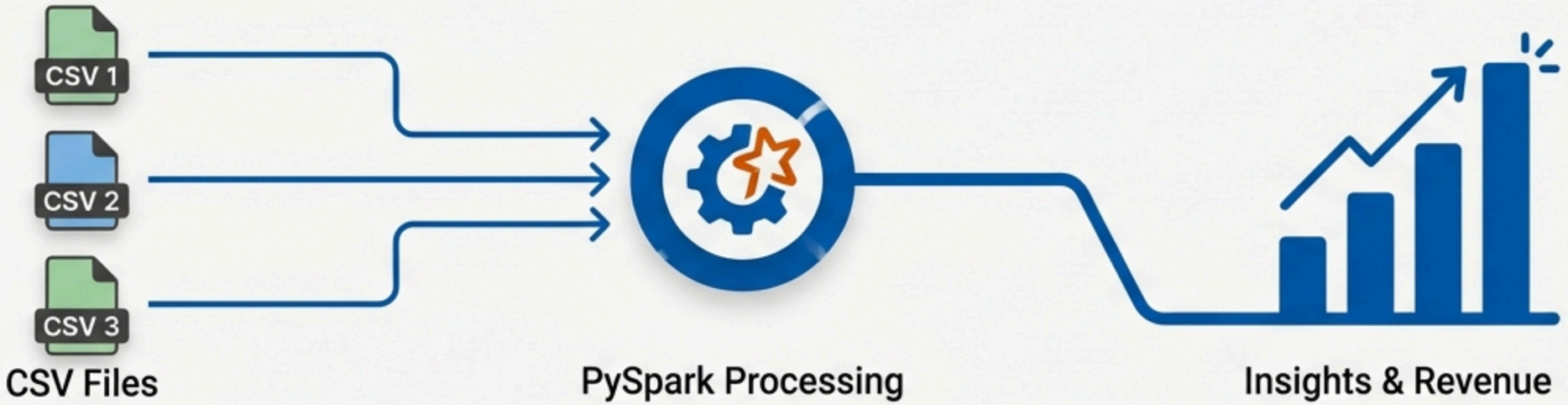
Unlocking Holiday Insights: A PySpark Case Study

From Raw CSVs to Revenue Optimization



Context

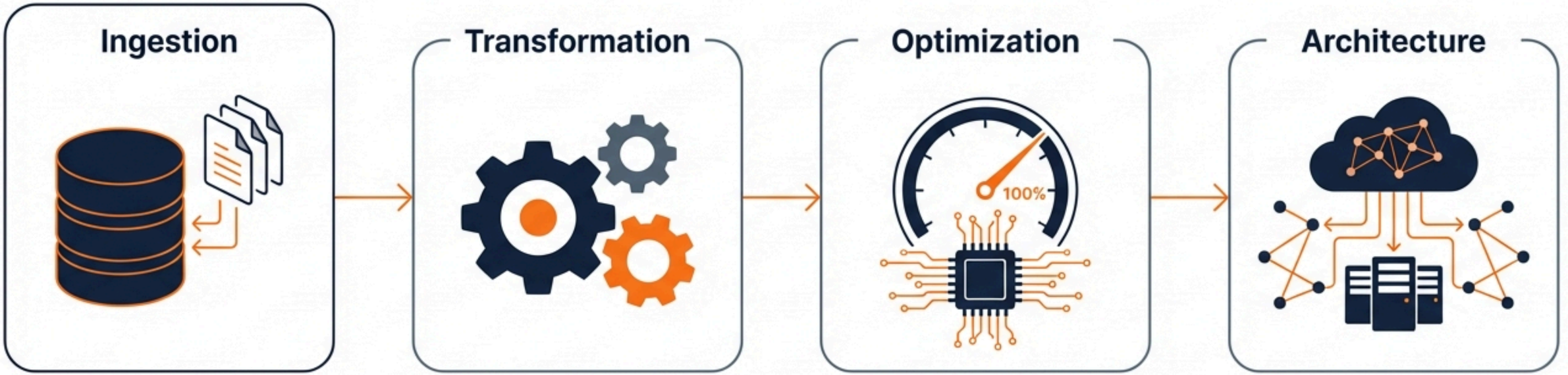
An analysis of user behavior and sales performance using the **"eCommerce Events History in Electronics Store"** dataset (**October & November 2019**).

