Efficient ML pipelines using Parquet and PyArrow

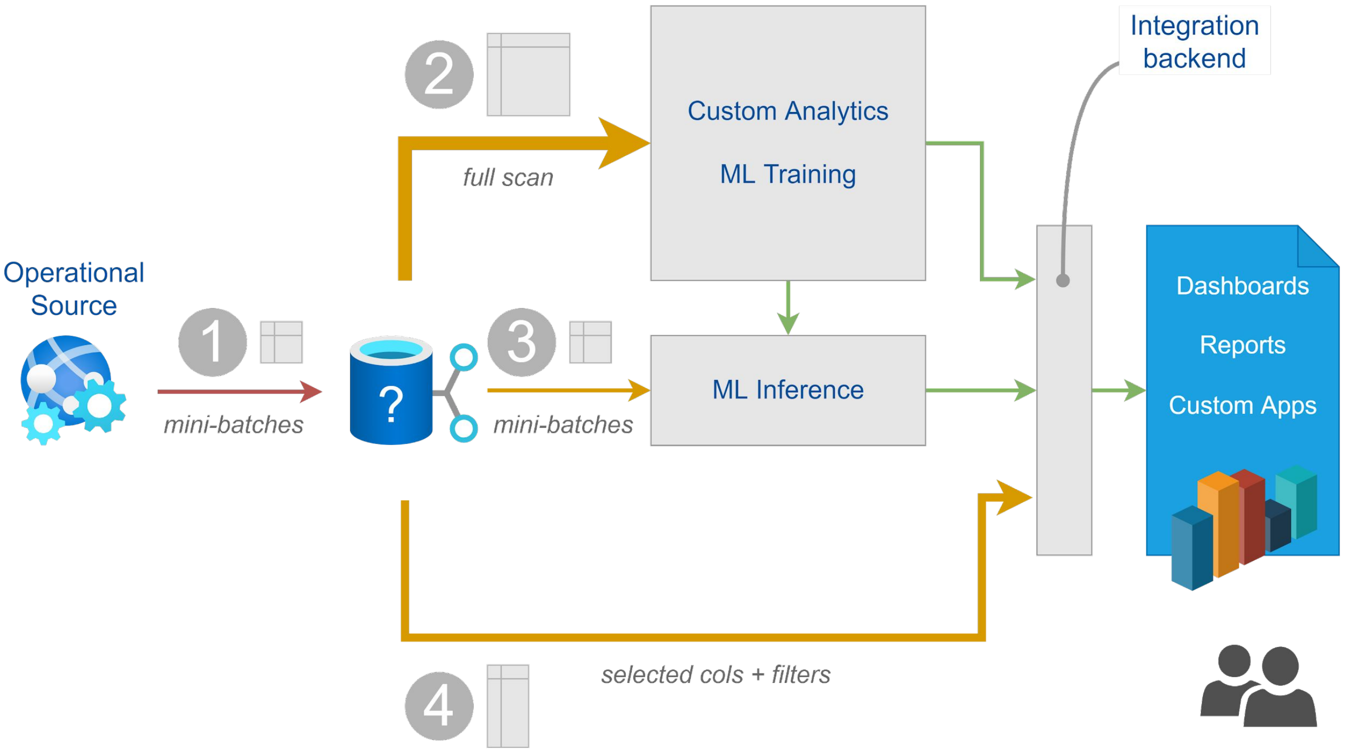
Antonino Ingargiola

PyCon IT 2022

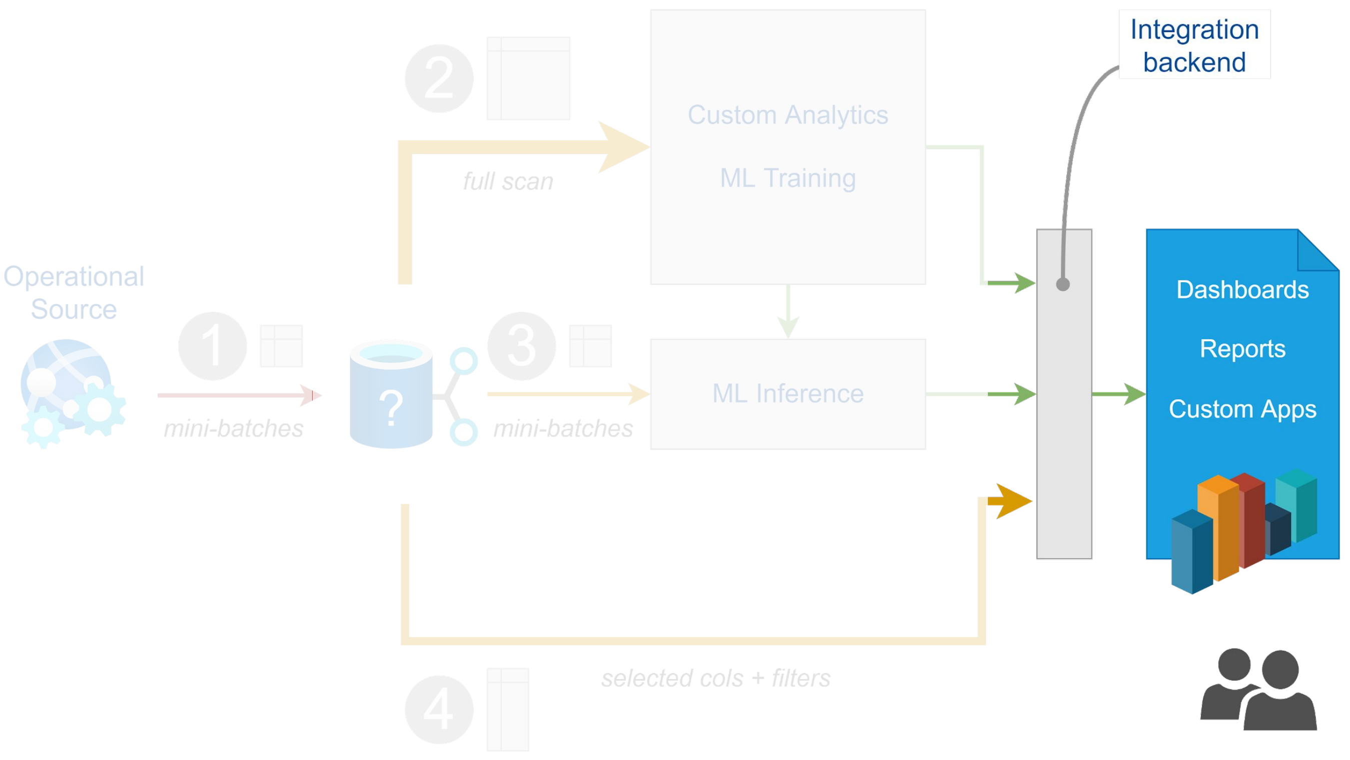
Overview

- ➔ Example of a ML data pipeline
- ➔ Parquet format
- ➔ Arrow and PyArrow
- ➔ Workflows
- ➔ Related tech

