bodo.ai

# Bodo DataFrames:
## a fast and scalable HPC-based drop-in replacement for Pandas

PyData Global, December 11, 2025

**Scott Routledge**
**Software Engineer**
**Bodo.ai**

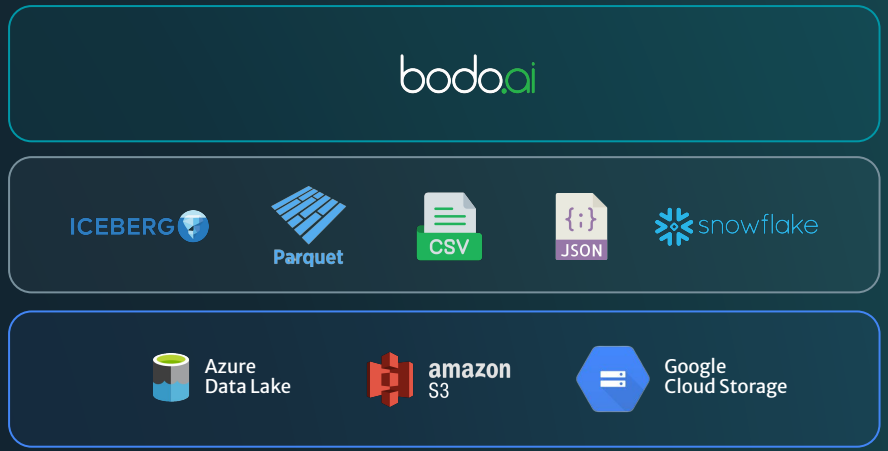
# About Me

- Software Engineer at Bodo.ai
- B.S. in CS and ML @ Carnegie Mellon
- Worked on Python Compiler & BodoSQL
- Currently working on Bodo DataFrames

# bodo.ai

## High Performance Data Processing in Python

- Open source:
  - `pip install bodo`
  - `conda install bodo -c conda-forge`
  - `import bodo.pandas as pd`
- Parallel computing with MPI
- JIT compilation & query optimization
- SQL engine (`pip install bodosql`)



# Agenda

- [x] The Challenge: Simple, Scalable Data Infrastructure
- [x] Bodo DataFrames Overview and Demo
- [x] JIT Compiler for Custom Code
- [x] Integration with Iceberg Table Format

# The Challenge: Simple Scalable Data Infra

**Challenges:**

- ❌ Frequent code rewrites and performance tuning
- ❌ Complexity in managing large cloud datasets
- ❌ Fragmentation of tools for different use cases

**Requirements:**

- ✅ Fast and scalable for data size without code changes
- ✅ Easy to use and understand
- ✅ Flexibility: ETL, analytics, preprocessing, AI workloads (inference, training)

# Bodo DataFrames

- Database-grade query optimizer & JIT compiler
- MPI backend with streaming execution
- Spill to disk to prevent Out of Memory (OOM) Errors
- Fallback to Pandas for unsupported operations

```
import bodo.pandas as pd

df = pd.read_parquet(path)
df["month"] = df.F.dt.month
df["year"] = df.F.dt.year
df = df[df.B == "gamma"]

df.to_parquet("result.pq")
```