Objectives

1. **Engineer:** Unify fragmented monthly logs into a seamless timeline.
2. **Analyze:** Track user loyalty and recency using `Window functions`.
3. **Optimize:** Calculate **conversion rates** and identify **revenue-driving** products.

PySpark Masterclass: From Efficient Code to Predictive Architecture

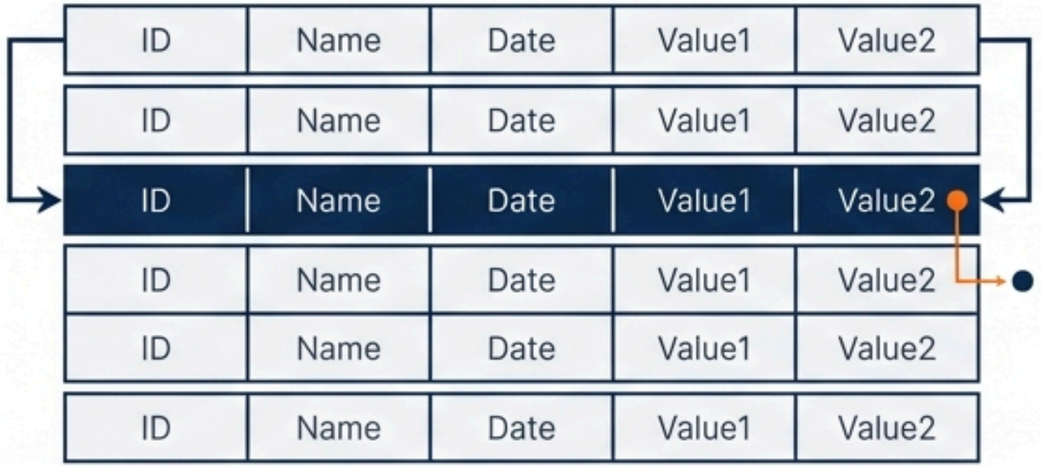
A comprehensive guide to the Data Engineering Lifecycle: Ingestion, Transformation, Optimization, and Deployment.



This guide deconstructs the mechanics of high-performance data pipelines, moving from storage format decisions to complex join logic, memoʙiry management, and finally, real-world e-commerce architecture on Databricks.

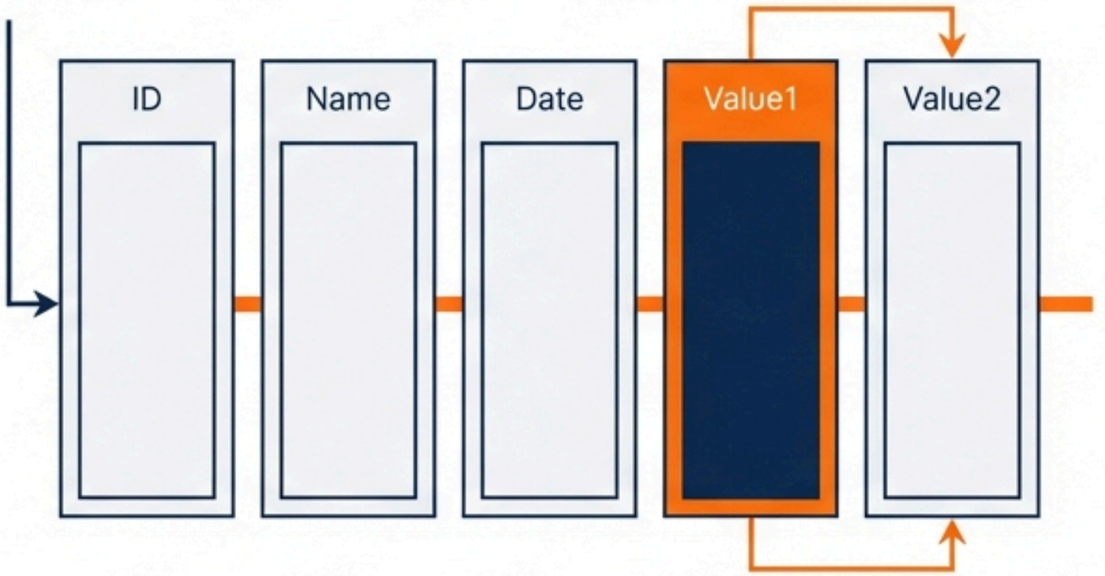
Storage Strategy: Why Parquet is the Gold Standard

