DATA INTELLECT

Building Real Time Event Driven KDB-X Systems

AUTHORS: JONNY PRESS, GARY DAVIES, SAMANTHA DEVLIN, SIMON FRAZER

NOVEMBER 2025

ADVISE.
DESIGN.
DELIVER.

Data Intellect provides independent subject matter experts who can leverage their experience across many installations to review system architectures, including the non-kdb+ components.

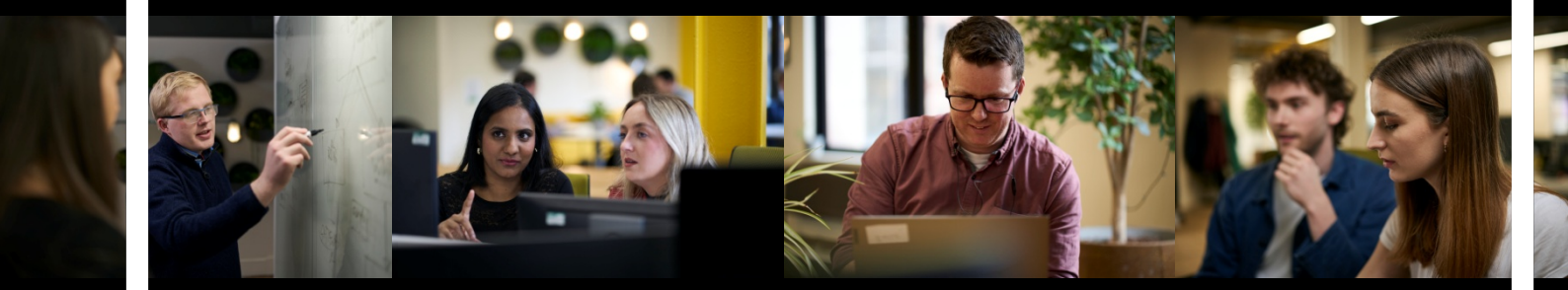
In the ARK, Data Intellect will utilize our kdb+ maturity matrix which provides a scoring system and characteristics across the key areas within a kdb+ application.

- ✓ kdb+ framework
- ✓ Hardware
- ✓ Environments
- ✓ Disaster Recovery/Continuity of Business
- ✓ Software Development Life Cycle
- ✓ Performance & Scalability
- ✓ Security & Access Controls
- ✓ Supportability
- ✓ Test-ability
- ✓ Data Quality
- ✓ Data Availability
- ✓ Code

Data Intellect will provide a detailed report of scoring, findings and any actions that must, should or could be actioned upon to improve the system.

The ARK can provide insight into:

- Changes needed to meet current / increased demand
- Design opportunities, leveraging the latest developments in the kdb+ space
- Improvements for system throughput
- System simplifications
- Performance improvements to meet current or improved Service Level Agreements
- Cloud Architecture
- kdb+ upgrades
- Improvements to development pipelines
- System stability



[CONTENTS

1 ABOUT DATA INTELLECT

Data Intellect is a global data and technology consultancy firm. We provide services to advise, design and deliver with a collaborative approach aligned to long-term partnership.

Our key area of expertise is financial and capital markets technology solutions. We bring together deep expertise in time series data, high performance software engineering, data analytics and solution delivery. By leveraging cutting-edge technologies and methodologies we provide solutions that are not just innovative but are also scalable, reliable and tailored to meet the unique needs of our clients.

This paper was written primarily with technology leaders and architects in mind, highlighting best practice approaches that Data Intellect have honed over many years of working with event driven systems in the world's top financial institutions. Real time, event driven systems are the cornerstone of the Trading Data Ecosystem.

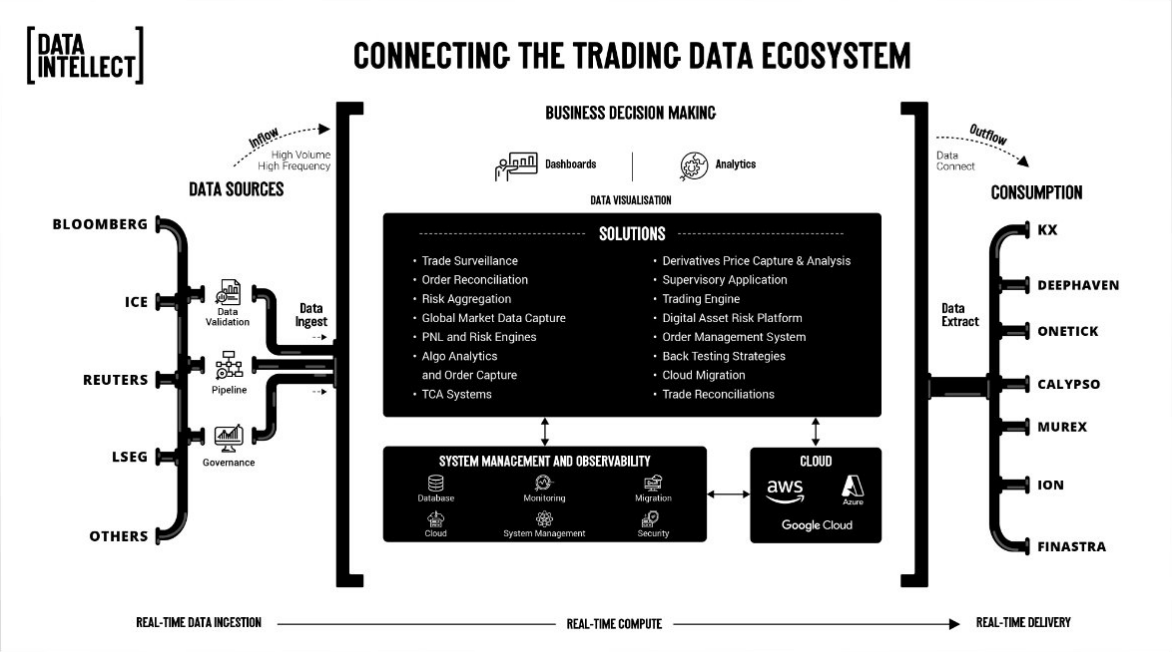


Figure 1 Data Intellect's Trading Data Ecosystem

2 INTRODUCTION

2.1 Document Purpose and Structure

The target audience for this whitepaper is anyone designing, building or reviewing an event driven, latency sensitive system utilising KDB-X or kdb+ as the primary technology. Everything outlined in this whitepaper will work with kdb+; we will keep this document updated as technological advancements of KDB-X become available.

KDB-X has two sweet spots:

1. Timeseries analytics on long time ranges of data from deep history right up to "now"
2. Building and maintaining real time analytics and views of derived data in a time sensitive manner

This whitepaper will focus on design choices to optimise for the second. It will advise on common design principles, building for performance, resilience and observability. Some of the items that will be discussed are not KDB-X specific and could be considered general systems engineering good practice but are added as reminders or for completeness. As with any engineering challenge, the optimal approach will depend on the specific context and the target objective.

One of the rewarding features of KDB-X is that it is very easy to build small, stand-alone prototype examples to experiment with an approach or investigate an issue. This whitepaper will utilise this technique where possible. Operation speed can be measured with \t or \ts. Relative timings will be presented, as the intention is to show differences in approach rather than absolute performance, and absolute timings will depend on system configuration.

In a Real Time Analytics system data flows from an upstream source, through a feed handler, to a distribution layer (a tickerplant or message transport) and on to a real time engine component for analytic calculation. The structure of this document reflects this flow.

2.2 KDB-X Is Not Just a Database

KDB-X is hugely effective at storing and analysing large volumes of timeseries data, but it is also a fully featured programming language called q. The product genesis is first as a programming language which was later extended to become a database. This brings significant flexibility, and means that systems are built using a "database" technology which bears no resemblance to anything else in the database space.

The flexibility can lead to a blurring of boundaries between what KDB-X should and shouldn't be used for. For a proficient q developer it can be attractive to build out more and more functionality within KDB-X, rather than employing a specialised technology. We will aim to address some of those potential hand-off points within this document.

2.3 Latency Targets

It would be easily possible to build a specialised KDB-X system to ingest data, analyse it, and produce an analytic to drive further action in a sub-millisecond timeframe. However, where KDB-X excels is using vectorised operations to calculate analytics on large sets of data particularly as the complexity of the analytic workload increases. The q language allows analytics to be created efficiently with respect to developer time, execution time, and throughput.

For the purposes of this whitepaper, the assumed target end-to-end latency range is between 1 and 200 milliseconds whilst achieving high throughput.

2.4 Reference Architecture

The starting reference architecture is a kdb+ tick style system. KDB-X production systems are multi-process architectures, but commonly consist of at least a

- Feed Handler to receive data from an upstream source
- Tickerplant to log and distribute data
- Real Time Database as an in-memory store
- Real Time Engine to calculate and compute additional analytics.

It is assumed that the reader has worked with this type of system previously. In a real production system, there will be many discrete components which connect to each other. From a concurrency point-of-view each KDB-X process is usually single threaded, with parallelism achieved through multiple processes.

The end-to-end latency is the time taken from when a piece of data enters the system to the corresponding derived analytic being made available to consumers (which could be systems or people). The throughput of the system is the amount of data the system can process in a given unit of time. These are the measures we are trying to optimise for in a reliable, predictable way.

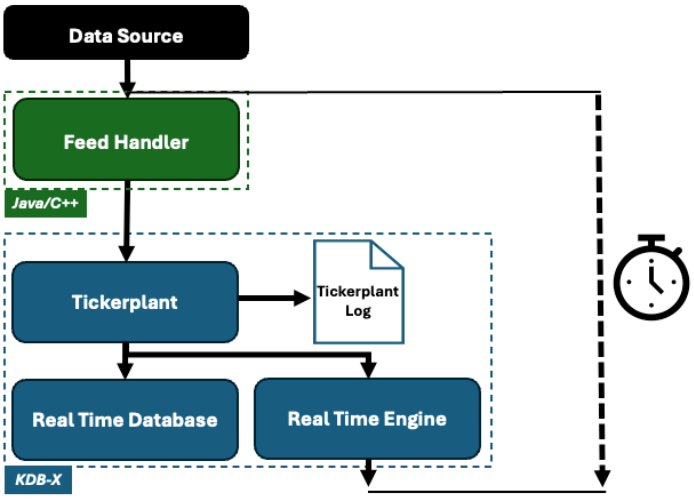


Figure 2 End-to-end latency in a data analytics system

2.5 Disaster Recovery

Disaster recovery is addressed within each section as appropriate rather than being covered as a separate topic. For a highly available latency sensitive analytics platform the usual approach is to replicate the environment with the ability to failover between them in the event of significant failures.

