lineage.md

Data Lineage

DremioFrame provides access to the data lineage graph for datasets, allowing you to understand dependencies and data flow.

Usage

Access lineage via `client.catalog.get_lineage(id)`.

```
from dremioframe.client import DremioClient

client = DremioClient()
dataset_id = "..."

# Get lineage graph
lineage = client.catalog.get_lineage(dataset_id)

# Inspect parents (upstream dependencies)
parents = lineage.get("parents", [])
for parent in parents:
    print(f"Parent: {parent['path']} ({parent['id']})")

# Inspect children (downstream dependencies)
children = lineage.get("children", [])
for child in children:
    print(f"Child: {child['path']} ({child['id']})")
```

Source: lineage_tracking.md

Data Lineage Tracking

DremioFrame provides a `LineageTracker` to track, visualize, and export data lineage for governance, impact analysis, and documentation.

LineageTracker

The `LineageTracker` maintains a graph of data transformations showing how data flows through your pipelines.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.lineage import LineageTracker

client = DremioClient()
tracker = LineageTracker(client)
```

Tracking Transformations

Record data transformations as they occur:

```
# Track an INSERT operation
tracker.track_transformation(
    source="raw.events",
    target="staging.events",
    operation="insert",
    metadata={"rows": 10000, "timestamp": "2024-01-01"}
)

# Track a SELECT transformation
tracker.track_transformation(
    source="staging.events",
    target="analytics.daily_summary",
    operation="aggregate",
    metadata={"group_by": "date"}
)

# Track a JOIN
tracker.track_transformation(
    source="staging.events",
    target="analytics.enriched_events",
    operation="join",
    metadata={"join_table": "dim.customers"}
)
```

Querying Lineage

Get upstream or downstream dependencies:

```
# Get all upstream sources for a table
lineage = tracker.get_lineage_graph(
    table="analytics.daily_summary",
    direction="upstream",
    max_depth=5
)
```

```
print(f"Upstream dependencies: {list(lineage.nodes.keys())}")
# Output: ['staging.events', 'raw.events']

# Get all downstream consumers
lineage = tracker.get_lineage_graph(
    table="raw.events",
    direction="downstream"
)

print(f"Downstream consumers: {list(lineage.nodes.keys())}")
# Output: ['staging.events', 'analytics.daily_summary', 'analytics.enriched_events']
```

Visualizing Lineage

Create interactive HTML visualizations:

```
# Visualize the entire lineage graph
html = tracker.visualize(format='html', output_file='lineage.html')

# Visualize lineage for a specific table
lineage = tracker.get_lineage_graph("analytics.daily_summary", direction="both")
tracker.visualize(lineage, format='html', output_file='daily_summary_lineage.html')
```

The HTML visualization uses vis.js to create an interactive, draggable graph.

Static Visualizations

Create static images using Graphviz (requires optional dependency):

```
# Requires: pip install dremioframe[lineage]

# Generate PNG
tracker.visualize(format='png', output_file='lineage.png')

# Generate SVG
tracker.visualize(format='svg', output_file='lineage.svg')
```

Exporting Lineage

Export lineage data for integration with external tools:

```
# Export to JSON
json_data = tracker.export_lineage(format='json', output_file='lineage.json')

# Export to DataHub format
datahub_data = tracker.export_lineage(format='datahub')

# Export to Amundsen format
amundsen_data = tracker.export_lineage(format='amundsen')
```

Use Cases

1. Impact Analysis

Understand what will be affected by schema changes:

```
# Before modifying raw.events, check downstream impact
lineage = tracker.get_lineage_graph("raw.events", direction="downstream")

print(f"Tables affected by changes to raw.events:")
for node_id in lineage.nodes:
    print(f"  - {node_id}")
```

2. Data Governance

Document data flows for compliance:

```
# Track all transformations in your pipeline
def run_etl_pipeline():
    # Extract
    tracker.track_transformation("source.db.customers", "raw.customers", "extract")

    # Transform
    tracker.track_transformation("raw.customers", "staging.customers", "clean")
              tracker.track_transformation("staging.customers",   "analytics.customers",
"aggregate")

    # Export lineage for audit
    tracker.export_lineage(format='json', output_file='audit/lineage_2024_01.json')
```

3. Pipeline Documentation

Auto-generate pipeline documentation:

```
# Create visual documentation
for table in ["analytics.sales", "analytics.customers", "analytics.products"]:
    lineage = tracker.get_lineage_graph(table, direction="both")
    tracker.visualize(lineage, format='html',
                      output_file=f'docs/{table.replace(".", "_")}_lineage.html')
```

4. Root Cause Analysis

Trace data quality issues to their source:

```
# If analytics.daily_summary has bad data, trace it back
lineage = tracker.get_lineage_graph("analytics.daily_summary", direction="upstream")

print("Data sources to investigate:")
```

```
for node_id in lineage.nodes:
    node = lineage.nodes[node_id]
    if node.type == 'table':
        print(f"  - {node_id}")
```

LineageGraph API

The `LineageGraph` object provides programmatic access to lineage data:

```
lineage = tracker.get_lineage_graph("my.table")

# Access nodes
for node in lineage.nodes.values():
    print(f"{node.name} ({node.type})")

# Access edges
for edge in lineage.edges:
    print(f"{edge.source_id} --[{edge.operation}]--> {edge.target_id}")

# Convert to NetworkX for advanced analysis
nx_graph = lineage.to_networkx()  # Requires networkx

# Export to dict
data = lineage.to_dict()
```

Integration with External Tools

DataHub

```
# Export lineage for DataHub ingestion
datahub_json = tracker.export_lineage(format='datahub')

# Use DataHub's REST API or CLI to ingest
# datahub ingest -c lineage_config.yml
```

Amundsen

```
# Export for Amundsen
amundsen_json = tracker.export_lineage(format='amundsen')

# Load into Amundsen's metadata service
```

Best Practices

Track at Key Points: Record transformations at major pipeline stages (extract,

transform, load)

Include Metadata: Add context like row counts, timestamps, and transformation logic

Regular Exports: Periodically export lineage for backup and audit

Visualize Often: Use visualizations to communicate with stakeholders

Combine with DQ: Link lineage with data quality checks for comprehensive governance

Limitations

Manual Tracking: Transformations must be explicitly tracked (not auto-detected)

In-Memory: Lineage graph is stored in memory (export for persistence)

No Version History: Current implementation doesn't track lineage changes over time

Source: masking_and_row_access.md

Row Access and Column Masking

`dremioframe` provides methods to manage Dremio's Row Access and Column Masking policies, allowing you to secure your data dynamically.

User Defined Functions (UDFs)

Policies rely on User Defined Functions (UDFs) to define the logic for access control and masking.

Creating a UDF

Use `admin.create_udf` (which delegates to `client.udf.create`) to create a UDF.

```
from dremioframe.client import DremioClient

client = DremioClient(pat="...", project_id="...")

# Create a masking UDF
client.admin.create_udf(
    name="target.protect_ssn",
    args="ssn VARCHAR",
    return_type="VARCHAR",
    body="CASE WHEN is_member('hr') THEN ssn ELSE '***-**-****' END",
    replace=True
)

# Create a row access UDF
client.admin.create_udf(
    name="target.region_filter",
    args="region VARCHAR",
```

```
    return_type="BOOLEAN",
    body="is_member('sales') OR region = 'public'",
    replace=True
)
```

Dropping a UDF

```
client.admin.drop_udf("target.protect_ssn", if_exists=True)
```

Column Masking Policies

Column masking policies dynamically mask data in a column based on a UDF.

Applying a Masking Policy

```
# Apply the 'protect_ssn' UDF to the 'ssn' column of 'employees' table
client.admin.apply_masking_policy(
    table="target.employees",
    column="ssn",
    policy="target.protect_ssn(ssn)"
)
```

Removing a Masking Policy

```
# Unset the masking policy
client.admin.drop_masking_policy(
    table="target.employees",
    column="ssn",
    policy="target.protect_ssn" # Optional, but good practice to specify
)
```

Row Access Policies

Row access policies filter rows based on a UDF.

Applying a Row Access Policy

```
# Apply the 'region_filter' UDF to the 'employees' table
client.admin.apply_row_access_policy(
    table="target.employees",
    policy="target.region_filter(region)"
)
```

Removing a Row Access Policy

```
# Drop the row access policy
client.admin.drop_row_access_policy(
    table="target.employees",
    policy="target.region_filter(region)"
)
```

Source: privileges.md

Grants and Privileges

DremioFrame allows you to manage access control lists (grants) for catalog entities (datasets, folders, sources).

Usage

Access grants via `client.catalog`.

Get Grants

Retrieve the current grants for an entity.

```
from dremioframe.client import DremioClient

client = DremioClient()
entity_id = "..."

grants_info = client.catalog.get_grants(entity_id)
print(grants_info)
# Output example:
# {
#   "grants": [
#     {"granteeType": "USER", "id": "...", "privileges": ["SELECT", "ALTER"]},
#     {"granteeType": "ROLE", "id": "...", "privileges": ["SELECT"]}
#   ],
#   "availablePrivileges": ["SELECT", "ALTER", "MANAGE_GRANTS", ...]
# }
```

Set Grants

Update the grants for an entity. **Note:** This replaces the existing grants list.

```
new_grants = [
    {
        "granteeType": "USER",
        "id": "user-uuid-...",
        "privileges": ["SELECT"]
```

```
    }
]

client.catalog.set_grants(entity_id, new_grants)
```

Source: reflections.md

Guide: Reflections Management

Reflections are Dremio's query acceleration technology. DremioFrame allows you to manage them programmatically.

Listing Reflections

View all reflections in your Dremio environment.

```
from dremioframe.client import DremioClient

client = DremioClient(...)

# List all reflections
reflections = client.admin.list_reflections()
for r in reflections['data']:
    print(f"Name: {r['name']}, Status: {r['status']['availability']}")
```

Creating Reflections

You can create Raw or Aggregation reflections on a dataset.

```
# Create a Raw Reflection
client.admin.create_reflection(
    dataset_id="<dataset-uuid>",
    name="raw_sales_reflection",
    type="RAW",
    display_fields=["sale_date", "amount", "region"]
)

# Create an Aggregation Reflection
client.admin.create_reflection(
    dataset_id="<dataset-uuid>",
    name="agg_sales_by_region",
    type="AGGREGATION",
    dimension_fields=["region", "sale_date"],
    measure_fields=["amount"]
)
```

Refreshing Reflections

Trigger an immediate refresh of a reflection.

```
client.admin.refresh_reflection("<reflection-uuid>")
```

Automating Refreshes

While Dremio has internal scheduling, you might want to trigger refreshes as part of an external pipeline (e.g., immediately after data ingestion).

```
from dremioframe.orchestration import Pipeline, RefreshReflectionTask

pipeline = Pipeline("ingest_and_refresh")

# ... Ingestion tasks ...

refresh = RefreshReflectionTask(
    name="refresh_sales_agg",
    client=client,
    reflection_id="<reflection-uuid>"
)

pipeline.add_task(refresh)
# refresh.set_upstream(ingest_task)

pipeline.run()
```

Checking Status

You can poll for reflection status to ensure it's available.

```
status = client.admin.get_reflection("<reflection-uuid>")
print(status['status']['availability']) # e.g., 'AVAILABLE', 'EXPIRING', 'FAILED'
```

Source: security.md

Security Best Practices

Ensuring the security of your data and credentials is paramount. Here are best practices for using DremioFrame securely.

Credential Management

Never Hardcode Credentials

BAD:

```
client = DremioClient(pat="my-secret-token")
```

GOOD:
Use environment variables. DremioFrame automatically looks for `DREMIO_PAT`, `DREMIO_PROJECT_ID`, etc.

```
export DREMIO_PAT="my-secret-token"
```

```
client = DremioClient() # Reads from env
```

Using .env Files

For local development, use a `.env` file but **add it to `.gitignore`**.

.env

```
DREMIO_PAT=...
```

.gitignore

```
.env
```

Network Security

TLS/SSL

Always use TLS (HTTPS) when connecting to Dremio, especially over public networks.

Dremio Cloud: TLS is enforced (HTTPS/443).

Dremio Software: Enable TLS on the coordinator and set `tls=True` in the client.

```
client = DremioClient(..., tls=True)
```

Certificate Verification

Do not disable certificate verification (`disable_certificate_verification=True`) in production. This leaves you vulnerable to Man-in-the-Middle (MITM) attacks.

Least Privilege

Personal Access Tokens (PATs)

Create PATs with the minimum necessary expiration time. Rotate them regularly.

Service Accounts

For production pipelines, use a dedicated Service Account (if available in your Dremio edition) or a dedicated user account with restricted permissions, rather than your personal admin account.

Role-Based Access Control (RBAC)

Ensure the user/role used by DremioFrame only has access to the datasets and spaces it needs.

Read-Only: If the pipeline only reads data, grant `SELECT` only.

Write: Grant `CREATE TABLE`, `INSERT`, `UPDATE` only on specific target folders/spaces.

Injection Prevention

Parameterized Queries

While DremioFrame's builder API generates safe SQL, be careful when using raw SQL with `client.query()`. Avoid f-strings with untrusted user input.

BAD:

```
user_input = "'; DROP TABLE users; --"
client.query(f"SELECT * FROM table WHERE id = '{user_input}'")
```

GOOD:
Validate and sanitize inputs before constructing SQL strings, or use the Builder API which handles quoting.

```
# Builder API handles quoting
client.table("table").filter(f"id = '{sanitized_input}'")
```

Note: Dremio Flight currently supports parameter binding in limited contexts; DremioFrame relies on string construction, so input validation is key.

Source: security_patterns.md

Security Patterns

This guide covers advanced security patterns using Dremio's governance features.

1. Row-Level Security (RLS) with Lookup Tables

Instead of hardcoding users in RLS policies, use a lookup table to manage permissions dynamically.

Step 1: Create Lookup Table

Create a table `admin.permissions` mapping users to regions.

user_email	region
alice@co.com	NY
bob@co.com	CA

Step 2: Create Policy Function

The function checks if the current user matches the region in the lookup table.

```
CREATE FUNCTION check_region_access(r VARCHAR)
RETURNS BOOLEAN
RETURN SELECT count(*) > 0 FROM admin.permissions WHERE user_email = query_user() AND region = r;
```

Step 3: Apply Policy

```
client.admin.apply_row_access_policy("sales", "check_region_access(region)")
```

2. Hierarchy-Based Access

Allow managers to see data for their entire hierarchy.

Store the hierarchy in a flattened table or use recursive CTEs (if supported) in the policy function.

Common pattern: `path` column (e.g., `/US/NY/Sales`). Policy: `user_path LIKE row_path || '%'`.

3. Column Masking Patterns

Dynamic Masking based on Role

```
# Mask email for non-HR users
client.admin.create_policy_function(
    "mask_email",
    "email VARCHAR",
    "VARCHAR",
    "CASE WHEN is_member('HR') THEN email ELSE '***@***.com' END"
)

client.admin.apply_masking_policy("employees", "email", "mask_email(email)")
```

Format Preserving Masking

If downstream tools expect a valid email format, mask the characters but keep the structure.

```
-- Simple example
CASE WHEN is_member('HR') THEN email ELSE 'user_' || hash(email) || '@masked.com' END
```

Source: spaces_folders.md

Space and Folder Management

DremioFrame provides methods to manage Spaces and Folders in both Dremio Cloud and Dremio Software.

Methods

`create_folder`

Creates a folder using SQL. This is the primary method for creating folders in Dremio Cloud and Iceberg Catalogs.

```
client.admin.create_folder("my_space.my_folder")
```

Syntax: `CREATE FOLDER [IF NOT EXISTS] <folder_name>`

`create_space`

Creates a Space using the REST API. This is specific to **Dremio Software**.

```
client.admin.create_space("NewSpace")
```

`create_space_folder`

Creates a folder within a Space using the REST API. This is specific to **Dremio Software**.

```
client.admin.create_space_folder("NewSpace", "SubFolder")
```

Differences between Cloud and Software

Dremio Cloud: Uses `create_folder` for all folder creation. Top-level folders act as spaces.

Dremio Software: Uses `create_space` for top-level containers (Spaces) and `create_space_folder` for folders within them. `create_folder` can also be used if the backend supports the SQL syntax (e.g. Nessie/Iceberg sources).

Source: tags.md

Dataset Tagging

DremioFrame allows you to manage tags for datasets (tables and views) using the `Catalog` client. Tags are useful for organizing and classifying data assets (e.g., "PII", "Gold", "Deprecated").

Usage

Access tagging methods via `client.catalog`.

Get Tags

Retrieve the current tags for a dataset using its ID.

```
from dremioframe.client import DremioClient

client = DremioClient()
dataset_id = "..." # Get ID via client.catalog.get_entity(...)

tags = client.catalog.get_tags(dataset_id)
print(tags) # e.g. ["pii", "sales"]
```

Set Tags

Set the tags for a dataset. **Note:** This overwrites existing tags.

```
# To safely update tags, it's recommended to fetch the current version first
# to avoid conflicts if tags were modified concurrently.

# 1. Get current tag info (includes version)
tag_info = client.catalog.get_tag_info(dataset_id)
current_version = tag_info.get("version")
current_tags = tag_info.get("tags", [])

# 2. Modify tags
new_tags = current_tags + ["new-tag"]

# 3. Set tags with version
client.catalog.set_tags(dataset_id, new_tags, version=current_version)
```

Remove Tags

To remove all tags, pass an empty list.

```
client.catalog.set_tags(dataset_id, [], version=current_version)
```

Source: udf.md

UDF Manager

DremioFrame provides a pythonic interface to manage SQL User Defined Functions (UDFs).

Usage

Access the UDF manager via `client.udf`.

Create a UDF

```
# CREATE FUNCTION my_space.add_ints (x INT, y INT) RETURNS INT RETURN x + y

# Option 1: Using a dictionary for arguments
client.udf.create(
    name="my_space.add_ints",
    args={"x": "INT", "y": "INT"},
    returns="INT",
    body="x + y",
    replace=True
)

# Option 2: Using a string for arguments (useful for complex types)
client.udf.create(
    name="my_space.complex_func",
    args="x INT, y STRUCT<a INT, b INT>",
    returns="INT",
    body="x + y.a",
    replace=True
)
```

Drop a UDF

```
client.udf.drop("my_space.add_ints", if_exists=True)
```

List UDFs

```
# List all functions matching 'add'
funcs = client.udf.list(pattern="add")
for f in funcs:
```

```
    print(f["ROUTINE_NAME"])
```

Source: agent.md

DremioAgent Class

The `DremioAgent` class is the core of the AI capabilities in `dremioframe`. It uses LangGraph and LangChain to create an agent that can understand your request, consult documentation, inspect the Dremio catalog, and generate code, SQL, or API calls.

Purpose

The `DremioAgent` is designed to:

- **Generate Python Scripts**: Create complete, runnable scripts using `dremioframe` to automate tasks.
- **Generate SQL Queries**: Write complex SQL queries, validating table names and columns against the actual catalog.
- **Generate API Calls**: Construct cURL commands for the Dremio REST API by referencing the documentation.

Constructor

```
class DremioAgent(
    model: str = "gpt-4o",
    api_key: Optional[str] = None,
    llm: Optional[BaseChatModel] = None,
    memory_path: Optional[str] = None,
    context_folder: Optional[str] = None
)
```

Arguments

Argument	Type	Default	Description
`model`	`str`	`"gpt-4o"`	The name of the LLM model to use. Supported providers are OpenAI, Anthropic, and Google.
`api_key`	`Optional[str]`	`None`	The API key for the chosen model provider. If not provided, the agent will look for the corresponding environment variable (`OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GOOGLE_API_KEY`).
`llm`	`Optional[BaseChatModel]`	`None`	A pre-configured LangChain Chat Model instance. If provided, `model` and `api_key` arguments are ignored. Use this to support other providers (AWS Bedrock, Ollama, Azure, etc.).
`memory_path`	`Optional[str]`	`None`	Path to a SQLite database file for persisting conversation history. If provided, conversations can be resumed using a `session_id`. Requires `langgraph-checkpoint-sqlite` (included in `ai` optional dependencies).

| `context_folder` | `Optional[str]` | `None` | Path to a folder containing additional context files (schemas, documentation, etc.). The agent can list and read these files when generating code or SQL. |

Supported Models

The `model` argument supports string identifiers for major providers. The agent automatically selects the correct LangChain class based on the string.

OpenAI

Requires `OPENAI_API_KEY`

˜gpt-4o` (Default)

˜gpt-4-turbo`

˜gpt-3.5-turbo`

Anthropic

Requires `ANTHROPIC_API_KEY`

˜claude-3-opus-20240229`

˜claude-3-sonnet-20240229`

˜claude-3-haiku-20240307`

Google Gemini

Requires `GOOGLE_API_KEY`

˜gemini-1.5-pro`

˜gemini-pro`

Custom LLMs

To use a model provider not natively supported by the string shortcuts (like AWS Bedrock, Ollama, or Azure OpenAI), you can instantiate the LangChain model object yourself and pass it to the `llm` argument.

Example: Local LLM (Ollama)

Use `ChatOpenAI` pointing to a local server (e.g., Ollama running Llama 3).

```
from dremioframe.ai.agent import DremioAgent
from langchain_openai import ChatOpenAI

# Connect to local Ollama instance
```

```
local_llm = ChatOpenAI(
    base_url="http://localhost:11434/v1",
    api_key="ollama", # Required but ignored by Ollama
    model="llama3",
    temperature=0
)

agent = DremioAgent(llm=local_llm)
print(agent.generate_sql("List all tables in the 'Samples' space"))
```

Example: AWS Bedrock

Use `ChatBedrock` from `langchain-aws`.

```
pip install langchain-aws
```

```
from dremioframe.ai.agent import DremioAgent
from langchain_aws import ChatBedrock

bedrock_llm = ChatBedrock(
    model_id="anthropic.claude-3-sonnet-20240229-v1:0",
    model_kwargs={"temperature": 0}
)

agent = DremioAgent(llm=bedrock_llm)
```

Example: Azure OpenAI

Use `AzureChatOpenAI` from `langchain-openai`.

```
from dremioframe.ai.agent import DremioAgent
from langchain_openai import AzureChatOpenAI

azure_llm = AzureChatOpenAI(
    azure_deployment="my-gpt-4-deployment",
    openai_api_version="2023-05-15",
    temperature=0
)

agent = DremioAgent(llm=azure_llm)
```

Memory Persistence

By default, the `DremioAgent` does not persist conversation history between sessions. Each invocation starts fresh. To enable memory persistence, provide a `memory_path` when creating the agent.

Basic Usage

```
from dremioframe.ai.agent import DremioAgent

# Create agent with memory
agent = DremioAgent(memory_path="./agent_memory.db")

# First conversation
code1 = agent.generate_script("Create a table from CSV", session_id="user123")

# Later, in the same or different session
code2 = agent.generate_script("Now add a column to that table", session_id="user123")
# The agent remembers the previous conversation and knows which table you're referring
to
```

How It Works

SQLite Database: Conversation history is stored in a SQLite database at the specified path.

Session ID: Each conversation thread is identified by a `session_id`. Use the same `session_id` to continue a conversation.

Thread Isolation: Different `session_id` values create independent conversation threads.

Example: Multi-Turn Conversation

```
agent = DremioAgent(memory_path="./conversations.db")

session = "project_alpha"

# Turn 1
sql1 = agent.generate_sql("Get all users from the users table", session_id=session)
print(sql1)  # SELECT * FROM users

# Turn 2 - Agent remembers context
sql2 = agent.generate_sql("Filter to only active users", session_id=session)
print(sql2)  # SELECT * FROM users WHERE status = 'active'

# Turn 3 - Agent still has context
sql3 = agent.generate_sql("Add their email addresses", session_id=session)
print(sql3)  # SELECT *, email FROM users WHERE status = 'active'
```

Managing Sessions

```
# Different projects/users can have separate conversations
agent.generate_script("Create ETL pipeline", session_id="project_alpha")
agent.generate_script("Create dashboard", session_id="project_beta")
```

```
# Each session maintains its own context
```

Clearing Memory

To clear all conversation history, simply delete the SQLite database file:

```
import os
os.remove("./agent_memory.db")
```

Context Folder

The `context_folder` feature allows the agent to access files from a specified directory. This is useful for:

- Reading project-specific schemas
- Referencing data dictionaries
- Using custom documentation
- Accessing configuration files

Basic Usage

```
from dremioframe.ai.agent import DremioAgent

# Create agent with context folder
agent = DremioAgent(context_folder="./project_docs")

# Agent can now reference files in ./project_docs
script = agent.generate_script(
    "Create a table based on the schema in schema.sql"
)
```

Example: Project Documentation

```
project_docs/
├── schema.sql
├── data_dictionary.md
└── business_rules.txt
```

```
agent = DremioAgent(context_folder="./project_docs")

# Agent can read these files when needed
sql = agent.generate_sql(
    "Create a query following the business rules in business_rules.txt"
)
```

How It Works

When `context_folder` is set, the agent gains two additional tools:

`list_context_files()`: Lists all files in the context folder.

`read_context_file(file_path)`: Reads the content of a specific file.

The agent automatically uses these tools when your prompt mentions files or context.

Example: Schema-Driven Development

```
# schema.sql contains:
# CREATE TABLE customers (
#     id INT,
#     name VARCHAR(100),
#     email VARCHAR(100),
#     created_at TIMESTAMP
# )

agent = DremioAgent(context_folder="./schemas")

# Agent reads schema.sql and generates appropriate code
code = agent.generate_script(
     "Create a Python script to validate that the customers table matches the schema in
schema.sql"
)
```

Combining Memory and Context

```
agent = DremioAgent(
    memory_path="./memory.db",
    context_folder="./project_docs"
)

session = "data_migration"

# Turn 1
agent.generate_script(
    "Read the source schema from source_schema.sql",
    session_id=session
)

# Turn 2 - Agent remembers the source schema from Turn 1
agent.generate_script(
    "Now generate a migration script to the target schema in target_schema.sql",
    session_id=session
)
```

Best Practices

Memory Persistence

Use descriptive session IDs: `"user_123_project_alpha"` is better than `"session1"`.

Clean up old sessions: Periodically delete the database or implement session expiration.

Don't share session IDs: Each user or project should have unique session IDs.

Context Folder

Keep files small: Large files may exceed LLM context limits.

Use clear file names: `customer_schema.sql` is better than `schema1.sql`.

Organize by topic: Group related files in subdirectories.

Mention files explicitly: "Use the schema in schema.sql" is clearer than "use the schema".

Performance

Limit context folder size: Too many files can slow down the agent.

Use memory sparingly: Long conversation histories increase token usage.

Clear memory between projects: Start fresh for unrelated work.

Source: api.md

AI API Call Generation

The `generate-api` command (or `agent.generate_api_call()` method) generates a cURL command for the Dremio REST API. It uses the library's documentation and native Dremio documentation (if available) to find the correct endpoint and payload.

Usage via CLI

The `generate-api` command takes the following arguments:

`prompt` (Required): The natural language description of the API call you want to generate.

`model` / `-m` (Optional): The LLM model to use. Defaults to `gpt-4o`.

Examples:

```
# Generate a cURL command to list all sources (using default gpt-4o)
dremio-cli generate-api "List all sources"

# Generate a command to create a space using Claude 3 Opus
dremio-cli generate-api "Create a space named 'Marketing'" --model claude-3-opus
```

```
# Generate a command to trigger a reflection refresh
dremio-cli generate-api "Refresh the reflection with ID 12345"
```

Usage via Python

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()
curl = agent.generate_api_call("List all sources")
print(curl)
```

How it Works

- **Context Awareness**: The agent is aware of the API specification (via documentation).
- **Security**: The agent generates the command but does not execute it automatically. You have full control to review and run the output.

Source: cli_chat.md

Interactive CLI Chat

The `dremioframe` CLI now includes an interactive chat mode, allowing you to converse with the Dremio Agent directly from your terminal.

Features

- **Natural Language Interface**: Ask questions, request scripts, or troubleshoot issues in plain English.
- **Context Awareness**: The agent maintains conversation history, allowing for follow-up questions.
- **Tool Access**: The agent can use all available tools (Catalog, Jobs, Reflections, etc.) during the chat.

Usage

To start the chat session, run:

```
dremioframe chat
```

You can also specify the model (default is `gpt-4o`):

```
dremioframe chat --model claude-3-opus
```

Example Session

```
$ dremioframe chat
Starting Dremio Agent Chat (gpt-4o)...
Type 'exit' or 'quit' to leave.

You: List the most recent failed jobs.
Thinking...
Agent: Here are the 3 most recent failed jobs:
1. Job ID: 123... - Error: Table not found
2. Job ID: 456... - Error: Syntax error
...
----------------------------------------
You: Analyze the first one.
Thinking...
Agent: I've analyzed job 123... The error "Table not found" suggests that the table
'sales.data' does not exist.
I checked the catalog and found 'sales.raw_data'. Did you mean that?
----------------------------------------
```

Source: data_quality.md

AI Data Quality Tools

The `DremioAgent` can automatically generate Data Quality (DQ) recipes for your datasets.

