bodo.ai

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

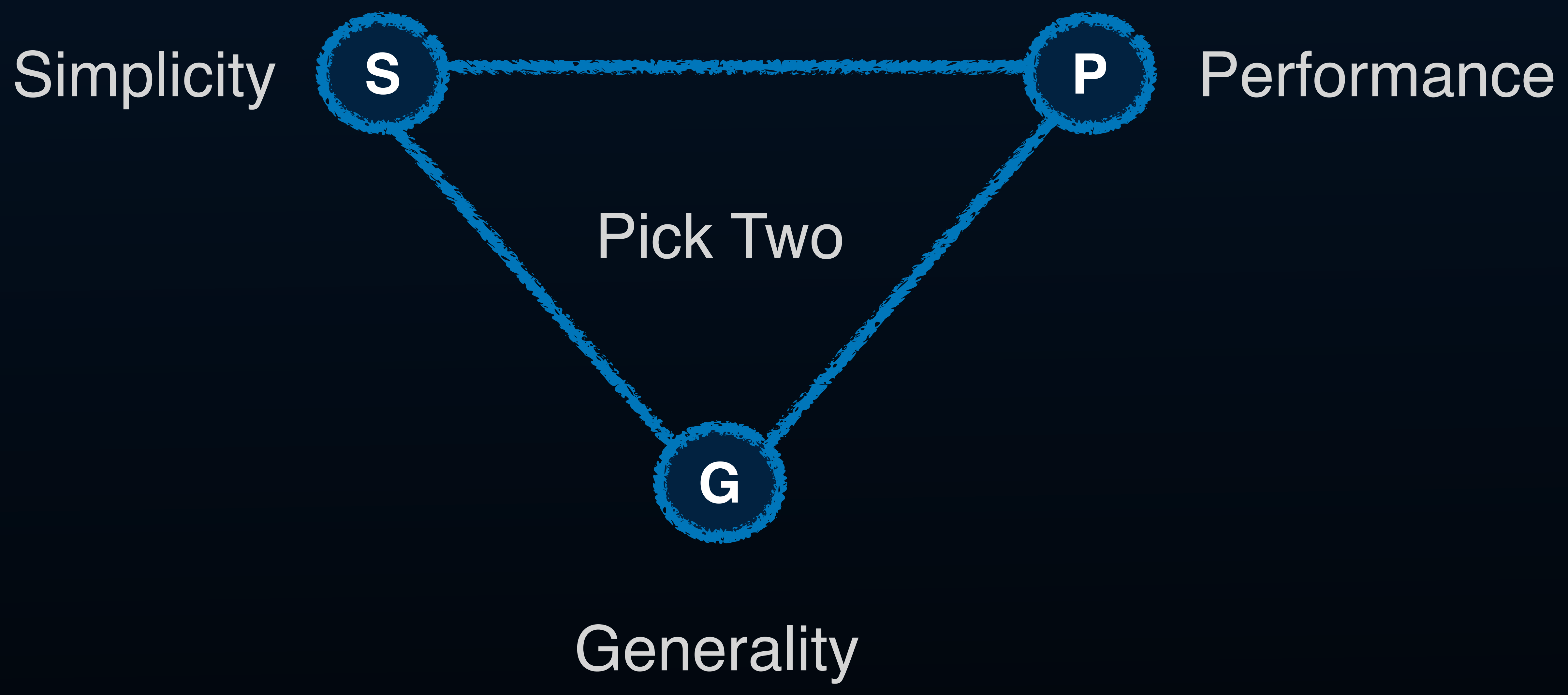
Low-Level MPI/Fortran Code

```
IF ( id < p - 1 ) THEN
    call MPI_SEND ( u2_local(i_local_hi), 1, &
     MPI_DOUBLE_PRECISION, id + 1, ltor,
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
```