2.6 Hardware

Hardware configurations and operating system tuning are considered out-of-scope for this document. For up-to-date benchmarks on optimal latency configurations refer to STAC.

2.7 Code

It is assumed that the reader has some familiarity with q code. Code examples in this paper were created with the Public Preview version of KDB-X. The code will also run using kdb+. Where possible example code is shared to produce test data sets, the reader can modify these as required to align more closely to their own datasets.

As mentioned previously, computational timings will be presented as relative values. If required, readers should conduct their own tests on their own hardware to measure absolute performance.

3 DESIGN PRINCIPLES FOR A KDB-X SYSTEM

Building a successful data platform requires understanding who and what the downstream users and applications are, what they care about and how they intend to use the data. Catering to all needs is challenging, but it is achievable within a well architected system.

3.1 Principles

At a high level, here are the main areas to consider when deploying KDB-X in a latency sensitive environment:

1. **Build for optimised latency and practicality:** Consider the largest latency that the system can tolerate. Allowing for even a small amount of batching (e.g. publish messages every 10ms) will potentially increase throughput and decrease resource utilisation of the system significantly.
2. **Event-driven:** Build processes in an event-driven manner by performing calculations "on update" rather than on timers.
3. **Ensure separation of system and user processes:** Separate the processes which are accessed by users from those which are used for system calculations to ensure control and flexibility over resource management.
4. **One job only:** Each process should have one and only one job.
5. **Fail fast:** Processes should fail and exit as quickly as possible if they cannot initialise correctly due to environmental or data issues.
6. **Aim for DAG:** Aim to ensure data flow within the system is a Directed Acyclic Graph (DAG)- no backtracking, re-routing, two-way publishing or feedback loops.
7. **Minimize process chaining:** Chains of dependent processes can complicate recovery and resiliency.
8. **Optimise datatypes:** This is considered in depth in section 7.1.
9. **Be aware of Inter Process Communication cost:** This is considered in depth in section 0.
10. **Minimise replication:** If the same analytic is used by multiple processes, calculate it independently and publish.
11. **Resilient and recoverable:** A KDB-X system contains multiple processes with dependencies. The system should be designed to take multiple bullets, i.e. a single process failing does not cripple the system, and processes should be individually recoverable.
12. **Technology choice:** Use the most appropriate technology for the task at hand. "Most appropriate" should include considerations around development team skillset, ongoing support and maintenance and complexity of resultant environment. Examples might be using specialised message buses or writing some components in other languages such as Java, C++ or Python.
13. **Ensure observability:** Treat observability, monitoring and supportability as a first class design consideration as they are critical to system success.

Some of these principles contradict one another. For example, minimising replication and ensuring tasks are discrete may lead to more process chaining. These trade-offs are resolved and decided by engineering choices on a case-by-case basis.

3.2 Implementing a Hot Path

Technology teams must constantly balance the practice of building for future scale and building to optimise the current business use case. An approach to cater for both highly latency sensitive use cases as well as more general real time analytics or future research use cases is to implement a hot path. This is a dedicated data flow within the system, which may enter via a separate tickerplant, along which all design choices are focussed on latency sensitivity (minimal datasets, streamlined datatypes, conflation, no batching). This may result in some data duplication.

4 FEED HANDLERS

4.1 Introduction

The feed handler is the gateway between a firm's data system and the outside world– it connects to an upstream API and captures streaming market data in its most raw format. The upstream API can be external or internal sources, including message transports. At a minimum, the job of the feed handler is to structure the data into a table format using appropriate KDB-X data types for ingestion and downstream consumption.

4.2 Language Choice

The feed handler must be built in a language which has an interface to both the upstream API and KDB-X. Common high-performance choices are C++ and Java. Python and C# are possible but less common options. Feed handlers can also be written in q, usually utilising an embedded shared object to handle the upstream connectivity. The optimal language choice is a combination of performance requirements, consistency with the rest of the application code base and internal standards, and availability of appropriately skilled development resources.

4.3 Latency Choices

4.3.1 Pruning, Enrichment and Bespoke Logic

It is unlikely that all the data available on the feed requires persistence in KDB-X. In a highly latency sensitive system the feed should prune as much data as possible.

The feed handler may apply a degree of enrichment or bespoke business logic to the data. An example of enrichment would be applying symbology mappings to ensure consistency across data sources, as data sources can represent the same information in different ways. More bespoke business logic might be generating an additional derived data set. An example of this might be a feed handler that receives a full order flow from an exchange, but additionally calculates and maintains orderbook levels and publishes a "top of book" view in addition to the order flow.

The enrichment and/or business logic steps can also be implemented downstream from the initial point of capture. It is a design trade-off. Keeping the transformation logic minimal in the feed handler is usually advisable in a highly latency sensitive environment. When a little more latency tolerance is possible then more transformation in the feed handler can greatly reduce repeated work by downstream components and make the data easier to use throughout the system.

4.3.2 Timestamping

Despite the fact that it will incur a slight increase to the processing time, it is good practice for the feed handler to apply a feed time timestamp to the data to mark the time that it received the message. In some cases multiple timestamps may be appropriate, to indicate when the raw data was received and when the processed data was published.

4.3.3 Batching

Employing a short batching period such that the feed handler publishes data on a timer or when a certain number of messages have been processed will increase the throughput of the feed handler and the system as a whole.

4.3.4 Conflation

Message rates on data feeds can be "spiky", particularly in financial markets, with peak message rates being multiple times the average message rates. In latency sensitive environments, feed conflation may be an option. Exchange top-of-book quotes are a candidate for conflation- if there are multiple quotes for the same instrument in the same

market to be processed then only the latest one is current and the others are stale. From a latency perspective it would be advisable to discard the stale quotes and only propagate the latest. This will however mean that the dataset is not complete, which may have a detrimental impact on other analytics use cases which may be served by the KDB-X environment.

4.3.5 Striping and Compute Resources

A feed handler running in a "steady state" under expected average message rate conditions should have sufficient spare capacity to allow it to cope with message rate spikes. It is usual to stripe feed handlers, i.e. have multiple feed handlers each of which deals with a different segment of the data universe. Striping can be by any readily available dimension, e.g. message types or instrument / topic subscription lists. When considering a striping pattern, ensure that downstream analytics use cases are considered and how ordering of data is maintained. For example, if receiving data from an exchange it makes more sense to stripe by instrument rather than splitting between trades and quotes to ensure that data ordering between trades and quotes when analysing an individual instrument is maintained.

4.3.6 Data Types and IPC

Refer to section 7 for data type and IPC considerations. The same considerations for KDB-X to KDB-X data transfers also apply to feed handler to KDB-X data transfers.

4.4 Resilience

Feed handlers can run on the same hardware hosting the tick capture components or on separate hosts. Resilience considerations must encompass strategies to mitigate against process, host and network failures. A resilient KDB-X environment will likely consist of replicated data captures running in a hot-hot primary and secondary configuration to mitigate against major failures. Recovery approaches will also depend on the upstream source- some upstream sources, including third party messaging middleware, offer the capability to replay data gaps, whereas others are "fire and forget", meaning there is no way to recover previously published data.

The most common approach is to replicate feed handlers, and rely on the fact that two independent failures are unlikely to occur. More resilient approaches introduce additional complexity. It's a design choice.

4.4.1 Replicated Feed Handlers

A standard approach to building for resilience is to run multiple copies of the feed handler into replicated environments, ideally with different connection points into the upstream data source. In cases where the data source is fire and forget (no recovery mechanism), this is the only approach available.

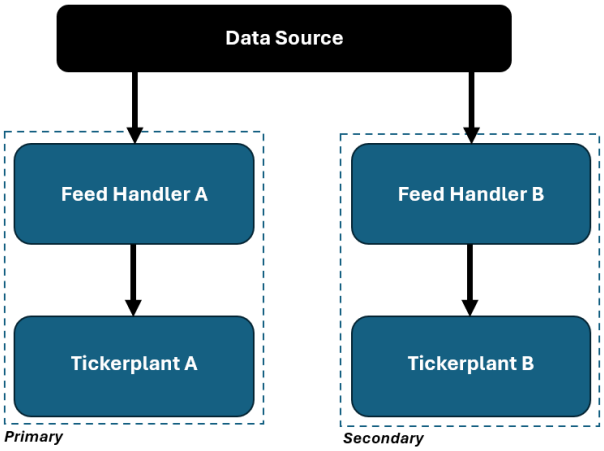


Figure 3 Example of multiple copies in a data system

A variant of the above is depicted below, which might be the case if only one upstream connection is possible. In this, one feed handler publishes to two tickerplant end points.

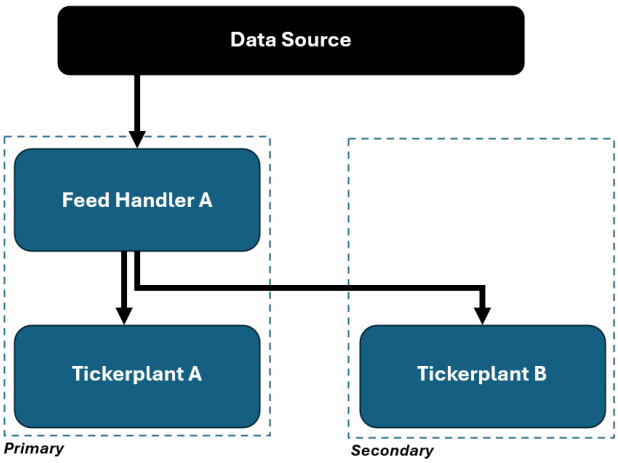


Figure 4 Alternative example of multiple copies in a data system

4.4.2 Asynchronous Publishing and Downstream Connectivity

The usual approach is for the feed handler to connect to the tickerplant and publish messages asynchronously, primarily for performance reasons. If the feed handler becomes disconnected from the tickerplant then it is not clear which data has successfully reached the tickerplant- it may not be the last message sent by the feed handler.