Features

1. Automated Recipe Generation

The agent inspects a dataset's schema (columns and types) and generates a YAML configuration file for the DremioFrame Data Quality framework. It intelligently suggests checks like `not_null` for IDs and `unique` for primary keys.

Tool: `generate_dq_checks(table)`

Method: `agent.generate_dq_recipe(table)`

Usage Examples

Generating a Recipe

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()

# Generate a DQ recipe for a table
```

```
table_name = "sales.transactions"
recipe = agent.generate_dq_recipe(table_name)

print("Generated DQ Recipe:")
print(recipe)

# You can save this to a file
with open("dq_checks.yaml", "w") as f:
    f.write(recipe)
```

Example Output

```
table: sales.transactions
checks:
  - column: transaction_id
    tests:
      - not_null
      - unique
  - column: amount
    tests:
      - positive
```

Source: document_extraction.md

Document Extraction to Tables

The DremioAgent can extract structured data from documents (PDFs, markdown files) in your context folder and generate code to create or insert data into Dremio tables.

Installation

```
pip install dremioframe[document]
```

Usage

Basic Workflow

1. Place documents in a context folder
2. Initialize agent with context folder
3. Ask agent to extract data with a specific schema
4. Agent generates code to create/insert into Dremio

Example: Extract Invoice Data from PDFs

```
from dremioframe.ai.agent import DremioAgent

# Initialize agent with context folder
agent = DremioAgent(
    model="gpt-4o",
    context_folder="./invoices"
)

# Ask agent to extract structured data
script = agent.generate_script("""
Read all PDF files in the context folder that contain invoice data.
Extract the following fields: invoice_number, date, customer_name, total_amount.
Generate code to create a table 'invoices' with this schema and insert the extracted
data.
""")

print(script)
```

Example: Extract Product Catalog from Markdown

```
agent = DremioAgent(
    model="gpt-4o",
    context_folder="./product_docs"
)

script = agent.generate_script("""
Read all markdown files in the context folder.
Extract product information: product_id, name, category, price, description.
Create a table 'products' and insert the data.
""")
```

Supported Document Types

PDF Files

- Uses `pdfplumber` for text extraction
- Works with text-based PDFs
- Scanned PDFs require OCR (not currently supported)

Markdown Files

- Direct text reading
- Supports tables, lists, and structured content

Text Files

- Any `.txt`, `.md`, `.csv` files
- Read via `read_context_file` tool

How It Works

List Files: Agent uses `list_context_files()` to see available documents

Read Content: Agent uses `read_pdf_file()` or `read_context_file()` to extract text

Extract Data: LLM analyzes content and extracts structured data

Generate Code: Agent creates Python code to load data into Dremio

Best Practices

Provide Clear Schema

Be specific about the fields you want extracted:

```
"""
Extract these exact fields:
- invoice_number (string)
- date (YYYY-MM-DD format)
- customer_name (string)
- total_amount (decimal)
"""
```

Use Examples

Provide example data if documents have varying formats:

```
"""
Example invoice format:
Invoice #: INV-12345
Date: 2024-01-15
Customer: Acme Corp
Total: $1,500.00
"""
```

Handle Missing Data

Specify how to handle missing fields:

```
"""
If a field is missing, use NULL or empty string.
"""
```

Limitations

- **OCR Not Supported**: Scanned PDFs won't work (text-based only)
- **Token Limits**: Very large documents may exceed LLM context windows
- **Accuracy**: Extraction quality depends on document structure and LLM capabilities
- **Performance**: Large batches of documents may be slow

Future Enhancements

- Image OCR support (via Tesseract or cloud services)
- Table extraction from PDFs
- Multi-page document handling
- Batch processing optimization

Source: generation.md

AI Script Generation

`dremioframe` includes an AI-powered module that can generate Python scripts for you based on natural language prompts. It uses LangChain and supports OpenAI, Anthropic, and Google Gemini models.

Installation

To use the AI features, you must install the optional `ai` dependencies:

```
pip install dremioframe[ai]
```

Configuration

You need to set the API key for your chosen model provider in your environment variables:

- **OpenAI**: `OPENAI_API_KEY`
- **Anthropic**: `ANTHROPIC_API_KEY`
- **Google Gemini**: `GOOGLE_API_KEY`

Usage via CLI

You can generate scripts directly from the command line using the `generate` command.

```
# Generate a script to list all sources
dremio-cli generate "List all sources and print their names" --output list_sources.py

# Use a specific model (default is gpt-4o)
dremio-cli generate "Create a view named 'sales_summary' from 'sales_raw'" --model
claude-3-opus --output create_view.py

# Use a prompt from a file
dremio-cli generate prompt.txt --output script.py
```

Usage via Python

You can also use the `DremioAgent` class directly in your Python code.

Arguments for `generate_script`

`prompt` (str): The natural language request describing the script you want to generate.

`output_file` (Optional[str]): The path to save the generated script to. If not provided, the script is returned as a string.

Using Custom LLMs

You can use any LangChain-compatible chat model by passing it to the `DremioAgent` constructor.

```
from dremioframe.ai.agent import DremioAgent
from langchain_openai import ChatOpenAI

# Initialize with a custom LLM instance
custom_llm = ChatOpenAI(model="gpt-4-turbo", temperature=0.5)
agent = DremioAgent(llm=custom_llm)

# Generate a script
prompt = "Write a script to connect to Dremio and list all spaces."
script = agent.generate_script(prompt)

print(script)
```

Example: Using a Local LLM (e.g., Ollama)

You can use a local LLM running via Ollama or any other OpenAI-compatible server.

```
from dremioframe.ai.agent import DremioAgent
from langchain_openai import ChatOpenAI

# Connect to local Ollama instance
local_llm = ChatOpenAI(
```

```
    base_url="http://localhost:11434/v1",
    api_key="ollama", # Required but ignored
    model="llama3"
)

agent = DremioAgent(llm=local_llm)
agent.generate_script("List all sources")
```

Example: Using Amazon Bedrock

To use Amazon Bedrock, you need to install `langchain-aws`.

```
pip install langchain-aws
```

```
from dremioframe.ai.agent import DremioAgent
from langchain_aws import ChatBedrock

# Initialize Bedrock client
bedrock_llm = ChatBedrock(
    model_id="anthropic.claude-3-sonnet-20240229-v1:0",
    model_kwargs={"temperature": 0.1}
)

agent = DremioAgent(llm=bedrock_llm)
agent.generate_script("Create a view from sales data")
```

How it Works

The agent has access to:

Library Documentation: It can list and read `dremioframe` documentation files.

Dremio Documentation: It can search and read native Dremio documentation (if available in `dremiodocs/`) to understand SQL functions and concepts.

It generates a complete Python script that includes:

Importing `DremioClient`.

Initializing the client (expecting `DREMIO_PAT` and `DREMIO_PROJECT_ID` env vars).

Performing the requested actions using the appropriate methods.

Source: governance.md

AI Governance Tools

The `DremioAgent` assists with governance tasks such as auditing access and

automating documentation.

Features

1. Access Auditing

The agent can list privileges granted on specific entities (tables, views, folders, spaces).

Tool: `show_grants(entity)`

2. Automated Documentation

The agent can inspect a dataset's schema and automatically generate a Wiki description and suggest Tags.

Method: `agent.auto_document_dataset(path)`

Usage Examples

Auditing Access

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()

# Check who has access to a sensitive table
response = agent.generate_script("Show grants for table 'finance.payroll'")
print(response)
```

Auto-Documenting a Dataset

```
# Generate documentation for a dataset
path = "sales.transactions"
documentation = agent.auto_document_dataset(path)

print("Generated Documentation:")
print(documentation)

# You can then use the client to apply this documentation
# import json
# doc_data = json.loads(documentation)
# client.catalog.update_wiki(dataset_id, doc_data['wiki'])
# client.catalog.set_tags(dataset_id, doc_data['tags'])
```

Source: mcp_client.md

Using MCP Tools in Dremio Agent

[!NOTE]
This guide explains how to use external MCP tools *within* the Dremio Agent. If you want to use DremioFrame as an MCP **Server** for other clients (like Claude), see MCP Server.

The DremioAgent supports integration with **Model Context Protocol (MCP)** servers, allowing you to extend the agent with custom tools from any MCP-compatible server.

What is MCP?

The Model Context Protocol is an open standard introduced by Anthropic that standardizes how AI systems integrate with external tools, systems, and data sources. It provides a universal interface for:

- File system access
- Database connections
- API integrations
- Custom tool implementations

Installation

```
pip install dremioframe[mcp]
```

Usage

Basic Example

```
from dremioframe.ai.agent import DremioAgent

# Configure MCP servers
mcp_servers = {
    "filesystem": {
        "transport": "stdio",
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-filesystem", "/path/to/data"]
    },
    "github": {
        "transport": "stdio",
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-github"]
    }
}

# Initialize agent with MCP servers
agent = DremioAgent(
```

```
    model="gpt-4o",
    mcp_servers=mcp_servers
)

# The agent now has access to tools from both MCP servers
script = agent.generate_script(
    "List all files in the data directory and create a table from the CSV files"
)
```

Available MCP Servers

Popular MCP servers include:

- **@modelcontextprotocol/server-filesystem**: File system operations
- **@modelcontextprotocol/server-github**: GitHub API access
- **@modelcontextprotocol/server-postgres**: PostgreSQL database access
- **@modelcontextprotocol/server-sqlite**: SQLite database access

See the MCP Server Registry for more.

Transport Protocols

MCP supports three transport protocols for client-server communication. Choose based on your deployment scenario:

`stdio` (Standard Input/Output)

Best for: Local integrations, command-line tools, development

How it works: The client launches the MCP server as a subprocess and communicates through standard input/output streams.

Characteristics:

- **Lowest latency** - No network overhead
- **Most secure** - No network exposure
- **Simplest setup** - No network configuration needed
- **Single client only** - One-to-one relationship
- **Same machine only** - Cannot communicate across network

When to use:

- Local AI tools running on your machine
- Development and testing
-

Security-sensitive operations

Command-line integrations

Example:

```
mcp_servers = {
    "local_files": {
        "transport": "stdio",
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-filesystem", "/data"]
    }
}
```

`http` (Streamable HTTP) - **Recommended for Production**

Best for: Web applications, remote servers, multi-client environments

How it works: MCP server runs as an independent HTTP server. Clients connect over HTTP/HTTPS. Optionally uses Server-Sent Events (SSE) for server-to-client streaming.

Characteristics:

- **Remote access** - Works across networks
- **Multiple clients** - Supports concurrent connections
- **Authentication** - Supports JWT, API keys, etc.
- **Scalable** - Enterprise-ready
- **Modern standard** - Latest MCP specification
- **More complex setup** - Requires server deployment

When to use:

- Web-based AI applications
- Cloud-hosted MCP servers
- Multiple AI clients accessing same server
- Enterprise integrations requiring authentication
- Production deployments

Example:

```
mcp_servers = {
    "remote_api": {
        "transport": "http",
        "url": "https://mcp-server.example.com",
        "headers": {
            "Authorization": "Bearer YOUR_TOKEN"
        }
```

```
    }
}
```

OAuth 2.0 Support:

Yes! The `http` transport supports OAuth authentication via the `headers` parameter. You can pass OAuth tokens (Bearer tokens, API keys, etc.) with every request:

```
mcp_servers = {
    "oauth_server": {
        "transport": "http",
        "url": "https://secure-mcp-server.example.com/mcp",
        "headers": {
            "Authorization": "Bearer YOUR_OAUTH_TOKEN",
            "X-Custom-Header": "additional-auth-data"
        }
    }
}
```

Note: The current implementation passes static headers. For OAuth token refresh, you'll need to manage token renewal externally and reinitialize the agent with updated tokens.

`sse` (Server-Sent Events) - **Legacy/Deprecated**

Best for: Backwards compatibility only

How it works: Uses Server-Sent Events for server-to-client streaming, paired with HTTP POST for client-to-server requests.

Status: ⚠ **Deprecated** - Replaced by Streamable HTTP

When to use:

Only for compatibility with older MCP servers

Not recommended for new implementations

Quick Comparison

Feature	stdio	http	sse
Latency	Lowest	Medium	Medium
Security	Highest (local)	Good (with auth)	Good (with auth)
Setup Complexity	Simplest	Moderate	Moderate
Multi-client	No	Yes	Yes
Remote Access	No	Yes	Yes
Authentication	N/A	Yes	Yes
Status	Active	Recommended	⚠ Deprecated

Configuration Format

Each MCP server configuration requires:

transport: Communication method (`"stdio"` or `"http"`)

command (stdio only): Executable command (e.g., `"npx"`, `"python"`)

args (stdio only): Command arguments (list of strings)

url (http only): Server URL

headers (http only, optional): Authentication headers

Requirements

Node.js: Required for npx-based MCP servers

langchain-mcp-adapters: Installed automatically with `pip install dremioframe[mcp]`

Example: File System Integration

```
mcp_servers = {
    "files": {
        "transport": "stdio",
        "command": "npx",
        "args": ["-y", "@modelcontextprotocol/server-filesystem", "/data"]
    }
}

agent = DremioAgent(mcp_servers=mcp_servers)

# Agent can now read files, list directories, etc.
script = agent.generate_script(
    "Read all CSV files in /data and merge them into a single Dremio table"
)
```

Troubleshooting

MCP Server Not Found

Ensure Node.js and npx are installed:

```
node --version
npx --version
```

Import Error

If you see "langchain-mcp-adapters not found":

```
pip install dremioframe[mcp]
```

Server Connection Issues

Check server logs and ensure the command/args are correct for your MCP server.

Source: mcp_server.md

Dremio Agent MCP Server

The Dremio Agent MCP Server exposes Dremio capabilities via the Model Context Protocol (MCP), allowing you to use Dremio tools directly within MCP-compliant AI clients like Claude Desktop or IDE extensions.

[!NOTE]
This guide explains how to use DremioFrame as an MCP **Server**. If you want to use external MCP tools *within* the Dremio Agent, see Using MCP Tools.

Installation

The MCP server requires the `mcp` optional dependency:

```
pip install "dremioframe[server]"
```

Configuration

To use the server, you need to configure your MCP client to run the `dremio-cli mcp start` command.

You can generate the configuration JSON using the CLI:

```
dremio-cli mcp config
```

This will output a JSON structure similar to:

```
{
  "mcpServers": {
    "dremio-agent": {
      "command": "/path/to/python",
      "args": [
        "-m",
        "dremioframe.cli",
        "mcp",
        "start"
      ],
      "env": {
        "DREMIO_PAT": "your_pat_here",
        "DREMIO_PROJECT_ID": "your_project_id_here"
      }
    }
  }
```

```
}
```

Environment Variables

Ensure you set the correct environment variables in the `env` section of the configuration:

Dremio Cloud: `DREMIO_PAT`, `DREMIO_PROJECT_ID`

Dremio Software: `DREMIO_SOFTWARE_HOST`, `DREMIO_SOFTWARE_PAT` (or `DREMIO_SOFTWARE_USER`/`PASSWORD`)

Available Tools

The server exposes the following tools to the AI model:

- **`list_catalog(path)`**: List contents of the catalog.
- **`get_entity(path)`**: Get details of a dataset or container.
- **`query_dremio(sql)`**: Execute a SQL query and get results as JSON.
- **`list_reflections()`**: List all reflections.
- **`get_job_profile(job_id)`**: Get details of a job.
- **`create_view(path, sql)`**: Create a virtual dataset.
- **`list_available_docs()`**: List available documentation files.

Resources

The server exposes documentation as MCP Resources. The AI client can read these resources to understand how to use `dremioframe` and Dremio SQL.

- **`dremio://docs/{category}/{file}`**: Read library documentation.
- **`dremio://dremiodocs/{category}/{file}`**: Read Dremio native documentation.

Usage Example (Claude Desktop)

1. **Configure**: Add the server config to your Claude Desktop configuration.
2. **Ask**: "How do I query a table using dremioframe?" or "Write a script to list all reflections."
3. **Context**: Claude will use the `list_available_docs` tool to find relevant documentation, read it via the Resources, and then generate the code for you.

Source: observability.md

AI Observability Tools

The `DremioAgent` includes powerful observability tools to help you monitor and debug your Dremio environment.

Features

1. Job Analysis

The agent can inspect job details, including status, duration, and error messages.

Tool: `get_job_details(job_id)`

Tool: `list_recent_jobs(limit)`

2. Failure Analysis

The agent can analyze failed jobs and provide actionable explanations and fixes using its LLM capabilities.

Method: `agent.analyze_job_failure(job_id)`

Usage Examples

Listing Recent Jobs

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()

# Ask the agent to list jobs
response = agent.generate_script("List the 5 most recent failed jobs")
print(response)
```

Analyzing a Failed Job

```
# Analyze a specific job failure
job_id = "12345-67890-abcdef"
analysis = agent.analyze_job_failure(job_id)

print("Failure Analysis:")
print(analysis)
```

Interactive Debugging

You can also use the agent to interactively debug issues:

```
agent.agent.invoke({"messages": [("user", "Why did my last query fail?")]})
```

Source: optimization.md

AI SQL Optimization Tools

The `DremioAgent` can act as your personal database administrator, analyzing query plans and suggesting optimizations.

Features

1. Query Plan Analysis

The agent runs `EXPLAIN PLAN` on your SQL query and analyzes the execution plan to identify bottlenecks, such as full table scans or expensive joins.

Tool: `optimize_query(query)`

Method: `agent.optimize_sql(query)`

Usage Examples

Optimizing a Query

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()

# A potentially slow query
query = """
SELECT *
FROM sales.transactions t
JOIN crm.customers c ON t.customer_id = c.id
WHERE t.amount > 1000
"""

# Ask the agent to optimize it
optimization = agent.optimize_sql(query)

print("Optimization Suggestions:")
print(optimization)
```

Example Output

The agent might suggest:

- Creating a Raw Reflection on `sales.transactions` covering `customer_id` and `amount`.

- Filtering `sales.transactions` before joining.

- Using a specific join type if applicable.

Source: overview.md

Dremio AI Agent

DremioFrame includes a powerful AI Agent powered by Large Language Models (LLMs) that acts as your intelligent co-pilot for Dremio development and administration.

Capabilities

The AI Agent is designed to assist with:

- Generation: Generate Python Scripts, SQL and cURL commands to interact with your Dremio instance

- Observability: Analyze job failures and monitor system health.

- Reflections: Recommend and manage reflections for query acceleration.

- Governance: Audit access and automate dataset documentation.

- Data Quality: Generate data quality recipes automatically.

- SQL Optimization: Analyze query plans and suggest performance improvements.

- Interactive Chat: Converse with the agent directly from the CLI.

note: this libraries embdedded agent is primarily meant as a code generation assist tool, not meant as an alternative to the integrated Dremio agent for deeper administration and natural language analytics. Login to your Dremio instance's UI to leverage integrated agent.

Getting Started

To use the AI features, you need to install the optional dependencies:

```
pip install dremioframe[ai]
```

You also need to set your LLM API key in your environment variables.

Required Environment Variables

The AI agent supports multiple LLM providers. Set the appropriate environment variable for your chosen provider:

Provider	Environment Variable	How to Get
OpenAI	`OPENAI_API_KEY`	platform.openai.com/api-keys
Anthropic	`ANTHROPIC_API_KEY`	console.anthropic.com/settings/keys
Google	`GOOGLE_API_KEY`	aistudio.google.com/app/apikey

Example `.env` file:

```
# Choose one based on your provider
OPENAI_API_KEY=sk-proj-...
# ANTHROPIC_API_KEY=sk-ant-...
# GOOGLE_API_KEY=AIza...

# Dremio credentials (required for agent to access catalog)
DREMIO_PAT=your_personal_access_token
DREMIO_PROJECT_ID=your_project_id
DREMIO_URL=data.dremio.cloud
```

Usage

You can use the agent programmatically in your Python scripts or interactively via the CLI.

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent(model="gpt-4o")
response = agent.generate_script("List all spaces")
print(response)
```

Source: reflections.md

AI Reflection Tools

The `DremioAgent` can help you manage and optimize Dremio Reflections, which are critical for query acceleration.

Features

1. Reflection Management

The agent can list existing reflections and create new ones.

Tool: `list_reflections()`

Tool: `create_reflection(dataset_id, name, type, fields)`

2. Smart Recommendations

The agent can analyze a SQL query and recommend the optimal reflection configuration (Raw vs. Aggregation) to accelerate it.

Method: `agent.recommend_reflections(query)`

Usage Examples

Listing Reflections

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()

# List all reflections
response = agent.generate_script("List all reflections in the system")
print(response)
```

Getting Recommendations

```
query = """
SELECT
    region,
    SUM(sales_amount) as total_sales
FROM sales.transactions
GROUP BY region
"""

recommendation = agent.recommend_reflections(query)
print("Recommended Reflection:")
print(recommendation)
```

Creating a Reflection

You can ask the agent to create a reflection based on a recommendation.

```
agent.agent.invoke({"messages": [("user", "Create an aggregation reflection on 'sales.transactions' for the query I just showed you.")]})
```

Source: sql.md

AI SQL Generation

The `generate-sql` command (or `agent.generate_sql()` method) generates a SQL query based on a natural language prompt. It uses the `list_catalog_items` and `get_table_schema` tools to inspect your Dremio catalog and validate table names and columns.

Usage via CLI

The `generate-sql` command takes the following arguments:

`prompt` (Required): The natural language description of the SQL query you want to generate.

`model` / `-m` (Optional): The LLM model to use. Defaults to `gpt-4o`.

Examples:

```
# Generate a query to select data from a specific table (using default gpt-4o)
dremio-cli generate-sql "Select the first 10 rows from the zips table in Samples"

# Generate a complex aggregation using Claude 3 Opus
dremio-cli generate-sql "Calculate the average population by state from the zips table"
--model claude-3-opus
```

Usage via Python

```
from dremioframe.ai.agent import DremioAgent

agent = DremioAgent()
sql = agent.generate_sql("Select the first 10 rows from the zips table in Samples")
print(sql)
```

How it Works

Context Awareness: The agent is aware of your Dremio environment's structure (via catalog tools).

Validation: It attempts to verify table names and columns against the actual catalog to prevent errors.

Security: The agent generates the SQL but does not execute it automatically. You have full control to review and run the output.

Source: charting.md

Charting

DremioFrame integrates with Matplotlib and Pandas to allow quick visualization of your data.

Prerequisites

Ensure `matplotlib` is installed:

```
pip install matplotlib
```

(It is installed by default with `dremioframe`)

Creating Charts

The `chart()` method collects data to a Pandas DataFrame and uses `df.plot()` to generate a chart.

```
# Create a bar chart
df.chart(kind="bar", x="category", y="count", title="Sales by Category")

# Save chart to file
df.chart(kind="line", x="date", y="sales", save_to="sales_trend.png")
```

Supported Kinds

˜line`

˜bar`

˜barh`

˜hist`

˜box`

˜kde`

˜density`

˜area`

˜pie`

˜scatter`

˜hexbin`

Customization

You can pass any argument supported by `pandas.DataFrame.plot()`:

```
df.chart(kind="scatter", x="age", y="income", c="red", s=50)
```

Source: plotting.md

Interactive Plotting with Plotly

DremioFrame supports interactive charts using Plotly.

Usage

Specify `backend="plotly"` in the `chart()` method.

```
# Create an interactive scatter plot
fig = df.chart(
    kind="scatter",
    x="gdpPercap",
    y="lifeExp",
    color="continent",
    size="pop",
    hover_name="country",
    log_x=True,
    title="Life Expectancy vs GDP",
    backend="plotly"
)

# Display in notebook
fig.show()

# Save to HTML
df.chart(..., backend="plotly", save_to="chart.html")
```

Supported Chart Types

˜line`

˜bar`

˜scatter`

˜pie`

˜histogram`

˜box`

˜violin`

˜area`

Dependencies

Requires `plotly` and `pandas`.
For static image export (e.g., `.png`), `kaleido` is required.

Source: profiling.md

Query Profile Analyzer

Analyze and visualize Dremio query execution profiles.

Usage

Get Job Profile

```
profile = client.admin.get_job_profile("job_id_123")
```

Summary

Print a summary of the job execution.

```
profile.summary()
# Job ID: job_id_123
# State: COMPLETED
# Start: 1600000000000
# End: 1600000005000
```

Visualize

Visualize the execution timeline using Plotly.

```
# Display interactive chart
profile.visualize().show()

# Save to HTML
profile.visualize(save_to="profile.html")
```

Source: api_compatibility.md

Dremio API Compatibility Guide

`dremioframe` is designed to work seamlessly with Dremio Cloud, Dremio Software v26+, and Dremio Software v25. This document outlines how the library handles the differences between these versions.

Supported Modes

The `DremioClient` accepts a `mode` parameter to configure behavior:

Mode	Description	Default Auth	Base URL Pattern
`cloud`	Dremio Cloud	PAT + Project ID	`https://api.dremio.cloud/v0`
`v26`	Software v26+	PAT (or User/Pass)	`https://{host}:9047/api/v3`
`v25`	Software v25	User/Pass (or PAT)	`https://{host}:9047/api/v3`

Key Differences & Handling

1. Authentication

- **Cloud**: Uses Personal Access Tokens (PAT) exclusively. Requires `DREMIO_PAT` and `DREMIO_PROJECT_ID`.
- **Software v26**: Supports PATs natively. Can also use Username/Password to obtain a token via `/apiv3/login`.
- **Software v25**: Primarily uses Username/Password via `/apiv2/login` (legacy) or `/login` to obtain a token.

`dremioframe` automatically selects the correct login endpoint and token handling based on the `mode`.

2. API Endpoints

- **Project ID**: Dremio Cloud API paths typically require a project ID (e.g., `/v0/projects/{id}/catalog`). Dremio Software paths do not (e.g., `/api/v3/catalog`).
- **Base URL**: Cloud uses `/v0`, Software uses `/api/v3`.

The library's internal `_build_url` methods in `Catalog` and `Admin` classes automatically append the project ID for Cloud mode and omit it for Software mode.

3. Feature Support

Feature	Cloud	Software	Handling
Create Space	No (Use Folders)	Yes	`create_space` raises `NotImplementedError` in Cloud mode.
Reflections	Yes	Yes	Unified interface via `admin.list_reflections`, etc.
SQL Runner	Yes	Yes	Unified via `client.query()`.
Flight	`data.dremio.cloud`	`{host}:32010`	Port and endpoint auto-configured.

Best Practices

- **Use Environment Variables**: Set `DREMIO_PAT`, `DREMIO_PROJECT_ID` (for Cloud), or `DREMIO_SOFTWARE_PAT`, `DREMIO_SOFTWARE_HOST` (for Software) to switch environments easily without changing code.
- **Check Mode**: If writing scripts that run across environments, check `client.mode` before calling software-specific methods like `create_space`.

```
if client.mode == 'cloud':
    client.admin.create_folder("my_folder")
else:
    client.admin.create_space("my_space")
```

Source: aggregation.md

Aggregation

DremioFrame supports standard SQL aggregation using `group_by` and `agg`.

Group By

Use `group_by` to group rows by one or more columns.

```
df.group_by("state")
df.group_by("state", "city")
```

Aggregation

Use `agg` to define aggregation expressions. The keys become the new column names.

```
# Calculate average population by state
df.group_by("state").agg(
    avg_pop="AVG(pop)",
    max_pop="MAX(pop)",
    count="COUNT(*)"
).show()
```

This generates SQL like:

```
SELECT AVG(pop) AS avg_pop, MAX(pop) AS max_pop, COUNT(*) AS count
FROM table
GROUP BY state
```

Source: builder.md

Dataframe Builder

The `DremioBuilder` provides a fluent interface for querying data, similar to Ibis or PySpark.

Getting a Builder

Start by selecting a table from the client:

```
from dremioframe.client import DremioClient

client = DremioClient()
df = client.table('finance.bronze.transactions')
```

Querying Data

You can chain methods to build a query:

```
result = (
    df.select("transaction_id", "customer_id", "amount")
      .filter("amount > 1000")
      .limit(10)
      .collect()
)

print(result)
```

Methods

˜select(*cols)`: Select specific columns.

˜mutate(**kwargs)`: Add calculated columns. Example: `.mutate(total="price * quantity")`**

Conflict Resolution: If a mutated column name matches a selected column, the mutation takes precedence.

Implicit Select: If `select()` is not called, `mutate` preserves all existing columns (equivalent to `SELECT *, mutation AS alias`).

˜filter(condition)`: Add a WHERE clause.

˜limit(n)`: Limit the number of rows.

˜collect(library='polars')`: Execute the query and return a DataFrame. Supported libraries: `polars` (default), `pandas`.

˜show(n=20)`: Print the first `n` rows.

