Source: advanced_features.md

Advanced Iceberg Features

IceFrame provides a suite of advanced features to give you a complete Iceberg experience, bridging gaps in the underlying libraries.

Iceberg Views

Manage cross-engine views (if supported by your catalog).

```
# Create a view
sql = "SELECT * FROM source_table WHERE id > 100"
ice.create_view("my_view", sql, replace=True)

# Drop a view
ice.drop_view("my_view")
```

Advanced Compaction

Optimize your data layout for better query performance.

```
from iceframe.compaction import CompactionManager

table = ice.get_table("my_table")
compactor = CompactionManager(table)

# Bin-pack small files into 128MB files
stats = compactor.bin_pack(target_file_size_mb=128)

# Sort data (Z-order approximation)
stats = compactor.sort(sort_order=["region", "ts"])
```

Partition Evolution

Evolve your table partitioning without rewriting data.

```
# Get evolution helper
evolver = ice.evolve_partition("my_table")

# Add partitions
evolver.add_day_partition("created_at")
evolver.add_bucket_partition("user_id", num_buckets=16)

# Remove partition
evolver.remove_partition("region")
```

Stored Procedures

Execute maintenance tasks using a familiar procedure call interface.

```
# Rewrite data files (compaction)
ice.call_procedure("my_table", "rewrite_data_files", target_file_size_mb=256)

# Expire snapshots (cleanup)
ice.call_procedure("my_table", "expire_snapshots", older_than_ms=...)

# Remove orphan files (GC)
ice.call_procedure("my_table", "remove_orphan_files")

# Fast-forward branch (WAP)
ice.call_procedure("my_table", "fast_forward", to_branch="audit_branch")
```

Merge-on-Read (MoR) Support

Note: Currently limited by underlying library support.

IceFrame includes structure for MoR writers (`MoRWriter`) to handle Position Deletes and Equality Deletes as support becomes available.

Garbage Collection

Parallelized cleanup operations for large tables.

```
from iceframe.gc import GarbageCollector

gc = GarbageCollector(table)
gc.expire_snapshots(retain_last=5)
```

Source: ai_agent.md

AI Agent

IceFrame includes an AI agent that provides a natural language interface for interacting with your Iceberg tables.

Features

- **Natural Language Queries**: Ask questions in plain English
- **Schema Discovery**: Explore tables and understand data structures
- **Code Generation**: Get Python code for complex operations
- **Query Optimization**: Receive suggestions for better performance

Multiple LLM Support: Works with OpenAI, Anthropic Claude, or Google Gemini

Setup

1. Install Dependencies

```
pip install "iceframe[agent]"
```

This installs: `openai`, `anthropic`, `google-generativeai`, and `rich`.

2. Configure LLM

Set one of these environment variables:

```
# OpenAI (GPT-4, GPT-3.5-turbo)
export OPENAI_API_KEY="your-key"

# Anthropic (Claude)
export ANTHROPIC_API_KEY="your-key"

# Google Gemini
export GOOGLE_API_KEY="your-key"
```

Optional: Specify provider and model explicitly:

```
export ICEFRAME_LLM_PROVIDER="openai"  # or "anthropic" or "gemini"
export ICEFRAME_LLM_MODEL="gpt-4"      # or specific model
```

Python API

```
from iceframe import IceFrame
from iceframe.agent.core import IceFrameAgent

# Initialize
ice = IceFrame(config)
agent = IceFrameAgent(ice)

# Chat
response = agent.chat("What tables do I have?")
print(response)

response = agent.chat("Show me users older than 30")
print(response)

# Reset conversation
agent.reset_conversation()
```

CLI Chat Interface

Start interactive chat:

```
iceframe-chat
```

Example session:

````
You: What tables do I have?
Assistant: You have the following tables:
- users
- orders
- products

You: Describe the users table
Assistant: The users table has the following schema:
- id (long, required)
- name (string)
- age (int)
- created_at (timestamp)

You: Show me users older than 30
Assistant: Here's a query to get users older than 30:

```python
from iceframe.expressions import Column

result = (ice.query("users")
    .filter(Column("age") > 30)
    .execute())
````

I found 15 users matching this criteria.

```

## Commands

- `exit` or `quit`: End session
- `clear` or `reset`: Clear conversation history

## Capabilities

### Schema Discovery

```

You: What columns are in the orders table?
Assistant: The orders table contains:

- order_id (long)

- user_id (long)

- amount (double)
```

- status (string)

- created_at (timestamp)

```
### Natural Language Queries
```

You: Find all orders over $100
Assistant: [Executes query and shows results]

```
### Code Generation
```

You: How do I join users with orders?
Assistant: Here's how to join users with orders:

```
result = (ice.query("users")
    .join("orders", on="user_id", how="inner")
    .select("name", "order_id", "amount")
    .execute())
```

```
### Query Optimization
```

You: How can I make my queries faster?
Assistant: Based on your query patterns, I recommend:

- Partition the orders table by date

- Use column pruning to select only needed columns

- Apply filters early to reduce data scanned

```
## Supported LLMs

| Provider | Models | Environment Variable |
|----------|--------|---------------------|
| OpenAI | GPT-4, GPT-3.5-turbo | `OPENAI_API_KEY` |
| Anthropic | Claude 3.5 Sonnet, Claude 3 Opus | `ANTHROPIC_API_KEY` |
| Google | Gemini 2.0 Flash, Gemini Pro | `GOOGLE_API_KEY` or `GEMINI_API_KEY` |

## Extending with Custom LLMs

Create a custom LLM provider:

```python
from iceframe.agent.llm_base import BaseLLM, LLMConfig
```
```

```python
class CustomLLM(BaseLLM):
    def chat(self, messages, tools=None):
        # Your implementation
        pass

    def stream_chat(self, messages):
        # Your implementation
        pass

# Use it
agent = IceFrameAgent(ice, llm=CustomLLM(config))
```

Source: async.md

Async Support

IceFrame provides async versions of core operations for non-blocking execution.