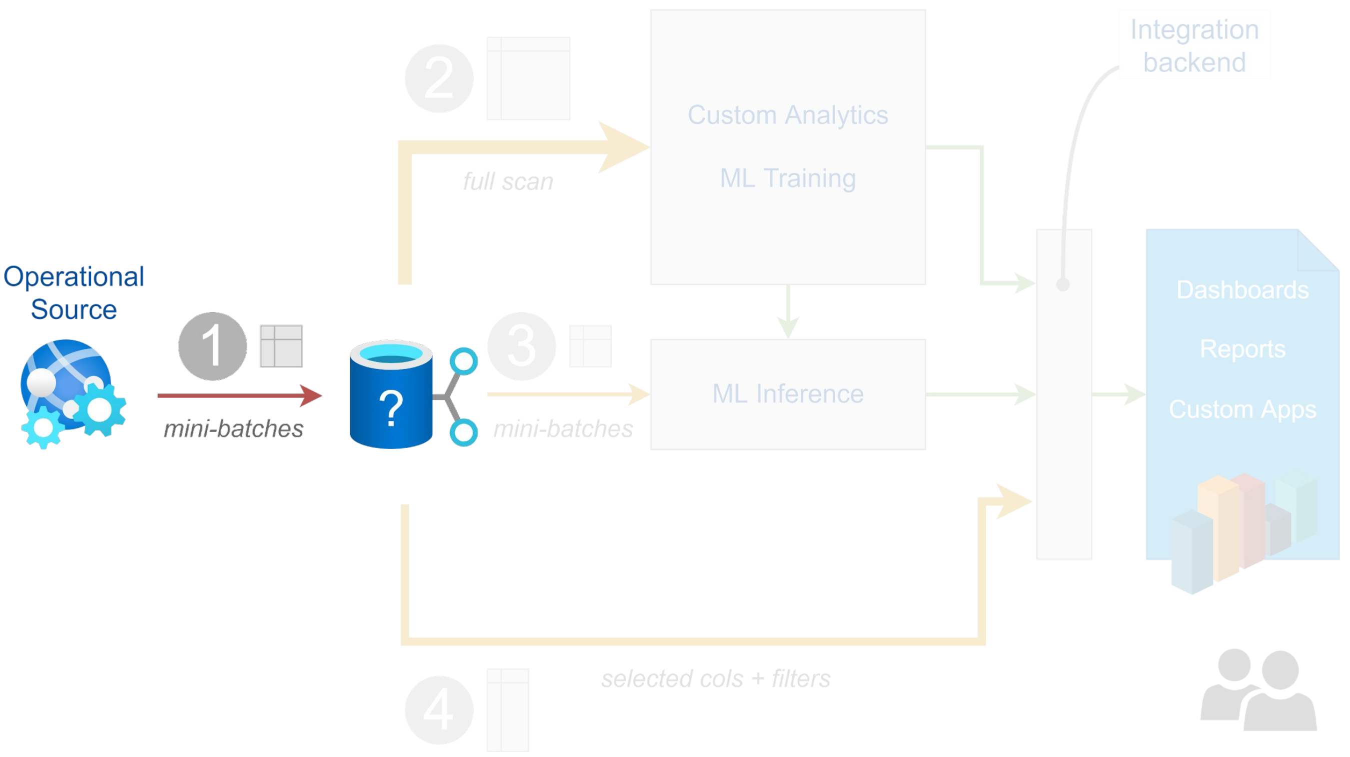
Example of a ML production pipeline



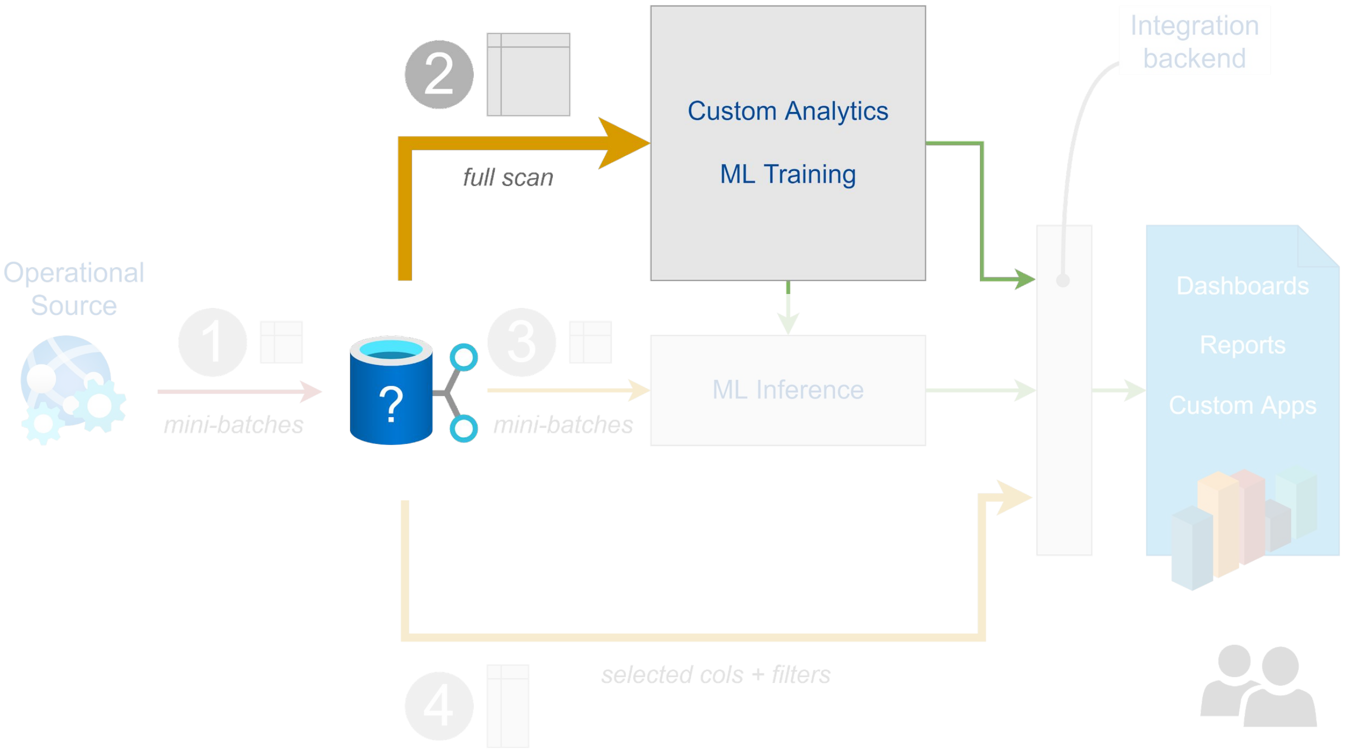
Example of a ML production pipeline



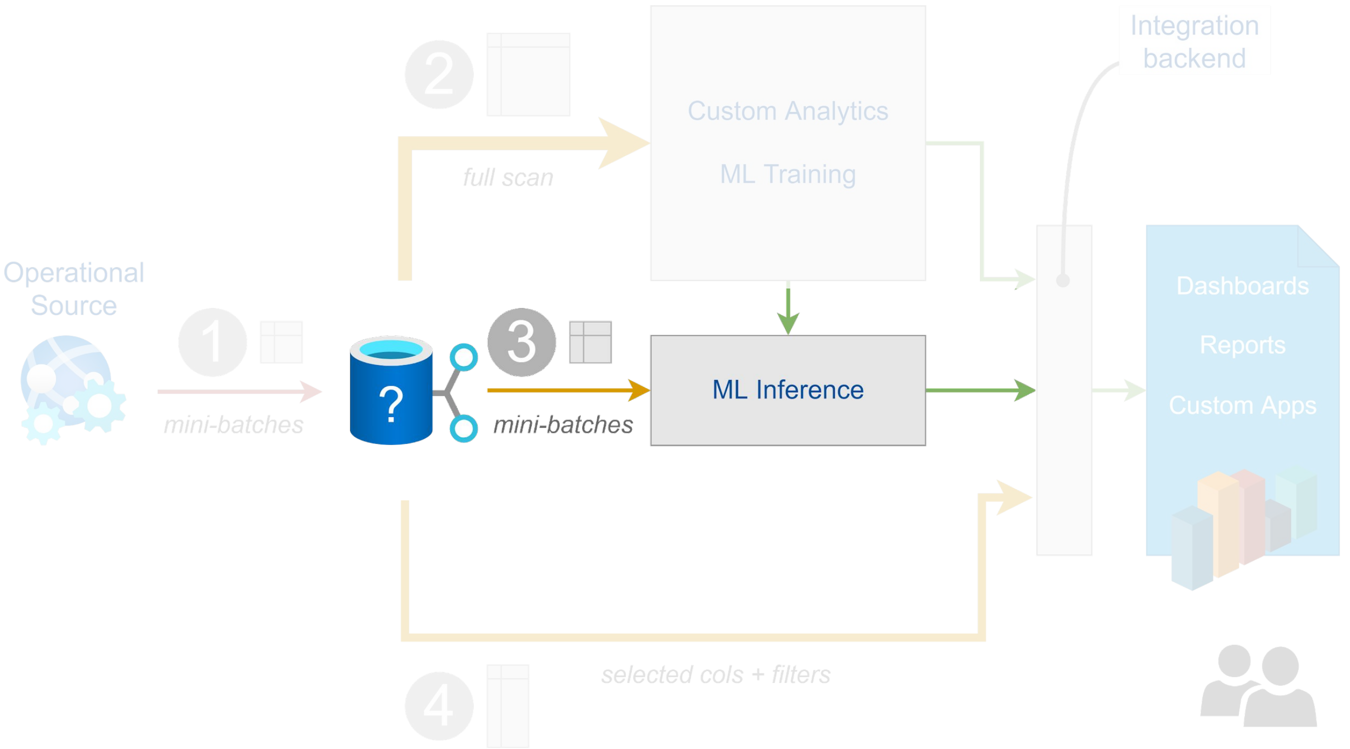
Example of a ML production pipeline



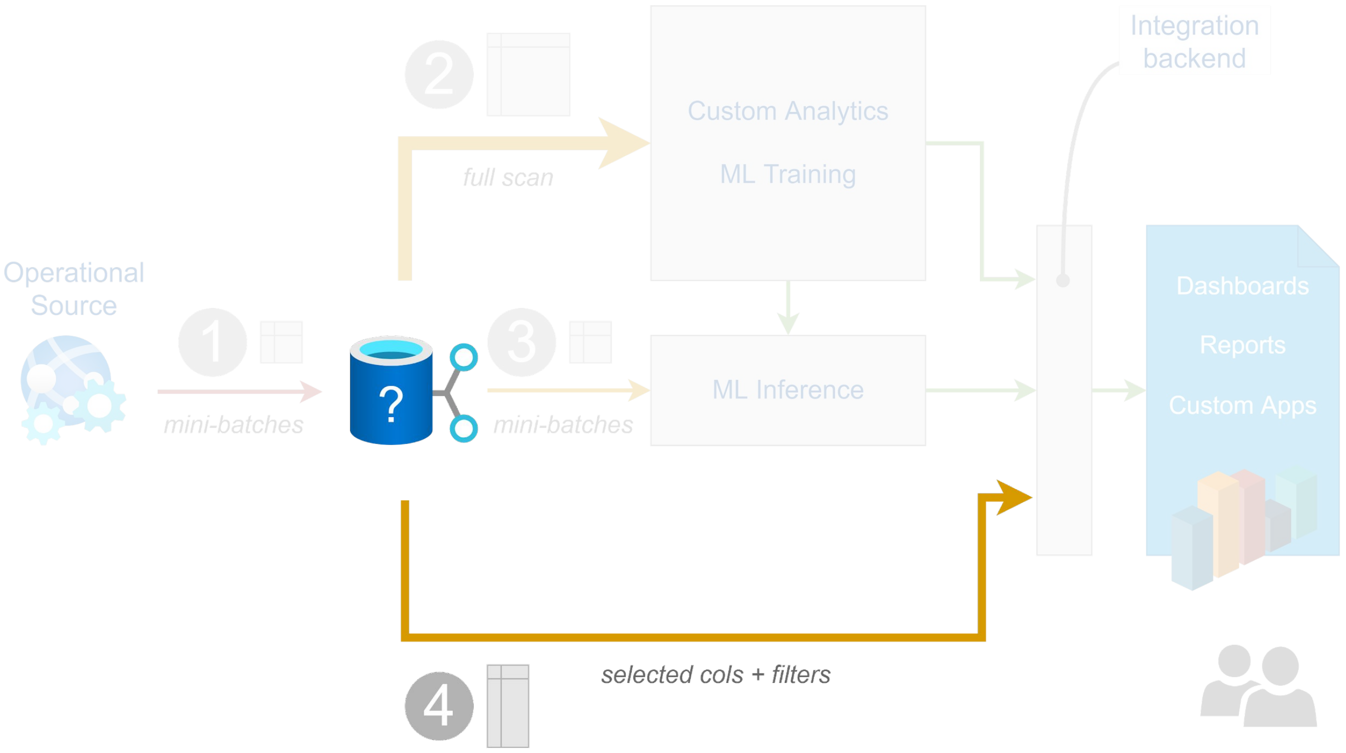
Example of a ML production pipeline



