Python & SQL data processing with HPC efficiency
My background

2014
PhD, CS from UIUC with a focus on HPC (programming systems, energy efficiency)

2015
Research Scientist at Intel Labs/CMU working to democratize HPC for data analytics (Julia, Python)

2019-present
Co-founder and CTO at Bodo.ai
Bodo is a high performance **SQL & Python** compute engine with extreme efficiency.

- **50% lower cloud costs**
  Reduce costs of resource-intensive queries that disproportionately consume budgets

- **10X faster performance**
  Faster, more efficient, more reliable — even at PB scale across thousands of cores.

- **Pluggable compute, no changes to workflows**
  No data migration, plugs into current workflow, Snowflake SQL compatible
Simplicity vs. Performance Gap

High-Level Scripting Code

```
df.rolling(9).apply(...)```

Simple, but Slow & Single Core

Low-Level MPI/Fortran Code

```
IF ( id < p - 1 ) THEN
   call MPI_SEND ( u2_local(i_local_hi), 1, &
      MPI_DOUBLE_PRECISION, id + 1, itor,
      MPI_COMM_WORLD, &
      error)
   call MPI_RECV ( u2_local(i_local_hi + 1), 1, &
      MPI_DOUBLE_PRECISION, id + 1, rtol,
      MPI_COMM_WORLD, &
      status, error)
ELSE
   x = 1.0D+00
   u2_local(i_local_hi + 1) = exact ( x, t )
END IF
```
The Challenge

Simplicity \( \rightarrow \) Performance

Pick Two

Generality
The Challenge

Simplicity $\rightarrow$ Performance

Pick Two

Generality

Pick Two

Simplicity $\rightarrow$ Generality

NumPy, pandas
The Challenge

Simplicity (S) - Performance (P) - Generality (G)

Pick Two

- Simplicity
- Performance
- Generality

Symbols:
- MPI
- C++
The Challenge

Simplicity  Halide  Performance

S -- Pick Two -- P

Generality
The Challenge

Simplicity  S  Performance

Pick Two

Generality

“Holy grail” solution:
Automatic compiler parallelization of simple code for general data problems
```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)
```

$ conda install bodo -c bodo.ai -c conda-forge
$ mpiexec -n 224 python ./process_data.py
Bridging Simplicity-Performance Gap

HPC Architecture, Native Python APIs

In-memory Processing

Scalability & Performance

Simplicity

MPI

Apache Spark

Hadoop

Python
Bodo Demo

Notebook Examples
Inferential Compiler Technology

**Ordinary Compiler**

- Human Readable Code
  - Translates to Sequential Code
  - Produces Sequential Machine Code Binary

**bodo.ai Inferential Compiler**

- Human Readable Code
  - Analyzes Code Structure
  - Optimizes Operations
  - Transforms Logic into Fully Parallel Execution
  - Translates to Native Parallel Code
  - Produces Native Parallel Machine Code Binaries
Automatic parallelization challenge

Previous Work
- Analyze loops and memory access patterns in C or Fortran\textsuperscript{1,2}
- Example: use integer programming to explore decision search space\textsuperscript{1}

Addition Challenges of Python:
- Dynamic typing
- Complex data structures like dataframes

Bodo Approach\textsuperscript{3}
- Focus on high-level APIs, not loops
- Treat Pandas/Numpy as deeply embedded DSLs
- High-level API semantics assist with parallelization