Fast & Scalable, but Complex

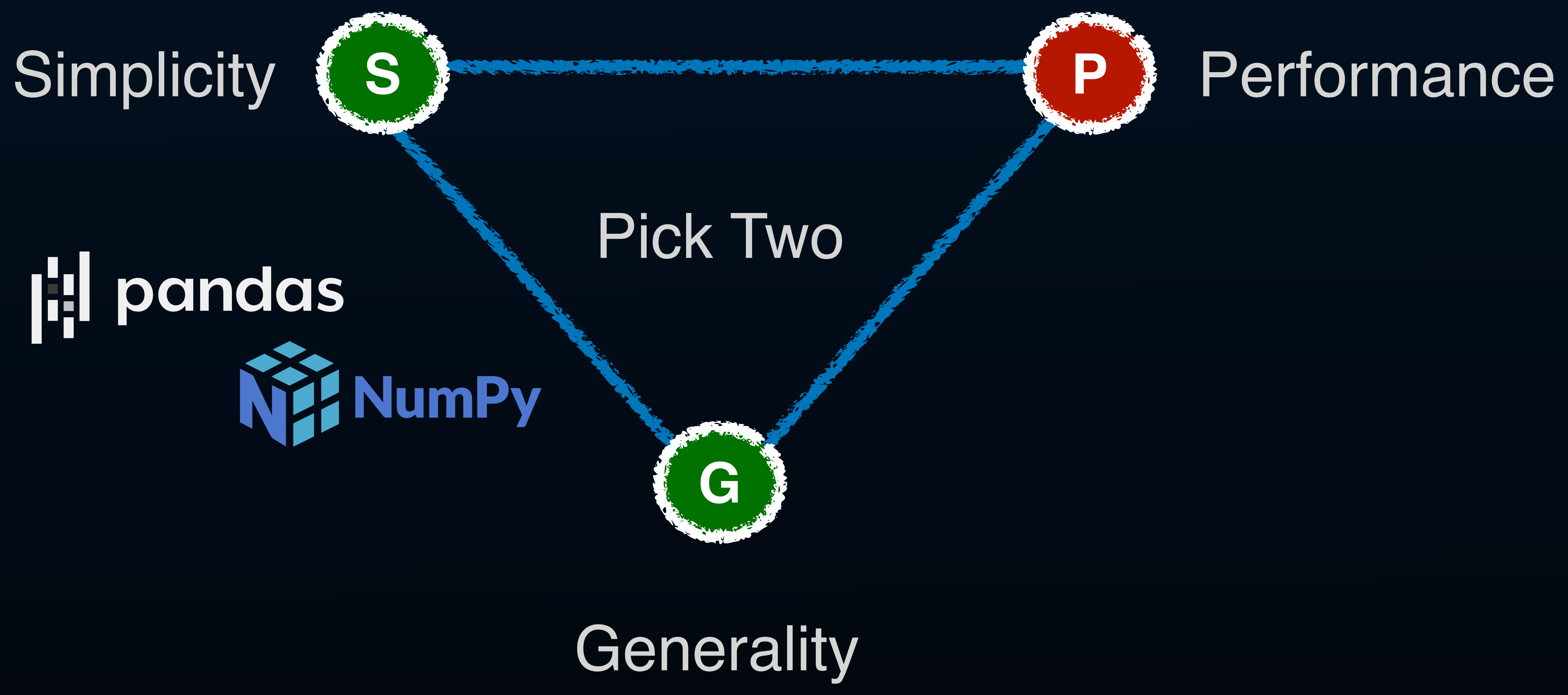
bodo.ai

The Challenge

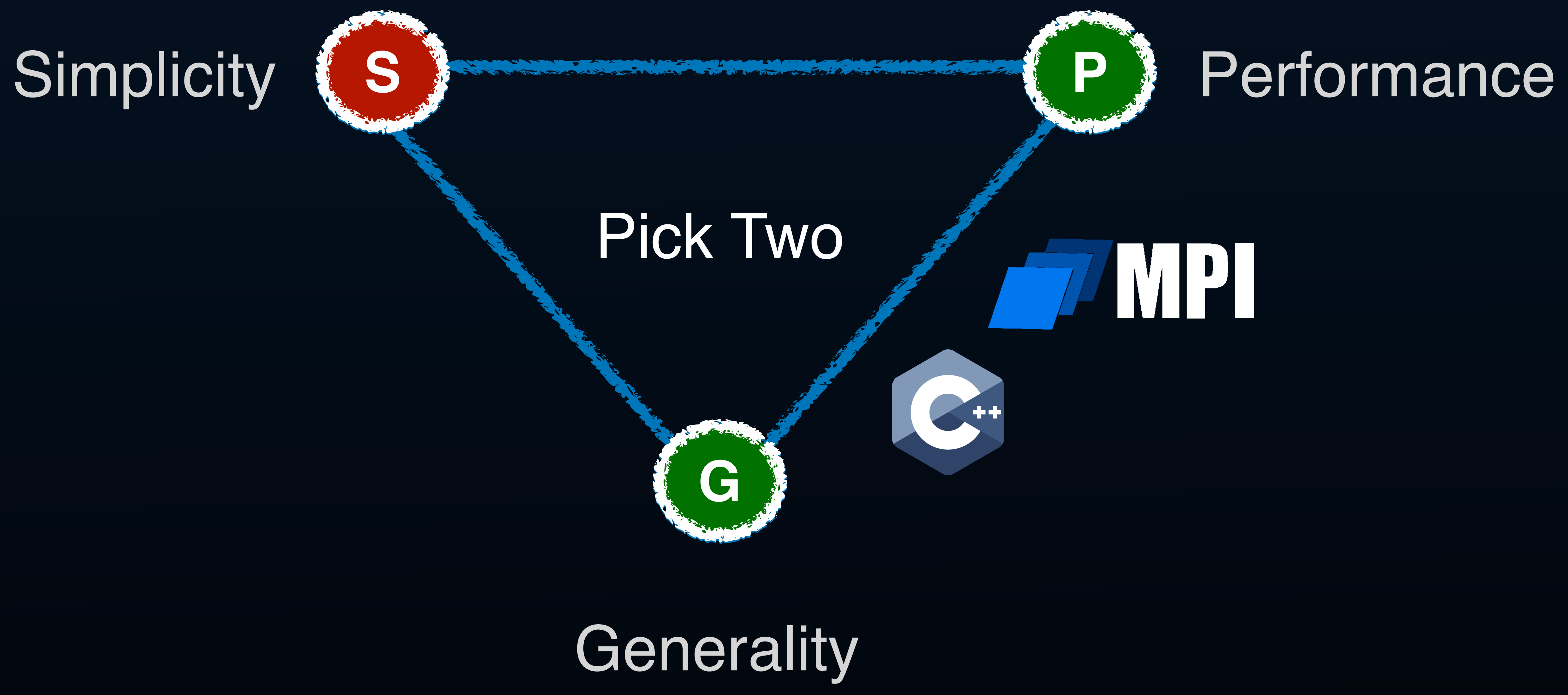


bodo.ai

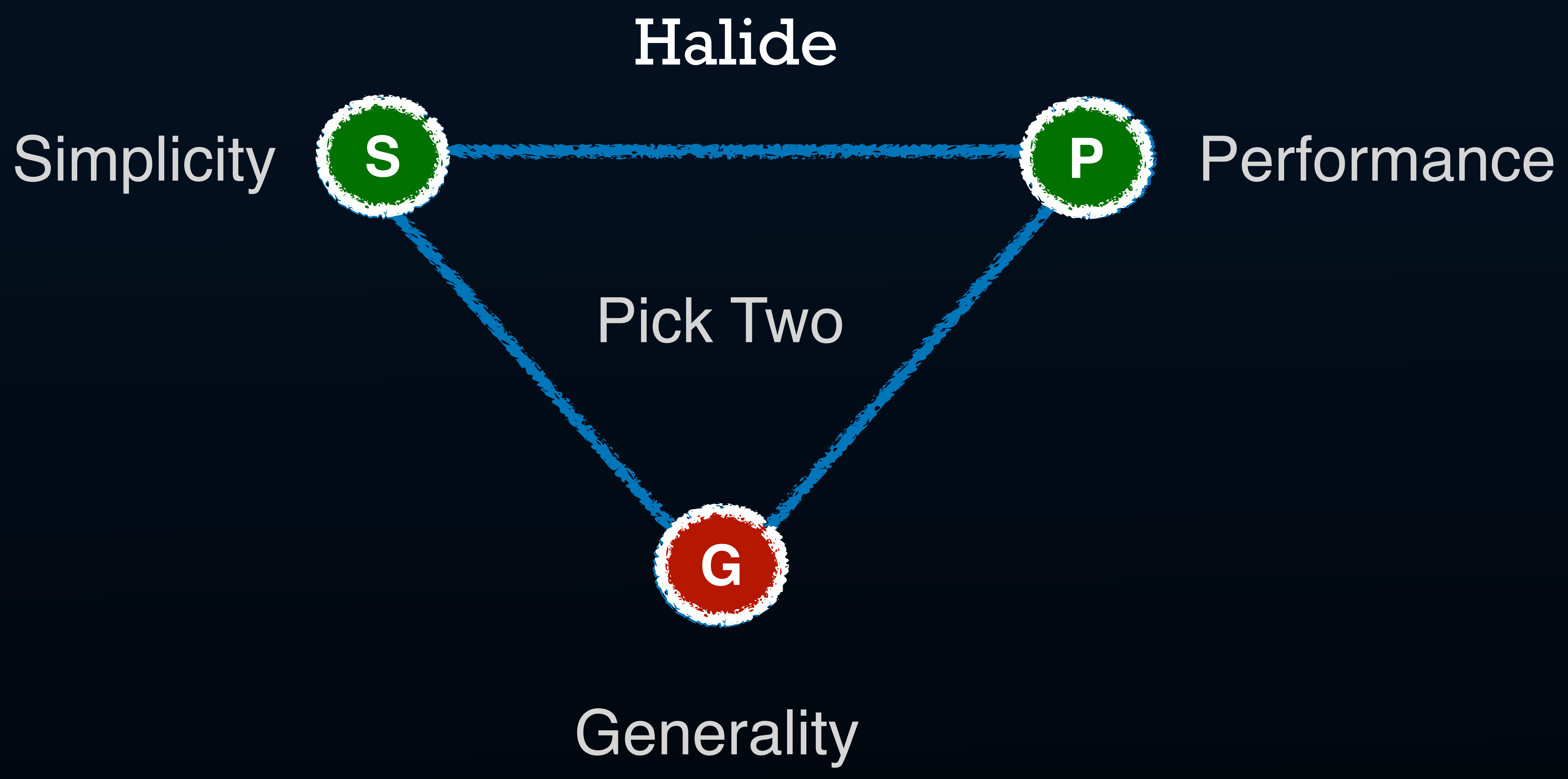
The Challenge



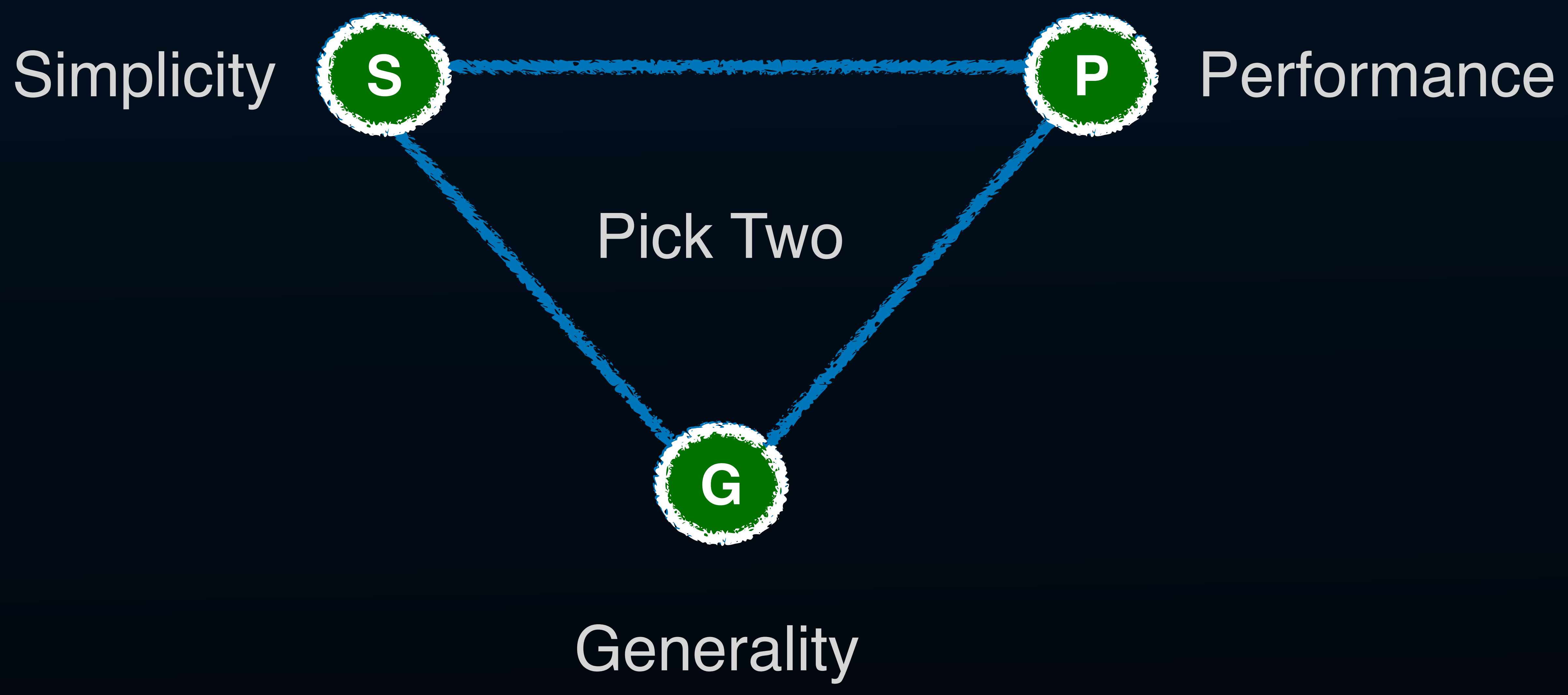
The Challenge



The Challenge



The Challenge



"Holy grail" solution:
Automatic compiler parallelization of simple code for general data problems

Bodo Example Code

```
@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
```

Simple

Native Python APIs and no "Pandas-Like" API Layers

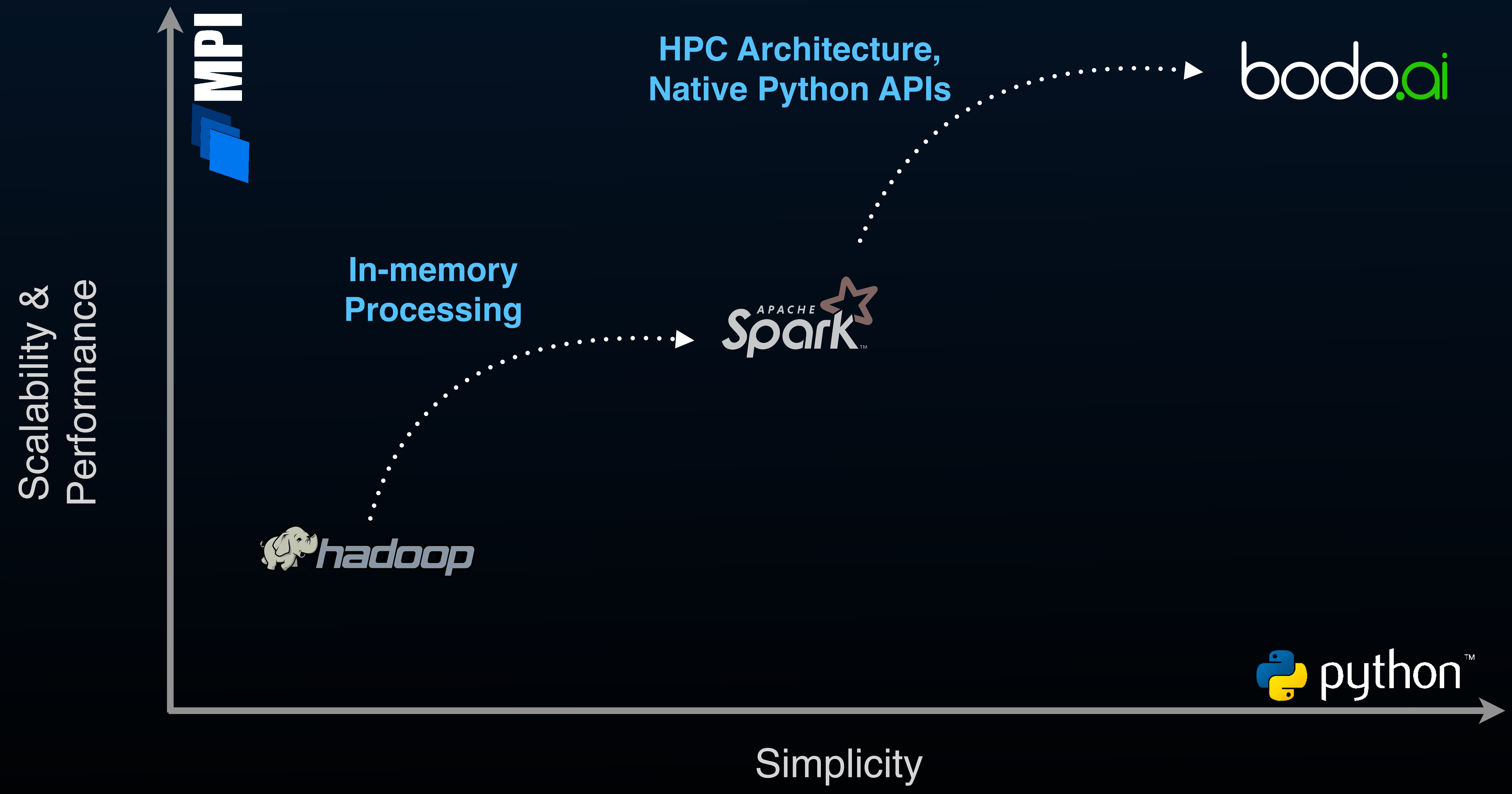
HPC Scaling

115X speed up on 4 nodes

Numba MPI

bodo.ai

Bridging Simplicity-Performance Gap



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[1,2]
- Example: use integer programming to explore decision search space[1]

Addition Challenges of Python:

- Dynamic typing
- Complex data structures like dataframes

Bodo Approach[3]

- Focus on high-level APIs, not loops
- Treat Pandas/Numpy as deeply embedded DSLs
- High-level API semantics assist with parallelization

The Array Privatization Transformation

```
DO i=1,n
 DO j=1,m
   work(j) = ...
 ENDDO
 ...
 DO j=1,m
   work(j) = ...
 ENDDO
ENDDO
```

→

```
CDOALL i=1,n
 REAL work(1:m)
  DO j=1,m
   work(j) = ...
  ENDDO
  ...
  DO j=1,m
   ... = work(j)
 ENDDO
ENDDO
```

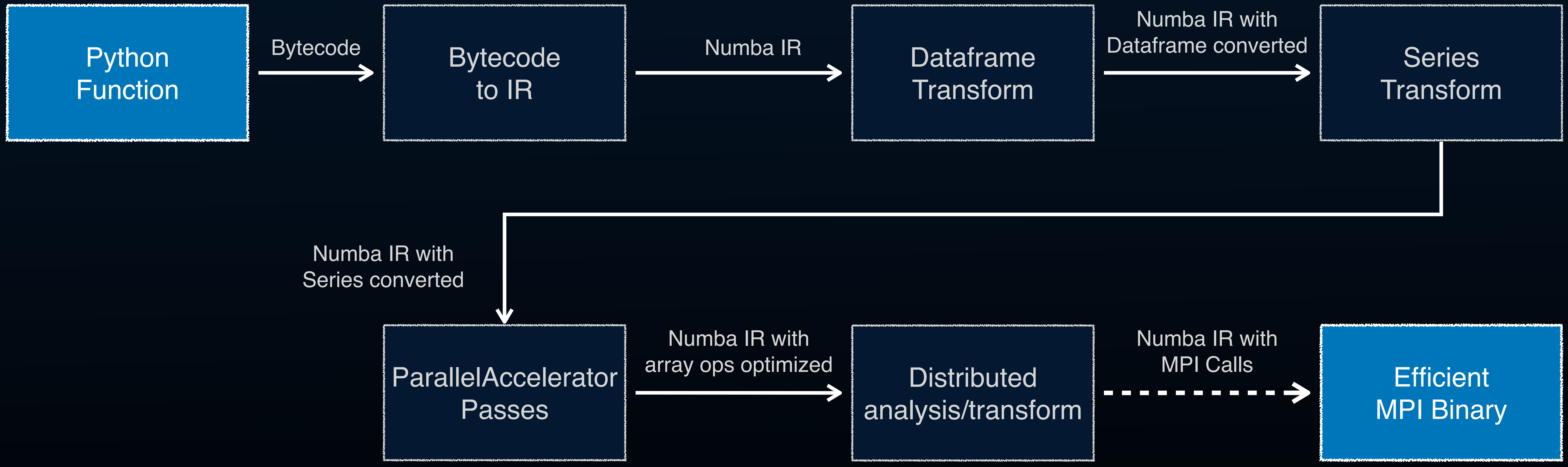
[1] Kennedy and Kremer, "Automatic data layout for high performance Fortran", SC'95