If the upstream data source is "fire and forget", then the most reliable approach is for the feed handler to always implement a local recovery log. If the feed handler becomes disconnected from the tickerplant then once the connection is re-established the feed handler must determine the last data received by the tickerplant (potentially by running a query against the RDB) and re-publish any data that has been missed.

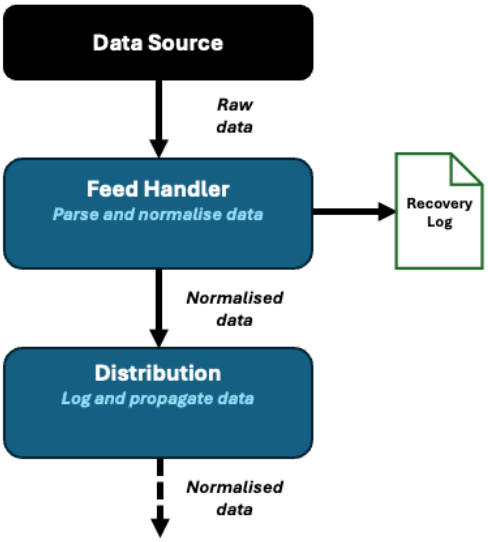


Figure 5 Feedhandler with Recovery Log

If the upstream data source is recoverable, then the recovery strategy may utilise this property. This means it must recover the sequence number of the last published message and recover the messages from that sequence number from the upstream and re-publish them to the tickerplant.

4.4.3 Impact on Downstream Consumers

If a feed handler recovers and publishes data it could publish old or late data into the system. Downstream components must be able to detect that the upstream data stream has failed, and upon any recovery action ensure that they don't act inappropriately on late data. This will be covered in more detail in section 8.

5 DATA LOGGING AND DISTRIBUTION

5.1 Introduction

Once data has been read from upstream it must be distributed. In a standard KDB-X data capture environment the usual component to facilitate this is the Tickerplant, though there are other options. A performance tuned Tickerplant approach is likely to lead to the lowest end-to-end latency, though a Tickerplant oriented architecture suffers from the following problems:

- The Tickerplant is a bottleneck as all data must flow through it
- The Tickerplant is a single point of failure. The standard mitigation strategy is to fully replicate the environment
- Slow consumers can cause backlogs and increased resource utilisation on the Tickerplant

5.2 Standard Tickerplant

The Tickerplant is responsible for:

1. Persisting data to a log file (similar to a Write Ahead Log in other database terminology)
2. Distributing data to downstream consumers

Consumers can subscribe for data and read the tickerplant log file for recovery purposes assuming it is written to a shared storage device. It is usual for a tickerplant to offer:

- Immediate or batch publishing of data. Immediate publishing propagates updates immediately, batch groups updates on a timer
- Subscriptions options to subscribe for specific tables, or specific datasets within each table

Given that the Tickerplant has the dual responsibility of logging and distribution data, it can be considered to break the "one job only" principle outlined in section 3.1.

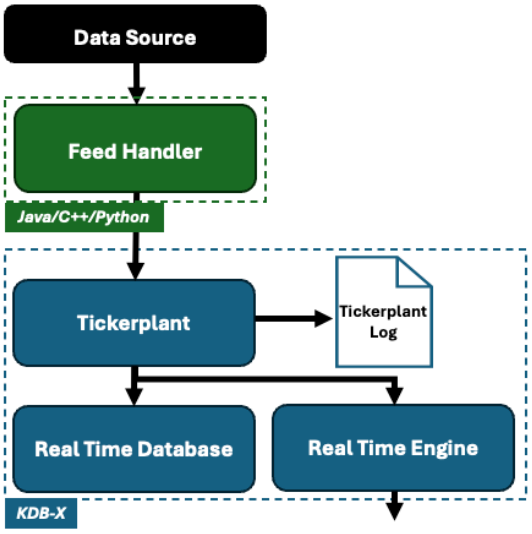


Figure 6 Role of the Tickerplant in a KDB-X environment

5.2.1 Performance Considerations

The Tickerplant can be a bottleneck to data flow through the system. To minimise the bottleneck, consider the following:

- Ensure the storage device for writing the Tickerplant Log File is as write performant as possible
- Minimise the number of data consumers and employ broadcast publishing (see section 7.3.2)
- Minimise the occurrence of custom filtered subscriptions. Filtered subscriptions reduce the amount of data flowing through the system, but increase the load on the Tickerplant as it must create the filtered view
- Batch the publish of data from the Tickerplant, as even a small timer batch of, for example, 10ms can greatly increase the system throughput
- Batch the publish of data to the Tickerplant from the Feed Handlers. Each update message is logged immediately, leading to a write to the storage device, therefore batching at the feed handler level will reduce the number of I/O operations and likely remove the need for batching at the Tickerplant
- Only log to disk when data is published. This introduces a resiliency consideration, but can lead to increased throughput.

If the target is to minimise latency then the initial approach should be:

- publish tick-by-tick from the Feed Handlers
- publish tick-by-tick from the Tickerplant
- minimise number of consumers
- only allow table based subscriptions (no further filtering)
- utilise broadcast publish

5.3 Non-Logging Tickerplant

It is possible to split the jobs of the Tickerplant and de-couple the data distribution component from the logging. This aligns to the "one job only" principle, minimising the time taken to distribute data to downstream components. There are more details on this approach here.

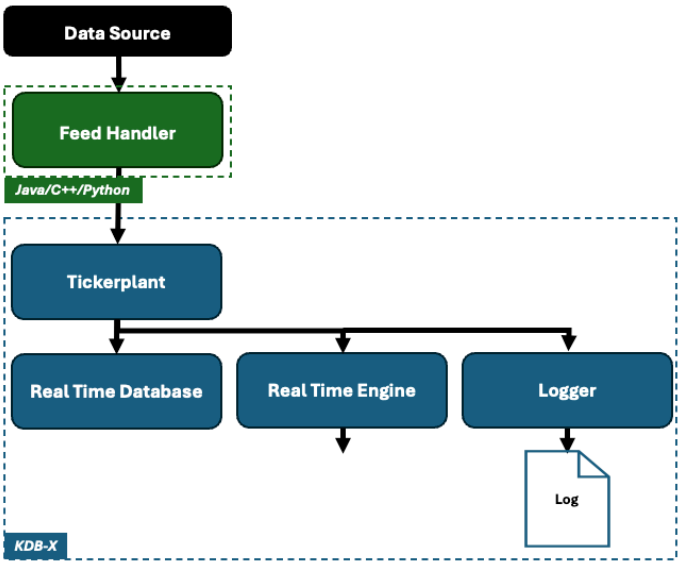


Figure 7 Non-logging Tickerplant variant

This approach introduces complexities with respect to resiliency which must be carefully considered. The decoupling of data logging and distribution introduces a race condition upon recovery, where a recovering downstream component must ensure that all prior data has been successfully logged by the Logger component before recovering.

The performance considerations related to publishing of data outlined in section 5.2 also apply to the non-logging variant.

5.4 Reliable Transport

Reliable Transport is a Tickerplant alternative introduced by KX through the kdb Insights SDK initiative. Data publishers (e.g. Feed Handlers) use the RT publishing API to write to a local message log, which is pushed to the RT cluster for consumption and distribution. The RT has the ability to run across multiple nodes. When compared to a standard Tickerplant, an RT based system is:

- more resilient system, when deployed in multi-node cluster configuration
- immune to the "slow consumer" problem as publishing, logging and subscribing are de-coupled
- more easily scalable to higher message rates
- usually higher latency, but dependent on message size. For larger messages sizes it may be lower than with a standard Tickerplant approach.

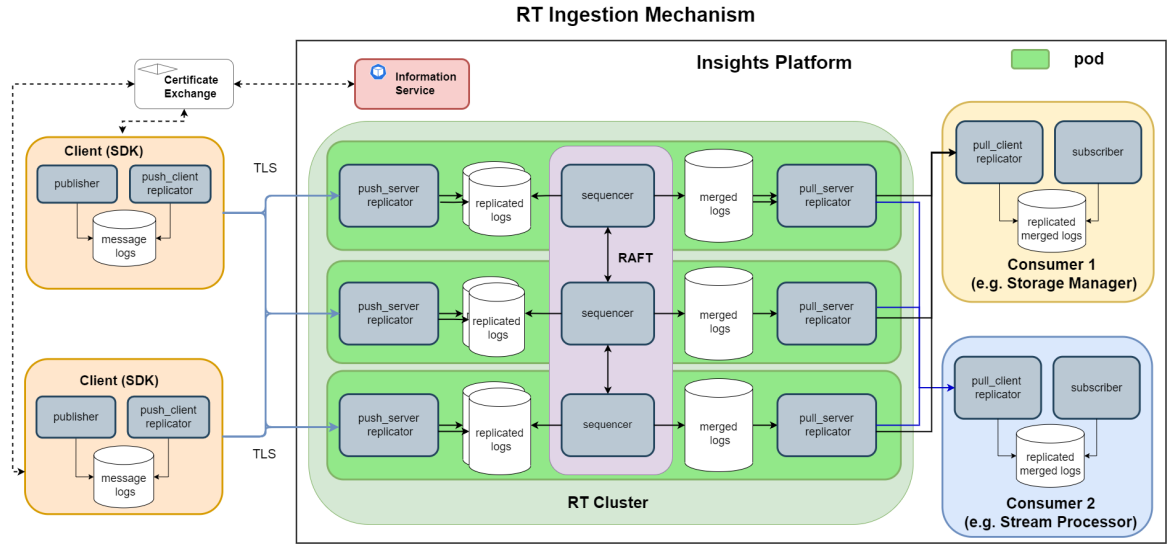


Figure 8 Reliable Transport architecture

5.5 Dedicated Messaging Technology

An alternative approach is to utilise a dedicated third party technology to propagate data around the system. This can be useful for both initial distribution and ongoing message transport throughout the system.