Query Explanation

You can view the execution plan for a query using the `explain()` method. This is useful for debugging performance issues.

```
plan = df.filter("amount > 1000").explain()
print(plan)
```

DML Operations

You can also perform Data Manipulation Language (DML) operations:

```
# Create Table As Select (CTAS)
df.filter("amount > 5000").create("finance.silver.large_transactions")

# Insert into existing table from query
df.filter("amount > 10000").insert("finance.silver.large_transactions")

# Insert from Pandas DataFrame or Arrow Table
```

```
import pandas as pd
data = pd.DataFrame({"transaction_id": [1001], "customer_id": [5], "amount": [15000.00]})
client.table("finance.silver.large_transactions").insert("finance.silver.large_transactions", data=data)

# Update rows
client.table("finance.bronze.transactions").filter("transaction_id = 1001").update({"amount": 16000.00})

# Delete rows
client.table("finance.bronze.transactions").filter("amount < 0").delete()

## Slowly Changing Dimensions (SCD2)

The `scd2` method automates the process of maintaining Type 2 Slowly Changing Dimensions. It closes old records (updates `valid_to`) and inserts new records (`valid_from`).

```python
builder.table("source_view").scd2(
    target_table="target_dim",
    on=["id"],
    track_cols=["name", "status"],
    valid_from_col="valid_from",
    valid_to_col="valid_to"
)
```

This executes two operations:

Close: Updates `valid_to` to `CURRENT_TIMESTAMP` for records in `target_dim` that have changed in `source_view`.

Insert: Inserts new versions of changed records and completely new records from `source_view` into `target_dim`.

Merge (Upsert)

You can perform `MERGE INTO` operations to upsert data.

```
# Upsert from a DataFrame
client.table("target").merge(
    target_table="target",
    on="id",
    matched_update={"val": "source.val"},
    not_matched_insert={"id": "source.id", "val": "source.val"},
    data=df_upsert
)
```

Batching
```

For `insert` and `merge` operations with in-memory data (Arrow Table or Pandas DataFrame), you can specify a `batch_size` to split the data into multiple chunks. This is useful for large datasets to avoid hitting query size limits.

```
# Insert in batches of 1000 rows
client.table("target").insert("target", data=large_df, batch_size=1000)
```

Batching

For `insert` and `merge` operations with in-memory data (Arrow Table or Pandas DataFrame), you can specify a `batch_size` to split the data into multiple chunks. This is useful for large datasets to avoid hitting query size limits.

```
# Insert in batches of 1000 rows
client.table("target").insert("target", data=large_df, batch_size=1000)
```

Schema Validation

You can validate data against a Pydantic schema before insertion.

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str

# Will raise ValidationError if data doesn't match
client.table("users").insert("users", data=df, schema=User)
```

Data Quality Checks

You can run data quality checks on a builder instance. These checks execute queries to verify assumptions about the data.

```
# Check that 'customer_id' is never NULL
df.quality.expect_not_null("customer_id")

# Check that 'transaction_id' is unique
df.quality.expect_unique("transaction_id")

# Check that 'status' is one of the allowed values
df.quality.expect_values_in("status", ["completed", "pending", "cancelled"])

# Custom Check: Row Count
# Check that there are exactly 0 rows where amount is negative
df.quality.expect_row_count("amount < 0", 0, "eq")

# Check that there are at least 100 rows total
df.quality.expect_row_count("1=1", 100, "ge")
```

```
---

# Source: caching.md

# Local Caching with DataFusion

DremioFrame allows you to cache query results locally as Arrow Feather files and query them using the DataFusion python library. This is useful for iterative analysis where you don't want to repeatedly hit the Dremio engine for the same data.

## Usage

Use the `cache()` method on a `DremioBuilder` object.

```python
# Cache the result of a query for 5 minutes (300 seconds)
# If the cache file exists and is younger than 5 minutes, it will be used.
# Otherwise, the query is executed on Dremio and the result is saved.
local_df = client.table("source.table") \
    .filter("col > 10") \
    .cache("my_cache", ttl_seconds=300)

# local_df is a LocalBuilder backed by DataFusion
# You can continue chaining methods
result = local_df.filter("col < 50") \
    .group_by("category") \
    .agg(avg_val="AVG(val)") \
    .collect()

print(result)
```

Features

- **TTL (Time-To-Live)**: Automatically invalidates cache if it's too old.
- **DataFusion Engine**: Executes SQL locally on the cached Arrow file, providing fast performance without network overhead.
- **Seamless Integration**: `LocalBuilder` mimics the `DremioBuilder` API for `select`, `filter`, `group_by`, `agg`, `order_by`, `limit`, and `sql`.

API

`cache(name, ttl_seconds=None, folder=".cache")`

- `name`: Name of the cache file (saved as `{folder}/{name}.feather`).
- `ttl_seconds`: Expiration time in seconds. If `None`, cache never expires (unless
```

manually deleted).

˜folder`: Directory to store cache files. Defaults to `.cache`.

Source: creating_tables.md

Creating Tables

DremioFrame provides multiple ways to create tables in Dremio, from simple schema definitions to automatic schema inference from DataFrames.

Quick Start

Create an Empty Table with Schema Dictionary

The simplest way to create a table is with a schema dictionary mapping column names to SQL types:

```
from dremioframe.client import DremioClient

client = DremioClient()

# Define schema
schema = {
    "id": "INTEGER",
    "name": "VARCHAR",
    "email": "VARCHAR",
    "created_at": "TIMESTAMP",
    "is_active": "BOOLEAN"
}

# Create the table
client.create_table("my_space.users", schema=schema)
```

Create Table from DataFrame

You can create a table directly from a pandas or polars DataFrame. The schema is automatically inferred:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "score": [95.5, 87.3, 92.1]
})

# Create table and insert data
```

```
client.create_table("my_space.scores", schema=df, insert_data=True)
```

Upload File as Table

You can upload local files (CSV, JSON, Parquet, Excel, etc.) directly as new tables:

```
# Upload a local CSV file
client.upload_file("data.csv", "my_space.my_data")

# Upload with explicit format
client.upload_file("data.xlsx", "my_space.excel_data", file_format="excel")
```

Methods

`client.create_table()`

The primary method for creating tables with full control over schema and data.

Signature:

```
client.create_table(
    table_name: str,
    schema: Union[Dict[str, str], DataFrame, Table],
    data: Any = None,
    insert_data: bool = True
)
```

Parameters:

˜table_name`: Full table name (e.g., `"space.folder.table"`)

˜schema`: Either:

D̄ictionary mapping column names to SQL types

p̄andas DataFrame (schema inferred)

p̄olars DataFrame (schema inferred)

p̄yarrow Table (schema inferred)

˜data`: Optional data to insert (only used with schema dict)

˜insert_data`: Whether to insert data when schema is a DataFrame/Table

`builder.create()`

Create a table using CTAS (Create Table As Select) from a query or data:

```
# CTAS from query
```

```
client.table("source_table") \\
    .filter("amount > 1000") \\
    .create("my_space.high_value_transactions")

# Create from DataFrame
import pandas as pd
df = pd.DataFrame({"id": [1, 2], "name": ["A", "B"]})
client.table("temp").create("my_space.new_table", data=df)
```

`builder.create_from_model()`

Create an empty table from a Pydantic model:

```
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name: str
    email: str
    active: bool

client.table("temp").create_from_model("my_space.users", User)
```

Supported Data Types

Dremio SQL Types

When using schema dictionaries, you can specify these SQL types:

SQL Type	Description	Example Values
`INTEGER`	32-bit integer	`-2147483648` to `2147483647`
`BIGINT`	64-bit integer	Very large integers
`FLOAT`	Single precision	`3.14`
`DOUBLE`	Double precision	`3.141592653589793`
`BOOLEAN`	True/False	`TRUE`, `FALSE`
`VARCHAR`	Variable-length string	`"Hello World"`
`DATE`	Date only	`2024-01-15`
`TIMESTAMP`	Date and time	`2024-01-15 10:30:00`
`TIME`	Time only	`10:30:00`
`DECIMAL`	Fixed precision	`123.45`
`VARBINARY`	Binary data	Byte arrays

Note: Dremio does not support `TINYINT` or `SMALLINT`. Use `INTEGER` instead.

Type Inference from DataFrames

When creating tables from DataFrames, types are automatically mapped:

Python/Arrow Type	Dremio SQL Type
`int8`, `int16`, `int32`	`INTEGER`
`int64`	`BIGINT`
`float32`	`FLOAT`
`float64`	`DOUBLE`
`bool`	`BOOLEAN`
`str`, `string`	`VARCHAR`
`datetime64`, `timestamp`	`TIMESTAMP`
`date32`, `date64`	`DATE`
`time32`, `time64`	`TIME`
`decimal128`, `decimal256`	`DECIMAL`

Usage Patterns

Pattern 1: Empty Table with Explicit Schema

Create an empty table to be populated later:

```
schema = {
    "transaction_id": "BIGINT",
    "customer_id": "INTEGER",
    "amount": "DOUBLE",
    "transaction_date": "TIMESTAMP",
    "status": "VARCHAR"
}

client.create_table("finance.bronze.transactions", schema=schema)
```

Pattern 2: Create and Populate from DataFrame

Create a table and immediately populate it with data:

```
import pandas as pd

df = pd.DataFrame({
    "product_id": [101, 102, 103],
    "product_name": ["Widget A", "Widget B", "Widget C"],
    "price": [19.99, 29.99, 39.99],
    "in_stock": [True, False, True]
})

# Create table with data
client.create_table(
    "catalog.products",
    schema=df,
    insert_data=True
)
```

Pattern 3: Create Empty Table from DataFrame Schema

Create an empty table using a DataFrame's schema without inserting data:

```
# Create a sample DataFrame to define schema
schema_df = pd.DataFrame({
    "id": pd.Series([], dtype='int64'),
    "name": pd.Series([], dtype='str'),
    "value": pd.Series([], dtype='float64')
})

# Create empty table
client.create_table(
    "my_space.my_table",
    schema=schema_df,
    insert_data=False
)
```

Pattern 4: Create with Schema Dict and Separate Data

Define schema explicitly, then insert data separately:

```
# Define schema
schema = {
    "id": "INTEGER",
    "name": "VARCHAR",
    "score": "DOUBLE"
}

# Prepare data
data = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "score": [95.5, 87.3, 92.1]
})

# Create table with schema and data
client.create_table(
    "my_space.scores",
    schema=schema,
    data=data,
    insert_data=True
)
```

Pattern 5: CTAS (Create Table As Select)

Create a table from a query result:

```
# Create table from filtered data
client.table("sales.transactions") \\
    .filter("amount > 1000") \\
```

```
    .filter("status = 'completed'") \\
    .create("sales.high_value_completed")

# Create aggregated table
client.table("sales.transactions") \\
    .group_by("customer_id") \\
    .agg(
        total_spent="SUM(amount)",
        transaction_count="COUNT(*)",
        avg_amount="AVG(amount)"
    ) \\
    .create("sales.customer_summary")
```

Pattern 6: Create from Polars DataFrame

Works seamlessly with polars DataFrames:

```
import polars as pl

df = pl.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"],
    "score": [95.5, 87.3, 92.1]
})

client.create_table("my_space.scores", schema=df, insert_data=True)
```

Pattern 7: Create from Arrow Table

Use pyarrow Tables directly:

```
import pyarrow as pa

# Define schema
schema = pa.schema([
    ('id', pa.int64()),
    ('name', pa.string()),
    ('value', pa.float64())
])

# Create table with data
table = pa.table({
    'id': [1, 2, 3],
    'name': ['A', 'B', 'C'],
    'value': [1.1, 2.2, 3.3]
}, schema=schema)

client.create_table("my_space.data", schema=table, insert_data=True)
```

Best Practices

1. Use Qualified Table Names

Always use fully qualified names (space.folder.table):

```
# Good
client.create_table("finance.bronze.transactions", schema=schema)

# Avoid (may cause ambiguity)
client.create_table("transactions", schema=schema)
```

2. Choose Appropriate Data Types

Use the most appropriate type for your data:

```
schema = {
    "id": "BIGINT",          # Use BIGINT for IDs that might grow large
    "amount": "DOUBLE",      # Use DOUBLE for currency/decimals
    "count": "INTEGER",      # Use INTEGER for counts
    "flag": "BOOLEAN",       # Use BOOLEAN for true/false
    "description": "VARCHAR" # Use VARCHAR for text
}
```

3. Validate Data Before Creating Tables

Ensure your DataFrame has the expected schema:

```
import pandas as pd

df = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["Alice", "Bob", "Charlie"]
})

# Check dtypes
print(df.dtypes)

# Create table
client.create_table("my_space.users", schema=df, insert_data=True)
```

4. Handle Large Datasets

For large datasets, create the table first, then insert in batches:

```
# Create empty table
schema = {"id": "INTEGER", "data": "VARCHAR"}
client.create_table("my_space.large_table", schema=schema)
```

```
# Insert in batches
for batch in large_data_batches:
    client.table("my_space.large_table").insert(
        "my_space.large_table",
        data=batch,
        batch_size=1000
    )
```

Comparison of Methods

Method	Use Case	Schema Source	Data Insertion
`create_table(schema=dict)`	Explicit schema definition	Manual	Optional
`create_table(schema=DataFrame)`	Infer from DataFrame	Automatic	Optional
`builder.create()`	CTAS from query	Query result	Automatic
`builder.create_from_model()`	Pydantic models	Pydantic	No

Troubleshooting

Table Already Exists

```
# Drop existing table first
client.execute("DROP TABLE IF EXISTS my_space.my_table")

# Then create
client.create_table("my_space.my_table", schema=schema)
```

Type Mismatch Errors

Ensure your data types match the schema:

```
# Convert types explicitly
df["id"] = df["id"].astype('int64')
df["score"] = df["score"].astype('float64')

client.create_table("my_space.data", schema=df, insert_data=True)
```

Permission Errors

Ensure you have CREATE TABLE privileges on the target space:

```
# Check your permissions or contact your Dremio administrator
```

See Also

Ingestion API - For loading data from various sources

Builder API - For query building and CTAS

Pydantic Integration - For schema validation

Data Types Reference - Complete list of Dremio types

Source: database_ingestion.md

Database Ingestion

DremioFrame provides a standardized way to ingest data from any SQL database (PostgreSQL, MySQL, SQLite, Oracle, etc.) into Dremio.

Installation

To use the database ingestion feature, you must install the optional dependencies:

```
pip install dremioframe[database]
```

This installs `connectorx` (for high-performance loading) and `sqlalchemy` (for broad compatibility).

Usage

The integration is exposed via `client.ingest.database()`.

Example: Loading from PostgreSQL

```
from dremioframe.client import DremioClient

client = DremioClient()

# Connection string (URI)
db_uri = "postgresql://user:password@localhost:5432/mydb"

# Ingest query results into Dremio
client.ingest.database(
    connection_string=db_uri,
    query="SELECT * FROM users WHERE active = true",
    table_name='"my_space"."my_folder"."users"',
    write_disposition="replace",
    backend="connectorx" # Default, faster
)
```

Parameters

- **connection_string**: Database connection URI (e.g., `postgresql://...`, `mysql://...`).

- **query**: SQL query to execute on the source database.

- **table_name**: The target table name in Dremio.

- **write_disposition**: `'replace'` or `'append'`.

- **backend**:

  - `'connectorx'` (default): Extremely fast, written in Rust. Supports Postgres, MySQL, SQLite, Redshift, Clickhouse, SQL Server.

  - `'sqlalchemy'`: Uses standard SQLAlchemy engines. Slower but supports any database with a Python driver.

- **batch_size**: (Only for `sqlalchemy` backend) Number of records to process per batch. Useful for large datasets to avoid memory issues.

Performance Tips

- Use `backend="connectorx"` whenever possible for significantly faster load times.

- For very large tables with `sqlalchemy`, set a `batch_size` (e.g., 50,000) to stream data instead of loading it all into memory.

Source: dlt_integration.md

dlt Integration

DremioFrame integrates with dlt (Data Load Tool) to allow you to easily ingest data from hundreds of sources (APIs, Databases, SaaS applications) directly into Dremio.

Installation

To use the `dlt` integration, you must install the optional dependencies:

```
pip install dremioframe[ingest]
```

Usage

The integration is exposed via `client.ingest.dlt()`. It accepts any `dlt` source or resource and loads it into a Dremio table.

Example: Loading from an API

```
import dlt
from dremioframe.client import DremioClient

# 1. Initialize Client
```

```
client = DremioClient()

# 2. Define a dlt resource (e.g., fetching from an API)
@dlt.resource(name="pokemon")
def get_pokemon():
    import requests
    url = "https://pokeapi.co/api/v2/pokemon?limit=10"
    response = requests.get(url).json()
    yield from response["results"]

# 3. Ingest into Dremio
# This will create (or replace) the table "space.folder.pokemon"
client.ingest.dlt(
    source=get_pokemon(),
    table_name='"my_space"."my_folder"."pokemon"',
    write_disposition="replace"
)
```

Parameters

- **source**: A `dlt` source or resource object.
- **table_name**: The target table name in Dremio (e.g., `"Space"."Folder"."Table"`).
- **write_disposition**:
  - `"replace"`: Drop table if exists and create new.
  - `"append"`: Append data to existing table.
- **batch_size**: Number of records to process per batch (default: 10,000).

Supported Sources

Since DremioFrame accepts standard `dlt` sources, you can use any of the verified sources from the dlt Hub, including:

- Salesforce
- HubSpot
- Google Sheets
- Notion
- Stripe
- And many more...

Source: export.md

Data Export

DremioFrame allows you to export query results to local files.

CSV Export

```
# Export to CSV
df.to_csv("output.csv", index=False)
```

Parquet Export

```
# Export to Parquet
df.to_parquet("output.parquet")
```

These methods internally collect the data to a Pandas DataFrame and call the respective Pandas export methods, so they support all standard Pandas arguments.

Source: export_formats.md

Export Formats

DremioFrame supports exporting query results to various file formats, including Parquet, CSV, JSON, and Delta Lake.

Supported Formats

Parquet

Export to Parquet format (using Pandas).

```
df.to_parquet("output.parquet", compression="snappy")
```

CSV

Export to CSV format.

```
df.to_csv("output.csv", index=False)
```

JSON

Export to JSON format.

```
df.to_json("output.json", orient="records")
```

Delta Lake

Export to Delta Lake format. This requires the optional `deltalake` dependency.

Installation:

```
pip install dremioframe[delta]
```

Usage:

```
# Create or overwrite a Delta table
df.to_delta("path/to/delta_table", mode="overwrite")

# Append to existing Delta table
df.to_delta("path/to/delta_table", mode="append")
```

Considerations

Memory Usage: All export methods currently collect the query result into memory (Pandas DataFrame) before writing. For very large datasets, consider using Dremio's `CTAS` (Create Table As Select) to write directly to data sources within Dremio, or use the `staging` method in `create/insert` for bulk loading.

Performance: Parquet is generally the most performant format for both reading and writing.

Source: file_system_ingestion.md

File System Ingestion

DremioFrame provides a convenient way to ingest multiple files from your local filesystem or network drives using glob patterns.

Installation

No additional dependencies required - this feature uses the core DremioFrame installation.

Usage

The integration is exposed via `client.ingest.files()`.

Example: Loading Multiple Parquet Files

```
from dremioframe.client import DremioClient

client = DremioClient()

```

```
# Ingest all parquet files in a directory
client.ingest.files(
    pattern="data/sales_*.parquet",
    table_name='"my_space"."my_folder"."sales"',
    write_disposition="replace"
)
```

Example: Recursive Directory Scan

```
# Ingest all CSV files in directory tree
client.ingest.files(
    pattern="data/**/*.csv",
    table_name='"my_space"."logs"."all_logs"',
    file_format="csv",
    recursive=True,
    write_disposition="append"
)
```

Parameters

pattern: Glob pattern (e.g., `"data/*.parquet"`, `"sales_2024_*.csv"`).

table_name: The target table name in Dremio.

file_format: File format (`'parquet'`, `'csv'`, `'json'`). Auto-detected from extension if not specified.

write_disposition:

`'replace'`: Drop table if exists and create new.

`'append'`: Append data to existing table.

recursive: If `True`, enables recursive glob (`` **pattern).**

Supported File Formats

Parquet (`.parquet`)

CSV (`.csv`)

JSON (`.json`, `.jsonl`, `.ndjson`)

How It Works

Finds all files matching the glob pattern

Reads each file into an Arrow Table

Concatenates all tables into a single table

- Uses the **staging method** (Parquet upload) for efficient bulk loading
- Creates or appends to the target table in Dremio

Performance

File system ingestion automatically uses the **staging method** for bulk loading, providing excellent performance even with large datasets:

- Handles 100+ files efficiently
- Supports files with millions of rows
- Minimal memory footprint (streaming read)

Use Cases

- **Data Lake Ingestion**: Load partitioned datasets from S3/HDFS mounted locally
- **Batch Processing**: Ingest daily/hourly file drops
- **Migration**: Bulk load historical data from file archives
- **Development**: Quick data loading from local test files

Source: file_upload.md

File Upload

DremioFrame allows you to upload local files directly to Dremio as Iceberg tables.

Supported Formats

- **CSV** (`.csv`)
- **JSON** (`.json`)
- **Parquet** (`.parquet`)
- **Excel** (`.xlsx`, `.xls`, `.ods`) - Requires `pandas`, `openpyxl`
- **HTML** (`.html`) - Requires `pandas`, `lxml`
- **Avro** (`.avro`) - Requires `fastavro`
- **ORC** (`.orc`) - Requires `pyarrow`
- **Lance** (`.lance`) - Requires `pylance`
- **Feather/Arrow** (`.feather`, `.arrow`) - Requires `pyarrow`

Usage

Use the `client.upload_file()` method.

```
from dremioframe.client import DremioClient

client = DremioClient()

# Upload a CSV file
client.upload_file("data/sales.csv", "space.folder.sales_table")

# Upload an Excel file
client.upload_file("data/financials.xlsx", "space.folder.financials")

# Upload an Avro file
client.upload_file("data/users.avro", "space.folder.users")
```

Arguments

`file_path` (str): Path to the local file.

`table_name` (str): Destination table name in Dremio (e.g., "space.folder.table").

`file_format` (str, optional): The format of the file ('csv', 'json', 'parquet', 'excel', 'html', 'avro', 'orc', 'lance', 'feather'). If not provided, it is inferred from the file extension.

`kwargs`: Additional arguments passed to the underlying file reader.

CSV: `pyarrow.csv.read_csv`

JSON: `pyarrow.json.read_json`

Parquet: `pyarrow.parquet.read_table`

Excel: `pandas.read_excel`

HTML: `pandas.read_html`

Avro: `fastavro.reader`

ORC: `pyarrow.orc.read_table`

Lance: `lance.dataset`

Feather: `pyarrow.feather.read_table`

Example with Options

```
# Upload Excel sheet "Sheet2"
client.upload_file("data.xlsx", "space.folder.data", sheet_name="Sheet2")
```

Source: files.md

Working with Files

Dremio allows you to query unstructured data (files) directly using the `LIST_FILES` table function. DremioFrame provides a helper method for this.

Usage

```
# List files in a folder
df = client.list_files("@my_user/documents")
df.select("file_path", "file_size").show()
```

This generates SQL:

```
SELECT file_path, file_size FROM TABLE(LIST_FILES('@my_user/documents'))
```

Integrating with AI Functions

`LIST_FILES` is powerful when combined with AI functions for processing unstructured text (RAG - Retrieval Augmented Generation).

```
from dremioframe import F

# 1. List PDF files
files = client.list_files("@my_source/contracts") \
    .filter("file_name LIKE '%.pdf'")

# 2. Use AI_GENERATE to extract data from files
# Note: We pass the 'file_content' column (implied reference) to the AI function
# The actual column name from LIST_FILES usually includes a 'file' struct or similar depending on Dremio version/source.
# Assuming 'file_path' is available or we pass the file reference.

# Example: Extracting entities from files
df = files.select(
    F.col("file_path"),
    F.ai_generate(
        "Extract parties and dates",
        F.col("file_content") # Hypothetical column, check your source schema
    ).alias("extracted_info")
)
```

Source: guide_iceberg_management.md

Guide: Iceberg Lakehouse Management

DremioFrame provides powerful tools to manage your Iceberg tables directly from Python. This guide covers maintenance tasks, snapshot management, and time travel.

Table Maintenance

Regular maintenance is crucial for Iceberg table performance.

Optimization (Compaction)

Compacts small files into larger ones to improve read performance.

```
from dremioframe.client import DremioClient

client = DremioClient(...)

# Optimize a specific table
client.table("warehouse.sales").optimize()

# Optimize with specific file size target (if supported by Dremio version/source)
# client.table("warehouse.sales").optimize(target_size_mb=128)
```

Vacuum (Expire Snapshots & Remove Orphan Files)

Removes old snapshots and unused data files to reclaim space.

```
# Expire snapshots older than 7 days
client.table("warehouse.sales").vacuum(retain_days=7)
```

Snapshot Management

Viewing Snapshots

Inspect the history of your table.

```
history = client.table("warehouse.sales").history()
print(history)
# Returns a DataFrame with snapshot_id, committed_at, etc.
```

Time Travel

Query the table as it existed at a specific point in time.

```
# Query by Snapshot ID
df_snapshot = client.table("warehouse.sales").at(snapshot_id=123456789).collect()

# Query by Timestamp
df_time = client.table("warehouse.sales").at(timestamp="2023-01-01 12:00:00").collect()
```

Rollback

Revert the table state to a previous snapshot.

```
# Rollback to a specific snapshot
client.table("warehouse.sales").rollback(snapshot_id=123456789)
```

Orchestrating Maintenance

You can automate these tasks using DremioFrame's Orchestration features.

```
from dremioframe.orchestration import Pipeline, OptimizeTask, VacuumTask

pipeline = Pipeline("weekly_maintenance")

optimize = OptimizeTask(
    name="optimize_sales",
    client=client,
    table="warehouse.sales"
)

vacuum = VacuumTask(
    name="vacuum_sales",
    client=client,
    table="warehouse.sales",
    retain_days=7
)

pipeline.add_task(optimize)
pipeline.add_task(vacuum)

# Ensure vacuum runs after optimize
vacuum.set_upstream(optimize)

pipeline.run()
```

Source: iceberg.md

Iceberg Client

Interact directly with Dremio's Iceberg Catalog using `pyiceberg`.

Iceberg Client

Interact directly with Dremio's Iceberg Catalog using `pyiceberg`.

Configuration

The Iceberg client requires specific configuration depending on whether you are using Dremio Cloud or Dremio Software.

Environment Variables

Variable	Description	Required	Default
`DREMIO_PAT`	Personal Access Token	Yes	None
`DREMIO_PROJECT_ID`	Project Name (Cloud) or Warehouse Name	Yes (Cloud)	None
`DREMIO_ICEBERG_URI`	Iceberg Catalog REST URI	No (Cloud), Yes (Software)	`https://catalog.dremio.cloud/api/iceberg`

Dremio Cloud

For Dremio Cloud, you typically only need `DREMIO_PAT` and `DREMIO_PROJECT_ID`.

```
export DREMIO_PAT="your_pat"
export DREMIO_PROJECT_ID="your_project_id"
```

Dremio Software

For Dremio Software, you must specify the `DREMIO_ICEBERG_URI`.

```
export DREMIO_PAT="your_pat"
export DREMIO_ICEBERG_URI="http://dremio-host:9047/api/iceberg"
# DREMIO_PROJECT_ID can be any string for Software, but is required by PyIceberg
export DREMIO_PROJECT_ID="my_warehouse"
```

Usage

Access the Client

```
iceberg = client.iceberg
```

List Namespaces

```
namespaces = iceberg.list_namespaces()
print(namespaces)
```

List Tables

```
tables = iceberg.list_tables("my_namespace")
print(tables)
```

Load Table

```
table = iceberg.load_table("my_namespace.my_table")
print(table.schema())
```

Append Data

Append a Pandas DataFrame to an Iceberg table.

```
import pandas as pd
df = pd.DataFrame({"id": [1, 2], "name": ["Alice", "Bob"]})

iceberg.append("my_namespace.my_table", df)
```

Create Table

```
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, IntegerType, StringType

schema = Schema(
    NestedField(1, "id", IntegerType(), required=True),
    NestedField(2, "name", StringType(), required=False),
)

table = iceberg.create_table("my_namespace.new_table", schema)
```

Source: incremental_processing.md

Incremental Processing

DremioFrame simplifies incremental data loading patterns, allowing you to efficiently process only new or changed data.

IncrementalLoader

The `IncrementalLoader` class provides helper methods for watermark-based loading and MERGE (upsert) operations.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.incremental import IncrementalLoader

client = DremioClient()
loader = IncrementalLoader(client)
```

Watermark-based Loading

This pattern loads data from a source table to a target table where a specific column (e.g., timestamp or ID) is greater than the maximum value in the target table.

```
# Load new data from 'staging.events' to 'analytics.events'
# based on the 'event_time' column.
rows_inserted = loader.load_incremental(
    source_table="staging.events",
    target_table="analytics.events",
    watermark_col="event_time"
)

print(f"Loaded {rows_inserted} new rows.")
```

How it works:

- Queries `MAX(event_time)` from `analytics.events`.
- Executes `INSERT INTO analytics.events SELECT * FROM staging.events WHERE event_time > 'MAX_VALUE'`.
- If the target table is empty, it performs a full load.

Merge (Upsert)

The `merge` method performs a standard SQL MERGE operation to update existing records and insert new ones.