1 Kennedy and Kremer, “Automatic data layout for high performance Fortran”, SC’95
3 Totoni et al, “HPAT: high performance analytics with scripting ease-of-use”, ICS’17
Python Function → Bytecode to IR → Numba IR → Dataframe Transform → Numba IR with Dataframe converted → Series Transform

ParallelAccelerator Passes → Numba IR with Series converted

Numba IR with array ops optimized → Distributed analysis/transform → Numba IR with MPI Calls → Efficient MPI Binary

Totoni et al, "HPAT: high performance analytics with scripting ease-of-use", ICS'17
Automatic Parallelization Approach

Exploit analytics program properties:
- High-level Pandas/Numpy operations are implicitly parallel
- Map/reduce + relational parallel patterns
  - One-dimensional block distribution of data and compute
  - “Big” collections are distributed, “small” collections are replicated
- Generate efficient Single Program Multiple Data (SPMD) binary

Data flow compiler algorithm:
- Transfer functions for operations
- Fixed-point iteration converges to optimal solution
Data Flow Framework

**Distribution meet-semilattice**

\[ L = \{1D_B, 2D_{BC}, REP\} \]

\[ REP \leq 2D_{BC} \leq 1D_B \]

\[ \bot = REP, T = 1D_B \]

**meet operator** \( \land \)

\[ D_a : A \rightarrow L \]
\[ D_p : P \rightarrow L \]

**Transfer function for each node type**

- Based on high-level semantics
- Overall program transfer function: \( (D_a, D_p) = F(D_a, D_p) \)
- Solve using fixed-point iteration
- Converges with monotone transfer functions
Transfer Functions

Assignments:
\[ f_{l=r} : D_a(l) = D_a(r) = D_a(l) \land D_a(r) \]

Binary Ops:
\[ f_{l=r_1+r_2} : D_a(l) = D_a(r_1) = D_a(r_2) = D_a(l) \land D_a(r_1) \land D_a(r_2) \]

Join:
\[ f_{\text{Join}(x_1,x_2,...)} : D_a(x_1) = D_a(x_2) = \ldots = D_a(x_1) \land D_a(x_2) \land \ldots \]

Function calls:
\[ f_{c(x_1,x_2,...)} : D_a(x_1), D_a(x_2), \ldots = \text{knownCalls}(c)(D_a(x_1), D_a(x_2), \ldots) \]
\[ f_{\text{unknownCall}(x_1,x_2,...)} : D_a(x_1) = D_a(x_2) = \ldots = \text{REP} \]

1 \[ D_B \]
2 \[ D_B \]
\[ \text{REP} \]
Automatic Parallelism Extraction

\[
\text{df} = \begin{pmatrix}
\text{A} & \text{B} & \text{C} \\
a_0 & b_0 & c_0 \\
a_1 & b_1 & c_1 \\
a_2 & b_2 & c_2 \\
a_3 & b_3 & c_3 \\
a_4 & b_4 & c_4 \\
a_5 & b_5 & c_5 \\
\end{pmatrix}
\]

\[
m = \text{df}.\text{mean}()
\]

\[
m = \begin{pmatrix}
m_A \\
m_B \\
m_C \\
\end{pmatrix}
\]

replicated

1D block distribution
Automatic Distribution

samples

\[
\begin{pmatrix}
  a_0 & b_0 & c_0 \\
  a_1 & b_1 & c_1 \\
  a_2 & b_2 & c_2 \\
  a_3 & b_3 & c_3 \\
  a_4 & b_4 & c_4 \\
  a_5 & b_5 & c_5 \\
\end{pmatrix}
\]

1D block distribution

labels

\[
\begin{pmatrix}
  y_0 & y_1 & y_2 & y_3 & y_4 & y_5 \\
\end{pmatrix}
\]

1D block distribution

w

\[
\begin{pmatrix}
  w_0 & w_1 & w_2 \\
\end{pmatrix}
\]

replicated

P₀

\[
\begin{pmatrix}
  a_0 & b_0 & c_0 \\
  a_1 & b_1 & c_1 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
  y_0 & y_1 \\
\end{pmatrix}
\]

P₁

\[
\begin{pmatrix}
  a_2 & b_2 & c_2 \\
  a_3 & b_3 & c_3 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
  y_2 & y_3 \\
\end{pmatrix}
\]

P₂

\[
\begin{pmatrix}
