**Trade Ingestion from Diverse Sources for Front Office / Middle Office**

**Sources (what’s coming in)**

* **Kafka topics**
  + kafka://trades.external.\* (front-office producers, low-latency messages)
* **Pulsar topics**
  + pulsar://trades.incoming.highprio
  + pulsar://trades.incoming.normal
  + pulsar://files.manifests
  + pulsar://ingest.audit
  + pulsar://ingest.dlq
* **S3 objects (file types)**
  + **Small files** — JSON (newline or JSON array), CSV, fixed-width, Parquet (each file ~10k rows)
  + **Large bulk files** — CSV / Parquet / JSON (single files with millions of rows)
* **Other sources**
  + REST API producers (front-office apps) → publish directly to Pulsar or Kafka
  + NiFi provenance / NiFi → Pulsar/Kafka (for file ingestion orchestration)

**Ingestion style for each source (stream vs batch)**

* **Kafka topics** — **streaming** (real-time low-latency). Consume directly from Kafka by Flink jobs (or route into Pulsar if you centralize on Pulsar).
* **Pulsar topics** — **streaming** (primary streaming backbone). Realtime messages and small-file records are published here.
* **S3 — small files (JSON/CSV/fixed-width/Parquet, ~10k rows)** — **streaming**: NiFi reads small files, converts/validates records, and **publishes each trade record to Pulsar** (treat each row as a stream message).
* **S3 — large bulk files (millions of rows)** — **batch / controlled bulk**: NiFi stages file to object store and emits a manifest to files.manifests. A **separate bulk job** (Flink batch or Airflow-triggered job) processes it with resource isolation.
* **REST API producers** — **streaming**: publish to Pulsar/Kafka; Flink consumes immediately.
* **NiFi-originated events (manifests, control)** — **control/event** messages, streaming into Pulsar.

**Recommended architecture (concise, step-by-step)**

1. **Primary streaming backbone: Pulsar (and Kafka where needed)**
   * Use Pulsar as the primary message bus for streaming ingestion (separate topics for priorities and manifests). Kafka may remain for legacy producers; Flink reads from both if necessary.
2. **Small-file ingestion: treat as streams**
   * NiFi: ListS3 → FetchS3 → ConvertRecord (normalize schema) → PublishPulsar (publish each row).
   * Publish to trades.incoming.normal (or highprio for front-office). Small S3 files become indistinguishable from native stream messages.
3. **Real-time processing: Flink stream jobs**
   * Flink job(s) read from Pulsar (and Kafka) with checkpointing + RocksDB.
   * Keyed processing: validation, dedupe, enrichment, business rules (e.g., “newer maturity wins”), emit audit events, write to DB.
   * Use same business-logic library for stream & batch paths to guarantee identical validation.
4. **Sink to Yugabyte (fast path)**
   * Real-time Flink jobs perform **idempotent conditional upserts**:  
     INSERT ... ON CONFLICT (trade\_id) DO UPDATE ... WHERE EXCLUDED.maturity\_date > trades.maturity\_date
   * Reserve a dedicated connection pool and set write parallelism for the real-time job. Keep transactions short.
   * Consider range partitioning / sharding on Maturity\_Date for performance.
5. **Large-file / bulk ingestion: separate, controlled path**
   * NiFi stages file to S3 and publishes manifest to files.manifests.
   * Bulk consumer (separate Flink batch job or Airflow-run job) picks up manifest. Options:
     + **Chunked streaming**: split large file into chunks (e.g., 50k rows) and stream to a **bulk topic** consumed by a bulk-only Flink job with capped parallelism and throttling.
     + **Staging + controlled merge**: bulk load into a staging table, then run a controlled merge/upsert into main table during limited concurrency.
     + **Parallel bulk reader**: use Flink FileSource (bulk) with parallel readers that write in controlled batches to staging or directly to DB under throttling.
   * **Do not** let bulk job share Flink job slots / DB connection capacity reserved for real-time jobs.
6. **Priority / QoS & resource isolation**
   * Use separate Pulsar topics for priority separation (highprio vs normal vs bulk).
   * Deploy **separate Flink job(s)** or **separate Flink clusters / task slot pools** for real-time and bulk consumers.
   * Reserve DB resources for real-time path (connection pool, CPU, IO). Bulk path uses capped connections and adaptive throttling.
7. **Rate limiting & circuit breaker**
   * Bulk processors apply **rate limits and batch sizes**.
   * Monitor Yugabyte latency; if latency crosses threshold, automatically **pause or slow** bulk consumers (circuit breaker). Real-time consumers remain prioritized.
8. **Deduplication & idempotency**
   * Final guard: DB uniqueness constraints + idempotent upserts.
   * In-flight guard: Flink keyed-state dedupe with TTL for short windows if needed.
9. **Schema & validation**
   * Use a Schema Registry (Pulsar or external) for Avro/Protobuf/JSON schema enforcement. NiFi normalizes and validates before publishing records.
10. **Observability & DLQ**
    * Emit audit events to ingest.audit and invalid records to ingest.dlq.
    * Instrument end-to-end metrics: ingestion rate, processing latency, DB write latencies, failures, backlog by topic. Correlation IDs for traces.
11. **Orchestration**
    * Use Airflow to manage bulk job scheduling, Flink savepoints, backfills, and to run controlled merges or staging promotions. NiFi handles continuous ingestion flows.