AsyncIceFrame

```python
import asyncio
from iceframe.async_ops import AsyncIceFrame

async def main():
    config = {...}
    async_ice = AsyncIceFrame(config)

    # Async read
    df = await async_ice.read_table_async("users")

    # Async write
    await async_ice.append_to_table_async("users", new_data)

    # Async stats
    stats = await async_ice.stats_async("users")

asyncio.run(main())
```

Async Query Builder

```python
from iceframe.expressions import Column

async def query_data():
    async_ice = AsyncIceFrame(config)

    query = await async_ice.query_async("users")
    result = await (query
        .filter(Column("age") > 25)
```

```
        .execute_async())

    return result

df = asyncio.run(query_data())
```

Use Cases

High Concurrency: Handle multiple table operations concurrently

Web Applications: Non-blocking API endpoints

Data Pipelines: Parallel processing of multiple tables

Source: branching.md

Branching and Tagging

IceFrame supports table branching and snapshot tagging (requires PyIceberg 0.6.0+ and catalog support).

Creating Branches

```
# Create branch from current snapshot
ice.create_branch("users", "experiment")

# Create branch from specific snapshot
ice.create_branch("users", "stable", snapshot_id=12345)
```

Tagging Snapshots

```
# Tag a snapshot for reference
table = ice.get_table("users")
snapshot_id = table.current_snapshot().snapshot_id

ice.tag_snapshot("users", snapshot_id, "v1.0")
```

Use Cases

Experimentation: Create branches for testing schema changes

Rollback: Tag stable snapshots for easy rollback

Versioning: Tag releases for reproducibility

Fast-Forwarding (Publishing)

You can fast-forward a branch (e.g., `main`) to another branch (e.g., `audit_branch`). This is essential for the Write-Audit-Publish (WAP) pattern.

```
from iceframe.branching import BranchManager

# Initialize manager
table = ice.get_table("users")
manager = BranchManager(table)

# Fast-forward main to audit_branch
manager.fast_forward("main", "audit_branch")
```

Write-Audit-Publish (WAP) Pattern

IceFrame supports the WAP pattern to ensure data quality:

- **Write**: Write data to a branch (e.g., `audit_branch`).
- **Audit**: Validate data in the branch.
- **Publish**: Fast-forward `main` to the branch.

```
# 1. Write to branch
ice.append_to_table("users", new_data, branch="audit_branch")

# 2. Audit (Validate)
# ... run checks ...

# 3. Publish
manager.fast_forward("main", "audit_branch")
```

Source: catalog_ops.md

Catalog Operations

Manage catalog-level operations beyond basic table CRUD.

Registering Tables

You can register an existing Iceberg table (metadata.json) into the catalog. This is useful for:

- Migrating tables between catalogs.
- Recovering tables from storage.
- Registering tables created by other engines.

```
# Register a table using its metadata location
metadata_url = "s3://bucket/warehouse/my_table/metadata/v1.metadata.json"

ice.register_table("my_new_table", metadata_url)
```

[!NOTE]
Not all catalogs support table registration. Check your catalog's documentation.

Source: cli.md

Command Line Interface (CLI)

IceFrame provides a CLI for managing Iceberg tables directly from the terminal.

Installation

The CLI requires the `cli` optional dependency:

```
pip install "iceframe[cli]"
```

Configuration

The CLI uses environment variables for configuration. You can set them in your shell or in a `.env` file.

```
export ICEBERG_CATALOG_URI="https://catalog.example.com"
export ICEBERG_CATALOG_TOKEN="your_token"
export ICEBERG_WAREHOUSE="s3://bucket/warehouse"
```

Commands

List Tables

List tables in a namespace (default is "default").

```
iceframe list
iceframe list --namespace marketing
```

Describe Table

Show table schema and partition spec.

```
iceframe describe my_table
```

Head Table

Show the first N rows of a table.

```
iceframe head my_table --n 10
```

Source: creating_tables.md

Creating Tables

IceFrame makes it easy to create Apache Iceberg tables with various schema formats.

Basic Usage

```
ice.create_table("my_table", schema)
```

Schema Formats

You can define schemas in several ways:

Dictionary Schema

Simple key-value pairs of column names and types.

```
schema = {
    "id": "long",
    "name": "string",
    "price": "double",
    "active": "boolean",
    "created_at": "timestamp",
    "birth_date": "date"
}
ice.create_table("products", schema)
```

Supported types: `string`, `int`, `long`, `float`, `double`, `boolean`, `timestamp`, `date`.

PyArrow Schema

For more control over types and nullability.

```
import pyarrow as pa

schema = pa.schema([
    pa.field("id", pa.int64(), nullable=False),
    pa.field("name", pa.string()),
```

```
    pa.field("tags", pa.list_(pa.string()))
])
ice.create_table("users", schema)
```

Polars DataFrame

Infer schema from an existing DataFrame.

```
import polars as pl

df = pl.DataFrame({"id": [1], "name": ["test"]})
ice.create_table("inferred_table", df)
```

Namespaces

You can specify namespaces (databases/schemas) in the table name:

```
# Creates table 'sales' in 'marketing' namespace
ice.create_table("marketing.sales", schema)
```

If the namespace doesn't exist, IceFrame will attempt to create it.

Advanced Options

Partitioning

Partition data for better query performance.

```
# Not yet fully exposed in high-level API, use underlying PyIceberg table object
# or pass partition_spec to create_table (requires PyIceberg PartitionSpec object)
```

Table Properties

Set table properties like compression codec.

```
properties = {
    "write.parquet.compression-codec": "zstd"
}
ice.create_table("optimized_table", schema, properties=properties)
```

Source: data_quality.md

Data Quality

IceFrame includes a Data Validator to ensure data quality before or after operations.

Accessing Data Validator

```
validator = ice.validator
```

Checking for Nulls

Check if specific columns contain null values.

```
import polars as pl

df = pl.DataFrame(...)

if not ice.validator.check_nulls(df, ["id", "created_at"]):
    print("Data contains nulls in required columns!")
```

Validating Constraints

Validate data against SQL-like constraints or custom functions.

```
import polars as pl

df = pl.DataFrame(...)

results = ice.validator.validate(df, [
    pl.col("age") > 0,
    pl.col("status").is_in(["active", "inactive"])
])

if not results["passed"]:
    print("Validation failed:", results["details"])
```

Source: deleting_tables.md

Deleting Tables

IceFrame allows you to delete tables and remove specific rows.