  a_4 & b_4 & c_4 \\
  a_5 & b_5 & c_5 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
  y_4 & y_5 \\
\end{pmatrix}
\]

w₀ w₁ w₂

w₀ w₁ w₂

w₀ w₁ w₂
Data Parallelism Extraction

\[ D = A \times B + C \]

* Recognize parallelism

**parfor** \( i=0:n \)
\[ t[i] = A[i] \times B[i] \]

**parfor** \( i=0:n \)
\[ D[i] = t[i] + C[i] \]

* Fuse loops

**parfor** \( i=0:n \)
\[ D[i] = A[i] \times B[i] + C[i] \]

Anderson et al. "Parallelizing Julia with a Non-invasive DSL", ECOOP'17
Transfer Function for Parfors

\[
distribution = 1D_B \\
parArrays = \emptyset \\
arrayAccesses = extractArrayAccesses(parfor.body) \\
parforIndexVar = parfor.loopNests[0].index_var
\]

for arrayAccess in arrayAccesses:
  # array is accessed in parallel in parfor (e.g. A[i,j])
  if parforIndexVar == arrayAccess.index_var[0]
    parArrays = parArrays \cup arrayAccess.array
    distribution = distribution \land D_a(arrayAccess.array)
  if parforIndexVar \in arrayAccess.index_var[1:..]
    # make parfor replicated if parfor's last index variable is used
    # in accessing any lower dimension of array (e.g. A[j,i+2])
    distribution = REP

\[
D_p(parfor) = distribution \\
for arr in parArrays:
  D_a(arr) = distribution
\]
TPCxBB Q26 Benchmark

Setup: 125 server nodes
Instance Type: AWS c5n.18xlarge
CPU cores: 4,500
DRAM: 20 TB
Dataset: 2.5TB data in Parquet format

Execution Time(s)
- 0s
- 35s
- 729s

Faster

20.8X Faster
Parallel Execution Models

Python Code

Rewrite

New API code

New Runtime

Cluster/Cloud

Driver

Executor 0

Executor 1

…

Executor

Waves of Tiny Tasks

Driver-Executor Tiny Task Runtime Overheads

Python Code

Compile

Parallel Binary (MPI)

Cluster/Cloud

Rank 0

Rank 1

…

Rank N-1

Few and Efficiently Running Processes

No Driver-Executor, No Task Overhead, Scientifically Correct by Design

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

<table>
<thead>
<tr>
<th>Untypable variable</th>
<th>Unresolvable function</th>
<th>Nonstatic dataframe schema</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if</code> flag1:</td>
<td><code>if</code> flag2:</td>
<td><code>if</code> flag2:</td>
</tr>
<tr>
<td><code>a = 2</code></td>
<td><code>f = np.zeros</code></td>
<td><code>df = pd.DataFrame({'A': [1,2,3]})</code></td>
</tr>
<tr>
<td><code>else:</code></td>
<td><code>else:</code></td>
<td><code>else:</code></td>
</tr>
<tr>
<td><code>a = np.ones(n)</code></td>
<td><code>f = np.ones</code></td>
<td><code>df = pd.DataFrame({'A': ['a', 'b', 'c']})</code></td>
</tr>
<tr>
<td><code>if</code> isinstance(a, np.ndarray):</td>
<td><code>b = f(m)</code></td>
<td><code>b = f(m)</code></td>
</tr>
<tr>
<td><code>doWork(a)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iterative transform-based typing

Transform code to be handled by type inference

- Start
- If typing failed
  - Type Inference
  - Transformations to enable Typing
  - If program could not change
- If program changed
  - Success
  - Failure
Iterative typing

- Example: avoid dataframe schema change by avoiding inplace
- Possible if inplace method post-dominates definitions (very common in practice)