Row-Based (CSV)



- Human-readable & universal.
- No compression (**Expensive storage**).
- **No schema** enforcement (**Error-prone**).
- **High I/O**: Reads entire rows even for single-column queries.

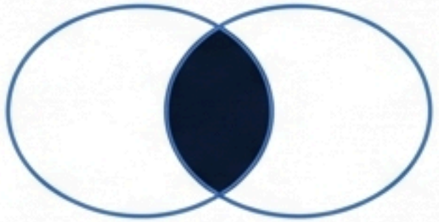
Columnar (Parquet)



- **Predicate Pushdown**: Skips unnecessary data using **metadata/indexes**.
- **High Efficiency**: Up to **10x smaller** via **Snappy**/**Gzip** compression.
- **Schema**: Self-describing with **embedded types**.

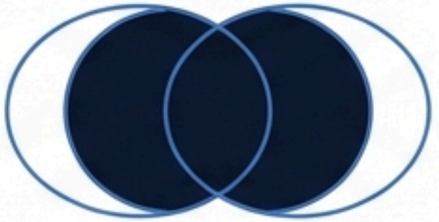
The PySpark Join Spectrum

Inner Join



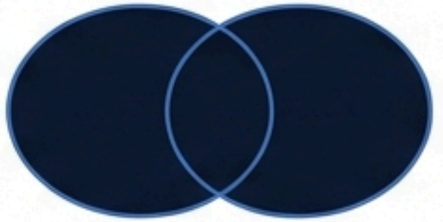
Returns rows with matches in BOTH tables. (Default)

Left/Right Outer



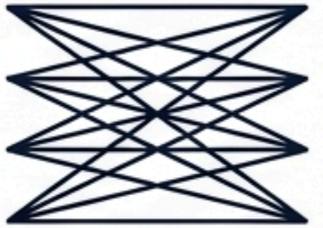
Returns all rows from one side, filling non-matches with Nulls.