[2] Eigenmann et al,"On the automatic parallelization of the Perfect Benchmarks", TPDS'98

[3] Totoni et al, "HPAT: high performance analytics with scripting ease-of-use", ICS'17

bodo.ai

Compiler Engine



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

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

$$REP \leq 2D_{BC} \leq 1D_B$$

$$\perp = REP, T = 1D_B$$

meet operator $\wedge$

$$D_a : A \rightarrow L$$
$$D_p : P \rightarrow L$$

$$\begin{tikzcd} 1D_B \arrow[d] \\ 2D_{BC} \arrow[d] \\ REP \end{tikzcd}$$

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:

$$f_{l=r} : D_a(l) = D_a(r) = D_a(l) \wedge D_a(r)$$

Arguments:

$$f_{Arg(x)} : D_a(x) = getArgFlags(x)$$

$$\begin{array}{c} 1D_B \\ \downarrow \\ 2D_{BC} \\ \downarrow \\ REP \end{array}$$

Binary Ops:

$$f_{l=r_1+r_2} : D_a(l) = D_a(r_1) = D_a(r_2) = D_a(l) \wedge D_a(r_1) \wedge D_a(r_2)$$

Join:

$$f_{Join(x_1,x_2,\ldots)} : D_a(x_1) = D_a(x_2) = \ldots = D_a(x_1) \wedge D_a(x_2) \wedge \ldots$$

Function calls:

$$f_{c(x_1,x_2,\ldots)} : D_a(x_1), D_a(x_2), \ldots = knownCalls(c)(D_a(x_1), D_a(x_2), \ldots)$$

$$f_{unknownCall(x_1,x_2,\ldots)} : D_a(x_1) = D_a(x_2) = \ldots = REP$$

Automatic Parallelism Extraction

df

$$
\begin{array}{ccc}
A & B & C \\
\end{array}
$$

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

1D block distribution

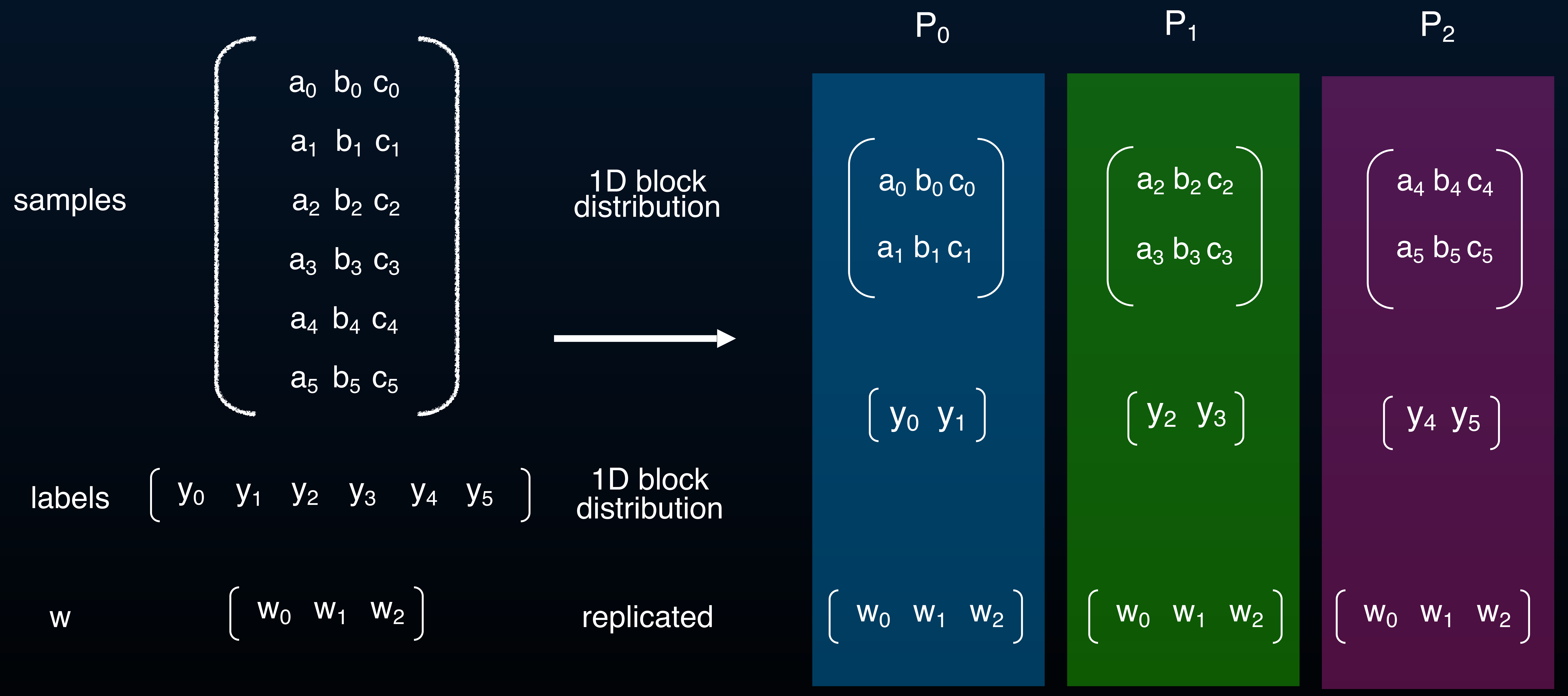
$$\xrightarrow{m\ =\ df.mean()}$$

m

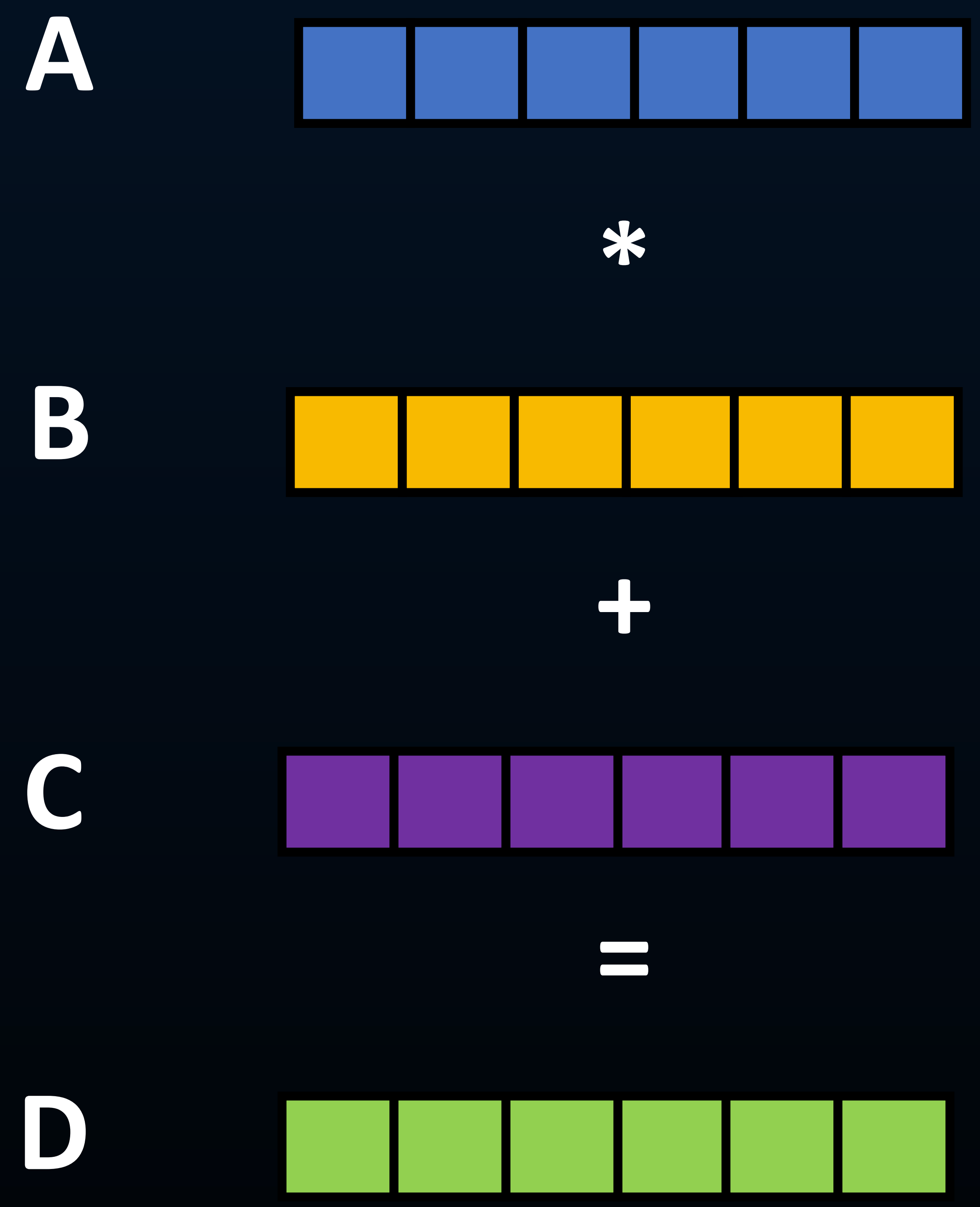
$$
\begin{pmatrix}
m_A \\
m_B \\
m_C
\end{pmatrix}
$$

replicated

Automatic Distribution



Data Parallelism Extraction



D = A * B + C

↓ Recognize parallelism

```
parfor i=0:n
        t[i]=A[i]*B[i]
parfor i=0:n
        D[i]=t[i]+C[i]
```

↓ Fuse loops

```
parfor i=0:n
        D[i]=A[i]*B[i]+C[i]
```

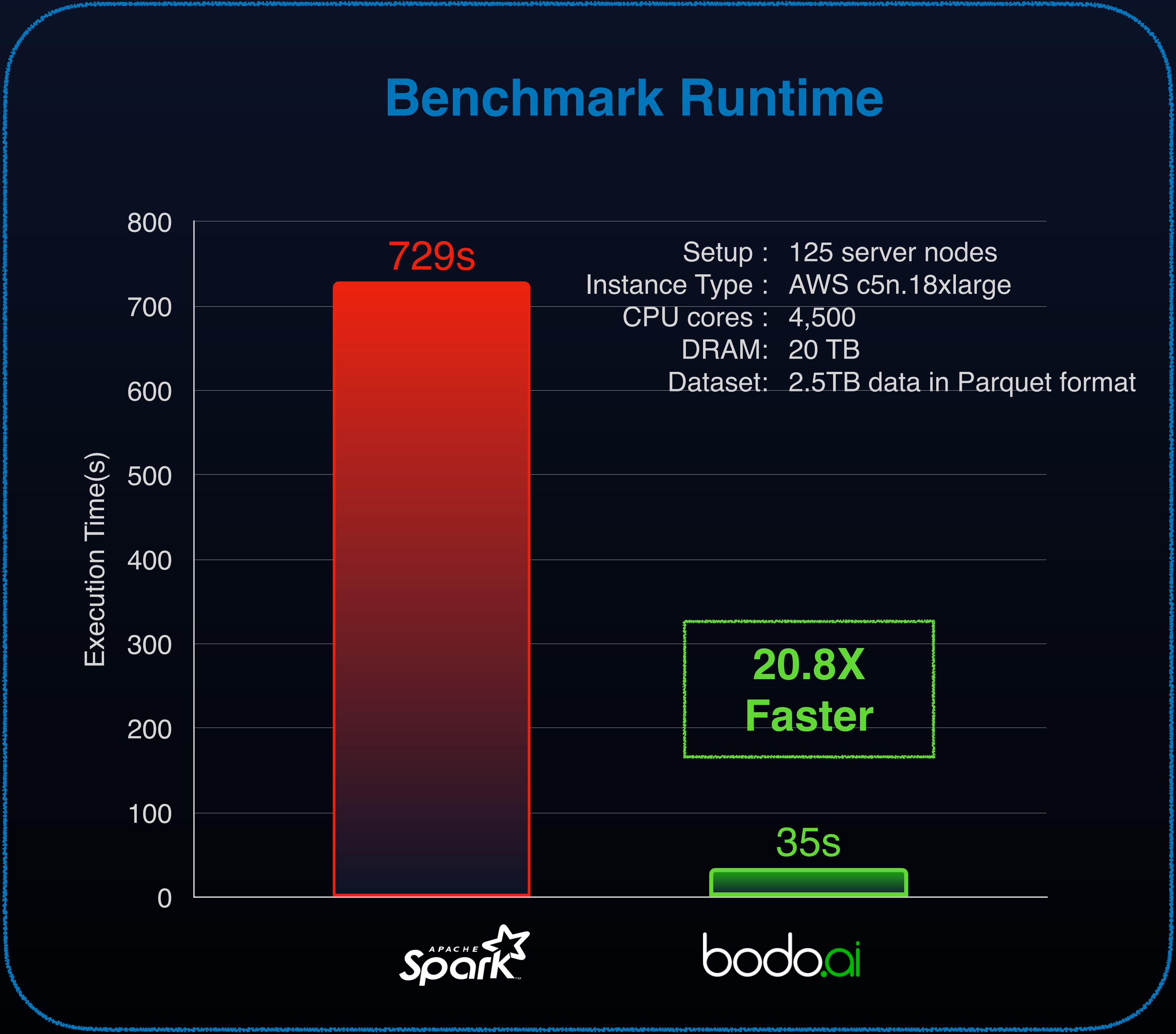
Anderson et al. “Parallelizing Julia with a Non-invasive DSL”, *ECOOP’17*