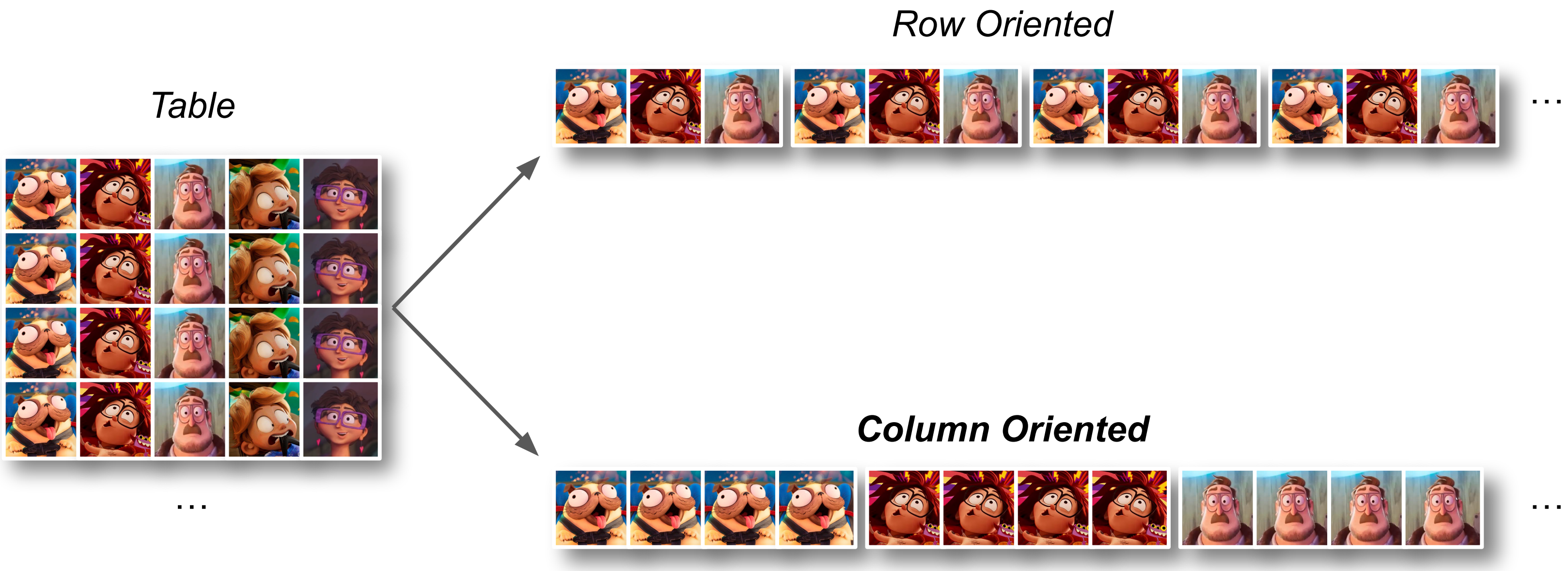
Example of a ML production pipeline



Example of a ML production pipeline



Data physical layout

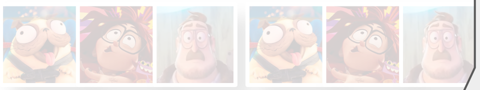


Data physical layout

Table

…

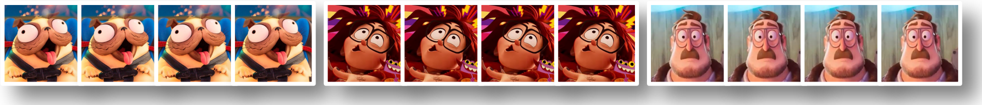
Row Or



Column-oriented layout

- ➔ **Limit IO**
  - ◆ *skip columns*
- ➔ **Save space**
  - ◆ *better compression*
- ➔ **Save time**
  - ◆ *vectorized computation*

Column Oriented



…

Apache Parquet

- ➔ Efficient binary **columnar** format for structured (tabular) data
- ➔ Includes schema (self-describing)
- ➔ Multi-file datasets
- ➔ Supports multiple languages (incl. Python)
- ➔ Open standard

