Building Trino data pipelines with SQL or Python

Lester Martin
Head of Developer Relations
Starburst (the Trino company)

trino

Connection before content

Lester Martin – https://linktr.ee/lestermartin

- Developer Relations @ Starburst
  - Blogging & forums
  - Webinars & videos
  - User groups & events
  - Training & tutorials
- 30+ years of technology experience
  - Started journey on TRS-80 Model III
  - Played most roles, but a programmer at my core
  - ½ career in OLTP and ½ in data analytics
  - Decade+ of “big data” experience to include
    - Trino/Starburst, Hadoop, Hive, Spark
    - NiFi, Kafka, Storm, Flink
    - HBase, MongoDB

The rise of big data

Querying large volumes of data was difficult and time consuming

Early 2000s: Data generation and collection has skyrocketed due to the rise of the Internet

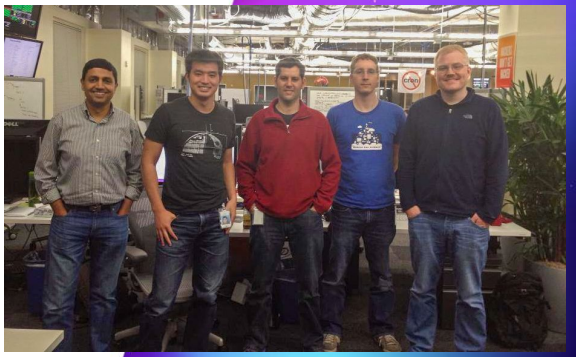
2006: Apache Hadoop was designed to meet the needs of large datasets on a scale previously unimaginable

2008: Facebook created Apache Hive to query terabytes of data in Hadoop using a SQL-like interface. Data consumers were limited by the number of queries they could run — often fewer than 10/day

The birth of Trino

A new query engine designed to solve the data accessibility problem

2012: Trino (*formerly known as Presto)* is created by Martin Traverso, Dain Sundstrom, David Phillips and Eric Hwang at Facebook



Trino is an open source query engine that:

- *Harnesses the power of distributed computing*
- *Separates compute from storage*
- *Super fast and performant*
- *Supports pluggable connectors to a variety of data sources*
- *ANSI-SQL BASED!!!!! Which means… SQL on anything!*

Trino trusted by industry leaders at PB scale

- Open-source query engine.
- Separates compute and storage.
- Queries across all data sources.
- Iceberg was designed for Trino.

Proven at exabyte scale/high concurrency:

25PB on S3

1 Exabyte of Data
100PB weekly data
1200 nodes
2.5M queries/week

600PB on S3
1000 nodes

10PB daily read data
250k queries per day

300PB data lake

Trino open source users

Starburst

Trino trusted by industry leaders at PB scale

- Open-source query engine.
- Separates compute and storage.
- Queries across all data sources.
- Iceberg was designed for Trino.

Proven at exabyte scale/high concurrency:

25PB on S3

1 Exabyte of Data
100PB weekly data
1200 nodes
2.5M queries/week

600PB on S3
1000 nodes

10PB daily read data
250k queries per day

300PB data lake

Trino open source users

Thriving open source community:

11300+
SLACK MEMBERS

10,000+
GITHUB STARS

750+
CONTRIBUTORS

Starburst

Trino trusted by industry leaders at PB scale

- Open-source query engine.
- Separates compute and storage.
- Queries across all data sources.
- Iceberg was designed for Trino.

Proven at exabyte scale/high concurrency:

25PB on S3

1 Exabyte of Data
100PB weekly data
1200 nodes
2.5M queries/week

600PB on S3
1000 nodes

10PB daily read data
250k queries per day

300PB data lake

Trino open source users

Starburst is the Trino company:

- **Bringing Trino to the enterprise**
- **Cofounded by Trino creators**
- **#1 Trino committer**
- **Largest team of Trino experts in the world**

Thriving open source community:

11300+
SLACK MEMBERS

10,000+
GITHUB STARS

750+
CONTRIBUTORS

Starburst

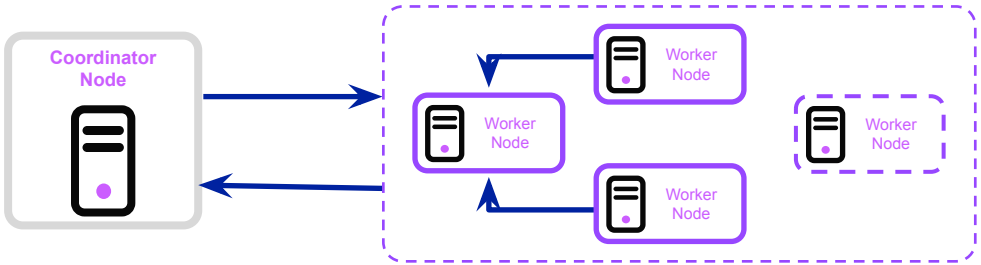
Server stereotypes

Coordinator node

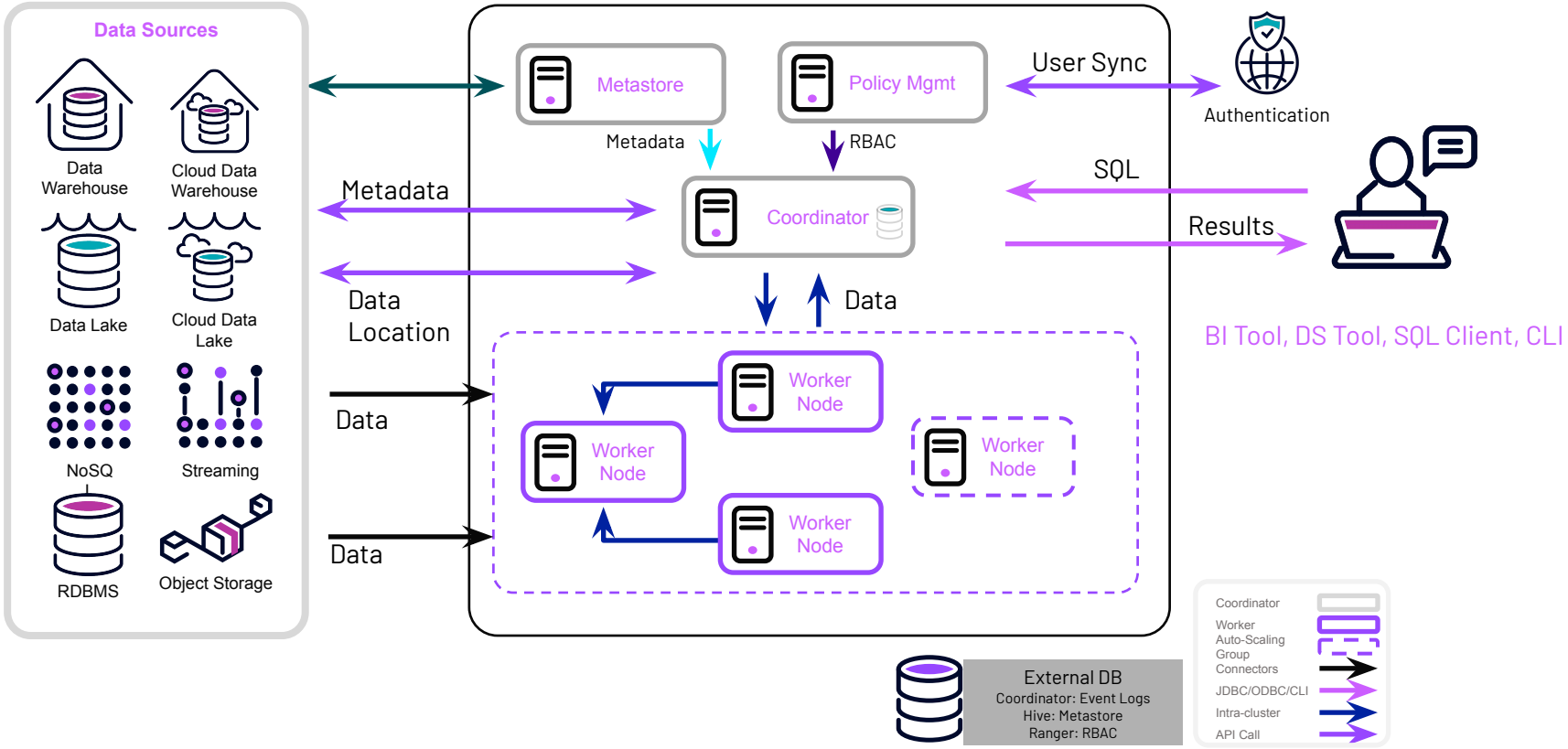
Server that is responsible for parsing statements, planning queries, and managing Trino worker nodes.

Worker nodes

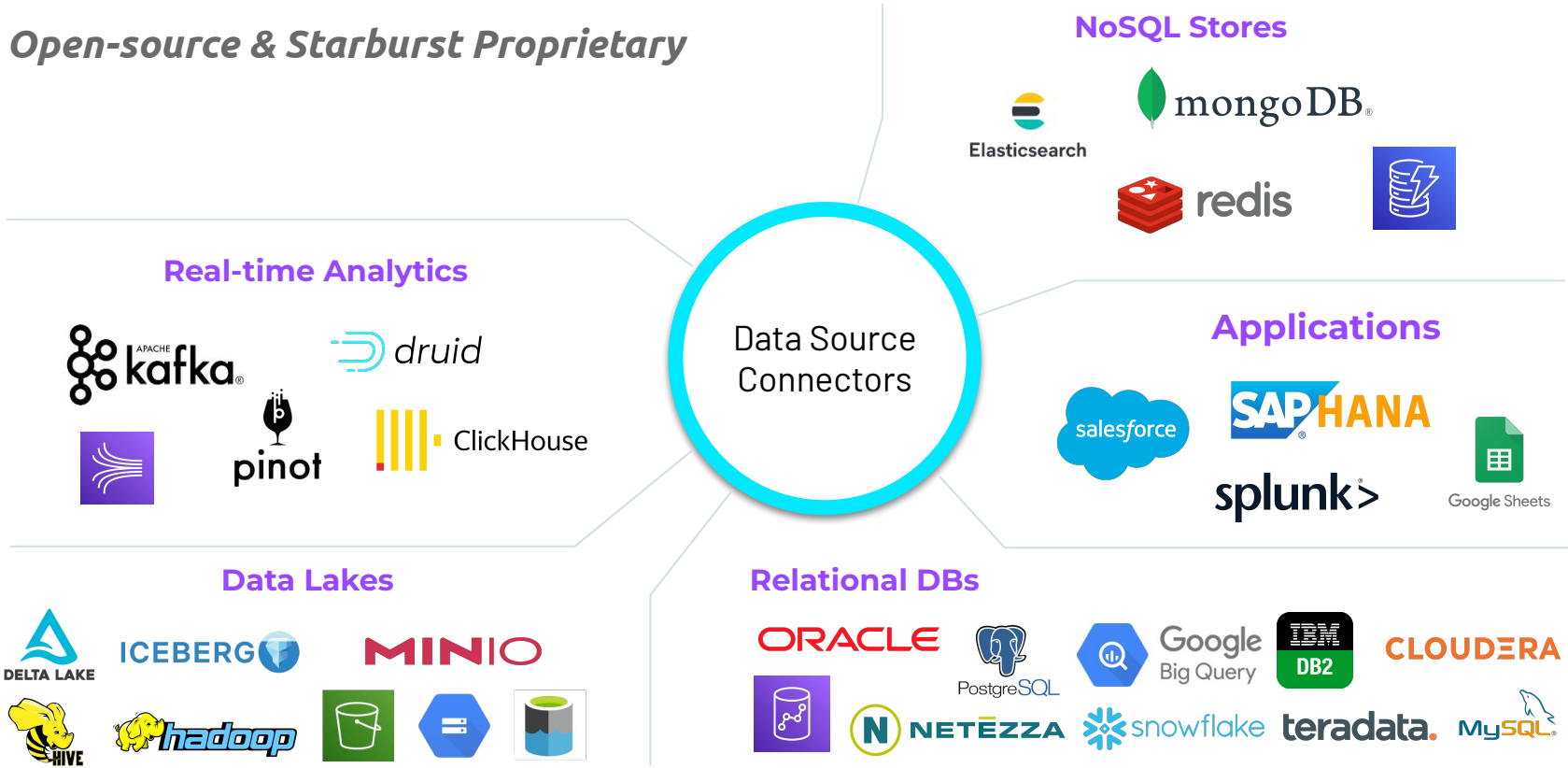
Server which is responsible for executing tasks and processing data. Worker nodes fetch data from connectors and exchange intermediate data with each other.



