First Derivatives plc

KDB+ Reference Manual 3.0

in association with kxsystems

First Derivatives plc
Kdb+ Reference Manual 3.0

Contents

About First Derivatives

First Derivatives plc (www.firstderivatives.com) is a recognised and respected service provider with a global client base. FDP specialises in providing services to both financial software vendors and financial institutions.

The company has drawn its consultants from a range of technical backgrounds; they have industry experience in equities, derivatives, fixed income, fund management, insurance and financial/mathematical modeling combined with extensive experience in the development, implementation and support of large-scale trading and risk management systems.

About Kx Systems

Kx Systems (www.kx.com) provides ultra high performance database technology, enabling innovative companies in finance, insurance and other industries to meet the challenges of acquiring, managing and analyzing massive amounts of data in real-time.

Their breakthrough in database technology addresses the widening gap between what ordinary databases deliver and what today's businesses really need.

Kx Systems offers next-generation products built for speed, scalability, and efficient data management.

Strategic Partnership

First Derivatives have been working with Kx technology since 1998 and accredited partners of Kx Systems worldwide.

First Derivatives offers a complete range of Kx technology services:

Training

Systems Architecture & Design

q development resources

Kdb+/tick implementation and customization

Database Migration

Production Support

Feedhandler developments

First Derivatives Services

First Derivatives team of Business Analysts, Quantitative Analysts, Financial Engineers, Software Engineers, Risk Professionals and Project Managers provide a range of general services including:

Financial Engineering
Risk Management
Project Management
Systems Audit and Design
Software Development
Systems Implementation
Systems Integration
Systems Support
Beta Testing

Contact

North American Office (NY):	+1 212-792-4230
European Office (UK):	+44 28 3025 4870

USA

John Conneely : jconneely@firstderivatives.com

Toni Kane: tkane@firstderivatives.com

Europe

Michael O’Neill : moneill@firstderivatives.com

Victoria Shanks: vshanks@firstderivatives.com

Introduction

This manual draws extensively from documentation (in some cases the content is produced verbatim) available on the KX Systems website including;

- Kdb+ Database and Language Primer
- Kdb+ Database Reference Manual
- Abridged kdb+ Database Manual
- Abridged q Language Manual
- q Language Reference Manual
- Entries on the kdb+ listbox

The purpose of this manual is to provide a reference guide which collates and organizes all publicly available documentation related to kdb+. First Derivatives personnel will update the manual on a regular basis as new features are added to the product. We have the largest concentrated pool of kdb+ expertise in the world and we will be including practical examples from our work in the field. Should you wish to make any contributions we will be happy to include them if they are appropriate. To receive the latest version of the manual e-mail Victoria Shanks (vshanks@firstderivatives.com).

The KX Systems website provides a succinct introduction to kdb+ and it is reproduced below.

What is kdb+?

Kdb+, introduced in 2003, is the new generation of the kdb database. Like kdb, kdb+ is designed to capture, analyze, compare, and store data -- all at high speeds and on high volumes of data. But more than that, kdb+ was architected specifically to meet the emerging needs of leading-edge, realtime business.

How is kdb+ suited for realtime business?

Most data management/data analysis solutions divide the world into realtime/in-memory/front-end data and historical/on disk/back-end data. The division makes it easier for partial approaches to claim proficiency at one or the other. Having separate front-end and back-end data management worked all right until recently. Now enormous growth in the data volumes collected by business, along with the need for instant analysis of data, and realtime comparisons of in-memory to historical data, are becoming critically important to competitive differentiation. The firms that are first to market with these realtime business applications are the ones who can maintain and expand their competitive strategies.

With kdb+ there is no architectural split between the front end and the back end data management and analysis. We provide a single architecture for managing and analyzing data across the entire data management chain, maintaining exceptional performance throughout. In addition, kdb+ was designed from the outset to use 64-bit memory, because 64-bit addressability is essential to holding increasing volumes of streaming data in memory. It was also architected for extremely low latency, enabling such time-critical applications as auto-trading and realtime risk management.

To assist customers transitioning from 32-bit to 64-bit architectures, we have added a binary-compatible 32-bit version. But the fundamental design of the software takes full advantage of 64-bit platforms. Kdb+ gives you unlimited room to grow.

Why is a unified architecture so important?

It enables leading-edge customers to rapidly develop and deploy realtime applications that deliver high-performance for business-critical applications including: operational risk management, backtesting of trading strategies, business activity monitoring, and other applications that quickly identify out-of-range patterns so that the business can respond in realtime.

The greater performance lead that kdb+ gives our customers translates to increased capability to create competitive strategies.

Why did you develop a next-generation database product?

Kx was founded in 1993, and our kdb database has been in use by leading firms since 1998. In that time, we have seen customer needs evolve. A major business driver for the enterprise today is the requirement to analyze increasing volumes of data – on financial or energy trading transactions, for telecom usage analysis, for realtime CRM, in regulatory compliance/risk management, and in other high-volume areas. Firms need immediate results on these analyses, even when billions of records are involved. That's what realtime business is all about: viewing and analyzing what is occurring in the business right now and comparing it on the fly to historical patterns. Developed for high data volume applications, kdb+ expands a firm's ability to capture, analyze, compare, and store enormous amounts of data -- both streaming and on disk -- with analysis results in realtime.

Is kdb+ used only as an in-memory database?

No. Kdb+ provides a full relational database management system with time-series analysis that handles data in memory as well as stored data on disk. For advanced applications such as backtesting of auto trading strategies or operational risk management, it is essential to be able to compare streaming data against history. You must be able to understand where the business has been in order judge and act upon realtime occurrences. Approaches that handle in-memory data alone or historical data alone can't meet the needs of today's realtime enterprise, where accurate comparison on the fly is becoming increasingly important. Approaches that try to combine a streaming or in-memory product from one vendor with a historical product from another can't deliver the performance necessary for realtime business, because they have to cope with two separate architectures. Excess overhead is unavoidable with multiple architectures.

Which platforms does kdb+ run on?

Kdb+ is available today for industry-standard 32- and 64-bit architectures (AMD Opteron, Intel Xeon, and Sun) running Linux, Windows or Solaris

I've heard it's not possible to run SQL series on streaming data. Is that true?

That's untrue. Our customers have been running time-series or SQL queries on streaming data since 2001 and achieving results in realtime, even on complex queries involving millions of records.

What features contribute to the performance of kdb+?

We've refined the architecture in a number of ways, based on the company's 10 years of experience:

· We expanded the data types for greater flexibility, particularly in writing time-series analytics. While other time-series companies supply a limited time-series language, kdb+ was specifically developed to let leading-edge customers go beyond limits.

· We enhanced the speed and efficiency of application development by combining our general programming, relational, and time-series languages into a single, concise programming language – q. The q language is integrated into the database, contributing to very high query performance. q uses English-like commands and a simple syntax. C++ or SQL programmers typically learn q in less than a day. (See the Kdb+ Primer written by Dennis Shasha, Associate Professor of Computer Science at NYU's Courant Institute.)

· We reduced overhead and latency to maintain leadership performance even as data volumes keep rising. For example, data on many securities exchanges is doubling each year. Our product strategy has always been to maintain the lead in performance for complex data analysis, and with kdb+ we have further extended that lead for our customers.

As a relational database vendor, how do you handle streaming data?

Our product kdb+tick is a realtime ticker-plant application layered on kdb+. As data streams in from a data feed or other source of streaming data, it becomes available for immediate relational analysis. In addition, the data is logged so that, in case of a system failure, you do not lose the day's data, as you would with products that support streaming or in-memory data only. Periodically, the log file is written to the historical database -- a day's worth of realtime data (easily 50 million records) can be written to the database in couple of minutes. In fact, kdb+tick is so fast at managing streaming, in-memory, and stored data that some of our customers have used it to eliminate the traditional end of day, where the database is taken off-line. Because kdb+ runs at top efficiency 24x7, it can be used to program advanced applications such as global 24x7 trading.

Is it really necessary to save all that data?
Only if your firm's strategy is to offer highly competitive, leading products. One of the reasons we developed kdb+tick originally was in response to trading departments asking us: isn't there a way we can save the streaming data so we can analyze it later? While it's true that small trading problems can be solved using a streaming data or in-memory database alone, big, strategic problems require you to be able to save data and to compare streaming or in-memory and historical data on the fly, without losing speed anywhere along the line.

Aside from kdb+tick, do you have other layered products for kdb+?
To date, we have two in addition kdb+tick:

- Kdb+tow is an application that enables traders to test sophisticated algorithms by replaying historical ticks through their models.
- Kdb+taq is a fast loader for NYSE TAQ data (distributed via CD/DVD or FTP) that enables you to create a full 10+ year history of NYSE TAQ data quickly, update it daily, and have it immediately available for relational, time-series analysis in kdb+.
- Kdb+x is a family of eXchange loaders for other sources, for example the LSE Tick and Best Price Data.

Why should development teams and IT departments invest in new technologies such as kdb+, when the trend is toward standard technologies?
Doing business in real time demands new technologies and fast ROI. The volumes of data encountered in business today are like nothing the world has seen before -- and they are growing rapidly. In addition, firms need to understand how streaming data relates to historical patterns. Conventional database paradigms are floundering, because the relational databases of the 1980s are no longer able to keep up with escalating volumes of data. The old model of overnight reporting is no longer acceptable in realtime business. The business intelligence/OLAP/data warehousing structures that were built to make relational databases more efficient are also under increasing pressure to deliver faster analysis -- and they can't. Newer in-memory databases and streaming data products deliver speed as long as the data is in memory, but they don't meet the needs of realtime business, because they solve only a small part of the data volume and data analysis problem.

What if I've already invested considerable resources in developing Java, C, and .net programs?
Kdb+ provides native C and Java interfaces. In addition, to make up for Java's inability to handle large arrays, you can use our JDBC driver. To further assist you, the q language data types map directly to Java and .NET.

Is it complicated to administer a kdb+ database?
Not at all. Kdb+ is remarkably simple to manage, because native operating system routines are used for much of the file management, including backup and restore.

Do you need a big server to run kdb+?
No. Most of our customers begin with a 2- or 3-CPU system and grow from there. As you build a historical database, you will need multi-terabyte storage, but kdb+ is flexible -- you can use local storage, SANs or any combination.

But what if you need a high-availability environment?
No problem -- get as big and redundant as you want to. Many Kx customers have implemented large, fully-redundant systems, including redundant ticker-plants for kdb+tick. We support failover, so there is no loss of data or performance. We provide local logging as well as complete replication between data centers. Through our relationship with Cassatt, we also enable IT organizations to

deploy kdb+ in distributed environments and dynamically allocate system resources to meet realtime spikes, such as unusual peaks in market data. That way you don't need to over-invest in big hardware dedicated to kdb+, but you also have extra capacity available instantly, when you need it. Contact us for a demo.

Do you think the kdb+ database will replace Oracle, DB2, SQL Server and other relational databases?

As long as another database meets your needs, use it. But for applications where you're waiting too long for reports, or you don't have the data for implementing a realtime business application, consider kdb+.

Sample uses of kdb+

Kdb+ is used for a wide variety of purposes in many of the world's largest institutions. Some sample usages are given below.

Market Data Capture and Distribution

- Capture data from worldwide markets seamlessly
- Data feed agnostic – Reuters, Bloomberg, internal feeds, Opra
- Create feeds directly to kdb+ ticker-plant
- Easy data loading capabilities, e.g.TAQ
- Publish data to TIBCO, Triarch, etc.
- Calculate and republish real-time stats
- Decommission legacy systems such as FAME
- Unlock data in legacy systems such as Asset Control
- Cleanse and enrich data
- Create new internal tickers for back testing/program trading purposes
- No fault tolerance, replication or redundancy issues
- Model calibration
- Store large volumes of analytical data

Research and Modelling

- Store large volumes of historical data
- Replay strategies quickly
- Store large volumes of derived data
- Refine strategies
- Research and develop cross asset strategies
- Monte Carlo simulations
- Simulate quote and order scenarios
- Integrate with charting applications
- Kdb+ facilitates a distributed architecture using thousands of blades

Equity Trading

- Capture equities, futures and options from worldwide markets seamlessly
- Capture Level 2 order book data
- Pre-trade and post-trade analysis
- Market impact calculations
- Relative performance measures
- Develop and create customised indices
- Create volatility surfaces on the fly
- Arbitrary time interval vwaps, nbbo, hlcv
- Easy interface with Excel to allow on the fly pricer creation
- Price complex options in real-time
- Real-time model calibration
- Integrate OTC options with program trading framework

Fixed Income Trading

- Capture bonds, futures and options from worldwide markets seamlessly

- Facilitates statistical arbitrage trading
- Develop and create customised indices
- Create volatility surfaces on the fly
- Analytics such as yield curve construction, tree building, lattice methods readily available
- Price complex options in real-time
- Real-time model calibration – e.g.HW calibration with swaptions
- Integrate OTC options with program trading framework

Compliance

- Surveillance on trade quote and order flow data
- Search for specific potential violation of regulatory rules
- Search for fraudulent trading patterns with aggregate views of potential violations
- Real-time use to avoid future regulatory penalties
- Run on historical data to provide info and ammunition to compliance officers addressing existing regulatory complaints
- Reduce the number of false positives from legacy systems
- Examine order books retrospectively
- Back test new surveillance algorithms
- Facilitate RegNMS compliance
- Facilitate pre and post trade analysis for MiFID compliance

Other Sample Financial Applications

- FX Correlation trading
- Data warehousing
- News delivery services
- Performance management
- Configuration management
- Pre-trade Risk Analysis
- Real-time PnL
- Convertible Bond Trading Systems
- Structured Products with Large Data Issues
- Mortgage Backed Securities Data Problems
- Credit Derivative Analytics
- Back Office Processing of High Trade Volumes
- Monte Carlo Simulations

How to use this manual

This manual can be used as a reference guide or as a means for learning kdb+.

In general users of kdb+ will be developing an application and the manual has been organized with this in mind. When developing an application or a process in kdb+ there are a number of important considerations. The following section entitled Architecture Discussions examines the considerations in the specific context of building a financial application. Similar considerations apply when building applications in other areas.

The table below gives some ideas in terms of how to tackle building an application from scratch.

Reference	
Getting Started	Being comfortable with using the console and displaying data in a web browser is crucial for prototyping. Scripts are also a valuable development aid.
Queries	A good place to start to see the power of kdb+ is to build queries. The sample queries show the ease of syntax of kdb+ and how vast amounts of data can be queried in milliseconds.
Tools for complex calculations	Kdb+ has vector language properties which facilitates elegantly expressing and rapid solving of complex algorithims. The various datattypes can be organized as lists and dictionaries and manipulated using powerful primitives.
Functions	There are a number of standard logical and arithmetic functions.
Working with the database	Kdb+ is a dialect of sql92 but has a number of extensions which make it vastly more powerful and easier to perform complex queries.
Database Administration	This section will be useful to DBAs in particular.
Developing analytics in q	Execution control and function definition are explored in this section.
Interprocess communication	Kdb+ has a number of features which make IPC tasks such as messaging much easier than in other systems
Interfacing with other technologies	Kdb+ can work alongside a number of other programmes such as Java,C# and C++.
TICK,TAQ and TOW	This section of the manual gives some details of three financial applications of kdb+
Failure Management	This section gives some guidance for redundancy, resilience and failover.