# Demo Notebook

NYC Taxi ETL

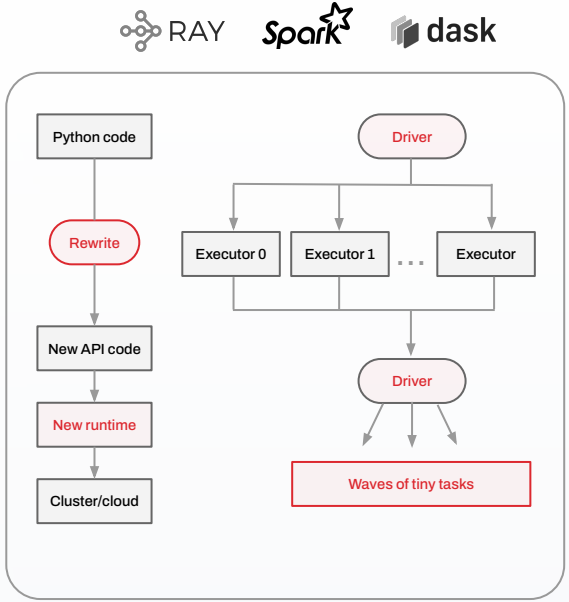
# Bodo Scales Pandas Efficiently

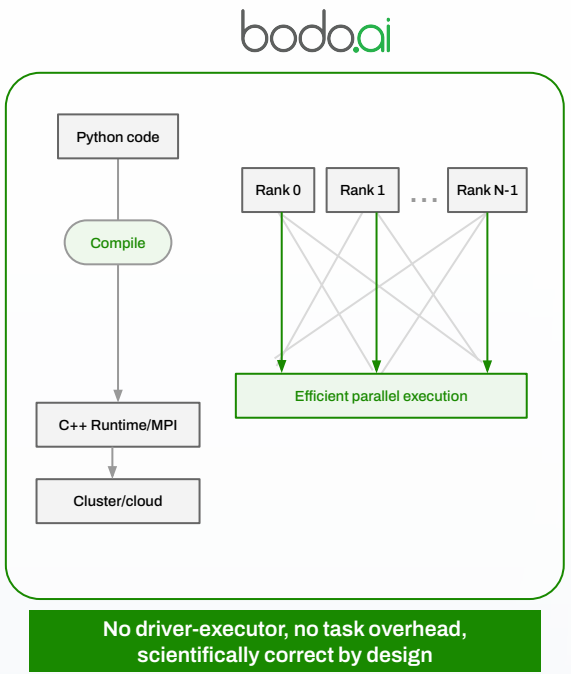
**Could use Spark, Dask, etc.**

- Implementation complexity
- Task-based model creates high overheads

**Or… Use Bodo!**

- Drop-in Pandas replacement
- HPC execution scales from laptop to large cluster
- JIT compilation & query optimization




**Source**: Totoni et al. “A Case Against Tiny Tasks in Iterative Analytics”, HotOS’17

# DataFrames Workflow

## User process

```
import bodo.pandas as pd

df = pd.read_parquet(path)
df["month"] = df.F.dt.month
df["year"] = df.F.dt.year
df = df[df.B == "gamma"]

df.to_parquet("result.pq")
```

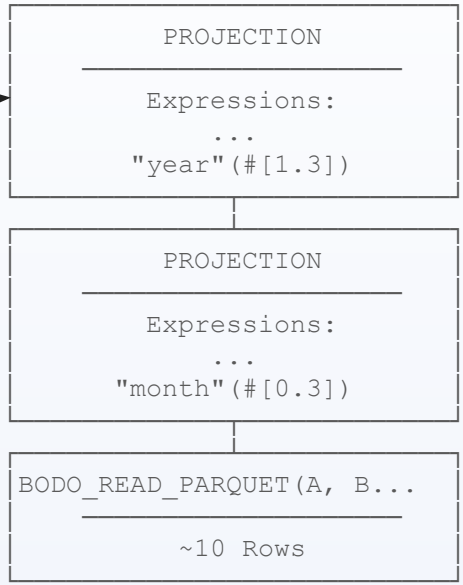
## Generated Plan

# DataFrames Workflow

## User process

```
import bodo.pandas as pd

df = pd.read_parquet(path)
df["month"] = df.F.dt.month
df["year"] = df.F.dt.year
df = df[df.B == "gamma"]

df.to_parquet("result.pq")
```