Dropping Tables

Remove a table from the catalog.

```
ice.drop_table("temp_table")
```

This removes the table metadata from the catalog. The underlying data files may be retained depending on catalog configuration (GC properties).

Deleting Rows

Delete rows matching a filter expression.

```
# Delete all users from 'test' region
ice.delete_from_table("users", "region = 'test'")
```

[!NOTE]
Row-level deletion requires Iceberg v2 tables and support from the underlying catalog and PyIceberg version.

Cleaning Up

After deleting tables or rows, you may want to perform Maintenance to clean up old files.

Source: dependencies.md

Dependencies

IceFrame is designed to be lightweight with a core set of dependencies and optional extras for specific features.

Core Dependencies

`pyiceberg`: Core Iceberg client

`polars`: High-performance DataFrame library

`pyarrow`: Apache Arrow support

`python-dotenv`: Environment variable management

Optional Dependencies

CLI (`[cli]`)

Required for the command-line interface.

`typer`: CLI application builder

`rich`: Terminal formatting

Install with:

```
pip install "iceframe[cli]"
```

Pydantic (`[pydantic]`)

Required for Pydantic integration.

`pydantic>=2.0.0`: Data validation and settings management

Install with:

```
pip install "iceframe[pydantic]"
```

Notebook (`[notebook]`)

Required for Jupyter Notebook integration.

`ipython>=8.0.0`: Interactive computing

`ipywidgets>=8.0.0`: Interactive HTML widgets

Install with:

```
pip install "iceframe[notebook]"
```

Cloud Storage (`[aws]`, `[gcs]`, `[azure]`)

Required for accessing cloud storage backends.

`s3fs`: AWS S3 support

`gcsfs`: Google Cloud Storage support

`adlfs`: Azure Data Lake Storage support

Install with:

```
pip install "iceframe[aws]"
# or
pip install "iceframe[aws,gcs]"
```

Async Support

Async operations use Python's built-in `asyncio` library (no additional dependencies required).

```
from iceframe.async_ops import AsyncIceFrame
```

AI Agent

The AI agent requires LLM provider packages. Install with:

```
pip install "iceframe[agent]"
```

This includes:

˜openai>=1.0.0` - For GPT models

˜anthropic>=0.18.0` - For Claude models

˜google-generativeai>=0.3.0` - For Gemini models

˜rich>=13.0.0` - For CLI formatting

You only need the API key for the LLM provider you want to use:

```
# Choose one:
export OPENAI_API_KEY="your-key"
export ANTHROPIC_API_KEY="your-key"
export GOOGLE_API_KEY="your-key"
```

Scalability Features

````
- `psutil>=5.9.0` - Memory monitoring
- `prometheus-client>=0.19.0` - Metrics export

### All Scalability Features
```bash
pip install "iceframe[cache,streaming,monitoring]"
````

Source: export.md

Exporting Data

IceFrame allows you to export Iceberg table data to common file formats.

Export to Parquet

Parquet is efficient for analytics and storage.

```
ice.to_parquet("my_table", "output/data.parquet")
```

With filtering and column selection:

```
ice.to_parquet(
    "my_table",
    "output/us_sales.parquet",
    columns=["id", "amount"],
    filter_expr="region = 'US'"
)
```
```

Export to CSV

Useful for spreadsheets and simple data exchange.

```
ice.to_csv("my_table", "output/data.csv")
```

Export to JSON

Useful for web APIs and NoSQL databases.

```
ice.to_json("my_table", "output/data.json")
```

Performance Note

Exports work by reading the data into memory (as a Polars DataFrame) and then writing to disk. For extremely large tables, consider filtering the data first or using a distributed engine.

Source: incremental.md

Incremental Processing

IceFrame supports incremental reads and change data capture (CDC) to efficiently process only new or changed data.

Reading Incremental Data

Read only data added since a specific snapshot:

```
# Get current snapshot ID
table = ice.get_table("logs")
snapshot_id = table.current_snapshot().snapshot_id

# ... time passes, more data is added ...

# Read only new data
new_data = ice.read_incremental(
    "logs",
    since_snapshot_id=snapshot_id
)
```

Or use a timestamp:

```
import time

timestamp_ms = int(time.time() * 1000)
# ... later ...
```

```
new_data = ice.read_incremental(
    "logs",
    since_timestamp=timestamp_ms
)
```

Change Data Capture (CDC)

Track changes between two snapshots:

```
changes = ice.get_changes(
    "users",
    from_snapshot_id=snapshot1,
    to_snapshot_id=snapshot2
)

print(f"Added: {changes['added'].height} rows")
print(f"Deleted: {changes['deleted'].height} rows")
```

Source: ingestion.md

Data Ingestion & Bulk Import

IceFrame supports efficient bulk import of existing data files into Iceberg tables without rewriting them.

Adding Files

If you have existing Parquet, Avro, or ORC files (e.g., from a migration or another job), you can register them directly into the table. This is a metadata-only operation and is extremely fast.

```
files = [
    "s3://bucket/data/file1.parquet",
    "s3://bucket/data/file2.parquet"
]

# Add files to the table
ice.add_files("my_table", files)
```

Requirements

- Files must match the table's schema.
- Files must be in a location accessible by the catalog/engine.
- If the table is partitioned, files should ideally align with partitions, though Iceberg can handle unpartitioned files in partitioned tables (they will be scanned to determine

partition values if metrics are available).

[!TIP]
Use this for migrating large datasets from legacy systems or other table formats.

Source: joins.md

JOIN Support

IceFrame Query Builder supports cross-table joins.