De-facto standard across the industry

for large structured datasets

Parquet

Parquet file format

- ➔ Row group

  Arrow default 64 MB

  - ➔ Column Chunks

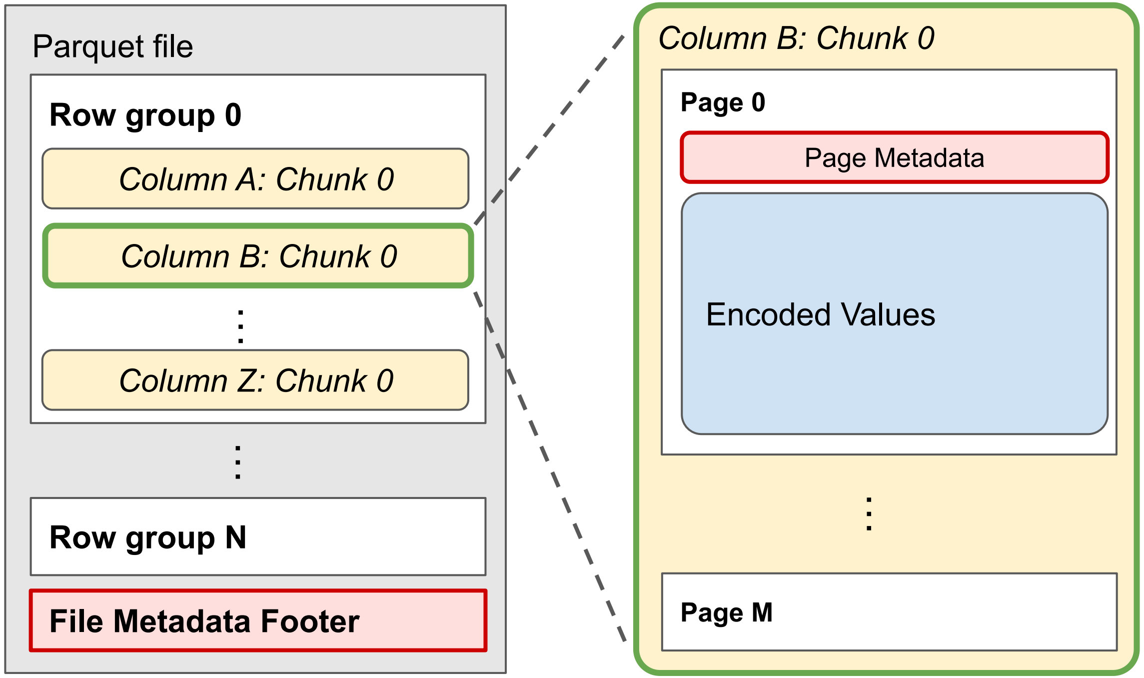
    - ➔ Pages

      Arrow default 1 MB

      Metadata:

      min / max / counts

- ➔ Footer
  - ◆ File stats
  - ◆ Row-group stats & offsets



Partitioning

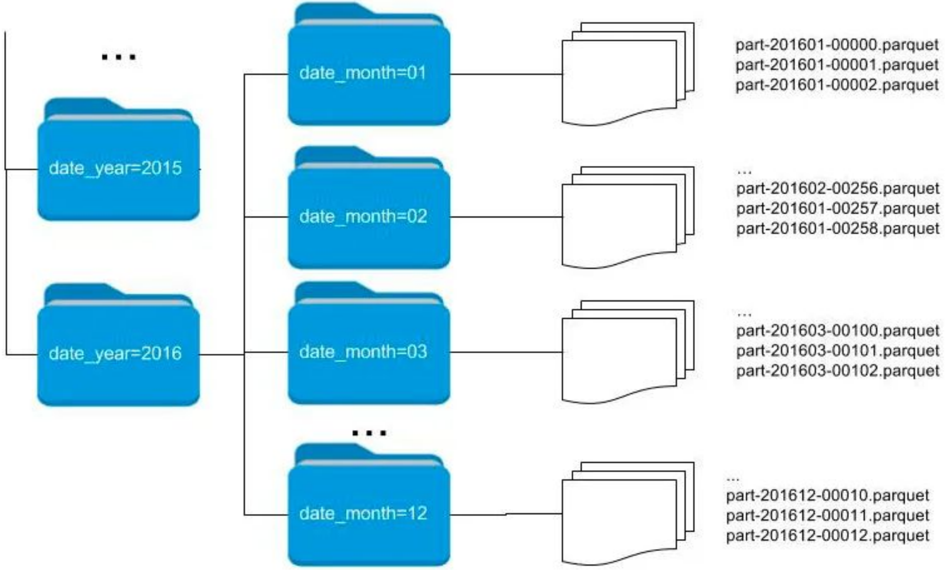


Image source: https://www.datio.com/iaas/understanding-the-data-partitioning-technique/

Tuning

- Parquet file size
  - optimal: ~100 MB
- Too many files
  - Slow “get file list” on cloud stores
  - Reduce columnar benefits
- Row group size
  - default: 64 MB

Apache Arrow

What is it?

- ➔ Language-agnostic in-memory columnar format
- ➔ Serialization and RPC protocol (Flight)
- ➔ Implementations:
  - C++ (with Python bindings, PyArrow)
  - Java + JNI
  - Rust, Go, JavaScript, C#, Julia

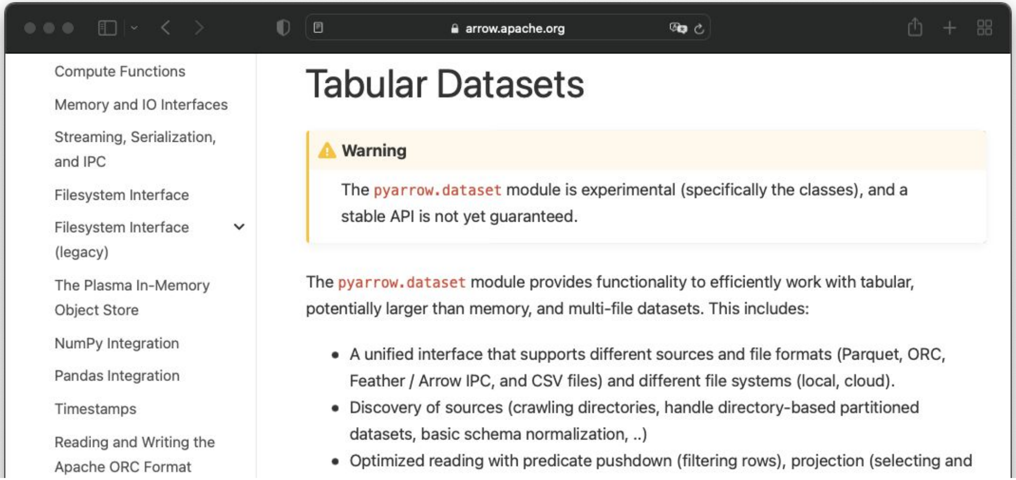
What it does?