## Generated Plan

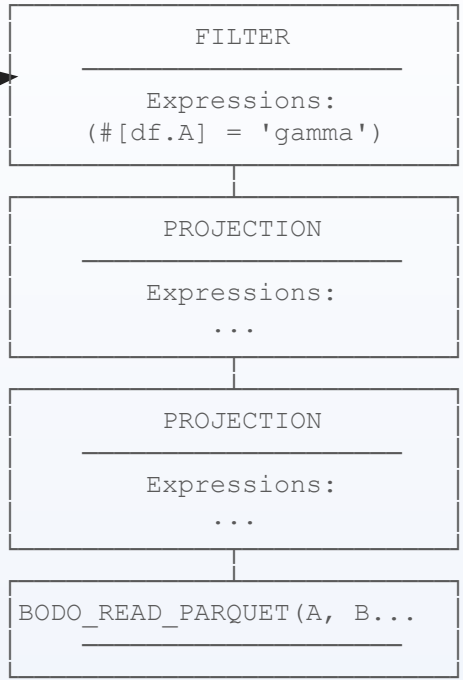


# DataFrames Workflow

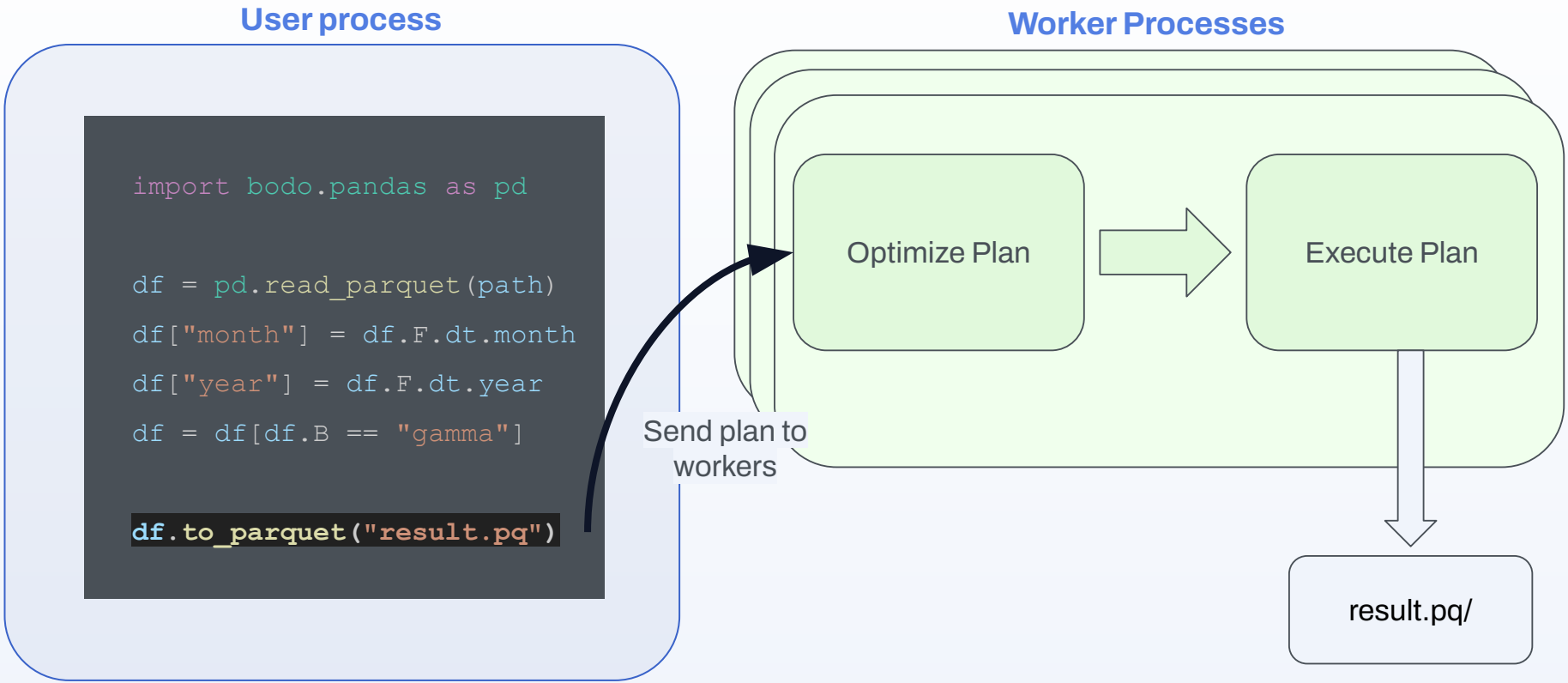
## User process

```
import bodo.pandas as pd

df = pd.read_parquet(path)
df["month"] = df.F.dt.month
df["year"] = df.F.dt.year
df = df[df.B == "gamma"]

df.to_parquet("result.pq")
```