Full Outer



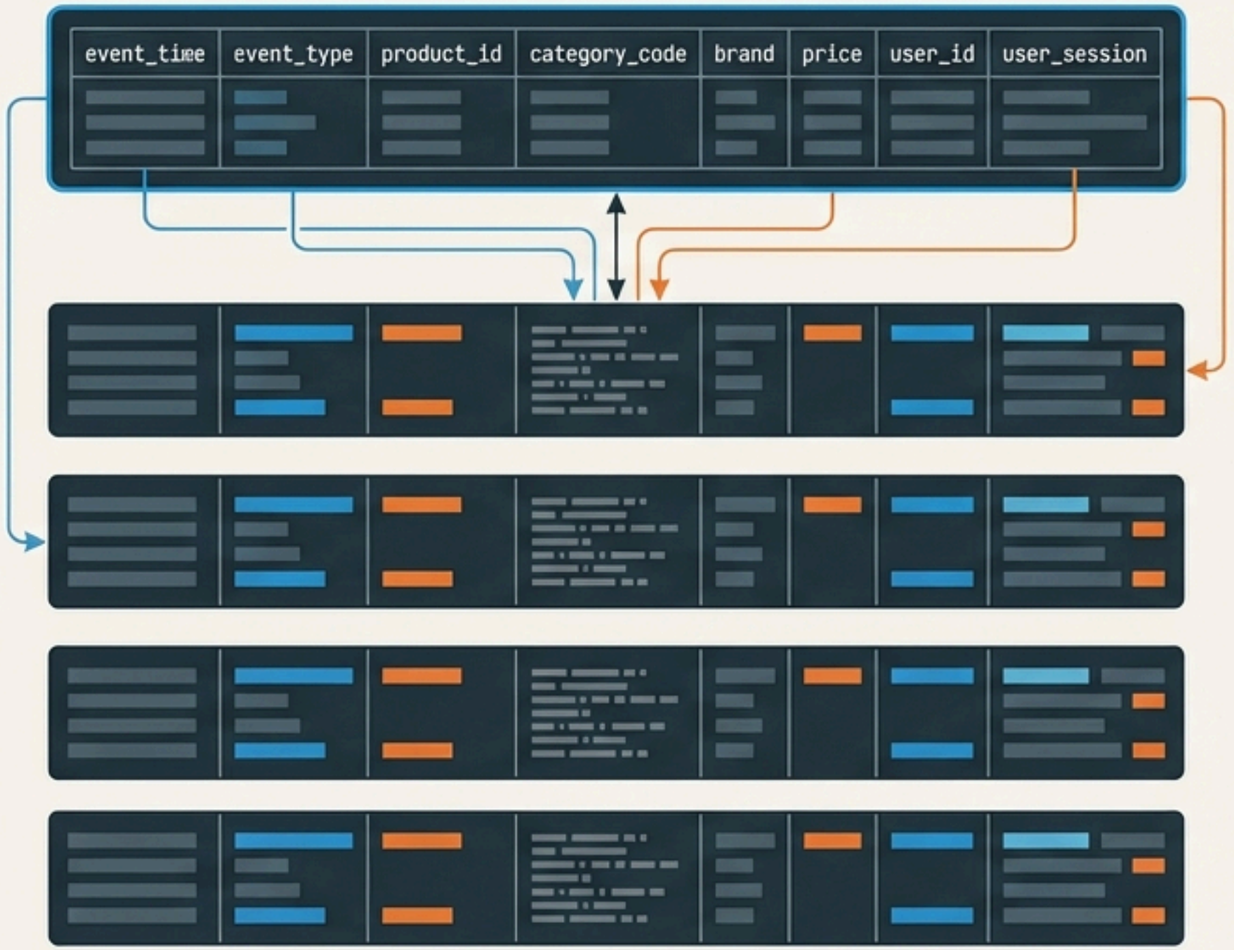
Returns all rows from BOTH sides.

Cross Join



Cartesian product. Computationally expensive. Avoid unless necessary.

```
df1.join(df2, join_condition, join_type)
# join_type options: 'inner', 'cross', 'outer', 'left', 'right', 'semi', 'anti'
```



The Raw Material: High-Volume Event Logs

The Challenge: Data arrives in monthly partitions (2019-oct.csv, 2019-nov.csv). To analyze the full Q4 trend, these distinct files must be treated as a single continuous stream.

The Scale: Millions of rows representing user interactions ranging from simple 'views' to 'purchases'.

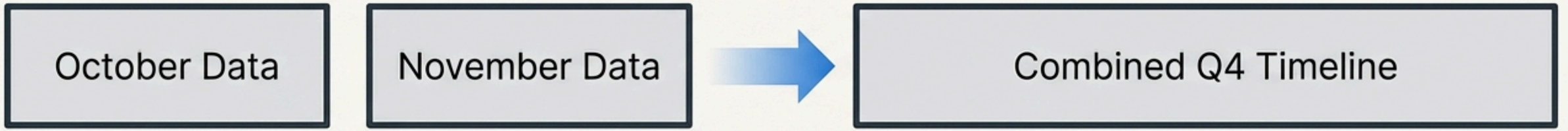
Key Columns:

- **event_type**: The user action (view, cart, purchase).
- **user_session**: The unique session ID for tracking temporary journeys.
- **price**: The financial value of the interaction.