Transfer Function for Parfors

```
distribution = 1D_B
parArrays = ∅
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 ∪ arrayAccess.array
        distribution= distribution Λ D_a(arrayAccess.array)
    if parforIndexVar ∈ 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
```

$$1D_B \downarrow 2D_{BC} \downarrow REP$$

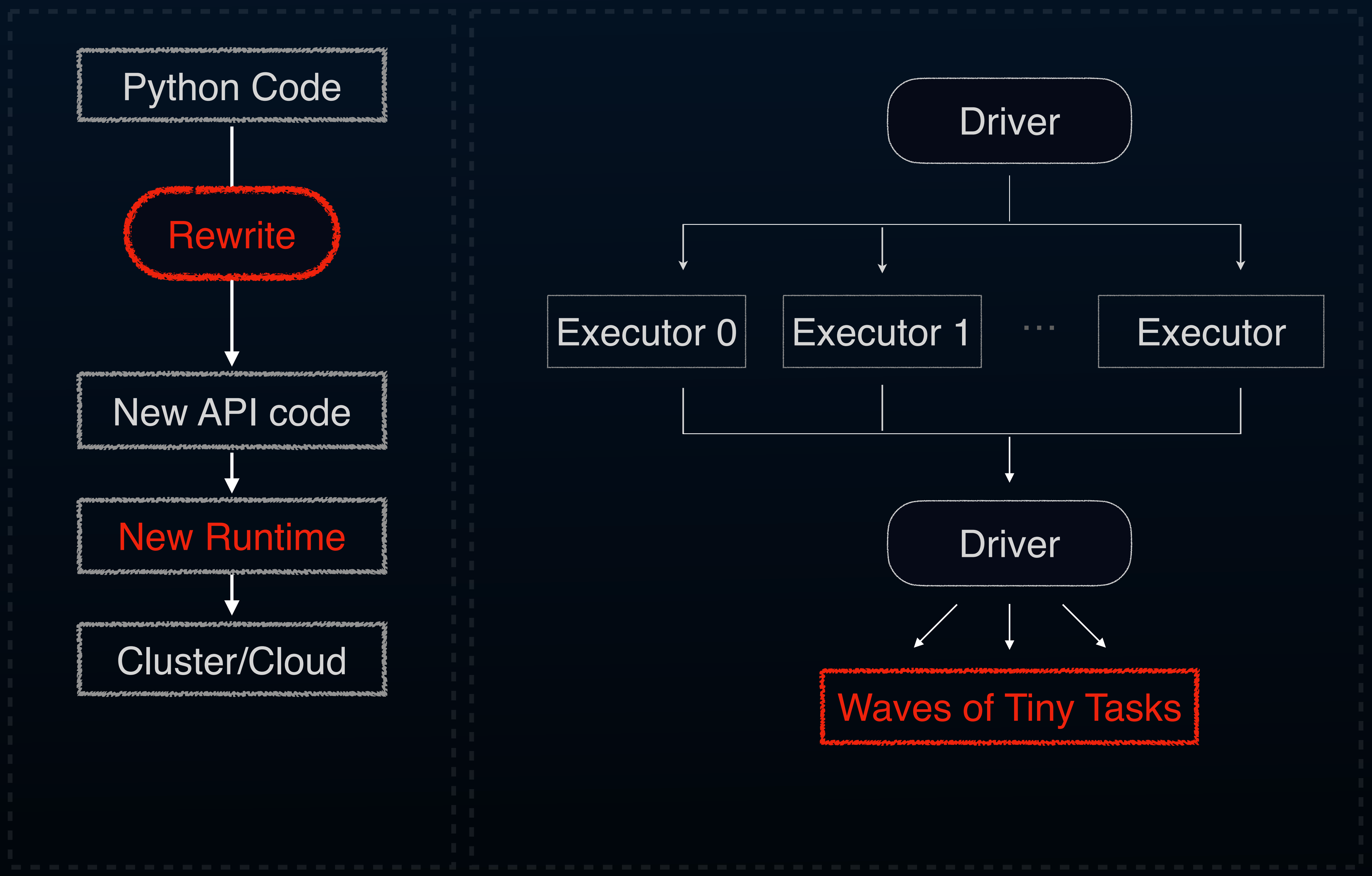
TPCxBB Q26 Benchmark



bodo.ai

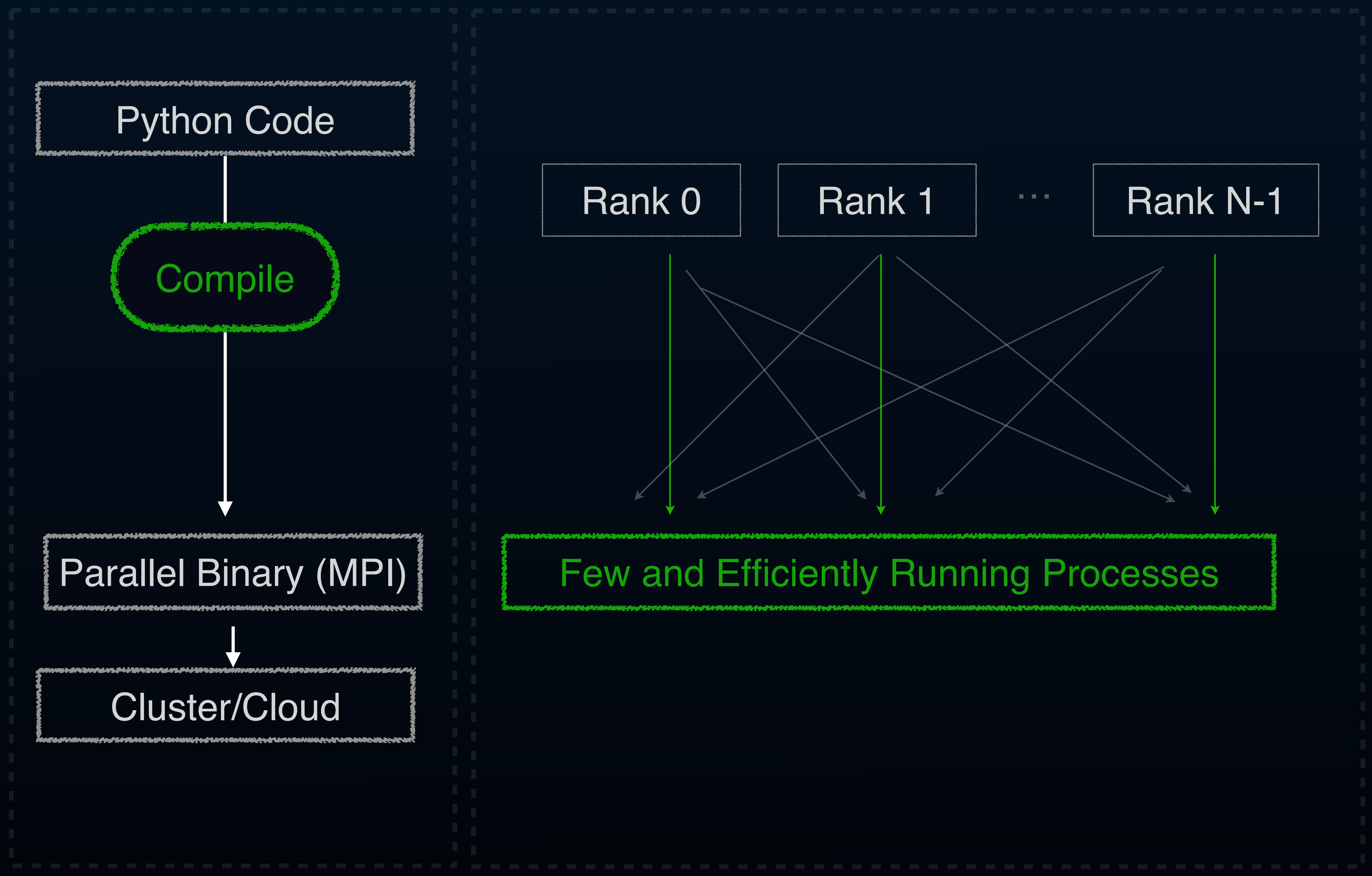
Parallel Execution Models

RAY Spark DASK



Driver-Executor Tiny Task Runtime Overheads

bodo.ai



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.ai

Bodo Limitation: type stability

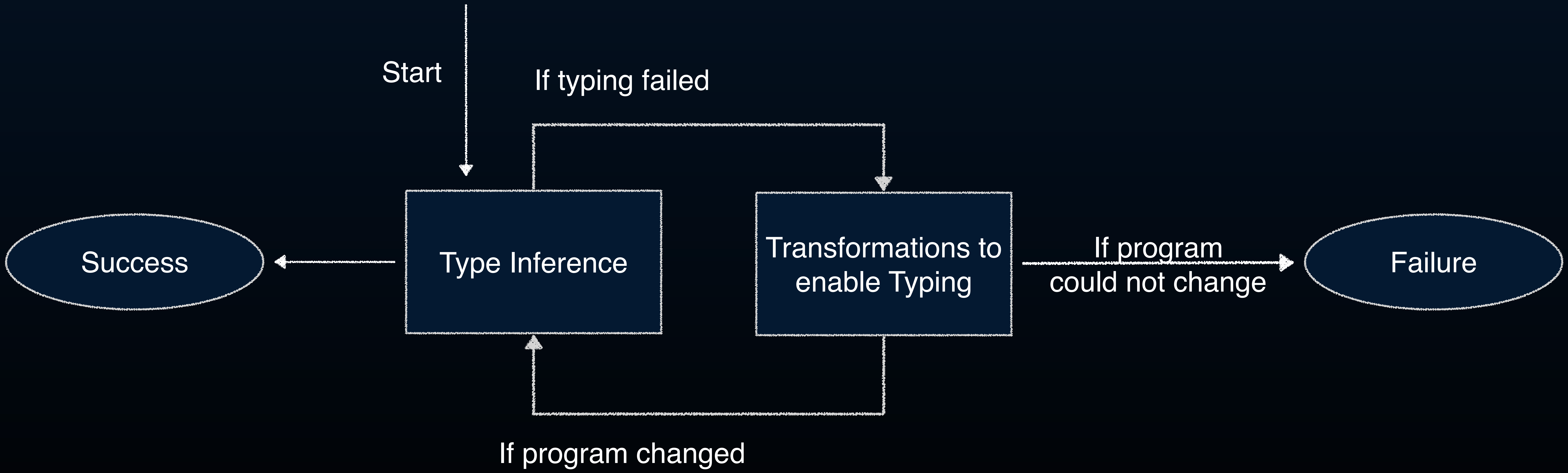
<table>
<tr><th>Untypable variable</th><th>Unresolvable function</th><th>Nonstatic dataframe schema</th></tr>
<tr><td><pre>if flag1:
    a = 2
else:
    a = np.ones(n)
if isinstance(a, np.ndarray):
    doWork(a)</pre></td><td><pre>if flag2:
    f = np.zeros
else:
    f = np.ones
b = f(m)</pre></td><td>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

b = f(m)</pre></td></tr>
</table>

bodo.ai

Iterative transform-based typing

Transform code to be handled by type inference



bodo.ai

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

```
@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"]]
```

