Tick Data to Signals

Real-Time Crypto Analytics on KDB-X

Draft Version 1.0

Philippe DAMAY

January 2026

Abstract

This paper presents a production-grade implementation of a real-time cryptocurrency analytics system built on KDB-X. The system ingests live market data from Binance WebSocket streams via C++ feed handlers, computes VWAP and order book imbalance metrics in KDB-X, and delivers them to downstream consumers with minimal latency.

Contents

1 Introduction

This paper presents a production-grade implementation of a real-time cryptocurrency analytics system designed to compute VWAP and order book imbalance from Binance market data with minimal latency.

The system runs on modest hardware: a WSL environment with 12 CPU threads and 7.6GB of RAM. This constraint is intentional. The goal is not to throw resources at the problem, but to demonstrate that **thoughtful architecture and code quality can achieve production-grade performance without enterprise-scale infrastructure**.

1.1 Philosophy

Modern infrastructure offers abundant compute power. Cloud instances scale on demand, memory is cheap, and the instinct is often to throw more resources at a problem rather than solve it elegantly. This works—until it doesn't. Latency-sensitive systems expose the inefficiencies that brute force cannot hide.

There is something worth revisiting in the mindset of early developers who worked under severe resource constraints. Limited memory and slow processors fostered significant ingenuity. Optimization was not a luxury—it was a necessity. Writing tight, efficient code required deep understanding of computer architecture, algorithms and data structures.

Arthur Whitney, the creator of kdb+, embodies this philosophy. He writes dense code in k with a specific belief: if code fits entirely in the CPU's L1 cache, it runs orders of magnitude faster. This discipline permeates kdb+. A well-written kdb+ system can accomplish with 8GB of RAM what many enterprise Java or Python stacks struggle to achieve with way more.

One of kdb+'s greatest strengths—and what makes development both exciting and challenging—is that it rewards adaptation over convention. There is no one-size-fits-all architecture. The optimal design depends on the specific problem: the data shape, the latency requirements, the query patterns. This demands more from the developer, but the payoff is a system precisely fitted to its purpose.

1.2 Technology choices

C++ was chosen for the feed handlers. It provides direct control over memory layout, deterministic performance, and efficient integration with both WebSocket libraries and the KDB-X IPC interface. For latency-sensitive ingestion where every microsecond of parsing and serialization matters, C++ remains the right tool.

After everything written above, the choice for analytics was hardly a choice at all—kdb+ had to be it. Its vector-oriented execution model and columnar data structures make it exceptionally efficient for the aggregations and windowed calculations that VWAP and order book imbalance require.

1.3 Approach

This implementation is not presented as definitive. Improvements remain to be found. That is precisely why telemetry is embedded throughout the system. Every event carries nanosecond timestamps at ingestion, microsecond measurements for parsing and IPC, and sequence numbers for gap detection. The architecture is designed not just to perform, but to reveal where it can perform better.

Observability is not an afterthought bolted on at the end. It is built into the foundation, because **chasing the optimal solution is an ongoing process—and you cannot optimize what you cannot measure**.

This paper applies these principles to a concrete problem: real-time crypto analytics.

2 The System in Action

Before diving into architecture diagrams and code, it is worth seeing what the system delivers. The following sections walk through four dashboards accessible from a navigation menu: Feed Handler Monitoring, Dataflow Monitoring, Trades and Quotes, and Analytics. Together they provide both the observability needed to trust the pipeline and the real-time signals it was built to produce. KX Dashboards was selected for its native kdb+ integration via WebSockets and its reactive data binding.

2.1 Feed Handler Monitoring

In real-time trading systems, the feed handler is the first point of contact with market data—and the first point of failure. A degraded feed handler does not announce itself with an alarm; it manifests as stale prices, missed trades, or analytics computed on incomplete data. By the time downstream consumers notice something is wrong, the damage is done. The Feed Handler Monitoring dashboard exists to make the invisible visible: to surface the health and performance of data ingestion *before* it impacts trading decisions.