```
# Upsert users from staging to production
loader.merge(
    source_table="staging.users",
    target_table="production.users",
    on=["user_id"],                 # Join condition
    update_cols=["email", "status"], # Columns to update when matched
     insert_cols=["user_id", "email", "status", "created_at"] # Columns to insert when
not matched
)
```

Generated SQL:

```
MERGE INTO production.users AS target
USING staging.users AS source
ON (target.user_id = source.user_id)
WHEN MATCHED THEN
    UPDATE SET email = source.email, status = source.status
WHEN NOT MATCHED THEN
    INSERT (user_id, email, status, created_at)
    VALUES (source.user_id, source.email, source.status, source.created_at)
```

Best Practices

Indexing: Ensure your watermark columns and join keys are optimized (e.g., sorted or partitioned) in Dremio for performance.

Reflections: Use Dremio Reflections to accelerate the `MAX(watermark)` query on large target tables.

Data Types: Ensure data types match between source and target to avoid casting issues during INSERT/MERGE.

Source: ingestion.md

Ingestion Overview

DremioFrame provides multiple ways to ingest data into Dremio, ranging from simple API calls to complex file system and database integrations.

1. API Ingestion

Ingest data directly from REST APIs. This method is built into the main client.

Method: `client.ingest_api(...)`

```
client.ingest_api(
    url="https://api.example.com/users",
    table_name="raw_users",
    mode="replace" # 'replace', 'append', or 'merge'
)
```

Read more about API Ingestion strategies (Modes, Auth, Batching)

2. File Upload

Upload local files (CSV, JSON, Parquet, Excel, etc.) directly to Dremio as tables.

Method: `client.upload_file(...)`

```
client.upload_file("data/sales.csv", "space.folder.sales_table")
```

Read the full File Upload guide

3. Ingestion Modules

Advanced ingestion capabilities are grouped under the `client.ingest` namespace.

DLT (Data Load Tool)

Integration with the `dlt` library for robust pipelines.

Method: `client.ingest.dlt(...)`

```
data = [{"id": 1, "name": "Alice"}]
client.ingest.dlt(data, "my_dlt_table")
```

Read the DLT Integration guide

Database Ingestion

Ingest query results from other databases (Postgres, MySQL, etc.) using JDBC/ODBC connectors via `connectorx` or `sqlalchemy`.

Method: `client.ingest.database(...)`

```
client.ingest.database(
    connection_string="postgresql://user:pass@localhost/db",
    query="SELECT * FROM users",
    table_name="postgres_users"
)
```

Read the Database Ingestion guide

File System Ingestion

Ingest multiple files from a local directory or glob pattern.

Method: `client.ingest.files(...)`

```
client.ingest.files("data/*.parquet", "my_dataset")
```

Read the File System Ingestion guide

Source: ingestion_patterns.md

Ingestion Patterns & Best Practices

This guide outlines common patterns for moving data into Iceberg tables using DremioFrame, covering both Dremio-connected sources and external data.

Why Move Data to Iceberg?

While Dremio can query data directly from sources like Postgres, SQL Server, or S3, moving data into **Apache Iceberg** tables (in Dremio's Arctic or S3/Data Lake sources) offers significant benefits:

- **Performance**: Iceberg tables are optimized for analytics (columnar, partitioned).

Features: Enables Time Travel, Rollback, and DML operations (Update/Delete/Merge).

Isolation: Decouples analytical workloads from operational databases.

Pattern 1: Source to Iceberg (ELT)

If your data is already in a source connected to Dremio (e.g., a Postgres database or a raw S3 folder), you can use Dremio to move it into an Iceberg table.

Initial Load (CTAS)

Use the `create` method to perform a `CREATE TABLE AS SELECT` (CTAS) operation. This pushes the work to the Dremio engine.

```
# Create an Iceberg table 'marketing.users' from a Postgres source table
client.table("postgres.public.users") \
    .filter("active = true") \
    .create("marketing.users")
```

Incremental Append

Use `insert` to append new rows from a source to an existing Iceberg table.

```
# Append new logs from S3 to Iceberg
client.table("s3.raw_logs") \
    .filter("event_date = CURRENT_DATE") \
    .insert("marketing.logs")
```

Upsert (Merge)

Use `merge` to update existing records and insert new ones.

```
# Upsert users from Postgres to Iceberg
client.table("postgres.public.users").merge(
    target_table="marketing.users",
    on="id",
    matched_update={"email": "source.email", "status": "source.status"},
          not_matched_insert={"id":  "source.id",  "email":  "source.email",  "status":
"source.status"}
)
```

Pattern 2: External Data to Iceberg (ETL)

If your data originates outside Dremio (e.g., REST APIs, local files, Python scripts), you can ingest it using DremioFrame.

API Ingestion

Use the `ingest_api` utility for REST APIs.

```
# Fetch users from an API and merge them into an Iceberg table
client.ingest_api(
    url="https://api.example.com/users",
    table_name="marketing.users",
    mode="merge",
    pk="id"
)
```

Local Dataframes (Pandas/Arrow)

If you have data in a Pandas DataFrame or PyArrow Table, the recommended approach for creating new tables is `client.create_table`.

```
import pandas as pd

# Load local CSV
df = pd.read_csv("local_data.csv")

# Option 1: Using create_table (Recommended for new tables)
# This is the cleanest API for creating tables from local data
client.create_table("marketing.local_data", schema=df, insert_data=True)

# Option 2: Using builder.create (CTAS approach)
# Note: The source table in client.table() is ignored when 'data' is provided.
# This pattern is useful if you are already working with a builder object.
client.table("marketing.local_data").create("marketing.local_data", data=df)

# Option 3: Appending to existing table
# Use this to add data to an existing table
client.table("marketing.local_data").insert("marketing.local_data", data=df)
```

Note: For large local datasets, use the `batch_size` parameter to avoid memory issues and timeouts.

```
client.table("target").insert("target", data=large_df, batch_size=5000)
```

See Creating Tables for more details on table creation methods.

Best Practices

1. Optimize Your Tables

After significant data ingestion (especially many small inserts), run `optimize()` to compact small files.

```
client.table("marketing.users").optimize()
```

2. Manage Snapshots

Iceberg keeps history for Time Travel. To save storage, periodically expire old snapshots using `vacuum()`.

```
# Retain only the last 5 snapshots
client.table("marketing.users").vacuum(retain_last=5)
```

3. Use Staging Tables for Complex Merges

If you need to perform complex transformations before merging, load data into a temporary staging table first.

```
# 1. Load raw data to staging
client.ingest_api(..., table_name="staging_users", mode="replace")

# 2. Transform and Merge from staging to target
client.table("staging_users") \
    .mutate(full_name="concat(first, ' ', last)") \
    .merge(target_table="marketing.users", on="id", ...)

# 3. Drop staging
client.table("staging_users").delete() # Or drop via SQL
```

4. Batching

When inserting data from Python (Pandas/Arrow), always use `batch_size` for datasets larger than a few thousand rows.

5. Type Consistency

Ensure your local DataFrame types match Dremio's expected types. DremioFrame handles basic conversion, but explicit casting in Pandas (e.g., `pd.to_datetime`) is recommended before ingestion.

Source: joins.md

Joins

DremioFrame allows you to join tables or builder instances.

Join Syntax

```
left_df.join(other, on, how='inner')
```

other: Can be a table name (string) or another `DremioBuilder` object.

on: The join condition (SQL string).

how: Join type (`inner`, `left`, `right`, `full`, `cross`).

Examples

Join with a Table Name

```
df = client.table("orders")
joined = df.join("customers", on="left_tbl.customer_id = right_tbl.id", how="left")
joined.show()
```

Join with Another Builder

This allows you to filter or transform the right-side table before joining.

```
orders = client.table("orders")
customers = client.table("customers").filter("active = true")

joined = orders.join(customers, on="left_tbl.customer_id = right_tbl.id")
joined.show()
```

Note on Aliases

When joining, DremioFrame automatically wraps the left and right sides in subqueries aliased as `left_tbl` and `right_tbl` respectively. You should use these aliases in your `on` condition.

Source: pydantic_integration.md

Pydantic Integration

DremioFrame integrates with Pydantic to allow for schema validation and table creation based on data models.

Creating Tables from Models

You can generate a `CREATE TABLE` statement directly from a Pydantic model.

```
from pydantic import BaseModel
from dremioframe.client import DremioClient
```

```
class User(BaseModel):
    id: int
    name: str
    active: bool

client = DremioClient(...)
builder = client.builder

# Create table 'users' with columns id (INTEGER), name (VARCHAR), active (BOOLEAN)
builder.create_from_model("users", User)
```

Validating Data

You can validate existing data in Dremio against a Pydantic schema. This fetches a sample of data and checks if it conforms to the model.

```
# Validate the first 1000 rows of 'users' table
builder.table("users").validate(User, sample_size=1000)
```

Inserting Data with Validation

When inserting data using `create` or `insert`, you can pass a `schema` argument to validate the data before insertion.

```
data = [{"id": 1, "name": "Alice", "active": True}]
builder.insert("users", data, schema=User)
```

Source: query_templates.md

Query Templates

DremioFrame provides a simple template system for managing parameterized SQL queries, helping you build a library of reusable query patterns.

TemplateLibrary

The `TemplateLibrary` allows you to register and render SQL templates using standard Python string substitution syntax (`$variable`).

Usage

```
from dremioframe.templates import library, TemplateLibrary

# Use the global library or create your own
my_lib = TemplateLibrary()
```

```
# Register a template
my_lib.register(
    name="user_activity",
    sql="SELECT * FROM logs WHERE user_id = $uid AND date >= '$start_date'",
    description="Get activity logs for a specific user"
)

# Render the query
sql = my_lib.render("user_activity", uid=123, start_date="2023-01-01")
print(sql)
# Output: SELECT * FROM logs WHERE user_id = 123 AND date >= '2023-01-01'
```

Built-in Templates

DremioFrame comes with a few common templates pre-registered in the global `library`:

- **`row_count`**: `SELECT COUNT(*) as count FROM $table`
- **`sample`**: `SELECT * FROM $table LIMIT $limit`
- **`distinct_values`**: `SELECT DISTINCT $column FROM $table ORDER BY $column`

```
# Example using built-in template
sql = library.render("row_count", table="my_table")
```

Best Practices

- **Security**: This is a simple string substitution mechanism. **It does not prevent SQL injection** if you pass untrusted user input directly. Always validate inputs or use it for internal query generation where inputs are controlled.
- **Organization**: Group related templates into separate libraries or modules for better organization in large projects.

Source: querying.md

Raw SQL Querying

While DremioFrame provides a powerful Builder API, sometimes you just want to run raw SQL.

Usage

Use the `query()` method on the `DremioClient`.

```
# Return as Pandas DataFrame (default)
df = client.query('SELECT * FROM finance.bronze.transactions LIMIT 10')
```

```
# Return as Arrow Table
arrow_table = client.query('SELECT * FROM finance.bronze.transactions LIMIT 10', format="arrow")

# Return as Polars DataFrame
polars_df = client.query('SELECT * FROM finance.bronze.transactions LIMIT 10', format="polars")
```

DDL/DML

For operations that don't return rows (like `CREATE VIEW`, `DROP TABLE`), use `execute()`:

```
client.execute("DROP TABLE IF EXISTS my_table")
```

Source: schema_evolution.md

Schema Evolution

DremioFrame provides tools to manage schema evolution for your Dremio tables, allowing you to detect changes and generate migration scripts.

SchemaManager

The `SchemaManager` class helps you compare the current schema of a table in Dremio against a desired schema (e.g., from your code or a config file) and synchronize them.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.schema_evolution import SchemaManager

client = DremioClient()
manager = SchemaManager(client)
```

Comparing Schemas

You can compare the current table schema with a new schema definition.

```
# Define desired schema
new_schema = {
    "id": "INT",
    "name": "VARCHAR",
    "email": "VARCHAR",
```

```
    "created_at": "TIMESTAMP"
}

# Get current schema
current_schema = manager.get_table_schema("space.folder.users")

# Compare
diff = manager.compare_schemas(current_schema, new_schema)

print("Added:", diff['added_columns'])
print("Removed:", diff['removed_columns'])
print("Changed:", diff['changed_columns'])
```

Generating Migration Scripts

Generate SQL statements to migrate the table.

```
sqls = manager.generate_migration_sql("space.folder.users", diff)

for sql in sqls:
    print(sql)
# Output:
# ALTER TABLE space.folder.users ADD COLUMN email VARCHAR
```

Syncing Table

Automatically apply changes (or dry run).

```
# Dry run (default) - returns SQL statements
sqls = manager.sync_table("space.folder.users", new_schema, dry_run=True)

# Execute changes
manager.sync_table("space.folder.users", new_schema, dry_run=False)
```

Limitations

Type Changes: Changing column types is complex and may not be supported directly by Dremio for all table formats. The tool generates a warning comment for type changes.

Data Migration: This tool handles schema changes (DDL), not data transformation.

Iceberg Support: Works best with Iceberg tables which support full schema evolution.

Source: sorting.md

Sorting and Distinct

Order By

Use `order_by` to sort the results.

```
# Sort by population descending
df.order_by("pop", ascending=False).show()

# Sort by state ascending, then city descending
df.order_by("state").order_by("city", ascending=False).show()
```

Distinct

Use `distinct` to remove duplicate rows.

```
# Get unique states
df.select("state").distinct().show()
```

Source: sql_linting.md

SQL Linting

DremioFrame provides a `SqlLinter` to validate SQL queries against Dremio and perform static code analysis to catch common issues.

SqlLinter

The `SqlLinter` can validate queries by requesting an execution plan from Dremio (ensuring syntax and table references are correct) and by checking for patterns that might lead to poor performance or unexpected results.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.linter import SqlLinter

client = DremioClient()
linter = SqlLinter(client)
```

Validating SQL

Validation runs `EXPLAIN PLAN FOR <query>` against Dremio. This confirms that the SQL syntax is valid and that all referenced tables and columns exist and are accessible.

```
sql = "SELECT count(*) FROM space.folder.table"
result = linter.validate_sql(sql)
```

```
if result["valid"]:
    print("SQL is valid!")
else:
    print(f"Validation failed: {result['error']}")
```

Static Linting

Static linting checks the SQL string for common anti-patterns without connecting to Dremio.

```
sql = "SELECT * FROM huge_table"
warnings = linter.lint_sql(sql)

for warning in warnings:
    print(f"Warning: {warning}")
# Output: Warning: Avoid 'SELECT *' in production queries. Specify columns explicitly.
```

Rules Checked

`SELECT *: Discourages selecting all columns in production.**

Unbounded DELETE/UPDATE: Warns if `DELETE` or `UPDATE` statements are missing a `WHERE` clause.

Source: framework.md

Data Quality Framework

DremioFrame includes a file-based Data Quality (DQ) framework to validate your data in Dremio.

Requirements

```
pip install "dremioframe[dq]"
```

Defining Tests

Tests are defined in YAML files. You can place them in any directory.

Example: `tests/dq/sales_checks.yaml`

```
tests:
  - name: Validate Sales Table
    table: "Space.Folder.Sales"
    checks:
      - type: not_null
```

```
        column: order_id

      - type: unique
        column: order_id

      - type: values_in
        column: status
        values: ["PENDING", "SHIPPED", "DELIVERED", "CANCELLED"]

      - type: row_count
        condition: "amount < 0"
        threshold: 0
        operator: eq  # Expect 0 rows where amount < 0

      - type: custom_sql
        condition: "discount > amount"
        error_msg: "Discount cannot be greater than amount"
```

Running Tests

Use the CLI to run tests in a directory.

```
dremio-cli dq run tests/dq
```

Check Types

Type	Description	Parameters
`not_null`	Ensures a column has no NULL values.	`column`
`unique`	Ensures a column has unique values.	`column`
`values_in`	Ensures column values are within a list.	`column`, `values`
`row_count`	Checks row count matching a condition.	`condition`, `threshold`, `operator` (eq, ne, gt, lt, ge, le)
`custom_sql`	Fails if any row matches the condition.	`condition`, `error_msg`

Source: recipes.md

Data Quality Recipes

This guide provides common patterns and recipes for validating your data using DremioFrame's Data Quality framework.

Recipe 1: Validating Reference Data

Ensure that reference tables (like country codes or status lookups) contain expected values and no duplicates.

```
- name: "Validate Country Codes"
```

```
  table: "reference.countries"
  checks:
    - type: "unique"
      column: "iso_code"

    - type: "not_null"
      column: "country_name"

    - type: "row_count"
      threshold: 190
      operator: "gt"
```

Recipe 2: Financial Integrity Checks

Validate financial transactions for non-negative amounts and referential integrity (via custom SQL).

```
- name: "Transaction Integrity"
  table: "finance.transactions"
  checks:
    - type: "custom_sql"
      condition: "MIN(amount) >= 0"
      error_msg: "Found negative transaction amounts"

    - type: "not_null"
      column: "transaction_date"

      # Check that all transactions belong to valid accounts (simplified referential
integrity)
    - type: "custom_sql"
              condition:  "(SELECT  COUNT(*)  FROM  finance.transactions  t  LEFT  JOIN
finance.accounts a ON t.account_id = a.id WHERE a.id IS NULL) = 0"
      error_msg: "Found transactions for non-existent accounts"
```

Recipe 3: Daily Ingestion Validation

Verify that a daily ingestion job loaded data correctly by checking row counts and freshness.

```
- name: "Daily Web Logs"
  table: "raw.web_logs"
  checks:
    # Ensure we loaded at least some rows
    - type: "row_count"
      threshold: 0
      operator: "gt"

    # Ensure data is from today (assuming 'event_timestamp' column)
    - type: "custom_sql"
      condition: "MAX(event_timestamp) >= CURRENT_DATE"
```

```
      error_msg: "No data loaded for today"
```

Recipe 4: Categorical Data Validation

Ensure columns with categorical data only contain allowed values.

```
- name: "User Status Validation"
  table: "users.profiles"
  checks:
    - type: "values_in"
      column: "subscription_tier"
      values: ["free", "basic", "premium", "enterprise"]

    - type: "values_in"
      column: "email_verified"
      values: [true, false]
```

Recipe 5: Running Tests Programmatically

You can run these YAML recipes from your Python code using the `DQRunner`.

```
from dremioframe.client import DremioClient
from dremioframe.dq.runner import DQRunner

client = DremioClient()
runner = DQRunner(client)

# Load tests from a directory containing your YAML files
tests = runner.load_tests("./dq_checks")

# Run all loaded tests
success = runner.run_tests(tests)

if not success:
    print("Data Quality checks failed!")
    exit(1)
```

Source: yaml_syntax.md

Data Quality YAML Syntax

DremioFrame's Data Quality framework allows you to define tests in YAML files. This enables a declarative approach to data quality, where checks are version-controlled and separated from your application code.

File Structure

A Data Quality YAML file can contain a list of test definitions. Each test targets a specific table and contains a list of checks to perform.

Root Element

The root of the YAML file can be:

Ā **list** of test objects.

Ā **dictionary** with a `tests` key containing a list of test objects.

Test Object

Field	Type	Required	Description
`name`	string	No	A descriptive name for the test suite. Defaults to "Unnamed Test".
`table`	string	**Yes**	The full path to the Dremio table or view being tested (e.g., `source.folder.table`).
`checks`	list	**Yes**	A list of check objects to execute against the table.

Check Object

Every check object requires a `type` field. Other fields depend on the check type.

`not_null`

Ensures a column contains no NULL values.

Field	Type	Description
`type`	string	Must be `not_null`.
`column`	string	The name of the column to check.

`unique`

Ensures all values in a column are unique.

Field	Type	Description
`type`	string	Must be `unique`.
`column`	string	The name of the column to check.

`values_in`

Ensures column values are within a specified allowed list.

Field	Type	Description
`type`	string	Must be `values_in`.

| `column` | string | The name of the column to check. |
| `values` | list | A list of allowed values (strings, numbers, etc.). |

`row_count`

Validates the total number of rows in the table based on a condition.

Field	Type	Description
`type`	string	Must be `row_count`.
`condition`	string	Optional SQL WHERE clause to filter rows before counting. Default: `1=1` (all rows).
`threshold`	number	The value to compare the count against.
`operator`	string	Comparison operator: `eq` (=), `gt` (>), `lt` (<), `gte` (>=), `lte` (<=). Default: `gt`.

`custom_sql`

Runs a custom SQL condition that must return TRUE for the check to pass.

Field	Type	Description
`type`	string	Must be `custom_sql`.
`condition`	string	A SQL boolean expression (e.g., `SUM(amount) > 0`).
`error_msg`	string	Optional error message to display if the check fails.

Example

```
tests:
  - name: "Customer Table Checks"
    table: "marketing.customers"
    checks:
      - type: "not_null"
        column: "customer_id"

      - type: "unique"
        column: "email"

      - type: "values_in"
        column: "status"
        values: ["active", "inactive", "pending"]

  - name: "Sales Data Validation"
    table: "sales.transactions"
    checks:
      - type: "row_count"
        threshold: 1000
        operator: "gt"

      - type: "custom_sql"
        condition: "SUM(total_amount) > 0"
        error_msg: "Total sales amount must be positive"
```

Source: cicd.md

CI/CD & Deployment

Deploying `dremioframe` pipelines and managing Dremio resources (Views, Reflections, RBAC) should be automated using CI/CD.

1. Managing Resources as Code

Store your Dremio logic in a version-controlled repository.

Views: Define views as SQL files or Python scripts using `create_view`.

Reflections: Define reflection configurations in JSON/YAML or Python scripts.

Example: Deploy Script (`deploy.py`)

```
import os
from dremioframe.client import DremioClient

client = DremioClient()

def deploy_view(view_name, sql_file):
    with open(sql_file, "r") as f:
        sql = f.read()
    client.catalog.create_view(["Space", view_name], sql, overwrite=True)
    print(f"Deployed {view_name}")

if __name__ == "__main__":
    deploy_view("sales_summary", "views/sales_summary.sql")
```

2. GitHub Actions Workflow

Here is an example workflow to deploy changes when merging to `main`.

```
name: Deploy to Dremio

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      DREMIO_PAT: ${{ secrets.DREMIO_PAT }}
      DREMIO_PROJECT_ID: ${{ secrets.DREMIO_PROJECT_ID }}

    steps:
      - uses: actions/checkout@v3
```

```
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: pip install dremioframe

      - name: Run Deploy Script
        run: python deploy.py
```

3. Environment Management

Use different Spaces or prefixes for environments (Dev, Staging, Prod).

```
# deploy.py
env = os.environ.get("ENV", "dev")
space = f"Marketing_{env}"

client.catalog.create_view([space, "view_name"], ...)
```

4. Testing

Run Data Quality checks as part of your CI pipeline before deploying.

```
# In CI step
dremio-cli dq run tests/dq
```

Source: configuration.md

Configuration Reference

This guide details all configuration options for DremioFrame, including environment variables and client arguments.

Quick Start Examples

Dremio Cloud

```
from dremioframe.client import DremioClient

# Simplest - uses environment variables DREMIO_PAT and DREMIO_PROJECT_ID
client = DremioClient()

# Or specify explicitly
```

```
client = DremioClient(
    pat="your_pat_here",
    project_id="your_project_id",
    mode="cloud"  # Optional, auto-detected
)
```

Dremio Software v26+

```
# With PAT (recommended for v26+)
client = DremioClient(
    hostname="v26.dremio.org",
    pat="your_pat_here",
    tls=True,
    mode="v26"  # Automatically sets correct ports
)

# With username/password
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password123",
    tls=False,
    mode="v26"
)
```

Dremio Software v25

```
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password123",
    mode="v25"  # Uses v25-specific endpoints
)
```

Environment Variables

Dremio Cloud Connection

Variable	Description	Required
`DREMIO_PAT`	Personal Access Token for authentication.	Yes (for Cloud)
`DREMIO_PROJECT_ID`	Project ID of the Dremio Cloud project.	Yes (for Cloud)

Dremio Software Connection

| Variable | Description | Default |

|----------|-------------|---------|
`DREMIO_SOFTWARE_HOST`	Hostname or URL of the Dremio coordinator.	`localhost`
`DREMIO_SOFTWARE_PAT`	Personal Access Token (v26+ only).	-
`DREMIO_SOFTWARE_USER`	Username (optional for v26+ with PAT, required for v25).	-
`DREMIO_SOFTWARE_PORT`	REST API port (optional, auto-detected).	`443` (TLS) or `9047`
`DREMIO_SOFTWARE_FLIGHT_PORT`	Arrow Flight port (optional, auto-detected).	`32010`
`DREMIO_SOFTWARE_PASSWORD`	Password for authentication.	-
`DREMIO_SOFTWARE_TLS`	Enable TLS (`true`/`false`).	`false`
`DREMIO_ICEBERG_URI`	Iceberg Catalog REST URI (Required for Software).	`https://catalog.dremio.cloud/api/iceberg` (Cloud default)

Orchestration Backend

Variable	Description	Example
`DREMIOFRAME_PG_DSN`	PostgreSQL connection string.	`postgresql://user:pass@host/db`
`DREMIOFRAME_MYSQL_USER`	MySQL username.	`root`
`DREMIOFRAME_MYSQL_PASSWORD`	MySQL password.	`password`
`DREMIOFRAME_MYSQL_HOST`	MySQL host.	`localhost`
`DREMIOFRAME_MYSQL_DB`	MySQL database name.	`dremioframe`
`DREMIOFRAME_MYSQL_PORT`	MySQL port.	`3306`

Celery Executor

Variable	Description	Default
`CELERY_BROKER_URL`	Broker URL for Celery (Redis/RabbitMQ).	`redis://localhost:6379/0`
`CELERY_RESULT_BACKEND`	Backend for Celery results.	`redis://localhost:6379/0`

AWS / S3 Task

Variable	Description
`AWS_ACCESS_KEY_ID`	AWS Access Key.
`AWS_SECRET_ACCESS_KEY`	AWS Secret Key.
`AWS_DEFAULT_REGION`	AWS Region (e.g., `us-east-1`).

DremioClient Arguments

When initializing `DremioClient`, you can pass arguments directly or rely on environment variables.

```
class DremioClient:
    def __init__(
        self,
```

```
        # Authentication
         pat: str = None,                      # Personal Access Token (Cloud or Software
v26+)
        username: str = None,                # Username (Software with user/pass auth)
        password: str = None,                # Password (Software with user/pass auth)
        project_id: str = None,              # Project ID (Cloud only)

        # Connection Mode
        mode: str = None,                     # 'cloud', 'v26', or 'v25' (auto-detected if
None)

        # Endpoints
        hostname: str = "data.dremio.cloud", # Dremio hostname
         port: int = None,                       # REST API port (auto-detected based on
mode)
        base_url: str = None,                # Custom base URL (overrides auto-detection)

        # Arrow Flight
         flight_port: int = None,             # Arrow Flight port (auto-detected based on
mode)
           flight_endpoint: str = None,           # Arrow Flight endpoint (defaults to
hostname)

        # Security
        tls: bool = True,                    # Enable TLS/SSL
        disable_certificate_verification: bool = False  # Disable SSL cert verification
    ):
        ...
```

Connection Mode Details

The `mode` parameter automatically configures ports and endpoints for different Dremio versions:

`mode="cloud"` (Default)

- **REST API**: `https://api.dremio.cloud/v0` (port 443)
- **Arrow Flight**: `grpc+tls://data.dremio.cloud:443`
- **Authentication**: PAT with Bearer token
- **Auto-detected when**: `hostname == "data.dremio.cloud"` or `project_id` is set

`mode="v26"` (Dremio Software v26+)

- **REST API**: `https://{hostname}:{port}/api/v3` (port 443 with TLS, 9047 without)
- **Arrow Flight**: `grpc+tls://{hostname}:32010` (or `grpc+tcp` without TLS)
- **Authentication**: PAT with Bearer token or username/password

Auto-detected when: `DREMIO_SOFTWARE_HOST` or `DREMIO_SOFTWARE_PAT` env vars are set

`mode="v25"` (Dremio Software v25 and earlier)

REST API: `http://{hostname}:9047/api/v3`

Arrow Flight: `grpc+tcp://{hostname}:32010`

Authentication: Username/password only

Login Endpoint: `/apiv2/login`

Port Configuration

Ports are automatically configured based on the `mode`, but can be overridden:

Mode	REST Port (default)	Flight Port (default)
`cloud`	443	443
`v26` (TLS)	443	32010
`v26` (no TLS)	9047	32010
`v25`	9047	32010

Override examples:

```
# Custom REST API port
client = DremioClient(hostname="custom.dremio.com", port=8443, mode="v26")

# Custom Flight port
client = DremioClient(hostname="custom.dremio.com", flight_port=31010, mode="v26")
```

Example .env File

For Dremio Cloud

```
DREMIO_PAT=your_cloud_pat_here
DREMIO_PROJECT_ID=your_project_id_here
```

For Dremio Software v26+

```
DREMIO_SOFTWARE_HOST=https://v26.dremio.org
DREMIO_SOFTWARE_PAT=your_software_pat_here
DREMIO_SOFTWARE_TLS=true
# Optional: Override default ports
# DREMIO_SOFTWARE_PORT=443
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

For Dremio Software v25

```
DREMIO_SOFTWARE_HOST=localhost
DREMIO_SOFTWARE_USER=admin
DREMIO_SOFTWARE_PASSWORD=password123
DREMIO_SOFTWARE_TLS=false
```

Troubleshooting

Connection Issues

Problem: "Connection timeout" or "Connection refused"

Solution: Verify the hostname and ports are correct. For Software, ensure the Dremio coordinator is running.