→

```
A,B,C = ReadParquet(["A", "B", "C"])
df = init_dataframe(("A", "B", "C"), (A, B, C))
df2 = init_dataframe(("A1", "B", "C"), (df.A, df.B, df.C))
A2 = np.full(len(df2), 1.1)
df3 = init_dataframe(("A1", "B", "C", "A2"), (df2.A1, df2.B,
df2.C, A2))
return init_dataframe(("A1", "A2", "B"), (df3.A1, df3.B,
df3.A2))
```

bodo.ai

Dataframe Optimizations - continued

```
A,B,C = ReadParquet(["A", "B", “C"])
df = init_dataframe((“A”, “B”, “C”), (A, B, C))
df2 = init_dataframe((“A1”, “B”, “C”), (df.A, df.B, df.C))
A2 = np.full(len(df2), 1.1)
df3 = init_dataframe((“A1”, “B”, “C”, “A2”), (df2.A1, df2.B,
df2.C, A2))
return init_dataframe((“A1”, “A2”, “B”), (df3.A1, df3.B,
df3.A2))
```

```
A,B,C = ReadParquet(["A", "B", “C"])
df = init_dataframe((“A”, “B”, “C”), (A, B, C))
df2 = init_dataframe((“A1”, “B”, “C”), (A, B, C))
A2 = np.full(len(df2), 1.1)
df3 = init_dataframe((“A1”, “B”, “C”, “A2”), (A, B, C, A2))
return init_dataframe((“A1”, “A2”, “B”), (A, B, A2))
```

```
A,B,C = ReadParquet(["A", "B", “C"])
A2 = np.full(len(A), 1.1)
return init_dataframe((“A1”, “A2”, “B”), (A, B, A2))
```

```
A,B = ReadParquet(["A", "B"])
A2 = np.full(len(A), 1.1)
return init_dataframe((“A1”, “A2”, “B”), (A, B, A2))
```

bodo.ai

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

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

```
@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
```

bodo.ai

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
```

bodo.ai

Pattern Matching Optimizations

Fuse operations to avoid intermediate data

```
S2 = S.str.split(“,”)
S3 = S2.explode()
```

→

```
S3 = str_split_explode(S)
```

Data Structure Optimization

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

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’]
…

Bodo changes data structures transparently:

Packed string array

0, 9, 18, 27, …

A1;B1;C1A2;B2;C2A3;B3;C3

A.str.split(';')

Packed array of string lists

0, 6, …

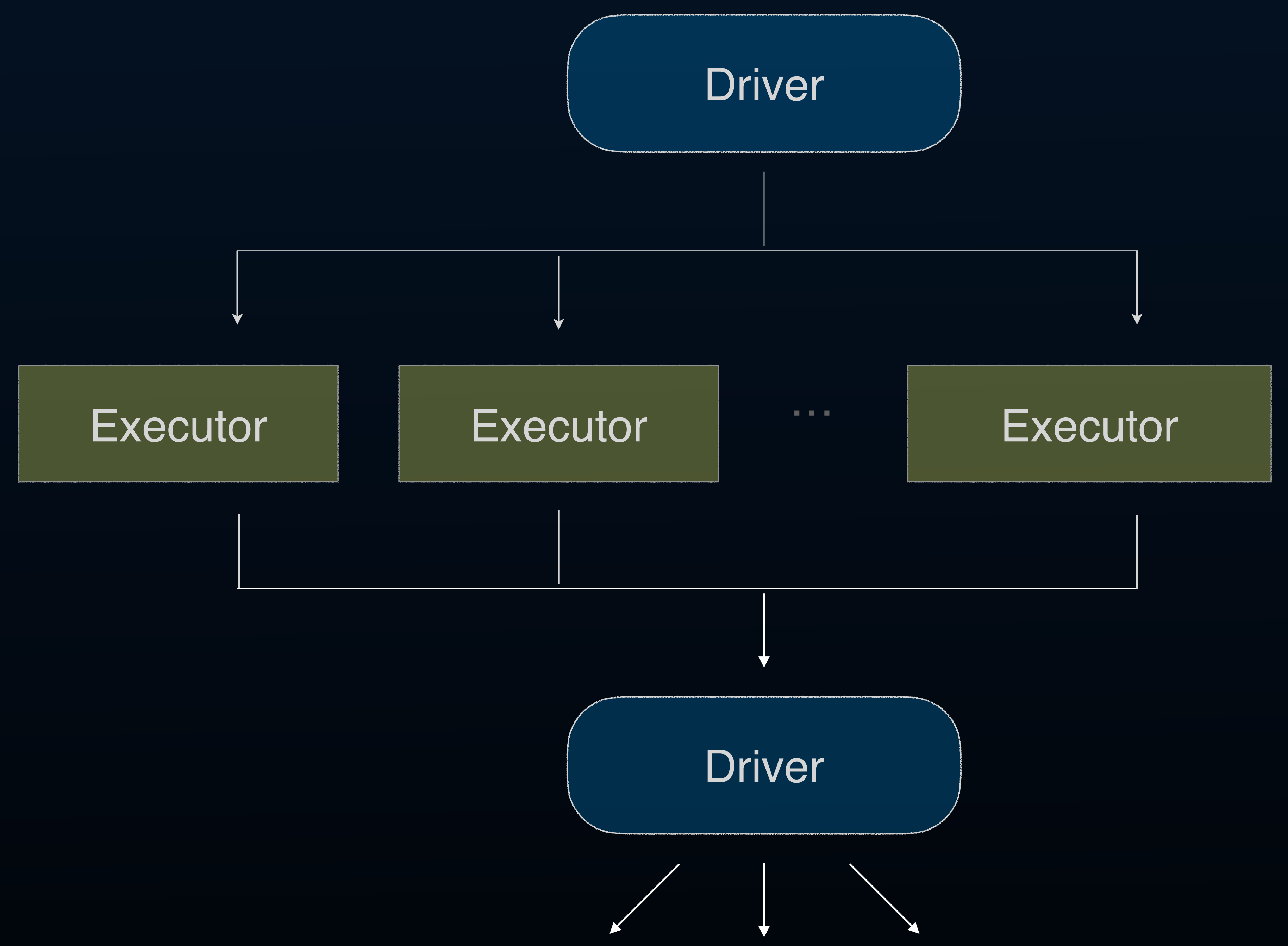
List offsets

0, 2, 3, 5, 6, 8, 8, 10 …

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



bodo.ai

Python vs. Spark/SQL

Python

Simple imperative code for compute functions

```
df.apply(lambda r: 1 if r.A == “AA” else 0, axis=1)
```

Step-by-step code for data processing

```
df3 = df1.merge(df2, ...)
....
df5 = df4.groupby(“A”).min()
df6 = df5.sort_values(by=“B”)
...
```

Spark

Exposes data partitions Lazy evaluation wrappers for SQL

```
sdf.withColumn("B", F.when(sdf["A"] == “AA”, 1).otherwise(0))
```

SQL

Long declarative expression

```
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=DJ1sGQryoAc
  - 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)

Backup

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

bodo.ai

BodoSQL: example

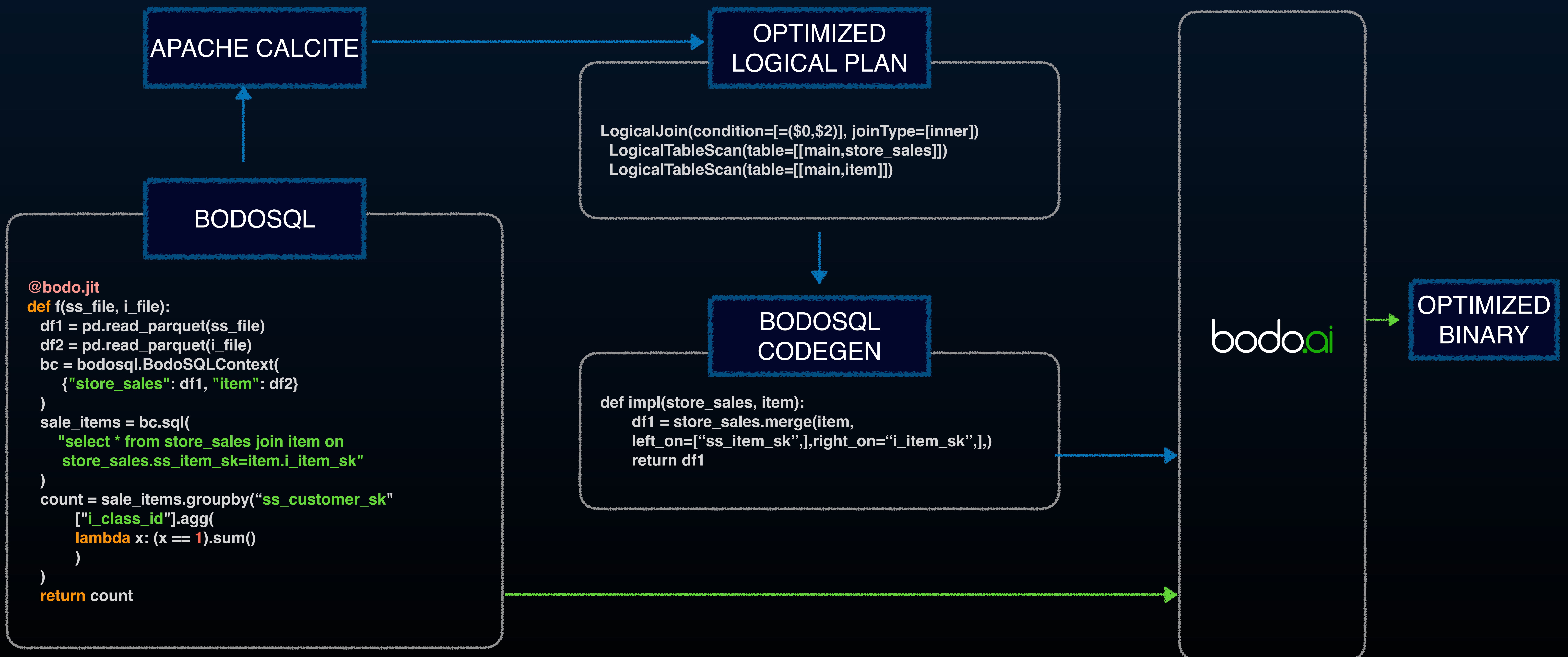
```
import pandas as pd
import bodo
import bodosql

@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"))
```