- ➔ Zero-copy data sharing across processes
- ➔ Low-overhead serialization-deserialization
- ➔ Parallel RPC-based data transfer
- ➔ Parallel/streaming compute

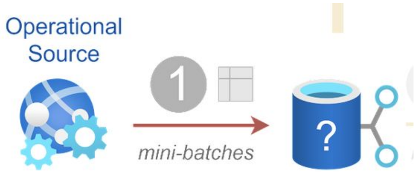
PyArrow Dataset API

- ➔ Wrapper around C++ Dataset API
- ➔ Multi-threaded read/write local or cloud datasets (Parquet, CSV, ORC)
- ➔ Parquet datasets:
  - Streaming read/write/compute
  - Column projection
  - Partition pruning
  - Row groups pruning

https://arrow.apache.org/docs/python/dataset.html

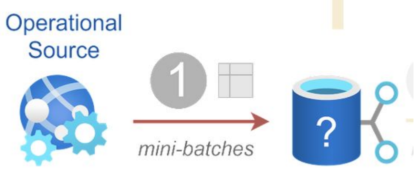


Read a single parquet file



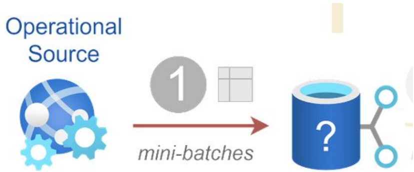

```
import pyarrow.dataset as ds
...
table = (ds.dataset(single_parquet_file, filesystem=input_filesystem)
              .to_table())
...
```

Transform batch




```
...
out_table = process_table(table)
...
```

Transform batch




```
...
out_table = process_table(table)
...
```

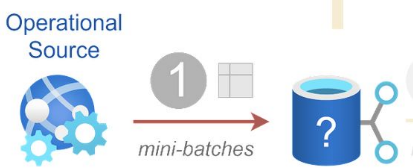
```python
def process_table(table: pa.Table) -> pa.Table:
    df = table.to_pandas()
    dt_series = df['datetime']

    datetime_series = {
        'year': dt_series.dt.year,
        'month': dt_series.dt.month,
        'day': dt_series.dt.day
    }
    for name, series in datetime_series.items():
        array = pa.Array.from_pandas(series)
        field = pa.field(name, array.type)
        table = table.append_column(field, array)
    return table
```

Write Dataset


```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Write Dataset: **input data**

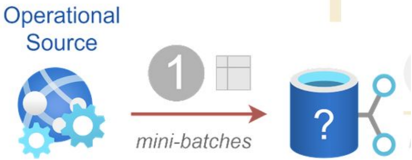



```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Input data options:

- ➔ `pyarrow.Table`
  - ◆ materialized data + schema
- ➔ `Iterator[pa.RecordBatches]`
  - ◆ Non-materialized batches
- ➔ `pyarrow.dataset.Scanner`
  - ◆ Non-materialized batches + schema

Write Dataset: **output location**

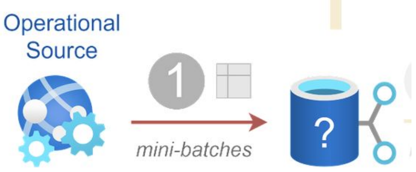



```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Output location options:

- ➔ **Local** fs folder
- ➔ **Cloud** location (AWS S3, Azure, Google)

Write Dataset: **format options**



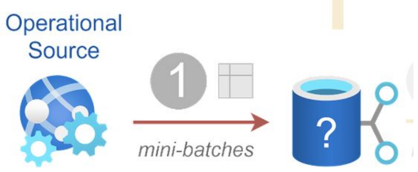

```python
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

```python
parquet = ds.ParquetFileFormat(
    # enable pre_buffer for high-latency filesystems
    # to read more than 1 col chunk per call
    pre_buffer=False,
    # use buffered stream to reduce memory usage
    use_buffered_stream=False, buffer_size=16*1024,
)

write_options = format.make_write_options(
    use_dictionary=True, compression='snappy',
version='2.6')
```

NOTE: also support CSV, ORC, JSON

Write Dataset: **partitioning**

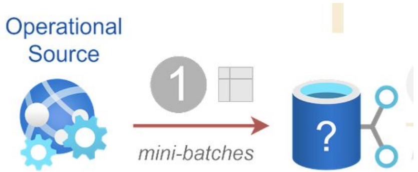



```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Partitioning scheme:

```
year=2022/
        month=1/
                File1.parquet
                File2.parquet
        month=2/
                File1.parquet
                File2.parquet
        month=3/
                File1.parquet
                File2.parquet
        ...
```

Write Dataset: **append** *vs* **overwrite**



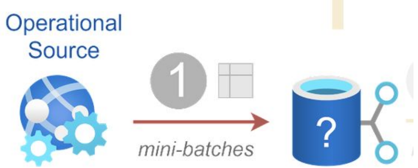

```python
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Write modes:

- ➔ `"error"`
  Any pre-existing data causes an error
- ➔ `"over_write_or_ignore"`
  Append to existing data
- ➔ `"delete_matching"`
  Overwrite any updated partition

https://arrow.apache.org/docs/python/generated/pyarrow.dataset.write_dataset.html

Write Dataset: **file naming**



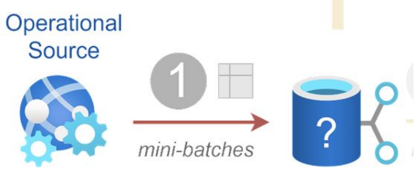

```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

File name is

a random UUID + "a number":

```
...
11c367ee-c26d-4be6-aab7-90ac6b1898b4-0.parquet
...
```

Write Dataset: **monitoring**




```
...
ds.write_dataset(
    out_table,
    base_dir=output_path,
    filesystem=output_filesystem,
    format=parquet,
    file_options=write_options,
    partitioning=['year', 'month'],
    partitioning_flavor='hive',
    existing_data_behavior='overwrite_or_ignore',
    basename_template=f'{uuid4()}-{{i}}.parquet',
    file_visitor=_file_visitor,
)
```

Callback for each written file:

```
def _file_visitor(written_file) -> None:
    path: str = written_file.path
    metadata: pa._parquet.FileMetaData = written_file.metadata
    print(f'VISITOR: {path=}')
    print(f'VISITOR: {metadata=}')
```

Compaction

Multiple file per partition