## Generated Plan

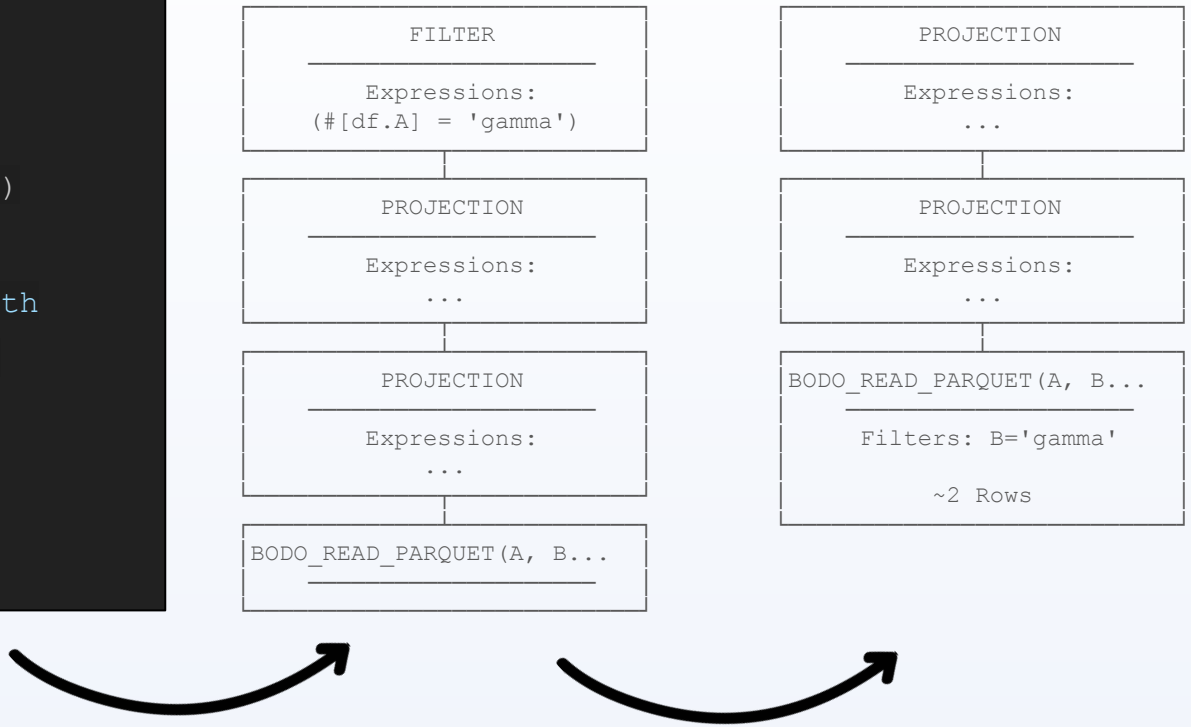


# DataFrames Workflow



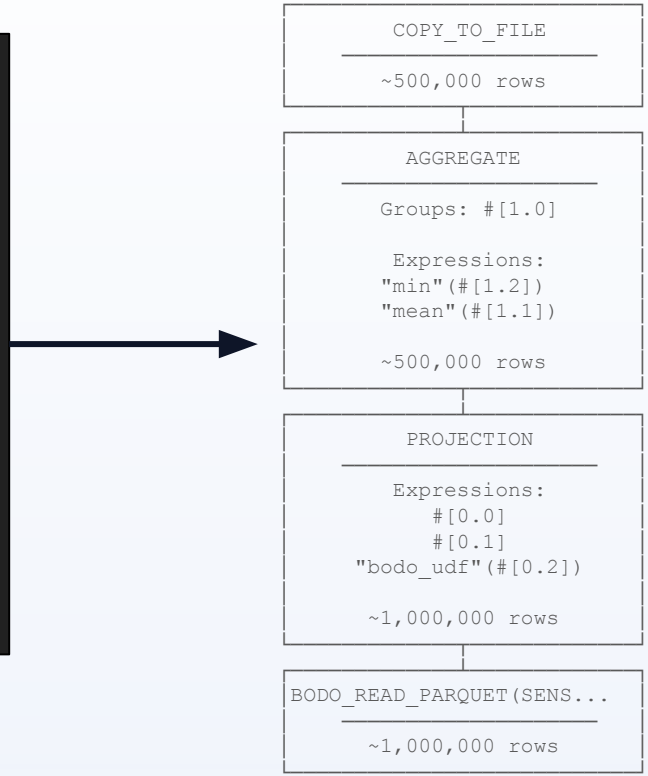
# Plan Optimization: Filter Pushdown

```python
import bodo.pandas as pd

df = pd.read_parquet(path)

df["month"] = df.F.dt.month
df["year"] = df.F.dt.year

df = df[df.B == "gamma"]
```