Logical architecture



Rich ecosystem of data source connectors



History of Trino - ETL processing

From purely interactive use-cases to multiple workloads

2013: Released into production at Facebook for interactive use cases

2014: Users start scheduling batch/ETL queries with Trino instead of Hive

2018: 50% of existing ETL workloads and 85% of new workloads on Trino

Why?

- *Trino can communicate with disparate data sources to federate data*
- *Trino is a distributed, massively parallel processing system*
- *Faster, Cheaper and ANSI-SQL BASED!*

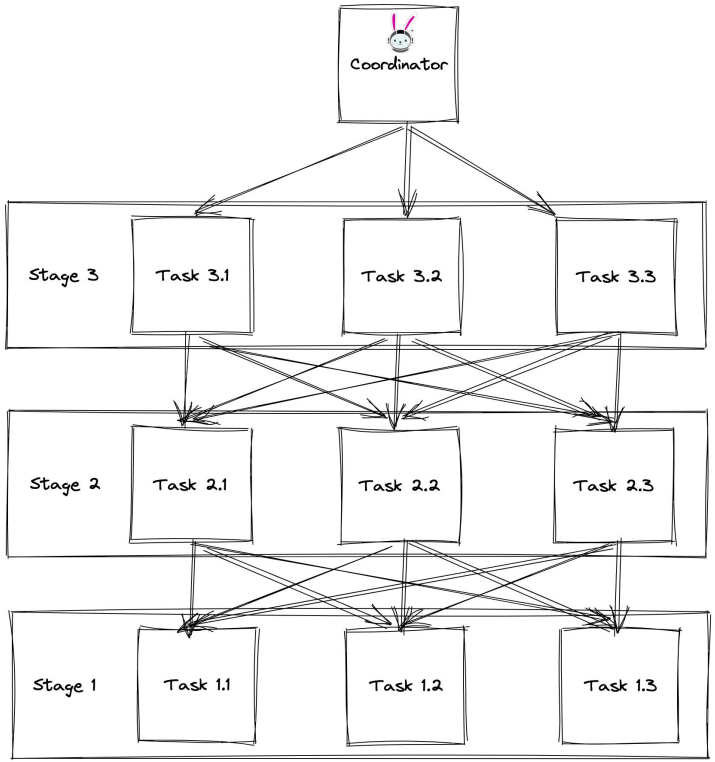
Soon others caught on, and teams like Salesforce and Lyft started utilizing Trino for Batch/ETL capabilities.

ETL concerns with the original architecture

The design goals for interactive querying performance did not provide sufficient support for long-running and memory-intensive queries:

- **Long running queries unreliable**: the all-or-nothing architecture makes it really hard to tolerate faults
- **Distributed memory limit**: with streaming shuffle, aggregations and joins have to process all at once

Also, with original architecture, it's really hard to apply classic techniques like adaptive query execution, speculative execution and skew handling, etc.