Basic JOIN

```
from iceframe.expressions import Column

# Inner join
result = (ice.query("users")
    .join("orders", on="user_id", how="inner")
    .select("name", "order_id", "amount")
    .execute())
```

JOIN Types

Supported join types: `inner`, `left`, `right`, `outer`

```
# Left join
result = (ice.query("users")
    .join("orders", on="user_id", how="left")
    .execute())

# Multiple joins
result = (ice.query("users")
    .join("orders", on="user_id")
    .join("products", on="product_id")
    .select("name", "product_name", "amount")
    .execute())
```

JOIN with Filters

```
result = (ice.query("users")
    .join("orders", on="user_id")
    .filter(Column("amount") > 100)
    .select("name", "order_id")
    .execute())
```

Source: maintenance.md

Table Maintenance

Iceberg tables require periodic maintenance to ensure optimal performance and manage storage costs. IceFrame provides simple methods for common maintenance tasks.

Expiring Snapshots

Remove old table snapshots to free up space and keep metadata size manageable.

```
# Remove snapshots older than 7 days, keeping at least the last 1
ice.expire_snapshots("my_table", older_than_days=7, retain_last=1)
```

Removing Orphan Files

Clean up data files that are no longer referenced by any snapshot (e.g., from failed writes).

```
# Remove orphan files older than 3 days
ice.remove_orphan_files("my_table", older_than_days=3)
```

Compacting Data Files

Combine small data files into larger ones to improve read performance (compaction).

```
# Compact files to target size of 512 MB
ice.compact_data_files("my_table", target_file_size_mb=512)
```

[!TIP]
Run compaction regularly on tables with frequent small updates (streaming ingestion).

Best Practices

Schedule Maintenance: Run these operations periodically (e.g., daily or weekly) via a scheduler like Airflow.

Order of Operations:

`expire_snapshots`

`remove_orphan_files`

`compact_data_files`

Source: namespaces.md

Namespace Management

IceFrame allows you to manage namespaces (schemas/databases) in your Iceberg catalog.

Accessing Namespace Manager

```
# Access via the namespaces property
ns_manager = ice.namespaces
```

Creating a Namespace

```
ice.create_namespace("marketing", {"owner": "team-marketing"})
```

Dropping a Namespace

```
ice.drop_namespace("marketing")
```

Listing Namespaces

```
# List top-level namespaces
namespaces = ice.list_namespaces()
print(namespaces)

# List nested namespaces
sub_namespaces = ice.list_namespaces("marketing")
```

Source: native_maintenance.md

Native Maintenance Operations

IceFrame provides native implementations of critical maintenance operations that work independently of PyIceberg version or catalog support.

Native Orphan File Removal

Remove data files that are no longer referenced by any snapshot.

```
from iceframe.gc import GarbageCollector

# Get table
table = ice.get_table("my_table")
gc = GarbageCollector(table)

# Dry run - list orphans without deleting
orphans = gc.remove_orphan_files(dry_run=True)
```

```
print(f"Found {len(orphans)} orphaned files")

# Remove orphans older than 3 days
import time
three_days_ago_ms = int((time.time() - 3 * 86400) * 1000)
removed = gc.remove_orphan_files(older_than_ms=three_days_ago_ms)
print(f"Removed {len(removed)} orphaned files")
```

How It Works

1. **Scan Manifests**: Reads all manifest files from the current snapshot to build a set of referenced data files
2. **List Storage**: Lists all files in the table's data directory
3. **Find Orphans**: Computes the difference (files in storage - files in manifests)
4. **Age Filter**: Optionally filters by file modification time
5. **Delete**: Removes orphaned files (unless `dry_run=True`)

Use Cases

- **After Failed Writes**: Clean up files from failed write operations
- **After Compaction**: Remove old data files after rewriting
- **Storage Optimization**: Reclaim storage from deleted/replaced files

[!TIP]
Always run with `dry_run=True` first to verify which files will be removed.

Native Snapshot Expiration

Expire old snapshots to reduce metadata size and improve query planning performance.

```
from iceframe.gc import GarbageCollector
import time

table = ice.get_table("my_table")
gc = GarbageCollector(table)

# Expire snapshots older than 7 days, keeping at least 5
seven_days_ago_ms = int((time.time() - 7 * 86400) * 1000)

try:
    expired = gc.expire_snapshots(
        older_than_ms=seven_days_ago_ms,
        retain_last=5
    )
    print(f"Expired {len(expired)} snapshots")
```

```
except NotImplementedError as e:
    print(f"Snapshot expiration not supported: {e}")
```

How It Works

List Snapshots: Gets all snapshots from table metadata

Apply Retention: Filters snapshots based on age and retention count

Expire: Uses PyIceberg's `expire_snapshots` if available, otherwise raises `NotImplementedError`

[!NOTE]
Snapshot expiration requires PyIceberg 0.7.0+ or catalog support. If not available, the operation will raise `NotImplementedError`.

Use Cases

Metadata Optimization: Reduce table metadata size

Query Performance: Improve query planning by reducing snapshot count

Compliance: Meet data retention policies

Comparison with PyIceberg

Operation	PyIceberg	IceFrame Native
Orphan File Removal	`table.remove_orphan_files()` (v0.7+)	Works on any version
Snapshot Expiration	`table.expire_snapshots()` (catalog-dependent)	⚠ Wraps PyIceberg, adds retry logic
Dry Run Support	Limited	Full support
Age Filtering	Basic	Enhanced with file stat checks

Best Practices

1. Schedule Regular Maintenance

```
# Weekly orphan file cleanup
gc.remove_orphan_files(older_than_ms=three_days_ago_ms)

# Monthly snapshot expiration
gc.expire_snapshots(older_than_ms=thirty_days_ago_ms, retain_last=10)
```

2. Use Dry Run First

```
# Always verify before deleting
```

```
orphans = gc.remove_orphan_files(dry_run=True)
if len(orphans) > 1000:
    print("Warning: Large number of orphans detected!")
    # Investigate before proceeding
```

3. Monitor Storage Savings

```
import os

# Calculate storage before
orphans = gc.remove_orphan_files(dry_run=True)
total_size = sum(os.path.getsize(f) for f in orphans if os.path.exists(f))
print(f"Potential savings: {total_size / 1024**3:.2f} GB")

# Perform cleanup
gc.remove_orphan_files()
```

Limitations

- **Snapshot Expiration**: Requires PyIceberg 0.7.0+ or catalog support
- **Concurrent Writes**: Orphan detection may miss files from concurrent write operations
- **Distributed Storage**: File listing performance depends on storage system (S3, HDFS, etc.)