A common mistake in KDB-X systems, especially ones that start small and grow, is that a large amount of message fan out is done from KDB-X components. A KDB-X system starts by publishing derived data to a small number of subscriber end points using point-to-point TCP, but the number of subscribers grows over time to the point that KDB-X is doing a large volume of IPC.

Message transports come in a variety of forms, from broker-less architectures like Chronicle Queue and Aeron, to broker-based like Solace and Apache Kafka. Broker-less tend to be lower latency, broker-based tend to be more feature rich. All of them allow components written in different technologies to subscribe to the same datasets, publish data for wider consumption, and usually provide a data recovery mechanism.

Chronicle Queue is utilised within extremely latency sensitive trading environments. The main use case is to propagate messages between components within a single host as quickly as possible. It employs memory mapped files to achieve this.

Solace targets more general purpose low latency messaging, provided via either software or hardware solutions, with the lower end of the latency spectrum achieved in the hardware solution. Solace provides subscription flexibility via hierarchical topics with wildcards, and also the ability to propagate data between hosts, sites, or on-premise and cloud based software installations. KDB-X related deployment options for Solace are discussed here.

Introducing a dedicated message transport increases the architectural options to KDB-X system designers, and may bring observability benefits. It allows some aspects of fault tolerance and recovery to be delegated to the message transport rather than trying to custom build within KDB-X itself. An example is subscriber analytic components could use the recovery features of the message transport rather than relying on the tickerplant log file. Broker based solutions in particular provide greater observability of the messages flowing through the system via a central broker. An example system is outlined below.

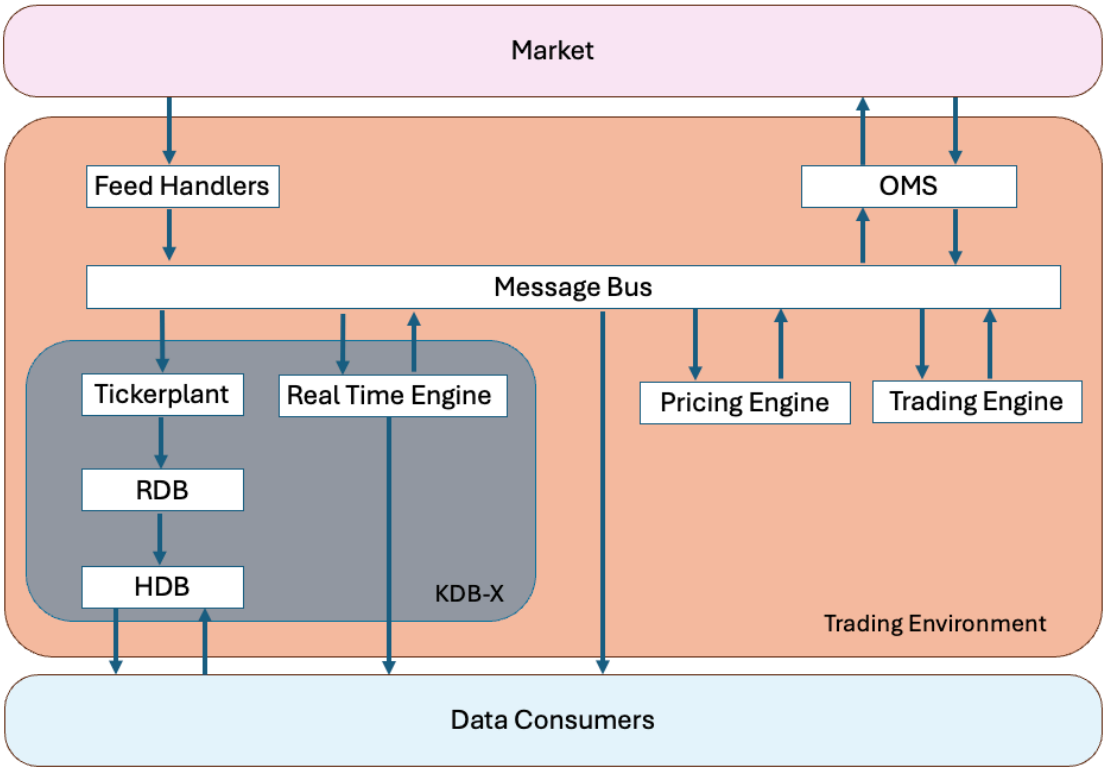


Figure 10 Example trading environment built around a central message bus

6 REAL TIME ENGINES

6.1 When Should We Build RTEs?

A Real Time Engine (RTE) receives data from an upstream source, performs a calculation or transformation of the data, and makes the results available to downstream consumers either via queries or by publishing a derived data set. Consider building an RTE when:

- Analytics are common - multiple users or systems require the same data and it can be calculated once and accessed via query or publication
- Analytics are required to be calculated in a latency sensitive, event driven way
- Analytics are expensive to compute but easy to keep track of intraday

Building an event driven RTE to calculate an analytic in an event driven manner is almost always more complicated than creating a query based approach where a query is run against a database component such as the RDB on a timer, but gives several advantages:

- A dedicated component, focused on a specific set of calculations
- Optimised data structures for the task at hand
- No contention, as might be seen if querying a shared component such as an RDB
- Bespoke handling of data across multiple days
- Potentially very lightweight with respect to both CPU and memory profile

A good first approach can be to develop a polling query equivalent to ensure the business case for the development cost is justified, then move to an event driven RTE approach.

6.2 Language Choice

RTEs can be built in q or any language which has a supported interface to KDB-X e.g. Java, C++, C#, C, or Python (potentially via PyKX). Generally in a KDB-X tick style data capture system the only components which must be written in q are the Tickerplant, RDB and HDB (historic database).

For the purposes of this document we will concentrate on writing them in q. Building in q allows us to lean on its expressiveness and performance, particularly in processing larger volumes of data.

6.3 Performance

6.3.1 Batching Or Tick-by-Tick

The most important thing to understand is how the data is going to arrive from upstream. If optimising for low latency, a tick-by-tick approach is common with every update propagated immediately. A batch based approach means multiple records (rows in a table) will arrive in a single update. A batch based approach is employed for greater throughput.

6.3.2 Performance Characteristics

The utilisation profile of an RTE will depend on the calculation being performed and the data structures employed to achieve it. The absolute values for CPU and memory load will likely have a dependency on:

- Calculation complexity
- Ingested data volume
- Uniqueness of data universe- it is common that an RTE stores state based on some keyed value, so as the number of unique keys increases the memory requirements will likely increase

An ideal RTE implementation will aim to achieve:

- Memory usage that is constant with respect to ingested data volume, with potentially a linear relationship to number of unique keys, assuming keyed structures are used
- CPU usage with sublinear relationship with respect to ingested data volume (due to batching) and a sublinear relationship with respect to number of unique keys

6.4 Example RTE – HLOC

Consider as a simple example a RTE that receives a stream of trade updates, and tracks high, low, open, close and total volume per instrument. The RTE must maintain state (the current values for each instrument) and analyse across every new trade record which arrives. This is a common pattern.

To track the required values, on each update we need to calculate:

- High as the greater of the currently stored value and the incoming value
- Low as the lower of the currently stored value and the incoming value
- Open as the first value received
- Close as the last value received
- Total volume as the sum of all the volumes received so far including the new volumes

If required to maintain state, aim to utilise a data structure which ensures constant time lookups for inserts. This usually means utilising a single keyed structure with a `u attribute on the key.

```
// table to store hloc data
hloc:([sym:`symbol$()] high:`float$(); low:`float$(); open:`float$(); close:`float$();vol:`int$());

// function to generate example trade data
// n is number of rows to generate
// syms is the unique list of symbols to use
makedata:{[n;syms] ([]time:asc .z.p+n?0D00:00:01; sym:n?syms; price:n?100f; size:n?1000)}

// function to process a single row of trade data
updsingle:{
 // retrieve current value (which could be empty)
 current:hloc[x`sym];

 // update existing values
 @[`hloc;x`sym;:;(x[`price]|current`high;
                  x[`price]&0Wf^current`low;
                  x[`price]^current[`open];
                  x[`price]; x[`size]+0^current`vol)];
 }

// time taken to process 100k trades from a universe of 10 unique instruments
// timings not shown, but relative timings depicted
q)d1:makedata[100000;-10?`4]
q)\t updsingle each d1
X

// and from a larger universe of 5000 instruments
// as universe gets larger, time increases
q)d2:makedata[100000;-5000?`4]
q)\t updsingle each d2
1.5X

// clear the table, add a u attribute
q)delete from `hloc
`hloc
q)update `u#sym from `hloc
`hloc

// with a u attribute, times are constant
// we have removed the impact of the larger universe of instruments
q)\t updsingle each d1
X
q)\t updsingle each d2
X

// processing time is linear w.r.t. the number of updates being processed
```

```
q)d3:makedata[200000;-5000?`4]
q)\t updsingle each d3
2X
```