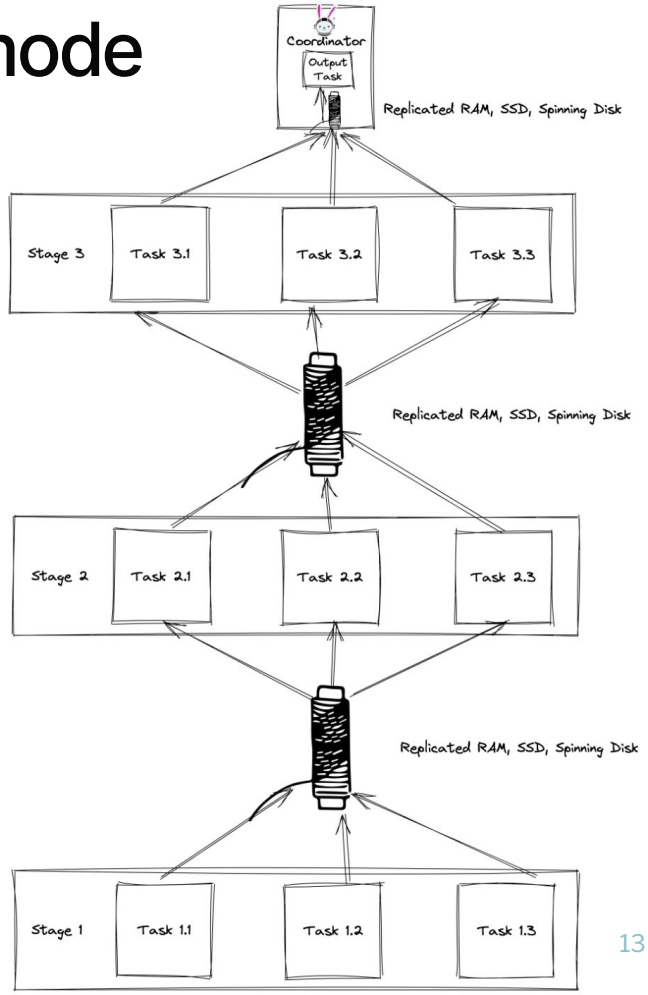


Enter fault-tolerant execution mode

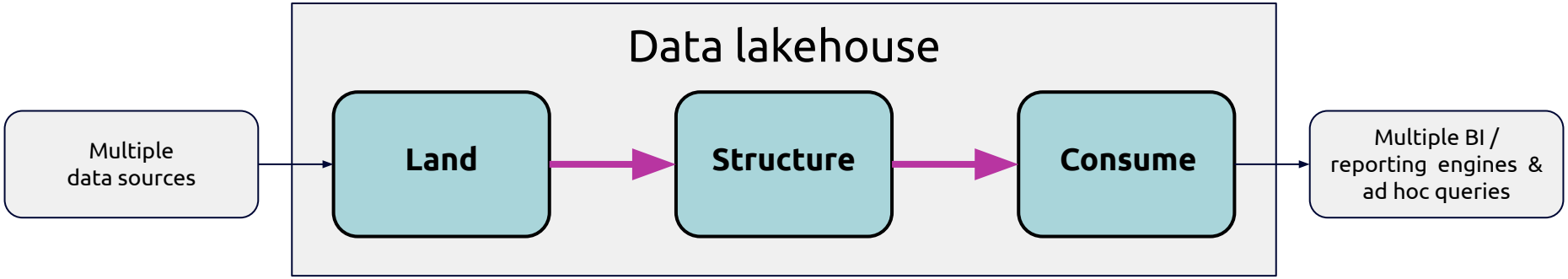
Introduced external exchanges:

- **Independent tasks**
- **Task retries**
- **Resource-aware scheduling**

Stage by stage execution

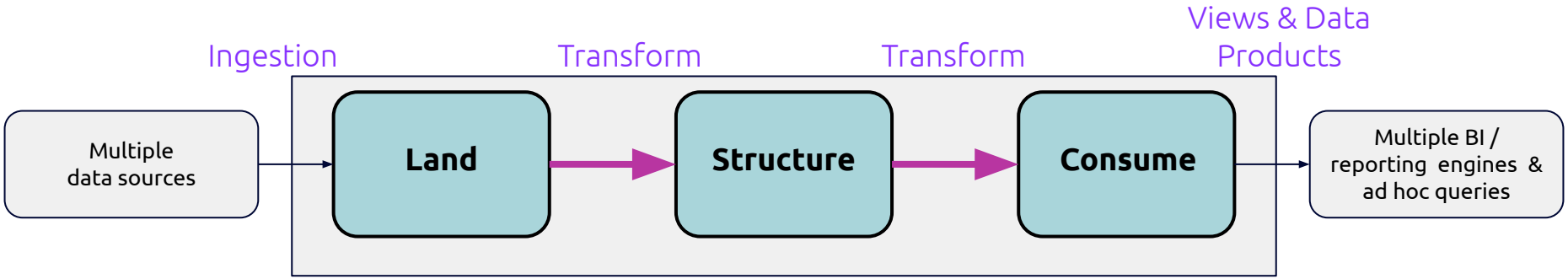


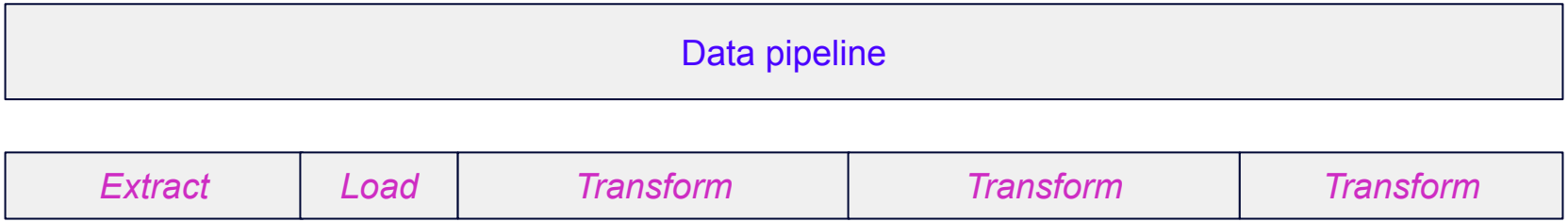
Activities across the medallion architecture