```python
@bodo.jit
def df_func():
    df = pd.read_parquet("example.pq")
    df.rename({"A": "A1"}, axis=1, inplace=True)
    df["A2"] = 1.1
    return df["A1", "A2", "B"]
```

```python
@bodo.jit
def df_func():
    df = pd.read_parquet("example.pq")
    df2 = df.rename({"A": "A1"}, axis=1, inplace=False)
    df3 = df2.assign(A2=1.1)
    return df["A1", "A2", "B"]
```
Dataframe Optimizations

- Normalize and break up dataframe operations to remove unused columns
- Use special IR nodes when necessary

```python
@bodo.jit
def df_func():
    df = pd.read_parquet("example.pq")
    df2 = df.rename({"A": "A1"}, axis=1, inplace=False)
    df3 = df2.assign(A2=1.1)
    return df["A1", "A2", "B"]
```

```python
A,B,C = ReadParquet(["A", "B", "C"])
df = init_dataframe(("A", "B", "C"), (A, B, C))
A2 = np.full(len(df2), 1.1)
```
A,B,C = ReadParquet(["A", "B", "C")
df = init_dataframe(("A", "B", "C"), (A, B, C))
A2 = np.full(len(df2), 1.1)
return init_dataframe(("A", "A2", "B"), (A, B, A2))

A,B,C = ReadParquet(["A", "B", "C")
df = init_dataframe(("A", "B", "C"), (A, B, C))
df2 = init_dataframe(("A", "B", "C"), (A, B, C))
A2 = np.full(len(df2), 1.1)
return init_dataframe(("A", "A2", "B"), (A, B, A2))

A,B = ReadParquet(["A", "B")
A2 = np.full(len(A), 1.1)
return init_dataframe(("A", "A2", "B"), (A, B, A2))

A,B = ReadParquet(["A", "B")
A2 = np.full(len(A), 1.1)
return init_dataframe(("A", "A2", "B"), (A, B, A2))
Distributed Transform

@bodo.jit
def func(n):
    A = np.arange(n)
    return A, A.sum()

A = np.empty(n)
parfor(i=0; i<n; i+=1):
    A[i] = i
    s += i
return A, s

rank = get_mpi_rank()
n_pes = get_mpi_size()
chunk_size = n/n_pes
A = np.empty(chunk_size)
start = rank*chunk_size
end = (rank+1)*chunk_size
parfor(i=start; i<end; i+=1):
    A[i] = i
    s += i
s = mpi_allreduce(s)
return A, s
Iterative typing - constant signature

- Some values are required to be constants for typing
- Force constant signature if necessary (specialized function version)

```python
@bodo.jit
def f(df, c):
  return df[c].sum()
f(df, "A")
```

```python
@bodo.jit
def f_A(df, c="A"):  
  return df["A"].sum()
```
Iterative typing - constant inference

Replace expressions with constants if necessary

```python
@bodo.jit
def f():
  df = pd.read_parquet(...)
  return df.groupby(list(set(df.columns) - set(['A', 'C'])).sum()
```

```python
@bodo.jit
def f():
  df = pd.read_parquet(...)
  return df.groupby(['B', 'D']).sum()
```
Iterative typing - loop unrolling

Loops over columns need to be unrolled for typing

for c in ["A1", "A2", "A3"]:  
    df[c] = df[c] + 1

df["A1"] = df["A1"] + 1  
df["A2"] = df["A2"] + 1  
df["A3"] = df["A3"] + 1
Filter Pushdown Optimization

- Optimizations in general code are harder than SQL
- Example: compiler transform for filter pushdown

@bodo.jit
def read_pq(filename, s, c):
    df = pd.read_parquet(filename)
    df = df[pd.to_datetime(df["time"])] >= pd.to_datetime(s) & 
    df["count"].astype(int) > c
    ...

@bodo.jit
def read_pq(filename, s, c):
    v1 = pd.to_datetime(s)
    df = ReadParquet(filename, [["time", ">=", v1], 
    "count", ">", c])
    ...

Pattern Matching Optimizations

• Many common analytics code patterns can be optimized
• Example: compare values of string array in place

Packed string array representation:
[“abc”, “bc”, ”cd”]

[0, 3, 5, 7]
[“a”, “b” “c”, “a”, “b”, “c”, ”d”]