Unifying the Timeline

```
events = events_oct.unionByName(events_nov)
```

Matches columns by name rather than position, preventing schema mismatch errors when files evolve.



To build a **holistic view** of the holiday season, we merge the October lead-up with the November peak. This creates a single source of truth for all subsequent analysis, utilizing the `unionByName` **syntax** for safe integration.

Handling Large Dataframes: The “Limit” Trap

THE TRAP

```
ERROR: OutOfMemoryError - Driver display limit exceeded. Cannot collect 10,000,000 rows to the driver.
```

When working with massive datasets, attempting to collect all data to the driver for display can crash the notebook or cause severe latency.

THE FIX

```
1
2  display(events.limit(10))
3
4
```

The Lesson: Chaining **.limit()** before execution is crucial for performance. Always explicitly limit the output when previewing transformations.

Measuring Loyalty: The Running Total

```
window = Window.partitionBy('user_id')
    .orderBy('event_time')

result = events.withColumn('cumulative_events',
    F.count('*').over(window))
```

Translator

Treat each user as an isolated timeline.

Translator

Sort their actions chronologically.

Translator

Count how many actions have occurred up to *this* specific moment.

By calculating a running total of events per user, we can identify high-frequency visitors and spot where engagement accelerates or drops off.

The User Journey in Data

event_time	event_type	product_id	category_code	brand		price	user_id	user_session	cumulative_events
2019-11-24T16:47:32.000+00:...	view	1004749	electronics.smartphone	xiaomi		239.36	9458479	ea6abb356-512a-4472-9d36-62e00a5fd01c	1
2019-11-28T08:51:47.000+00:...	view	4900378	20530135552208408...	appliances.kitchen.juicer	scarlett	112.99	12238479	c04d12ef-da1c-4e4a-b446-8f1bb5a81e35	2
2019-11-28T08:25:08.000+00:...	view	12702958	20530135558598963...	null	cordiant	42.47	12238479	6cee7edb-68ae-4bea-88ac-e0efd7d8c8f	3
2019-11-28T08:32:51.000+00:...	view	4900173	20530135552208408...	appliances.kitchen.juicer	moulinex	102.94	12238479	6cee7edb-68ae-4bea-88ac-e0efd7d8c8f	4
2019-11-28T08:11:30.000+00:...	view	4900173	20530135552208408...	appliances.kitchen.juicer	moulinex	102.94	12238479	6cee7edb-68ae-4bea-86ac-e0efd7d8c8f	5

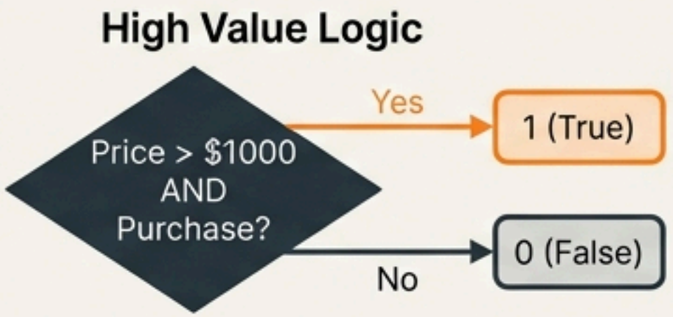
Observation:
The cumulative_events column acts as a ‘session depth' meter.