This dashboard is divided into two complementary views: a **health status grid** providing an at-a-glance operational summary, and **latency trend charts** enabling deeper performance analysis over a rolling 10-minute window.

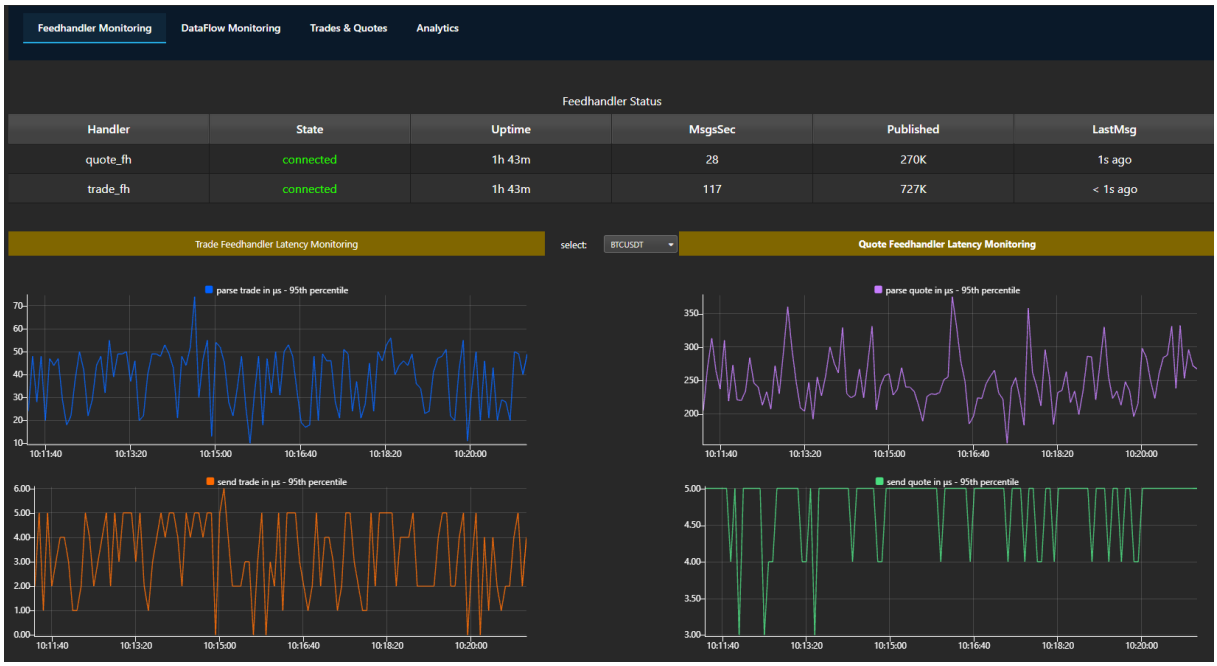

Figure 1: Feed Handler Monitoring dashboard showing health status grid (top) and 95th percentile latency charts (bottom) for trade and quote handlers.

2.1.1 Health Status Grid

The upper portion of the dashboard displays a data grid populated by the `.tel.vsFhStatus[]` function, which returns the current operational state of each feed handler. Each row represents one handler—trade or quote—and each column answers a specific operational question.

Column	Question It Answers
Handler	Which feed handler is this row describing?
State	Is this handler currently connected to Binance?
Uptime	How long has this handler been running without restart?
MsgsSec	What is the average throughput since startup?
Published	How many messages have reached the tickerplant?
LastMsg	How recently did we receive data from the exchange?

Table 1: Health status grid columns and their operational meaning.

MsgsSec (messages per second, averaged over uptime) establishes a baseline for expected throughput. Cryptocurrency markets exhibit predictable activity patterns—higher volume during US and European trading hours, quieter periods overnight. A sudden drop in MsgsSec during normally active hours warrants investigation: it could indicate exchange-side throttling, network degradation, or symbol subscription issues.