```
parfor(I=0; I<n; I++)
    If A[I] == “abc”:
       s += 1
```

```
parfor(I=0; I<n; I++)
    If str_inplace_eq(A, I, “abc”):
       s += 1
```
Pattern Matching Optimizations

Fuse operations to avoid intermediate data

\[
\begin{align*}
S2 &= S.str.split(\text{"","}) \\
S3 &= S2.explode() \\
\end{align*}
\]

\[
S3 = \text{str\_split\_explode}(S)
\]
Data Structure Optimization

Example: “str.split” creates many string and list objects:

Bodo changes data structures transparently:

Array of string objects

```
'A1;B1;C1'
'A2;B2;C2'
'A3;B3;C3'
...
```

A.str.split(';;')

Array of lists of string objects

```
['A1', 'B1', 'C1']
['A2', 'B2', 'C2']
['A3', 'B3', 'C3']
...
```

Packed string array

```
0, 9, 18, 27, ...
```

A.str.split(';;')

Packed array of string lists

```
0, 6, ...
```

List offsets

```
0, 2, 3, 5, 6, 8, 8, 10, ...
```

A str.split(';;;;')

```
A1;B1;C1A2;B2;C2A3;B3;C3
```
The “big data” approach

- Library with map/reduce APIs, distributed system backend
  - Driver/executor task scheduling
  - Complex, much slower than HPC, not scalable
- Distributed systems approach not fit for parallel computing
  - Heterogeneous components with unreliable network assumption is wrong
Python vs. Spark/SQL

**Python**

- Simple imperative code for compute functions
  
  ```python
  df.apply(lambda r: 1 if r.A == "AA" else 0, axis=1)
  ```

- Step-by-step code for data processing
  
  ```python
  df3 = df1.merge(df2, ...)
  ....
  df5 = df4.groupby("A").min()
  df6 = df5.sort_values(by="B")
  ...  
  ```

**Spark**

- Exposes data partitions
- Lazy evaluation wrappers for SQL
  
  ```scala
  sdf.withColumn("B", F.when(sdf["A"] == "AA", 1).otherwise(0))
  ```

**SQL**

- Long declarative expression
  
  ```sql
  SELECT MIN(B) ...
  FROM (  
    SELECT ...
    FROM TABLE1  
    INNER JOIN TABLE2 ...
    LEFT OUTER JOIN TABLE3 ...
    WHERE ...
    UNION ALL  
    SELECT...
  )
  GROUP BY ...
  ORDER BY ...
  WHERE ...
  ```
Careers in Compilers

- Traditional low-level compiler opportunities (hardware enabling, optimization)
  - Hardware companies, DL startups
- New high-level compiler opportunities
  - Compiler-based products for data infrastructure, etc.
  - Configuration languages (e.g. Terraform)
  - SQL-like query optimization
  - Bodo!
Careers in Compilers

• Technical skills for Bodo platform development
  • Compilers, scripting languages
  • Parallel computing, computer architecture
  • Databases, data systems, analytics computing
  • Distributed systems
  • CMU DB Bodo seminar: https://www.youtube.com/watch?v=DJ1sGQnyoAc
  • Data Engineering Podcast: https://www.dataengineeringpodcast.com/bodo-parallel-data-processing-python-episode-223/

• Non-technical skills
  • Understand target users, product, business model
  • Effective collaboration with technical and non-technical colleagues
  • CMU Swartz Entrepreneurship Bodo talk: https://www.youtube.com/watch?v=ofgRijReggw&t=701s
Conclusion

• Automatic parallelization compiler approach bridges simplicity-performance gap
• Compiler-based parallelism superior to distributed systems approach (much simpler & faster)
The “two-language” problem

The Problem:
- Python is dominant for data science, ML/AI
- SQL is a commonly used DSL for data processing
- Many data applications are a mix of the two languages

Challenges:
- Difficult to develop and deploy
- Lack of error-checking and optimizations
- Hard to scale end to end
- Developer skill mismatch

Our Solution: BodoSQL
import pandas as pd
import bodo
import bodosql

@bodo.jit
def f(data_folder):
    df1 = pd.read_parquet(data_folder + "store_sales.pq")
    df2 = pd.read_parquet(data_folder + "item.pq")
    bc = bodosql.BodoSQLContext({'store_sales': df1, 'item': df2})