**Short rationale (why this mapping)**

* Treating **small S3 files as streams** gives consistent, low-latency behavior for front/mid office workflows and avoids separate code paths.
* **Isolating bulk loads** (separate topics, jobs, throttles, DB reservation) prevents large-file ingestion from impacting real-time trade writes — preserves SLAs.
* **Flink + Pulsar** combination provides deterministic stateful processing, checkpointed offsets, and easy integration of both stream and bulk (via FileSource or chunked streaming) while keeping exactly-once *effects* through idempotent upserts to Yugabyte.

**Diagram notes (short)**

* **Small files**: NiFi converts each row into a Pulsar message so small S3 files are handled as streaming records (low-latency).
* **Large bulk files**: NiFi stages file and emits a manifest; bulk processing runs in separate Flink jobs (or Airflow-triggered batch) and writes to a staging area or streams chunked sub-files to a bulk topic.
* **Isolation**: Real-time Flink jobs and bulk Flink jobs are isolated (separate job/slot pools or clusters). DB connection pools are reserved for real-time path; bulk path is throttled.
* **Exactly-once / accuracy**: Flink checkpointing + RocksDB state + idempotent conditional UPSERTs to Yugabyte (ON CONFLICT ... WHERE EXCLUDED.maturity\_date > trades.maturity\_date) provide practical exactly-once effects.

If you want, I can:

* generate a PNG or SVG export of this diagram, or
* convert this Mermaid diagram into a PowerPoint slide or draw.io file, or
* produce a version annotated with deployment suggestions (k8s resources, Flink task slots, Pulsar namespace quotas).

Tell me which export/variation you want and I’ll create it here.

**Annotated deployment suggestions (Kubernetes, Flink, Pulsar quotas)**

Note: these are opinionated starting points. Tune based on your benchmarks and ticketed SLAs.

**Kubernetes clusters & node pools**

* **Separate node pools (or clusters) by workload:**
  + realtime-pool — for Flink real-time TaskManagers, NiFi small-flow workers, and low-latency services. Use high-CPU, moderate-memory instances (e.g., 8–16 vCPU, 32–64 GB RAM) and fast local SSD.
  + bulk-pool — for bulk Flink/batch jobs and heavy NiFi processors that stage large files. Use instances tuned for throughput and I/O (e.g., 16–32 vCPU, 64–128 GB RAM) and attach high-throughput network-attached storage for S3 multipart staging if needed.
  + infra-pool — for Pulsar brokers/bookies, Airflow schedulers/workers, Yugabyte DB nodes, and Prometheus/Grafana. Size according to vendor guidance; separate storage-optimized nodes for bookies and Yugabyte data.
* **Kubernetes best-practices:** use node taints/tolerations and pod affinity to keep realtime and bulk pods isolated. Use resource requests & limits for all pods.