If a batch processing approach can be used, the number of updates that can be processed per second can be greatly increased. Extending the example from above:

```
// table to store hloc data
hloc:([sym:`u#symbol$()] high:`float$(); low:`float$(); open:`float$(); close:`float$();vol:`int$());

// function to generate example trade data
// n is number of rows to generate
// syms is the unique list of symbols to use
makedata:{[n;syms] ([]time:asc .z.p+n?0D00:00:01; sym:n?syms; price:n?100f; size:n?1000)}

// function to process multiple rows of data, using a join approach
updjoin:{
 // aggregate across the inbound dataset
 newdata:select newhigh:max price,
               newlow:min price,
               newopen:first price,
               close:last price,
               newvol:sum size
  by sym from x;

 // join the current data onto the new data
 newdata:newdata lj `sym xkey select sym, high, low, open, vol
                            from hloc
                            where sym in exec sym from newdata;

 // store the data
 `hloc upsert `sym xkey select sym,
                             high:high|newhigh,
                             low:newlow&0Wf^low,
                             open:newopen^open,
                             close,
                             vol:newvol+0^vol from newdata;

 }


// time taken to process 100k trades from a universe of 10 unique instruments
// X is the time taken by the updsingle approach outlined above
q)d1:makedata[100000;-10?`4]
q)\t updjoin d1
0.002X

// and from a larger universe of 5000 instruments
q)d2:makedata[100000;-5000?`4]
q)\t updjoin d2
0.004X

// therefore the batch approach offers orders of magnitude performance improvement

// as the inbound update size grows by orders of magnitude
// the execution time increases sub-linearly
q)d3:makedata[100;-5000?`4]
q)\ts:100 updjoin d3
Y

q)d4:makedata[1000;-5000?`4]
q)\ts:100 updjoin d4
2Y

q)d5:makedata[10000;-5000?`4]
q)\ts:100 updjoin d5
8Y

q)d6:makedata[100000;-5000?`4]
q)\ts:100 updjoin d5
12Y

// how big can we go?
// try 100m records
q)d10:makedata[100000000;-5000?`4]
q)\ts updjoin d10
30Y
```

6.5 Recovery

RTEs should be able to recover from intraday failures, and their state after recovery should be the same as the state if it were not to have failed. This usually comes from an initialisation routine. This routine must ensure that the data is processed once and once only. There are two steps required to recover:

1. Read data that has already happened (including building any caches from history)
2. Subscribe to new data

These must be synchronised to ensure there are no duplicated or missing data. Reading first and then subscribing to data can cause some data to be missed. Subscribing first and then reading can cause data duplication. There are two recommended approaches for recovery to ensure there are no duplicated or missing data – one uses the Tickerplant log and one uses the RDB.

6.5.1 Recovery using the Tickerplant Log

A standard KDB-X system includes a Tickerplant, which subscribes to feed handlers for all the system's real-time data. Typically, the Tickerplant writes incoming messages to a logfile, adding timestamps of tick retrieval, and pushes the data to downstream subscribers.

This purpose of the Tickerplant logfile is solely for recovery. Usually, the logfile gets rolled at EOD, so it contains messages for a full day. In systems where intraday writes are performed, the logfile will contain smaller batches (e.g. it could be files written and rolled every hour, so 24 files in a day). Regardless, these files are used for data recovery for subscribed real-time processes on initialisation during normal startup or recovery.

Any real-time subscriber process (e.g. an RDB or RTE) could initialise with a call to the Tickerplant to get details of the current Tickerplant log file. These details will include the Tickerplant logfile path, the number of messages written to the current logfile, and number of messages that have been published to subscribers. Note there may be additional logic to handle the case of intraday writes or multiple logfiles. Discerning the difference between the counts of messages written to the logfile and messages published to a subscriber, can then be used to read from the Tickerplant log, as well as subscribing to the live flow coming from the Tickerplant.

This core KDB-X recovery logic is illustrated below.

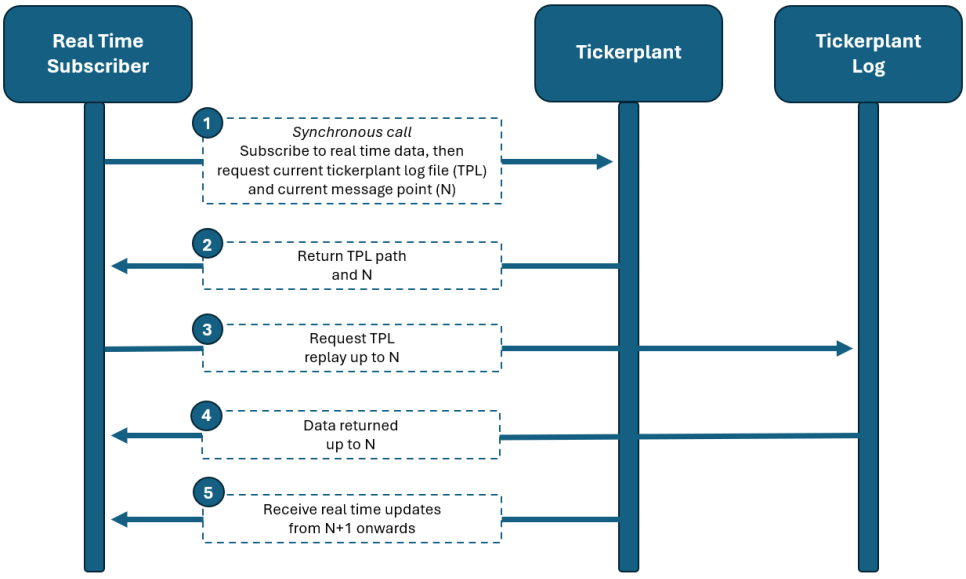


Figure 11 Breakdown of recovery using the tickerplant log

This recovery approach will ensure that the process does not get any duplicated data or miss any data between Tickerplant logfile replay and live subscription, due to the synchronous call from the subscribing process to the single threaded Tickerplant. The potential drawbacks are that each subscribing process must have access to the Tickerplant log file system, and that the replay can be time-consuming– especially towards the end of a day– and as a result could cause a backlog in the Tickerplant.

In some cases, the fact that the count of the number of messages (written and published) are not separated by table can be problematic, particularly in cases where only smaller subsets of tables or data is required by the subscribing process. This is where an RDB-oriented recovery approach would be recommended.

The TorQ framework introduces some modifications to the Tickerplant log file approach to allow for more granular log files. The approach is configurable, with the default operation being a log file per table and hour.

6.5.2 Recovery using the RDB

As mentioned above, it can be time-consuming to replay an entire logfile, particularly towards the end of a day. In some cases, where data for all tables is required, this may be a necessary hit. However, there are certain circumstances where it may not be needed or desirable to replay an entire logfile, for example:

- Not every message is required e.g. building a cache of latest prices
- Only specific tables are required so some tables can be ignored

Both these scenarios lend themselves to opting for an RDB-oriented recovery method.

Excluding tables from the replay will save time, as will excluding unnecessary data. The case of building a cache might be as simple as a query to the RDB, e.g.:

```
select last price by sym from quotes
```

However, replaying data for only specific tables, it is more complex. If the tickerplant records the number of ticks (N) processed per table, then subscribing processes could use that information to query the RDB to recover the current day's data, up to N messages for that table, e.g.:

```
select from quotes where i<N
```

The flow is outlined below.

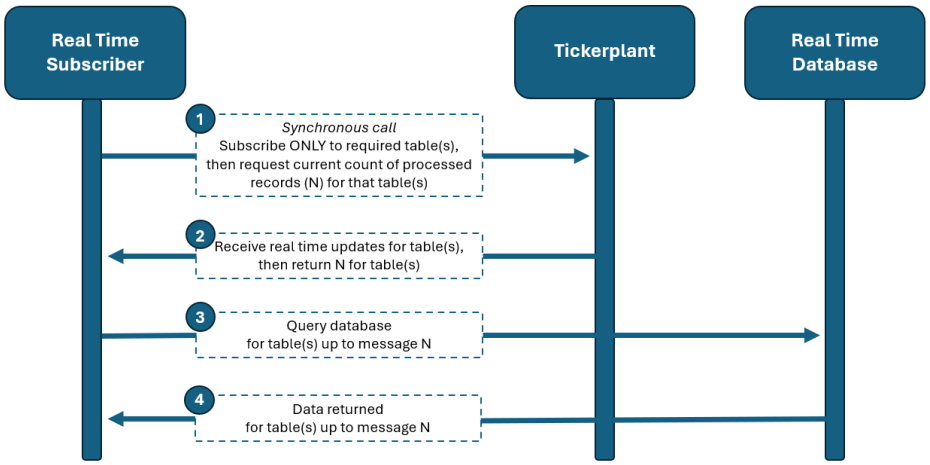


Figure 12 Recovery using the RDB

This assumes that the RDB holds all data for the current day and introduces an additional dependency. For systems with intraday writes, a similar method can be used with a combination of RDB and HDB queries. Understanding the

exact use case for the RTE is key to defining the right approach. An additional benefit of this approach is that subscribers do not need to have access to the Tickerplant log filesystem.

6.5.3 Recovery Using On-Disk Cache Snapshots

Some types of RTE are required to maintain state which may be computationally intensive to build and/or span data across multiple days. An example might be an orderbook building component which includes tracking Good-Til-Cancel orders which were entered many days previously.

In scenarios like this, periodically persisting a state snapshot in a format specific to the task at hand can vastly improve recovery times. In a recovery scenario, the RTE component must read in the snapshot and only replay the data that arrived subsequent to the snapshot being persisted to disk.

Depending on the size and structure of the on-disk cache, persisting to disk periodically may impact the latency profile of the RTE component, which must be considered before utilising this approach.

6.5.4 Downstream Recovery

If an RTE publishes updates directly to downstream subscribers, then it should support the ability for the subscribers to recover a snapshot of the latest data state upon subscription. This enables downstream component recovery.

7 PERFORMANCE TUNING

This section discusses lower level considerations when building latency sensitive KDB-X systems. Some of these approaches could be employed when fine-tuning the performance of a system once any low hanging fruit have been identified and removed.

7.1 Data Structuring and Datatypes

7.1.1 Use of Symbol and Strings

In KDB-X symbols are enumerated internally. The guidance in a KDB-X system is to be very careful about usage of symbol types, and to prefer strings for non-repeating values. A common example of a non-repeating value is order IDs, which do repeat but not enough to be candidates for symbols. Symbols give advantages in terms of storage space and analytic operations like filtering and grouping.

The driver for this recommendation is related to historic storage. The enumerated values feed into a single global list, and if number of enumerated values grows significantly over time it will lead to performance degradation. Therefore, in a real time system symbols could be used more liberally, for fields like order IDs, as long as they aren't persisted to disk in that form to pollute the on-disk database structure.

Encoding of alphanumeric values to other datatypes could be considered for performance reasons. KDB-X contains several utility functions to facilitate it, and custom encoders could be created to encode to larger types like GUID.

7.1.2 Nested Types

Nested types provide a large degree of flexibility within data structures. However, they may introduce an additional memory overhead and can be less performant for computations.