    sale_items = bc.sql("select * from store_sales join item on store_sales.ss_item_sk=item.i_item_sk")
    count = sale_items.groupby(\"ss_customer_sk\")[\"i_class_id\"].agg(lambda x: (x == 1).sum())
    return count

print(f("s3://my_bucket/my_data"))
import pandas as pd
import bodo
import bodosql

@bodo.jit
def f(data_folder):
    df1 = pd.read_parquet(data_folder + "store_sales.pq")
    df2 = pd.read_parquet(data_folder + "item.pq")
    bc = bodosql.BodoSQLContext({"store_sales": df1, "item": df2})
    sale_items = bc.sql("select * from store_sales join item on store_sales.ss_item_sk=item.i_item_sk")
    count = sale_items.groupby("ss_customer_si")["i_class_id"].agg(lambda x: (x == 1).sum())
    return count

# ERROR: Invalid Key ['ss_customer_si'] in DataFrame
print(f("s3://my_bucket/my_data"))
@bodo.jit
def f(ss_file, i_file):
    df1 = pd.read_parquet(ss_file)
    df2 = pd.read_parquet(i_file)
    bc = bodosql.BodoSQLContext({
        "store_sales": df1,
        "item": df2
    })
    sale_items = bc.sql("select * from store_sales join item on store_sales.ss_item_sk=item.i_item_sk")
    count = sale_items.groupby("ss_customer_sk")["i_class_id"].agg(lambda x: (x == 1).sum())
    return count

def impl(store_sales, item):
    df1 = store_sales.merge(item,
                            left_on="ss_item_sk",
                            right_on="i_item_sk",)
    return df1
BodoSQL: performance

TPC-H Query 10 Execution Time

 Setup: 4 server nodes
 Instance Type: AWS r5n.24xlarge
 CPU cores: 192
 Dataset: 1TB data in original TPC file format.

- BodoSQL: 10s
- Bodo: 9s
- SparkSQL: 38s
@bodo.jit
def read_pq():
    df = pd.read_parquet('example.pq')
    ...

@bodo.jit
def read_csv():
    df = pd.read_csv('example.csv')
    ...

@bodo.jit
def read_sql():
    df = pd.read_sql('select * from employees', 'snowflake://')
    ...

Dataset

Chunk 0  Chunk 1  Chunk 2  …  Chunk N-1
Distributed Fetch from Snowflake

- Submitting one query per core can overwhelm data warehouses
- New distributed fetch connector of Snowflake can prepare data in Arrow format with a single query
Efficient Parallel Join

- Main bottleneck: shuffling data (even with MPI)
- Solution: use Bloom filters to reduce shuffle data (cost vs. saving tradeoff)
- Implementation efficiency is critical:
  - Cache-friendly & SIMD
  - Topology-aware union reductions
- Heuristics to set parameters, e.g. table cardinality (HyperLogLog)
The Case Against Broadcast Join

- Broadcast join is often used to avoid shuffle
- Our results: broadcast join is rarely faster
- Broadcast join is inherently non-scalable
  - Parallel work and memory increases with number of cores:
    - $W_p \sim p \times |T_l|$
    - $M_p \sim p \times |T_l|$
- Bloom filters are scalable
Efficient Parallel Sort

- Partition-based sorting: number of samples vs. load balance tradeoff
  - Maximum load: $N(1 + \epsilon)/p$
  - Total samples: $\Theta(p \log N/\epsilon^2)$
  - Gather/broadcast/all-to-all efficiency critical

TeraSort Results

<table>
<thead>
<tr>
<th>Setup</th>
<th>125 server nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance Type</td>
<td>AWS c5n.18xlarge</td>
</tr>
<tr>
<td>CPU cores</td>
<td>4,500</td>
</tr>
<tr>
<td>DRAM</td>
<td>20 TB</td>
</tr>
<tr>
<td>Dataset</td>
<td>4TB data in Parquet format</td>
</tr>
</tbody>
</table>

Execution Time(s)

- Spark: 797s
- bodoo: 92s

8X Faster
Bodo vs. Others using TPC-H

- “Speed” and “scale” claims without evidence, or overly simplistic benchmarks
- TPC-H benchmarks are standard, but not too complex
- Open-source translation into Python: [https://github.com/Bodo-inc/Bodo-examples/](https://github.com/Bodo-inc/Bodo-examples/)

**TPC-H Q18: Large Value Customer Query**