**Flink deployment (K8s)**

* **JobManager:** 2 replicas, each with 4 vCPU, 16 GB RAM (use leader election + highly-available storage). Set jobmanager.memory.process.size per your Flink version recommendations.
* **TaskManagers (real-time pool):**
  + **TaskManager size:** 4–8 vCPU, 32–64 GB RAM.
  + **Task slots per TM:** typically set taskslots = vCPU or taskslots = 4. A good starting point: 4 slots per TM. That gives flexible slot utilization and reduces GC pressure.
  + **Parallelism:** set default-parallelism such that parallelism = num\_TMs \* slots\_per\_TM \* utilization\_factor. Example: 10 TMs × 4 slots = 40 slots; set job parallelism to 40 (tune after benchmarking).
* **TaskManagers (bulk pool):** separate Flink deployment or separate TaskManager group with larger VMs and fewer, larger slots (e.g., 16–32 vCPU with 8–16 slots) to allow high-throughput bulk reads.
* **State backend & storage:** RocksDB state backend with local SSD and durable checkpoint storage in S3. Allocate at least 100–200 GB local disk per TM if large state (adjust to state size).
* **Checkpoints:** configure checkpoint interval (e.g., 30s), maxConcurrentCheckpoints = 1, and retention on cancellation.
* **Slot isolation:** use Kubernetes labels/taints so real-time TMs never get preempted by bulk TMs.

**Pulsar deployment & namespace quotas**

* **Broker / bookie sizing:** follow Pulsar guidance. Example starting point for medium load: 3–5 brokers + 3–5 bookies (bookies sized with high disk throughput and NVMe). Use more bookies for higher disk capacity and throughput.
* **Namespace separation & quotas:** create namespaces per workload and set quotas:
  + realtime namespace: publishRateInMsg=100000 (example), publishRateInByte tuned to your message size, dispatchRateInMsg unlimited for consumers but monitored. Set retention and message TTL appropriately.
  + bulk namespace: lower publishRateInMsg per producer but larger storage quota; configure publishThrottling on consumers to limit how fast bulk consumers can pull.
  + audit/dlq namespace: small quotas but longer retention.
* **Bundle splitting:** use more bundles for high-topic-count scaling (helps distribute load across brokers).
* **Transaction Coordinator (if used):** run TC with redundancy and monitor transaction logs. If you rely on DB idempotent upserts, TC is optional.

**Yugabyte (DB) sizing & connection management**

* **Node sizing:** start with at least 3–5 Yugabyte nodes for HA; choose instances with good network and disk IOPS (e.g., 8–16 vCPU, 64–128 GB RAM, NVMe SSD).
* **Connection pooling:** real-time Flink pool: reserve a dedicated DB pool (e.g., 50–200 connections globally depending on parallelism). Bulk jobs: capped pool (e.g., 10–50 connections) to prevent saturation.
* **Sharding / primary keys:** choose trade\_id distribution to avoid hot partitions. Monitor RPC/latency and increase nodes as needed.
* **Staging tables:** if using staging merges, size staging tables on separate tablespaces or nodes if possible.

**NiFi**

* **NiFi nodes:** run a small cluster for high availability; give NiFi processors CPU to handle parsing/conversion. Use separate NiFi clusters or separate process groups for small-file streaming vs bulk staging.
* **Connection thresholds:** set back pressure and queue limits so NiFi does not overwhelm Pulsar or S3.

**Airflow**

* **Scheduler & workers:** ensure enough workers to run bulk orchestration tasks; schedule heavy jobs during controlled windows or trigger them via manifests.
* **Resource-aware executor:** use KubernetesExecutor / Celery with resource labels so bulk tasks run on bulk-pool only.

**Observability & throttling**

* **Prometheus rules:** add alerts for Yugabyte latency, Pulsar broker CPU/disk/IO, Flink checkpoint failures, backlog size in Pulsar topics. Create automated playbooks for throttling bulk consumers.
* **Circuit-breaker:** implement an automated controller (K8s operator or Airflow task) that reduces bulk consumers or pauses bulk jobs when DB latency or broker backlog exceeds thresholds.

I added this annotated deployment guidance into the canvas document.

If you want, I can now:

* export a draw.io .drawio file representing the diagram so you can open and edit it in diagrams.net, or
* produce a PNG/SVG with the annotated overlays for direct use in documentation.