BodoSQL: error checking example

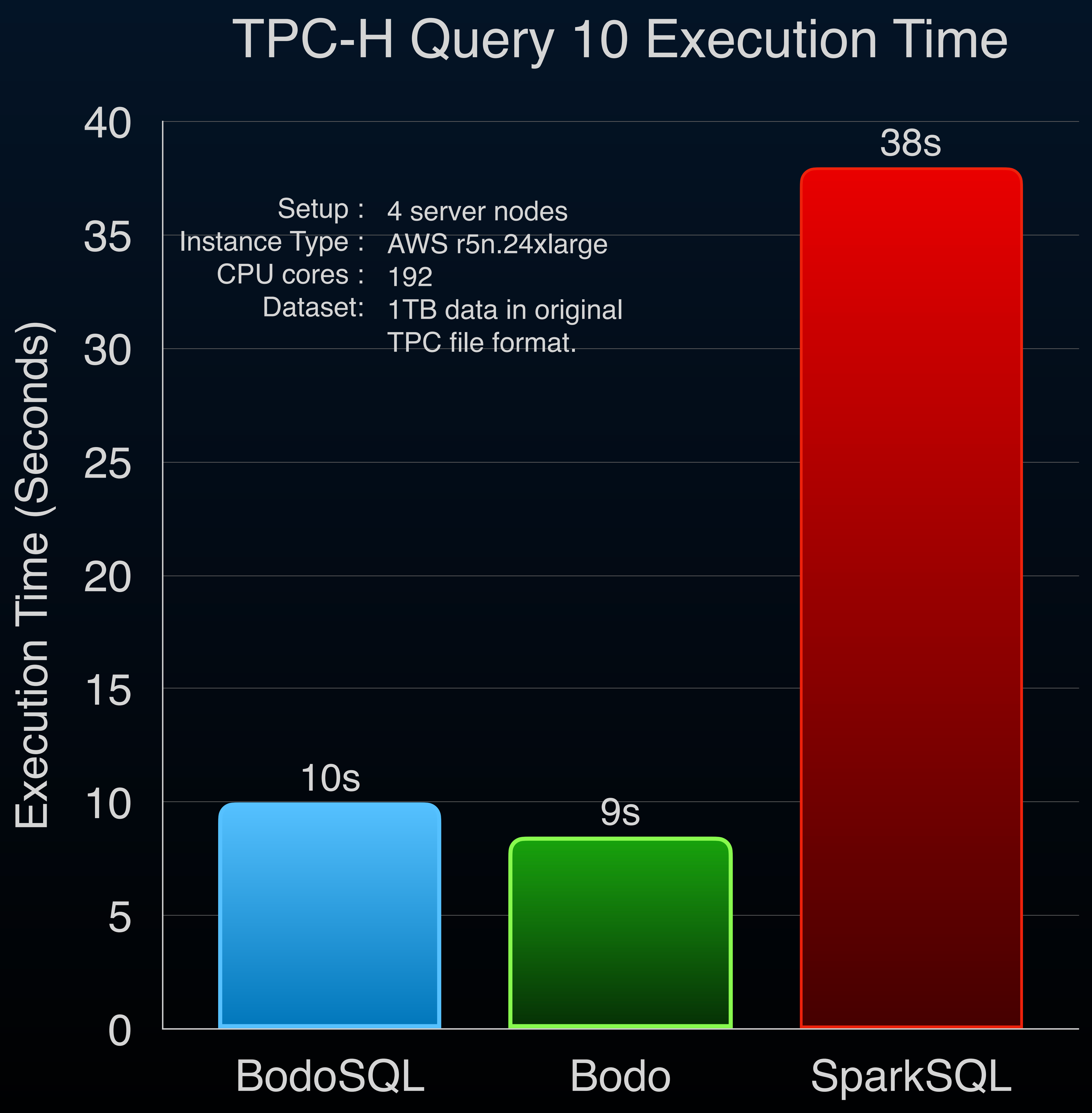
```
import pandas as pd
import bodo
import bodosql

@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.ai

BodoSQL: internal design



BodoSQL: performance



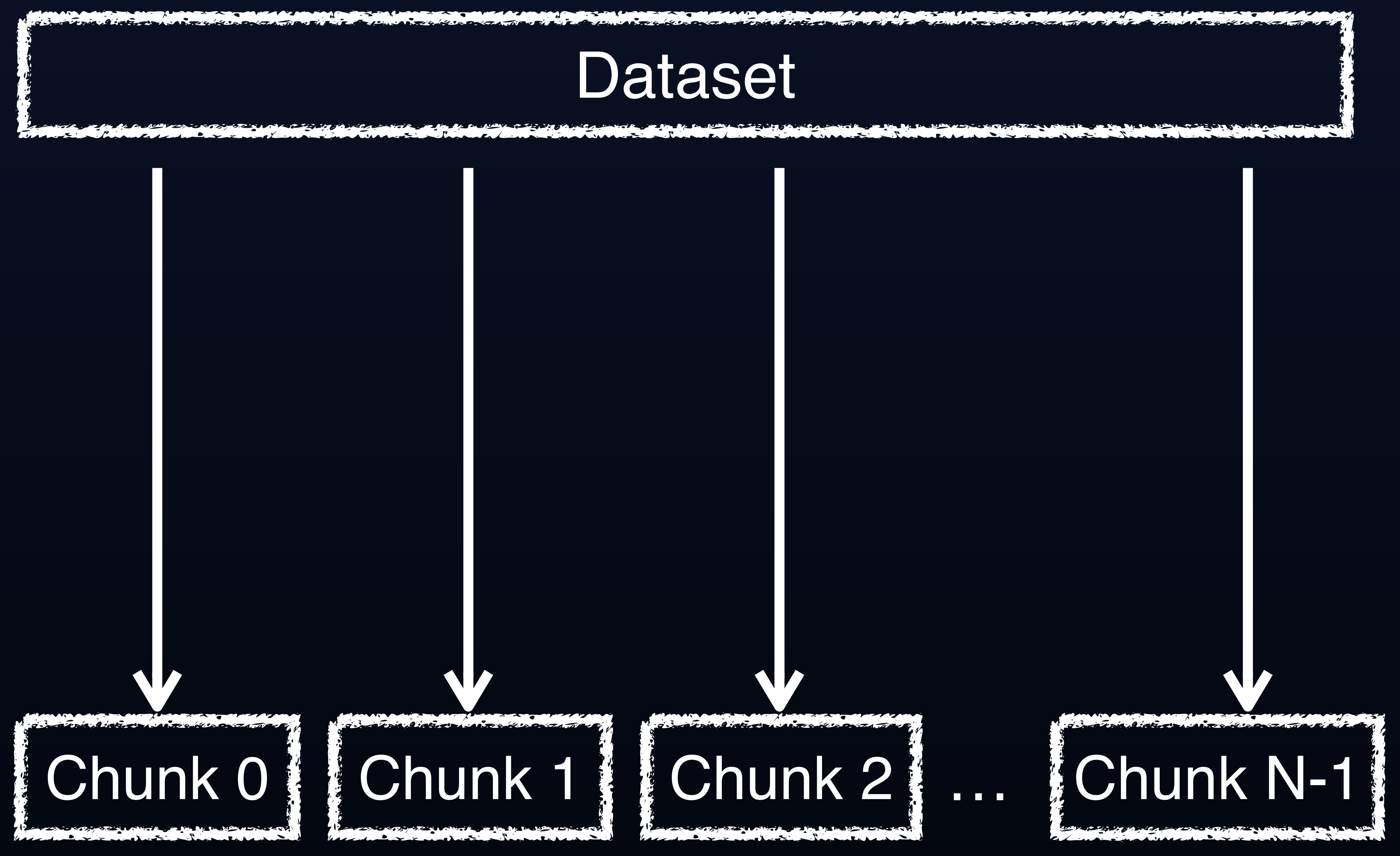
bodo.ai

Scalable I/O