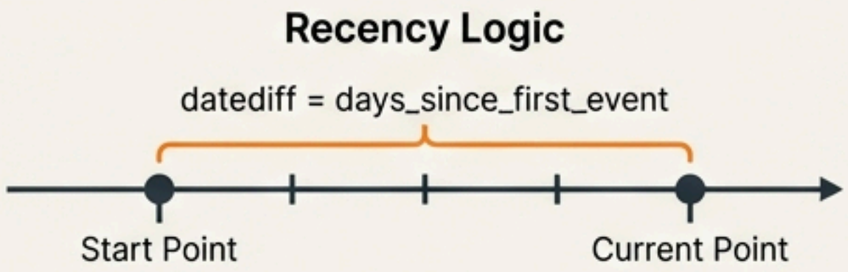
Business Application:
Users with high cumulative counts in short timeframes can be targeted with real-time 'nudge' offers, while low-count users may need re-engagement campaigns.

Editorial Technical

Advanced Segmentation: Recency & High-Value Logic

```
events.withColumn(
    'is_high_value_purchase',
    F.when((F.col('event_type') == 'purchase') & (F.col('price') > 1000), 1).otherwise(0)
).withColumn(
    'days_since_first_event',
    F.datediff(F.col('event_time'), F.first('event_time').over(window))
)
```



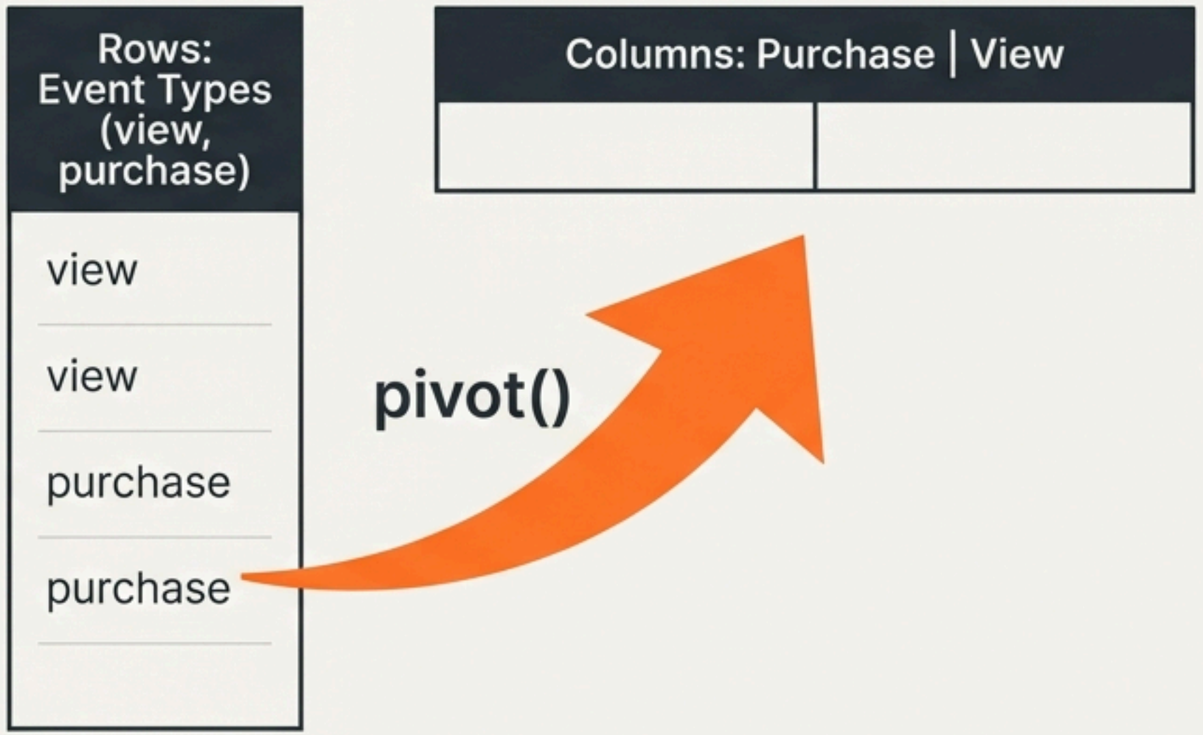


Defining High Value: We programmatically tag purchases over $1,000.
Defining Recency: We pin the user's start date to calculate exactly how long a user has been in our ecosystem.