```
└── year=2020
    ├── month=1
    │   ├── 0e2b77f7-b477-4e39-af40-620634fde181-0.parquet
    │   ├── 47fea6ff-8c97-4999-9b59-9147d5097fcb-0.parquet
    │   ├── 682a1933-c8d6-4e5b-b1e5-35cf735b8984-0.parquet
    │   └── e5ac931e-bf5f-4e5e-ad09-7cd1b24c2079-0.parquet
    ├── month=2
    │   ├── 1dc263ec-881c-4d62-a4ce-fb64c4cfbbe0-0.parquet
    │   ├── 682a1933-c8d6-4e5b-b1e5-35cf735b8984-0.parquet
    │   └── b4ec22c0-0d22-456e-8a59-5cc5a470ae32-0.parquet
    ├── month=3
    │   ├── 3dfa9f8f-941f-470f-b0a8-9b1280d6af0b-0.parquet
    │   ├── 4979833c-1cde-4969-8b91-7ddd5985ece3-0.parquet
    │   ├── 68efe58b-194b-4357-9dcc-4d016174f14d-0.parquet
    │   └── 87437bb6-ee77-4dd9-9b7c-afafe477c574-0.parquet
    └── month=4
        └── 3dfa9f8f-941f-470f-b0a8-9b1280d6af0b-0.parquet
```

compaction

Single file per partition

```
└── year=2020
    ├── month=1
    │   └── d4293db4-e00b-4a41-8c64-4c2a05af9f4a-0.parquet
    ├── month=2
    │   └── d4293db4-e00b-4a41-8c64-4c2a05af9f4a-0.parquet
    ├── month=3
    │   └── d4293db4-e00b-4a41-8c64-4c2a05af9f4a-0.parquet
    └── month=4
        └── d4293db4-e00b-4a41-8c64-4c2a05af9f4a-0.parquet
```

Compaction

```
dataset = ds.dataset(input_path, filesystem=input_fs,
                     partitioning='hive')

scanner = dataset.scanner()

...
ds.write_dataset(
    scanner,
    base_dir=output_path,
    filesystem=output_fs,
    ...
    existing_data_behavior='delete_matching',
)
```

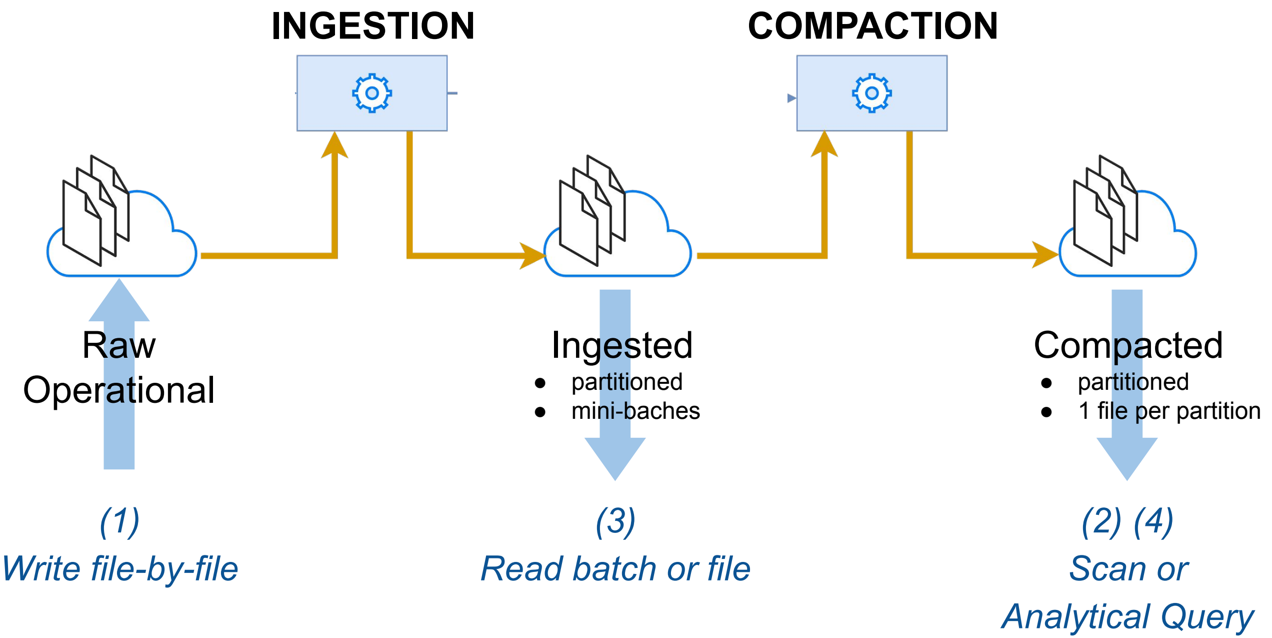
metadata-only

iterator

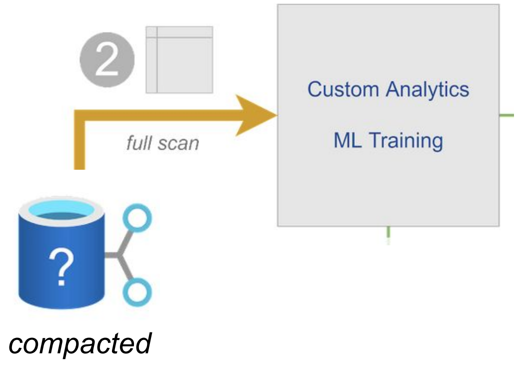
consumes entire dataset in batches

*cloud-based **input** and **output***

Core Architecture



Full Scan Workflow

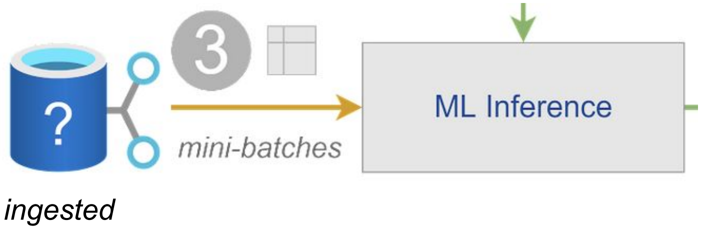


Example: incremental training

