**Pulsar-Flink-Postgres implementation:**

**Key Clarifications from Your Rules**

1. **Expired vs Active**
   * Only trades with expired\_flag = false (active) are considered when checking rejection rules.
   * Expiration is **not handled by Flink** in real time. Instead, an **overnight batch job** flips expired\_flag → true for matured trades before trading hours begin.  
     ✅ This keeps the Flink ingestion logic simpler and focused on the rejection/ingestion rules.
2. **Rejection Rules**
   * **Rule 1**: If an incoming trade has trade\_id with **lower version than the latest active version**, it’s **rejected**.
   * **Rule 2**: If an incoming trade has trade\_id + version matching an active record but its **maturity\_date < today**, it’s **rejected**.
   * All rejected trades → DLQ topic persistent://front-office/trades/ingest.dlq.
3. **Ingestion Rules**
   * If trade\_id + version exists and is active → replace it (upsert).
   * If new trade\_id or new version → insert as new active record.
   * If maturity\_date < today → still insert, but as **active (expired\_flag=false)**.  
     (Overnight batch will later flip expired\_flag=true.)
4. **Postgres Guardrails**
   * Flink does near-time stateful filtering to reduce load.
   * But Postgres enforces rules too via **SQL UPSERT constraints** (so late/delayed events can’t break consistency).

**Key Points in this Implementation**

* **Keyed State** per trade\_id: remembers the latest version of an active trade.
* **Rejection Rules** implemented before writing to Postgres.
* **Postgres UPSERT** ensures DB-level guardrails (no lower versions overwriting).
* **DLQ** gets full JSON with rejection reason.
* **Exactly Once**: achieved at-least once + idempotent UPSERT. (True 2PC XA is explicitly *out of scope* per your spec).

This design fits a **front-office/middle-office ingestion pipeline**:

* Near-real-time
* Accurate (DB + Flink guardrails)
* Idempotent
* No 2PC complexity
* Overnight expiry handled separately.

**DESIGN:**

* PyFlink application (single-file) that reads from Pulsar (Table API source), applies near-time keyed-state rejection logic, writes accepted records to Postgres with idempotent UPSERT, and writes rejected records (full JSON + reason) to a Pulsar DLQ topic.
* Postgres DDL (two variants): HASH partitioning by trade\_id (recommended) and optional RANGE partitioning by maturity\_date; plus indexes and a PL/pgSQL trigger-based guardrail that enforces the rejection rules at DB-level.
* Docker Compose (Pulsar standalone, Flink JobManager/TaskManager, Vault). Postgres is external per your request.
* get\_credentials\_from\_vault() stub (uses hvac) with safe env-var fallback — no hard-coded secrets.
* Flink configuration notes tuned for ~20k msgs/min (estimates) and recommendations.
* GitHub Actions workflow that runs a Python security scan (bandit), lints (flake8), builds the Docker image, and pushes to Docker registry (optional step left as a secret-configured job).

I focused on correctness, idempotence, and practical deployability. Copy/paste and adapt. If you want, I can split files into a Git repo structure next.

***create\_trades.sql***

Notes:

* The trigger throws an exception for inserts/updates that violate rules. The Flink job will see DB exceptions on write and route those messages to DLQ (we will implement handling).
* The Partial indexes are only for active rows; expired rows are intentionally excluded to minimize index size and not impact writes as much.
* Hash partitioning is chosen for even distribution of trade\_ids; choose modulus equal to expected Postgres cluster capabilities.
* Optionally you can add a unique constraint/index on (trade\_id) for additional access patterns — but not included here to preserve historical versions.