Editorial Technical

The Conversion Funnel: Pivoting for Clarity

```
events.groupBy('category_code')
    .pivot('event_type', ['purchase', 'view'])
    .agg(F.count('event_time'))
```



- **The Goal:** Compare "window shoppers" (Views) against "buyers" (Purchases).
- **The Technique:** The pivot transformation reshapes the data, turning unique values in the `event_type` column into their own distinct columns.

Editorial Technical

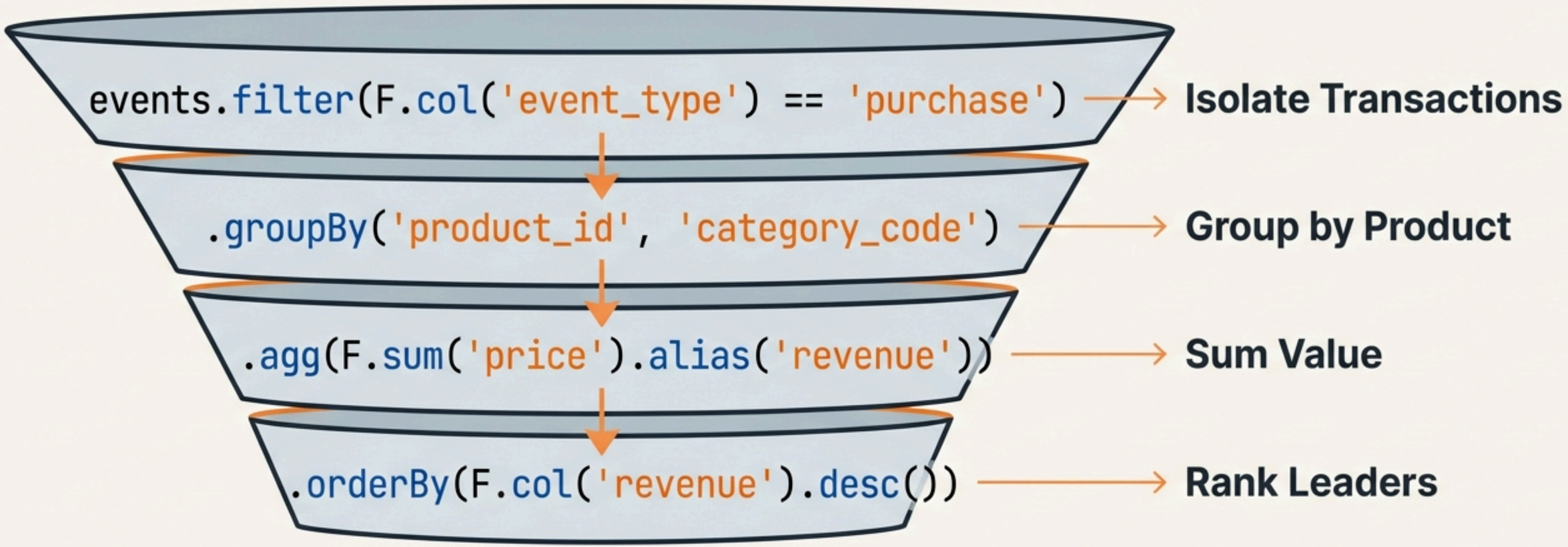
Category Performance: Volume vs. Conversion

category_code JetBrains Mono Deep Charcoal	purchase JetBrains Mono Deep Charcoal	view JetBrains Mono Deep Charcoal	conversion_rate JetBrains Mono Deep Charcoal
electronics.smartphone	1004870	3030325	0.3316
appliances.kitchen.washer	7	1750	0.4000
accessories.wallet	366	68909	0.5311

The Formula:
(purchase / view) * 100

Insight: High traffic doesn't guarantee high sales. While electronics drive millions of views, niche appliances often show stronger intent-to-buy ratios.