Architecture Discussions

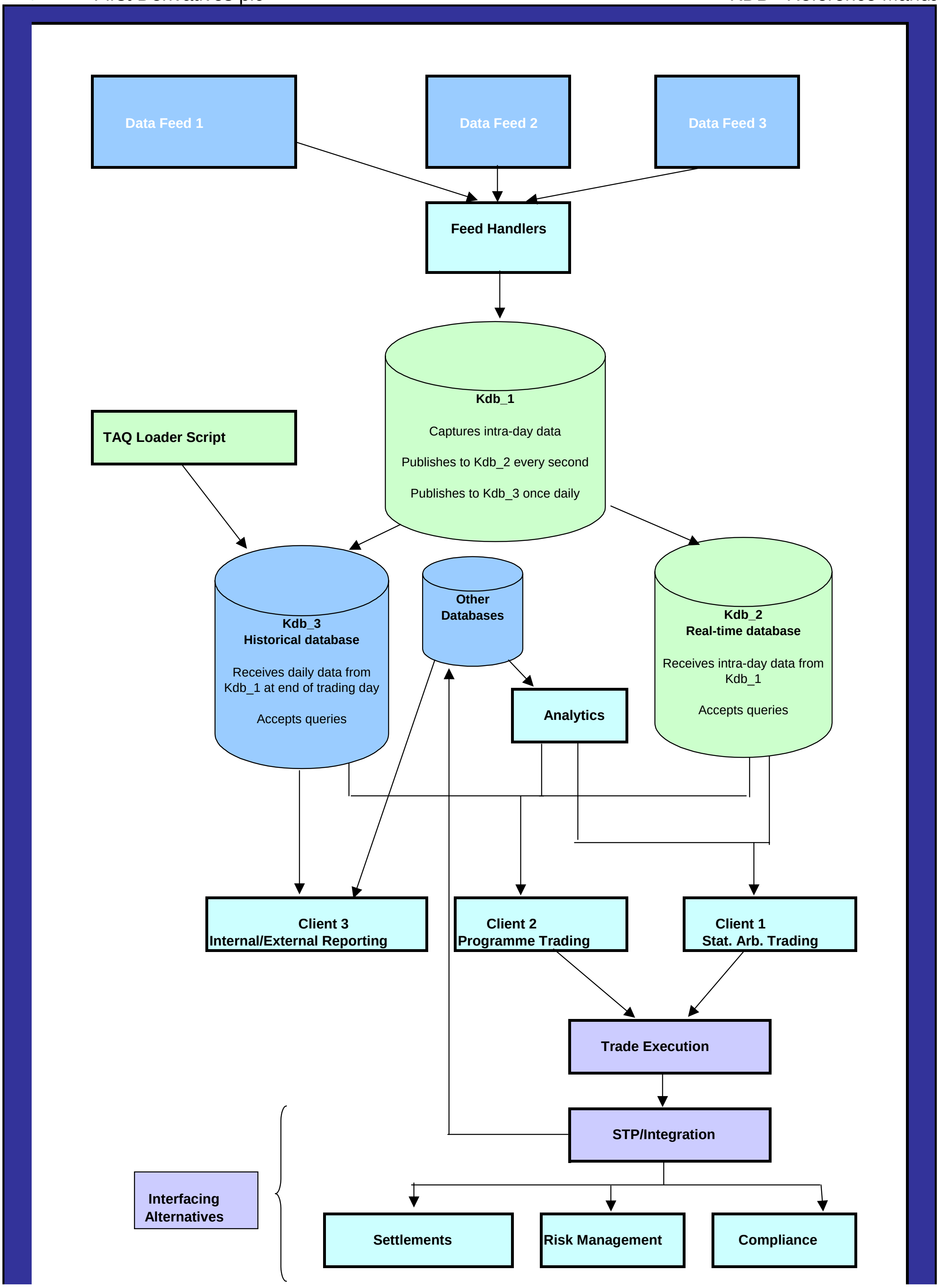
The diagram (see **Kdb+/tick**) shows a simplified schematic representation of how a theoretical kdb+ implementation fits within an equity trading environment. The same principles apply for other trading environments such as an FX or fixed income trading environment. In practice the implementation of kdb+ will vary from institution to institution.

Trading decisions rely on the timely capture of market data from a multitude of sources and the cleansing and enriching of this data to make it suitable for analysis. Kdb+ gives traders a competitive advantage in terms of the speed and quality of data which can be accessed (see **Data Capture and Cleansing**).

The real-time analysis of this scrubbed data facilitates automatic trading based on pre-defined programme trading and statistical arbitrage trading algorithms. The former generally requires historical trading information as does the back testing of strategies. Kdb+ has a number of features facilitating the implementation of real-time trading strategies which would be impossible under other architectures. The historical database can also be used for internal and external reporting purposes (see **Analytics**).

The automatic execution of a transaction is vital otherwise the pre-defined event or profitable opportunity may disappear in fast moving markets. Theoretically before a trade is executed certain checks must be made to ensure for example that market and credit risk limits are not breached. In practice these are often overlooked due to the time taken to undertake the checks. As well as capturing and analysing data kdb+ supports instantaneous transaction execution (see **Trade Execution**).

Each new transaction has associated consequences for other front, middle and back office functional areas such as risk management, settlements, compliance, accounting and portfolio management. Kdb+ reduces the problems associated with Straight Through Processing and interfacing with other internal and external systems (see **Straight Through Processing and Interfacing**).


Data Feed 1
Data Feed 2
Data Feed 3
Feed Handlers
Kdb_1
Captures intra-day data
Publishes to Kdb_2 every second
Publishes to Kdb_3 once daily
TAQ Loader Script
Kdb_3
Historical database
Receives daily data from Kdb_1 at end of trading day
Accepts queries
Other Databases
Kdb_2
Real-time database
Receives intra-day data from Kdb_1
Accepts queries
Analytics
Client 3
Internal/External Reporting
Client 2
Programme Trading
Client 1
Stat. Arb. Trading
Trade Execution
STP/Integration
Interfacing Alternatives
Settlements
Risk Management
Compliance

Data Capture and Cleansing

The volume of market data needed by equity trading desks continues to grow rapidly. As well as stock prices and quotes the data needed includes futures, options, index data (including index options and futures), interest rate and foreign exchange data. The data is published by many sources including exchanges, specialised market data organisations such as Bloomberg and Reuters and data collated internally.

Collecting and cleansing market data poses a number of technical problems which must be resolved if the institution’s trading operation is to rely on the data:

- There are a large number of sources of data which may span internal and external sources and different time zones
- The sheer volume of data can lead to data storage issues and impossible demands on existing system architecture including hardware and databases
- There are peaks and troughs in data which may lead to unacceptable delay in capturing data during peak flow times and may lead to stability and reliability issues
- Enriching captured data (e.g. adding a timestamp or calculating an implied volatility) may not be possible in practice without slowing down the capture process
- Data is stored in different formats and may be needed in different forms for different purposes leading to data mapping issues
- Implementing data cleansing procedures such as filtering, checking data integrity and correctness and correcting data may slow down the overall process
- Different data sources may have different ways of treating market depth which will lead to consistency and comparability issues
- Consolidating streaming market data with data from multiple database sources can be extremely difficult
- The captured data should be capable of handling corporate actions such as stock splits and dividends
- The distinction between real-time and historical data can be blurred in organisations which trade round the clock – it is important that there are no data gaps when transferring real-time data in batch to historical databases
- Procedures must be in place to handle feed failure and server process failures intraday
- It must be possible to query data as it is being stored
- End users should be capable of generating a wide range of ad hoc queries without recourse to additional development resources
- Where possible the mechanism for displaying the data should be independent of how the data is stored and the operating system used
- Migration of new data sources to the existing infrastructure should be relatively straightforward

Kx Systems (www.kx.com) offer a solution which overcomes all of the above issues in the form of the Kdb+/tick product. Kdb+/tick is a single integrated product that enables trading firms to capture, store and allow their traders to query the volume of market data that is required in order to gain a competitive advantage in today’s markets.

Kdb+/tick

- The **Ticker-plant**, **Real-Time Database** and **Historical Database** are operational on a 24/7 basis.
- The data from the data feed is parsed by the feed handler.
- The feed handler publishes the parsed data to the **ticker-plant**.
- Immediately upon receiving the parsed data, the **ticker-plant** publishes the new data to the log file and updates its own internal tables.
- On a timer loop, the **ticker-plant** publishes all the data held in its tables to the **real-time database** and publishes to each subscriber the data they have requested. The ticker-plant then purges its tables. *So the ticker-plant captures intra-day data but does not store it*.
- The **real-time database** holds the intra-day data and accepts queries.
- In general, clients who need immediate updates of data (for example custom analytics) will subscribe directly to the ticker-plant (becoming a **real-time subscriber**). Clients who don't require immediate updates, but need a view the intra-day data will query the real-time database.
- A real-time subscriber can also be a **chained ticker-plant**. In this case it receives updates from a **ticker-plant** (which could itself be a **chained ticker-plant**) and publishes to its subscribers.
- At the end of the day the log file is deleted and a new one is created, also the **real-time database** saves all of its data to the **historical database** and then purges its tables.

The Kdb+/tick solution consists of several unique components:

1 – The Ticker-plant

This is the core component, which is responsible for collecting the daily data from the specified market feed. Currently there are ready built feed handlers for Reuters Triarch and Bloomberg. However it is relatively simple to build custom feed handlers using the C-q interface to link into custom data feeds or even the customer's order and execution feeds. In practice multiple feed handlers can be used to gather data from a number of different sources, both internal and external, and collate the data so that the user has access to all the data that they require simultaneously.

The ticker-plant is a specialized kdb+ process that operates as a link between the client's data feed and a number of subscribers. It receives data from the data feed, appends a time stamp to it, and saves it to a log file. On a timer loop it publishes new data to a real-time database and any clients which have subscribed to it, and purges its tables of data. In this way the ticker-plant uses very little memory, whilst a full record of intra-day data is maintained in the real time database. The real time database can be queried like any other database.

All the data is logged to a log file as it is received to allow for disaster recovery events. The real time database saves the data collected to the historical database on a daily basis.

2 – Real-Time Database

At startup, the real time database sends a message to the ticker-plant and receives a reply containing the data schema, the location of the log file, and the number of lines to read from the log file. The real time database reads the log file to obtain the historic data and subscribes to the ticker-plant to receive the subsequent updates.

It is possible to have multiple real time databases, which are dedicated to specific tasks (see Real Time Subscribers).

The real-time database can support hundreds of clients simultaneously with no noticeable effect on performance. Clients can connect to a real-time database using one of the many interfaces available on kdb+, including C/C++, C#, Java, QDBC and the embedded HTTP server, which can format query results in HTML, XML, TXT, and CSV. Using the C-q Interface or TCP/IP socket programming, custom subscribers can be created using virtually any programming language, running on virtually any platform.

3 – Historical Database

As we have already stated, the real-time database can save market data daily from the real time streaming application to disk and create a historical database. This enables the user to store and analyze virtually unlimited volumes of data. The only limitation on the volume of data that can be stored is hard disk size. With typical volumes of daily trading data for the NYSE alone growing to greater than 60 million records per day (greater than 2GB of storage) and continuing to grow the scalability of any solution will become more and more crucial.

Kdb+/tick analytical performance keeps up with this massive amount of data. For example, Kdb+/tick can analyze one million prices per second per drive and because the historical database is composed of independent segments then we can make use of additional disk drives and CPUs beyond the standard configuration. The recommended minimum configuration for a full tick system is 4 CPUs with 16GB RAM per machine (2 CPUs per machine). Having two disk drives then moving half the segments to the new drive would double performance on multi-day queries by doubling the throughput.

4 - Real Time Subscribers

Real-time subscribers are processes that subscribe to the ticker-plant and receive updates of the requested data, similar to the real-time database. A real time subscriber can subscribe to all the data or a subset of the data. Generally subscription is on a table and list of symbols basis, although table, columns and list of symbols is also possible.

Typical real-time subscribers are kdb+ databases that process the data received from the ticker-plant and/or store it in local tables. The subscription, data processing, and schema of a real-time subscriber can be easily customized.

Kdb+/tick includes a set of default real-time subscribers, which are in-memory kdb+ databases that can be queried in real-time, taking full advantage of the powerful analytical capabilities of q and the incredible speed of kdb+. Each real-time database subscribing to the ticker-plant can support hundreds of clients and still deliver query results in milliseconds. Clients can connect to the subscribers using the same interfaces available to real-time databases.

Multiple real-time subscribers to the ticker-plant may be used, for example, to off-load queries that employ complex, special-purpose analytics. The update data they receive may simply be used to update special-purpose summary tables. Data-cleansing processes such as filtering and corrections can also be created in this way.

Real-time subscribers are not necessarily kdb+ databases. Using the C-q Interface or TCP/IP socket programming, custom subscribers can be created using virtually any programming language, running on virtually any platform.

5 - Chained Ticker-plants

Real-time subscribers can also be chained ticker-plants. This means that they have subscribers themselves which they publish updates to on a timer loop. In most cases, the chained ticker-plant will be publishing processed data. The timer should be tuned to minimize the latency through the system whilst still coping with the potentially large volumes of data.

If a real-time subscriber services a lot of queries every second from the same set of clients, it may be advisable to make it a chained ticker-plant. This will reduce load on the real-time subscriber by reducing the number of queries per second, whilst still providing all clients with the up-to-date information that they require.

However, this may not be a possibility if some of the clients require information as soon as it is available (for example arbitrage program trading), as the chained ticker-plant will increase the latency from the feed due to the extra timer loop. If the timer loop cannot be made short enough, a real-time subscriber which publishes data immediately to clients but is not a ticker-plant may be the better option.

Another reason to use a chained ticker-plant would be if the system infrastructure allowed for ad-hoc ticker-plant subscriptions from potentially many clients. The processing load on the main ticker-plant should be clearly defined to cope with volumes at peak-times comfortably, to guarantee service to mission critical applications. Since ad-hoc subscriptions may substantially increase the load, the subscriptions should be to a chained ticker-plant, ideally residing on a separate server to avoid processor conflict with the main ticker-plant.

Multiple Ticker-Plant Environments

It may often be the case that data is captured from multiple different feeds. It may not be possible to consolidate the data at the feedhandler level and if this is the case multiple ticker-plants should be used, one to capture the data from each feed.

Real-time subscribers can subscribe to multiple different ticker-plants. The real-time subscribers can then be used to consolidate and/or process the data however is required. An example of this would be a risk management application would require data across multiple asset classes.

Analytics

Real-time accurate market data is used as input parameters to analytics used in the practical application of trading decisions and strategies. These calculations are needed in:

- Statistical arbitrage trading
- Program trading
- Benchmarking
- Back testing of trading strategies (see Kdb+/tow)
- Publishing additional market data such as implied volatilities
- Retrospective adjustment of market data – for example in adjusting indices as constituents change or for corporate actions
- Implementing hedging strategies
- Pricing complex equity derivatives

In practice some of the issues associated with applying analytics include:

- The calculation response time is too slow and the market data has changed by the time the result of the calculation is known. This eliminates statistical arbitrage opportunities in practice.
- The volume of data needed in many calculations is too large for the system to handle.
- The analytics may be too complex and too slow to develop in practice – for example most programming languages cannot elegantly handle certain types of queries such as recursive queries which are widely used in program trading.
- Analytics developed in different languages (e.g. C, C++ and Java) cannot be combined easily. Migration of existing analytics to new systems is also often not possible.
- Meaningful interpretation and analysis of time series data is problematical for many programming languages.
- Data mapping issues arise when translating stored and captured data as input parameters for applying analytics.
- Unwieldy legacy architectures mean that only certain front end GUIs can be used.
- Often the required market data is in multiple databases and in multiple formats.
- It is often difficult to allow users to create ad hoc queries without needing additional development resources.
- Integration of the analytical features of an application with the trade execution mechanism is often not possible.