Problem: "Authentication failed"

Solution:

- For Cloud: Verify your PAT and Project ID are correct
- For Software v26+: Ensure your PAT has the necessary permissions
- For Software v25: Verify username/password are correct

Problem: "Flight queries fail but REST API works"

Solution: Check that the `flight_port` is correct (default: 32010 for Software). Verify Arrow Flight is enabled on your Dremio instance.

Mode Selection

If auto-detection isn't working correctly, explicitly set the `mode` parameter:

```
# Force v26 mode
client = DremioClient(hostname="my-dremio.com", pat="...", mode="v26")

# Force cloud mode
client = DremioClient(pat="...", project_id="...", mode="cloud")
```

Source: connection.md

Connecting to Dremio

This guide provides detailed instructions on how to connect `dremioframe` to your Dremio environment, whether it's Dremio Cloud or a self-managed Dremio Software instance.

Overview

DremioFrame supports three connection modes:

`**cloud**`: Dremio Cloud (SaaS)

`**v26**`: Dremio Software v26+ (with PAT support)

`**v25**`: Dremio Software v25 and earlier

The `mode` parameter automatically configures ports, endpoints, and authentication methods. In most cases, the mode is auto-detected, but you can specify it explicitly for clarity.

1. Dremio Cloud

Dremio Cloud is the default connection mode. It uses Arrow Flight SQL over TLS.

Prerequisites

Personal Access Token (PAT): Generate this in your Dremio Cloud User Settings

Project ID: The ID of the project you want to query

Environment Variables

Set these in your `.env` file or environment:

```
DREMIO_PAT=your_cloud_personal_access_token
DREMIO_PROJECT_ID=your_project_id_here
```

Connection Examples

Using Environment Variables (Recommended)

```
from dremioframe.client import DremioClient

# Client automatically picks up env vars and detects Cloud mode
client = DremioClient()

# Or explicitly specify mode for clarity
client = DremioClient(mode="cloud")
```

Using Explicit Parameters

```
client = DremioClient(
    pat="your_pat_here",
    project_id="your_project_id_here",
```

```
    mode="cloud"  # Optional, auto-detected when project_id is provided
)
```

Custom Flight Configuration

You can specify a custom Flight endpoint if needed:

```
client = DremioClient(
    pat="my_token",
    project_id="my_project",
    flight_endpoint="flight.dremio.cloud",
    flight_port=443,
    mode="cloud"
)
```

Default Ports (Cloud)

REST API: Port 443 (`https://api.dremio.cloud/v0`)

Arrow Flight: Port 443 (`grpc+tls://data.dremio.cloud:443`)

2. Dremio Software v26+

Dremio Software v26+ supports Personal Access Tokens (PAT) for authentication, similar to Cloud.

Prerequisites

Hostname: The address of your Dremio coordinator (e.g., `v26.dremio.org` or `localhost`)

Personal Access Token (PAT) OR **Username/Password**

TLS: Whether your instance uses TLS/SSL

Environment Variables

Set these in your `.env` file:

```
DREMIO_SOFTWARE_HOST=https://v26.dremio.org
DREMIO_SOFTWARE_PAT=your_software_personal_access_token
DREMIO_SOFTWARE_TLS=true

# Optional: Override default ports
# DREMIO_SOFTWARE_PORT=443
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

Note: `DREMIO_SOFTWARE_HOST` can include the protocol (`https://` or `http://`). If TLS is true and no port is specified in the URL, port 443 is used for REST API.

Important: For Arrow Flight connections in v26+, the client uses Basic Authentication (Username + PAT). If you only provide a PAT, the client will attempt to automatically discover your username from the catalog. If discovery fails, you may need to provide `DREMIO_SOFTWARE_USER` explicitly.

Connection Examples

Using PAT with Environment Variables (Recommended)

```
from dremioframe.client import DremioClient

# Auto-detects v26 mode from DREMIO_SOFTWARE_* env vars
client = DremioClient()

# Or explicitly specify mode
client = DremioClient(mode="v26")
```

Using PAT with Explicit Parameters

```
client = DremioClient(
    hostname="v26.dremio.org",
    pat="your_pat_here",
    tls=True,
    mode="v26"
)
```

Using Username/Password

```
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password123",
    tls=False,
    mode="v26"
)
```

Default Ports (v26)

REST API: Port 443 (with TLS) or 9047 (without TLS)

Arrow Flight: Port 32010

Login Endpoint: `/api/v3/login` or `/apiv3/login`

TLS/SSL Configuration

If your Dremio Software cluster uses TLS (Encryption), set `tls=True`:

```
client = DremioClient(
    hostname="secure.dremio.com",
    pat="your_pat",
    tls=True,
    mode="v26"
)
```

Self-Signed Certificates

For self-signed certificates (common in dev/test environments):

```
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password123",
    tls=True,
    disable_certificate_verification=True,
    mode="v26"
)
```

Warning: Disabling certificate verification is insecure and should not be used in production.

3. Dremio Software v25 and Earlier

Dremio Software v25 and earlier versions use username/password authentication only (no PAT support).

Prerequisites

Hostname: The address of your Dremio coordinator

Username and Password: Valid Dremio credentials

Environment Variables

Set these in your `.env` file:

```
DREMIO_SOFTWARE_HOST=localhost
DREMIO_SOFTWARE_USER=admin
DREMIO_SOFTWARE_PASSWORD=password123
DREMIO_SOFTWARE_TLS=false

# Optional: Override default ports
# DREMIO_SOFTWARE_PORT=9047
```

```
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

Connection Examples

Using Environment Variables

```
from dremioframe.client import DremioClient

client = DremioClient(mode="v25")
```

Using Explicit Parameters

```
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password123",
    tls=False,
    mode="v25"
)
```

Default Ports (v25)

REST API: Port 9047 (`http://localhost:9047/api/v3`)

Arrow Flight: Port 32010

Login Endpoint: `/apiv2/login`

4. Port Configuration Reference

Ports are automatically configured based on the `mode`, but can be overridden:

Mode	REST Port (default)	Flight Port (default)	Protocol
`cloud`	443	443	HTTPS/gRPC+TLS
`v26` (TLS)	443	32010	HTTPS/gRPC+TLS
`v26` (no TLS)	9047	32010	HTTP/gRPC+TCP
`v25`	9047	32010	HTTP/gRPC+TCP

Overriding Ports

```
# Custom REST API port
client = DremioClient(
    hostname="custom.dremio.com",
    port=8443,
    flight_port=31010,
```

```
    mode="v26"
)
```

5. Mode Auto-Detection

The client automatically detects the mode based on:

Cloud mode is detected when:

- `hostname == "data.dremio.cloud"`, OR
- `project_id` is provided, OR
- `DREMIO_PROJECT_ID` environment variable is set

v26 mode is detected when:

- `DREMIO_SOFTWARE_HOST` or `DREMIO_SOFTWARE_PAT` environment variables are set

Default: Falls back to `cloud` mode if no indicators are found

Explicit Mode Selection

For clarity and to avoid ambiguity, you can always specify the mode explicitly:

```
# Force v26 mode
client = DremioClient(
    hostname="my-dremio.com",
    pat="...",
    mode="v26"
)

# Force cloud mode
client = DremioClient(
    pat="...",
    project_id="...",
    mode="cloud"
)

# Force v25 mode
client = DremioClient(
    hostname="localhost",
    username="admin",
    password="password",
    mode="v25"
)
```

6. Complete .env File Examples

For Dremio Cloud

```
# Required
DREMIO_PAT=dremio_pat_abc123xyz456...
DREMIO_PROJECT_ID=12345678-1234-1234-1234-123456789abc

# Optional
# DREMIO_FLIGHT_ENDPOINT=data.dremio.cloud
# DREMIO_FLIGHT_PORT=443
```

For Dremio Software v26+ (with PAT)

```
# Required
DREMIO_SOFTWARE_HOST=https://v26.dremio.org
DREMIO_SOFTWARE_PAT=dremio_pat_xyz789...
DREMIO_SOFTWARE_TLS=true

# Optional
# DREMIO_SOFTWARE_PORT=443
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

For Dremio Software v26+ (with Username/Password)

```
# Required
DREMIO_SOFTWARE_HOST=localhost
DREMIO_SOFTWARE_USER=admin
DREMIO_SOFTWARE_PASSWORD=password123
DREMIO_SOFTWARE_TLS=false

# Optional
# DREMIO_SOFTWARE_PORT=9047
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

For Dremio Software v25

```
# Required
DREMIO_SOFTWARE_HOST=localhost
DREMIO_SOFTWARE_USER=admin
DREMIO_SOFTWARE_PASSWORD=password123
DREMIO_SOFTWARE_TLS=false

# Optional
# DREMIO_SOFTWARE_PORT=9047
# DREMIO_SOFTWARE_FLIGHT_PORT=32010
```

7. Troubleshooting & Common Errors

`FlightUnavailableError` / Connection Refused

Symptoms: The client hangs or raises an error saying the service is unavailable.

Causes:

Wrong Port: Ensure you are using the **Arrow Flight Port** (default `32010` for Software, `443` for Cloud), NOT the UI port (`9047`) or ODBC/JDBC port (`31010`)

Firewall: Ensure the port is open and accessible

Dremio Down: Check if the Dremio service is running

Wrong Mode: Ensure you're using the correct mode (`cloud`, `v26`, or `v25`)

Solution:

```
# Verify your mode and ports
client = DremioClient(
    hostname="your-host",
    mode="v26",  # Explicitly set mode
    flight_port=32010  # Explicitly set Flight port if needed
)
```

`FlightUnauthenticatedError` / Auth Failed

Symptoms: "Invalid credentials" or "Unauthenticated".

Causes:

Expired PAT: Tokens expire. Generate a new one

Wrong Project ID: For Cloud, ensure the Project ID matches

Wrong Mode: Using Cloud credentials with Software mode or vice versa

Typo: Double-check credentials

Solution:

```
# For Cloud, ensure both PAT and project_id are correct
client = DremioClient(pat="...", project_id="...", mode="cloud")

# For Software v26+, ensure PAT is valid
client = DremioClient(hostname="...", pat="...", mode="v26")

# For Software v25, use username/password
client = DremioClient(hostname="...", username="...", password="...", mode="v25")
```

`FlightInternalError` (Certificate Issues)

Symptoms: "Handshake failed", "Certificate verify failed".

Causes:

- **TLS Mismatch**: You set `tls=True` but the server uses `tls=False` (or vice versa)
- **Self-Signed Cert**: Connecting to TLS-enabled server with self-signed certificate

Solution:

```
# For self-signed certificates
client = DremioClient(
    hostname="localhost",
    tls=True,
    disable_certificate_verification=True,
    mode="v26"
)
```

Environment Variable Conflicts

Symptoms: Client connects to wrong environment or uses wrong credentials.

Causes:

- Both `DREMIO_PAT` and `DREMIO_SOFTWARE_PAT` are set
- Both Cloud and Software environment variables are set

Solution:

```
# Explicitly specify mode to avoid ambiguity
client = DremioClient(mode="v26")  # Forces Software mode

# Or unset conflicting environment variables
import os
if "DREMIO_PROJECT_ID" in os.environ:
    del os.environ["DREMIO_PROJECT_ID"]
```

8. Testing Connectivity

Verify your connection with a simple test:

```
from dremioframe.client import DremioClient

try:
    # Create client (adjust mode as needed)
    client = DremioClient(mode="v26")

    # Test catalog access
    catalog = client.catalog.list_catalog()
    print(f" Connected successfully! Found {len(catalog)} catalog items.")

    # Test query execution (requires Arrow Flight)
    result = client.query("SELECT 1 as test")
```

```
    print(f" Query execution successful!")
    print(result)

except Exception as e:
    print(f" Connection failed: {e}")
    import traceback
    traceback.print_exc()
```

Quick Diagnostic Script

```
from dremioframe.client import DremioClient
import os

print("Environment Variables:")
print(f"  DREMIO_PAT: {'SET' if os.getenv('DREMIO_PAT') else 'NOT SET'}")
print(f"  DREMIO_PROJECT_ID: {'SET' if os.getenv('DREMIO_PROJECT_ID') else 'NOT SET'}")
print(f"  DREMIO_SOFTWARE_HOST: {os.getenv('DREMIO_SOFTWARE_HOST', 'NOT SET')}")
print(f"    DREMIO_SOFTWARE_PAT:  {'SET'  if  os.getenv('DREMIO_SOFTWARE_PAT')  else  'NOT
SET'}")
print(f"  DREMIO_SOFTWARE_USER: {os.getenv('DREMIO_SOFTWARE_USER', 'NOT SET')}")

client = DremioClient(mode="v26")  # Adjust mode as needed
print(f"\nClient Configuration:")
print(f"  Mode: {client.mode}")
print(f"  Hostname: {client.hostname}")
print(f"  REST Port: {client.port}")
print(f"  Flight Port: {client.flight_port}")
print(f"  Base URL: {client.base_url}")
print(f"  Project ID: {client.project_id}")
```

Source: cookbook.md

DremioFrame Cookbook

A collection of recipes for common data engineering tasks using valid Dremio SQL patterns.

Deduplicating Data

Remove duplicate records based on specific columns, keeping the most recent one.

```
# Keep the row with the latest 'updated_at' for each 'id'
df = client.table("source_table") \
    .select("*",
            F.row_number().over(
                F.Window.partition_by("id").order_by("updated_at", ascending=False)
            ).alias("rn")
    ) \
```

```
    .filter("rn = 1") \
    .drop("rn")

# Save as new table
df.create("deduplicated_table")
```

Pivoting Data

Transform rows into columns (e.g., monthly sales).

```
# Source: region, month, sales
# Target: region, jan_sales, feb_sales, ...

df = client.table("monthly_sales") \
    .group_by("region") \
    .agg(
        jan_sales="SUM(CASE WHEN month = 'Jan' THEN sales ELSE 0 END)",
        feb_sales="SUM(CASE WHEN month = 'Feb' THEN sales ELSE 0 END)",
        mar_sales="SUM(CASE WHEN month = 'Mar' THEN sales ELSE 0 END)"
    )

df.show()
```

Incremental Loading (Watermark)

Load only new data based on a watermark (max timestamp).

```
# Get max timestamp from target
max_ts                                                                                  =
client.table("target_table").agg(max_ts="MAX(updated_at)").collect().iloc[0]['max_ts']

# Fetch only new data from source
new_data = client.table("source_table").filter(f"updated_at > '{max_ts}'")

# Append to target
new_data.insert("target_table")
```

Exporting to S3 (Parquet)

```
# Option 1: Client-Side Export (Requires local credentials & s3fs)
# Fetches data to client and writes to S3
df = client.table("warehouse.sales").collect()
df.to_parquet("s3://my-bucket/export/data.parquet")

# Option 2: Materialize to Source (CTAS)
# Creates a new table (Iceberg or Parquet folder) in the S3 source
client.sql("""
    CREATE TABLE "s3_source"."bucket"."folder"."new_table"
```

```
    AS SELECT * FROM "warehouse"."sales"
""")
```

Handling JSON Data

Access nested fields in JSON/Struct columns using dot notation.

```
# Source has a 'details' column (Struct or Map): {"color": "red", "size": "M"}
df = client.table("products") \
    .select(
        "id",
        "name",
        F.col("details.color").alias("color"),
        F.col("details.size").alias("size")
    )

df.show()
```

Unnesting Arrays (Flatten)

Explode a list column into multiple rows.

```
# Source: id, tags (["A", "B"])
# Target: id, tag (one row per tag)

df = client.table("posts") \
    .select(
        "id",
        F.flatten("tags").alias("tag")
    )

df.show()
```

Date Arithmetic

Perform calculations on dates.

```
# Calculate deadline (created_at + 7 days) and days_overdue
df = client.table("tasks") \
    .select(
        "id",
        "created_at",
        F.date_add("created_at", 7).alias("deadline"),
        F.date_diff(F.current_date(), "created_at").alias("days_since_creation")
    )

df.show()
```

String Manipulation

Clean and transform text data.

```
# Normalize email addresses
df = client.table("users") \
    .select(
        "id",
        F.lower(F.trim("email")).alias("clean_email"),
        F.substr("phone", 1, 3).alias("area_code")
    )

df.show()
```

Window Functions (Running Total)

Calculate cumulative sums or moving averages.

```
# Calculate running total of sales by date
df = client.table("sales") \
    .select(
        "date",
        "amount",
        F.sum("amount").over(
            F.Window.order_by("date").rows_between("UNBOUNDED PRECEDING", "CURRENT ROW")
        ).alias("running_total")
    )

df.show()
```

Approximate Count Distinct

Estimate the number of distinct values for large datasets (faster than COUNT DISTINCT).

```
# Estimate unique visitors
df = client.table("web_logs") \
    .agg(
        unique_visitors=F.approx_distinct("visitor_id")
    )

df.show()
```

AI Functions (Generative AI)

Use Dremio's AI functions to classify text or generate content.

```
# Classify customer feedback
df = client.table("feedback") \
```

```
    .select(
        "comment",
        F.ai_classify("comment", ["Positive", "Negative", "Neutral"]).alias("sentiment")
    )

df.show()
```

Time Travel (Snapshot Querying)

Query an Iceberg table as it existed at a specific point in time.

```
# Query specific snapshot
df = client.table("iceberg_table").at_snapshot("1234567890")

# Query by timestamp
df = client.table("iceberg_table").at_timestamp("2023-10-27 10:00:00")

df.show()
```

Schema Evolution

Add a new column to an existing Iceberg table.

```
# Add 'status' column
client.sql('ALTER TABLE "iceberg_table" ADD COLUMNS (status VARCHAR)')
```

Creating Partitioned Tables

Create a new table partitioned by specific columns for better performance.

```
# Create table partitioned by 'region' and 'date'
client.sql("""
    CREATE TABLE "iceberg_source"."new_table"
    PARTITION BY (region, date)
    AS SELECT * FROM "source_table"
""")
```

Map & Struct Access

Access values within Map and Struct data types.

```
# Struct: details.color
# Map: properties['priority']

df = client.table("events") \
    .select(
        "id",
```

```
        F.col("details.color").alias("color"),
        F.col("properties['priority']").alias("priority")
    )

df.show()
```

Source: dependencies.md

Optional Dependencies

DremioFrame uses optional dependencies to keep the core package lightweight.

Server

`server`: `mcp` (Required for running the MCP Server)

AI

with a core set of dependencies. However, many advanced features require additional packages. You can install these optional dependencies individually or in groups.

Installation Syntax

To install optional dependencies, use the square bracket syntax with pip:

```
pip install "dremioframe[group_name]"
```

To install multiple groups:

```
pip install "dremioframe[group1,group2]"
```

Dependency Groups

Core Features

Group	Dependencies	Features Enabled
`cli`	`rich`, `prompt_toolkit`	Enhanced CLI experience with rich text and interactive prompts for working with Orchestration and AI features.
`s3`	`boto3`	S3 integration for direct file operations and source management.
`scheduler`	`apscheduler`	Built-in task scheduling capabilities.
`dq`	`pyyaml`	Data Quality framework configuration parsing.
`ai`	`langchain`, `langchain-openai`, `langchain-anthropic`, `langchain-google-genai`	AI-powered Agent for Generating Python Scripts, SQL and cURL commands and light admin work.

| `mcp` | `langchain-mcp-adapters` | Model Context Protocol server integration for extending AI agent with custom tools. |
| `document` | `pdfplumber` | PDF document extraction for AI agent to read and extract data from PDF files. |
note: this libraries embdedded agent is primarily meant as a code generation assist tool, not meant as an alternative to the integrated Dremio agent for deeper administration and natural language analytics.

File Formats & Export

Group	Dependencies	Features Enabled
`excel`	`openpyxl`	Reading and writing Excel files.
`html`	`lxml`, `html5lib`	Parsing HTML tables.
`avro`	`fastavro`	Support for Avro file format.
`lance`	`pylance`	Support for Lance file format.
`image_export`	`kaleido`	Exporting Plotly charts as static images (PNG, JPG, PDF).

Data Ingestion

Group	Dependencies	Features Enabled
`ingest`	`dlt`	Load data from 100+ sources (APIs, SaaS, databases) using dlt integration.
`database`	`connectorx`, `sqlalchemy`	High-performance SQL database ingestion (Postgres, MySQL, SQLite, etc.).
`notebook`	`tqdm`, `ipywidgets`	For Jupyter notebook integration.
`delta`	`deltalake`	For Delta Lake export.
`lineage`	`networkx`, `graphviz`	For data lineage visualization.

External Backends

Group	Dependencies	Features Enabled
`postgres`	`psycopg2-binary`	Support for using PostgreSQL as an orchestration backend.
`mysql`	`mysql-connector-python`	Support for using MySQL as an orchestration backend.
`celery`	`celery`, `redis`	Distributed task execution using Celery and Redis.
`airflow`	`apache-airflow`	Integration with Apache Airflow for orchestrating Dremio workflows.

Development & Documentation

Group	Dependencies	Features Enabled
`dev`	`pytest`, `pytest-asyncio`, `requests-mock`	Running the test suite and contributing to DremioFrame.
`docs`	`mkdocs`, `mkdocs-material`, `mkdocstrings[python]`	Building and serving the documentation locally.

Feature-Specific Requirements

Orchestration

Local Execution: No extra dependencies required.

Distributed Execution: Requires `celery`.

Persistent State: Requires a backend like `postgres` or `mysql` (or uses local SQLite by default).

AI Functions

To use the AI agent for script/SQL generation, you must install the `ai` group:

```
pip install "dremioframe[ai]"
```

This includes support for:

- Script, SQL, and API call generation
- Conversation memory persistence (via SQLite)
- Context folder integration for project-specific files

Chart Exporting

To save charts as images using `chart.save("plot.png")`, you need the `image_export` group:

```
pip install "dremioframe[image_export]"
```

Source: s3_integration.md

S3 Integration

DremioFrame provides seamless integration with Amazon S3 through the optional `s3` dependency. This allows you to perform file operations directly within your orchestration pipelines, such as uploading data files before ingestion or downloading results.

Installation

To use S3 features, you must install the `s3` optional dependency group, which includes `boto3`.

```
pip install "dremioframe[s3]"
```

S3Task

The primary way to interact with S3 is through the `S3Task` in the orchestration module. This task wraps `boto3` operations into a reusable pipeline component.

Supported Operations

`**upload_file**`: Upload a local file to an S3 bucket.

`**download_file**`: Download a file from an S3 bucket to a local path.

Configuration

You can configure credentials directly in the task or rely on environment variables (recommended).

Environment Variables:

`AWS_ACCESS_KEY_ID`

`AWS_SECRET_ACCESS_KEY`

`AWS_DEFAULT_REGION`

Examples

Uploading a File

```
from dremioframe.orchestration import Pipeline
from dremioframe.orchestration.tasks import S3Task

# Define the task
upload_task = S3Task(
    name="upload_csv",
    operation="upload_file",
    bucket="my-datalake-bucket",
    key="raw/data.csv",
    local_path="./data/local_data.csv"
)

# Create and run pipeline
pipeline = Pipeline("s3_ingest")
pipeline.add_task(upload_task)
pipeline.run()
```

Downloading a File

```
download_task = S3Task(
    name="download_report",
    operation="download_file",
```

```
    bucket="my-reports-bucket",
    key="monthly/report.pdf",
    local_path="./downloads/report.pdf"
)
```

Using Custom Credentials

```
custom_s3_task = S3Task(
    name="upload_secure",
    operation="upload_file",
    bucket="secure-bucket",
    key="data.csv",
    local_path="./data.csv",
    aws_access_key_id="AKIA...",
    aws_secret_access_key="SECRET...",
    region_name="us-west-2"
)
```

Using MinIO or S3-Compatible Storage

You can connect to S3-compatible storage like MinIO by providing an `endpoint_url`.

```
minio_task = S3Task(
    name="upload_minio",
    operation="upload_file",
    bucket="test-bucket",
    key="test.csv",
    local_path="./test.csv",
    endpoint_url="http://localhost:9000",
    aws_access_key_id="minioadmin",
    aws_secret_access_key="minioadmin"
)
```

Source: troubleshooting.md

Troubleshooting Guide

Common issues and solutions when using DremioFrame.

Connectivity Issues

Arrow Flight: Connection Refused

Error: `pyarrow.lib.ArrowIOError: Flight returned unavailable error, with message: failed to connect to all addresses`

Cause: The client cannot reach the Dremio Flight endpoint.
Solution:

- Verify `hostname` and `port` (default 32010 for Software, 443 for Cloud).
- Ensure firewall allows traffic on the Flight port.
- If using Docker, ensure ports are mapped (`-p 32010:32010`).

Authentication Failed

Error: `pyarrow.flight.FlightUnauthenticatedError`
Cause: Invalid credentials.
Solution:

- **Cloud**: Check `DREMIO_PAT` is valid and not expired.
- **Software**: Verify username/password.
- Ensure `DREMIO_PROJECT_ID` is set for Cloud.

Orchestration Issues

Backend Import Errors

Error: `ImportError: psycopg2 is required...`
Cause: Missing optional dependencies.
Solution:

- Install with extras: `pip install "dremioframe[postgres]"` or `pip install "dremioframe[mysql]"`.

Task Execution Fails Immediately

Cause: Missing environment variables or configuration in the execution environment (e.g., inside a Docker container).
Solution:

- Pass environment variables to the container or worker process.

Debugging

Enable debug logging to see detailed request/response info:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

This will output REST API calls and Flight connection details.

Source: tutorial_etl.md

Tutorial: Building a Production ETL Pipeline

This tutorial guides you through building a complete ETL pipeline using DremioFrame. You will ingest data, transform it, validate it, and schedule the job.

Prerequisites

Dremio Cloud or Software instance.

Python 3.8+.

`dremioframe` installed.

Step 1: Connect to Dremio

Create a script `etl_pipeline.py`.

```
import os
from dremioframe.client import DremioClient
from dremioframe.orchestration import Pipeline, DremioQueryTask
from dremioframe.orchestration.tasks.dq_task import DataQualityTask
from dremioframe.orchestration.scheduling import schedule_pipeline

# Initialize Client
client = DremioClient(
    pat=os.environ.get("DREMIO_PAT"),
    project_id=os.environ.get("DREMIO_PROJECT_ID")
)
```

Step 2: Define the Pipeline

```
pipeline = Pipeline("daily_sales_etl")
```

Step 3: Ingest Data (Extract & Load)

Assume we are ingesting from an external source into a staging table.

```
ingest_task = DremioQueryTask(
    name="ingest_sales",
    client=client,
    sql="""
    CREATE TABLE IF NOT EXISTS "Target.staging.sales" AS
    SELECT * FROM "Source.external.sales_data"
    """
)
pipeline.add_task(ingest_task)
```

Step 4: Transform Data

Clean and aggregate the data.

```
transform_task = DremioQueryTask(
    name="transform_sales",
    client=client,
    sql="""
    CREATE OR REPLACE TABLE "Target.mart.daily_sales" AS
    SELECT
        date_trunc('day', sale_date) as sale_day,
        region,
        SUM(amount) as total_revenue
    FROM "Target.staging.sales"
    GROUP BY 1, 2
    """
)
pipeline.add_task(transform_task)

# Set dependency
transform_task.set_upstream(ingest_task)
```

Alternative: Using DataFrame API (DremioBuilderTask)

Instead of raw SQL, you can use the Pythonic DataFrame API.

```
from dremioframe.orchestration import DremioBuilderTask

# Define transformation using Builder
builder = client.table("Target.staging.sales") \
    .group_by("sale_date", "region") \
    .agg(total_revenue="SUM(amount)")

# Create task to merge results into mart
transform_task = DremioBuilderTask(
    name="transform_sales_builder",
    builder=builder,
    command="merge",
    target="Target.mart.daily_sales",
    options={
        "on": ["sale_date", "region"],
        "matched_update": {"total_revenue": "source.total_revenue"},
        "not_matched_insert": {
            "sale_date": "source.sale_date",
            "region": "source.region",
            "total_revenue": "source.total_revenue"
        }
    }
)
pipeline.add_task(transform_task)
```

Step 5: Validate Data (Quality Check)

Ensure the transformed data is valid. Create a test file `tests/dq/sales_checks.yaml`:

```
- name: check_revenue_positive
  table: "Target.mart.daily_sales"
  type: custom_sql
  condition: "total_revenue >= 0"
  error_msg: "Revenue cannot be negative"

- name: check_regions_exist
  table: "Target.mart.daily_sales"
  type: not_null
  column: region
```

Add the DQ task to the pipeline:

```
dq_task = DataQualityTask(
    name="validate_sales",
    client=client,
    directory="tests/dq"
)
pipeline.add_task(dq_task)

dq_task.set_upstream(transform_task)
```

Step 6: Run or Schedule

To run immediately:

```
if __name__ == "__main__":
    pipeline.run()
```

To schedule daily at 2 AM:

```
if __name__ == "__main__":
    schedule_pipeline(pipeline, "cron", hour=2, minute=0)
```

Conclusion

You have built a robust pipeline that ingests, transforms, and validates data, with automatic failure handling and scheduling!