Starburst

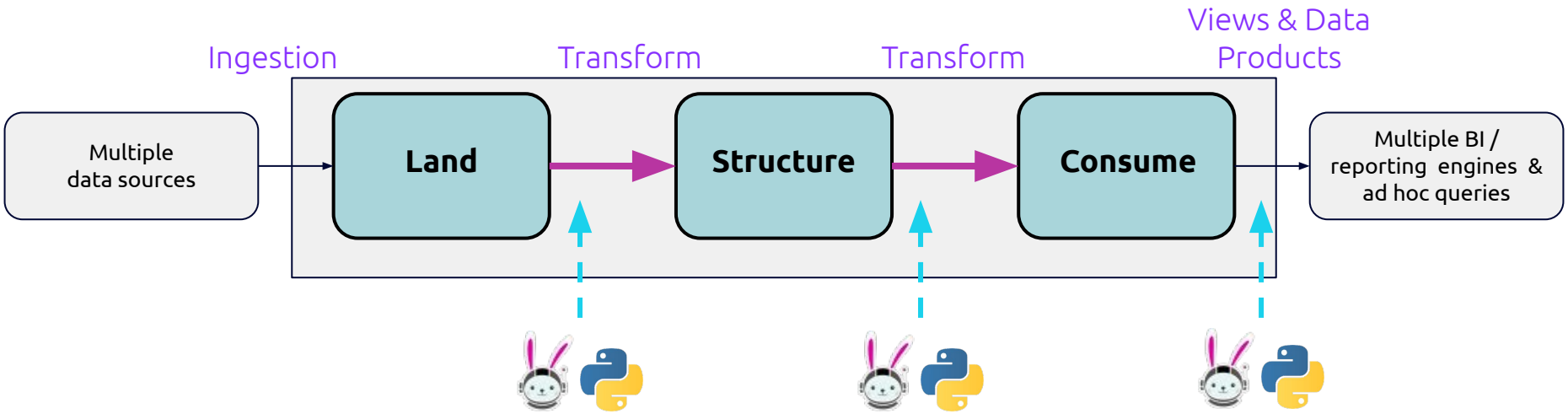
Data pipelines across the medallion architecture





Starburst

Data pipelines with SQL and/or Python



Data pipeline

Extract	*Load*	*Transform*	*Transform*	*Transform*

Starburst

SQL for transformation processing

```sql
SELECT
  CASE
    WHEN product_id IN (1, 2, 3, 4) THEN 'product_one'
    ELSE 'product_two'
  END as product,
  SUM(order_amount) as sales
FROM
  tmp_sales
GROUP BY
  CASE
    WHEN product_id IN (1, 2, 3, 4) THEN 'product_one'
    ELSE 'product_two'
  END
```

Pros

- Compact
- Easy to understand for simple logic

Cons

- No real time debugging
- Can’t perform other actions on results
- No unit tests
- Version Diffs likely hard to read

Complexity makes SQL even more difficult

```sql
SELECT
  order_month,
  order_day,
  COUNT(DISTINCT order_id) AS num_orders,
  COUNT(book_id) AS num_books,
  SUM(price) AS total_price,
  SUM(COUNT(book_id)) OVER (
    PARTITION BY
      order_month
    ORDER BY
      order_day
  ) AS running_total_num_books
FROM
  (
    SELECT
      DATE_FORMAT(co.order_date, '%Y-%m') AS order_month,
      DATE_FORMAT(co.order_date, '%Y-%m-%d') AS order_day,
      co.order_id,
      ol.book_id,
      ol.price
    FROM
      cust_order co
      INNER JOIN order_line ol ON co.order_id = ol.order_id
  ) sub
GROUP BY
  order_month,
  order_day
ORDER BY
  order_day ASC;
```

As complexity grows

- Schema changes are hard to debug
- Logic errors are difficult to find
- Collaboration and sharing is more difficult

Python Dataframes

A Dataframe is a Python data structure that represents a Table that can be operated on

PySpark Example

```python
from pyspark.sql import DataFrame, SparkSession
import spark.sql.functions as F

def read_sales_data(uri: str = 's3a://sales-data/customer-orders/2021/*') -> DataFrame:
    df = spark.read.parquet(uri)
    return df

def define_product(input_df: DataFrame) -> DataFrame:
    output_df = input_df.withColumn('product',
                                    F.when(
                                           F.col('product_id').isin([1,2,3,4]), F.lit('product_one')).otherwise(F.lit('product_two')
                                    )
                )
    return output_df

def agg_sales_by_product(input_df: DataFrame, gb: str = 'product', ag: str = 'order_amount') -> DataFrame:
     output_df = input_df.groupBy(gb).agg(F.sum(F.col(ag).alias('sales'))
     return output_df


df = read_sales_data()
products = define_product(df)
metrics = agg_sales_by_product(products)
```

Pros

- Reusable functions
- Version diffs are easy to read
- Easily create unit tests
- Could combine Python libraries like Numpy, Scipy
- Perform real-time debugging
- Easy to follow complex logic
- Allows for logging

Cons

- A bit verbose

Behind the scenes (1)

SQL Query
Human

Query text

```sql
SELECT
  c.first_name,
  c.last_name,
  c.estimated_income,
  a.products,
  a.cc_number,
  a.mortgage_id,
  cp.customer_segment,
  pp.cc_type
FROM
  glue.burst_bank.customer c
  JOIN mysql.burst_bank.account a on c.custkey = a.custkey
  JOIN mysql.burst_bank.product_profile pp on a.custkey = pp.custkey
  JOIN postgresql.burst_bank.customer_profile cp on c.custkey = cp.custkey
WHERE
  cp.customer_segment = 'diamond'
  and c.estimated_income > 200000
  and pp.cc_type = 'basic'
LIMIT 10
```