See Also

- Maintenance Guide
- Garbage Collection
- Table Optimization

Source: notebooks.md

Notebook Integration

IceFrame provides rich integration with Jupyter Notebooks and IPython environments.

Installation

To use notebook features, install IceFrame with the `notebook` extra:

```
pip install "iceframe[notebook]"
```

Rich Display

When you display an `IceFrame` instance in a notebook, it shows a formatted summary of the connection and available namespaces.

```
from iceframe import IceFrame

ice = IceFrame(config)
ice  # Displays HTML summary
```

Magic Commands

IceFrame includes IPython magic commands to simplify interaction.

Loading the Extension

First, load the extension:

```
%load_ext iceframe.magics
```

%iceframe

Set the active IceFrame instance for magic commands.

```
ice = IceFrame(config)
%iceframe ice
```

Check status:

```
%iceframe status
```

%%iceql

Execute SQL queries directly in a cell using the active IceFrame instance.

```
%%iceql
SELECT * FROM my_table LIMIT 10
```

You can also perform joins:

```
%%iceql
SELECT
    t1.name,
    t2.order_total
FROM users t1
JOIN orders t2 ON t1.id = t2.user_id
```

Note: `%%iceql` uses Polars SQL context under the hood. It automatically registers tables referenced in the query from your Iceberg catalog.

Source: partitioning.md

Partition Management

IceFrame allows you to manage table partitioning to optimize query performance.

Accessing Partition Manager

```
# Get partition manager for a table
partitioner = ice.partition_by("logs")
```

Adding Partition Fields

IceFrame supports various transforms: `identity`, `bucket`, `truncate`, `year`, `month`, `day`, `hour`.

```
# Partition by 'category' (identity)
ice.partition_by("logs").add_partition_field("category")

# Partition by day of 'timestamp'
ice.partition_by("logs").add_partition_field("timestamp", "day", name="day_ts")

# Partition by bucket of 'user_id'
ice.partition_by("logs").add_partition_field("user_id", "bucket", 16,
name="user_bucket")
```

Dropping Partition Fields

```
# Drop partition field
ice.partition_by("logs").drop_partition_field("category")
```

Source: pydantic.md

Pydantic Integration

IceFrame supports integration with Pydantic (v2) for schema definition and data validation.

Installation

To use Pydantic features, install IceFrame with the `pydantic` extra:

```
pip install "iceframe[pydantic]"
```

Defining Tables with Pydantic Models

You can use Pydantic models to define your table schema instead of PyArrow or PyIceberg schemas.

```
from pydantic import BaseModel
from typing import Optional
from datetime import datetime
from iceframe import IceFrame

# Define your model
class User(BaseModel):
    id: int
    name: str
    email: Optional[str] = None
    created_at: datetime = datetime.now()
    is_active: bool = True

# Initialize IceFrame
ice = IceFrame(config)

# Create table using the model
ice.create_table("my_namespace.users", schema=User)
```

Inserting Data

You can insert a list of Pydantic model instances directly into a table using `insert_items`.

```
# Create user instances
users = [
    User(id=1, name="Alice", email="alice@example.com"),
    User(id=2, name="Bob", is_active=False)
]

# Insert into table
ice.insert_items("my_namespace.users", users)
```

Type Mapping

IceFrame maps Python/Pydantic types to Iceberg types as follows:

Python Type	Iceberg Type
`str`	`string`
`int`	`long`

`float`	`double`
`bool`	`boolean`
`datetime`	`timestamp`
`date`	`date`
`List[T]`	`list<T>`
`Optional[T]`	`T` (nullable)

Advanced Usage

For more complex schemas (nested structs, maps), you can nest Pydantic models.

```
class Address(BaseModel):
    street: str
    city: str
    zip: str

class UserWithAddress(BaseModel):
    id: int
    name: str
    address: Address  # Maps to Iceberg Struct
```

Source: query_builder.md

Query Builder API

IceFrame provides a powerful, fluent Query Builder API for constructing complex queries with SQL-like capabilities.

Overview

The Query Builder allows you to:

- Select columns and apply expressions
- Filter data with predicate pushdown support
- Group by and aggregate data
- Sort and limit results
- Use window functions and case statements
- Perform write operations (Insert, Update, Delete, Merge)

Basic Usage

Start a query using `ice.query("table_name")`:

```
from iceframe.expressions import col, lit

df = (ice.query("sales")
```

```
      .select("id", "amount")
      .filter(col("amount") > 100)
      .execute())
```

Expressions

IceFrame provides a unified expression system that works with both PyIceberg (for pushdown) and Polars (for local processing).

```
from iceframe.expressions import col, lit

# Binary operations
col("age") > 18
col("status") == "active"

# Boolean logic
(col("age") > 18) & (col("status") == "active")
(col("category") == "A") | (col("category") == "B")

# IN / IS NULL
col("id").is_in([1, 2, 3])
col("name").is_null()
```

Aggregations

Use standard SQL aggregate functions:

```
from iceframe.functions import count, sum, avg, min, max

df = (ice.query("sales")
      .select(
          col("region"),
          sum(col("amount")).alias("total_sales"),
          avg(col("amount")).alias("avg_sales")
      )
      .group_by("region")
      .execute())
```

Window Functions

Support for window functions like `row_number`, `rank`, `dense_rank`:

```
from iceframe.functions import row_number

df = (ice.query("sales")
      .select(
          col("id"),
          col("amount"),
          row_number().over(
```

```
            partition_by=col("region"),
            order_by=col("amount")
        ).alias("rank")
    )
    .execute())
```

Case Statements

Conditional logic with `when/otherwise`:

```
from iceframe.functions import when

df = (ice.query("users")
      .select(
          col("name"),
          when(col("age") < 18, "Minor")
          .when(col("age") < 65, "Adult")
          .otherwise("Senior")
          .alias("age_group")
      )
      .execute())
```

Write Operations

The Query Builder also supports write operations.