Source: airflow.md

Airflow Integration

DremioFrame provides native integration with Apache Airflow, allowing you to orchestrate Dremio workflows within your Airflow DAGs.

Installation

Install DremioFrame with Airflow support:

```
pip install "dremioframe[airflow]"
```

This will install `apache-airflow` as a dependency.

Components

DremioHook

The `DremioHook` wraps the `DremioClient` and integrates with Airflow's connection management system.

Configuration

Create a Dremio connection in Airflow:

Via Airflow UI:

1. Navigate to Admin → Connections
2. Add a new connection with the following details:
   - **Connection ID**: `dremio_default` (or custom name)
   - **Connection Type**: `Dremio` (or `Generic` if custom type not available)
   - **Host**: `data.dremio.cloud` (or your Dremio hostname)
   - **Port**: `443` (or your port)
   - **Login**: Username (for Dremio Software with username/password auth)
   - **Password**: Password or PAT
   - **Extra**: JSON with additional config

Extra JSON Fields:

```
{
  "pat": "your_personal_access_token",
  "project_id": "your_project_id",
  "tls": true,
  "disable_certificate_verification": false
}
```

Via Environment Variable:

```
export AIRFLOW_CONN_DREMIO_DEFAULT='{"conn_type": "dremio", "host": "data.dremio.cloud",
"port": 443, "extra": {"pat": "YOUR_PAT", "project_id": "YOUR_PROJECT_ID"}}'
```

Usage

```
from dremioframe.airflow import DremioHook

hook = DremioHook(dremio_conn_id="dremio_default")
client = hook.get_conn()

# Execute SQL
df = hook.get_pandas_df("SELECT * FROM my_table LIMIT 10")

# Get records as list of dicts
records = hook.get_records("SELECT * FROM my_table LIMIT 10")
```

DremioSQLOperator

Execute SQL queries in Dremio as part of your Airflow DAG.

Parameters

`sql` (str): The SQL query to execute. Supports Jinja templating.

`dremio_conn_id` (str): Connection ID. Default: `"dremio_default"`.

`return_result` (bool): Whether to return results as XCom. Default: `False`.

Example

```
from airflow import DAG
from airflow.utils.dates import days_ago
from dremioframe.airflow import DremioSQLOperator

default_args = {
    'owner': 'data_team',
    'start_date': days_ago(1),
}

with DAG('dremio_etl', default_args=default_args, schedule_interval='@daily') as dag:

    # Create a table
    create_table = DremioSQLOperator(
        task_id='create_staging_table',
        sql='''
            CREATE TABLE IF NOT EXISTS staging.daily_metrics AS
            SELECT
```

```
                DATE_TRUNC('day', event_time) as date,
                COUNT(*) as event_count,
                COUNT(DISTINCT user_id) as unique_users
            FROM raw.events
            WHERE DATE(event_time) = CURRENT_DATE - INTERVAL '1' DAY
            GROUP BY 1
        '''
    )

    # Run aggregation
    aggregate_data = DremioSQLOperator(
        task_id='aggregate_metrics',
        sql='''
            INSERT INTO analytics.daily_summary
            SELECT * FROM staging.daily_metrics
        '''
    )

    # Optimize table
    optimize_table = DremioSQLOperator(
        task_id='optimize_table',
        sql='OPTIMIZE TABLE analytics.daily_summary'
    )

    create_table >> aggregate_data >> optimize_table
```

Templating

The `sql` parameter supports Jinja templating:

```
DremioSQLOperator(
    task_id='process_partition',
    sql='''
        SELECT * FROM events
        WHERE date = '{{ ds }}'  -- Airflow execution date
    '''
)
```

DremioDataQualityOperator

Run data quality checks on Dremio tables.

Parameters

˜table_name` (str): The table to check. Supports Jinja templating.

˜checks` (list): List of check definitions.

˜dremio_conn_id` (str): Connection ID. Default: `"dremio_default"`.

Check Types

`not_null`: Check that a column has no NULL values.

`unique`: Check that a column has only unique values.

`row_count`: Check row count with a condition.

`values_in`: Check that column values are in a specified list.

Example

```
from dremioframe.airflow import DremioDataQualityOperator

dq_check = DremioDataQualityOperator(
    task_id='validate_daily_metrics',
    table_name='staging.daily_metrics',
    checks=[
        {
            "type": "not_null",
            "column": "date"
        },
        {
            "type": "row_count",
            "expr": "event_count > 0",
            "value": 1,
            "op": "ge"  # greater than or equal
        },
        {
            "type": "unique",
            "column": "date"
        }
    ]
)
```

If any check fails, the operator will raise a `ValueError` and fail the task.

Complete DAG Example

```
from airflow import DAG
from airflow.utils.dates import days_ago
from dremioframe.airflow import DremioSQLOperator, DremioDataQualityOperator

default_args = {
    'owner': 'data_team',
    'start_date': days_ago(1),
    'retries': 2,
}

with DAG(
    'dremio_daily_pipeline',
    default_args=default_args,
```

```
    schedule_interval='0 2 * * *',  # 2 AM daily
    catchup=False
) as dag:

    # Extract
    extract = DremioSQLOperator(
        task_id='extract_raw_data',
        sql='''
            CREATE TABLE staging.raw_events_{{ ds_nodash }} AS
            SELECT * FROM raw.events
            WHERE DATE(event_time) = '{{ ds }}'
        '''
    )

    # Transform
    transform = DremioSQLOperator(
        task_id='transform_data',
        sql='''
            CREATE TABLE staging.metrics_{{ ds_nodash }} AS
            SELECT
                user_id,
                COUNT(*) as event_count,
                MAX(event_time) as last_event
            FROM staging.raw_events_{{ ds_nodash }}
            GROUP BY user_id
        '''
    )

    # Data Quality
    dq_check = DremioDataQualityOperator(
        task_id='validate_metrics',
        table_name='staging.metrics_{{ ds_nodash }}',
        checks=[
            {"type": "not_null", "column": "user_id"},
            {"type": "row_count", "expr": "event_count > 0", "value": 1, "op": "ge"}
        ]
    )

    # Load
    load = DremioSQLOperator(
        task_id='load_to_analytics',
        sql='''
            INSERT INTO analytics.user_metrics
            SELECT '{{ ds }}' as date, *
            FROM staging.metrics_{{ ds_nodash }}
        '''
    )

    # Cleanup
    cleanup = DremioSQLOperator(
        task_id='cleanup_staging',
        sql='''
            DROP TABLE staging.raw_events_{{ ds_nodash }};
            DROP TABLE staging.metrics_{{ ds_nodash }};
```

```
        '''
    )

    extract >> transform >> dq_check >> load >> cleanup
```

Best Practices

1. Use XComs Sparingly

Avoid returning large datasets via XCom:

```
# Bad - returns large dataset
DremioSQLOperator(
    task_id='get_data',
    sql='SELECT * FROM large_table',
    return_result=True  # Avoid this for large results
)

# Good - process in Dremio, only return metadata
DremioSQLOperator(
    task_id='process_data',
    sql='CREATE TABLE result AS SELECT * FROM large_table WHERE ...'
)
```

2. Leverage Templating

Use Airflow's templating for dynamic queries:

```
DremioSQLOperator(
    task_id='partition_process',
    sql='''
        OPTIMIZE TABLE my_table
        WHERE partition_date = '{{ ds }}'
    '''
)
```

3. Idempotent Operations

Make your SQL idempotent for safe retries:

```
# Use CREATE TABLE IF NOT EXISTS
DremioSQLOperator(
    task_id='create_table',
    sql='CREATE TABLE IF NOT EXISTS ...'
)

# Or use MERGE for upserts
DremioSQLOperator(
```

```
    task_id='upsert_data',
    sql='''
        MERGE INTO target USING source
        ON target.id = source.id
        WHEN MATCHED THEN UPDATE SET ...
        WHEN NOT MATCHED THEN INSERT ...
    '''
)
```

4. Connection Pooling

Reuse connections within a DAG by using the same `dremio_conn_id`.

5. Error Handling

Use Airflow's built-in retry mechanism:

```
DremioSQLOperator(
    task_id='flaky_operation',
    sql='...',
    retries=3,
    retry_delay=timedelta(minutes=5)
)
```

Comparison with Native Orchestration

DremioFrame includes its own lightweight orchestration engine. Here's when to use each:

Use Airflow When...	Use DremioFrame Orchestration When...
You already have Airflow infrastructure	You want a lightweight, standalone solution
You need complex scheduling (cron, sensors)	You need simple task dependencies
You integrate with many other systems	You only work with Dremio
You need enterprise features (RBAC, audit logs)	You want minimal dependencies
You have a dedicated data engineering team	You're a data analyst or scientist

You can also use both: develop pipelines with DremioFrame orchestration, then migrate to Airflow for production.

Troubleshooting

Connection Issues

If you see `ModuleNotFoundError: No module named 'dremioframe'`:

- Ensure DremioFrame is installed in the Airflow environment

- Check `pip list | grep dremioframe`

Authentication Errors

If you see authentication failures:

- Verify PAT is valid: `curl -H "Authorization: Bearer YOUR_PAT" https://api.dremio.cloud/v0/projects`
- Check connection configuration in Airflow UI
- Ensure `project_id` is set for Dremio Cloud

Query Timeouts

For long-running queries:

- Increase Airflow task timeout: `execution_timeout=timedelta(hours=2)`
- Consider breaking into smaller tasks
- Use Dremio reflections to accelerate queries

See Also

- Orchestration Overview
- Data Quality
- Administration

Source: notebook.md

Notebook Integration

DremioFrame provides rich integration with Jupyter Notebooks and other interactive environments (VS Code, Colab), making it an excellent tool for data exploration and analysis.

Features

- **Rich DataFrame Display**: Automatically displays query results as formatted HTML tables.
- **Progress Bars**: Shows download progress for large datasets using `tqdm`.
- **Magic Commands**: IPython magic commands for quick SQL execution.

Installation

Ensure you have the notebook dependencies installed:

```
pip install dremioframe[notebook]
```

Rich Display

When you display a `DremioBuilder` object in a notebook, it automatically executes a preview query (LIMIT 20) and displays the results as a formatted HTML table, along with the generated SQL.

```
from dremioframe.client import DremioClient

client = DremioClient()
df = client.table("finance.bronze.trips")

# Displaying the builder object shows a preview
df
```

Progress Bars

When collecting large datasets, you can enable a progress bar to track the download status.

```
# Download with progress bar
pdf = df.collect(library='pandas', progress_bar=True)
```

Magic Commands

DremioFrame includes IPython magic commands to simplify your workflow.

Loading Magics

First, load the extension:

```
%load_ext dremioframe.notebook
```

Connecting

Connect to Dremio using `%dremio_connect`. You can pass arguments or rely on environment variables.

```
%dremio_connect pat=YOUR_PAT project_id=YOUR_PROJECT_ID
```

Executing SQL

Use `%%dremio_sql` to execute SQL queries directly in a cell.

```
%%dremio_sql my_result
SELECT
    trip_date,
    passenger_count
FROM finance.bronze.trips
WHERE passenger_count > 2
LIMIT 100
```

The result is automatically displayed and saved to the variable `my_result` (if specified).

Source: dimensional.md

Dimensional Modeling Guide

Dimensional modeling is a data structure technique optimized for data warehousing and reporting. It prioritizes **query performance** and **ease of use** for business analysts over the write-efficiency of normalized (3NF) schemas.

Core Concepts

1. Star Schema vs. Snowflake Schema

Star Schema (Recommended): A central Fact table surrounded by denormalized Dimension tables.

Pros: Simpler queries (fewer joins), faster performance in Dremio (Starflake optimization).

Cons: Redundant data in dimensions (e.g., repeating "USA" for every customer in New York).

Snowflake Schema: Dimensions are normalized into multiple related tables (e.g., `Fact_Sales` -> `Dim_Customer` -> `Dim_City` -> `Dim_Country`).

Pros: Less data redundancy.

Cons: Complex queries (many joins), slower performance.

[!TIP]
Dremio Recommendation: Prefer **Star Schemas**. Dremio's columnar nature handles the redundancy of denormalized dimensions efficiently (via compression), and fewer joins leads to better query planning.

2. Fact Tables

Fact tables store the quantitative data (metrics) of the business. They are typically the largest tables.

- **Transactional Fact**: One row per event (e.g., a single sale, a web click). High volume, most granular.

- **Periodic Snapshot Fact**: One row per entity per time period (e.g., Monthly Account Balance, Daily Inventory Level). Good for trend analysis.

- **Accumulating Snapshot Fact**: One row per lifecycle of a process (e.g., Order Fulfillment: Order Date, Ship Date, Delivery Date). Good for calculating lag times.

3. Dimension Tables

Dimension tables provide the "who, what, where, when, and why" context to the facts.

- **Conformed Dimension**: A dimension shared across multiple fact tables (e.g., `Dim_Date`, `Dim_Customer`). Crucial for cross-process analysis (Drill-Across).

- **Junk Dimension**: A collection of low-cardinality flags and indicators combined into a single table to avoid cluttering the fact table.

- **Degenerate Dimension**: A dimension key that has no associated attributes (e.g., `Transaction ID` in the Fact table).

- **Role-Playing Dimension**: A single physical dimension table referenced multiple times for different purposes (e.g., `Dim_Date` used for `Order Date`, `Ship Date`, and `Delivery Date`).

Implementation in DremioFrame

We recommend a **Bronze -> Silver -> Gold** workflow.

- **Bronze**: Raw data ingestion.

- **Silver**: Cleaned, deduplicated, and standardized data.

- **Gold**: Dimensional models (Star Schemas) ready for BI.

1. Generating Surrogate Keys

While Dremio handles string joins well, integer surrogate keys are standard in data warehousing for decoupling from source system keys and handling Slowly Changing Dimensions (SCD).

Strategy: Use `row_number()` or a hash function if you don't have a sequence generator.

```
# Creating a Dimension with a Surrogate Key
client.sql("""
    CREATE TABLE "gold"."dim_product" AS
    SELECT
        ROW_NUMBER() OVER (ORDER BY product_id) as product_sk, -- Surrogate Key
        product_id as product_nk, -- Natural Key (from source)
        product_name,
        category,
        brand,
```

```
        current_date() as valid_from,
        NULL as valid_to,
        TRUE as is_current
    FROM "silver"."products"
""")
```

2. The Date Dimension (`Dim_Date`)

A robust Date Dimension is essential. Do not rely on SQL date functions alone; a table allows you to filter by "Fiscal Quarter", "Holiday", "Weekday/Weekend", etc.

Generator Script:

```
import pandas as pd
from dremioframe.client import DremioClient

# 1. Generate Data in Pandas
start_date = "2020-01-01"
end_date = "2030-12-31"
dates = pd.date_range(start_date, end_date)

df = pd.DataFrame({"date_key": dates})
df["date_id"] = df["date_key"].dt.strftime("%Y%m%d").astype(int)
df["year"] = df["date_key"].dt.year
df["quarter"] = df["date_key"].dt.quarter
df["month"] = df["date_key"].dt.month
df["day_of_week"] = df["date_key"].dt.dayofweek + 1
df["day_name"] = df["date_key"].dt.day_name()
df["is_weekend"] = df["day_of_week"].isin([6, 7])

# 2. Write to Dremio (Iceberg)
client = DremioClient()
client.table("gold.dim_date").create("gold.dim_date", data=df)
```

3. Building the Fact Table

The Fact table joins Silver data with Dimensions to retrieve Surrogate Keys.

```
client.sql("""
    CREATE TABLE "gold"."fact_sales" AS
    SELECT
        p.product_sk,
        c.customer_sk,
        d.date_id,
        s.transaction_id,
        s.quantity,
        s.amount
    FROM "silver"."sales" s
    JOIN "gold"."dim_product" p ON s.product_id = p.product_nk AND p.is_current = TRUE
     JOIN "gold"."dim_customer" c ON s.customer_id = c.customer_nk AND c.is_current =
TRUE
```

```
    JOIN "gold"."dim_date" d ON s.sale_date = d.date_key
""")
```

Performance Optimization

1. Partitioning (Iceberg)

Partition your **Fact Tables** by the main time-based filter field (usually `date_id` or `transaction_date`).

Small/Medium Data: Partition by `Month` or `Year`.

Large Data: Partition by `Day`.

```
# Create table with partition transform
client.sql("""
    CREATE TABLE "gold"."fact_sales" (
        ...
    ) PARTITION BY (day(transaction_date))
""")
```

[!WARNING]
Avoid high-cardinality partitions (e.g., partitioning by `User ID` or `Timestamp`). This creates too many small files (Small File Problem).

2. Sorting

Sorting data before writing improves file pruning (Min/Max skipping). Sort Fact tables by the columns most frequently used in `WHERE` clauses (e.g., `customer_id`, `region`).

```
# In DremioFrame Builder
client.table("source").sort("region", "transaction_date").create("gold.fact_sales")
```

3. Aggregation Reflections

For Star Schemas, **Aggregation Reflections** are the most powerful optimization. They pre-calculate aggregates across dimensions.

Best Practice: Create an Aggregation Reflection on the **Fact Table**.

Dimensions: Add the Foreign Keys (e.g., `product_sk`, `customer_sk`, `date_id`).

Measures: Add the metrics (e.g., `SUM(amount)`, `COUNT(*)`).

```
client.admin.create_reflection(
    dataset_id="fact_sales_uuid",
    name="agg_sales_by_keys",
    type="AGGREGATION",
    dimension_fields=["product_sk", "customer_sk", "date_id"],
```

```
    measure_fields=["amount", "quantity"]
)
```

When a user queries `JOIN dim_product ... GROUP BY category`, Dremio automatically substitutes the reflection, avoiding the scan of the raw Fact table.

Source: documentation.md

Documenting Datasets

Documentation is critical for a self-service data platform. Dremio allows you to attach a Wiki (Markdown) and Tags to every dataset, source, space, and folder.

Wikis

The Wiki is the first thing users see when they open a dataset in Dremio. It should provide context, ownership, and usage instructions.

Updating Wikis Programmatically

You can automate documentation updates using `client.catalog.update_wiki`.

```
# Get the dataset ID
dataset = client.catalog.get_entity_by_path("marketing.customer_360")
dataset_id = dataset['id']

# Define Wiki Content (Markdown)
wiki_content = """
# Customer 360

This view provides a holistic view of customer activity, aggregated at the user level.

## Key Metrics
- **Lifetime Value (LTV)**: Total revenue generated by the customer.
- **Total Orders**: Count of all completed orders.

## Usage
Use this view for:
- Churn analysis
- Segmentation
- Email marketing campaigns

## Owner
**Marketing Data Team** (marketing-data@example.com)
"""

# Get current wiki version (required for updates to avoid 409 Conflict)
try:
    current_wiki = client.catalog.get_wiki(dataset_id)
    version = current_wiki.get("version")
```

```
except Exception:
    version = None

# Update the Wiki
client.catalog.update_wiki(dataset_id, wiki_content, version=version)
```

[!NOTE]
When updating a Wiki, it is best practice to fetch the current version first and pass it to `update_wiki`. This prevents overwriting concurrent changes and avoids `409 Conflict` errors, which are common in Dremio Software.

Retrieving Wiki Content

```
wiki = client.catalog.get_wiki(dataset_id)
print(wiki.get("text"))
```

Tagging

Tags help organize and discover datasets. You can use tags to indicate status, department, or project.

Best Practices for Tags

Status: `production`, `staging`, `deprecated`

Department: `marketing`, `finance`, `engineering`

Compliance: `pii`, `gdpr`, `hipaa`

Managing Tags

```
# Tag the dataset as 'certified' and 'production'
# Note: set_tags overwrites existing tags
client.catalog.set_tags(dataset_id, ["certified", "production", "marketing"])

# Retrieve tags to verify
tags = client.catalog.get_tags(dataset_id)
print(tags) # ['certified', 'production', 'marketing']
```

Source: medallion.md

Medallion Architecture

The Medallion Architecture is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally improving the quality and structure of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold).

Bronze Layer (Raw)

The **Bronze** layer is where we land all the data from external source systems. The table structures in this layer correspond to the source system table structures "as-is," along with any additional metadata columns that capture the load date/time, process ID, etc.

Characteristics

Raw Data: Data is ingested in its native format.

Append-Only: New data is appended; history is preserved.

No Validation: Minimal to no data validation is performed.

Example: Ingesting Raw Logs

```
from dremioframe.client import DremioClient
client = DremioClient()

# Ingest raw JSON logs from an S3 source into the Bronze layer
# We use CTAS to create a table from the raw files
client.sql("""
    CREATE TABLE "bronze"."app_logs"
    AS SELECT
        *,
        CURRENT_TIMESTAMP as _ingestion_time
    FROM "s3_source"."bucket"."raw_logs"
""")
```

Silver Layer (Cleaned & Conformed)

In the **Silver** layer of the lakehouse, the data from the Bronze layer is matched, merged, conformed and cleaned ("just-enough") so that the Silver layer can provide an "Enterprise view" of all its key business entities, concepts and transactions.

Characteristics

Cleaned: Nulls handled, types cast, formatting standardized.

Deduplicated: Duplicate records are removed.

Enriched: Data may be joined with reference data.

Example: Cleaning and Deduplicating

```
from dremioframe import F

# Read from Bronze
df_bronze = client.table("bronze.app_logs")
```

```
# Transformation Logic
df_silver = df_bronze \
    .filter("user_id IS NOT NULL") \
    .select(
        "user_id",
        F.to_timestamp("event_time").alias("event_time"),
        F.lower("event_type").alias("event_type"),
        "metadata" # Keeping JSON struct
    ) \
    .drop_duplicates(["user_id", "event_time"])

# Materialize to Silver
df_silver.create("silver.app_events")
```

Gold Layer (Curated)

Data in the **Gold** layer of the lakehouse is typically organized in consumption-ready "project-specific" databases. The Gold layer is for reporting and uses more de-normalized and read-optimized data models with fewer joins.

Characteristics

Aggregated: Business-level aggregates (e.g., Daily Active Users).

Dimensional: Star schemas (Fact and Dimension tables).

Business Logic: Complex business rules applied.

Example: Daily Active Users (DAU)

```
# Read from Silver
df_silver = client.table("silver.app_events")

# Calculate DAU
df_gold = df_silver \
    .group_by(F.to_date("event_time").alias("date")) \
    .agg(
        dau=F.count("user_id"),
        events_count=F.count("*")
    )

# Materialize to Gold
df_gold.create("gold.daily_active_users")
```

Folder Structure

A common pattern in Dremio is to use Spaces or Folders to represent these layers:

Space: `Bronze` (or `Raw`)

Space: `Silver` (or `Staging`)

Space: `Gold` (or `Curated`)

You can manage these spaces programmatically:

```
# Create spaces if they don't exist
for layer in ["Bronze", "Silver", "Gold"]:
    try:
        client.catalog.create_space(layer)
    except:
        pass # Already exists
```

Source: scd.md

Slowly Changing Dimensions (SCD)

Slowly Changing Dimensions (SCD) are techniques used in data warehousing to manage how data changes over time. DremioFrame provides support for the two most common types: Type 1 and Type 2.

Type 1 (Overwrite)

SCD Type 1 overwrites the old data with the new data. No history is kept. This is useful for correcting errors or when historical values are not significant (e.g., correcting a spelling mistake in a name).

Implementation

Use the `merge` method to perform an upsert (Update if exists, Insert if new).

```
# Source data contains the latest state of users
new_user_data = client.table("staging.users")

# Target dimension table
target_table = "gold.dim_users"

# Perform Merge
client.table(target_table).merge(
    target_table=target_table,
    on="user_id",
    matched_update={
        "email": "source.email",
        "status": "source.status",
        "updated_at": "CURRENT_TIMESTAMP"
    },
    not_matched_insert={
        "user_id": "source.user_id",
        "email": "source.email",
        "status": "source.status",
```

```
        "created_at": "CURRENT_TIMESTAMP",
        "updated_at": "CURRENT_TIMESTAMP"
    },
    data=new_user_data
)
```

Type 2 (History)

SCD Type 2 tracks historical data by creating multiple records for a given natural key, each representing a specific time range. This allows you to query the state of a record at any point in the past.

Table Design

To use SCD2, your target table must be designed with two special columns to track the validity period of each record:

`valid_from` (TIMESTAMP): The time when the record became active.

`valid_to` (TIMESTAMP): The time when the record ceased to be active. `NULL` indicates the current active record.

Using the Helper

DremioFrame provides a `scd2` helper method to automate the complex logic of closing old records and inserting new ones.

```
from dremioframe.client import DremioClient

client = DremioClient(...)

# Define your source (e.g., a staging table or view)
source = client.table("staging.customers")

# Apply SCD2 logic to the target dimension table
# This generates and executes the necessary SQL statements
source.scd2(
    target_table="warehouse.dim_customers",
    on=["id"],                      # Natural key(s) to join on
    track_cols=["name", "status"],  # Columns to check for changes
    valid_from_col="valid_from",    # Name of your valid_from column
    valid_to_col="valid_to"         # Name of your valid_to column
)
```

Before and After Example

Initial State (Target Table)

id	name	status	valid_from	valid_to
1	Alice	Active	2023-01-01 10:00:00	NULL
2	Bob	Active	2023-01-01 10:00:00	NULL

Source Data (New Batch)

id	name	status
1	Alice	Inactive	< Changed Status
2	Bob	Active	< No Change
3	Charlie	Active	< New Record

After `scd2` Execution

id	name	status	valid_from	valid_to	Note
1	Alice	Active	2023-01-01 10:00:00	**2023-01-02 12:00:00**	Closed (Old Version)
1	Alice	Inactive	**2023-01-02 12:00:00**	NULL	**New Version**
2	Bob	Active	2023-01-01 10:00:00	NULL	Unchanged
3	Charlie	Active	**2023-01-02 12:00:00**	NULL	**New Record**

Logic Breakdown

Identify Changes: The method joins the source and target on `id`. It compares `name` and `status`.

Update: For ID 1, `status` changed. The old record (where `valid_to` is NULL) is updated with `valid_to = NOW()`.

Insert:

ID 1 (New Version): Inserted with `valid_from = NOW()`, `valid_to = NULL`.

ID 3 (New Record): Inserted with `valid_from = NOW()`, `valid_to = NULL`.

ID 2: Ignored because it matched and no columns changed.

Source: views.md

Creating Semantic Views

The Semantic Layer is the interface between your physical data (tables, files) and your business users (BI tools, analysts). In Dremio, this is implemented using **Virtual Datasets (Views)**.

Creating Views

Use `catalog.create_view` to define business logic programmatically.

Example: Customer 360 View

You can pass a SQL string or a `DremioBuilder` object.

```
# Option 1: Using SQL String
client.catalog.create_view(
    path=["marketing", "customer_360"],
    sql="SELECT * FROM source.table"
)

# Option 2: Using DremioBuilder (DataFrame API)
# Define logic using the DataFrame API
df = client.table("gold.dim_users").alias("u") \
        .join(client.table("gold.fact_orders").alias("o"),  on="u.user_id  =  o.user_id",
how="left") \
    .group_by("u.user_id", "u.email") \
    .agg(total_orders=F.count("o.order_id"))

# Create view from the dataframe definition
client.catalog.create_view(
    path=["marketing", "customer_360_v2"],
    sql=df
)
```

Best Practices

1. Business-Friendly Naming

- Use clear, descriptive names for views and columns.
- Avoid technical jargon or abbreviations (e.g., use `customer_id` instead of `c_id`).
- Rename columns to match business terminology.

2. Pre-Calculate Metrics

- Aggregate common metrics (e.g., `total_sales`, `avg_order_value`) in the view to ensure consistency across all BI tools.
- Encapsulate complex logic (e.g., "Active User" definition) within the view.

3. Star Schema Abstraction

- Join Fact and Dimension tables in the view so users don't have to perform complex joins themselves.
- Present a "wide" table that is easy to filter and group.

4. Security

Use Row-Level and Column-Level permissions (if available) or create specific views for different user groups to restrict access to sensitive data.

Version Control

Since Views are defined by SQL, you can version control their definitions in Git.

Store the SQL definition in a `.sql` file in your repo.

Use a CI/CD pipeline (using DremioFrame) to deploy the view when the SQL file changes.

```
# CI/CD Script Example
import glob

# Deploy all views in the 'views/marketing' directory
for sql_file in glob.glob("views/marketing/*.sql"):
    view_name = sql_file.split("/")[-1].replace(".sql", "")
    with open(sql_file, "r") as f:
        sql = f.read()

    print(f"Deploying {view_name}...")
    client.catalog.create_view(
        path=["marketing", view_name],
        sql=sql
    )
```

Source: backend.md

Orchestration Backend

By default, `dremioframe` pipelines store their state in memory. This means if the process exits, the history of pipeline runs is lost.
To persist pipeline history and enable features like the Web UI, you can use a persistent backend.

SQLite Backend

The `SQLiteBackend` stores pipeline runs and task statuses in a local SQLite database file.