SQL Query
Dataframe

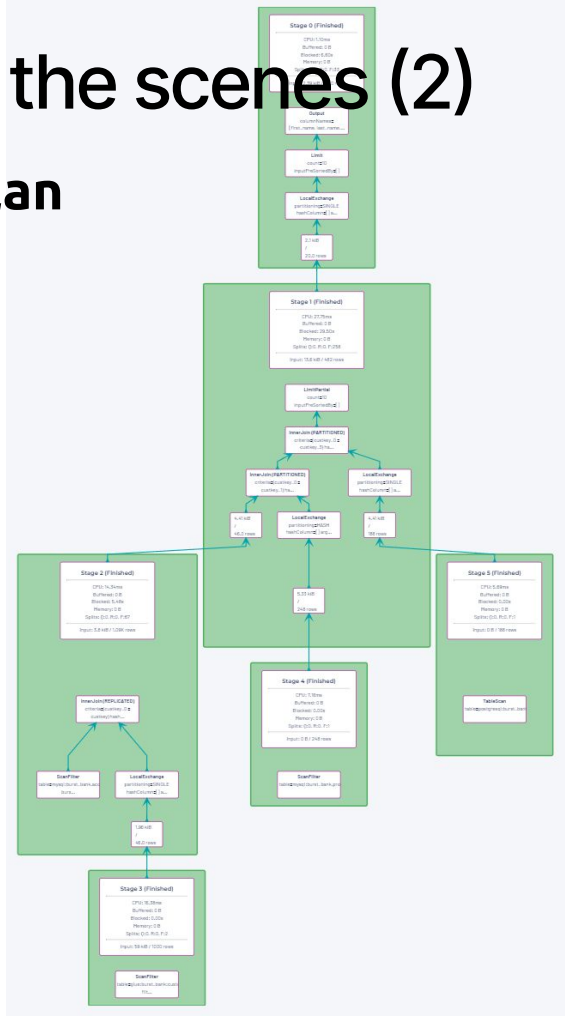
Query text

```sql
SELECT
  "first_name"
, "last_name"
, "estimated_income"
, "products"
, "cc_number"
, "mortgage_id"
, "cc_type"
, "customer_segment"
FROM
  (
   SELECT *
   FROM
     (
      SELECT *
      FROM
        (
         SELECT *
         FROM
           (
            SELECT *
            FROM
              (
               SELECT *
               FROM
                 ((
                  SELECT
                    "l_jnri_custkey" "l_jnri_custkey"
                  , "first_name" "first_name"
                  , "last_name" "last_name"
                  , "estimated_income" "estimated_income"
                  , "r_mkez_custkey" "r_mkez_custkey"
                  , "products" "products"
                  , "cc_number" "cc_number"
                  , "mortgage_id" "mortgage_id"
                  , "custkey" "l_belm_custkey"
                  , "cc_type" "cc_type"
                  FROM
                    (
                     SELECT *
                     FROM
                       ((
                        SELECT
                          "l_jnri_custkey" "l_jnri_custkey"
                        , "first_name" "first_name"
                        , "last_name" "last_name"
                        , "estimated_income" "estimated_income"
                        , "r_mkez_custkey" "r_mkez_custkey"
                        , "products" "products"
                        , "cc_number" "cc_number"
                        , "mortgage_id" "mortgage_id"
                        FROM
                          (
                           SELECT *
                           FROM
                             ((
                              SELECT
                                "custkey" "l_jnri_custkey"
                              , "first_name" "first_name"
                              , "last_name" "last_name"
                              , "estimated_income" "estimated_income"
                              FROM
                                (
                                 SELECT
                                   "custkey"
                                 , "first_name"
                                 , "last_name"
                                 , "estimated_income"
                                 FROM
                                   (
                                    SELECT *
                                    FROM
                                      glue.burst_bank.customer
                                   )
                                )
                             ) INNER JOIN (
                              SELECT
                                "custkey" "r_mkez_custkey"
                              , "products" "products"
                              , "cc_number" "cc_number"
                              , "mortgage_id" "mortgage_id"
                              FROM
                                (
                                 SELECT
                                   "custkey"
                                 , "products"
                                 , "cc_number"
                                 , "mortgage_id"
                                 FROM
                                   (
                                    SELECT *
                                    FROM
                                      mysql.burst_bank.account
                                   )
                                )
                             ) ON ("r_mkez_custkey" = "l_jnri_custkey"))
                          )
                       ) INNER JOIN (
                          SELECT
                            "custkey" "custkey"
                          , "cc_type" "cc_type"
                          FROM
                            (
                             SELECT
                               "custkey"
                             , "cc_type"
                             FROM
                               (
                                SELECT *
                                FROM
                                  mysql.burst_bank.product_profile
                               )
                            )
                       ) ON ("custkey" = "l_jnri_custkey"))
                    )
                 ) INNER JOIN (
                    SELECT
                      "custkey" "r_wza9_custkey"
                    , "customer_segment" "customer_segment"
                    FROM
                      (
                       SELECT
                         "custkey"
                       , "customer_segment"
                       FROM
                         (
                          SELECT *
                          FROM
                            postgresql.burst_bank.customer_profile
                         )
                      )
                 ) ON ("r_wza9_custkey" = "l_jnri_custkey"))
              )
           WHERE ("customer_segment" = 'diamond')
        )
      WHERE ("estimated_income" > CAST(200000 AS BIGINT))
     )
   WHERE ("cc_type" = 'basic')
)
OFFSET 0 ROWS
LIMIT 10
```

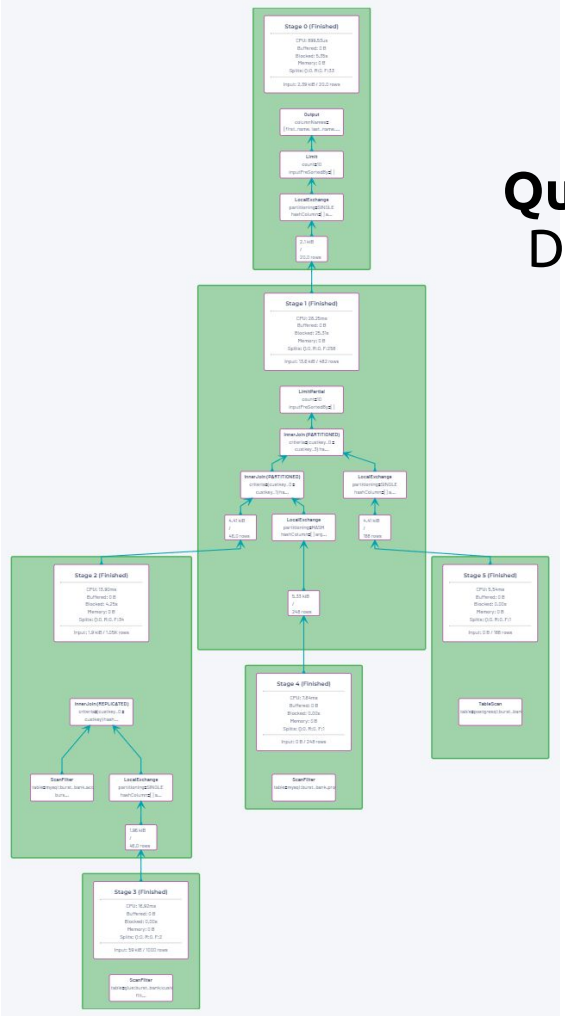
Starburst

Behind the scenes (2)