Insert

```
ice.query("users").insert(new_data_df)
```

Update

Update rows matching a filter:

```
(ice.query("users")
 .filter(col("id") == 123)
 .update({"status": "inactive"}))
```

[!WARNING]
Updates are currently implemented as Copy-on-Write (overwrite entire table). Use with caution on large tables.

Delete

Delete rows matching a filter:

```
(ice.query("users")
 .filter(col("status") == "deleted")
 .delete())
```

Merge (Upsert)

Merge source data into the target table:

```
(ice.query("target_table")
 .merge(
     source_data=source_df,
     on="id",
     when_matched_update={"status": "status", "updated_at": "updated_at"},
             when_not_matched_insert={"id":   "id",   "status":   "status",   "created_at":
"created_at"}
 ))
```

Source: reading_tables.md

Reading Tables

IceFrame provides a simple API to read Iceberg tables into Polars DataFrames.

Basic Reading

```
df = ice.read_table("my_table")
```

Column Selection

Read only specific columns to improve performance.

```
df = ice.read_table("users", columns=["id", "email"])
```

Filtering

Filter data at the source (predicate pushdown).

```
# Filter expression using SQL-like syntax
df = ice.read_table("sales", filter_expr="amount > 100 AND region = 'US'")
```

Limiting Results

Limit the number of rows returned.

```
df = ice.read_table("logs", limit=100)
```

Time Travel

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

By Snapshot ID

```
df = ice.read_table("my_table", snapshot_id=123456789012345)
```

By Timestamp

```
# Read as of 1 hour ago
timestamp_ms = int((time.time() - 3600) * 1000)
df = ice.read_table("my_table", as_of_timestamp=timestamp_ms)
```

Accessing Underlying Table

For advanced operations, you can access the underlying PyIceberg Table object.

```
table = ice.get_table("my_table")
# Use PyIceberg API directly
scan = table.scan()
```

Source: data_quality_gate.md

Data Quality Gate Recipe

This recipe demonstrates how to implement a "Write-Audit-Publish" pattern using IceFrame's branching and data quality features.

Scenario

You want to load data into a production table, but only if it passes strict quality checks (e.g., no nulls in critical columns, values within range). If checks fail, the data should be isolated for review without affecting production.

Implementation

```
from iceframe import IceFrame
from iceframe.branching import BranchManager
```

```python
ice = IceFrame(config)
table_name = "finance.transactions"
staging_branch = "staging_audit"

def load_with_quality_gate(new_data_df):
    # 1. Create a Branch for Staging
    # We write to a branch first so production readers don't see incomplete/bad data
    branch_manager = BranchManager(ice.catalog)

    # Create or replace branch pointing to current main
    try:
        branch_manager.create_branch(table_name, staging_branch)
    except:
        # If exists, fast-forward or reset (simplified here)
        pass

    print(f"Writing data to branch '{staging_branch}'...")

    # 2. Write to the Branch
    # (IceFrame write methods support a 'branch' argument if implemented,
    # or we configure the write to target the branch ref)
    # For this recipe, we assume we can write to the branch:
    ice.append_to_table(table_name, new_data_df, branch=staging_branch)

    # 3. Run Data Quality Checks on the Branch
    print("Running quality checks...")

    # Define constraints
    constraints = [
        {"type": "not_null", "columns": ["transaction_id", "amount"]},
        {"type": "range", "column": "amount", "min": 0},
        {"type": "unique", "column": "transaction_id"}
    ]

    # Validate data in the branch
    # (We read from the branch to validate)
    validation_results = ice.validate_data(
        table_name,
        constraints,
        branch=staging_branch
    )

    if validation_results["passed"]:
        print(" Quality checks passed. Promoting to main.")

        # 4. Fast-Forward 'main' to 'staging_branch'
        # This makes the data visible to production users atomically
        branch_manager.fast_forward(table_name, "main", staging_branch)

    else:
        print(" Quality checks FAILED.")
        print(f"Violations: {validation_results['violations']}")
        print(f"Bad data is isolated in branch '{staging_branch}' for review.")
```

```
        # Do NOT merge to main. Alert the team.

# Example Run
df = pl.read_parquet("incoming_transactions.parquet")
load_with_quality_gate(df)
```

Key Features Used

- **Branching**: Isolates unverified data from production readers.
- **Data Validator**: Automates quality checks.
- **Atomic Promotion**: Fast-forward merge ensures all-or-nothing visibility.

Source: etl_pipeline.md

ETL Pipeline Recipe

This recipe demonstrates how to build a simple ETL (Extract, Transform, Load) pipeline using IceFrame.

Scenario

You need to ingest raw JSON logs, clean the data, enrich it, and load it into an Iceberg table partitioned by date.

Implementation

```
from iceframe import IceFrame
import polars as pl
from datetime import datetime

# 1. Initialize IceFrame
config = {
    "uri": "http://localhost:8181",
    "type": "rest",
    "warehouse": "s3://warehouse"
}
ice = IceFrame(config)

def run_etl_job(source_file: str, target_table: str):
    print(f"Starting ETL job for {source_file}...")

    # --- EXTRACT ---
    # Read raw data using Polars
    raw_df = pl.read_json(source_file)

    # --- TRANSFORM ---
    # Clean and enrich data
```

```
    processed_df = (
        raw_df
        .with_columns([
            # Convert timestamp string to datetime
            pl.col("timestamp").str.to_datetime(),
            # Extract date for partitioning
            pl.col("timestamp").str.to_datetime().dt.date().alias("event_date"),
            # Clean strings
            pl.col("user_agent").str.strip_chars(),
            # Add processing metadata
            pl.lit(datetime.now()).alias("processed_at")
        ])
        .filter(
            # Remove invalid records
            pl.col("user_id").is_not_null()
        )
    )

    # --- LOAD ---
    # Check if table exists, create if not
    if not ice.table_exists(target_table):
        print(f"Creating table {target_table}...")
        ice.create_table(
            target_table,
            schema=processed_df.schema,
            partition_spec=[("event_date", "identity")]
        )

    # Append data to Iceberg table
    print(f"Writing {processed_df.height} records to {target_table}...")
    ice.append_to_table(target_table, processed_df)

    print("ETL job completed successfully!")

# Run the job
run_etl_job("raw_logs.json", "analytics.web_logs")
```

Key Features Used

Polars Integration: Uses Polars for efficient in-memory transformation.