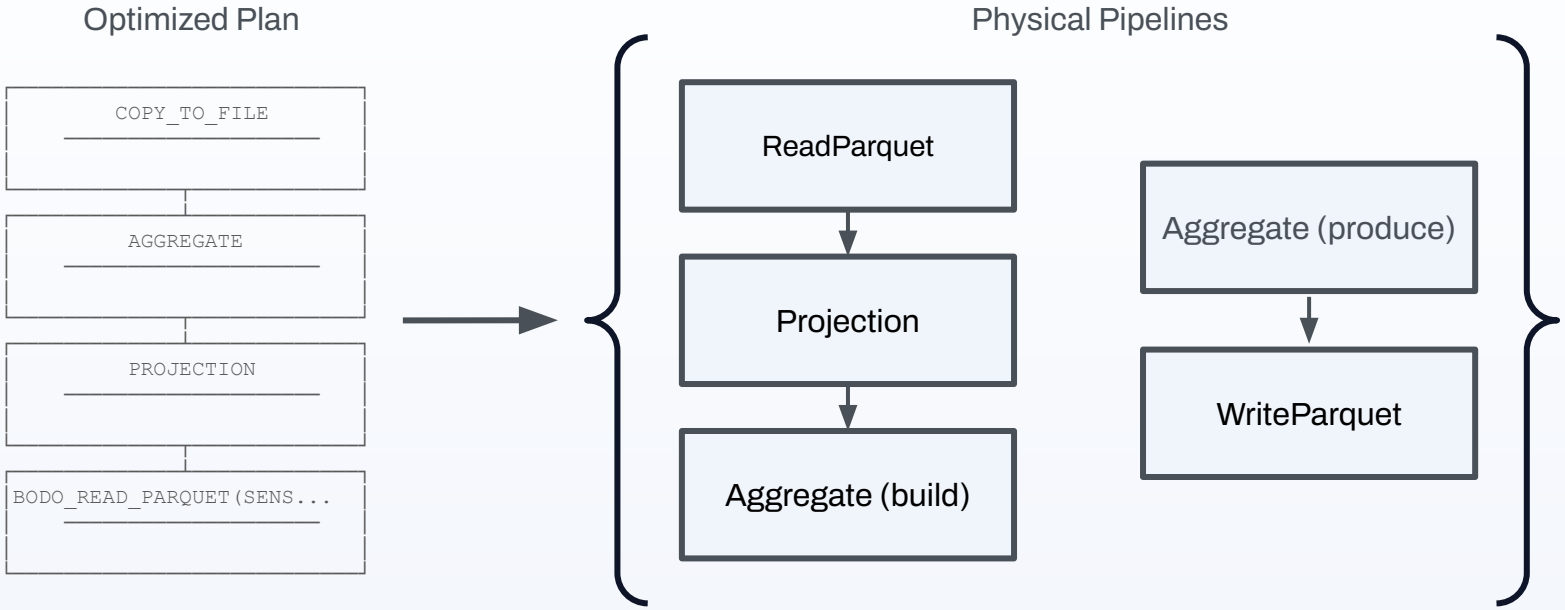


# Vectorized Execution Example

```python
import bodo.pandas as pd

df = pd.read_parquet("raw_data.pq")
df["ts"] = df.ts.dt.ceil("100ms")
df.groupby("sensor_id", as_index=False
).agg({"value": "mean", "ts": "min"})

df.to_parquet("data_agg.pq")
```

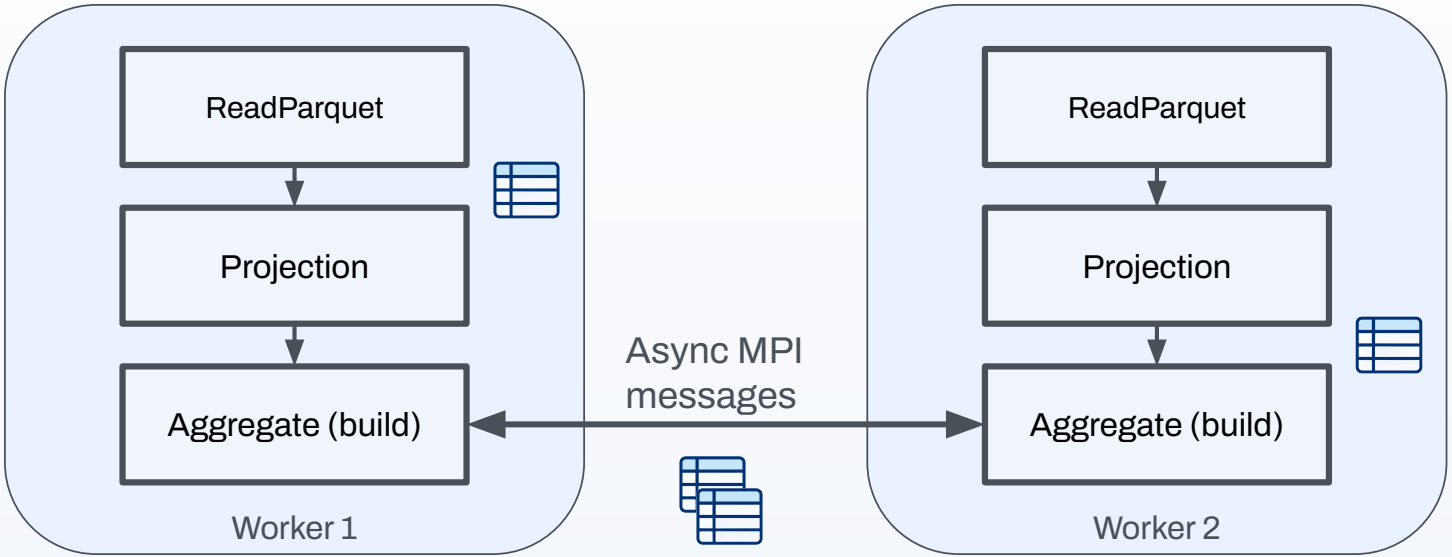


# Vectorized Execution Example




- Logical plan is converted to a series of pipelines
- Push based execution model: data flows from source node(s) to a sink

# Async Messaging in Pipelines




- Workers process pipelines independently
- Async MPI communication can overlap with compute
- Workers sync only at pipeline boundaries

# DataFrame Libraries Comparison

| | Ease of Use | Query Optimizer | Execution |
|---|---|---|---|
| **Bodo DataFrames** | Full Pandas compatibility with fallback | Database-grade optimizer | HPC/MPI and database-based backend |
| **Pandas on Spark** | Pandas compatible; JVM complexity | Spark Catalyst (database-grade) | Task scheduling on JVM |
| **Dask** | Mostly Pandas compatible; explicit ".compute()" and code changes | Basic compute graph optimization | Task scheduling in Python |
| **Modin/Ray** | Pandas-compatible with fallbacks | No optimizer | Slow task scheduling (Ray-based) |
| **Polars** | SQL-like API, not Pandas-compatible | Simple heuristic rules | Single-node threaded task scheduling |
| **Daft** | SQL-like API, not Pandas-compatible | Simple heuristic rules | Ray-based task scheduling (non-vectorized) |