LastMsg is the heartbeat indicator. In a healthy system, this value should always show "< 1s ago" or at most a few seconds. A LastMsg value drifting to 30 seconds or more—while State still shows `connected`—reveals a subtle failure mode: the WebSocket connection remains open, but

no data is arriving. This can occur when the exchange silently drops a subscription or when an upstream proxy buffers data unexpectedly.

Together, these six fields allow an operator to assess feed handler health in under five seconds. Green across the board means data is flowing; any anomaly provides a starting point for investigation.

2.1.2 Latency Trend Charts

The lower portion of the dashboard displays four time-series charts, each tracking the **95th percentile latency** for a specific operation over the past 10 minutes:

1. **Parse Latency (Trade Handler)** — Time to deserialize incoming trade JSON
2. **Send Latency (Trade Handler)** — Time to transmit parsed trade to the tickerplant
3. **Parse Latency (Quote Handler)** — Time to process order book updates
4. **Send Latency (Quote Handler)** — Time to transmit L5 book snapshot to the tickerplant

Why 95th Percentile? Average latency is a comforting but misleading metric. A feed handler reporting 50µs average parse time might still exhibit 500µs spikes on 5% of messages—spikes that coincide with high-volatility moments when timely data matters most. The 95th percentile captures the "worst reasonable case": the latency experienced by the slowest 1-in-20 messages. If P95 remains stable, the system is performing consistently. If P95 spikes while average remains flat, something is causing intermittent delays—garbage collection, CPU contention, or network congestion.

The system also captures P50 (median) and max latency for deeper analysis, but P95 strikes the right balance for operational monitoring: sensitive enough to surface problems, stable enough to avoid false alarms from one-off outliers.

Parse Latency: The Cost of Understanding Parse latency measures the time elapsed from receiving raw bytes on the WebSocket to producing a structured, validated message ready for transmission. For trades, this involves JSON deserialization and field extraction—a relatively lightweight operation. For quotes, parsing is substantially more complex: each incoming delta must be validated against sequence numbers, applied to the in-memory order book, and the top 5 levels extracted for downstream consumption.

Elevated parse latency in the trade handler typically indicates CPU pressure or inefficient JSON parsing. In the quote handler, elevated parse latency may additionally signal order book reconstruction overhead—particularly after a sequence gap forces a full snapshot reload.

A well-tuned trade handler should exhibit P95 parse latency under 100µs. Quote handlers, due to order book maintenance, typically run 2–3x higher but should remain under 300µs for a flat-array implementation processing 100+ symbols.

Send Latency: The Cost of Delivery Send latency measures the time from completing message parsing to successful transmission over IPC to the tickerplant. This metric isolates network and serialization overhead from parsing costs.

In a co-located deployment where feed handlers and tickerplant run on the same host, send latency should be minimal—typically under 50µs for trades and under 100µs for the larger quote payloads (22 price/quantity fields per message). Elevated send latency suggests:

- **Network congestion** between handler and tickerplant
- **Tickerplant backpressure** if the TP cannot process messages as fast as they arrive
- **Serialization overhead** for complex message structures

A gradual upward trend in send latency over hours may indicate tickerplant memory pressure or log-writing contention. A sudden spike often correlates with market volatility—more messages per second means more contention for the same resources.

2.1.3 Operational Value

This dashboard transforms feed handler monitoring from a reactive practice ("why is our data wrong?") into a proactive discipline ("what is changing in our data pipeline?"). An operator reviewing these panels at the start of a trading session can verify that both handlers are connected, throughput matches expected levels, and latency remains within acceptable bounds.

More importantly, the 10-minute trend view enables pattern recognition. A slow upward drift in P95 parse latency might go unnoticed in a point-in-time check but becomes obvious as a visual trend. This early warning allows operators to investigate—and potentially remediate—performance degradation before it impacts downstream analytics or trading signals.

In production trading systems, the cost of stale or missing data is measured in lost alpha. The Feed Handler Monitoring dashboard is the first line of defense.