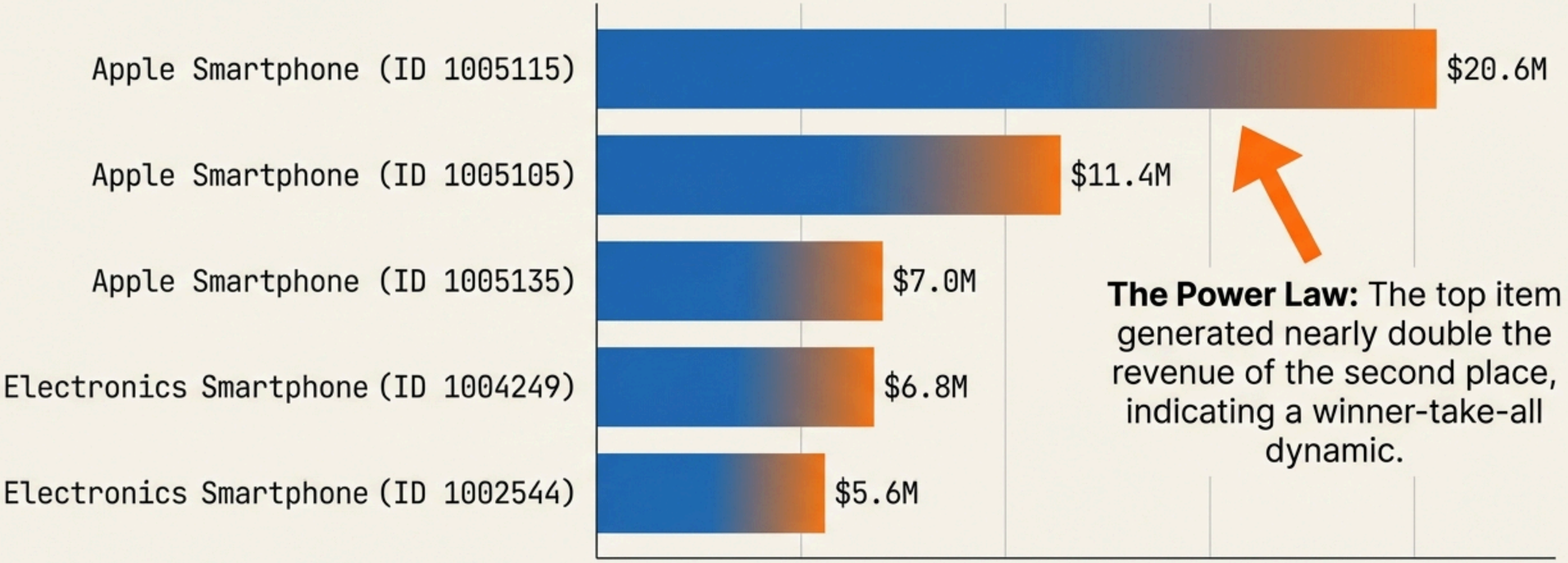
Revenue Analysis: Filtering for Value



This query cuts through the noise of millions of views to identify the core economic engine of the holiday season.

"The Revenue Kings"

Top 5 Products by Revenue



From Code to Strategy

Technical Toolkit

- `unionByName`: Unifying fragmented time-series data.
- **Window Functions**: Tracking user lifecycles (Loyalty/Recency).
- `pivot`: Creating funnel metrics (Conversion Rates).

Strategic Insights

- **Volume ≠ Value**: High view counts often have lower conversion rates.
- **The 1% Rule**: Revenue is heavily concentrated in flagship smartphones ($20M+ per item).
- **Segmentation**: Identifying 'High Value' users (> $1000 spend) allows for VIP targeting.

PySpark provides the scalability to turn millions of raw logs into a clear roadmap for Q4 strategy.