Kdb+ includes a range of powerful features that enable the easy integration of existing analytics and the rapid development of new functionality. These include:

- Support for database connectivity standards.
- APIs.
- A built in web server.
- Time series extensions to SQL that are incorporated in q.

q is ideal for analysing ordered information such as tick data. The power of q and the ease with which it can be extended mean that complex analytics can often be performed within the database, thereby eliminating the need to extract data into a separate application.

Perhaps the most important factor which makes kdb+ extremely useful for analysing equity data is that its design allows it to handle the huge quantities of data required (often several gigabytes per day) and to perform queries that are 100 to 1000 times faster than with other databases. This can often make the difference between being able to run the queries in real-time as opposed to having to run them as an overnight batch process or between querying the full set of historical tick data and only being able to use end of day data.

Trade Execution

Rapid price changes mean that often there is only a small window of opportunity to execute a trading strategy. Whilst more complex OTC equity derivatives can be priced over a longer period of time and long-term investments are less price-sensitive, in the short-term the execution of programme trading strategies is best done automatically. Strategies based on statistical arbitrage trading can only be traded automatically as any arbitrage opportunities are generally quickly traded away, especially in liquid markets.

Apart from the development of robust analytics and the timely capture of market data, the challenges facing any organisation implementing an automatic trading infrastructure include:

- Execution of trades based on triggers is often difficult in practice due to speed constraints and data storage constraints
- To take full advantage of trade matching opportunities the various protocols and standards must be observed
- Cancelled and corrected trades must be dealt with efficiently to avoid misstatement of positions, risk, margins, etc.
- Trade execution based on market quotes must ensure that account is taken of market depth
- Once a trade has been executed positions must be updated to ensure that subsequent calculations are using current figures
- Only authorised users should be permitted to enter into transactions
- Transactions should not breach designated limits such as credit or risk limits – in practice these calculations are often based on approximation or done ex post due to the complexity or speed of the analytics involved
- With the move towards T+1 settlement interfacing and integration issues are becoming increasingly important

Kdb+ offers a solution to this through the way that it can store and manage the data required to satisfy the demand in the current trading cycle. Kdb+ offers the ability to build analytics into its triggers, this fact coupled with the speed and performance of these analytics, means that kdb+ is the ideal platform upon which to build trading strategies that can provide trading decisions in real time.

A trigger is an expression associated with a global variable that is executed immediately whenever the value of the variable is set or modified. The purpose of a trigger is to have side effects, such as setting the value of another global variable, updating a table or making a function call to another program, such as a trade execution system. By using triggers on columns of the real time database then we can execute certain trading strategies every time that an update is received from the feed handler.

Because kdb+ adheres fully to open standards, runs on Linux, Unix and Windows, it can easily be implemented into existing trading architectures, including risk management systems. Developers can leverage their existing knowledge, communicating with kdb+ tables using one of the interfaces provided.

In addition to building analytics into the triggers it is also possible to link to other database tables so that a form of pre trading checks can be built into the functionality so that counterparty limits, permissions, etc. can be checked prior to transaction completion and updating the data real time so that further checks are based on accurate updated information.

Also, First Derivatives have developed a FIX interface to enable automatic electronic trade execution from kdb+.

Straight Through Processing & Interfacing

The drive towards Straight Through Processing is based on a number of substantive benefits such as a reduction in costs and risks, a reduction in settlement failures, the potential to increase trading volumes and to support expansion into new electronic channels

One of the major problems in implementing STP is that in general front office technology is more modern than back office technology. The move towards STP has lead to an increase in the use of newer technology (e.g. the Internet and distributed client/server) such as middleware-centric architectures. The Enterprise Application Integration technology (which facilitates for example the use of queuing technology, guaranteed delivery and rule-based data transformation) needed to enhance legacy technology and to deal with increasing transaction volumes includes:

- Middleware

- XML type technology and;
- Object technologies such as CORBA

Aside from non-technical issues such as variations in national regulations, practices and legal issues there are a number of technical barriers to the goal of achieving T+1 settlement:

- The chain of systems including point-of-trade to point-of-settlement and cross border trade matching utilities is currently fragmented
- There are a number of competing messaging standards such as ISO5022, the FIX protocol and XML. Even within XML there are a number of different standards including FpML.
- In multi-tiered environments, middleware encompass numerous different pipes (e.g., RPC, MOM, ORB) and platforms (e.g., TCP/IP, TIBCO).
- Legacy code may have to be turned into objects
- Any application developed to facilitate STP requires robustness and scalability to cope with increasing transaction volumes
- Security over the Internet is a key issue and any mechanism such as PKI or VPN used to transmit data over the Internet must address issues such as encryption, data integrity, authentication and data integrity.
- Wireless devices are becoming increasingly more common and ideally the infrastructure developed must cater for transactions and operations originating from wireless based applications
- Consideration may be given to the use of an ASP model and shared back office facilities to reduce cost and complexity
- Disaster recovery and high availability is a key issue

Kdb+ includes many features that can address some of these issues:

- Database drivers and easy integration with various programming languages make it ideal for a integrating the range of different technologies and languages that typically need to work together within an STP framework
- Much of the data processing can be done within kdb+ reducing the quantity of data that needs to be passed between applications and providing the necessary performance and scalability.
- While legacy systems can be easily integrated with kdb+ initially, it is possible to gradually transfer much of this functionality into the database itself allowing for a reduction in the number of different middleware applications and simplified system architecture.
- The vector based q programming language can be used for converting large quantities of data between different formats, such as for generating XML, encryption or producing the formats required for wireless communication. This can take advantage of kdb+'s bulk operators to work with multidimensional data to perform high-speed manipulations that are difficult or impossibly slow in other languages.
- Data can be exported from the database in a range of formats.
- Kdb+ has been proven to be extremely robust and can handle much higher level of throughput than traditional databases. The application itself is very small (the setup file is about 200K) and this simplicity means that there is less possibility for errors compared to other databases that are hundreds of Megabytes in size.
- Kdb+'s logging and replication features and simple file management make it easy to develop suitable disaster recovery procedures.
- Database administration is extremely straightforward and therefore there is less risk of error or mis-configuration.

Available Interfaces

Database Drivers

Kdb+ includes a JDBC driver that can allow existing applications to be easily modified to work with the new database. However there is also a lower level QDBC driver that is straightforward to use with Java and C/C++/C# and which can offer higher performance and additional features such as bulk inserts.

There is an ODBC database loader which can load data from an existing database.

Web Server

Kdb+ comes with a built in web server which can be used to return query results in multiple formats - HTML, TXT, XML and CSV. This can, for example, provide options for integrating with Microsoft Excel (using its web queries functionality) or connecting to Perl analytics.

APIs

There are a number of simple APIs that can be used to connect between kdb+ and Java, C/C++ and C#, which allows these languages to be used for the development of analytics if desired. For example, a custom analytic function could be written in C and included in a database query as if it was a built in part of q. Through the languages mentioned above, one can easily connect to Matlab, Mathematica, R etc.

q

Kdb+ comes with the built in programming language q. This incorporates a superset of standard SQL which is extended for time-series analysis and has many advantages over the standard version. It is usually possible for anyone familiar with SQL to learn q in a matter of days and to be able to quickly write their own ad-hoc queries.

Please note that English sentences will be in black, whilst q-language expressions will be in blue.

q contains a number of features and functions that are particularly useful for dealing with time-series data. One of the key ways in which kdb+ differs from standard relational databases is that data is stored in an ordered form. This allows the use of queries that are difficult or impossible in standard SQL to be written simply in q.

e.g. select the closing price for each stock by date:

```
select last price by date,sym from trade
```

This approach can also be used to query the best 5 stocks over an interval, median values, rank, etc.

Another useful feature is that dates and times are stored in an enumerated format that allows their component parts to be used in queries. This allows components of dates and times (like weeks or quarters) to be easily used in aggregation expressions, such as for calculating running totals. In a traditional database handling dates and times can often require the use of multiple tables for these kinds of queries.

e.g. ten minute roll-ups on a stock:

```
select last price, sum size by 10 xbar time.minute from trade where sym=`MSFT
```

q also includes a number of built in functions that can be particularly useful in writing simple queries to retrieve information that is often required for analysing equity data.

e.g. Volume Weighted Average Price:

```
select size wavg price from trade where sym=`MSFT
```

Trading volume above average price by exchange:

```
select above:sum size by date, sym from trade where price > fby[(avg;price);ex]
```

There are also q key words for moving sums/products/averages/minimums/maximums, standard deviation, variance, covariance, correlation, ratios and deltas, as well as a range of mathematical functions.

The q language can often allow additional functions to be written more concisely and to often perform significantly faster than when written in other languages. Much of the power of q is due to the vector-based nature of the language which allows operations to be performed on lists of data.

Efficient Programming

Several things should be noted when attempting to build a system which is as efficient as possible.

Server-side queries and stored procedures

As much work as possible should be done on the kdb+ servers. This is for two reasons:

1. To take advantage of the speed and efficiency of kdb+.
2. To minimize the amount of data being transmitted across the network.

For example, consider the case of finding the maximum price from a trade table for each symbol for one day of trading. Doing this in a client program would entail pulling the entire trade table across the network and writing a function to find the maximum price. On a kdb+ server, this is simply:

```
select max price by sym from trade
```

and then transmitting the resultant table, which will consist of two columns (sym and price) and a number of rows equivalent to the number of distinct symbols. At all times, whether it be in querying a database or executing a function, the data being used should be made as small as possible. For example, assuming mytable has 5 columns but only the first 3 are used, then:

```
select col1, col2, col3 from mytable where col4 = x
```

is always better practice than

```
select from mytable where col4 = x
```

Transmitting data across a network is a common bottleneck. Most networks in production environments (including gigabit Ethernet) will not be able to keep up with kdb+'s maximum data transfer rate of 100Mb/s.

Common queries should be put into stored procedures whenever possible. A stored procedure gives the programmer greater control over the way the server will be used, as malformed ad-hoc queries can block the server. Also, greater restrictions can be placed on a user when the only form of query are stored procedures – a user may be allowed to execute some stored procedures but not others.

Dedicated Servers

If certain time consuming queries are executed often it is usually better to create a dedicated server (real-time subscriber) to handle such queries. For example, consider querying for the volume weighted average price (VWAP) of a given symbol during the trading day. Every time this is calculated the entire trade table must be searched for all the entries for symbol x and the calculation done on the result. The trade table could be millions of rows long. The query would look like this:

```
select size wavg price from trade where sym = x
```

A better approach would be to implement a dedicated VWAP Server as a subscriber to the ticker-plant. For each symbol, the VWAP server would store three things:

1. sum of size * price for all previous trades;
2. sum of size for all previous trades;
3. the VWAP – this is simply (1) divided by (2).

When a new trade is published the calculation to update the table consists of one multiplication, two additions and one division. The query to get the VWAP for a symbol then becomes a look up, greatly increasing the speed of execution.

When dealing with intra-day data it can often be beneficial to nest data in a table by symbol – a lot of queries have the form:

```
select … by … from … where sym = x
```

If this data is nested by symbol then one row of data is required to perform this query. If the data is stored in a non-nested, unordered form the whole table must be searched to get all the relevant entries for this symbol.

QDBC v JDBC

Kdb+ supports interfacing with java via either JDBC or QDBC. JDBC can be useful for quickly interfacing kdb+ with existing front ends and applications. When building applications from scratch, QDBC should always be used, as it is faster, simpler and more powerful.

QDBC pros:

- Faster
- Easier to use
- More powerful & flexible

JDBC pros:

- Can use with existing infrastructure
- Useful for interfacing with 3rd party applications where there is no access to source code

Getting Started

Installation

To install kdb+ you need to have a valid licencing agreement with KX Systems. The installation and license files for Kdb+ must be obtained directly from Kx Systems. When received, copy the license file **‘k4.lic’,** into the q/ directory.

The installation of Kdb+ creates the q directory in the top level directory for Unix and Linux or in C:\ (Windows).

Install

Unzip kdb+ (and k4.lic) in QHOME (default: q).

```
SOL64> unzip s64.zip (executable file is q/s64/q)
LIN64> unzip l64.zip (q/l64/q)
SOL32> unzip s32.zip (q/s32/q)
LIN32> unzip l32.zip (q/l32/q)
WIN32> w32.exe
WIN64> unzip w64.zip (install w32.exe beforehand, see WIN32)
```

Executable is q (on linux/solaris), and q.exe (windows). Put QHOME/{s64|l64|s32|l32} on PATH.

The development environment

q is a programming and runtime environment that starts from and runs in a shell, which is either a Unix/Linux shell or a Windows Command Prompt. After kdb+ is started, the shell becomes a q shell, or q console, in which expressions are entered and evaluated. The activity of entering and evaluating expressions in a q shell is called a q session; the entered expressions and their results are maintained in a q session log. Data created and assigned a name remains available during the current session. The data created during a session are collectively referred to as the workspace.

Using the console

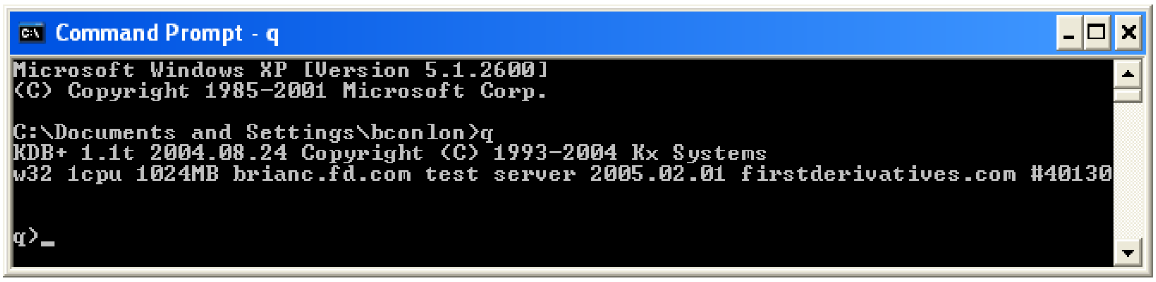
To start using Kdb+ you must start a q session.

An MS Dos Command Prompt terminal can be started on Windows by clicking ***Start/Run.*** Follow this by typing ***cmd*** in the 'Open' textbox and click OK.

The simplest q startup command is simply q. That is, type

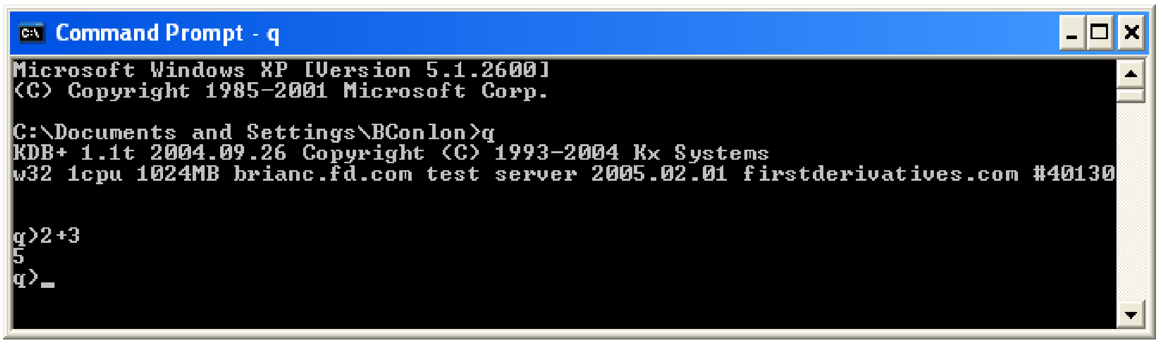
>q

alone on a shell command line and press [Enter]. When q starts up you will see a copyright notice and license information. The OS shell has now become a q shell with a blinking cursor indented 2 spaces. Type any expression and then press [Enter]. The expression you entered will be evaluated and the result (if any) will be displayed below typed, starting at the left side of the shell window. After the result display is complete you will see the blinking cursor again, indented 2 spaces, ready for the next input.