Query Plan
Human



Query Plan
Dataframe



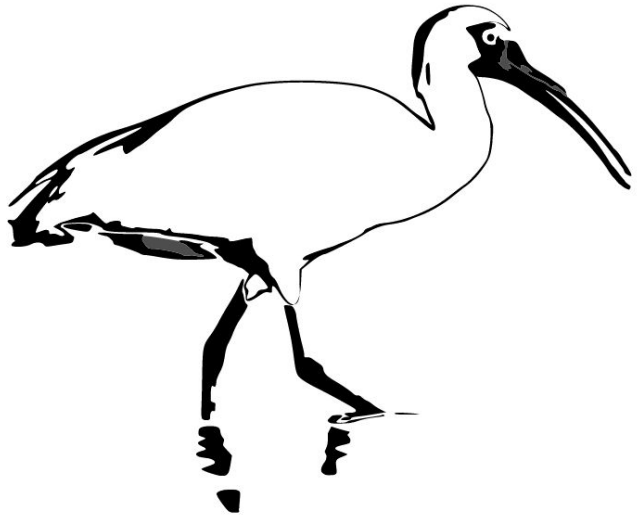
Starburst

Dataframe API options with Trino

PyStarburst

PyStarburst

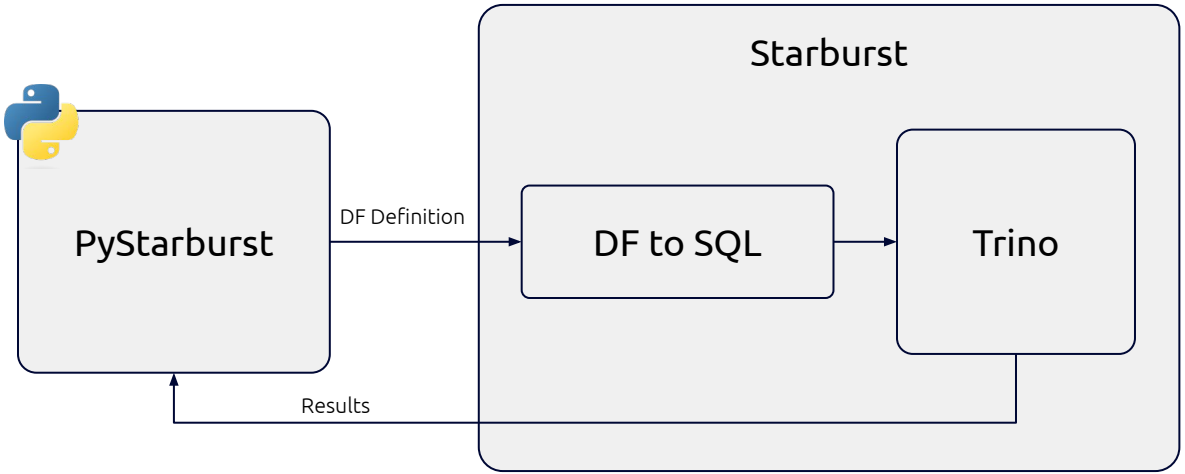
Ibis



Starburst

PyStarburst overview

```
df_missions = df_missions.with_column("date", f.sql_expr("COALESCE(TRY(date_parse(\"date\", '%a %b %d, %Y %H:%i UTC')), NULL)"))

print(df_missions.schema)

df_missions = df_missions\
    .filter(col("date") > datetime(2000, 1, 1))\
    .sort(col("date"), ascending=True)

df_missions.show()
```



- PySpark-like Syntax
- Lazy execution
- Python gets converted to SQL
- Heavy lifting done by Trino