# JIT Compilation for Custom Code

- Bodo DataFrames tries JIT compilation for ser.map / df.apply calls
- Falls back to Python if compilation fails

```
def simulate(row):
    balance = row.start_balance
    for _ in range(row.steps):
        balance += random.randint(-1, 1) * row.drift
    return balance

df["final_balance"] = df.apply(simulate, axis=1)
```

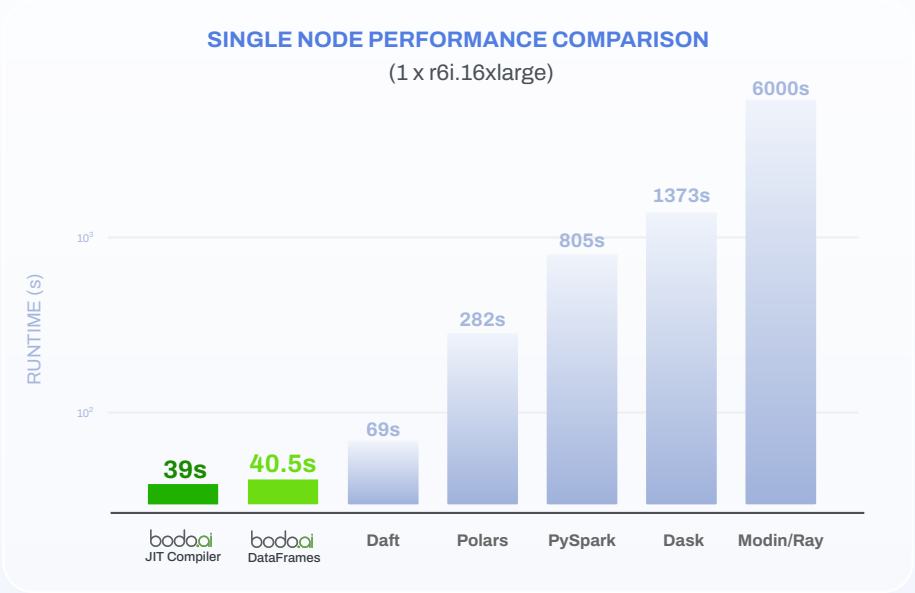
# Bodo JIT Compiler

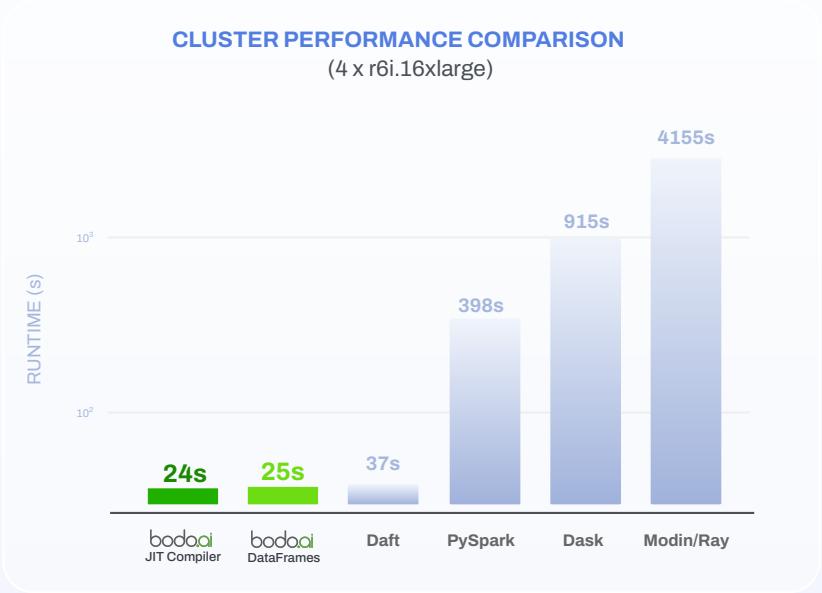
```python
@bodo.jit
def example():
  table = pd.read_parquet('data_parquet')
  data = table[table['A'].str.contains('ABC*',regex=True)]
  stats = data['B'].describe()
  print(stats)
```

- ✅ Compiles Pandas/Numpy to optimized binary with MPI automatically
- ✅ Automatically applied to UDFs of Bodo DataFrames
- ✅ Alternative to Bodo DataFrames for higher performance for custom code
- ❌ Limitations: Compilation delays, unsupported ops, no streaming/spilling support