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bconlon>q
KDB+ 1.1t 2004.08.24 Copyright (C) 1993-2004 Kx Systems
w32 1cpu 1024MB brianc.fd.com test server 2005.02.01 firstderivatives.com #40130

q)_
```

For example:
q) 2+3 / you type this
5   / the result is displayed (result displays are not indented)
  _   / a blinking cursor is waiting for the next input expression



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\BConlon>q
KDB+ 1.1t 2004.09.26 Copyright (C) 1993-2004 Kx Systems
w32 1cpu 1024MB brianc.fd.com test server 2005.02.01 firstderivatives.com #40130

q)2+3
5
q)_
```

The annotations on the right are q language comments. A q comment starts with a slash (/). There must be whitespace to the left of the slash. This entry-result format is used throughout this manual to display examples. You can repeat these examples in your shell, or you can make up some of your own.

To exit from the q session at any stage simply enter the command `\\` (double backslash).

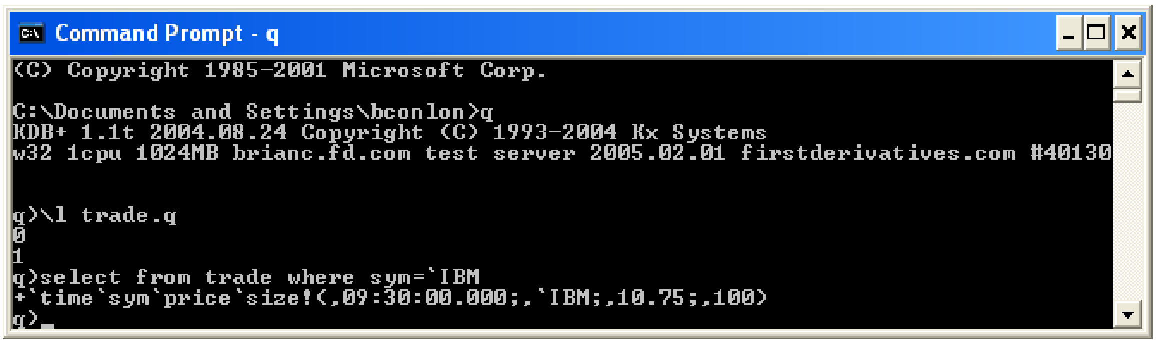
Using script files

Many users prefer to use script files. They are easier to edit and can be stored for reuse. These can be created using textpad or notepad for example and then called from the console. For example a simple trade table has been created in a text file and saved.

```
trade:([]time:();sym:();price:();size:())

`trade insert(09:30:00.000;`IBM;10.75;100)

`trade insert(09:31:00.000;`MSFT;11.00;200)
```

This text file can then be loaded from within the console using the following command:

```
\l [pathname]trade.q
```



```
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bconlon>q
KDB+ 1.1t 2004.08.24 Copyright (C) 1993-2004 Kx Systems
w32 1cpu 1024MB brianc.fd.com test server 2005.02.01 firstderivatives.com #40130


q)\l trade.q
0
1
q)select from trade where sym=`IBM
+`time`sym`price`size!(,09:30:00.000;,`IBM;,10.75;,100)
q)_
```

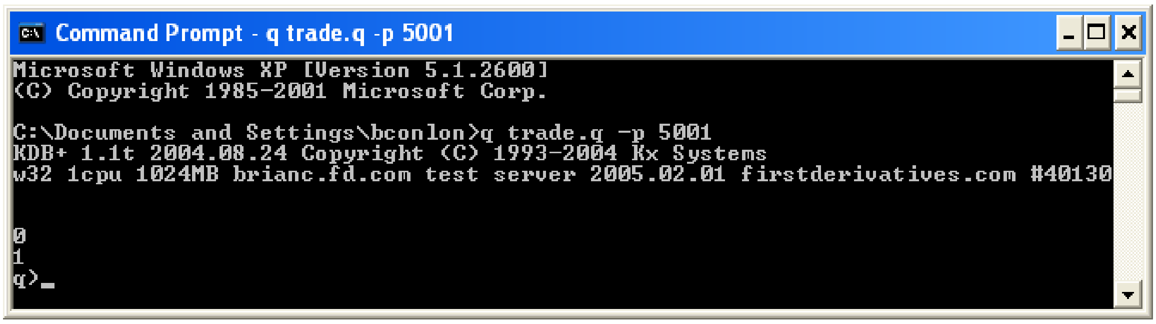
The text file must be given the extension .q (or .k). The pathname used for loading the script in the console is relative to the current working directory. Subsequent commands in the console can refer to data, functions, etc. created in the script. Scripts can contain any expressions or commands, which are executed from top to bottom when the script is loaded. A script can also be named in the startup command or loaded by another script with a load command.

Using a Web Browser

It is possible to view kdb+ tables in a web browser and to use the browser for prototyping and rapid development. A http port (e.g., 5001) needs to be set and this can be done in a number of ways.

-Through the console on start up

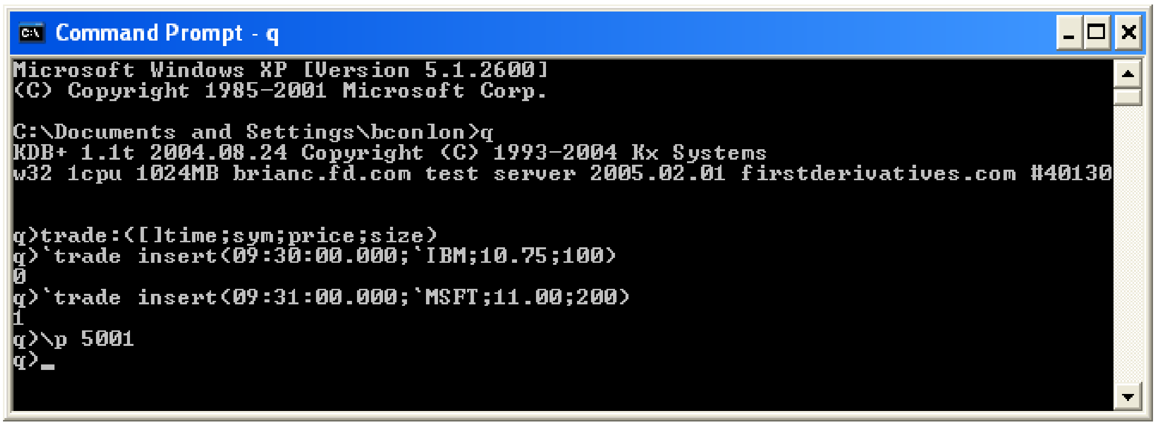
```
q trade.q –p 5001
```



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bconlon>q trade.q -p 5001
KDB+ 1.1t 2004.08.24 Copyright (C) 1993-2004 Kx Systems
w32 1cpu 1024MB brianc.fd.com test server 2005.02.01 firstderivatives.com #40130


0
1
q)_
```

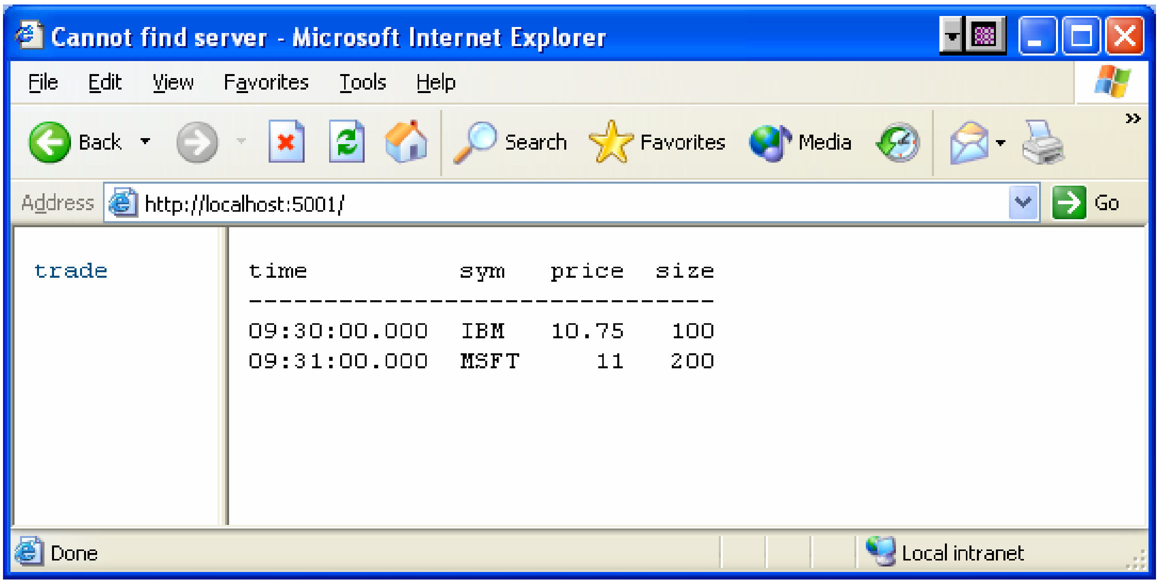
-Through the console once some data has been created

`\p 5001`



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bconlon>q
KDB+ 1.1t 2004.08.24 Copyright (C) 1993-2004 Kx Systems
w32 1cpu 1024MB brianc.fd.com test server 2005.02.01 firstderivatives.com #40130


q)trade:([]time;sym;price;size)
q)`trade insert(09:30:00.000;`IBM;10.75;100)
0
q)`trade insert(09:31:00.000;`MSFT;11.00;200)
1
q)\p 5001
q)_
```

Once the data has been created and the port set, simply open a Web browser and type http://localhost:5001 at the browser. You will then see a browser display of the trade table. For larger databases you will see a list of tables displayed to the left along with other information.



Address: http://localhost:5001/

trade

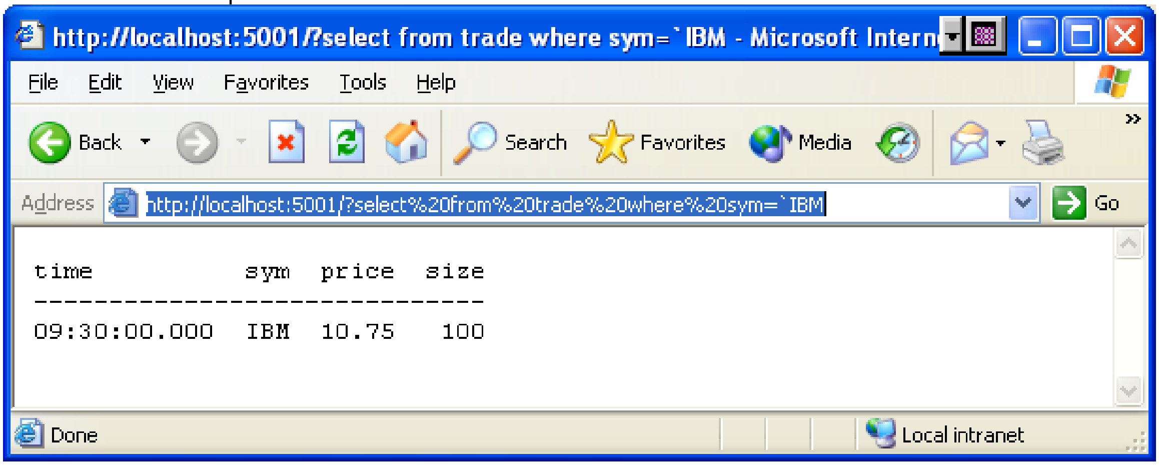
```
time          sym   price size
------------------------------
09:30:00.000  IBM   10.75 100
09:31:00.000  MSFT  11    200
```

Another alternative is to create a batch file which will automatically execute the relevant commands and launch the browser. The script for such a batch file is shown below.

```
start q trade.q -p 5001
start iexplore.exe http://localhost:5001
```

Queries can be tested by entering them them in the url banner, always preceded by the question mark symbol ?. Note that the query is lost once the results are returned.

`http://localhost:5001/?select from trade where sym=`IBM`

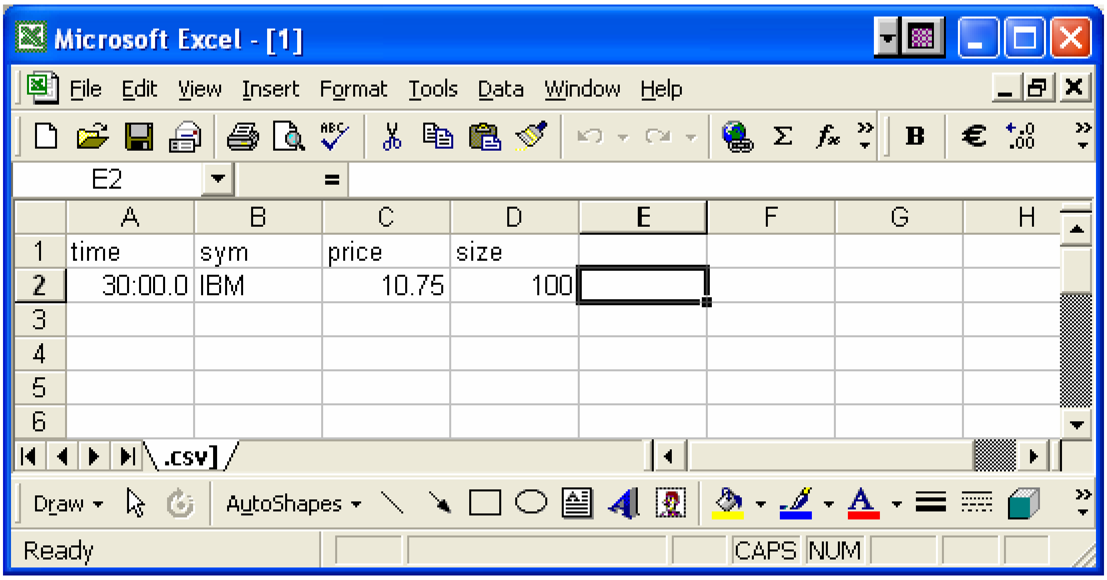

http://localhost:5001/?select from trade where sym=`IBM - Microsoft Intern
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go
Done
Local intranet

http://localhost:5001/?select%20from%20trade%20where%20sym=`IBM

```
time          sym  price  size
------------------------------
09:30:00.000  IBM  10.75  100
```

The results can be directly exported to Excel by prefixing the query with .csv as follows.

http://localhost:5001/.csv?select from trade where sym=`IBM



	A	B	C	D	E	F	G	H
1	time	sym	price	size				
2	30:00.0	IBM	10.75	100				
3								
4								
5								
6								

Commands

Commands are special statements for interacting with the programming environment. All commands have a leading back-slash. The brackets indicate optional content. If the content is omitted then the command displays its current value. The table below gives a selection of the main commands.