2.2 Dataflow Monitoring

The Feed Handler Monitoring dashboard answers whether data is arriving. The Dataflow Monitoring dashboard answers what happens to it next. Once a message leaves the feed handler, it traverses the tickerplant and lands in the real-time database. Each hop introduces latency and potential failure modes. This dashboard traces that journey, quantifying the cost of each stage and surfacing bottlenecks before they cascade into data gaps.

The dashboard is divided into two sections: **status grids** showing current volume and resource consumption, and **segment latency charts** decomposing end-to-end latency into its constituent hops.

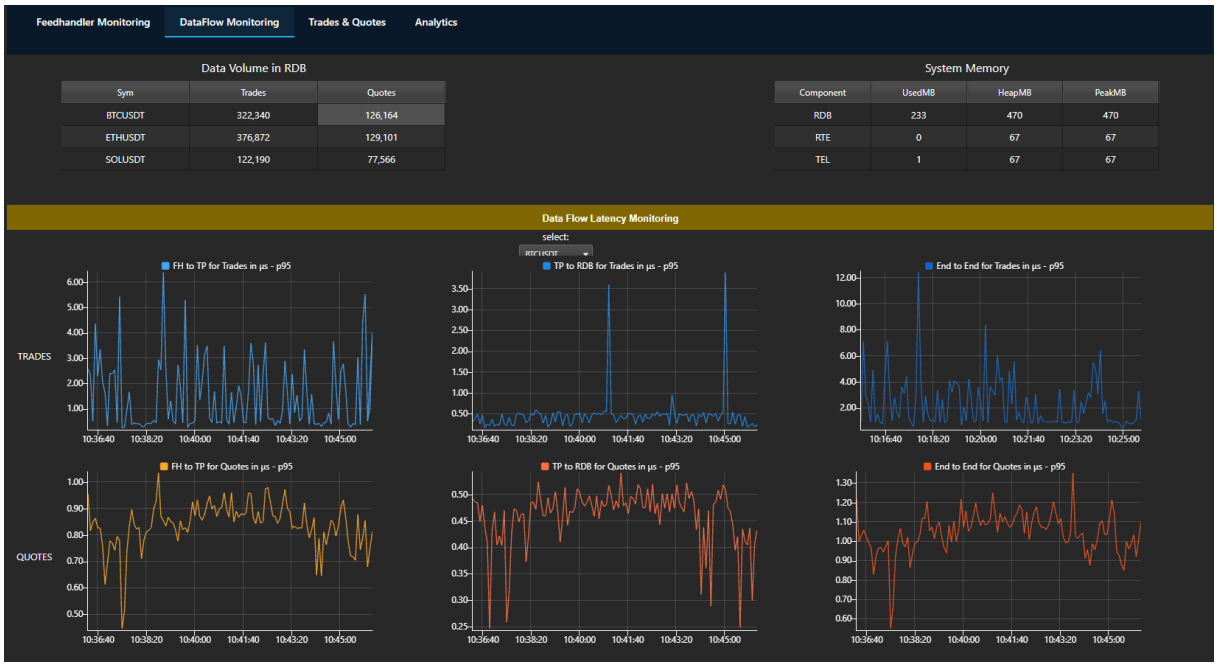

Figure 2: Dataflow Monitoring dashboard showing data volume and system resources (top) with segment latency breakdown for trades and quotes (bottom).

2.2.1 Status Grids

The upper portion displays two data grids side by side: Data Volume and System Resources.

Data Volume The Data Volume grid, populated by `.tel.vsDataVolume[]`, shows the cumulative count of trades and quotes stored in the RDB for each symbol. This answers a simple but critical question: is the system capturing what it should be capturing?

Column	What It Shows
Sym	Trading symbol
Trades	Total trade records in RDB for this symbol
Quotes	Total quote records in RDB for this symbol

Table 2: Data Volume grid columns.