**Optional DDL: Range partition by maturity\_date (alternative)**

If you prefer range partitioning by maturity\_date (e.g., to speed queries by time windows or retention), use this. It retains same indexes and triggers.

create\_trades\_alternative.sql –

**PyFlink application:**

Below is a single-file PyFlink job trade\_ingest\_job.py. It:

* Uses Flink Table API to define a Pulsar source (requires Flink Pulsar connector jar in classpath).
* Converts to DataStream of Python dicts.
* Keyed by trade\_id and uses ValueState to remember latest active version (to pre-filter many rejections), but still writes to Postgres where DB triggers enforce guardrails.
* Writes rejected records to Pulsar DLQ topic via Table API sink (or Pulsar producer).
* Writes accepted records to Postgres JDBC sink via a RichSinkFunction that executes an UPSERT SQL INSERT ... ON CONFLICT (trade\_id, version) DO UPDATE SET ... (this is idempotent on duplicate same pair).

Save as ***trade\_ingest\_job.py.***

**Important implementation notes & caveats**:

1. **Pulsar Connector**: The job assumes a Flink-Pulsar connector (Java jar) is available on the Flink classpath. You must add the correct flink-connector-pulsar jar compatible with your Flink version into the job cluster (e.g., via the Docker image or lib/ folder). The exact connector property names may vary — adapt the Table DDL accordingly. I used generic 'connector'='pulsar' keys as placeholders.
2. **Python DataStream <-> Java connectors**: Using Pulsar connectors from PyFlink usually requires the connector JARs and using Table API SQL DDL as shown. For production you will include the connector and test the exact options.
3. **Idempotence & Exactly-once**: This implementation relies on idempotent upserts (INSERT...ON CONFLICT) plus Flink retries to reach "effectively exactly-once" behavior. This is not distributed 2PC; it aligns with your explicit requirement.
4. **DB Guardrails**: The trigger ensures late/delayed events cannot violate the rules in DB even if Flink pre-filter missed something.
5. **DLQ**: Rejected messages are formatted in JSON with rejection reason. The job writes them to the DLQ Pulsar topic.
6. **Error handling**: If the PostgresUpsertSink throws an exception due to trigger, you should catch that (or configure Flink to direct failed messages to DLQ). The code above lets exceptions bubble — in production, catch them and publish the original message + error to DLQ.
7. **State size**: Keyed state holds only latest version per trade\_id (a single int). For many unique trade\_ids you will need sufficient heap and RocksDB backend. Use RocksDB state backend for large state.

**Flink tuning & cluster sizing notes (for ~20k messages/min)**

Assumptions: average message size small (1–2 KB), 20k/min ≈ 333 msgs/sec.

Recommendations:

* Use Flink with RocksDB state backend if state is large or many distinct trade\_ids. Configure state.backend.rocksdb.memory.managed and off-heap options.
* Parallelism: start with parallelism=4 for processed job (we used 4), scale up as throughput demands. For 333 msgs/s, 4 task slots is fine; for spikes, increase to 8–16.
* Task slots: ensure TaskManager has at least as many slots as parallelism. Use 2–4 TaskManagers with 2–4 slots each.
* Checkpointing: enable checkpointing every 10s (tune). This influences recovery and exactly-once semantics for connectors. Example Flink config:
* RocksDB: set taskmanager.memory.process.size and JVM options according to nodes.
* Pulsar client tuning: set batching and producer sendTimeOut appropriately. Use Pulsar partitioned topic pre-created with 32 partitions (you already have that).
* Postgres pool: set connection pool in the sink or reuse connections carefully. The implementation uses simple connections per sink instance.

state.checkpoints.dir: hdfs:// or s3

execution.checkpointing.interval: 10000

execution.checkpointing.timeout: 600000

execution.checkpointing.min-pause: 5000

**Docker Compose (Pulsar standalone, Flink, Vault):**

Minimal docker-compose.yml that runs Pulsar standalone, a basic Flink JobManager/TaskManager image (from official Flink), and Vault dev server for local testing. Postgres is expected external.

Notes:

* Put the Pulsar connector jars into ./flink/plugins/ (or Flink lib/) so they appear on classpath. Use the connector version matching Flink.
* This compose is for local/dev. For production use Kubernetes / managed Flink clusters and Vault in HA mode.