```
dataset = ds.dataset(compacted_path, filesystem=fs,
                     partitioning='hive')

model = get_trained_model()
for i, batch in enumerate(dataset.to_batches()):
    df_batch = batch.to_pandas()
    model.incremental_train(df_batch)
```

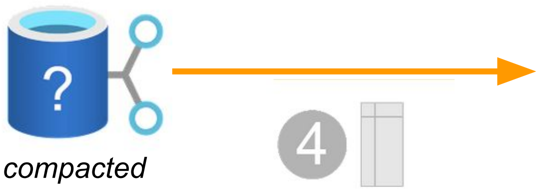
Inference Workflow




```
import pyarrow.dataset as ds
...
df = ds.dataset(single_parquet_file, filesystem=input_filesystem) \
       .to_table() \
       .to_pandas()
...

model = load_trained_model()
prediction = model.predict(df)
```

Analytic Query Workflow




```python
dataset = ds.dataset(path, filesystem=fs, partitioning='hive')

columns = ['datetime', 'cat_col_01', 'num_col_01']  # column projection
filters = pq._filters_to_expression([               # partition + row_group pruning
    ('year',        '=',  2020),
    ('month',       '<',     4),
    ('day',         '<',     5),
    ('cat_col_00', '=', 'foo')
])

table = dataset.to_table(columns=columns, filter=filters)
```

Scaling: parallelization

Vertical scaling

Single node/VM

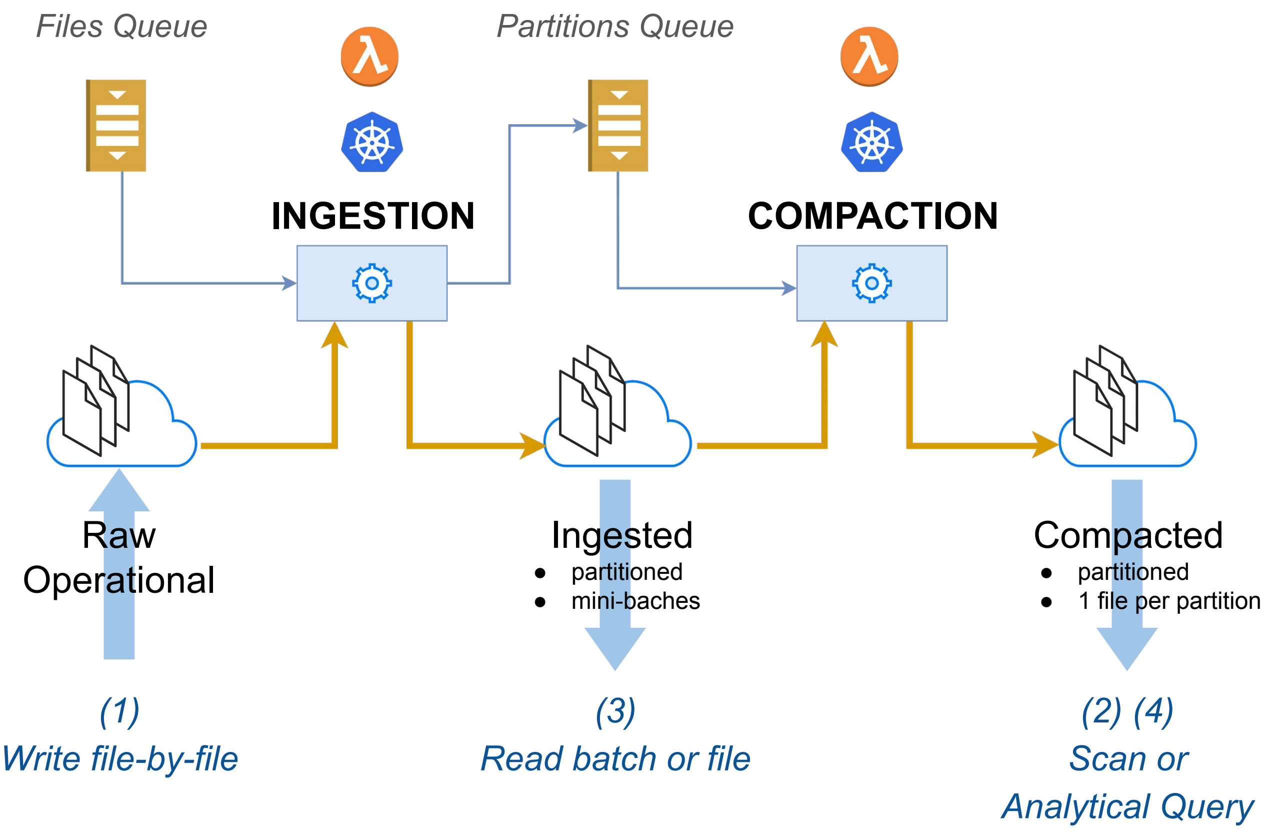
- ➔ multi-threading (Arrow default): saturate CPU, saturate network
- ➔ increase # of cores, RAM, network bandwidth

Horizontal scaling

Distribute across many nodes

- ➔ A “queue” to distribute “jobs”
- ➔ Worker IO: message from queue, read/save to cloud stores
- ➔ Execute: Lambda functions, auto-scaling VM group, Kubernetes

Core Architecture



Open source parquet-based storage

ACID & versioned

- ➔ Delta Lake
  - Java runtime, use in Python via PySpark or Rust bindings
  - Linux Software Foundation, mainly Databricks-driven
- ➔ Iceberg
  - Java runtime
  - Python API: early-stage
  - **Hidden Partitioning**
  - **Schema evolution**
  - Apache Software Foundation, community driven

Arrow Ecosystem

➔ Data Fusion

Query execution framework

- Multi-thread query execution framework
- Rust-based

➔ Ballista

Distributed computation platform

- **Arrow compute** kernel
- **Flight** protocol for inter-process data transfer
- **Data Fusion** for query execution

Conclusions

With **Parquet** and **PyArrow** you can:

*build efficient **analytics workflow** and **ML pipelines***

Features

- ➔ Efficient
- ➔ Works on **local** or **cloud** storage
- ➔ Scales to large datasets
- ➔ Open source & Community-driven

Thank you!

Get runnable code examples:

https://github.com/tritemio/parquet-pyarrow-pyconit-2022