Command	**Syntax and Examples**
Tables	\a
This command is used to show the tables in the current workspace.	
Views	\b
This command shows the views in the current workspace.	
console display	\c
This command shows the number of horizontal and vertical lines to show in the console. It can also be used to set them to whatever the user requires.	
browser display	\C
Same as \c except for the browser display.	
dictionary	\d [dict]
This command is used to specify the current dictionary / directory within namespace.	
Error flag	\e
Functions	\f
This command displays the names of the functions in the current workspace.	
variables	\v [dict]
This command displays the names of all global variables in the specified dictionary	
load	\l file
This command loads the named script into the current q session. In most examples in these manuals the script to be loaded is given in the startup command. However, it is also possible to load a script from within the kdb+ console. For example, \l sp.q This is useful when testing changes to a script because changes can be made and repeatedly loaded without exiting from the kdb+ process. Also, commands can be included in scripts. Consequently applications can be partitioned into sets of scripts, all of which are loaded by the top-level script given in the command line.	
Offset	\o
Get or set the offset from Greenwich Mean Time (GMT).	
port	\p [digits]
Set the listening port. As with script loading, in this manual we have set the port with the -p command line option. It can also be set (or changed) in a console, e.g. \p 5001	
Precision	\P [digits]
Get or set the number of significant figures displayed in the console.	
Slaves	\s [digits]
Get or set the number of slaves.	
timer	\t [milliseconds]
Set the timer to the number of milliseconds given by the integer represented by digits, or display the current setting if any arguments are not present. If the setting is a positive integer n then the function .z.ts is called every n milliseconds. No calls are made if the setting is 0 (the default). The function .z.ts is user defined and has no arguments.	
time	\t expression

The execution time of the expression, in milliseconds, will be printed out when execution completes. It often happens that the result is 0, in which case you can execute the expression a sufficient number of times to get a positive result, with which you can compute the average number of milliseconds required for one execution. For example,

```
  \t sum til 15000
0
  \t do[10000;sum til 15000]
750
```

The last result shows that, on average, execution of sum til 15000 takes 0.075 milliseconds (on the author's PC).

workspace \w

This command displays 4 longs summarizing the memory usage of the current kdb+ session. For example,

```
  \w
57904 1056784 0 0j
```

The first number is the number of bytes actually in use at the present time. The second number is the number of bytes allocated by the kdb+ process. The third is the max used memory. The fourth number is the size of the current mapped file space.

First day of the week \W

Gets or sets the first day of the week, with Saturday = 0. The default is 2, meaning that the first day of the week is Monday.

o/s command \text

When the text following the back-slash is not the text of one of the above commands, it is passed to the operating system for evaluation.

Exit kdb+ \\

End the kdb+ session.

Interrupt Ctrl-C

This interrupts the current session.

Debugging

Errors are reported in the console by displaying the error type, generally type or length, the failed primitive function, and its argument(s). If you see the default prompt “q)”, you can simply continue with your work. If you see a leading ")" on the prompt, type "\" and press [Enter]. You should then see the default prompt. Some common errors are shown in the table below.

Common Errors

Error	Meaning	Recreate
'(		
')		
'x	You have referenced a var that hasnt been instantiated	1+a
'access	trying to access an protected (-u) database without a valid username and password	
'assign	trying to assing a built in q function to your own function	abs:2
'branch	Too many statements in a do or if statement	
'cast		s:`a`b`c; a:([]s:`s$`a`b`c`a;b:2 3 4 5); `a insert (`e;1)
'cpu	too many cpus for licence	
'char	invalid character	
'conn	Can't connect - server is down or doesn't exist	h:hopen`:2000
'constants	Too many constants in a q function	
'domain		
'exp	expiry date passed	
'from	Badly formed select statement	select price trade where sym=`A
'globals	Too many global variables in a function. Max of 31.	

'host	invalid host	
'k4.lic	k4.lic file not found, check QHOME/QLIC	
'length		a:([]a:());`a insert 2 3;
'limit	enumerate negative number, trying to send too big a message via IPC (in 2.2 the message has to be less then 536 million bytes in 2.3 less than 2147 million bytes)	key -1
'locals	Too many local variables in a function. Max of 26.	
'loop	dependency loop	a::a
'mismatch	columns that can't be aligned for R,R or K,K	
'Mlim	more than 999 nested columns in splayed tables	
'mq	Multi-threading not allowed.	
'nyi	not yet implemented	load `:a.csv
'os	Operating System error OR wrong os (if licence error)	
'params	Too many params in a function - max of 8. Use a dictionary or array as a workaround.	f:{[a;b;c;d;e;f;g;h;i]}
'Q7	nyi op on file nested array	
'parse	invalid syntax	?1
'part	something wrong with the partitions in the hdb	
'pn	I got this by loading a splayed db, then deleting the files on disk before trying to inspect the table.	
'rank	applying too many arguments to a function	{x+y}[3;4;5]
'splay	nyi op on splayed table	
'srv	attempt to use client-only license in server mode	
'stack	ran out of stack space	{.z.s[]}[]
'type	argument of wrong data type	1+"d"
'u	Trying to partition non-partitioned data	`p#1 2 1 1
'upd	The update date in your license is before the date of the current q version. (2nd date in .z.l<.z.k)	
'user	invalid user	
'value	no value	
'vd1	attempted multithread update	
'view	Trying to re-assign a view to something else	a::2;a:2;
'wha	invalid system date	
'wsfull	Workspace full - out of memory.	

Debugging is explored further once functions have been examined in greater detail.

Queries

The q language extends the capabilities of sql but the expressions are generally shorter and simpler. The main query expression is the ‘select expression,’ which creates new tables from existing ones. In their simplest form, they can extract subtables; however, it is also possible for them to create new columns. Select expressions have the following general form:

```
select columns by columns from table where conditions
```

The subexpression columns following the select keyword is called the ‘select phrase’. Analogously, there is a ‘by phrase’ following the by keyword (columns), a ‘from expression’ following the from keyword (table) and a where phrase following the where keyword (conditions). Note that only the ‘from expression’ is required.

These all run at several million records per second. In general:

```
select [a] [by b] from t [where c]
update [a] [by b] from t [where c]
```

These are similar to (but more powerful than) the equivalent sql

```
select [b,] [a] from t [where c] [group by b order by b]
update t set [a] [where c]
```

In sql the where and group clauses are atomic and the select and update clauses are atomic or aggregate if grouping. In q the where and by clauses are uniform and the select and update clauses are uniform or aggregate if grouping (by). All clauses execute on the columns and therefore q can take advantage of order. Sql can't tell the difference. Sql repeats the group by expressions in the select clause and the where clause is one boolean expression. The q where clause is a cascading list of constraints which nicely obviates some complex sql correlated subqueries and also gets rid of some parentheses. q relational queries are generally half the size of the corresponding sql. Ordered and functional queries do things that are difficult in sql. See http://www.kx.com/q/e for examples.

Note that i is a special token that indicates record handle (row index). Unspecified column names default to the last token in the expression or x, e.g.

```
select count i,sum qty by order.customer.nation ..
```

is short for

```
select x:count i,qty:sum qty by nation:order.customer.nation ..
```

Sample Queries

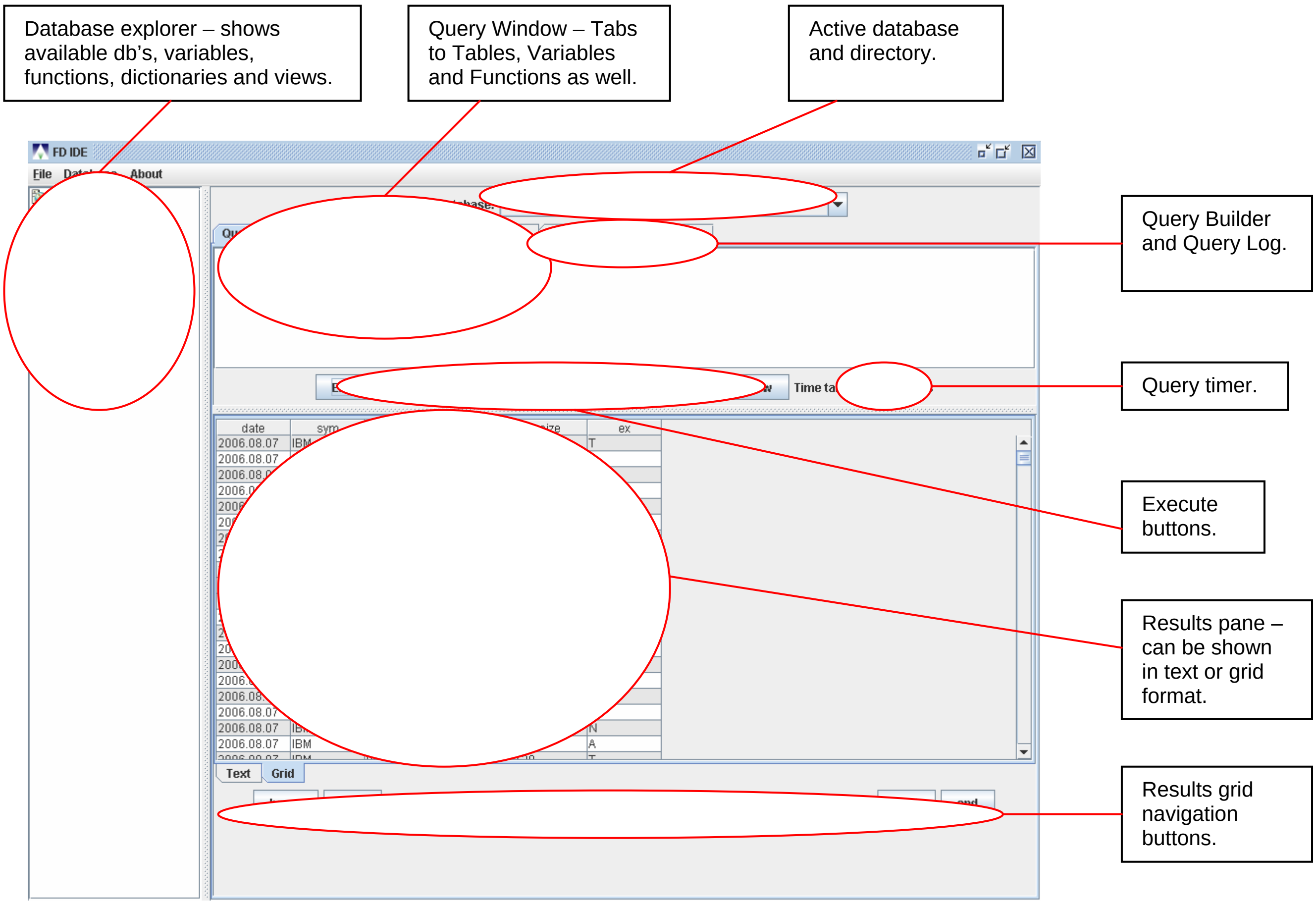
The following queries are based on a standard TAQ (Trade and Quote) database schema. To help with this section save the following script and load.

```
/ build test trade/quote database – increase n to increase the database size

sym:asc`AIG`CITI`CSCO`IBM`MSFT,100?`4;
ex:"NTA";
dst:`$":c:/fdplc/data/";      /Destination of database – can be changed to whatever the user desires
@[dst;`sym;:;sym];
n:1000000;
trade:([]sym:n?`sym;time:09:30:00.0+til n;price:n?2.3e;size:n?9;ex:n?ex);
quote:([]sym:n?`sym;time:09:30:00.0+til n;bid:n?2.3e;ask:n?2.3e;bsize:n?9;asize:n?9;ex:n?ex);

{@[;`sym;`p#]`sym xasc x}each`trade`quote;

d:2006.08.07 2006.08.08 2006.08.09 2006.08.10 2006.08.11; /Date vector can also be changed by the user
```

```
dt:{[d;t].[dst;(`$string d;t;`);::;value t]};
d dt/:\:`trade`quote;
```

Introduction to queries

Queries can be performed in the q-console or in the FD IDE/qDBA (The qDBA cab be obtained from the FD website www.firstderivatives.com/html/dba_reg.asp). Either the IDE or the qDBA application can be used to edit queries in a word-processing style environment, allowing for instance, multiple statements in the Query window, execution only of highlighted text etc. Other functionality includes a Query Log, lists of available functions, variables, tables and the choice of either a grid/table or text result format. In short it is much more versatile than the q-console. A brief overview of the application is given below:


Database explorer – shows available db's, variables, functions, dictionaries and views.
Query Window – Tabs to Tables, Variables and Functions as well.
Active database and directory.
Query Builder and Query Log.
Query timer.
Execute buttons.
Results pane – can be shown in text or grid format.
Results grid navigation buttons.
FD IDE
File
About
Text
Grid

What follows is a number of queries, showing the scope and power of the q-language. They range from simple requests for prices to calculating full trade information, for example High Price, Low Price, Standard Deviance etc. From the above script prices will be generated for the dates 7th to the 11th of August 2006 (this can be easily changed).

It should be noted that when querying a TAQ historical database, if a 'by clause' is being used it should be of the form "…by date, sym, x, y, z …" This offers great speed up in query time, due to the way the data is formatted. The data is partitioned by date (hence the date clause appearing first), and enumerated by sym (hence the sym clause appearing second). All other clauses can be in arbitrary order.

A caveat to the previous paragraph is that a real time database does not have a date column, as it would be superfluous, therefore the "..by date.." construct is not required. In the following queries, the ones which have to be run on a historical database will be prefixed by an asterisk *.

A final note concerning queries is that some q functions do not work over splayed databases, for instance 'exec' and 'median'. A splayed database is a historical database saved to disk where the data is partitioned over several folders (usually the partitions are dates, but theoretically it could be months etc). The solution is to load the data into memory first.

Queries with constraints

* Denotes HDB query.

Select all IBM trades

```
select from trade where sym in `IBM
```

* Select all IBM trades on a certain day

```
thisday: 2006.08.11
select from trade where date=thisday,sym=`IBM
```

Select all IBM trades with a price >100

```
select from trade where sym=`IBM, price>100.0
```

Select all IBM trades with a price less than or equal to 100

```
select from trade where sym=`IBM,not price>100.0
```

* Select all IBM trades between 9.30 and 9.40, in the morning, on a certain date

```
thisday: 2006.08.11
select from trade where date=thisday,sym=`IBM,time>09:30:00.000,time<09:40:00.000
```

Select all IBM trades in ascending order of price

```
`price xasc select from trade where sym=`IBM
```

* Select all IBM trades in descending order of price in a certain time frame

```
`price xdesc select from trade where date within 2006.08.07 2006.08.11, sym=`IBM
```

Select all IBM or MSFT trades

```
select from trade where sym in `IBM`MSFT
```

* Calculate count of all symbols in ascending order within a certain time frame

```
`numsym xasc select numsym: count i by sym from trade where date within 2006.08.07 2006.08.11
```

* Calculate count of all symbols in descending order within a certain time frame

```
`numsym xdesc select numsym: count i by sym from trade where date within 2006.08.07; 2006.08.11
```

* What is the maximum price of IBM stock within a certain time frame, and when does this first happen?

```
select time,ask from quote where date within 2006.08.07 2006.08.11,sym=`IBM,ask=max exec ask from
quote where sym=`IBM
```

Select the last price for each sym in hourly buckets

```
select last price hour:time.hh, sym from trade
```

Queries with aggregations

* Calculate vwap (Volume Weighted Average Price) of all symbols

```
select vwap:size wavg price by sym from trade
```

* Count the number of records (in millions) for a certain month

```
(select trade:1e-6*count i by date.dd from trade where date.month=2006.08m)+select quote:1e-6*count i by
date.dd from quote where date.month=2006.08m
```

* HLOC – Daily High, Low, Open and Close for CSCO in a certain month

```
select high:max price,low:min price,open:first price,close:last price by date.dd from trade where
date.month=2006.08m,sym=`CSCO
```

* Daily Vwap for CSCO in a certain month

```
select vwap:size wavg price by date.dd from trade where date.month=2006.08m,sym=`CSCO
```

* Calculate the hourly mean, variance and standard deviation of the price for AIG

```
select mean:avg price, variance:var price, stdDev:dev price by date, hour:time.hh from trade where sym=`AIG
```

Select the price range in hourly buckets

```
select range:max[price] – min price by date,sym,hour:time.hh from trade
```

* Daily Spread (average bid-ask) for CSCO in a certain month

```
select spread:avg bid-ask by date.dd from quote where date.month=2006.08m,sym=`CSCO
```