# Benchmarks

## NYC Taxi (~1B rows)



# Bodo Iceberg Support

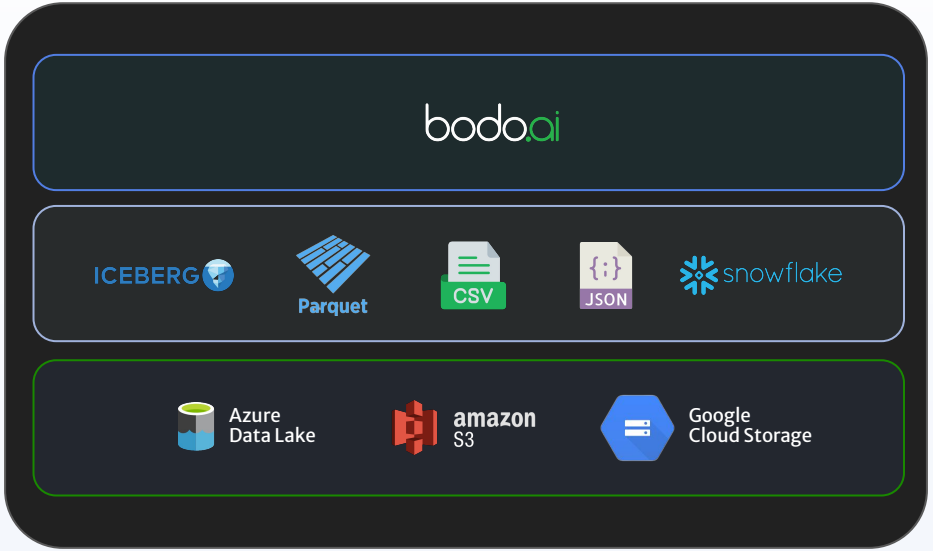
- Supports `read_iceberg` and `to_iceberg` (new Pandas 3.0 APIs)
- Easy set up: single line of Python code to write or read
- Query past versions of datasets via time travel
- Leverages iceberg's metadata for better query planning
- Filter pushdown becomes even faster with Iceberg!

```
import bodo.pandas as pd

df = pd.read_iceberg(
    table_name, location=table_bucket,
    snapshot_id=prev_snapshot
)
filtered_df = df[df.ts > CUTOFF_DATE]
```

# Summary

Bodo provides high-performance data processing in native Python

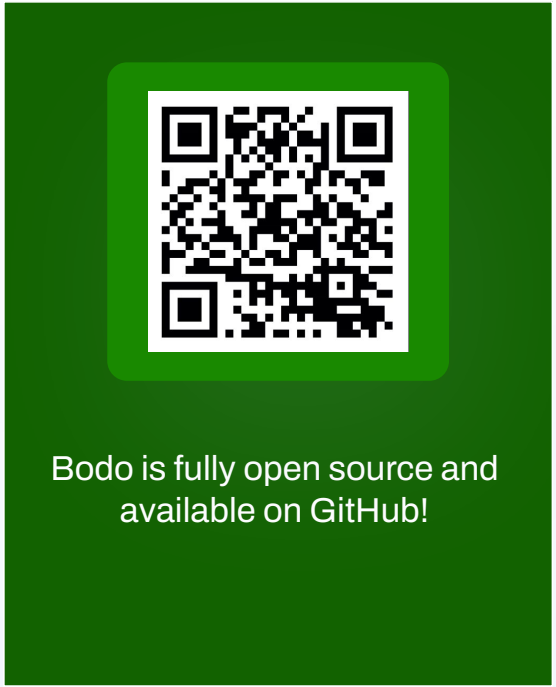
- True parallel computing with powerful MPI backend
- Pandas drop-in replacement
- JIT compiler & query optimizer



# Thank you!

- Questions?
- Try Bodo Yourself 🚀
  - `pip install bodo`

    `conda install bodo -c conda-forge`
  - Join our Slack community
  - Star us on Github



# Supplemental Slides (if time allows)

# Plan Optimization: Join Reordering

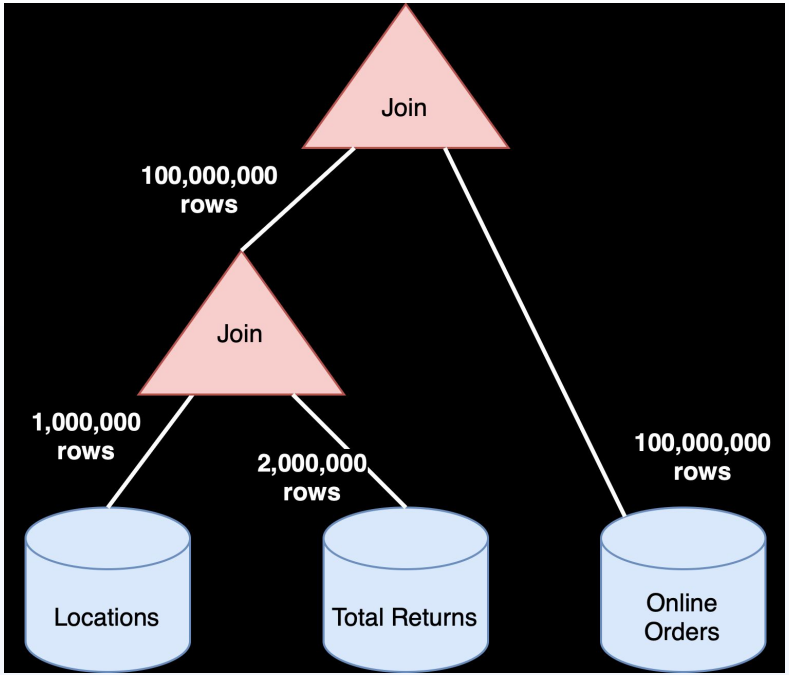
- Joins can be applied in different orders and produce equivalent results
- Join order effects intermediate result sizes
- Bad join order can lead to >100x slower execution times!

```
df1 = returns.merge(online_orders, on="order_id")
df2 = locations.merge(df1, on="customer")
```

```
df1 = locations.merge(returns, on="customer")
df2 = online_orders.merge(df1, on="order_id")
```