Usage

```
from dremioframe.orchestration import Pipeline
from dremioframe.orchestration.backend import SQLiteBackend

# Initialize backend
backend = SQLiteBackend(db_path="pipeline_history.db")
```

```
# Pass backend to Pipeline
pipeline = Pipeline("my_pipeline", backend=backend)

pipeline.run()
```

Custom Backends

You can implement your own backend (e.g., Postgres, Redis, S3) by extending `BaseBackend`.

The `PipelineRun` Object

Your backend will need to store and retrieve `PipelineRun` objects.

```
@dataclass
class PipelineRun:
    pipeline_name: str
    run_id: str
    start_time: float
    status: str          # "RUNNING", "SUCCESS", "FAILED"
    end_time: float      # Optional
    tasks: Dict[str, str] # Map of task_name -> status
```

Required Methods

You must implement the following 4 methods:

`save_run(self, run: PipelineRun)`:

Called when a pipeline starts and finishes.

Should upsert the run record in your storage.

`get_run(self, run_id: str) -> Optional[PipelineRun]`:

Called to retrieve a specific run.

Return `None` if not found.

`update_task_status(self, run_id: str, task_name: str, status: str)`:

Called every time a task changes state (RUNNING, SUCCESS, FAILED, SKIPPED).

Must be efficient and thread-safe if possible.

`list_runs(self, pipeline_name: str = None, limit: int = 10) -> List[PipelineRun]`:

Called by the UI to show history.

Should return the most recent runs, optionally filtered by pipeline name.

Example Implementation Skeleton

```
from dremioframe.orchestration.backend import BaseBackend, PipelineRun
from typing import List, Optional

class MyRedisBackend(BaseBackend):
    def __init__(self, redis_client):
        self.redis = redis_client

    def save_run(self, run: PipelineRun):
        # Serialize run to JSON and save to Redis key `run:{run.run_id}`
        pass

    def get_run(self, run_id: str) -> Optional[PipelineRun]:
        # Get JSON from Redis and deserialize to PipelineRun
        pass

    def update_task_status(self, run_id: str, task_name: str, status: str):
        # Update the specific field in the stored JSON or Hash
        pass

          def  list_runs(self,  pipeline_name:  str  =  None,  limit:  int  =  10)  ->
List[PipelineRun]:
        # Scan keys or use a sorted set for time-based retrieval
        pass
```

Postgres Backend

The `PostgresBackend` stores pipeline state in a PostgreSQL database.

Requirements

```
pip install "dremioframe[postgres]"
```

Usage

```
from dremioframe.orchestration.backend import PostgresBackend

# Uses DREMIOFRAME_PG_DSN env var if dsn not provided
backend = PostgresBackend(dsn="postgresql://user:password@localhost:5432/mydb")
pipeline = Pipeline("my_pipeline", backend=backend)
```

MySQL Backend

The `MySQLBackend` stores pipeline state in a MySQL database.

Requirements

```
pip install "dremioframe[mysql]"
```

Usage

```
from dremioframe.orchestration.backend import MySQLBackend

# Uses DREMIOFRAME_MYSQL_* env vars if config not provided
backend = MySQLBackend(config={
    "user": "myuser",
    "password": "mypassword",
    "host": "localhost",
    "database": "mydb"
})
pipeline = Pipeline("my_pipeline", backend=backend)
```

Source: best_practices.md

Orchestration Best Practices

This guide provides recommendations and patterns for building robust data pipelines using `dremioframe.orchestration`.

1. Organizing Your Tasks

Use the `@task` Decorator

The decorator syntax is cleaner and keeps your code readable.

```
from dremioframe.orchestration import task

@task(name="extract_data")
def extract():
    ...
```

Keep Tasks Atomic

Each task should do one thing well. This makes debugging easier and allows for better retry granularity.

Bad:

```
@task(name="do_everything")
def run():
    # Extract
```

```
    # Transform
    # Load
    # Email
```

Good:

```
@task(name="extract")
def extract(): ...

@task(name="transform")
def transform(): ...

@task(name="load")
def load(): ...
```

2. Managing Dependencies

Linear Chains

For simple sequences, chain the calls:

```
t1.set_downstream(t2).set_downstream(t3)
```

Fan-Out / Fan-In

Run multiple tasks in parallel and then aggregate results.

```
extract_users = extract("users")
extract_orders = extract("orders")
extract_products = extract("products")

consolidate = consolidate_data()

# Fan-in
extract_users.set_downstream(consolidate)
extract_orders.set_downstream(consolidate)
extract_products.set_downstream(consolidate)
```

3. Handling Failures

Use Retries for Transient Errors

Network blips happen. Always add retries to tasks that interact with external systems (Dremio, S3, APIs).

```
@task(name="query_dremio", retries=3, retry_delay=2.0)
```

```
def query():
    ...
```

Use Branching for Alerts

Don't let a failure go unnoticed. Use the `one_failed` trigger rule to send notifications.

```
@task(name="alert_slack", trigger_rule="one_failed")
def alert(context=None):
    # Send message to Slack
    pass

critical_task.set_downstream(alert)
```

Use `all_done` for Cleanup

Ensure temporary resources are cleaned up even if the pipeline fails.

```
@task(name="cleanup_tmp", trigger_rule="all_done")
def cleanup():
    # Delete tmp files
    pass
```

4. Data Passing (Context)

Return Small Metadata, Not Big Data

Do not pass large DataFrames between tasks via return values. The context is kept in memory.
Instead, pass **references** (e.g., table names, S3 paths, file paths).

Bad:

```
@task
def get_data():
    return huge_dataframe # Don't do this
```

Good:

```
@task
def get_data():
    df = ...
    df.to_parquet("s3://bucket/data.parquet")
    return "s3://bucket/data.parquet"

@task
def process(context=None):
```

```
    path = context.get("get_data")
    # Load from path
```

5. Project Structure

Organize your pipelines into a dedicated directory.

```
my_project/
├── pipelines/
│   ├── __init__.py
│   ├── daily_etl.py
│   └── weekly_report.py
├── tasks/
│   ├── __init__.py
│   ├── common.py
│   └── dremio_tasks.py
└── main.py
```

6. Testing

Write unit tests for your tasks by calling the underlying functions directly (if possible) or checking the Task object.
Use `dremioframe`'s testing utilities to mock Dremio responses.

Source: cli.md

Orchestration CLI

DremioFrame provides a CLI to manage your pipelines and orchestration server.

Installation

The CLI is installed with `dremioframe`.

```
dremio-cli --help
```

Pipeline Commands

List Pipelines (Runs)

List recent pipeline runs from the backend.

```
# Default (SQLite)
dremio-cli pipeline list
```

```
# Custom SQLite path
dremio-cli pipeline list --backend-url sqlite:///path/to/db.sqlite

# Postgres
dremio-cli pipeline list --backend-url postgresql://user:pass@host/db
```

Start UI

Start the Orchestration Web UI.

```
dremio-cli pipeline ui --port 8080 --backend-url sqlite:///dremioframe.db
```

Environment Variables

You can also configure the backend via environment variables if supported by the specific backend class, but the CLI currently relies on the `backend-url` for instantiation logic.

Source: deployment.md

Deployment Guide

DremioFrame Orchestration is designed to be easily deployed using Docker.

Docker Deployment

We provide a `Dockerfile` and `docker-compose.yml` to get you started quickly with a full stack including:

- **Orchestrator**: Runs the Web UI and Scheduler.
- **Worker**: Runs Celery workers for distributed tasks.
- **Postgres**: Persistent backend for pipeline history.
- **Redis**: Message broker for Celery.

Prerequisites

- Docker and Docker Compose installed.

Quick Start

1. **Configure Environment**:
   Create a `.env` file with your Dremio credentials:

```
DREMIO_PAT=your_pat
```

```
DREMIO_PROJECT_ID=your_project_id
```

3. **Start Services**:

```
docker-compose up -d
```

4. **Access UI**:
Open `http://localhost:8080` in your browser.

Customizing the Image

If you need additional Python packages (e.g. for custom tasks), you can extend the Dockerfile:

```
FROM dremioframe:latest
RUN pip install pandas numpy
```

Production Considerations

- **Security**: Enable Basic Auth by setting `USERNAME` and `PASSWORD` env vars (requires updating entrypoint script) or putting the UI behind a reverse proxy (Nginx/Traefik).
- **Database**: Use a managed Postgres instance (RDS/CloudSQL) instead of the containerized one for production data safety.
- **Scaling**: Scale workers using Docker Compose:

```
docker-compose up -d --scale worker=3
```

Source: distributed.md

Distributed Execution

DremioFrame Orchestration supports distributed task execution using **Celery**. This allows you to scale your pipelines across multiple worker nodes.

Executors

The `Pipeline` class now accepts an `executor` argument.

LocalExecutor (Default)

Executes tasks locally using a thread pool.

```
from dremioframe.orchestration import Pipeline
```

```
from dremioframe.orchestration.executors import LocalExecutor

# Default behavior (uses LocalExecutor with 1 worker)
pipeline = Pipeline("my_pipeline")

# Explicitly configure LocalExecutor
executor = LocalExecutor(backend=backend, max_workers=4)
pipeline = Pipeline("my_pipeline", executor=executor)
```

CeleryExecutor

Executes tasks on a Celery cluster. This requires a message broker (like Redis or RabbitMQ).

Requirements

```
pip install "dremioframe[celery]"
```

Configuration

1. **Start a Redis Server** (or other broker).
2. **Start a Celery Worker**:
   You need a worker process that can import `dremioframe` and your task code.

   Create a `worker.py`:

```
from celery import Celery

# Configure the app to match the executor's settings
app = Celery("dremioframe_orchestration", broker="redis://localhost:6379/0")
app.conf.update(
    result_backend="redis://localhost:6379/0",
    task_serializer="json",
    result_serializer="json",
    accept_content=["json"],
    imports=["dremioframe.orchestration.executors"] # Important!
)
```

   Run the worker:

```
celery -A worker worker --loglevel=info
```

3. **Configure the Pipeline**:

```
from dremioframe.orchestration import Pipeline
from dremioframe.orchestration.executors import CeleryExecutor
```

```
executor = CeleryExecutor(backend=backend, broker_url="redis://localhost:6379/0")
pipeline = Pipeline("my_pipeline", executor=executor)

pipeline.run()
```

Task Serialization

The `CeleryExecutor` uses `pickle` to serialize your task objects and their actions. **Important**: Ensure your task actions are top-level functions or importable callables. Lambdas and nested functions may fail to pickle.

Source: dq_task.md

Data Quality Task

The `DataQualityTask` integrates the Data Quality Framework into your orchestration pipelines. It allows you to run a suite of DQ checks as a step in your DAG. If any check fails, the task fails, halting the pipeline (unless handled).

Usage

```
from dremioframe.orchestration import Pipeline
from dremioframe.orchestration.tasks.dq_task import DataQualityTask
from dremioframe.client import DremioClient

client = DremioClient()
pipeline = Pipeline("dq_pipeline")

# Run checks from a directory
dq_task = DataQualityTask(
    name="run_sales_checks",
    client=client,
    directory="tests/dq"
)

pipeline.add_task(dq_task)
pipeline.run()
```

Arguments

Argument	Type	Description
`name`	`str`	Name of the task.
`client`	`DremioClient`	Authenticated Dremio client.
`directory`	`str`	Path to a directory containing YAML test files.
`tests`	`list`	List of test dictionaries (alternative to directory).

Behavior

Success: If all checks pass, the task completes successfully.

Failure: If any check fails, the task raises a `RuntimeError`, marking the task as failed.

Source: dremio_jobs.md

Dremio Job Integration

`dremioframe` provides specialized tasks for interacting with Dremio Jobs.

DremioQueryTask

The `DremioQueryTask` submits a SQL query to Dremio, waits for its completion, and supports cancellation.

Features

Job Tracking: Tracks the Dremio Job ID.

Cancellation: If the pipeline is killed or the task is cancelled, it attempts to cancel the running Dremio Job.

Polling: Efficiently polls for job status.

Usage

```
from dremioframe.orchestration import DremioQueryTask, Pipeline
from dremioframe.client import DremioClient

client = DremioClient(...)

# Create a task
t1 = DremioQueryTask(
    name="run_heavy_query",
    client=client,
    sql="SELECT * FROM my_heavy_table"
)

pipeline = Pipeline("dremio_pipeline")
pipeline.add_task(t1)
pipeline.run()
```

Source: extensions.md

Orchestration Extensions

DremioFrame includes advanced tasks for orchestration, including dbt integration and sensors.

dbt Task

The `DbtTask` allows you to run dbt commands within your pipeline.

```
from dremioframe.orchestration import Pipeline, DbtTask

pipeline = Pipeline("dbt_pipeline")

dbt_run = DbtTask(
    name="run_models",
    command="run",
    project_dir="/path/to/dbt/project",
    select="my_model+"
)

pipeline.add_task(dbt_run)
pipeline.run()
```

Sensors

Sensors are tasks that wait for a condition to be met before proceeding.

SqlSensor

Polls a SQL query until it returns data (or a specific condition).

```
from dremioframe.orchestration import Pipeline, SqlSensor

pipeline = Pipeline("sensor_pipeline")

# Wait until data arrives in staging table
wait_for_data = SqlSensor(
    name="wait_for_staging",
    client=client,
    sql="SELECT 1 FROM staging_table LIMIT 1",
    poke_interval=60, # Check every 60 seconds
    timeout=3600      # Timeout after 1 hour
)

pipeline.add_task(wait_for_data)
pipeline.run()
```

FileSensor

Checks for the existence of a file in a Dremio source.

```
from dremioframe.orchestration import Pipeline, FileSensor

# Wait for file to appear
wait_for_file = FileSensor(
    name="wait_for_file",
    client=client,
    path="s3_source.bucket.folder",
    poke_interval=60
)

pipeline.add_task(wait_for_file)
pipeline.run()
```

Source: iceberg.md

Iceberg Maintenance Tasks

`dremioframe` simplifies Iceberg table maintenance with pre-built tasks.

OptimizeTask

Runs `OPTIMIZE TABLE` to compact small files.

Arguments

˜name` (str): The unique name of the task.

˜client` (DremioClient): The authenticated Dremio client.

˜table` (str): The full path to the Iceberg table (e.g., `source.folder.table`).

˜rewrite_data_files` (bool, default=True): Whether to include `REWRITE DATA USING BIN_PACK`.

Example

```
from dremioframe.orchestration import OptimizeTask

t_opt = OptimizeTask(
    name="optimize_sales",
    client=client,
    table="my_catalog.sales",
    rewrite_data_files=True
)
```

VacuumTask

Runs `VACUUM TABLE` to remove unused files and expire snapshots.

Arguments

˜name` (str): The unique name of the task.

˜client` (DremioClient): The authenticated Dremio client.

˜table` (str): The full path to the Iceberg table.

˜expire_snapshots` (bool, default=True): Whether to include `EXPIRE SNAPSHOTS`.

˜retain_last` (int, optional): Number of recent snapshots to retain.

˜older_than` (str, optional): Timestamp string (e.g., '2023-01-01 00:00:00') to expire snapshots older than.

Example

```
from dremioframe.orchestration import VacuumTask

t_vac = VacuumTask(
    name="vacuum_sales",
    client=client,
    table="my_catalog.sales",
    expire_snapshots=True,
    retain_last=5,
    older_than="2023-10-01 00:00:00"
)
```

ExpireSnapshotsTask

A specialized wrapper for expiring snapshots.

Arguments

˜name` (str): The unique name of the task.

˜client` (DremioClient): The authenticated Dremio client.

˜table` (str): The full path to the Iceberg table.

˜retain_last` (int, default=5): Number of recent snapshots to retain.

Example

```
from dremioframe.orchestration import ExpireSnapshotsTask

t_exp = ExpireSnapshotsTask(
    name="expire_sales",
    client=client,
```

```
    table="my_catalog.sales",
    retain_last=3
)
```

Source: overview.md

Orchestration

DremioFrame includes a lightweight orchestration engine to define, schedule, and run sequences of tasks (DAGs). This allows you to build reliable data pipelines directly within your Python application.

Core Concepts

Task

A unit of work, typically wrapping a Python function. Tasks can have dependencies, retries, and can pass data to downstream tasks.

Pipeline

A collection of tasks with defined dependencies. The pipeline manages execution, ensuring tasks run in the correct order (topological sort) and handling parallel execution.

Usage

Basic Example

```
from dremioframe.orchestration import Task, Pipeline

def step_1():
    print("Step 1")

def step_2():
    print("Step 2")

t1 = Task("step_1", step_1)
t2 = Task("step_2", step_2)
t1.set_downstream(t2)

pipeline = Pipeline("my_pipeline")
pipeline.add_task(t1).add_task(t2)
pipeline.run()
```

Decorator API

You can use the `@task` decorator for cleaner syntax:

```
from dremioframe.orchestration import task, Pipeline

@task(name="extract", retries=3)
def extract():
    return [1, 2, 3]

@task(name="transform")
def transform(context=None):
    data = context.get("extract")
    return [x * 2 for x in data]

t_extract = extract()
t_transform = transform()
t_extract.set_downstream(t_transform)

pipeline = Pipeline("etl")
pipeline.add_task(t_extract).add_task(t_transform)
pipeline.run()
```

Parallel Execution

Specify `max_workers` in the `Pipeline` constructor to run independent tasks in parallel:

```
pipeline = Pipeline("parallel_etl", max_workers=4)
```

Visualization

You can generate a Mermaid graph of your pipeline:

```
print(pipeline.visualize())
# or save to file
pipeline.visualize("pipeline.mermaid")
```

Scheduling

Use the `schedule_pipeline` helper to run pipelines at fixed intervals:

```
from dremioframe.orchestration import schedule_pipeline

# Run every 60 seconds
schedule_pipeline(pipeline, interval_seconds=60)
```

Branching & Trigger Rules

You can control when a task runs based on the status of its upstream tasks using `trigger_rule`.

Available rules:

`all_success` (Default): Runs only if all parents succeeded.

`one_failed`: Runs if at least one parent failed. Useful for error handling/notifications.

`all_done`: Runs regardless of parent status (Success, Failed, Skipped). Useful for cleanup.

Example

```
@task(name="process_data")
def process():
    # ...
    pass

@task(name="send_alert", trigger_rule="one_failed")
def alert():
    print("Something went wrong!")

t_proc = process()
t_alert = alert()

t_proc.set_downstream(t_alert)
```

Source: reflections.md

Reflection Management

`dremioframe` simplifies managing Dremio Reflections.

RefreshReflectionTask

Triggers a refresh of all reflections on a specific dataset.

Arguments

`name` (str): The unique name of the task.

`client` (DremioClient): The authenticated Dremio client.

`dataset` (str): The full path to the dataset (e.g., `source.folder.dataset`).

Example

```
from dremioframe.orchestration import RefreshReflectionTask

t_refresh = RefreshReflectionTask(
    name="refresh_sales_reflections",
    client=client,
    dataset="my_catalog.sales"
)
```

This task executes:

```
ALTER DATASET my_catalog.sales REFRESH REFLECTIONS
```

Source: scheduling.md

Orchestration Scheduling

You can also schedule by a simple interval in seconds.

```
# Run every 60 seconds
schedule_pipeline(pipeline, interval_seconds=60)
```

Source: tasks.md

Orchestration Tasks

DremioFrame provides a set of general-purpose tasks to extend your pipelines beyond Dremio operations.

General Tasks

Import these from `dremioframe.orchestration.tasks.general`.

HttpTask

Performs HTTP requests. Useful for triggering webhooks or fetching external data.

```
from dremioframe.orchestration.tasks.general import HttpTask

task = HttpTask(
    name="trigger_webhook",
    url="https://api.example.com/webhook",
    method="POST",
    json_data={"status": "pipeline_started"}
)
```

EmailTask

Sends emails via SMTP. Useful for notifications.

```
from dremioframe.orchestration.tasks.general import EmailTask

task = EmailTask(
    name="send_alert",
    subject="Pipeline Failed",
    body="The pipeline encountered an error.",
    to_addr="admin@example.com",
    smtp_server="smtp.example.com",
    smtp_port=587,
    use_tls=True,
    username="user",
    password="password"
)
```

ShellTask

Executes arbitrary shell commands.

```
from dremioframe.orchestration.tasks.general import ShellTask

task = ShellTask(
    name="run_dbt",
    command="dbt run",
    cwd="/path/to/dbt/project",
    env={"DBT_PROFILES_DIR": "."}
)
```

S3Task

Interacts with AWS S3. Requires `boto3`.

Requirements:

```
pip install "dremioframe[s3]"
```

Usage:

```
from dremioframe.orchestration.tasks.general import S3Task

# Upload
upload = S3Task(
    name="upload_report",
    operation="upload_file",
    bucket="my-bucket",
    key="reports/daily.csv",
    local_path="/tmp/daily.csv"
```

```
)

# Download
download = S3Task(
    name="download_config",
    operation="download_file",
    bucket="my-bucket",
    key="config/settings.json",
    local_path="/app/settings.json"
)
```

Source: ui.md

Orchestration Web UI

`dremioframe` includes a lightweight Web UI to visualize pipeline runs and task statuses.

Features

Dashboard: View all pipelines and their recent runs.

Real-time Updates: Auto-refreshing status of tasks and runs.

Manual Trigger: Trigger pipeline runs directly from the UI.

Task Status: Visual indicators for task success, failure, and skipping.

Starting the UI

You can start the UI from your Python script:

```
from dremioframe.orchestration import start_ui, Pipeline
from dremioframe.orchestration.backend import SQLiteBackend

# Setup backend and pipelines
backend = SQLiteBackend("history.db")
pipeline1 = Pipeline("etl_job", backend=backend)
pipeline2 = Pipeline("maintenance", backend=backend)

# Start UI
# Pass the pipelines dict to enable manual triggering
start_ui(backend=backend,  pipelines={"etl_job":  pipeline1,  "maintenance":  pipeline2},
port=8080)
```

Visit `http://localhost:8080` in your browser.

Security

The UI supports Basic Authentication.

Enabling Authentication

Pass `username` and `password` to `start_ui` or via the CLI.

```
from dremioframe.orchestration.ui import start_ui

start_ui(backend, port=8080, username="admin", password="secret_password")
```

CLI Usage

(CLI support for auth args is pending, currently only via python script or hardcoded in custom entrypoint)
Note: The `dremio-cli pipeline ui` command does not yet expose auth flags, but you can wrap `start_ui` in your own script.

Architecture

The UI is a Single Page Application (SPA) built with **Vue.js** (loaded via CDN). It communicates with the Python backend via a simple REST API:

˜GET /api/runs`: List recent pipeline runs.

˜GET /api/pipelines`: List available pipelines.

˜POST /api/pipelines/{name}/trigger`: Trigger a new run.
nd
backend = SQLiteBackend("pipeline.db")

2. Start UI Server (in a separate thread or process)

Note: In production, you might run this as a separate script.

ui_thread = threading.Thread(target=start_ui, args=(backend, 8080))
ui_thread.daemon = True
ui_thread.start()

3. Run Pipeline

pipeline = Pipeline("my_pipeline", backend=backend)

... add tasks ...

pipeline.run()

```
Access the UI at `http://localhost:8080`.


---

# Source: bulk_loading.md

# Bulk Loading Optimization

For large datasets (10,000+ rows), using the default `VALUES` clause method can be slow and may hit SQL statement size limits. DremioFrame provides a **staging method** that dramatically improves performance by using Parquet files as an intermediate format.

## Usage

Both `create()` and `insert()` methods support a `method` parameter:

```python
from dremioframe.client import DremioClient
import pandas as pd

client = DremioClient()

# Create large dataset
data = pd.DataFrame({
    "id": range(100000),
    "name": [f"user_{i}" for i in range(100000)],
    "value": range(100000)
})

# Use staging method for fast bulk load
client.table('"my_space"."my_folder"."large_table"').create(
    '"my_space"."my_folder"."large_table"',
    data=data,
    method="staging"  # Much faster than default "values"
)
```

How It Works

Values Method (Default)

```
method="values"  # Default
```

- Generates SQL `INSERT INTO ... VALUES (...)` statements
- Good for small datasets (< 10,000 rows)
- Simple and straightforward
-
```

Can hit SQL statement size limits with large data

Staging Method (Recommended for Large Data)

```
method="staging"
```

For `create()`:

- Writes data to a temporary local Parquet file
- Uploads the Parquet file to Dremio (creates the table)
- Cleans up the temporary file

For `insert()`:

- Writes data to a temporary local Parquet file
- Uploads to a temporary staging table in Dremio
- Executes `INSERT INTO target SELECT * FROM staging_table`
- Drops the staging table
- Cleans up the temporary file

Performance Comparison

Rows	Values Method	Staging Method	Speedup
1,000	~2s	~3s	0.67x
10,000	~20s	~5s	4x
100,000	Fails*	~15s	∞

*SQL statement size limit exceeded

When to Use Staging

Use `method="staging"` when:

- Loading more than 10,000 rows
- Experiencing slow `INSERT` performance
- Hitting SQL statement size limits
- Working with wide tables (many columns)

Use `method="values"` (default) when:

- Loading small datasets (< 1,000 rows)
- Simplicity is preferred over performance

-

You don't have write access to create temporary tables

Source: connection_pooling.md

Connection Pooling

DremioFrame includes a `ConnectionPool` to manage and reuse `DremioClient` instances, which is essential for high-concurrency applications or long-running services.

ConnectionPool

The `ConnectionPool` manages a thread-safe queue of clients.

Initialization

```
from dremioframe.connection_pool import ConnectionPool

# Create a pool with 5 connections
pool = ConnectionPool(
    max_size=5,
    timeout=30,
    # DremioClient arguments
    pat="YOUR_PAT",
    project_id="YOUR_PROJECT_ID"
)
```

Using the Context Manager

The recommended way to use the pool is via the context manager, which ensures connections are returned to the pool even if errors occur.

```
with pool.client() as client:
    # Use client as normal
    df = client.sql("SELECT * FROM sys.version").collect()
    print(df)
```

Manual Management

You can also manually get and release clients.

```
try:
    client = pool.get_client()
    # Use client...
finally:
    pool.release_client(client)
```

Configuration

- **max_size**: Maximum number of connections to create.
- **timeout**: Seconds to wait for a connection if the pool is empty and at max size. Raises `TimeoutError` if exceeded.
- **client_kwargs**: Arguments passed to `DremioClient` constructor (e.g., `pat`, `username`, `password`, `flight_endpoint`).

Source: cost_estimation.md

Query Cost Estimation

DremioFrame provides a `CostEstimator` to analyze query execution plans, estimate costs, and suggest optimizations before running expensive queries.

CostEstimator

The `CostEstimator` uses Dremio's `EXPLAIN PLAN` to analyze queries and provide actionable insights.

Initialization

```
from dremioframe.client import DremioClient
from dremioframe.cost_estimator import CostEstimator

client = DremioClient()
estimator = CostEstimator(client)
```

Estimating Query Cost

Get a detailed cost estimate for any query:

```
sql = """
SELECT customer_id, SUM(amount) as total
FROM sales.transactions
WHERE date >= '2024-01-01'
GROUP BY customer_id
"""

estimate = estimator.estimate_query_cost(sql)

print(f"Estimated rows: {estimate.estimated_rows}")
print(f"Total cost: {estimate.total_cost}")
print(f"Plan summary: {estimate.plan_summary}")
print(f"Optimization hints: {estimate.optimization_hints}")
```

Output:

```
Estimated rows: 50000
Total cost: 125.5
Plan summary: Query plan includes: 1 table scan(s), aggregation, 1 filter(s)
Optimization hints: ['Consider adding LIMIT for large tables']
```

Cost Estimate Details

The `CostEstimate` object includes:

- **estimated_rows**: Number of rows expected to be processed
- **estimated_bytes**: Approximate data size
- **scan_cost**: Cost of table scans
- **join_cost**: Cost of join operations
- **total_cost**: Overall query cost metric
- **plan_summary**: Human-readable plan description
- **optimization_hints**: List of suggestions

Optimization Hints

The estimator automatically detects common anti-patterns:

```
hints = estimator.get_optimization_hints(sql)

for hint in hints:
    print(f"{hint.severity}: {hint.message}")
    print(f"  Suggestion: {hint.suggestion}")
```

Common hints:

- **SELECT ***: Suggests specifying only needed columns
- **Missing WHERE**: Warns about full table scans
- **Multiple JOINs**: Suggests using CTEs for readability
- **ORDER BY without LIMIT**: Recommends adding LIMIT
- **DISTINCT usage**: Suggests alternatives like GROUP BY

Comparing Query Variations

Compare multiple approaches to find the most efficient:

```
result = estimator.compare_queries(
    # Approach 1: Subquery
```

```
    """
    SELECT * FROM (
        SELECT customer_id, amount FROM sales.transactions
    ) WHERE amount > 1000
    """,

    # Approach 2: Direct filter
    """
    SELECT customer_id, amount
    FROM sales.transactions
    WHERE amount > 1000
    """
)

print(result['recommendation'])
# Output: "Query 2 has the lowest estimated cost (45.2)"

for query in result['queries']:
    print(f"Query {query['query_id']}: Cost = {query['total_cost']}")
```

Use Cases

1. Pre-execution Validation

```
# Check cost before running expensive query
estimate = estimator.estimate_query_cost(expensive_sql)

if estimate.total_cost > 1000:
    print("Warning: This query may be expensive!")
    print("Hints:", estimate.optimization_hints)
    # Decide whether to proceed
```

2. Query Optimization Workflow

```
# Iteratively improve query
queries = [
    "SELECT * FROM large_table",
    "SELECT id, name FROM large_table",
    "SELECT id, name FROM large_table WHERE active = true"
]

comparison = estimator.compare_queries(*queries)
print(f"Best approach: Query {comparison['best_query_id']}")
```

3. Automated Query Review

```
# Review all queries in a pipeline
```

```
for sql in pipeline_queries:
    estimate = estimator.estimate_query_cost(sql)
    if len(estimate.optimization_hints) > 0:
        print(f"Query needs review: {sql[:50]}...")
        for hint in estimate.optimization_hints:
            print(f"  - {hint}")
```

Limitations

- **Cost Metrics**: Costs are relative estimates, not absolute resource usage
- **Plan Parsing**: Based on Dremio's EXPLAIN output format (may vary by version)
- **Optimization Hints**: Static analysis; may not catch all issues
- **Reflection Impact**: Doesn't account for reflection acceleration

Best Practices

- **Use Early**: Check costs during development, not just production
- **Iterate**: Use `compare_queries()` to test different approaches
- **Combine with Profiling**: Use cost estimation for planning, profiling for actual performance
- **Set Thresholds**: Define acceptable cost limits for your use case

Source: tuning.md

Performance Tuning Guide

Optimizing `dremioframe` applications involves tuning both the client-side Python code and the server-side Dremio execution.