```
@bodo.jit
def read_pq():
    df = pd.read_parquet('example.pq')
    ...
```

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

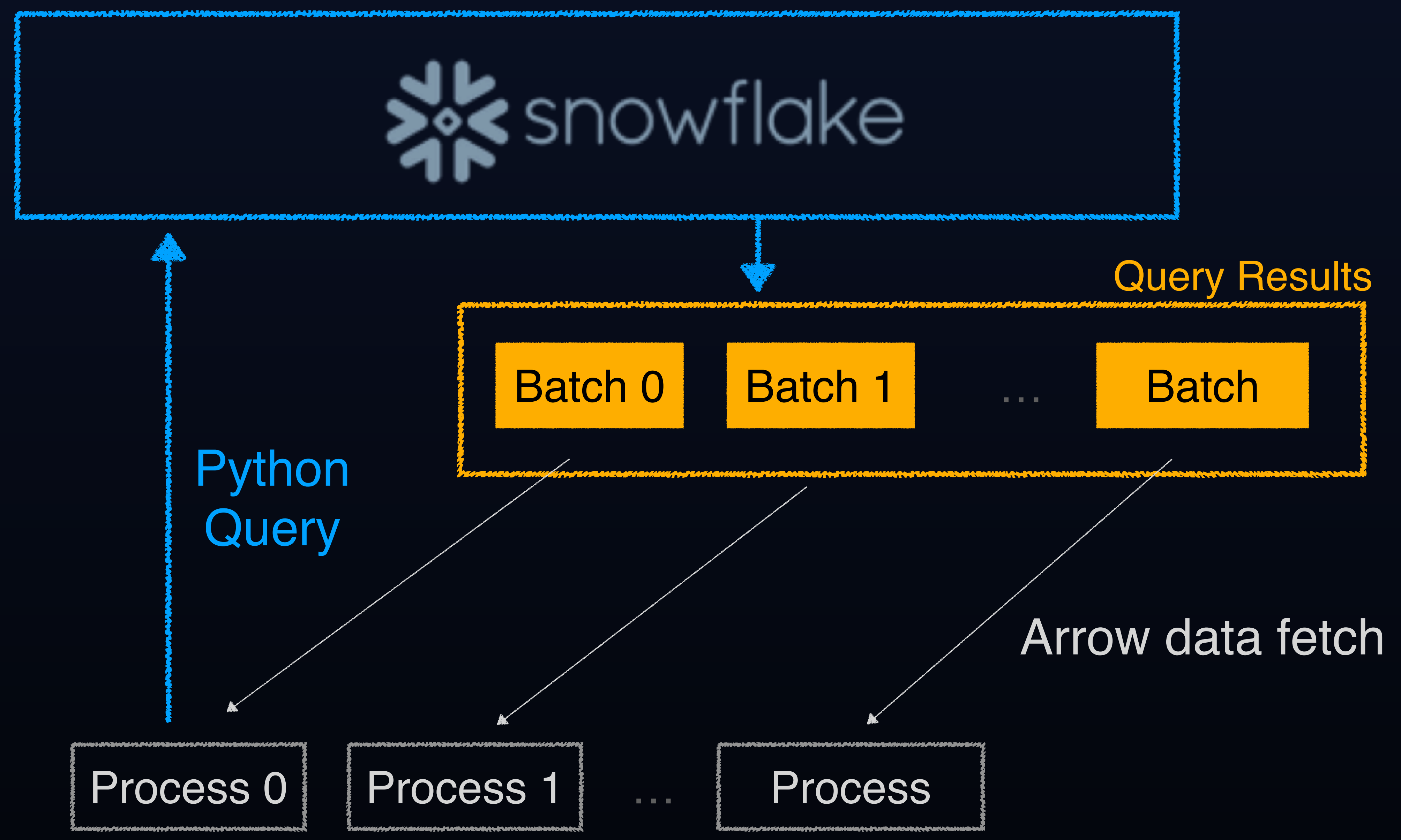
```
@bodo.jit
def read_sql():
    df = pd.read_sql('select * from employees',
'snowflake://')
     ...
```



bodo.ai

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

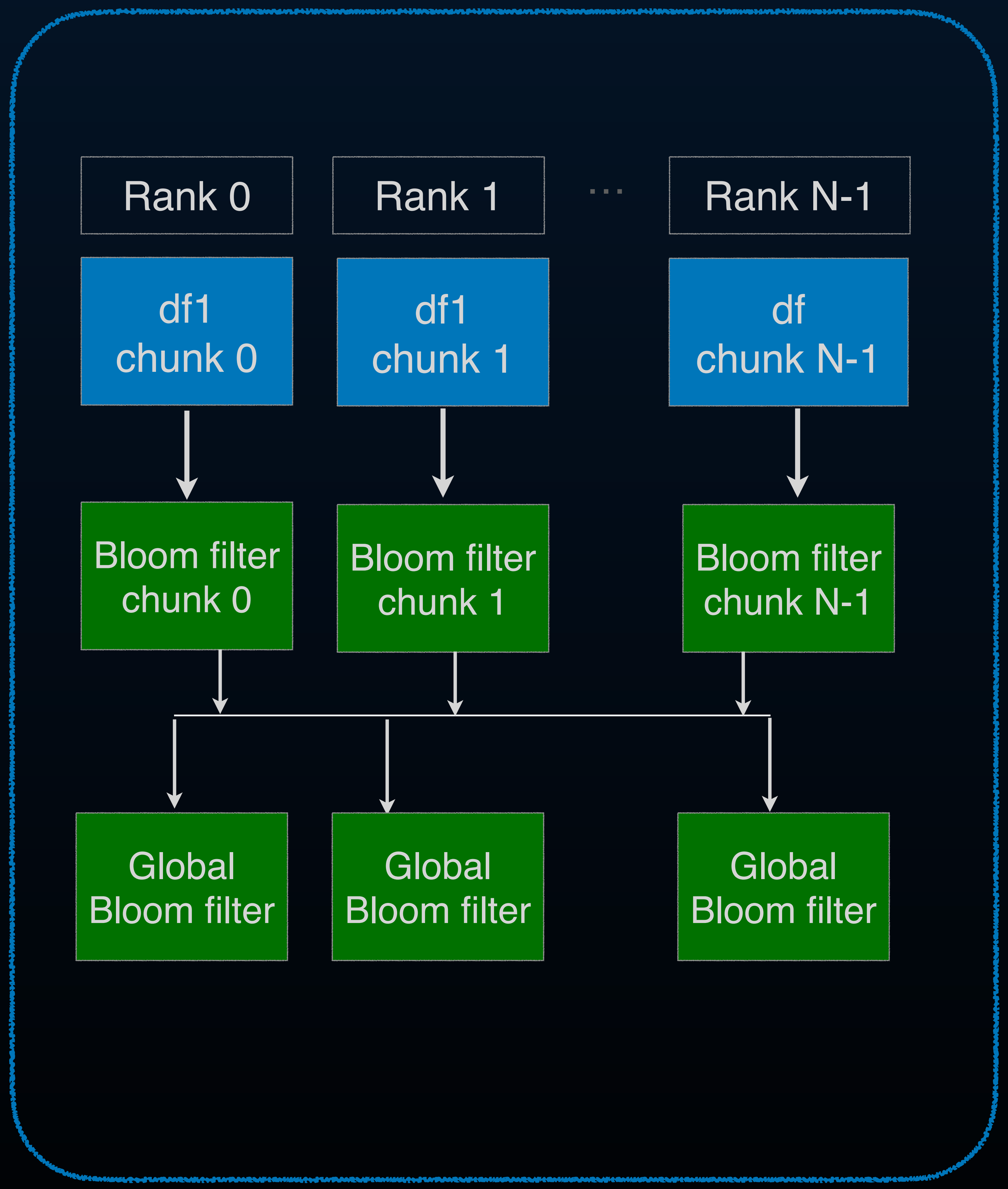


bodo.ai

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)

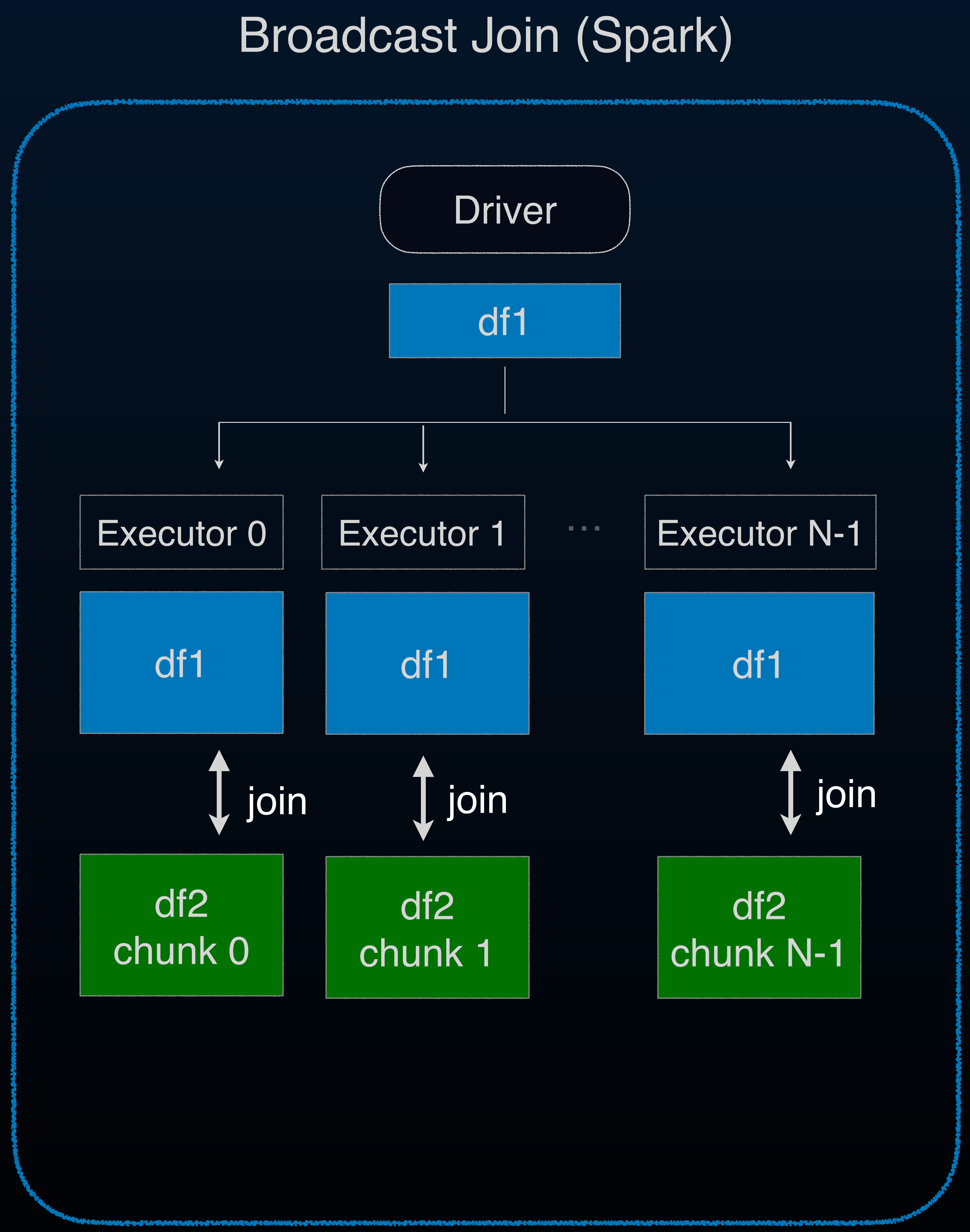
Bloom Filter Construction



bodo.ai

The Case Against Broadcast Join

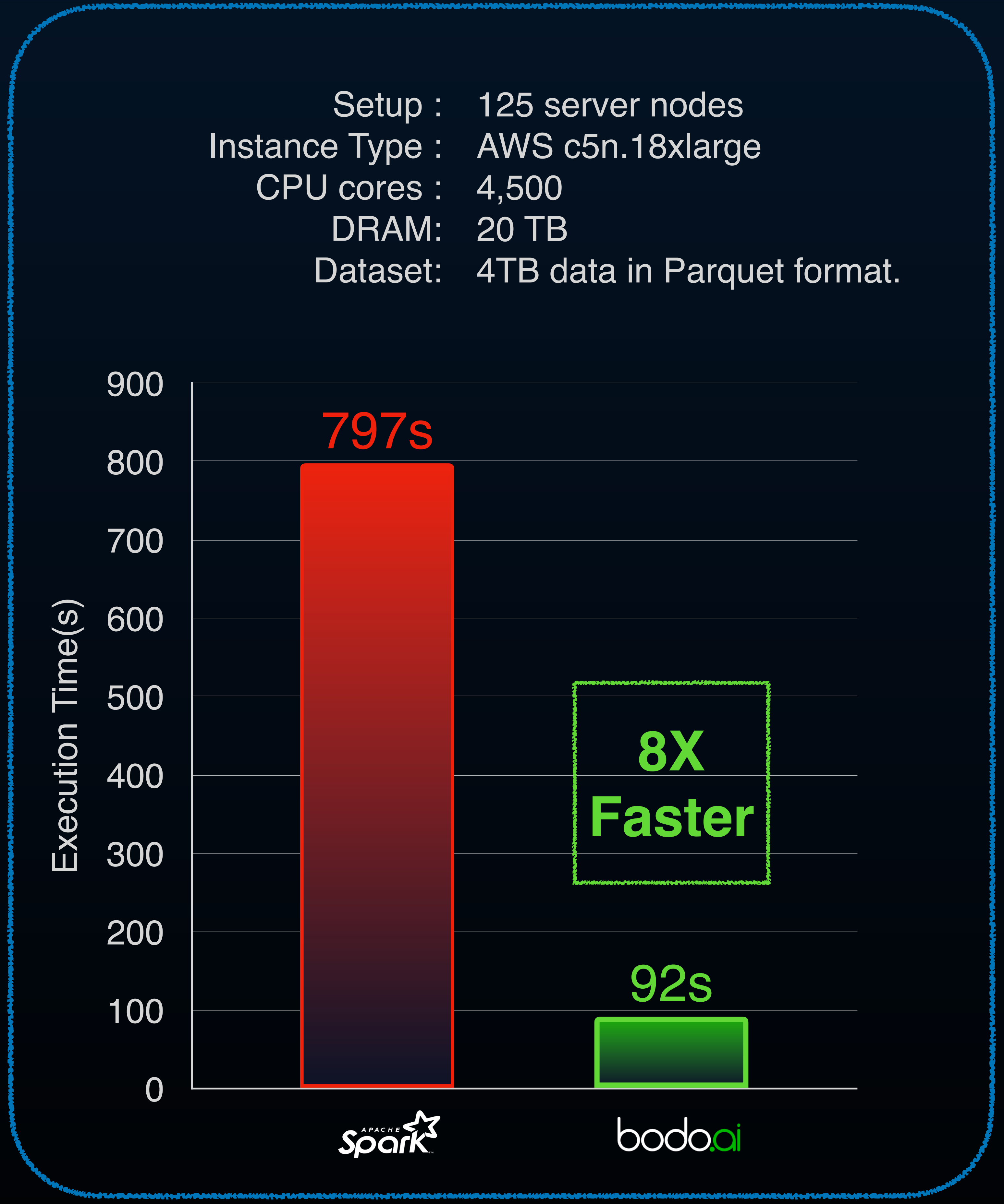
- Broadcast join is often used to avoid shuffle
- Our results: broadcast join is rarely faster
- Broadcast join is inherently non-scalable
  - Parallel work and memory increases with number of cores:
    - $W_p \sim p \times |T_l|$
    - $M_p \sim p \times |T_l|$
- Bloom filters are scalable



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



bodo.ai

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/

TPC-H Q18 : Large Value Customer Query

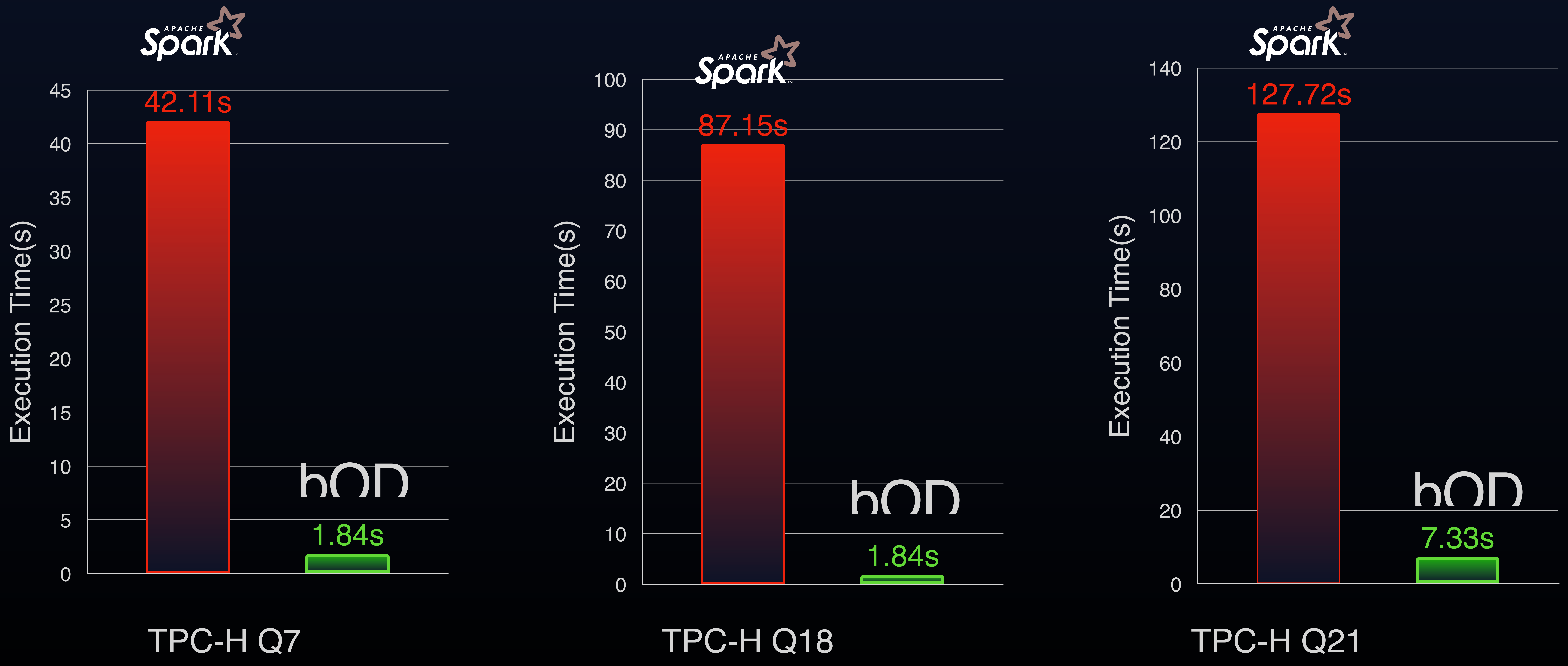
```sql
select c_name, c_custkey, o_orderkey, o_orderdate,
   o_totalprice, sum(l_quantity)
from customer, orders, lineitem
where o_orderkey in (
   select l_orderkey from lineitem group by l_orderkey
      having sum(l_quantity) > 300
   )
   and c_custkey = o_custkey and o_orderkey = l_orderkey
group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

```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,
   )["L_QUANTITY"].sum()
   total = gb2.sort_values(
      ["O_TOTALPRICE", "O_ORDERDATE"], ascending=[False, True])
   print(total.head(100))
```

bodo.ai

Bodo vs. Spark (TPC-H)