Schema Inference: Automatically infers Iceberg schema from the DataFrame.

Partitioning: Creates a partitioned table for optimized querying.

Idempotency: Checks for table existence before creation.

Source: incremental_ingestion.md

Incremental Ingestion Recipe

This recipe demonstrates how to process data incrementally, reading only what has changed since the last run.

Scenario

You have a downstream table `daily_summary` that aggregates data from an upstream `raw_events` table. You want to run this aggregation periodically, processing only new data added to `raw_events`.

Implementation

```
from iceframe import IceFrame
import json
import os

ice = IceFrame(config)
state_file = "ingestion_state.json"

def get_last_snapshot_id():
    if os.path.exists(state_file):
        with open(state_file, "r") as f:
            return json.load(f).get("last_snapshot_id")
    return None

def save_state(snapshot_id):
    with open(state_file, "w") as f:
        json.dump({"last_snapshot_id": snapshot_id}, f)

def run_incremental_job():
    source_table = "raw_events"
    target_table = "daily_summary"

    # 1. Get last processed state
    last_snapshot = get_last_snapshot_id()

    # 2. Read incremental data
    print(f"Reading {source_table} since snapshot {last_snapshot}...")

    # Use IceFrame's incremental reader
    # This returns a DataFrame containing only new rows appended since snapshot
    new_data = ice.read_incremental(
        source_table,
        since_snapshot_id=last_snapshot
    )

    if new_data.height == 0:
        print("No new data found.")
        return

    # 3. Process the data (Aggregation)
    summary_df = (
        new_data
        .group_by("event_date", "category")
```

```
        .agg([
            pl.count().alias("event_count"),
            pl.sum("amount").alias("total_amount")
        ])
    )

    # 4. Write to target (Upsert/Merge)
    # We merge into the summary table to update counts
    (ice.query(target_table)
        .merge(summary_df, on=["event_date", "category"])
        .when_matched_update({
            "event_count": pl.col("target.event_count") + pl.col("source.event_count"),
            "total_amount": pl.col("target.total_amount") + pl.col("source.total_amount")
        })
        .when_not_matched_insert({
            "event_date": pl.col("source.event_date"),
            "category": pl.col("source.category"),
            "event_count": pl.col("source.event_count"),
            "total_amount": pl.col("source.total_amount")
        })
        .execute())

    # 5. Update state
    # Get the current snapshot ID of the source table to save for next time
    current_snapshot = ice.get_table(source_table).current_snapshot().snapshot_id
    save_state(current_snapshot)
    print(f"Job finished. State updated to snapshot {current_snapshot}")

run_incremental_job()
```

Key Features Used

- **Incremental Read**: `read_incremental` efficiently fetches only new data.
- **Merge/Upsert**: Updates existing aggregates or inserts new ones.
- **State Management**: Tracks progress via snapshot IDs.

Source: scd_type_2.md

Slowly Changing Dimensions (SCD) Type 2 Recipe

This recipe demonstrates how to implement SCD Type 2 (retaining full history) using IceFrame's merge capabilities.

Scenario

You have a `users` table and receive updates. You want to track historical changes to user profiles (e.g., address changes) by creating new records for updates while marking old records as inactive.

Implementation

```
from iceframe import IceFrame
from iceframe.expressions import Column
import polars as pl

ice = IceFrame(config)
target_table = "dim.users"

def process_scd_type_2(updates_df: pl.DataFrame):
    """
    Apply SCD Type 2 updates to the users dimension table.
    Schema: user_id, name, address, is_active, valid_from, valid_to
    """

    # 1. Identify records that have actually changed
    # Read current active records
    current_df = (
        ice.query(target_table)
        .filter(Column("is_active") == True)
        .execute()
    )

    # Join to find changes
    # (In a real scenario, you'd hash columns to detect changes efficiently)

    # 2. Prepare the MERGE operation
    # For SCD Type 2, we typically do this in two steps or use a complex merge.
    # Here is a simplified approach using IceFrame's merge:

    # Step A: Expire old records
    # Update existing records where ID matches but content differs
    (ice.query(target_table)
        .merge(updates_df, on="user_id")
        .when_matched_update({
            "is_active": False,
            "valid_to": pl.col("source.valid_from")
        })
        .execute())

    # Step B: Insert new versions
    # Insert new records for all updates
    new_records = updates_df.with_columns([
        pl.lit(True).alias("is_active"),
        pl.lit(None).alias("valid_to")
    ])

    ice.append_to_table(target_table, new_records)

# Example Usage
```

```
updates = pl.DataFrame({
    "user_id": [101, 102],
    "name": ["Alice", "Bob"],
    "address": ["New Address St", "Same Address"],
    "valid_from": [datetime.now(), datetime.now()]
})

process_scd_type_2(updates)
```

[!NOTE]
True SCD Type 2 often requires complex logic to handle out-of-order data and exact timestamp alignment. This recipe shows the fundamental pattern of expiring old rows and inserting new ones.

Key Features Used

- **Merge Operation**: Used to update existing records based on a key.
- **Predicate Pushdown**: Efficiently reads only active records.

Source: rollback.md

Rollback & Snapshot Management

IceFrame provides tools to manage table history, allowing you to rollback to previous states or manage branch pointers.

Rollback

Revert the table state to a specific snapshot or point in time.

```
# Rollback to a specific snapshot ID
ice.rollback_to_snapshot("my_table", 1234567890)

# Rollback to a timestamp (milliseconds since epoch)
ice.rollback_to_timestamp("my_table", 1704067200000)
```

Snapshot Management

Explicitly set the current snapshot of the table. This is useful for advanced workflows like cherry-picking or manually moving branch pointers.

```
# Set the current snapshot of the table
ice.call_procedure("my_table", "set_current_snapshot", snapshot_id=9876543210)
```

[!WARNING]
Rolling back changes the current state of the table. Ensure you have the correct

snapshot ID or timestamp before proceeding.