A symbol showing zero trades while others accumulate thousands indicates a subscription failure or symbol-specific filtering issue. A sudden plateau in counts—visible when refreshing the grid—suggests upstream data has stopped flowing for that symbol. The ratio between trades and quotes also provides a sanity check: quote updates typically outnumber trades by an order of magnitude, so a symbol with more trades than quotes warrants investigation.

System Resources The System Resources grid, populated by `.tel.vsSystemResources[]`, reports memory consumption for each kdb+ component in the pipeline.

Column	What It Shows
Component	Process name (RDB, RTE, TEL)
UsedMB	Currently allocated memory
HeapMB	Total heap size
PeakMB	Maximum memory used since startup

Table 3: System Resources grid columns.

UsedMB trending upward without corresponding data growth may indicate a memory leak or unbounded table accumulation. **PeakMB** reveals whether the system has experienced memory pressure spikes—useful for capacity planning and diagnosing intermittent slowdowns that have since recovered. A large gap between UsedMB and HeapMB suggests memory fragmentation or over-provisioned heap allocation.

2.2.2 Segment Latency Charts

The lower portion displays six time-series charts arranged in two rows: trades on top, quotes below. Each row decomposes the end-to-end latency into three segments, tracking the 95th percentile over a rolling 10-minute window.

Segment	Metric	What It Measures
FH → TP	`fhToTpMs_p95`	Feed handler to tickerplant transmission
TP → RDB	`tpToRdbMs_p95`	Tickerplant to real-time database delivery
E2E	`e2eMs_p95`	Total latency from FH receive to RDB storage

Table 4: Latency segments tracked for both trade and quote flows.

FH → TP: Ingestion to Distribution This segment measures the time from when the feed handler completes parsing to when the tickerplant receives the message. In a well-configured system with co-located processes, this should be sub-millisecond. Elevated FH→TP latency points to network issues between processes, serialization overhead, or tickerplant backpressure during high-volume periods.

TP → RDB: Distribution to Storage This segment measures the tickerplant's publish latency to the RDB subscriber. The tickerplant's core function is to receive, log, and fan out—it should add minimal latency. Spikes here indicate either log-writing contention (the TP writes to disk before publishing) or RDB-side slowness propagating back as subscriber backpressure.

E2E: The Full Picture End-to-end latency is the sum of all segments: parse time, FH→TP, and TP→RDB. This is the number that matters for downstream consumers. An analytics engine querying the RDB sees data that is E2E milliseconds old. For VWAP calculations on fast-moving markets, every millisecond of staleness reduces signal quality.

The value of segment decomposition becomes clear when E2E spikes. Rather than investigating the entire pipeline, operators can immediately identify which segment degraded. If FH→TP is stable but TP→RDB spiked, the issue lies in tickerplant logging or RDB processing—not the feed handler.

2.2.3 Operational Value

This dashboard answers the question every data engineer dreads: "where is the latency coming from?" By decomposing the pipeline into measurable segments, it transforms vague complaints about "slow data" into actionable diagnostics.

The separation of trade and quote flows is deliberate. Quote messages are larger (22 price/quantity fields versus 6 for trades) and arrive more frequently. It is common for quote latency to run higher than trade latency—but if the gap widens unexpectedly, it may indicate that the tickerplant or RDB is struggling with the larger payload size or that order book updates are arriving in bursts that exceed steady-state capacity.

Combined with the resource grids, this dashboard enables capacity planning. If UsedMB approaches available RAM while TP→RDB latency climbs, the system is signaling that it needs more resources—or smarter retention policies—before it fails.

2.3 Trading View

2.4 Analytics View

3 Architecture Overview

The system follows a classic kdb+ tick architecture: feed handlers capture and normalize external data, a tickerplant logs and distributes it, and downstream processes store, analyze, and monitor.