Detail on how KDB-X allocates memory is outlined here. When creating a list of data KDB-X uses a 16 byte header, allocates space for attributes and the data itself, then rounds up to the nearest power of 2. Given the 16 byte header, the smallest size any list can be is therefore 32 bytes. A nested list is a list-of-lists, which uses 8 byte pointers to determine the position in the list. Therefore the minimum memory required for a nested list is 40 bytes per element.

Consider the example of storing an orderbook of data, where the orderbook consists of price levels and corresponding sizes. The orderbook could be stored as two nested lists (prices and corresponding sizes) or as separate columns- individual price and size columns. A graphic example of how the price data of these two options is laid out in memory is depicted below, and performance considerations outlined in the subsequent code sample.

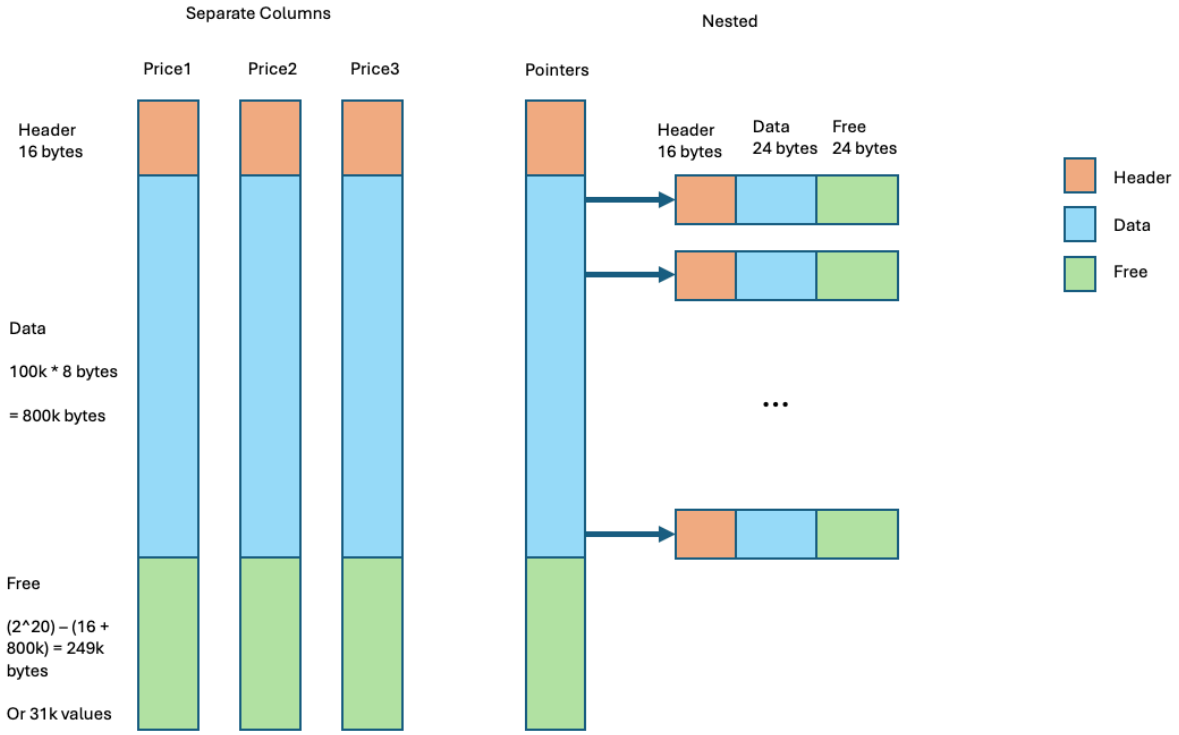


Figure 13 Memory layout of orderbook depth data

The nested approach provides greater flexibility, as it can accommodate orderbooks of potentially any depth. The separate column approach has a fixed depth, which may be acceptable for the analytics employed, and much better storage efficiency.

```
// generate a dataset
// in this case, 100000 rows and 3 price levels
q)builddata:{[n;l] flip (`prices`sizes!(flip p;flip s)),(`$"ps"cross string
1+tl)!(p:(tl:til[l])+\:.1*10+n?1000),s:(l,n)#(l*n)?1000}
q)book:builddata[100000;3]

// sample of the table
q)5 sublist book
prices         sizes       p1   p2   p3   s1  s2  s3
-----------------------------------------------------
88.8 89.8 90.8 537 333 316 88.8 89.8 90.8 537 333 316
55.4 56.4 57.4 861 459 21  55.4 56.4 57.4 861 459 21
44.6 45.6 46.6 953 770 779 44.6 45.6 46.6 953 770 779
24.5 25.5 26.5 592 430 338 24.5 25.5 26.5 592 430 338
95.2 96.2 97.2 583 365 933 95.2 96.2 97.2 583 365 933

// p1->p3 and s1->s3 columns contain 8 byte values
// they each require 1MB of storage space
// 100k 8 byte values, rounded up to the nearest power-of-2
// therefore 6MB in total
q)6*2 xexp ceiling 2 xlog 16+100000*8
6291456f

// size of each element of the price and size books in the nested case are 64 bytes
q)2 xexp ceiling 2 xlog 16+3*8
64f

// nested list as a whole is (64*100000) + (size of list of pointers = 1048576)
// double it for the two lists
q)2*(64*100000) + 1048576f
14897152f
// nested lists in this instance are therefore twice the memory of non-nested

// performance
```

```
// calculate the total quantity available in each book
q)\ts:1000 select i,sum(s1;s2;s3) from book
X

// nested versions
q)\ts:1000 select i,sum each sizes from book
30X

// can also do like this, though this is essentially
// the same as converting back to the separated format
q)\ts:1000 select i,sum flip sizes from book
10X

// what is the weighted average price for each book entry?
q)\ts:100 select i, (s1;s2;s3) wavg (p1;p2;p3) from book
Y
q)\ts:100 select i,sizes wavg' prices from book
23Y

// assuming we wish to trade 1000 shares,
// for each book entry, how many levels of the book do we have to trade through?
q)\ts:100 select i, sum 1000>=sums(s1;s2;s3) from book
Z
q)\ts:100 select i, 1+(sums each sizes)bin' 1000 from book
20Z

// assuming we wish to trade 1000 shares
// what VWAP price will we achieve through each book entry?
q)\ts:100 select i, {[p;s;o] (deltas o&sums s) wavg p}[(p1;p2;p3);(s1;s2;s3);1000] from book
A
q)\ts select i, {[p;s;o](deltas o&sums s)wavg p}[;;1000]'[prices;sizes] from book
100A
```

7.2 Parallelisation via Secondary Threads

Secondary threads can be used to parallelise in-memory calculations. However, in a latency sensitive environment the datasets operated on may not be sufficiently large to benefit from the parallelisation overhead.

```
// extending the example above
// generate two datasets
// 100k rows, 3 price levels and
// 100 rows, 3 price levels
q)builddata:{[n;l] flip (`prices`sizes!(flip p;flip s)),(`$"ps"cross string
1+tl)!(p:(tl:til[l])+\:.1*10+n?1000),s:(l,n)#(l*n)?1000};
q)book:builddata[100000;3];
q)smallbook:builddata[100;3];

// assuming we wish to trade different sizes
// what VWAP price will we achieve through each book entry?

// start with 0 secondary threads
q)\s 0
q)\t:100 exec i,{[p;s;o] (deltas o&s) wavg p}[(p1;p2;p3);sums (s1;s2;s3)] peach 500 1000 1500 2000 from book
X
q)\t:10000 exec i,{[p;s;o] (deltas o&s) wavg p}[(p1;p2;p3);sums (s1;s2;s3)] peach 500 1000 1500 2000 from smallbook
Y

// try with 4 threads
q)\s 4
// speed improvement for larger table
q)\ts:100 exec i,{[p;s;o] (deltas o&s) wavg p}[(p1;p2;p3);sums (s1;s2;s3)] peach 500 1000 1500 2000 from book
0.35X
// but a slow down for the smaller table
q)\ts:10000 exec i,{[p;s;o] (deltas o&s) wavg p}[(p1;p2;p3);sums (s1;s2;s3)] peach 500 1000 1500 2000 from smallbook
2Y
```

7.3 Inter-Process Communication

7.3.1 Serialisation and Deserialisation

Every message between two KDB-X processes must be serialised before it is sent and deserialised upon receipt. These operations require CPU resource. The time taken to serialise and deserialise an object can be measured. In a production system it is very difficult to isolate the time taken for data serialisation and deserialization, and it can become an unseen overhead impacting the system performance. Even if a KDB-X process is being used in a pass through or relay type capacity (i.e. it doesn't modify the message in anyway) the message still has to be fully deserialised upon receipt and serialised on send.

```
// generate a dataset
// in this case, a table containing 10 columns of 0f and 1000 rows
q)builddata:{[n;c;d] flip (neg[c]?`4)!(c;n)#d};
q)floattable:builddata[10;1000;0f];

// time how long 1000 serialisations takes
q)\t:1000 -8!floattable
X

// and de-serialisations
q)r:-8!floattable
q)count r
91023
q)\t:1000 -9!r
3X
```

The times are also impacted by the datatypes.

```
// symbols take longer, even though they produce a smaller result
q)symtable:builddata[10;1000;`abc]
q)\t:1000 -8!symtable
3X
q)count -8!symtable
51023

// strings longer still, and larger size
q)stringtable:builddata[10;1000;enlist"abc"]
q)\t:1000 -8!stringtable
7X
q)count -8!stringtable
101023
```

These serialisation and deserialiation times feed into overall data transfer times.

```
// open connection
// all transfers are on the same host
q)h:hopen 9999

// measure transfer times
q)\t:1000 h(set;`r1;floattable)
Y
q)\t:100 h(set;`r2;symtable)
6Y
q)\t:1000 h(set;`r3;stringtable)
6Y

// full transfer time only marginally more than serialise/deserialise times
q)\t:1000 -9!-8!stringtable
6Y

// nested columns take longer than the equivalent wider table
q)nestedints:builddata[10;1000;enlist 1 2 3i]
q)\t:1000 -9!-8!nestedlongs
Z
q)wider:builddata[30;1000;1i]
q)\t:1000 -9!-8!wider
0.15Z
```

Compression on the wire and encryption will increase the serialisation and deserialisation times. Encryption is not enabled by default, compression is enabled by default for remote host connections where the message size is greater than 2MB.