1. Arrow Flight Optimization

DremioFrame uses Apache Arrow Flight for high-performance data transfer.

Batch Sizes

When fetching large datasets using `collect()`, the data is streamed in chunks.

- **Default**: Dremio controls the chunk size.
- **Optimization**: Ensure your network has high throughput. Flight is bandwidth-bound.

When **writing** data (`insert`, `create`), `dremioframe` splits data into batches to avoid hitting message size limits (usually 2GB, but practically smaller).

```
# Default batch size is often safe, but for very wide tables, reduce it.
client.table("target").insert("target", data=df, batch_size=5000)
```

Compression

Flight supports compression (LZ4/ZSTD). DremioFrame negotiates this automatically. Ensure your client machine has CPU cycles to spare for decompression.

2. Client-Side vs. Server-Side Processing

Always push filtering and aggregation to Dremio (Server-Side) before collecting data to Python (Client-Side).

Bad Pattern (Client-Side Filtering):

```
# Fetches ALL data, then filters in Python
df = client.table("sales").collect()
filtered_df = df.filter(pl.col("amount") > 100)
```

Good Pattern (Server-Side Filtering):

```
# Filters in Dremio, fetches only matching rows
df = client.table("sales").filter("amount > 100").collect()
```

3. Parallelism

Pipeline Parallelism

Use the `orchestration` module to run independent tasks in parallel.

```
pipeline = Pipeline("etl", max_workers=4)
```

Async Client

For high-concurrency applications (e.g., a web app backend), use `AsyncDremioClient` to avoid blocking the main thread while waiting for Dremio.

```
async with AsyncDremioClient() as client:
    result = await client.query("SELECT * FROM large_table")
```

4. Caching

If you query the same dataset multiple times in a script, cache it locally.

```
# Cache the result of a heavy query to a local Parquet file
cached_df = client.table("heavy_view").cache("local_cache_name", ttl_seconds=600)

# Subsequent operations use the local file (via DuckDB/DataFusion)
cached_df.filter("col1 = 1").show()
```

Source: advanced.md

Advanced Features

External Queries

Dremio allows you to push queries directly to the underlying source, bypassing Dremio's SQL parser. This is useful for using source-specific SQL dialects or features not yet supported by Dremio.

```
# Run a native Postgres query
df = client.external_query("Postgres", "SELECT * FROM users WHERE active = true")

# You can then chain DremioFrame operations on top
df.filter("age > 21").select("name").show()
```

This generates SQL like:

```
SELECT name FROM (
      SELECT * FROM TABLE(Postgres.EXTERNAL_QUERY('SELECT * FROM users WHERE active =
true'))
) AS sub
WHERE age > 21
```

Source: async_client.md

Async Client

`dremioframe` provides an asynchronous client for high-concurrency applications.

Usage

The `AsyncDremioClient` is designed to be used as an async context manager.

```
import asyncio
from dremioframe.async_client import AsyncDremioClient

async def main():
    async with AsyncDremioClient(pat="my-pat") as client:
        # Get catalog item
```

```
        item = await client.get_catalog_item("dataset-id")
        print(item)

        # Execute SQL (REST API)
        job = await client.execute_sql("SELECT 1")
        print(job)

if __name__ == "__main__":
    asyncio.run(main())
```

Methods

`get_catalog_item(id)`: Get catalog item by ID.

`get_catalog_by_path(path)`: Get catalog item by path list.

`execute_sql(sql)`: Submit a SQL job via REST API.

`get_job_status(job_id)`: Check job status.

Source: builder.md

Builder API Reference

::: dremioframe.builder.DremioBuilder
    options:
      show_root_heading: true
      show_source: true

Source: cli.md

DremioFrame CLI

DremioFrame includes a command-line interface (CLI) for quick interaction with Dremio.

Installation

The CLI is installed automatically with `dremioframe`.

```
pip install dremioframe
```

Interactive Shell (REPL)

DremioFrame provides an interactive shell with syntax highlighting and auto-completion.

```
dremio-cli repl
```

Commands:

˜SELECT ...`: Execute SQL query and display results.

˜tables`: List tables in the root catalog.

˜exit` or `quit`: Exit the shell.

Requires `rich` and `prompt_toolkit` (install with `pip install dremioframe[cli]`).

Configuration

Set the following environment variables:

˜DREMIO_PAT`: Personal Access Token

˜DREMIO_URL`: Dremio URL (e.g., `data.dremio.cloud`)

˜DREMIO_PROJECT_ID`: Project ID (optional, for Cloud)

Usage

Run a Query

```
dremio-cli query "SELECT * FROM my_table LIMIT 5"
```

List Catalog

```
# List root catalog
dremio-cli catalog

# List specific path
dremio-cli catalog --path "source.folder"
```

List Reflections

```
dremio-cli reflections
```

Source: client.md

Client API Reference

::: dremioframe.client.DremioClient
    options:
      show_root_heading: true
      show_source: true

::: dremioframe.client.AsyncDremioClient
    options:
      show_root_heading: true
      show_source: true

Client Helpers

These classes are accessed via properties on the `DremioClient` instance (e.g., `client.admin`, `client.catalog`).

Admin

::: dremioframe.admin.Admin
    options:
      show_root_heading: true
      show_source: true

Catalog

::: dremioframe.catalog.Catalog
    options:
      show_root_heading: true
      show_source: true

Iceberg

::: dremioframe.iceberg.Iceberg
    options:
      show_root_heading: true
      show_source: true

UDF

::: dremioframe.udf.UDF
    options:
      show_root_heading: true
      show_source: true

Profile

::: dremioframe.profile.Profile
    options:
      show_root_heading: true
      show_source: true

Source: dq.md

Data Quality API Reference

::: dremioframe.dq.runner.DQRunner
    options:
      show_root_heading: true

::: dremioframe.dq.checks.DataQuality
    options:
      show_root_heading: true

Source: function_reference.md

Function Reference

This document lists the SQL functions supported by `dremioframe.functions`.

General Functions

`col(name)`: Creates a column expression.

`lit(val)`: Creates a literal expression.

Aggregates

`sum(col)`: Calculates the sum of a column.

`avg(col)`: Calculates the average of a column.

`min(col)`: Finds the minimum value in a column.

`max(col)`: Finds the maximum value in a column.

`count(col)`: Counts the number of non-null values in a column.

`stddev(col)`: Calculates the standard deviation.

`variance(col)`: Calculates the variance.

`approx_distinct(col)`: Approximates the count of distinct values.

Math

`abs(col)`: Absolute value.

`ceil(col)`: Ceiling.

`floor(col)`: Floor.

`round(col, scale=0)`: Rounds to the specified scale.

`sqrt(col)`: Square root.

`exp(col)`: Exponential.

`ln(col)`: Natural logarithm.

˜log(base, col)`: Logarithm with specified base.

˜pow(col, power)`: Power.

String

˜upper(col)`: Converts to uppercase.

˜lower(col)`: Converts to lowercase.

˜concat(*cols)`: Concatenates strings.

˜substr(col, start, length=None)`: Substring.

˜trim(col)`: Trims whitespace from both ends.

˜ltrim(col)`: Trims whitespace from left.

˜rtrim(col)`: Trims whitespace from right.

˜length(col)`: String length.

˜replace(col, pattern, replacement)`: Replaces occurrences of pattern.

˜regexp_replace(col, pattern, replacement)`: Replaces using regex.

˜initcap(col)`: Capitalizes first letter of each word.

Date/Time

˜current_date()`: Current date.

˜current_timestamp()`: Current timestamp.

˜date_add(col, days)`: Adds days to date.

˜date_sub(col, days)`: Subtracts days from date.

˜date_diff(col1, col2)`: Difference in days between dates.

˜to_date(col, fmt=None)`: Converts string to date.

˜to_timestamp(col, fmt=None)`: Converts string to timestamp.

˜year(col)`: Extracts year.

˜month(col)`: Extracts month.

˜day(col)`: Extracts day.

˜hour(col)`: Extracts hour.

˜minute(col)`: Extracts minute.

˜second(col)`: Extracts second.

˜extract(field, source)`: Extracts field from source.

Conditional

˜coalesce(*cols)`: Returns first non-null value.

˜when(condition, value)`: Starts a CASE statement builder.

Window Functions

˜rank()`: Rank.

˜dense_rank()`: Dense rank.

˜row_number()`: Row number.

˜lead(col, offset=1, default=None)`: Lead.

˜lag(col, offset=1, default=None)`: Lag.

˜first_value(col)`: First value in window.

˜last_value(col)`: Last value in window.

˜ntile(n)`: N-tile.

AI Functions

˜ai_classify(prompt, categories, model_name=None)`: Classifies text into categories.

˜ai_complete(prompt, model_name=None)`: Generates text completion.

˜ai_generate(prompt, model_name=None, schema=None)`: Generates structured data.

Complex Types

˜flatten(col)`: Explodes a list into multiple rows.

˜convert_from(col, type_)`: Converts from serialized format.

˜convert_to(col, type_)`: Converts to serialized format.

Source: aggregate.md

Aggregate Functions

Aggregate functions operate on a set of values to compute a single result.

Usage

```
from dremioframe import F

df.group_by("dept").agg(
```

```
    total=F.sum("salary"),
    count=F.count("*")
)
```

Available Functions

Function	Description
`sum(col)`	Returns the sum of values in the column.
`avg(col)`	Returns the average of values in the column.
`min(col)`	Returns the minimum value.
`max(col)`	Returns the maximum value.
`count(col)`	Returns the count of non-null values. Use `*` for total rows.
`stddev(col)`	Returns the sample standard deviation.
`variance(col)`	Returns the sample variance.
`approx_distinct(col)`	Returns the approximate number of distinct values (HyperLogLog).

Source: ai.md

AI Functions

Dremio provides AI-powered functions for classification, text completion, and structured data generation.

Usage

```
from dremioframe import F

# Classify
df.select(
    F.ai_classify(F.col("review"), ["Positive", "Negative"]).alias("sentiment")
)

# Complete
df.select(
    F.ai_complete("Summarize this text: " + F.col("text")).alias("summary")
)

# Generate Structured Data
df.select(
    F.ai_generate(
        "Extract entities",
        schema="ROW(person VARCHAR, location VARCHAR)"
    ).alias("entities")
)
```

Raw SQL Usage

You can also use AI functions by writing the SQL string directly in `mutate` or `select`.

```
df.mutate(
    spice_level="AI_CLASSIFY('Identify the Spice Level:' || ARRAY_TO_STRING(ingredients, ','), ARRAY [ 'mild', 'medium', 'spicy' ])"
)
```

Available Functions

Function	Description
`ai_classify(prompt, categories, model_name=None)`	Classifies text into one of the provided categories.
`ai_complete(prompt, model_name=None)`	Generates a text completion for the prompt.
`ai_generate(prompt, model_name=None, schema=None)`	Generates structured data based on the prompt. Use `schema` to define the output structure (e.g., `ROW(...)`).

Examples

AI_CLASSIFY

```
F.ai_classify("Is this email spam?", ["Spam", "Not Spam"])
F.ai_classify("Categorize product", ["Electronics", "Clothing"], model_name="gpt-4")
```

AI_COMPLETE

```
F.ai_complete("Write a SQL query to find top users")
F.ai_complete("Translate to French", model_name="gpt-3.5")
```

AI_GENERATE

```
# Generate structured data
F.ai_generate(
    "Extract customer info",
    schema="ROW(name VARCHAR, age INT)"
)

# With specific model
F.ai_generate(
    "Extract info",
    model_name="gpt-4",
    schema="ROW(summary VARCHAR)"
```

```
)
```

Using with LIST_FILES

You can combine AI functions with `client.list_files()` to process unstructured data.

```
# Process all text files in a folder
client.list_files("@source/folder") \
    .filter("file_name LIKE '%.txt'") \
    .select(
        F.col("file_name"),
        F.ai_classify("Sentiment?", F.col("file_content"), ["Positive", "Negative"])
    )
```

Source: complex.md

Complex Type Functions

Functions for working with complex types like Arrays, Maps, and Structs.

Usage

```
from dremioframe import F

# Flatten an array
df.select(F.flatten("items"))

# Convert from JSON
df.select(F.convert_from("json_col", "JSON"))

# Convert to JSON
df.select(F.convert_to("map_col", "JSON"))
```

Available Functions

Function	Description
`flatten(col)`	Explodes a list into multiple rows.
`convert_from(col, type)`	Convert from a serialized format (e.g. 'JSON') to a complex type.
`convert_to(col, type)`	Convert a complex type to a serialized format (e.g. 'JSON').

Source: conditional.md

Conditional Functions

Functions for conditional logic.

Usage

```
from dremioframe import F

# Coalesce
df.select(F.coalesce(F.col("phone"), F.col("email"), F.lit("Unknown")))

# Case When
df.select(
    F.when("age < 18", "Minor")
     .when("age < 65", "Adult")
     .otherwise("Senior").alias("age_group")
)
```

Available Functions

Function	Description
`coalesce(*cols)`	Returns the first non-null value.
`when(cond, val).otherwise(val)`	CASE WHEN statement builder.

Source: date.md

Date & Time Functions

Functions for date and time manipulation.

Usage

```
from dremioframe import F

df.select(
    F.year(F.col("date")),
    F.date_add(F.col("date"), 7)
)
```

Available Functions

Function	Description
`current_date()`	Current date.
`current_timestamp()`	Current timestamp.
`date_add(col, days)`	Add days to date.
`date_sub(col, days)`	Subtract days from date.
`date_diff(col1, col2)`	Difference in days between dates.

`to_date(col, fmt)`	Convert string to date.
`to_timestamp(col, fmt)`	Convert string to timestamp.
`year(col)`	Extract year.
`month(col)`	Extract month.
`day(col)`	Extract day.
`hour(col)`	Extract hour.
`minute(col)`	Extract minute.
`second(col)`	Extract second.
`extract(field, source)`	Extract specific field (e.g., 'YEAR' from date).

Source: math.md

Math Functions

Mathematical functions for numeric operations.

Usage

```
from dremioframe import F

df.select(
    F.abs(F.col("diff")),
    F.round(F.col("price"), 2)
)
```

Available Functions

Function	Description
`abs(col)`	Absolute value.
`ceil(col)`	Ceiling (round up).
`floor(col)`	Floor (round down).
`round(col, scale=0)`	Round to specified decimal places.
`sqrt(col)`	Square root.
`exp(col)`	Exponential (e^x).
`ln(col)`	Natural logarithm.
`log(base, col)`	Logarithm with specified base.
`pow(col, power)`	Power (x^y).

Source: string.md

String Functions

Functions for string manipulation.

Usage

```
from dremioframe import F

df.select(
    F.upper(F.col("name")),
    F.concat(F.col("first"), F.lit(" "), F.col("last"))
)
```

Available Functions

Function	Description
`upper(col)`	Convert to uppercase.
`lower(col)`	Convert to lowercase.
`concat(*cols)`	Concatenate strings.
`substr(col, start, length)`	Extract substring.
`trim(col)`	Trim whitespace from both ends.
`ltrim(col)`	Trim whitespace from left.
`rtrim(col)`	Trim whitespace from right.
`length(col)`	Length of string.
`replace(col, pattern, replacement)`	Replace substring.
`regexp_replace(col, pattern, replacement)`	Replace using regex.
`initcap(col)`	Capitalize first letter of each word.

Source: window.md

Window Functions

Window functions operate on a set of rows related to the current row.

Usage

```
from dremioframe import F

window = F.Window.partition_by("dept").order_by("salary")

df.select(
    F.rank().over(window).alias("rank")
)
```

Window Specification

Use `F.Window` to create a specification:

˜partition_by(*cols)`

˜order_by(*cols)`

˜rows_between(start, end)`

`range_between(start, end)`

Available Functions

Function	Description
`rank()`	Rank with gaps.
`dense_rank()`	Rank without gaps.
`row_number()`	Unique row number.
`lead(col, offset, default)`	Value from following row.
`lag(col, offset, default)`	Value from preceding row.
`first_value(col)`	First value in window frame.
`last_value(col)`	Last value in window frame.
`ntile(n)`	Distribute rows into n buckets.

Source: functions.md

SQL Functions API Reference

Helper functions for constructing SQL expressions (e.g., `F.col("a")`, `F.sum("b")`).

::: dremioframe.functions
    options:
      show_root_heading: true
      show_source: true
      members:
        - col
        - lit
        - sum
        - avg
        - min
        - max
        - count
        - row_number
        - rank
        - dense_rank
        - Window

Source: functions_guide.md

SQL Functions

DremioFrame provides a comprehensive set of SQL functions via `dremioframe.functions` (aliased as `F`).

Categories

1. Aggregate Functions

Math Functions

String Functions

Date & Time Functions

Window Functions

Conditional Functions

AI Functions

Complex Type Functions

Usage

You can use functions in two ways:

1. Function Builder (Recommended)

Import `F` and chain methods. This provides autocomplete and type safety.

```
from dremioframe import F

df.select(
    F.col("name"),
    F.upper(F.col("city")),
    F.sum("salary").over(F.Window.partition_by("dept"))
)
```

2. Raw SQL Strings

You can write raw SQL strings directly in `mutate` or `select`. This is useful for complex expressions or functions not yet wrapped in `dremioframe`.

```
# In mutate
df.mutate(
    upper_city="UPPER(city)",
    total_salary="SUM(salary) OVER (PARTITION BY dept)"
)

# In select
df.select(
    "name",
    "UPPER(city) AS upper_city"
)
```

Expressions (`Expr`)

The `Expr` class allows you to build complex SQL expressions using Python operators.

Arithmetic: `+`, `-`, `*`, `/`, `%`

Comparison: `==`, `!=`, `>`, `<`, `>=`, `<=`

Logical: `&` (AND), `|` (OR), `~` (NOT)

Methods:

`alias(name)`: Rename the expression.

`cast(type)`: Cast to a SQL type.

`isin(values)`: Check if value is in a list.

`is_null()`, `is_not_null()`: Check for NULLs.

Source: orchestration.md

Orchestration API Reference

Pipeline

::: dremioframe.orchestration.pipeline.Pipeline
    options:
      show_root_heading: true

Tasks

::: dremioframe.orchestration.task.Task
    options:
      show_root_heading: true

Dremio Tasks

::: dremioframe.orchestration.tasks.dremio_tasks.DremioQueryTask
    options:
      show_root_heading: true

::: dremioframe.orchestration.tasks.builder_task.DremioBuilderTask
    options:
      show_root_heading: true

General Tasks

::: dremioframe.orchestration.tasks.general.HttpTask
    options:
      show_root_heading: true

::: dremioframe.orchestration.tasks.general.EmailTask
    options:

show_root_heading: true

::: dremioframe.orchestration.tasks.general.ShellTask
options:
show_root_heading: true

::: dremioframe.orchestration.tasks.general.S3Task
options:
show_root_heading: true

Extension Tasks

::: dremioframe.orchestration.tasks.dbt_task.DbtTask
options:
show_root_heading: true

::: dremioframe.orchestration.tasks.dq_task.DataQualityTask
options:
show_root_heading: true

Iceberg Tasks

::: dremioframe.orchestration.iceberg_tasks.OptimizeTask
options:
show_root_heading: true

::: dremioframe.orchestration.iceberg_tasks.VacuumTask
options:
show_root_heading: true

::: dremioframe.orchestration.iceberg_tasks.ExpireSnapshotsTask
options:
show_root_heading: true

Reflection Tasks

::: dremioframe.orchestration.reflection_tasks.RefreshReflectionTask
options:
show_root_heading: true

Sensors

::: dremioframe.orchestration.sensors.SqlSensor
options:
show_root_heading: true

::: dremioframe.orchestration.sensors.FileSensor
options:
show_root_heading: true

Executors

::: dremioframe.orchestration.executors.LocalExecutor
    options:
      show_root_heading: true

::: dremioframe.orchestration.executors.CeleryExecutor
    options:
      show_root_heading: true

Scheduling

::: dremioframe.orchestration.scheduling.schedule_pipeline
    options:
      show_root_heading: true

Backends

::: dremioframe.orchestration.backend.BaseBackend
    options:
      show_root_heading: true

::: dremioframe.orchestration.backend.PostgresBackend
    options:
      show_root_heading: true

::: dremioframe.orchestration.backend.MySQLBackend
    options:
      show_root_heading: true

::: dremioframe.orchestration.backend.SQLiteBackend
    options:
      show_root_heading: true

::: dremioframe.orchestration.backend.InMemoryBackend
    options:
      show_root_heading: true

Source: testing.md

Testing Guide

DremioFrame tests are categorized into three groups.

1. Unit & Integration (Dremio Cloud)

These tests cover the core logic and integration with Dremio Cloud. They should always pass.

Requirements:

`DREMIO_PAT`: Personal Access Token for Dremio Cloud.

-

`DREMIO_PROJECT_ID`: Project ID for Dremio Cloud.

˜DREMIO_TEST_SPACE`: Writable space/folder for integration tests (e.g., "Scratch").

Command:

```
# Run all unit tests and cloud integration tests
pytest -m "not software and not external_backend"
```

Note: If credentials are missing, cloud integration tests will skip.

2. Dremio Software

Tests specifically for Dremio Software connectivity.

Requirements:

˜DREMIO_SOFTWARE_HOST`: Hostname (e.g., localhost).

˜DREMIO_SOFTWARE_PORT`: Flight port (default 32010).

˜DREMIO_SOFTWARE_USER`: Username.

˜DREMIO_SOFTWARE_PASSWORD`: Password.

˜DREMIO_SOFTWARE_TLS`: "true" or "false" (default false).

Command:

```
pytest -m software
```

3. External Backends

Tests for persistent orchestration backends (Postgres, MySQL).

Requirements:

Postgres: `DREMIOFRAME_PG_DSN` (e.g., `postgresql://user:pass@localhost/db`)

MySQL:

˜DREMIOFRAME_MYSQL_USER`

˜DREMIOFRAME_MYSQL_PASSWORD`

˜DREMIOFRAME_MYSQL_HOST`

˜DREMIOFRAME_MYSQL_DB`

˜DREMIOFRAME_MYSQL_PORT`

Command:

```
pytest -m external_backend
```

Running All Tests

To run everything (skipping what isn't configured):

```
pytest
```

Source: mocking.md

Mock/Testing Framework

DremioFrame provides a comprehensive testing framework to write tests without requiring a live Dremio connection.

MockDremioClient

The `MockDremioClient` mimics the `DremioClient` interface, allowing you to configure query responses for testing.

Basic Usage

```
from dremioframe.testing import MockDremioClient
import pandas as pd

# Create mock client
client = MockDremioClient()

# Configure a response
users_df = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

client.add_response("SELECT * FROM users", users_df)

# Use in your code
result = client.sql("SELECT * FROM users").collect()
print(result)  # Returns the mocked DataFrame
```

Query Matching

The mock client supports both exact and partial query matching:

```
# Exact match
client.add_response("SELECT * FROM users", users_df)

# Partial match (matches any query containing "FROM users")
client.add_response("FROM users", users_df)
```

```
# This will match:
client.sql("SELECT id FROM users WHERE age > 25").collect()
```

Query History

Track which queries were executed:

```
client.sql("SELECT * FROM table1")
client.sql("SELECT * FROM table2")

# Check history
print(client.query_history)  # ['SELECT * FROM table1', 'SELECT * FROM table2']
print(client.get_last_query())  # 'SELECT * FROM table2'

# Clear history
client.clear_history()
```

FixtureManager

Manage test data fixtures for consistent, reusable test datasets.

Creating Fixtures

```
from dremioframe.testing import FixtureManager

manager = FixtureManager()

# Create from data
test_data = [
    {'product_id': 1, 'name': 'Widget', 'price': 9.99},
    {'product_id': 2, 'name': 'Gadget', 'price': 19.99}
]

df = manager.create_fixture('products', test_data)
```

Loading from Files

```
# Load CSV
df = manager.load_csv('customers', 'tests/fixtures/customers.csv')

# Load JSON
df = manager.load_json('orders', 'tests/fixtures/orders.json')

# Retrieve loaded fixture
customers = manager.get('customers')
```

Saving Fixtures

```
# Save to CSV
manager.save_csv('products', 'tests/fixtures/products.csv')

# Save to JSON
manager.save_json('products', 'tests/fixtures/products.json')
```

Test Assertions

Helper functions for common test assertions.

DataFrame Equality

```
from dremioframe.testing import assert_dataframes_equal

expected = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})
actual = client.sql("SELECT * FROM test").collect()

assert_dataframes_equal(expected, actual)
```

Schema Validation

```
from dremioframe.testing import assert_schema_matches

result = client.sql("SELECT * FROM users").collect()

expected_schema = {
    'id': 'int64',
    'name': 'object',
    'age': 'int64'
}

assert_schema_matches(result, expected_schema)
```

Query Validation

```
from dremioframe.testing import assert_query_valid

sql = "SELECT * FROM users WHERE age > 25"
assert_query_valid(sql)  # Checks basic SQL syntax
```

Row Count Assertions

```
from dremioframe.testing import assert_row_count
```

```
result = client.sql("SELECT * FROM users").collect()

assert_row_count(result, 10, 'eq')   # Exactly 10 rows
assert_row_count(result, 5, 'gt')    # More than 5 rows
assert_row_count(result, 100, 'lt')  # Less than 100 rows
```

Complete Test Example

```
import pytest
from dremioframe.testing import (
    MockDremioClient,
    FixtureManager,
    assert_dataframes_equal,
    assert_schema_matches
)

@pytest.fixture
def mock_client():
    """Fixture providing a configured mock client"""
    client = MockDremioClient()

    # Set up test data
    users = pd.DataFrame({
        'id': [1, 2, 3],
        'name': ['Alice', 'Bob', 'Charlie'],
        'age': [25, 30, 35]
    })

    client.add_response("FROM users", users)
    return client

def test_user_query(mock_client):
    """Test querying users"""
    result = mock_client.sql("SELECT * FROM users WHERE age > 20").collect()

    # Assertions
    assert len(result) == 3
    assert_schema_matches(result, {
        'id': 'int64',
        'name': 'object',
        'age': 'int64'
    })

    # Verify query was executed
    assert "users" in mock_client.get_last_query()

def test_data_transformation(mock_client):
    """Test a data transformation pipeline"""
    # Your application code that uses the client
    raw_data = mock_client.sql("SELECT * FROM users").collect()
```

```
    # Transform
    transformed = raw_data[raw_data['age'] > 25]

    # Assert
    assert len(transformed) == 2
    assert all(transformed['age'] > 25)
```

Integration with pytest

Shared Fixtures

Create reusable fixtures in `conftest.py`:

```
# tests/conftest.py
import pytest
from dremioframe.testing import MockDremioClient, FixtureManager

@pytest.fixture
def mock_client():
    return MockDremioClient()

@pytest.fixture
def fixture_manager():
    return FixtureManager(fixtures_dir='tests/fixtures')
```

Parametrized Tests

```
@pytest.mark.parametrize("age,expected_count", [
    (20, 3),
    (30, 2),
    (40, 0)
])
def test_age_filter(mock_client, age, expected_count):
    result = mock_client.sql(f"SELECT * FROM users WHERE age > {age}").collect()
    assert len(result) == expected_count
```

Best Practices

Use Fixtures: Create pytest fixtures for common mock setups

Realistic Data: Use fixtures that mirror production data structure

Test Isolation: Clear query history between tests

Partial Matching: Use partial query matching for flexibility

Schema Validation: Always validate schema in addition to data

Limitations

No Actual Execution: Queries aren't validated against Dremio

Simple Matching: Query matching is string-based, not semantic

No Side Effects: Mock doesn't simulate Dremio-specific behaviors

In-Memory Only: All data must fit in memory