3.1 Data Flow

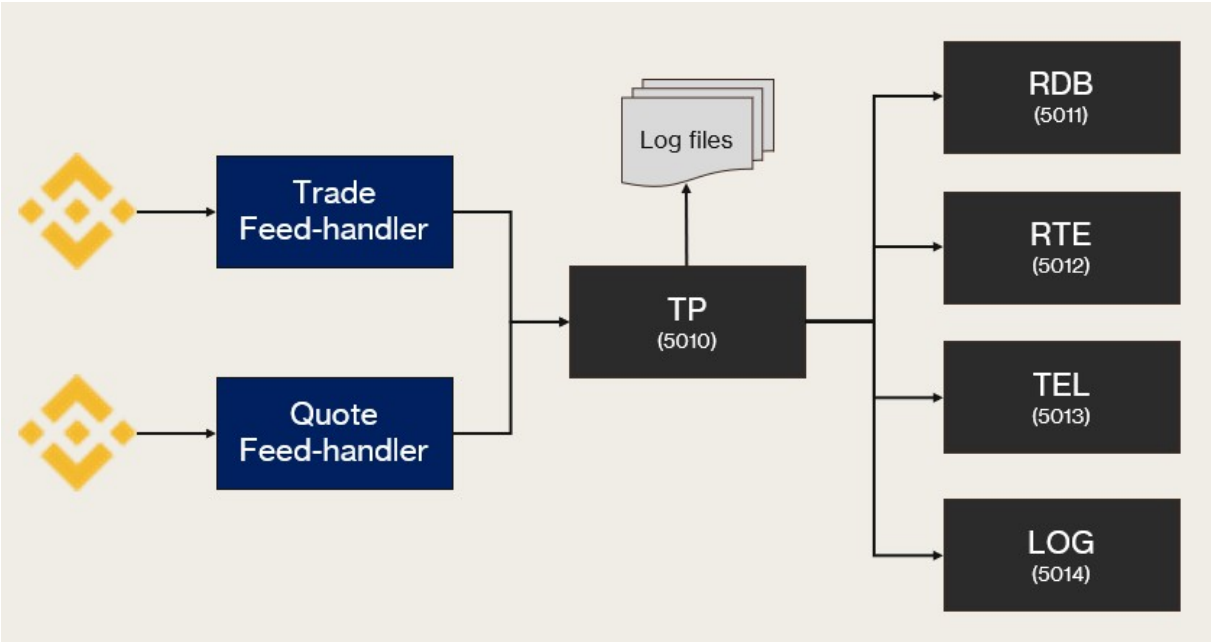

Figure 3: System architecture: data flows from Binance through feed handlers to kdb+ components

Market data enters via two WebSocket connections to Binance—one for trades, one for order book depth updates. Each stream is handled by a dedicated C++ feed handler that normalizes the data, adds timing instrumentation, and publishes to the tickerplant via asynchronous IPC.

The tickerplant serves as the central hub: it logs every message to disk (separate files for trades and quotes) and fans out to subscribers. Downstream, the RDB stores raw events, the RTE computes analytics, and TEL aggregates telemetry. A log manager handles retention and integrity verification.

3.2 Component Responsibilities

Component	Port	Responsibility
Trade FH	–	Captures trades from Binance, adds timestamps, publishes to TP
Quote FH	–	Maintains L5 order book via snapshot + delta reconciliation
TP	5010	Logs messages to disk, distributes to subscribers
RDB	5011	Stores raw trades and quotes, supports replay from logs
RTE	5012	Computes VWAP and order book imbalance in real-time
TEL	5013	Aggregates latency metrics, monitors feed handler health
LOG	5014	Manages log retention, verifies integrity

3.3 Design Principles

Several principles from the Data Intellect whitepaper guided this architecture:

One job only. Each process has a single responsibility. The tickerplant logs and distributes—it does not compute analytics. The RTE computes analytics—it does not store raw data. This separation simplifies debugging and allows independent scaling.

Fail fast. Processes validate their environment on startup and exit immediately if dependencies are missing. A feed handler that cannot connect to the tickerplant does not silently buffer—it logs the failure and retries with backoff.

Event-driven. Analytics are computed on update, not on timers. When a trade arrives, VWAP buckets update immediately. When a quote arrives, order book imbalance recalculates. This minimizes latency between data arrival and derived metric availability.

