MLB Game Simulation Dashboard: A Lambda Architecture Approach

Aadarsha Gopala Reddy
Department of Computer Science
Washington University in St. Louis
St. Louis, MO, USA
a.gopalareddy@wustl.edu

Eddy Sul
Department of Computer Science
Washington University in St. Louis
St. Louis, MO, USA
eddysul@wustl.edu

***Abstract*—We present an interactive analytics dashboard for Major League Baseball (MLB) using *lambda architecture*. Our system combines batch processing of historical STATCAST data (2015–2025) with a streaming layer that enables realistic game simulation. The pipeline uses KAFKA for stream processing, SPARK for micro-batch operations, SNOWFLAKE for data warehousing, and AIRFLOW for orchestration. Users can explore historical games, analyze team matchups, and simulate past games pitch-by-pitch with configurable playback speeds.**

***Index Terms*—baseball analytics, real-time streaming, lambda architecture, Apache Kafka, Spark Streaming, data visualization**

I. INTRODUCTION

The introduction of Statcast in 2015 transformed statistical analysis across major league baseball (MLB). Prior to its introduction, most of the data in mlb consisted of static, historical box-score data. Statcast introduced a shift from 2D to 3D analytics, capturing granular pitching metrics such as velocity, spin rate and for every batted ball including exit velocity and launch angle. The abundance of new data allows teams to develop new strategies to gain a competitive edge in the league.

Most of this data can be found on Baseball Savant, which provides advanced analytics and visualizations in baseball. Also on mlb.com, viewers can watch live game simulations in real-time using a feature called MLB gameday. MLB gameday focuses on the current pitch in the game and real-time streaming but lacks historical analysis of data. Therefore, we wanted to create an interactive dashboard that handles both live data and provides statistical analysis backed by historical data on each data point entering the system. Specifically, during live simulation we wanted to add aggregational statistics showing each hitter's percentages on specific pitches the pitcher throws and the ability to live simulate games in the past even after they have finished.

II. SYSTEM ARCHITECTURE

Our system follows lambda architecture (Fig. 1), combining two processing layers:

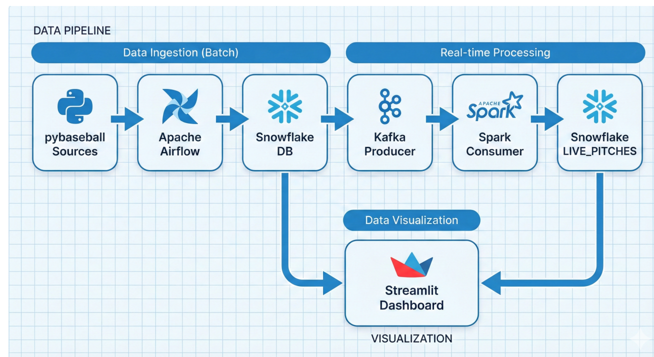

Fig. 1. Lambda architecture overview with batch and streaming layers.

- **Batch Layer:** Daily ingestion of historical data via AIRFLOW DAGs, stored in SNOWFLAKE using a star schema.
- **Streaming Layer:** KAFKA and SPARK Streaming process pitch events for game simulation updates.

III. DATA PIPELINE

A. Ingestion and Transformation

We use `pybaseball`, a Python wrapper for the MLB API, to fetch pitch-by-pitch logs. Each pitch contains 118+ fields; we filter to retain only analysis-relevant metrics. It doesn't provide real-time data from live games, but it does provide the most comprehensive data dating back to 2015, and it is updated daily. Other websites and APIs could provide real-time data, but were either not comprehensive and/or compatible with pybaseball, or were not easily accessible.

The processing workflow:

1. Fetch daily data via `pybaseball`
2. Transform: remove deprecated fields, nulls, and out-of-scope metrics
3. Stage as Parquet files in SNOWFLAKE
4. Load into star schema tables
5. Execute data quality checks

B. Database Schema

We employ a *star schema* optimized for OLAP queries:

- **Fact table:** `fact_pitches` — one row per pitch
- **Dimensions:** `player`, `team`, `pitch_type`, `game`

C. Orchestration

AIRFLOW DAGs manage the pipeline (Fig. 2–3):

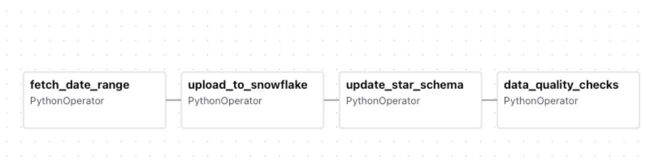

Fig. 2. Daily ingestion DAG.