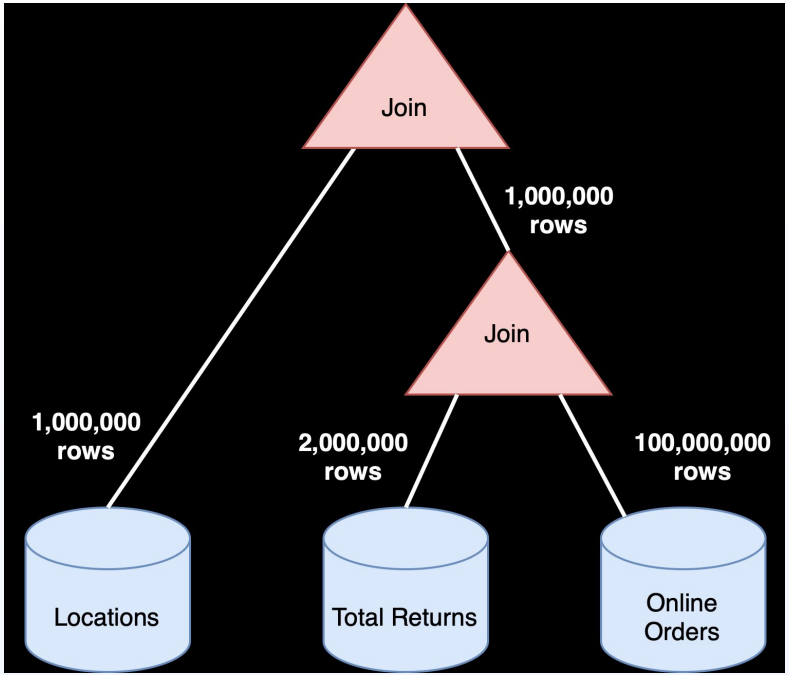
# Plan Optimization: Join Reordering

- ✗ Bad join ordering: intermediate result size explodes to much larger than inputs
- ✗ Difficult to predict: simple heuristics like “join the smaller tables first” don’t work



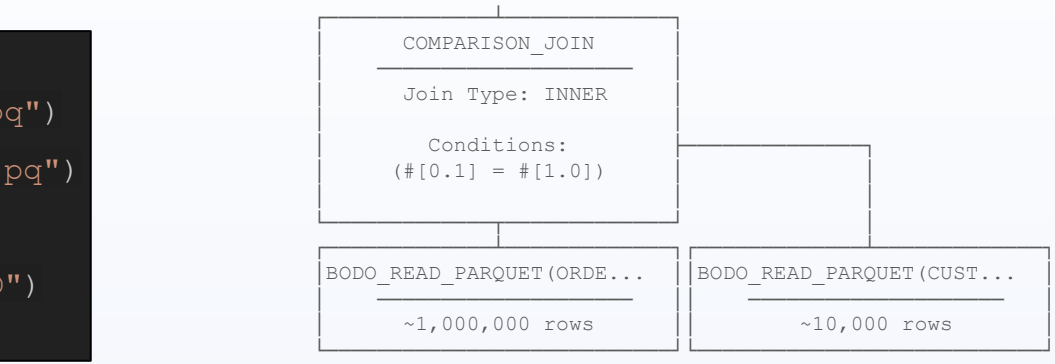
# Plan Optimization: Join Reordering

- Good join ordering: reduce intermediate result size by 100x!
- Planner uses dynamic programming to explore different orderings and picks the best one based on cost estimates



# Plan Optimization: Join Filters

```python
orders = pd.read_parquet("orders.pq")
cust = pd.read_parquet("customers.pq")

orders.merge(cust, on="CUSTOMER_ID")
```



# Plan Optimization: Join Filters

- Use runtime information from build table to filter probe table before the join
- Join filters can be pushed down to reduce intermediate results sizes
- Recently added to Bodo DataFrames