Datatype choices impact data transfer times. To fully optimise data transfer times, avoid nested columns and symbol types. Symbol types can be passed as enumerated values (integer types) and converted into symbols when presented to the end user or application, though this will increase the complexity of system development. Be aware of overheads associated with "pass through" KDB-X processes which don't modify the message payload.

7.3.2 Async Broadcast

If the same message is being sent to multiple locations, asynchronous broadcast can be utilised to reduce the serialisation overhead. The message is serialised once and sent to all, rather than serialised multiple times.

```
// build data set
q)builddata:{[n;c;d] flip (neg[c]?`4)!(c;n)#d}
q)stringtable:builddata[10;1000;enlist"abc"]

// open 3 connections on localhost
q)hs:hopen each 3#9999

// timings for serialisation
q)\t:1000 -8!stringtable
X

// timing to send async message to one client
// is similar to serialisation time
q)\t:1000 (neg hs[0])({x};stringtable)
X

// timing to send to 3 clients
// approx 3x of sending to 1
q)\t:1000 {x({x};stringtable)} each neg hs
3X

// timing to send to 3 clients using async broadcast
// similar to sending to 1
q)\t:1000 -25!(hs;({x};stringtable))
X
```

7.3.3 Domain sockets

Unix domain sockets can be created for local process connections on Unix based deployments. These are likely to give improved performance in comparison to standard TCP sockets.

7.3.4 Use A Dedicated Message Transport

Consider adding specialist software within a latency sensitive architecture to assist with jobs that could be done in KDB-X but potentially shouldn't. A common example is data distribution to a large number of subscribers with unique subscription requirements. See section 5.5 for more detail.

7.3.5 Changes in IPC

Future versions of KDB-X may bring more configurability to IPC. This document will be updated in time to reflect those changes.

7.4 Memory

7.4.1 Non-Constant Time Inserts

As previously discussed, KDB-X uses a power-of-2 memory allocation system. A driver for this is to over-allocate memory so that when a vector grows in size the system doesn't have to incur the cost of frequent memory re-allocation. When a vector runs out of space and has to grow, the insert becomes expensive.

```
// create a dataset- each column in the table is a vector
// align the size to a memory boundary
q)builddata:{[n;c;d] flip (neg[c]?`4)!(c;n)#d}
q)d:builddata[16777213;5;0f]
```

```
// first 2 inserts are cheap
q)d:builddata[16777212;5;0f]
q)\ts `d insert (1f;1f;1f;1f;1f)
X
q)\ts `d insert (1f;1f;1f;1f;1f)
X
// then expensive
q)\ts `d insert (1f;1f;1f;1f;1f)
>1000X  // much longer

// then cheap again
q)\ts `d insert (1f;1f;1f;1f;1f)
X
```

7.4.2 Deletions

Deletions require memory re-allocation, and this is reflected in their performance.

```
// create a dataset- each column in the table is a vector
q)d:builddata[17000000;5;0f]
q)\ts delete from `d where i>16900000
X

// deleting one record is expensive
q)\ts delete from `d where i>16899999
0.4X
q)\ts delete from `d where i>16899998
0.4X
```

7.4.3 Garbage Collection

The default with KDB-X is to never return unused memory to the operating systems. Memory is instead internally recycled. This can be changed via the garbage collection settings under the -g parameter. Some details are further explained here. With immediate mode garbage collection, KDB-X will return any fully freed memory slices of 64MB or above to the operating system but this has an impact on performance.

.Q.gc can be used for explicit garbage collect which will attempt to more aggressively coalesce memory blocks to return them to the operating system.

```
// create a dummy data set
// 100m rows, universe of 100 syms
q)makedata:{[n;syms] ([]time:asc .z.p+n?0D00:00:01; sym:n?syms; price:n?100f; size:n?1000)}
q)d:makedata[100000000;-100?`4]

// garbage collect, show memory usage
q).Q.gc[]
4294967296
q).Q.w[]
used| 4295395824
heap| 4362076160
peak| 8657043456
wmax| 0
mmap| 0
mphy| 17179869184
syms| 833
symw| 40863

// deferred garbage collect
q)\g 0

// time queries
// test a couple of runs
// subsequent runs are quicker due to memory already allocated
q)\t select sum price, max size by sym from d
X
q)\t select sum price, max size by sym from d
0.2X
q)\t select sum price, max size by sym from d
0.2X

// show memory usage after – heap size has increased
q).Q.w[]
used| 4295396640
```

```
heap| 5435817984
peak| 8657043456
wmax| 0
mmap| 0
mphy| 17179869184
syms| 833
symw| 40863

// Change to immediate garbage collection
q)\g 1
q).Q.gc[]
0

// test queries
// consistent overhead of returning memory to OS
// Queries are slower than with deferred garbage collect
q)\ts select sum price, max size by sym from d
1.15X
q)\ts select sum price, max size by sym from d
0.3X
q)\ts select sum price, max size by sym from d
0.3X

// resulting heap size is unchanged (memory returned to OS)
q).Q.w[]
used| 4295396640
heap| 4362076160
peak| 8657043456
wmax| 0
mmap| 0
mphy| 17179869184
syms| 833
symw| 40863
```

7.4.4 Memory fragmentation

KDB-X can suffer from memory fragmentation. When it attempts to return memory blocks to the operating system it must be able to return them in the same blocks that it requested. If one portion of the allocated block is still in use the block will be retained, even though it is largely unused. Temporary utilisation of lots of small lists can lead to memory fragmentation.

```
q).Q.w[]
used| 427024
heap| 67108864
peak| 67108864
wmax| 0
mmap| 0
mphy| 17179869184
syms| 727
symw| 37786
q){@[`.;x;:;1000000?100]} each objects:`$'10#.Q.a
`.`.`.`.`.`.`.`.`.`.
q).Q.w[]
used| 84313472
heap| 134217728
peak| 134217728
wmax| 0
mmap| 0
mphy| 17179869184
syms| 729
symw| 37847
q){eval(!;enlist`.;();0b;enlist enlist x)} each -1 _ objects
`.`.`.`.`.`.`.`.`.
q).Q.w[]
used| 8815808
heap| 134217728
peak| 134217728
wmax| 0
mmap| 0
mphy| 17179869184
syms| 729
symw| 37847
q).Q.gc[]
0
q){eval(!;enlist`.;();0b;enlist enlist x)} last objects
`.
```

```
q).Q.gc[]
67108864
q).Q.w[]
used| 427200
heap| 67108864
peak| 134217728
wmax| 0
mmap| 0
mphy| 17179869184
syms| 730
symw| 37877
```

8 VISUALISATION

Real time analytics systems usually employ visualisation components to allow users to see and interact with the data, and for monitoring and observability purposes. KDB-X is provided with KX Dashboards embedded for visualisation.

8.1 Polling Or Streaming

The two data delivery models possible when building a real time visualisation are:

- Polling: the UI component periodically re-requests data from the KDB-X system
- Streaming: the KDB-X system publishes updates to the UI component. Updates can be as frequent as the data changes.

A polling model is usually straight forward to implement. A query is built to run against a KDB-X process and return data in the form required by the UI component. A downside of polling is server side load and additional network bandwidth, as the front end may make frequent, potentially un-necessary data requests. The UI is also not event driven, so updates will only be observed as upon each poll.

With thin clients polling requests are usually limited to a single query, meaning that if data is required from multiple sources it must be joined on the server side. Thicker clients allow more flexibility, possibly joining data from multiple sources closer to the UI.

A streaming model usually requires more development investment. A server side component must be built to create the data in the correct shape for the visualisation and published to the front end. The advantages are that data is sent to the UI in an event driven manner, and it is easier to support a larger user base.

With streaming components it may be necessary to recover history. Imagine the example of a ticking price chart covering the last hour- when the UI is opened the history must be recovered from the backend, rather than the price chart "building" from the point that it was initiated. Therefore streaming usually requires implementation of an initial snapshot or recovery query as well.

A common pattern is to build visualisations with polling first, especially when using a dashboarding tool. If the visualisation is proven to be valuable to the business, and event driven updates are required, then it can be converted to a streaming model before being rolled out to a wider user base. Extending the example of a ticking price chart above, it is simpler to implement a single polling query which extracts the last hour of data and refreshes every second, than it is to create an appropriate recovery query and event driven subscription model.

Any streaming updates which are solely for visualisation purposes should have a minimum update latency of 200ms which is approximately the time period required by the human eye to detect simple visual changes.

8.2 Dashboards

Dashboarding packages allow users to rapidly build data visualisations using a set of pre-configured visualisation components. There are three dashboard packages commonly used in KDB-X environments:

- KX Dashboards, built by KX
- Pulse, built by TimeStored
- Panopticon, built by Altair

All three of these packages support rich visualisations for timeseries data and complex interactions with data. They support both streaming and polling use cases.

Dashboarding tools allow for rapid prototyping of solutions. Combined with a productive KDB-X development environment it can lead to workflows where ideas can be rapidly prototyped. Where dashboarding tools sometimes suffer is providing the fine grained control necessary to produce a truly custom visualisation.

Grafana is also used within the KDB-X space, but usually targeted more at monitoring and support use cases than end business users.

8.3 Custom Builds

Custom builds are supported by any of the available interfaces to KDB-X. Custom builds provide greater flexibility at increased development cost compared to Dashboards. Sometimes visualisations are prototyped in a Dashboarding tool before a custom build is initiated.

KDB-X provides a websocket interface, allowing browser based UIs to be built in both a polling and event driven manner.

Integration with the Python ecosystem is via either the officially supported PyKX library, qPython supported by FINOS, or kola. Plotly, Streamlit and Jupyter are common examples.

"Thicker" clients can be built via the C# or Java interfaces. 3Forge provide visualisation components which hook into KDB-X via the Java interface.