```python
@bodo.jit
def q18(lineitem, orders, customer):
    gb1 = lineitem.groupby("L_ORDERKEY", as_index=False)["L_QUANTITY"].sum()
    fgb1 = gb1[gb1.L_QUANTITY > 300]
    jn1 = fgb1.merge(orders, left_on="L_ORDERKEY", right_on="O_ORDERKEY")
    jn2 = jn1.merge(customer, left_on="O_CUSTKEY", right_on="C_CUSTKEY")
    gb2 = jn2.groupby(["C_NAME", "C_CUSTKEY", "O_ORDERKEY", "O_ORDERDATE", "O_TOTALPRICE"],
                     as_index=False)
    total = gb2.sort_values(["O_TOTALPRICE", "O_ORDERDATE"], ascending=[False, True])
    print(total.head(100))
```

```sql
select c_name, c_custkey, o_orderkey, o_orderdate,
    o_totalprice, sum(l_quantity)
from customer, orders, lineitem
where o_orderkey in (select l_orderkey from lineitem group by l_orderkey
    having sum(l_quantity) > 300)
    and c_custkey = o_custkey and o_orderkey = l_orderkey
    group by c_name, c_custkey, o_orderkey,
    o_orderdate, o_totalprice
order by o_totalprice desc, o_orderdate
limit 100
```
Bodo vs. Spark (TPC-H)

- TPC-H SF1000 (~1 TB) on 16 c5n.18xlarge AWS (576 physical cores, 3 TB memory), EFA
- Bodo 23x faster than Spark on average (up to 65x), ~90% infrastructure cost saving
Bodo vs. Dask (TPC-H)

- Dask couldn’t load 1 TB dataset, switched to 100 GB dataset
- Bodo is 150x faster Dask on average (up to 500x, excluding Dask failure)
Bodo vs. Ray/Modin (TPC-H)

- Ray/Modin ran out of memory for any dataset >10 GB
- Ray was originally built for reinforcement learning, data engineering support not ready yet
- Task scheduling systems like Dask & Ray can achieve at most Spark performance
Resilience in Data Analytics

- Finish computation ASAP in presence of failures
- Distributed system “continuous availability” not necessary
- Analytics applications are HPC problems (same techniques apply)
- Sources of random failure:
  - Software: e.g Spark/JVM errors
  - Hardware: e.g failing power supply
Resilience in Spark

- RDD/DataFrame lineage used to recompute lost idempotent tasks

- Issues:
  - Communication dependencies across processors
  - Driver ends up restarting the whole computation
  - Manual RDD checkpointing necessary (cache/persist calls)

Spark’s resilience model is actually a checkpoint/restart model!
Resilience in Bodo

• Bare metal execution & SPMD avoids software failures
• Faster execution -> runtime much lower than cluster MTBF
• Easy to integrate with middleware like Kubernetes
• No need to worry about failures in practice
  • E.g. 32-node cluster MTBF is much longer than 2 hour jobs

Bodo approach provides high resilience while reducing complexity