Recoverable. Both RDB and RTE replay from tickerplant logs on startup. If a process crashes mid-day, it restarts with full state reconstruction—no manual intervention required.

Observable. Every message carries latency instrumentation. TEL aggregates these into percentiles every 5 seconds. Feed handlers publish health metrics. The system is designed to reveal its own performance characteristics.

3.4 Why Two Feed Handlers?

Trades and quotes have fundamentally different semantics:

- **Trades** are discrete events. Each message is final and append-only. Processing is stateless—parse, timestamp, publish.

- **Quotes** represent evolving state. The order book must be reconstructed from a REST snapshot and maintained through delta updates. Processing requires a state machine with sequence validation.

Combining these into a single process would mix simple and complex code paths, couple their failure modes, and complicate debugging. Separate handlers keep each focused and independently recoverable.

4 C++ Feed Handler Implementation and Features

The system comprises two dedicated feed handlers—one for trades, one for L5 order book quotes—that ingest real-time market data from Binance WebSocket streams and publish to a kdb+ tickerplant. The architecture prioritizes correctness, latency observability, and reliable state management to support downstream VWAP and Order Book Imbalance calculations for 20 symbols.

4.1 Design Decision: Separate Handlers for Trades and Quotes

Trades and quotes have fundamentally different data semantics. Trades are discrete events: each message is final and append-only. Quotes represent continuously evolving state requiring snapshot reconciliation and delta application. Separating these concerns into distinct processes yields three benefits: independent failure isolation (a crash in quote reconciliation does not halt trade ingestion), simpler code paths within each handler, and clearer debugging when issues arise.

4.2 Order Book State Machine

The quote handler implements a state machine for each symbol to correctly synchronize with Binance's depth stream protocol:

```
enum class BookState {
    INIT,        // Initial state, buffering deltas
    SYNCING,     // Snapshot applied, replaying buffered deltas
    VALID,       // Normal operation, applying live deltas
    INVALID      // Sequence gap detected, needs rebuild
};
```

Upon connection, each symbol enters INIT state and buffers incoming deltas while requesting a REST snapshot. The critical synchronization logic validates that the first delta after snapshot satisfies Binance's reconciliation rule:

```
if (state == BookState::SYNCING) {
    // First delta after snapshot
    // Must satisfy: U <= snapshotUpdateId+1 <= u
    if (firstUpdateId > snapshotUpdateIds_[idx] + 1) {
        invalidate(idx, "Snapshot too old");
        return false;
```

```
    }
    if (finalUpdateId < snapshotUpdateIds_[idx] + 1) {
        // Delta is stale, skip it
        return true;
    }
    // Transition to VALID
    states_[idx] = BookState::VALID;
}
```

This approach ensures the published L5 book is never stale or corrupted, which is essential for meaningful Order Book Imbalance calculations.

4.3 Cache-Efficient Flat Array Storage

The OrderBookManager stores all book data in contiguous arrays rather than per-symbol maps or trees:

```
// Price and qty arrays: [numSymbols * BOOK_DEPTH]
// Access: bidPrices_[symIdx * BOOK_DEPTH + level]
std::vector<double> bidPrices_;     // Bids sorted high->low
   (index 0 = best bid)
std::vector<double> bidQtys_;
std::vector<double> askPrices_;     // Asks sorted low->high
   (index 0 = best ask)
std::vector<double> askQtys_;
```

For 100 symbols at L5 depth, the entire dataset occupies approximately 16KB—fitting within CPU L1 cache. L5 extraction becomes a simple offset calculation:

```
L5Quote OrderBookManager::getL5(int idx, long long
   fhRecvTimeUtcNs, long long fhSeqNo) const {
    L5Quote q;
    q.sym = idxToSym_[idx];

    const int offset = idx * BOOK_DEPTH;

    q.bidPrice1 = bidPrices_[offset + 0]; q.bidQty1 =
       bidQtys_[offset + 0];
    q.bidPrice2 = bidPrices_[offset + 1]; q.bidQty2 =
       bidQtys_[offset + 1];
    // ... remaining levels

    return q;
}
```

This design eliminates pointer chasing and cache thrashing as the symbol count scales.