* Daily Traded Values for all syms in a certain month

```
select dtv:sum size by date,sym from trade where date.month=2006.08m
```

Extract a 5 minute vwap for CSCO

```
select size wavg price 5 xbar time.minute from trade where sym=`CSCO
```

* Extract 10 minute bars for CSCO

```
select high:max price,low:min price,close:last price by date,10 xbar time.minute from trade where sym=`CSCO
```

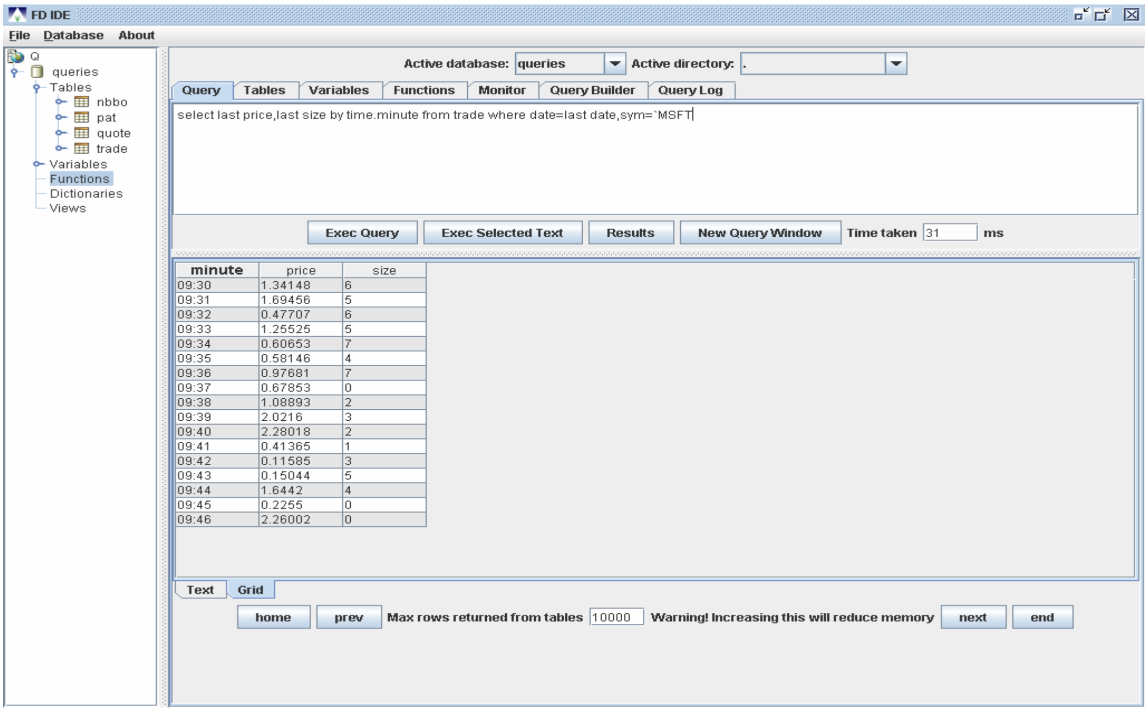
* Find the times when the price exceeds 100 basis points (100e-4) over the last price for CSCO for a certain day

```
select time from trade where date=2006.08.11,sym=`CSCO,price>1.01*last price
```

* Full Day Price and Volume for MSFT in 1 Minute Intervals for the last date in the database

```
select last price,last size by time.minute from trade where date=last date, sym=`MSFT
```

The screenshot below from the FD IDE, shows the above query performed, as one can see it took 31ms on the author’s PC.



Active database: queries
Active directory: .

```
select last price,last size by time.minute from trade where date=last date,sym=`MSFT
```

Time taken 31 ms

minute	price	size
09:30	1.34148	6
09:31	1.69456	5
09:32	0.47707	6
09:33	1.25525	5
09:34	0.60653	7
09:35	0.58146	4
09:36	0.97681	7
09:37	0.67853	0
09:38	1.08893	2
09:39	2.0216	3
09:40	2.28018	2
09:41	0.41365	1
09:42	0.11585	3
09:43	0.15044	5
09:44	1.6442	4
09:45	0.2255	0
09:46	2.26002	0

Max rows returned from tables 10000

* Average 30 minute trade volume over the last date in the database for MSFT

```
select halfhourvol:avg(sum;size)fby 30 xbar time.minute from trade where date=last date,sym=`MSFT
```

* Average total trade volume over a certain week, for each 30 minute bucket for MSFT

```
(select sum size by 30 xbar time.minute from trade where date within 2006.08.07 2006.08.11,sym=`MSFT)% sum date within 2006.08.07 2006.08.11
```

* Average total trade volume over the month for each 30 minute bucket for MSFT

```
(select monthavg:sum size by 30 xbar time.minute from trade where date within 2006.08.01;
2006.08.31,sym=`MSFT)% sum date within 2006.08.01 2006.08.31
```

* Monthly total trade volume average between 09.30 and 12.30 each day for MSFT

```
(select intradayavg:sum size  from trade where date within 2006.08.01 2006.08.31,sym=`MSFT,time within
09:30:00.000 12:29:59.999)%sum date within 2006.08.01 2006.08.31
```

Select the two heaviest traded stocks

```
select sym from 2# desc select sum size by sym from trade
```

* The average, over a certain month, of the total volume in each 30 minute bucket for all syms. This query is made faster by caching advs using a function denoted by curly brackets {}. Functions will be explained later in the document.

```
{[sd;ed]syms:where 1000000<exec avg size by sym from select sum size by date,sym from trade where date
within(sd;ed);(select monthlyavg:sum size by sym,30 xbar time.minute from trade where date
within(sd;ed),sym in syms)% sum date within(sd;ed)}[ 2006.08.01; 2006.08.31]
```

* Construct a table of (closing price – closing price N days ago)/(closing price N days ago) – set x to the required number of days

```
func:{[y] p:x xprev y;(y-p)%p}
select func[price] by sym from select last price by sym,date from trade
```

* Select the spreads that are below the average for every sym over a certain time period

```
select from (select date,sym,spread:bid-ask from quote where date within ((first date),last date),sym in sym)
where spread<(avg;spread)fby ([]date;sym)
```

* Select the 3 smallest spreads for every sym and each date.

```
select from (select date,sym,spread:bid-ask from quote where date within ((first date),last date),sym in sym)
where spread<({(asc x)[3]};spread)fby ([]date;sym)
```

* Hourly trade details for all stocks

```
select start:first price,close: last price,high:max price,low:min price,change:last[price] – first price,range
max[price]-min price,mean:avg price, variance:var price,stdDev:dev price,vwap:size wavg price by
date,sym,hour:time.hh from trade
```

Queries involving table joins

* Generate a NBBO table for a certain month: (National best bid and offer)

```
nbbo: select ask: min ask,bid: max bid by date,sym,time from quote where
date.month=2006.08m,asize>0,bsize>0
```

* RegNMS - trades inside and outside of NBBO for CSCO

```
select sum inside,outside:sum not inside by date.dd from update inside:price within(bid;ask)from
aj[`date`time;select date,time,price from trade where sym=`CSCO;select date,time,bid,ask from nbbo where
sym=`CSCO]
```

* Synthesize and extract a Level1 (last, quantity, best bid, best ask) view on the data for CSCO and MSFT for a certain date

```
aj[`sym`time;select sym,time,price,size from trade where date=2006.08.07,sym in `MSFT`CSCO;select
sym,time,bid,ask from nbbo where date=2006.08.07,sym in `MSFT`CSCO]
```

More information on table joins can be obtained from the JOINS or FUNCTIONS section.

Grouping Without Aggregating

Usually the biggest cost in execution time for evaluating an aggregation, particularly for large tables, is the grouping caused by the "by phrase". Moreover, that cost may occur repeatedly because the same grouping may be done for many different aggregations. In q it is possible to precompute the grouping and save it in a separate table to which aggregations are applied later. The aggregations must be computed differently than in an ordinary select-by expression, but a few examples will show you what to do.

q tables allow any lists as columns, as long as the columns all have the same length. In particular, the following q statement, which groups but does not aggregate, is valid and produces a q table.

```
q) trade:([]time:();sym:();price:();size:())
q) `trade insert(09:30:00.000;`AIG;10.75;200)
q) `trade insert(09:30:01.000;`IBM;10.55;500)
q) `trade insert(09:30:02.000;`IBM;10.75;100)
q) `trade insert(09:30:03.000;`AIG;10.75;400)
q) `trade insert(09:30:04.000;`MSFT;10.45;100)
q) `trade insert(09:30:05.000;`IBM;10.35;100)
q) `trade insert(09:30:06.000;`MSFT;10.75;400)
q) nestedtrade:select price,size by sym from trade
```

```
 nestedtrade
 sym|  price                  size
--------------------------------------------------
AIG    10.75 10.75            200 400
IBM   10.55 10.75 10.35   500 100 100
MSFT   10.45 10.75          100 400
```

You can see that the items of the price and size columns are simple lists, not atoms. For example, compare the row of nestedtrade for the sym `AIG to the result of the select expression that follows.

```
q) flipnested: (value flip value nestedtrade)[;1]
q) flipnested
(10.55 10.75 10.35;500 100 100)
q) newnested:select price,size from trade where sym=`IBM
```

```
newnested
price  size
---------------
10.55  500
10.75  100
10.35  100
```

You can see that the price column in this result is identical to the price item in the row of table nestedtrade where sym is `IBM. Similarly, the size column in this result is identical to the size item in flipnested. The table nestedtrade is an example of grouping without aggregating, and we can say it is partitioned on nested by sym. Columns price and size of nestedtrade are called nested columns.

Now the question is, how are queries formulated for table nestedtrade? We'll show two representative examples. Here is the first.

```
 q) r1:select avg price by sym from trade where sym in `AIG`IBM
```

```
  r1
sym|  price
AIG   10.75
IBM   10.55
```

The equivalent query for nestedtrade is as follows.

```
q) r2:select sym,each[avg] price from nestedtrade where sym in `AIG`IBM
```

Without the Each modifier, the query would average the simple price list for the sym `AIG with the simple price list for the sym `IBM, resulting in another simple list if those two had the same count, or a length error if not. What we want, however, is to average the items in each list separately, which is what each[avg] does.

```
 r2
sym  price
AIG   10.75
IBM   10.55
```

Note that the only difference in the two results is that sym is a primary key of r1, but not r2.
Now that we have seen that the two methods of querying produce the same results (apart from the key) look at the execution time, we can see the execution time for a statement by preceding it with \t, this gives the execution time in milliseconds as in

```
q) \t do[10000;r1:select avg price by sym from trade where sym in `AIG`IBM]
1862
```

compared to

```
q) \t  nestedtrade:select price,size by sym from trade
0
q) \t do[10000;r2:select sym,each[avg] price from nestedtrade where sym in `AIG`IBM]
180
```

note that above we have looped 10000 times and used the same aggregation on the partitioned table each time, this is merely to illustrate the time differences. By avoiding having to regroup the trade table each time and instead storing this grouping, we can significantly reduce overall execution time.

The second example is an aggregation where part of the "where phrase" in the trade query applies to a partitioned column in flipnested.

```
q) r3:select avg price by sym from trade where sym in `AIG`IBM,size>200

  r3
sym             |   price
----------------|----------
AIG             |  10.75
IBM             |  10.55
```

The restriction size>200 cannot be done in the "where phrase" for table nestedtrade because the size column is not a simple list; it must therefore be done in the "select phrase". The other part of the "where phrase", which applies to the sym column, can be done as usual. Consequently the query will look like

```
  select  <exp> from nestedtrade where sym in `AIG`IBM
```

where the <exp> to be applied to items from price and size is

```
  avg price where size>200
```

That is, the expression must be applied to the price and size items for the sym `AIG and separately to those items for the sym `IBM. To do that we define

```
q) af:{[price;size]avg price where size>200}
```

and then

```
 q) r4:select sym,ap:price af'size from nestedtrade where sym in `AIG`IBM
```

Again the difference between r3 and r4 is that r3 is keyed by sym.
And comparing times as before

```
q) \t do[10000;r3:select avg price by sym from trade where sym in `AIG`IBM,size>200]
1892
q) \t do[10000;r4:select sym,ap:price af'size from nestedtrade where sym in `AIG`IBM]
220
```

Rollups

The expressions below show examples of rollups.

```
q) select last price by time.minute from trade  / 1 minute bars
q) select last price by 5 xbar time.minute from trade  / 5 minute bars
q) t:([]date:2000.01.01+til 9;size:til 9)  / 2000.01.01 is saturday
q) select last size by date.week from t    / monday week bars
```

Tools for complex calculations

Datatypes

The table below summarises the various types in kdb+ with some comments where appropriate and their equivalents in other languages. Individual numbers, character strings, binary vectors etc. A list is an ordered collection of atoms. It should be noted that positive type numbers (given below) are related only to lists, the type numbers for atoms are the respective negative numbers.

Name	Example	char	type	size	null	sql	java	.net	Xml
bool	1b	b	1	1			boolean	bool	boolean
The bool atoms are denoted 0b and 1b. A bool occupies 1 byte. Bools have no null.									
byte	0xff	x	4	1			byte	byte	byte
Bytes have no null.									
short	23h	h	5	2	0Nh	smallint	short	short	short
Shorts are used to functionally determine the types of objects and to cast objects from one type to another. Shorts (2 bytes) are written like ints, but with a trailing h.									
int	23	i	6	4	0N	int	int	int	int
Int (4 bytes) atoms are written in the usual way									
long	23j	j	7	8	0Nj	bigint	long	long	long
long atoms (8 bytes) are written like ints but also have a trailing j.									
real	2.3e	e	8	4	0Ne	real	float	float	single
Real atoms (4 bytes) are written like floats or ints but with a trailing e.									
float	2.3	f	9	8	0n	float	double	double	double
float atoms (8 bytes) can be written in the usual decimal and exponential formats.									
char	“a”	c	10	1	“”	char	char	char	
varchar	`ab	s	11	*	`	varchar	String	string	string
Varchars are character strings preceded by back-quote, as in `abc. (Not all syms can be formed simply by putting back-quote in front a sequence of characters.) Syms are also called symbols and varchars.There must be no spaces in sym lists, e.g. `a`b`c . Symbols can have any non-zero characters (e.g. `$"a_b") but identifiers must be alphanumeric. Varchars are interned in a hash table.									
month	2003.03m	m	13	4	0Nm				
date	2003.03.23	d	14	4	0Nd		date	Date	date
Date atoms must be specified in the form yyyy.mm.dd. For example, 2003.03.23 for March 23 2003 is correct whereas 2003.3.23 is not. Date units are days.									
datetime	2003.03.23T15:32:37.271	z	15	8	0Nz	timestamp	timestamp	DateTime	dateTime
Datetime units are days.u									
minute	08:31	u	17	4	0Nu				
second	08:31:53	v	18	4	0Nv				
time	09:10:35.000	t	19	4	0Nt				
Time values have millisecond resolution and are specified in the form hh:mm:ss.uuu. For example 09:10:35.021									
enum	`s$`b,where s:`a`b	*	20	4	`s$..				
Enumerations are foreign keys. They are like atomic types, in that individual enumerated values can be atoms or simple lists, atoms have negative datatype values, and simple lists have positive values.									

Note that each type can be specified in a list; as in the following nested list;

q) (01b;0x00ff;0 1h;0 1;0 1j;0 1e;0 1.0;"ab";`a`b;2000.01 2000.02m;2000.01.01 2000.01.02)

which when typed into the q-console, gives output identical to the input.

In particular, the list (1 2 3f) is equivalent to (1.0 2.0 3.0)

Assignment

Construct	Details and Example
Assignment	Assignment in q is denoted by colon.