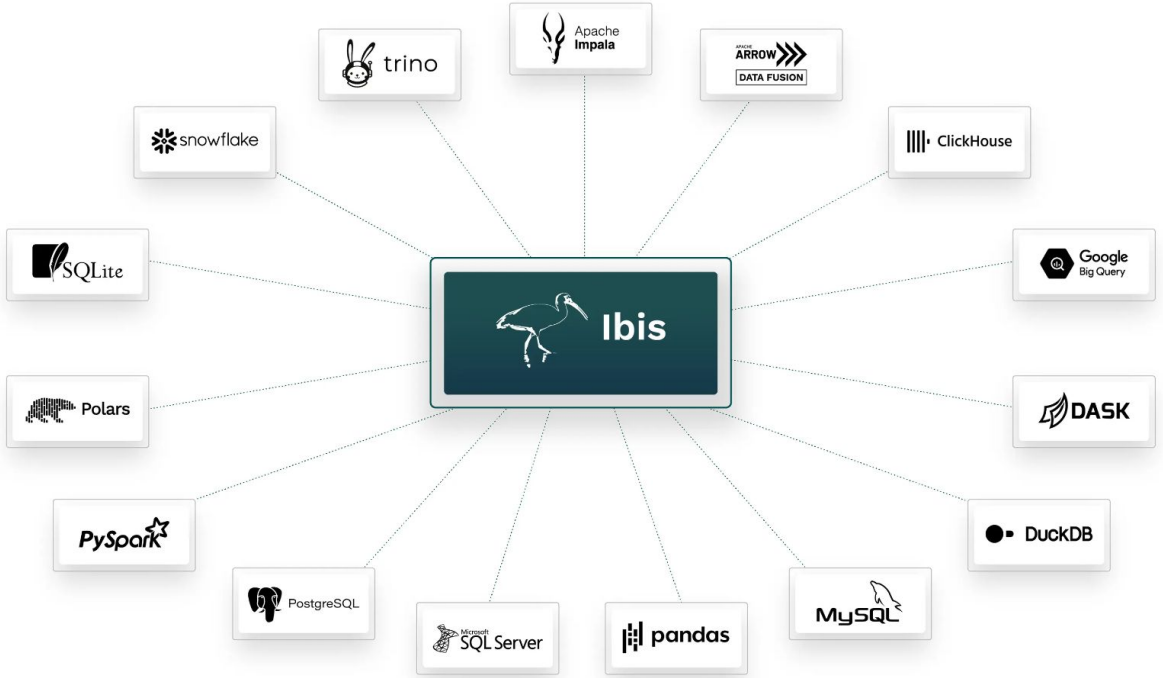
Links: API Docs & Example Code

Only supported on Starburst Galaxy and Starburst Enterprise

Starburst

Ibis overview

A common Dataframe API for data manipulation in Python, and compiling that API into a variety of backend native languages



Starburst

Similar APIs

PyStarburst

```
nationDF = session.table("tpch.tiny.nation") \
            .drop("regionkey", "comment") \
            .rename("name", "nation_name") \
            .rename("nationkey", "n_nationkey")
```

```
custDF = session.table("tpch.tiny.customer") \
            .select("name", "acctbal", "nationkey") \
            .filter(col("acctbal") > 9900.0)

apiSQL = custDF.join(nationDF,
    col("nationkey") == nationDF.n_nationkey) \
            .drop("nationkey").drop("n_nationkey") \
            .sort(col("acctbal"), ascending=False)

apiSQL.show()
```

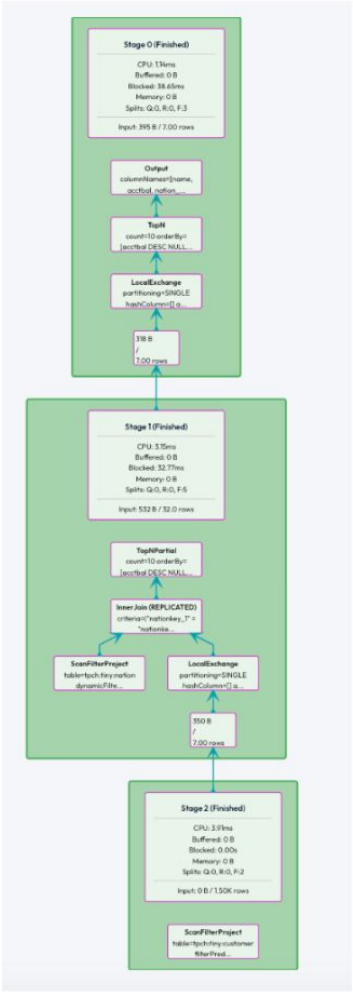
Ibis

```
nationDF = con.table("nation") \
            .drop("regionkey", "comment") \
            .rename(
                dict(
                    nation_name="name",
                    n_nationkey="nationkey"
                )
            )
```

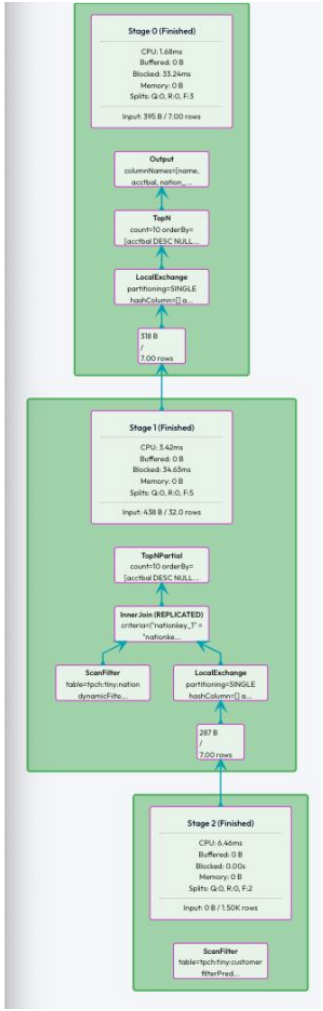
```
custDF = con.table("customer") \
            .select("name", "acctbal", "nationkey") \
            .filter(projectedDF["acctbal"] > 9900.0)

apiSQL = custDF.join(nationDF,
    custDF.nationkey == nationDF.n_nationkey) \
            .drop("nationkey", "n_nationkey") \
            .order_by([ibis.desc("acctbal")])

print(apiSQL.head(10))
```

Similar results

Query Plan
PyStarburst



Query Plan
Ibis



Starburst

Trino for your data pipelines

Trino is, and has been, an appropriate clustering technology for running you transformation processing jobs

- Fault-tolerant execution mode makes it even more robust
- Data engineers can leverage SQL and/or Python

trino

Starburst

Dataframe API recommendations

If you use Starburst products → Choose PyStarburst

- More similar to PySpark Dataframe API
- More likely to be optimized better for Trino than Ibis

If you are using OSS Trino → Choose Ibis

- PyStarburst is not supported here
- Gain optionality for backend execution engine

Thank You

Starburst

Starburst

Rocketfuel for AI

starburst.ai