4.4 Latency Instrumentation

Each published message includes three timing fields captured at distinct points in the processing pipeline:

```
// Capture wall-clock receive time (for cross-process correlation)
auto recvWall = std::chrono::system_clock::now();
long long fhRecvTimeUtcNs =
    std::chrono::duration_cast<std::chrono::nanoseconds>(
        recvWall.time_since_epoch()).count();

// Start monotonic timer for parse latency
auto parseStart = std::chrono::steady_clock::now();

// ... JSON parsing and book update ...

// End parse timer
auto parseEnd = std::chrono::steady_clock::now();
long long fhParseUs =
   std::chrono::duration_cast<std::chrono::microseconds>(
    parseEnd - parseStart).count();
```

This granularity allows precise identification of latency sources. When combined with tickerplant-side timestamps, end-to-end latency from exchange to kdb+ table is fully observable.

4.5 Sequence Validation and Gap Detection

The trade handler validates that exchange-assigned trade IDs arrive in strictly increasing order per symbol:

```
void TradeFeedHandler::validateTradeId(const std::string& sym,
   long long tradeId) {
    auto it = lastTradeId_.find(sym);

    if (it != lastTradeId_.end()) {
        long long last = it->second;

        if (tradeId < last) {
            spdlog::warn("OUT OF ORDER: {} last={} got={}", sym,
               last, tradeId);
        } else if (tradeId == last) {
            spdlog::warn("DUPLICATE: {} tradeId={}", sym, tradeId);
        } else if (tradeId > last + 1) {
            long long missed = tradeId - last - 1;
            spdlog::warn("Gap: {} missed={} (last={} got={})",
               sym, missed, last, tradeId);
        }
    }
```

```
    lastTradeId_[sym] = tradeId;
}
```

These checks provide immediate visibility into exchange-side anomalies or network issues that could compromise data integrity.

4.6 Reconnection with Exponential Backoff

Both handlers implement automatic reconnection with exponential backoff capped at 8 seconds. The backoff sleep is interruptible to allow rapid shutdown response:

```
bool TradeFeedHandler::sleepWithBackoff(int attempt) {
    int delay = INITIAL_BACKOFF_MS;
    for (int i = 0; i < attempt && delay < MAX_BACKOFF_MS; ++i) {
        delay *= BACKOFF_MULTIPLIER;
    }
    delay = std::min(delay, MAX_BACKOFF_MS);

    // Sleep in small increments to allow quick shutdown response
    const int checkIntervalMs = 100;
    int slept = 0;
    while (slept < delay && running_) {
        std::this_thread::sleep_for(std::chrono::milliseconds(checkIntervalMs));
        slept += checkIntervalMs;
    }

    return running_;
}
```

This prevents rate-limit violations during exchange instability while maintaining availability.

4.7 Asynchronous IPC Publishing

Publication to kdb+ uses negative handle IPC, which is fire-and-forget:

```
K result = k(-tpHandle_, (S)".u.upd", ks((S)"trade_binance"), row,
    (K)0);
```

The negative handle ensures the handler writes to the socket buffer and continues without waiting for acknowledgment. This decouples feed handler throughput from tickerplant processing speed, preventing backpressure from stalling market data capture.

4.8 Summary

The feed handler layer is optimized for its specific role: accurate timestamping, correct state management, and reliable transport to kdb+. It deliberately excludes analytics computation, which belongs in the kdb+ layer where q's vector operations and in-memory data access provide superior performance for VWAP and OBI calculations. This separation ensures the feed handlers

remain simple, testable, and focused on the single task of delivering clean, well-instrumented market data to the tickerplant.

5 kdb Implementation and Features

6 Analytics

7 Results

8 Conclusion