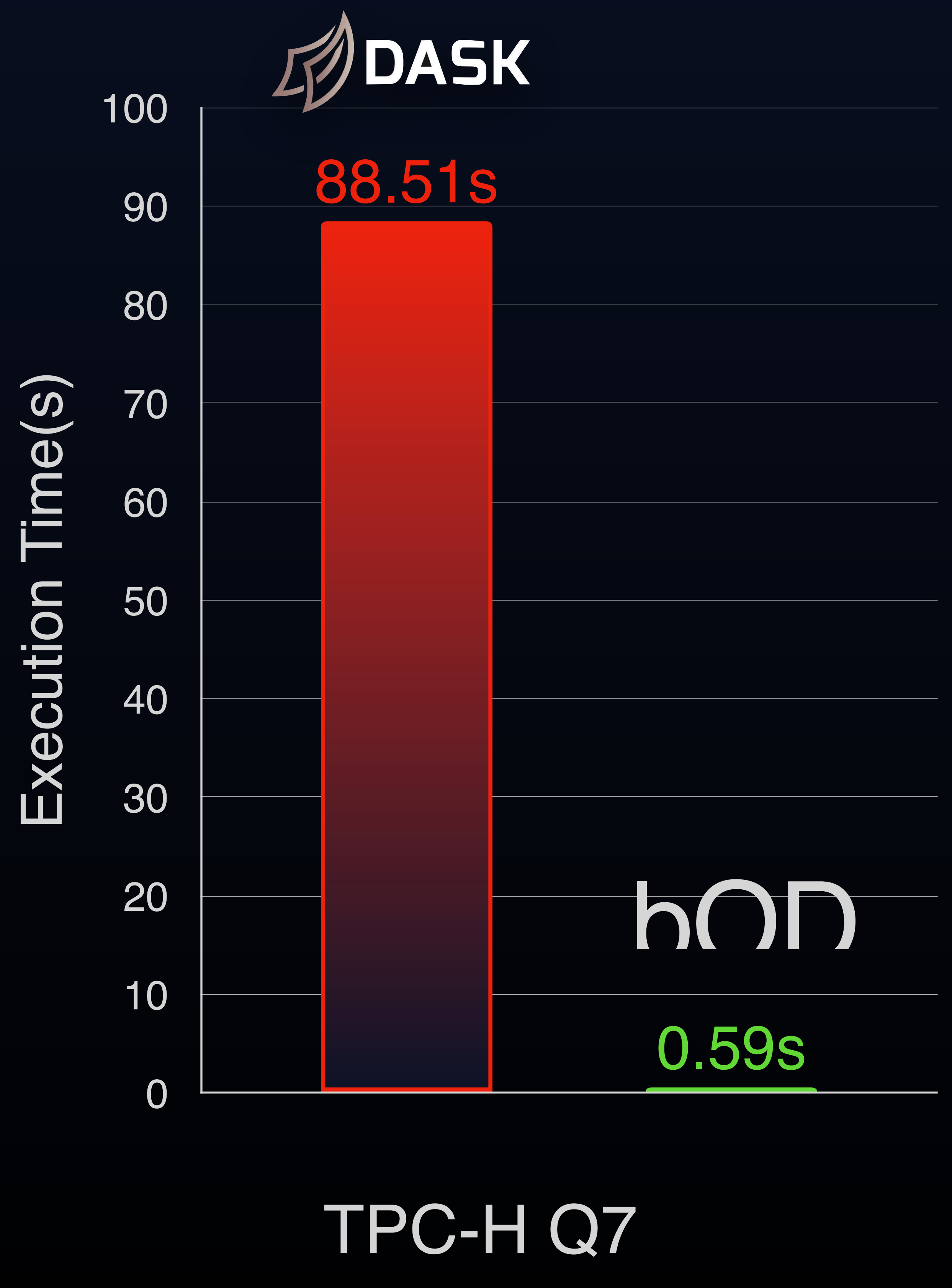
- TPC-H SF1000 (~1 TB) on 16 c5n.18xlarge AWS (576 physical cores, 3 TB memory), EFA
- Bodo 23x faster than Spark on average (up to 65x), ~90% infrastructure cost saving

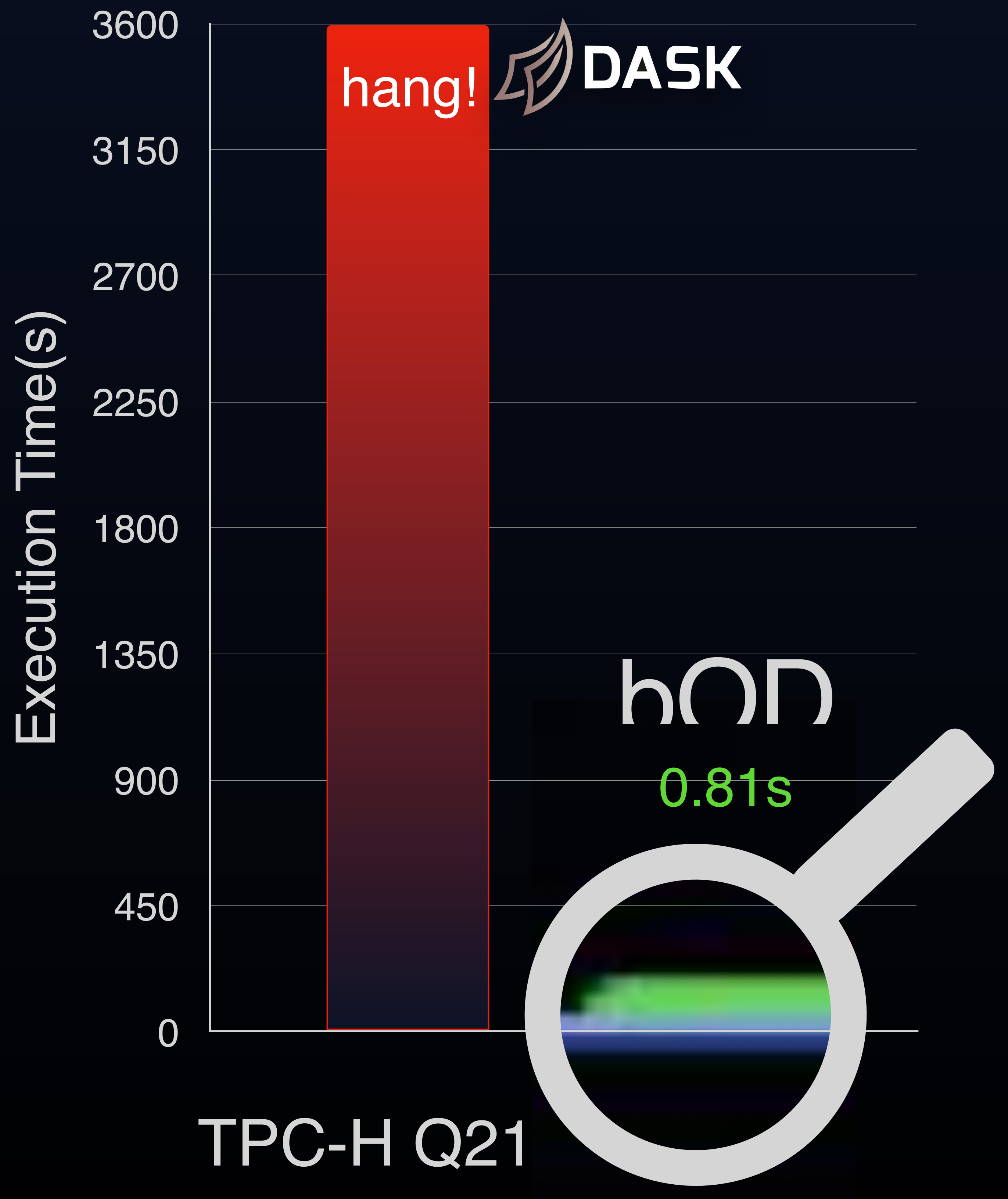


bodo.ai

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)

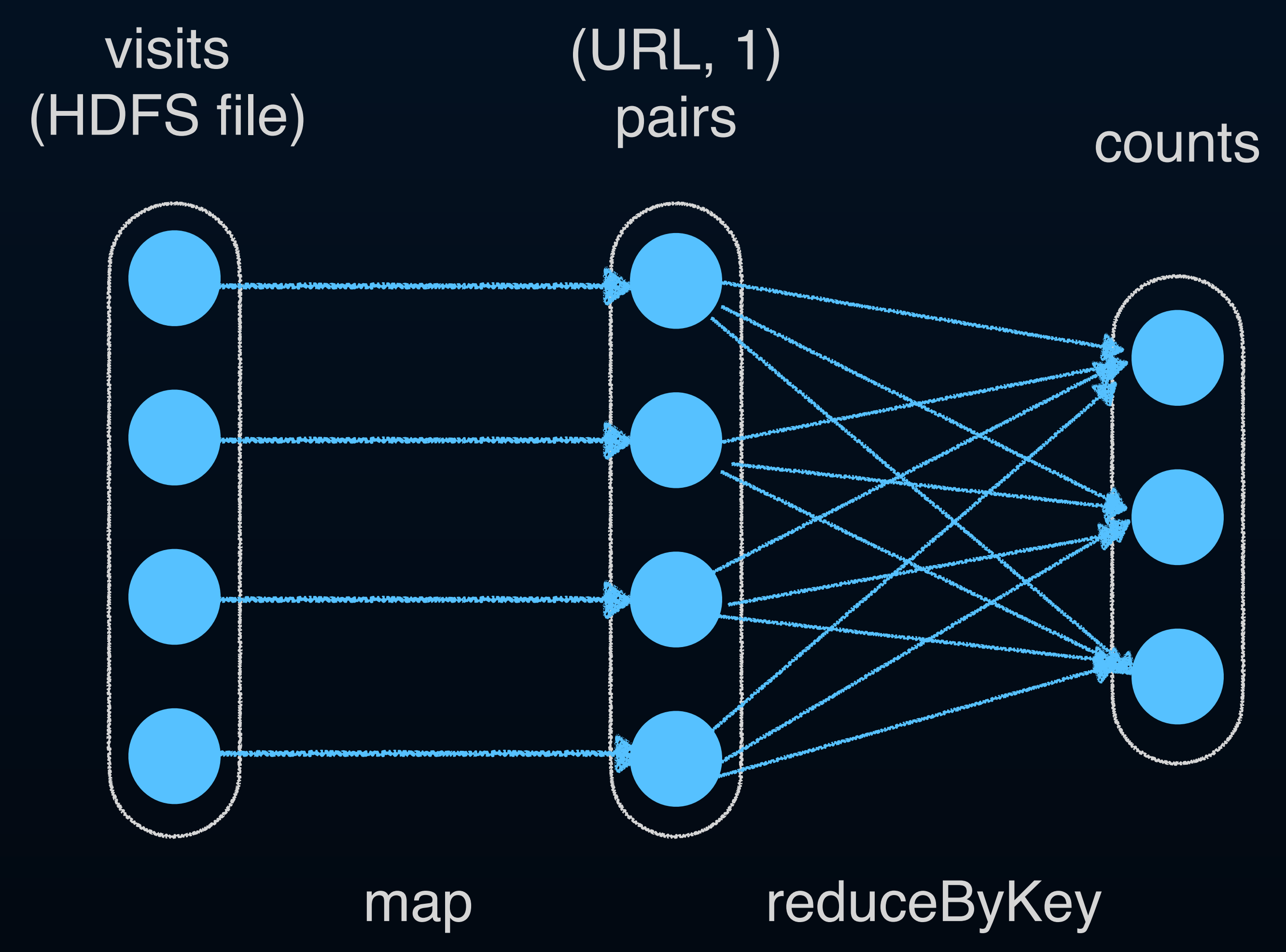
- Ray/Modin ran out of memory for any dataset >10 GB
- Ray was originally built for reinforcement learning, data engineering support not ready yet
- Task scheduling systems like Dask & Ray can achieve at most Spark performance

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

bodo.ai