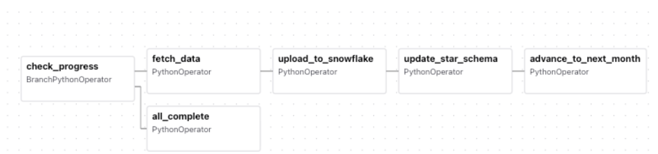

Fig. 3. Historical backfill DAG (runs every 3 minutes).

IV. Streaming Layer

The streaming pipeline consists of three components:

1. **FASTAPI Producer:** Queries SNOWFLAKE and publishes pitch events to KAFKA
2. **KAFKA Broker:** Buffers events (Docker container with Zookeeper)
3. **SPARK Consumer:** Processes micro-batches and writes to `Live_Pitches`

Decoupling Strategy: STREAMLIT exhibits refresh latency that would cause pitch skipping. We decoupled the dashboard from the producer, making the dashboard poll `Live_Pitches` independently, thereby ensuring every pitch displays accurately at the cost of slightly higher latency.

Trade-off: Data accuracy over minimal latency. Missing pitches would confuse viewers; slight delay is acceptable.

V. Dashboard Features

A. Game Explorer

Select historical games and analyze pitcher statistics: velocity per pitch type, spin rate, and strikeout percentage. Includes strike zone visualization and velocity progression tracking.

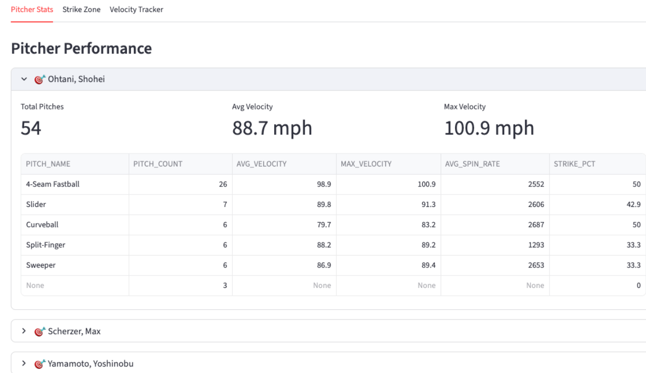

Fig. 4. Pitcher repertoire breakdown and key statistics.

Fig. 5. Pitch velocity progression throughout the game.

B. Team Matchups

Compare head-to-head statistics between teams across a season (Fig. 6).

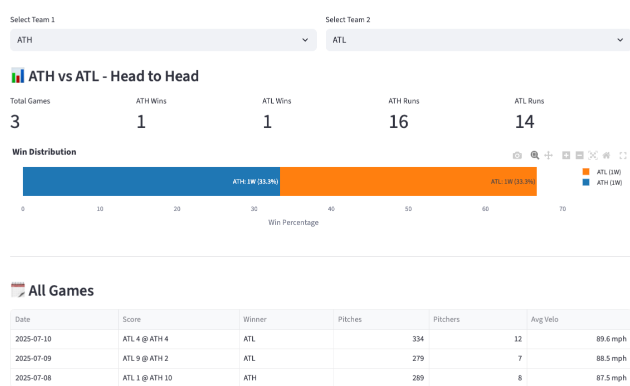

Fig. 6. Team matchup: pitcher vs. batter statistics.

C. Game Simulation

Watch historical games unfold pitch-by-pitch with playback controls (0.5–5 pitches/second). Allows replaying any past game. One thing to note is that the replay is not to scale with the real game, but it does provide a good visualization of the game.