KDB-X supports Postgres SQL Interface (pgwire) allowing standard BI tools to connect into KDB-X.

8.4 Scaling

A common pattern is that visualisation components start being used by a small number of users but then roll out across wider groups. If allowed to go unchecked, this can increase load on the KDB-X components either via an increased number of queries running repetitively, or a large amount of IPC overhead. This is similar to the problems outlined in section 5.5.

Consider the use of dedicated server side UI caches, either written in KDB-X or a different technology. These can improve performance and reduce load by centralising and reducing replicated requests to the back end components. They are a relatively straightforward first step towards scaling to a broader user base.

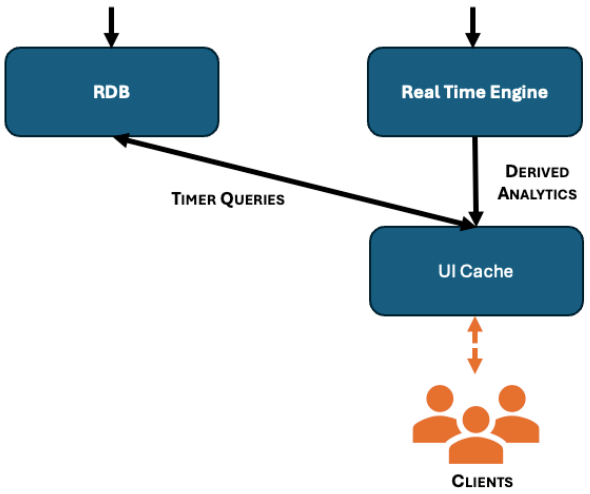


Figure 14 KDB-X UI Cache

Avoid doing a lot of data fan-out directly from KDB-X. For example, consider implementing a dedicated web server to distribute streaming data to large numbers of end users.

9 MONITORING

9.1 Introduction

The quality of the monitoring in a latency sensitive analytics system is critical for its success. Monitoring in a latency sensitive system can be divided into two types:

1. **Operational (latency insensitive):** to be acted upon by humans, used for general system health and operational/capacity planning.
2. **Programmatic (latency sensitive):** to be reacted to systematically by latency sensitive components.

We will cover both, but programmatic monitoring is more relevant for the purposes of this document. An event driven KDB-X system consists of a series of processes with asynchronous data flows. A latency sensitive component may have a dependency on an upstream component or data flow but without a direct connection to it- it is critical that components are able to respond to upstream failures or staleness.

9.2 Operational Monitoring

The services and applications within a data system must be constantly watched to ensure they are functioning as expected. This is done by collecting and analysing data related to the status and performance of the system. Doing this will allow support teams to keep an eye on the health, availability, component response times and allow for longer term capacity planning.

There are a number of standard monitoring and observability tools. Some are cloud based, and some are deployed on premise. They include ITRS Geneos, DataDog, Dynatrace and Prometheus. They all follow a similar pattern- a monitoring agent runs on the host and collects system and process level statistics, for example on usage of memory, CPU, storage and network. The host level statistics are collected and aggregated, providing a centralised view. Log file aggregation capabilities are also important, though may be provided by a separate tool.

Monitoring tools also allow for collection of application level metrics which provide indicators specific to the health of an application. In a KDB-X system these will be specific to the task at hand but likely include metrics associated to:

- Table counts and update rates
- Tickerplant queue lengths
- Process responsiveness
- Query counts
- Etc.

Application metrics should be centrally collected on the KDB-X side and then pushed to the monitoring application. This can be done by a dedicated KDB-X process. It is better practice to only collect and propagate metrics, and delegate status determination and alerting to the monitoring application. The configuration of alerts and comparison to historic values is best done in the monitoring application.

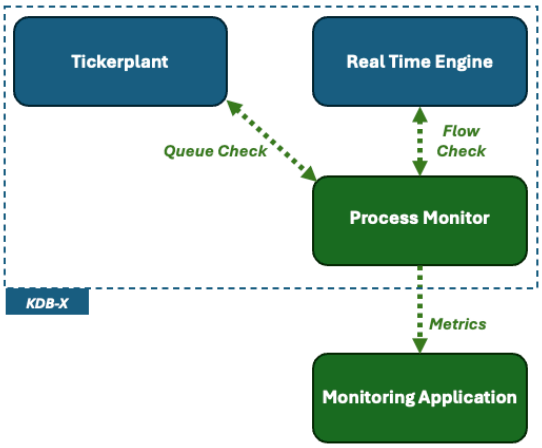


Figure 15 Operational Monitoring

Consider the example of a table flow check metric, where the goal is to ensure that data is flowing into a table at a suitable rate. There are nuances to consider:

- How do we determine what an appropriate flow rate is? Data flows at different rates at different times of the day
- A single table may be updated from several upstream sources, we must ensure that all the upstream sources are monitored

To determine appropriate rate, most monitoring tools allow for comparison to historic profiles with allowances for time of day, time of week, or time of year. This allows for alerting on failing data sources and early warning indicators of high activity days.

For the latter point, we need to ensure that the monitoring is broken down granularly enough to attribute it back to the original collection source, and each source must be monitored independently.

9.3 Programmatic Monitoring

Programmatic monitoring is concerned with ensuring that the data being acted upon by analytic components is non-stale. Whilst the "flow check" outlined above is simple and useful from a general system health point of view, it is not useful from the trading analytics standpoint primarily because it is too slow- the system will have been running on stale information for too long.

Consider the following architecture, where a series of processes have upstream dependencies.

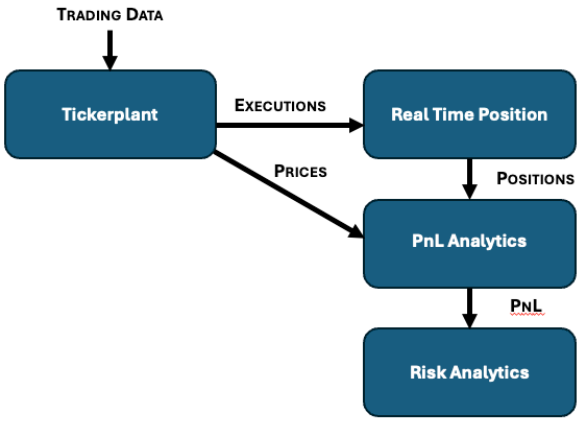


Figure 16 Example of internal system dependencies

- Real Time Position has a dependency on the Tickerplant and the execution data
- PnL Analytics has a dependency on the Tickerplant, Real Time Position and the prices dataset
- Risk Analytics has a dependency on PnL Analytics

Risk Analytics therefore has an indirect dependency on Real Time Position, Tickerplant, execution and price data. If any one of these fails, the output of Risk Analytics is invalid.

The mechanism to achieve this dependency management is for each component in the chain to publish it's status in a programmatic way, and for downstream components to act accordingly on that, which may include updating their own status. If Real Time Position publishes a "stale" status (which may be invoked by a "stale" status coming from the execution feed), PnL Analytics should follow suit as should Risk Analytics.

As with the Operational Monitoring and "flow check" example, there may be nuance- there are scenarios where the status needs to be more granular, rather than an "all or nothing" update. For example, a segment of the pricing feed may become stale, which would mean that only some of the PnL analytics are stale.

From the KDB-X standpoint, the status propagation approach should be combined with process status dependencies. For example, if the connection between Real Time Position and the Tickerplant is broken, triggering a .z.pc callback, this same information should propagate through the stack via publication of a stale message from Real Time Position.

In some scenarios an additional monitoring process can be introduced to calculate monitoring metrics and enable latency sensitive monitoring along with remedial actions.

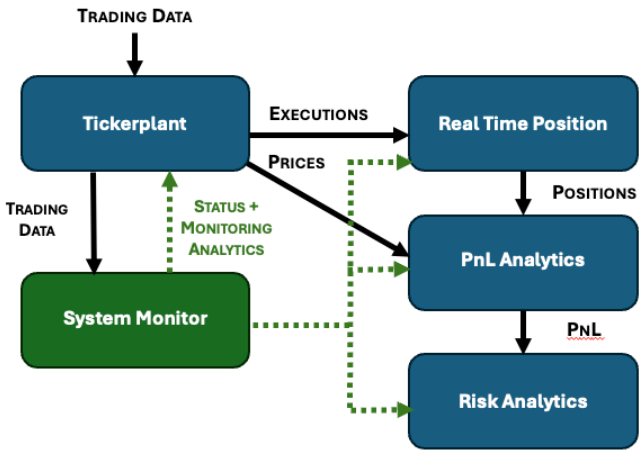


Figure 17 Dedicated System Monitor

An example of this type of system monitoring is with stale checking on high volume data flows. Crude checks such as "must receive an update every 2 seconds" will not suffice in these scenarios. We previously presented a blog highlighting an approach on how to detect and react to stale data in low millisecond time ranges. This analytics are relatively expensive to track, so it is best to calculate once and distribute the information within the components of the system, allowing each dependent process to react accordingly.

9.4 Environment Consistency Monitoring

When running in a replicated Hot-Hot set up, both sides will be producing analytics. We need to ensure that, given the same inputs, the analytics produced are the same. To do this, we can implement a consistency checker which subscribes to data from both sides and compares them. The consistency checker needs to check that analytics are:

- Being published for the same set of data.
- Being published with the same values within an acceptable time period

This type of monitoring is difficult to react to programmatically because assuming that none of the operational or programmatic alerts have been triggered on either side, it will usually require human investigation to work out the difference and which side of the system is correct. They will usually be related to an unforeseen difference in a dataset or potentially a race condition.

9.5 Monitoring Subsystem

Monitoring data is timeseries data, KDB-X is a timeseries database. Monitoring data can be fed back into KDB-X and used for analysis alongside other market and system data.

10 CONCLUSION

Thank you for reading this document. We welcome feedback and corrections.

Designing a performant system requires an understanding of the target objectives, high level design principles, and how low level choices will impact the system as a whole.

If you would like to talk to us about any aspect of kdb+ or KDB-X please get in touch to info@dataintellect.com.

[CHALLENGE ACCEPTED.