Source: scalability.md

Scalability Features

IceFrame includes comprehensive scalability features for high-performance data processing.

Query Result Caching

Cache query results to avoid redundant computation:

```
from iceframe.cache import QueryCache

# In-memory cache
cache = QueryCache(max_size=100)

# Use with queries
result = ice.query("users").filter(Column("age") > 30).cache(ttl=3600).execute()
```

Install: No additional dependencies required

Parallel Table Operations

Read multiple tables concurrently:

```
from iceframe.parallel import ParallelExecutor

executor = ParallelExecutor(max_workers=4)
results = executor.read_tables_parallel(ice, ["users", "orders", "products"])
```

Install: No additional dependencies required

Connection Pooling

Reuse catalog connections for better performance:

```
from iceframe.pool import CatalogPool

pool = CatalogPool(catalog_config, pool_size=5)
conn = pool.get_connection()
# Use connection
pool.return_connection(conn)
```

Install: No additional dependencies required

Memory Management

Process large tables in chunks:

```
from iceframe.memory import MemoryManager

manager = MemoryManager(max_memory_mb=1000)

# Read in chunks
for chunk in manager.read_table_chunked(ice, "huge_table", chunk_size=10000):
    process(chunk)
```

Install: `pip install "iceframe[monitoring]"` (for psutil)

Query Optimization

Automatic query optimization:

```
from iceframe.optimizer import QueryOptimizer

optimizer = QueryOptimizer()
analysis = optimizer.analyze_query("users", select_exprs, filter_exprs, group_by_exprs)
print(analysis["suggestions"])
```

Install: No additional dependencies required

Monitoring & Observability

Track query performance:

```
from iceframe.monitoring import MetricsCollector

collector = MetricsCollector()
query_id = collector.start_query("users")
# Execute query
collector.end_query(query_id, rows_returned=1000)

stats = collector.get_stats()
print(f"Avg duration: {stats['avg_duration_ms']}ms")
```

Install: `pip install "iceframe[monitoring]"` (for psutil, prometheus-client)

Streaming Support

Stream data to Iceberg tables:

```
from iceframe.streaming import StreamingWriter, stream_from_kafka

# Micro-batch streaming
```

```
writer = StreamingWriter(ice, "events", batch_size=1000)
writer.write({"id": 1, "event": "click"})
writer.flush()

# Kafka integration
stream_from_kafka(ice, "kafka-topic", "events_table", kafka_config)
```

Install: `pip install "iceframe[streaming]"` (for kafka-python)

Data Skipping

Skip unnecessary data files using statistics:

```
from iceframe.skipping import DataSkipper

skipper = DataSkipper()
stats = skipper.get_stats()
print(f"Skip rate: {stats['skip_rate']:.2%}")
```

Install: No additional dependencies required

Catalog Federation

Query across multiple catalogs:

```
from iceframe.federation import CatalogFederation

federation = CatalogFederation()
federation.add_catalog("prod", prod_config)
federation.add_catalog("dev", dev_config)

# Read from specific catalog
df = federation.read_table("prod", "users")

# Union across catalogs
combined = federation.union_tables([
    ("prod", "users"),
    ("dev", "users")
])
```

Install: No additional dependencies required

Installation

Install all scalability features:

```
pip install "iceframe[cache,streaming,monitoring]"
```

Or install individually as needed.

Source: schema_evolution.md

Schema Evolution

IceFrame provides a simple API for evolving table schemas without rewriting data.

Accessing Schema Evolution

```
# Get schema evolution interface for a table
schema_evo = ice.alter_table("users")
```

Adding Columns

```
# Add a new column
ice.alter_table("users").add_column("email", "string", doc="User email address")
```

Dropping Columns

```
# Drop a column
ice.alter_table("users").drop_column("temp_field")
```

Renaming Columns

```
# Rename a column
ice.alter_table("users").rename_column("name", "full_name")
```

Updating Column Types

Iceberg supports safe type promotion (e.g., int -> long, float -> double).

```
# Update column type
ice.alter_table("users").update_column_type("age", "long")
```

Source: statistics.md

Table Statistics

IceFrame provides comprehensive table statistics and column profiling.

Table-Level Statistics

Get overall table metadata:

```
stats = ice.stats("users")

print(f"Total snapshots: {stats['snapshots']['count']}")
print(f"Columns: {stats['schema']['columns']}")
print(f"Total records: {stats['data']['total_records']}")
```

Column Profiling

Profile individual columns:

```
# Numeric column
profile = ice.profile_column("users", "age")
print(f"Min: {profile['numeric_stats']['min']}")
print(f"Max: {profile['numeric_stats']['max']}")
print(f"Mean: {profile['numeric_stats']['mean']}")

# String column
profile = ice.profile_column("users", "name")
print(f"Avg length: {profile['string_stats']['avg_length']}")
print(f"Distinct values: {profile['distinct_count']}")
```

Source: updating_tables.md

Updating Tables

IceFrame supports appending to and overwriting Iceberg tables.

Appending Data

Add new rows to an existing table.

```
import polars as pl

new_data = pl.DataFrame({
    "id": [3, 4],
    "name": ["Charlie", "David"]
})

ice.append_to_table("users", new_data)
```

You can also append using PyArrow Tables or Python dictionaries:

```
data_dict = {
    "id": [5],
```

```
    "name": ["Eve"]
}
ice.append_to_table("users", data_dict)
```

[!IMPORTANT]
Ensure data types match the table schema exactly. For example, use `int32` for `int` columns and `int64` for `long` columns.

Overwriting Data

Replace all data in the table with new data.

```
# Replaces entire table content
ice.overwrite_table("daily_report", today_data)
```

Upserts / Merge

Currently, IceFrame supports Append and Overwrite. Full Merge/Upsert functionality (Merge-on-Read) depends on the underlying PyIceberg support for your specific catalog and table version (v2).

For basic updates, you can:

- Read the table
- Modify the DataFrame locally
- Overwrite the table (for small tables)

For large tables, use SQL-based engines (like Spark, Trino, or Dremio) connected to the same catalog for complex merge operations.