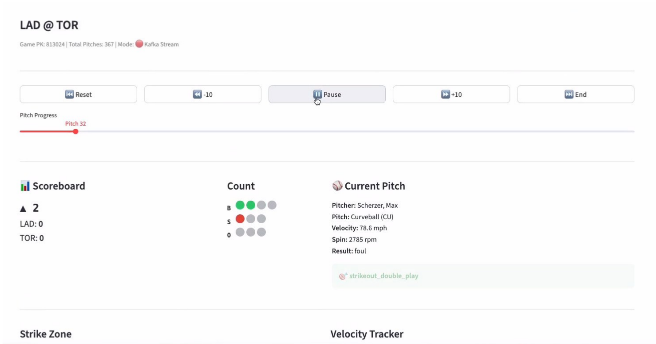

Fig. 7. Game simulation with game state and playback controls.

VI. Challenges and Design Decisions

A. Schema Design

With 110+ fields per pitch, we invested significant effort selecting relevant columns and designing intuitive dimension tables.

B. Views vs. Materialized Tables

We chose SNOWFLAKE views for dashboard aggregations:

TABLE I
VIEWS VS. MATERIALIZED TABLES TRADE-OFFS

Factor	Views	Materialized
Auto-sync with data	✓	Requires refresh
Query performance	Slower	Faster
Staleness risk	None	Possible
Our choice	✓	—

For our workload (frequent writes, user-triggered reads), views provided acceptable latency with guaranteed consistency.

C. Dashboard Latency

STREAMLIT, depending on the system it is running on, imposes $\sim$1 second minimum refresh. Our KAFKA producer can emit faster, but the dashboard cannot display sub-second updates. WebSocket-based alternatives (Dash, React) could improve this.

D. Historical Data Range

We focused on 2020–present for two reasons:

1. Data quality improved post-2019 with upgraded tracking hardware
2. Smaller backfill enabled faster development iteration

The pipeline supports 2015+ data with minimal modification.

E. Infrastructure Issues

AIRFLOW exhibited significant bugs and dependency issues when migrating from local device to the LinuxLab environment.

VII. Future Work

A. True Real-Time Integration

Current limitation: `pybaseball` provides data with $\sim$1-day delay.

Required changes for live-season support:

1. WebSocket connection to MLB Gameday API
2. KAFKA producer emitting events as they arrive
3. Reconciliation logic: merge real-time events with next-day STATCAST data

Our lambda architecture already supports this—streaming layer handles live events; batch layer provides quality-assured corrections.

B. Data Corrections

MLB issues scoring changes and data corrections. Potential solutions:

- Change Data Capture (CDC) to detect source modifications
- Targeted updates to affected records
- Handle suspended games spanning multiple dates

C. Performance Optimizations

- **REDIS caching:** Lower latency for live updates
- **FLINK streaming:** Event-by-event processing (vs. SPARK micro-batches)
- **Materialized views:** For high-concurrency scenarios

D. Additional Data Sources

TABLE II
POTENTIAL DATA SOURCES AND APPLICATIONS

Data Type	Source/Library	Application
Weather	OpenWeatherMap API, Visual Crossing	Correlate temperature, humidity, and wind with pitch movement and home run distances
Stadium Info	MLB Park Factors, Chadwick Bureau	Normalize statistics across venues; account for altitude effects (e.g., Coors Field)
Historical Stats	Baseball Reference, Lahman Database	All-time head-to-head matchups; career splits analysis
Player Tracking	MLB Film Room API	Defensive positioning; sprint speed correlations

These integrations would enable advanced analytics such as weather-adjusted ERA, venue-normalized exit velocity, and predictive matchup modeling based on historical tendencies.

E. Planned Features

Batter splits against specific pitch types:

Ohtani vs. Righty Sliders: .333 Average

VIII. Demonstration

Video demo: https://www.youtube.com/watch?v=z01fHvzd-8w

IX. Conclusion

We presented an MLB game simulation dashboard combining batch and streaming processing via lambda architecture. The system demonstrates effective integration of Kafka, Spark, Snowflake, and Airflow for interactive baseball analytics, enabling users to replay historical games and explore detailed statistics. Future work includes true real-time integration during live seasons.

References

[1] MLB Advanced Media, "Statcast," https://www.mlb.com/glossary/statcast.

[2] Baseball Savant, "Statcast Search," https://baseballsavant.mlb.com/statcast_search.

[3] pybaseball, "Python package for baseball data," https://github.com/jldbc/pybaseball.

[4] Apache Kafka, https://kafka.apache.org/.

[5] Apache Spark, https://spark.apache.org/.

[6] Snowflake Inc., https://www.snowflake.com/.