For example,

```
  w:3.141
```

The value of a data object can be displayed simply by typing its name, as follows.

```
  w
3.141
```

Note that there is no result display following the specification of w, even though specification has a result (the value given to the name). This is simply the way the q console is designed.

Lists

A list is an ordered collection of atoms or other types including lists. Lists are denoted by atoms separated by semi-colons all within parentheses. As mentioned in the previous section, an atom of whatever type has a negative value, say –n, and a simple list of those atoms has positive type value n. For example, `abc is a varchar atom with datatype value -11h , and the simple varchar list `abc`s has datatype value 11h . Lists do not have to be homogeneous (i.e., all items have the same data types). Heterogeneous lists are permitted such as (23;`abc;27). Heterogeneous lists are always of type 0h.

Lists of lists can be created as can lists of lists of lists etc. The lists, in a list of lists, are separated by semi-colons and the result is surrounded by parentheses. The individual atoms in the list are called the atoms of the list.

Lists:((`abc,`cfe;1,2,3);(`des;1,2))

Construct	Details and Example
One Item Lists and Enlist	The various forms of list notation in kdb+ do not provide a way to create a one-item list. The monadic primitive function enlist enables one-item lists to be created.

For example,

```
enlist 34.5
```

is a one-item simple float list whose item is 34.5 If you enter this expression in a kdb+ console you will seen the following.

```
  enlist 34.5
,34.5
```

As you can see, a one-item list is displayed with a leading comma. In general, enlist applies to all data objects; the result for any data object is a one-item list whose item is that object. For example,

```
  enlist(2 5;3.5 10 12)
,(2 5;3.5 10 12)
```

Extract from Denis Shasha Primer: Atom and List Formation

```
/ Note that comments begin with a slash "/" and cause the parser
/ to ignore everything up to the end of the line.
/ (The / operator is overloaded
/ however and has a different meaning if it follows a two argument operator.
/ We will see how.)
x: `abc / x is the symbol `abc (a symbol is represented internally as a number).
y: (`aaa; `bbbdef; `c) / a list of three symbols
y1: `aaa`bbbdef`c / another way to represent this list (no blanks between symbols)
y2: (`$"symbols may have interior blanks";`really;`$"truly!")

y[0] / returns `aaa
y 0 / juxtaposition eliminates the need for brackets
    / This also returns `aaa
y 0 2 / returns `aaa `c as does y[0 2]
z: (`abc; 10 20 30; (`a; `b); 50 60 61) / lists can be complex
z 2 0 / returns (`a`b;`abc) because z[2] is `a`b and z[0] is `abc
z[2;1] / returns `b
       / The second element of z[2]
z[2;0] / returns `a
       / The first element of z[2]
x: "palo alto" / a list of characters
x 2 3 / gives "lo"
```

Dictionaries and Associations

An easy way to conceptualise dictionaries is to view them as maps of lists to lists. The concept of dictionaries can be further understood by recalling from elementary algebra that a mathematical function has a domain and a range, where each element in the domain has a single corresponding element in the range. So, f(x) = x*x is a function because for every domain element (i.e., x value) there is a single f(x) value. Lists can be viewed as functions where the domain is a set of locations 0, 1, 2, ... and the range of a location is the list element at that location.

Dictionaries are a kind of function (as are keyed tables) and very much like lists except that the domain may be any set of unique elements rather than indexes. Dictionaries are created with the primitive dyadic function denoted by ! and called Xkey – in other words create a dictionary by writing the domain then an exclamation point and then the range.

Extract from Denis Shasha Primer: Dictionary Formation and Access

```
fruitcolor: `cherry`plum`tomato!`brightred`violet`brightred
fruitcolor `plum / gives `violet
stringfruit: ("cherry"; "plum"; "tomato") !`brightred`violet`brightred
/ so the domain can be a list of lists
stringfruit "tomato" / returns `brightred

/ Dictionaries are combined using "upsert" semantics.
/ An upsert will do an update if the keys match and otherwise
/ will do an insert.
fruitcolor2:`grannysmith`plum`prune!`green`reddish`black
fruitcolor,fruitcolor2
`cherry`plum`tomato`grannysmith`prune!`brightred`reddish`brightred`green`black
/ You notice that plum has simply had its color updated, but
/ the other entries from fruitcolor2 are new.
```

Tables viewed as dictionaries

This is a very useful relationship as it facilitates, for example, complex calculations and manipulations to be performed on uploaded data before creating a table. A dictionary can be viewed as having a domain column and a range column. Pictorially, the domain is to the left of the range. For example from above

```
`cherry `brightred
 `plum   `violet
`tomato `brightred
```

Because the range need not be a singleton, we could imagine a dictionary having list elements in the range, e.g.,

```
`name    `tom`dick`harry
`salary  30 30 35
```

The domain is `name and `salary. The list `tom`dick`harry is the range element corresponding to `name. The list 30 30 35 is the range element corresponding to `salary
A table is just a list of similar dictionaries but is stored as - and sometimes created from - a flipped dictionary of lists. Since the dictionaries all have the same keys the keys are just shown once at the top:

```
`name     `salary
`tom      30
`dick     30
`harry    35
```

Dict and Flip

As stated earlier dictionaries are maps of lists to lists. A flip is a list of dicts (records).

```
d:`x`y!(`a;2)      / a dict is a map from a list to a list
```

```
f:(d;`x`y!(`b;3))  / a flip is a list of dictionaries
([]x:`a`b;y:2 3)   / a column notation for the same flip
```

Flip is a monadic primitive that applies to lists and associations. The effect is to interchange the top two levels of it argument. That is, if c:flip b then b[i][j] and c[j][i] are identical for all valid indices i and j. In particular, all items of b must have the same count in order that b can be flipped. For example,

```
  b:((1;2 3);(4 5 6;7);(8;9 10 11 12))
```

Each item of b has 2 items, and therefore can be flipped.

```
  c:flip b
  c
((1;4 5 6;8);(2 3;7;9 10 11 12))
  b[1][0]
4 5 6
  c[0][1]
4 5 6
```

The list c has 2 items because each item of b has 2 items; each item of c has 3 items because b has three items.

Associations, Xkey, Key and Value

Associations are the only primitive datatype that has no syntactic form; associations are created with the primitive dyadic function denoted by ! and called Xkey. Associations are associative lists; the items of the left argument to Xkey are the indices, or keys, and the items of the right argument are the values. The left and right arguments must have the same length. The arguments can be any lists, although duplicate items in left argument lists are ineffective. **An association whose left argument is a simple varchar list is called a dictionary**.

Note that, as with other datatypes, the console display of an association is an executable form from which the data can be recreated.

Construct	Details and Example
Associations	Associations are associative lists

For example, the following associations relate type values to type names.

```
  n2v:"csif"!10 11 6 9h  / type values associated with type names
  n2v["i"]             / select an item
6h
  n2v["sf"]             / select multiple items
11 9h
  v2n:10 11 6 9h!"csif"  / type names associated with type values
  v2n[9 6h]
"fi"
```

An example in Processing Nulls uses the following expression to replace a leading null with another value. The example assumes that v is a simple float list.

```
if[null first v;v[0]:0.0]
```

Associations provide an easy way to extend this expression to handle more than one datatype. For example, the following association provides initial values for simple lists of type char, varchar, int and float.

```
  iv:10 11 6 9h!("f";`first;0;0.0)
```

The replacement expression then becomes

```
if[null first v;v[0]:iv type v]
```

which applies to simple lists of those four types. For example,

```
  v:``a`c`h
  if[null first v;v[0]:iv type v]
  v
`first`a`c`h
```

This association is easily extended to apply to all lists.

Construct	Details and Example
Key function	The Key function gives a list of the keys and the Value function gives a list of the values, in the same order as the keys.

```
 key n2v
"csif"
  value n2v
10 11 6 9
```

The following association has a duplicate key (3).

```
  d:1 2 3 4 3 5!`a`b`c`d`e`f
```

The first value item keyed by 3 (`c) is accessible by key indexing; the other one (`e) is not.

```
  d[3]
`c
```

The result of selecting the value of an association for a non-existent key is the null of the value list. For example,

```
  n2v["X"]
0N
  v2n[0h]
" "
  d[`abc]
()
```

The Key function also applies to a positive integer n, giving the simple list of all ints from 0 to n-1, in order. In particular, if n is the count of a list then key n contains all indices of that list. For example,

```
  key count (1;2.00 3.50 7.00;`a;"xyz")
0 1 2 3
```

Indexing at depth

The following example illustrates indexing at depth in associations.

```
  d:`a`0x`dc`ch!(1; 2 3.5 7;`a;"xyz")  / d is a dictionary
  d[`a]                              / select individual values
1
  d[`0x]
2 3.5 7                            / this value item is a list
  d[`0x][1]                        / select an item of this value item
3.5
  d[`0x;1]                         / another way to do it
3.5
  d[`0x`ch`a`ch]                     / select a list of value items
(2 3.5 7;"xyz";1;"xyz")
  d[`0x`ch;1 0 0]                    / select items from each value in a list
(3.5 2 2;"yxx")
```

Primitive functions and associations

Here are examples of primitive functions applied to associations.

```
 count d                   / the number of items of d
4
  value d                  / the value list of d
(1;2.00 3.50 7.00;`a;"xyz")
  key d                   / the key list of d
`a`0x`dc`ch
  string d                 / string applies to all the values
`a`0x`dc`ch!(,"1";("2.00";"3.50";"7.00");,"a";(,"x";,"y";,"z"))
```

Append

Unlike non-associative lists, new items can be appended to an association with indexing.

For example, the byte datatype can be added to the n2v and v2n associations as follows.

```
  v2n[4]:"x"
  n2v["x"]:4h
  v2n
10 11 6 9 4!"csifx"
  n2v
"csifx"!10 11 6 9 4
```

Verbs and Adverbs

Syntactically, kdb+ has nouns, verbs and adverbs.

- All data objects are nouns, as are all functions.
- Verbs are primitive symbols and names that can be evaluated with infix notation, as in a+b. If Plus were a dyadic function and not a verb, it would have to be evaluated as +[a;b] . It turns out that Plus can be evaluated in this way because, operationally, it's a function. Syntactically, whenever Plus appears in an infix expression, it is a verb. Otherwise, syntactically, it is a noun.
- Verbs, juxtaposition for monadic functions and indexing, and function projections enhance readability by reducing the number of square brackets and parentheses in expressions.
- Adverbs modify dyadic functions and verbs to produce new, related verbs. The functions produced by adverbs are called derived functions or derived verbs, depending on context.

- Kdb+, like any vector language needs adverbs because one doesn't always want list op list to mean element-by-element application of op (for atomic functions) or whole list op whole list (for non-atomic ones).

Construct	Details and Example
Each	The adverb Each, denoted by quote ('), modifies dyadic functions and verbs to apply to the items of lists instead of the lists themselves.

For example,

```
  1 2 3,4 5                 / join
1 2 3 4 5
  (1 2 3;"abcd"),'(4 5;"e")  / Join-Each (,')
(1 2 3 4 5;"abcde")
```

The arguments of a dyadic function derived from Each must be lists of the same length, or either argument can be an atom. Actually, both arguments can be atoms, but then each has no effect. For example,

```
  ("one";"two";"three"),'","  / append comma to each item
("one,";"two,";"three,")
```

It's common to need to join a list, not an atom, to every item in another list. For example, suppose, in the last example, instead of appending comma to each item, you wanted to append comma-blank, as in ", ". To do that with Each, a 3-item list of comma-blanks must first be created, as follows.

```
  v:("one";"two";"three")    / give the list a name for convenience
  v,'(count v)#enlist ", "
("one, ";"two, ";"three, ")
```

There is a form of Each for monadic functions that uses the keyword "each". For example,

```
  reverse (1 2 3;"abc")       / Reverse
("abc";1 2 3)
  each[reverse](1 2 3;"abc")  / Reverse-Each
(3 2 1;"cba")
```

Each-Left and Each-Right	There are two variants of Each for dyadic functions called Each-Left and Each-Right that simplify cases like this.

Here is Each-Left (\:) used to append comma-blank to each item of v.

```
  v,\:", "
("one, ";"two, ";"three, ")
```

Each-Right is analogous.

Extract from Denis Shasha Primer: Adverbs

```
x: 10 30 20 40
y: 13 34 25 46
x,y / returns 10 30 20 40 13 34 25 46
x,'y / (each) returns (10 13;30 34;20 25;40 46) that is, a list of pairs
x,\: y / (each left) returns a list of each element from x with all of y.
/ (10 13 34 25 46;30 13 34 25 46;20 13 34 25 46;40 13 34 25 46)
x,/: y / (each right) returns a list of all the x with each element of y
/ (10 30 20 40 13;10 30 20 40 34;10 30 20 40 25;10 30 20 40 46)
x: 1 _ x / drops the first element
x / returns 30 20 40
y: -2 _ y / drops the last two elements
y / returns 13 34
/ Combine each left and each right to be a cross-product (cartesian product)
x,/:\:y / returns ((30 13;30 34);(20 13;20 34);(40 13;40 34))
/ So a cross-product combines each element from x with each from y
/ Because the above format may not be convenient there is a special
/ unary operator that undoes a level of nesting: "raze"
raze x,/:\:y / returns (30 13;30 34;20 13;20 34;40 13;40 34)
/ Sometimes a function is meant to be applied to each element of a list.
/ That is, it is unary with respect to each element of the list.
reverse (1 2 3 4;"abc")      / reverses the two elements of this list:
/ returns ("abc";1 2 3 4)
each[reverse](1 2 3 4;"abc") / reverses each list within this pair
/ returns (4 3 2 1;"cba")
reverse each (1 2 3 4;"abc") / returns the same (4 3 2 1;"cba")
/ Here is an example that shows how you can use your function with this.
/ Suppose we want to compute a random selection of the elements
rockroll: `i`love`rockandroll
/ for each element of a numerical list, e.g.
numlist: 4 7 3 2
/ We define a function
myrand:{[n] n ? rockroll}
/ Then we can apply myrand to each element of numlist.
myrand each numlist
/ one output: (`rockandroll`rockandroll`love`rockandroll;`i`i`love`love`i ...
/ The point is that it creates a list for each element in numlist.
/ The fact that this function performs on each element is why this is
/ called each.
```

Manipulating Atoms, Lists, Dictionaries and Verbs

Kdb+ provides a variety of tools for working with atoms, lists, dictionaries and verbs. As all the data types coexist in the same process, there is no overhead in going from one to the other. The table below outlines a variety of ways to transform, add to, join and access data.

Construct	Details and Example
Index	Index has the syntactic form []. Indices start at 0. That is, the index of the first item of a list is 0.

```
  (`8n3;2245;-3.5)[0]
`8n3
  A:(`8n3;2245;-3.5)   / give the list a name for convenience
  A[1]
2245
  A[2]
-3.5
```

Lists of indices produce lists of selected items.

```
  A[0 2 1 2 0 0]
(`8n3;-3.5;2245;-3.5;`8n3;`8n3)
```

The index 0 is located at the first, fourth and fifth positions of the index list 0 2 1 2 0 0 . The 0-th item of the list A, which is `8n3, is located at the first, fourth and fifth positions of result. Similarly, the index 1 is located at the third position of the index list and A[1], or 2245, is located at the third position of the result. Likewise for the index 2.We say that the result is congruent to the index. In general, the index and result are conforming lists in the sense of atom functions. That is, the structures of the two match up in certain ways. Congruency means that equal values in the index correspond to equal values in the result (but not necessarily vice versa).

Congruency applies to any index list, not just to simple lists. For example,

```
B:`x`y`z
  B[(0 1 2;2 2;0 0 1)]
(`x`y`z;`z`z;`x`x`y)
```

The index list (0 1 2;2 2;0 0 1) and the result (`x`y`z;`z`z;`x`x`y) are congruent, as before.

Index Assignment	Index assignment replaces specified items of a list with new values.

```
  A                / show A's value (assigned above)
(`8n3;2245;-3.5)
  A[0]:("asdq";2 3 4)  / an item can be replaced with any data value
```

Assignment results are not displayed; the cursor (blinking _) is waiting for the next input.

```
  A                   / show A's new value
(("asdq";2 3 4);2245;-3.5)
  A[2 0]:(`a`b`c;173.45)    / multiple items can be replaced in any order
  A
(173.45;2245;`a`b`c)
```

Index assignment into simple lists is strict with regard to the type of the new value. The replacement atoms must have the same datatype as those that are replaced. For example,

```
  v:`x`y`z
  v[2 0]:`a`b
  v
`b`y`a
  w:10 2.5 0 -8.34
  w[2]:5           / replace an item of a simple float list with an int
```

A type error is reported in the console

```
  w[2]:5.0     / replacing an item with another float atom is permitted
  w
10 2.5 5 -8.3
```

Duplicate indices are permitted. For example,

```
  s:3 -2 5 -6
  s[0 1 0 2]:10 20 30 40
  s
30 20 40 -6
```

The replacement rule for a simple list of indices (0 1 0 2 in this case) is that the indices and corresponding values are used in index order. That is, the following individual steps give the same result as above.

```
  s:3 -2 5 -6
  v:10 20 30 40
  i:0 1 0 2
  s[i 0]:v[0]
  s[i 1]:v[1]
  s[i 2]:v[2]
  s[i 3]:v[3]
  s
30 20 40 -6
```

Finally, you can replace multiple items with a single atomic value.

```
  x:1010101010b
  x[1 3 5 7 9]:1b
  x
1111111111b
```

The result of index assignment is the value to the right of the colon, not the new value of the identifier. For example,

```
  b:8 -3 5
  10+b[2]:7
17
```

If the result of the assignment was the new value of b, which is 8 -3 7 , then the result of 10+b[2]:7 would be 18 7 17 .

Amend

Items of a list can be modified with a combination of indexing and index assignment

Items of a list can be modified with a combination of indexing and index assignment. For example,

```
  s:3 20 -4 10
  s[0 2]:s[0 2]+100
  s
103 20 96 10
```

This can be expressed more concisely, and often executed more efficiently, as follows.

```
  s:3 20 -4 10
  s[0 2]+:100
  s
103 20 96 10
```

The value of the right does not have to be an atom. The rule is that the value on the right must conform to the index of the left, in the way of atom functions.

```
  s:3 20 -4 10
  s[0 2]+:100 1000
  s
103 20 996 10
```

As before, modifying simple lists is strict; the new items must be the same type as the old. s:3 20 -4 10

```
s[0 2]+:100h  / causes a type error
```

Any dyadic primitive atom function can be used in place of +.

Indexing at Depth

Items of lists of lists can be accessed using indexing at depth

The following list has several levels. Example results may be displayed on multiple lines to aid comparisons with the indices that created them.

```
 d:((1 2 3;4 5 6 7);(8 9;10;11 12);(13 14;15 16 17 18;19 20))
```

You can select a top-level item, as in

```
  d[1]           / select item 1
(8 9;10;11 12)
```

You can select an item of an item and an item of an item of an item.

```
  d[1;2]    / select item 2 of item 1
11 12
  d[1;2;0]  / select item 0 of item 2 of item 1
11
```

If the indices are lists then the result is a cross-section of d.

```
  d[2 0;0 1]
((13 14        / item 0 of item 2
  15 16 17 18)  / item 1 of item 2
 (1 2 3        / item 0 of item 0
  4 5 6 7))     / item 1 of item 0
```

Note that the selected items are arranged in the result in the same order as the atoms in (2 0;0 1). There is no limit on the structure of the indices.

Elided Indices

An example of an elided index is the item index expression d[].

There are more interesting situations when indices in an index at depth expression are elided. For example, d[;0] denotes the 0th item of each item of d.

```
  d[;0]
(1 2 3   / item 0 of item 0
 8 9     / item 0 of item 1
 13 14)  / item 0 of item 2
```

The situation is more complicated when an index other than the top-level index is elided. For example, in d[0 2;;1 0] , the elided index corresponds to 0 1 when indexing d[0], but to 0 1 2 when indexing d[1].

```
  d[0 2;;1 0]
((2 1          / items 1 and 0 of item 0 of item 0
  5 4)         / items 1 and 0 of item 1 of item 0
 (14 13        / items 1 and 0 of item 0 of item 2
  16 15        / items 1 and 0 of item 1 of item 2
  20 19))      / items 1 and 0 of item 2 of item 2
```

Functional forms of Amend

There are functional forms of Amend based on @ and Dot.

The @ form, @[d;i;f;y] is analogous to d[i]f:y and @[d;i;:;y] is analogous to d[i]:y . Repeating an example from above,

```
  s:3 20 -4 10
  @[s;0 2;:;5 6]
5 20 6 10
  @[s;0 2;+;100]
103 20 96 10
```

Unlike the bracket form of Amend, s is not modified; the result is a copy of s with the specified modifications. This form of Amend is more like select and update expressions. And like select and update, the modifications are applied directly to s if the first argument is `s.

```
  @[`s;0 2;+;100]
`s
  s
103 20 96 10
```

The functional form is also more general than the bracket form because the function f can be any dyadic function, not only primitive atom functions.

The functional form using dot is .[d;i;f;y] . The difference is that a list i represents indexing at depth. For example,

```
  s:(1 3 -5;"xyzw")
  s[0;2]+:3         / the bracket form
  s
(1 3 -2;"xyzw")
  s:(1 3 -5;"xyzw")  / initialize s again
  .[s;0 2;+;3]       / Dot Amend
(1 3 -2;"xyzw")
  .[`s;0 2;+;3]      / update s
`s
  s
(1 3 -2;"xyzw")
```

Type

The Type function is a monadic function that gives the numeric datatype value of its argument

```
bghi..             / letters
1 4 5 6h..         / shorts
`bool`byte`short..  / names
`sym`s`sp..        / user enum types
```

For example,

```
  type 100
-6h                / an int atom
  type 0.4 -2 10.76e
8h                 / a simple real list
  type(`a;2h;3 4)
0h                 / a list of mixed types
```

Note that values of the Type function are shorts. These examples illustrate a general rule. For datatypes that come as both atoms and simple lists, the type value of an atom is the negative of the type value for a simple list.

Cast

Cast (a$b) is a dyadic primitive that converts the atom b to the type specified by the atom a.

```
"I"$"23"   / use capital letter for data from string
"i"$23.4   / use small letter for data from data
`hh$12:34  / or use type names (incl. datetime fragments)
`sym$`a  / check referential integrity
`sym?`a  / append if necessary
```

Consider, for example, the case in which the left argument a is one of the values in the datatype name column of the table in the datatypes summary. For example,

```
  "x"$97                  / cast the int 97 using the datatype name x for a byte
0x61
  "x"$"a"                / cast a char to a byte
0x61
  "c"$0x41                / cast a byte to a char
"A"
  "d"$2003.03.23T08:31:53  / extract the date from a datetime value
2003.03.23
  "t"$2003.03.23T08:31:53  / extract the time from a datetime value
08:31:53.000
```

An int d, representing a day count, can be added to a date to give the date d days ahead or -d days previous. The concept of a day count is also associated with a date. For example, casting a date as an int gives the day count for that date; casting an int as a date gives the date for that day count.

```
  "i"$2003.03.23
1177
  "d"$1177
2003.03.23
  "d"$0
2000.01.01
```

Cast is an atomic function. For example,

```
  "x"$(0 100 200;255)
 (0x0064c8;0xff)
```

The positive value of the Type Value column of the table in the datatypes summary can also be used as a left argument to Cast. For example,

```
  6h$1 -4.2 3.78
1 -5 3
  8h$1b
1e
```

Creating varchars

This is a very important operation, used, for example, for importing text (varchar) data into tables.

Casting simple char lists to varchars There is a special case of Cast that does it, where the left argument is the empty sym `. For example,

```
  `$"varchar"
`varchar
  `$("varchar0";"varchar1";"etc")
`varchar0`varchar1`etc
```

This form of Cast applies to right argument lists differently than the forms in the preceding section. Those forms, with left argument "x" or "d" or 6, apply independently to all atoms in the right argument, giving an atom result for each one. However, when the left argument is `, Cast applies independently to all simple char lists in the right argument, giving an atom for each one.

Note that, as these examples show, all varchars are displayed in the console with back-quote followed by the varchar's contents.

Casting with Computed Types

Casting with datatype values is particularly useful when associated with computed type values.

Casting with datatype values is particularly useful when associated with computed type values. For example, replacing an item in a simple list is strict, in that the datatype of the replacement value (short, float, etc.) must be the same as the simple list. There may be situations in which more latitude can be allowed. For example, if a simple list has type int, a replacement of type bool, byte, short or int may be acceptable. A general expression that accepts all such values is c[i]:(type c)$d , which casts the replacement value d to the datatype of c. For example,

```
  c:10 345 -20 11
  c[2]:(type c)$0xab
  c[0]:(type c)$23h
  c
23 345 171 11
```

Extracting Data from Text

kdb+ provides primitive functions for extracting data from text files, but sometimes you have to do it yourself.

A text file is read into the workspace as a list of simple char lists. The simple char lists are processed by partitioning them into smaller lists containing the text of individual values. If the values can be extracted with a special form of casting. For example,

```
  "S"$"abc012"       / extract a symbol
`abc012
  type "S"$"abc012"  / check the type of the result
-11h                 / a varchar atom
```

Note in this example that the left argument of Cast is upper case "S" instead of the lower case type name "s". Other conversions can also be done this way.

```
  "I"$"271828"
271828
  "D"$"2003.03.23"
2003.03.23
```

Casting with "D" will also extract dates in other standard formats.

```
  "D"$"2003-03-23"
2003.03.23
  "D"$"03/23/2003"
2003.03.23
  "D"$"03/23/03"
2003.03.23
```

Creating Text from Data

The primitive function named String produces simple char lists from atomic data.

For example,

```
  string 345
"345"
  string`xyz
"xyz"
  string 23h
"23"
  string 4294967296j
"4294967296"
```

This function is used to produce char strings for display or export to text files. Consequently, the results contain none of the special notation that distinguishes types in q. For instance, in the above examples the resulting string for the short 23h drops the type name "h". So does the result for long value 4294967296j. You must use Cast with the appropriate left argument to retrieve the q values. For example,

```
  string 4294967296j
"4294967296"
  "J"$"4294967296"
4294967296j
```

Like an atom function, the string function applies independently to every atom in a list argument. However, unlike an atom function, the result when applied to an atom is a simple char list, not an atom. For example,

```
  string(5;23h;`a`uvw`xz)
(,"5";"23";(,"a";"uvw";"xz"))
```

Note that the result of string 5 and string`a are the one-item simple lists ,5 and ,"a" , respectively.

Join or Concatenation

The join of two data objects a and b is denoted by a,b.

Any two atoms or lists can be joined (concatenated). For example,

```
  1 2 3,4
1 2 3 4
  1 2,3 4
1 2 3 4
  (0;1 2.5;01b),(`a;"abc")
(0;1.00 2.50;01b;`a;"abc")
```

Join provides another important example of casting with computed types. When two simple lists are joined, or an atom and a simple list, the result is a general list unless the types of the two operands are the same. For example,

```
  1 2.4 5,-7.9 10
1.00 2.40 5.00 -7.90 10.00      / both operands are simple float lists
  1 2.4 5,-7.9
1.00 2.40 5.00 -7.90            / one is a simple float list, the other a float atom
  1 2.4 5, -7.9 10e
(1.00;2.40;5.00;-7.90e;10.00e)  / one is float, the other real.
```

As when replacing items in a simple list, there may be situations in which more latitude is required. For example, if the left argument is a simple short list then a right argument of type int, bool or byte may also be acceptable.

```
  v:1 2.34 -567.1 20e
  v,(type v)$789               / cast an int to a real
1 2.34 -567.1 20 789e
  v,(type v)$1b                / cast a bool to a real
1 2.34 -567.1 201e
  v,(type v)$0xab
1 2.34 -567.1 20 171e
```

Tables can be joined. The result has all the columns of the two tables. Columns with same names are joined as lists. For example,

```
  t:([]a:1 2 3;b:`a`b`c)
  s:([]b:`d`e`f;a:10 11 12)
  t uj s
a     b
-
1     a
2     b
3     c
10    d
11    e
12    f
```

The order of the columns in the two tables doesn't matter; the order of the result is the order of the left argument. If the datatypes of corresponding columns don't match then the resulting columns are general lists. If column names don't match then the both sets of columns are in the result. For example,

```
r:([]c:10 20 30;d:1.2 3.4 5.6)
 t uj r
a    b  c   d
------------------
1    a
2    b
3    c
          10   1.2
          20   3.4
          30   5.6
```

Since the columns of a table must all have the same length, these two cases cannot be mixed. That is, the operands cannot have some columns with matching names and some that do not match.

Join for key tables is strict; both the key and data columns must match in names and datatypes.

Enumeration	The process of enumeration is a form of cast. The general form is`d$v where d is the name of an existing data object.

There are two possibilities for d, either a key table or a simple list of unique atoms. The object v is either an atom whose value is a key value or item of d, respectively, or a list of those atoms. We say that the projection `d$ is an enumeration, d is an enumeration domain, and `d$v is an enumerated value.

If `d names a key table then the enumeration defines a foreign key of that table, whether or not the enumeration is a column in another table. For example, using the table s from the product distribution script sp.q,

```
  v:`s$`s2`s4`s3`s1`s5
  v.name                    / dot notation works
`jones`clark`blake`smith`adams
```

Individual enumerations are distinguished datatypes; enumerations with different domains have different datatype values. Datatype values are assigned in the order in which the enumerations are defined, starting at +-20. (As for the primitive atomic types, an enumerated atom has a negative datatype value and an enumerated simple list has a positive datatype value. For example,

```
  s:`a`b`c
  t:1 2 3 4
  e:`s$        / enumeration of s is defined first
  f:`t$
  type e `b     / e has type +-20
-20h
  type e `b`a
20h
  type f 1      / f has type +-21
-21h
  type f 1 3 1
21h
```

If another enumeration is subsequently defined it will have datatype value +-22, and so on.

Enumerating simple varchar lists with large item counts and relatively few distinct items is an important optimization technique. Kdb+ data is self-describing. For example, a simple int list is stored as a few bytes of descriptive information followed by the int atoms in a contiguous array. When saved to disk this list has essentially the same format. When read from disk it is simply mapped into memory, a nearly instantaneous operation.

Varchars are different because their contents are text, which means that their storage requirements vary. When a simple varchar list is saved to disk, the contents of its items are written end-to-end. When read from disk the individual varchar atoms are recreated. This is not a problem when files are seldom read, say only when an application is initialized. But select expressions can be applied directly to stored tables, which can cause performance problems due to repeated creation of the varchar atoms in varchar columns.

The performance problem is overcome by enumerating varchar columns; a column c would be replaced by the enumeration

```
  `u$c
```

where u is a list of the unique items in c, e.g. distinct c . For example,

```
  c:`c`b`c`c`a`b`a`a`c`c`c`b`c`b`a
  u:distinct c                   / the distinct items of c
  e:`u$c                         / e is the enumeration of c on u
  e                            / view the console display
`u$`c`b`c`c`a`b`a`a`c`c`c`b`c`b`a
  k:0 1 0 0 2 1 2 2 0 0 0 1 0 1 2   / k is the list of indices
  c~u[k]                         / c is identical to u[k]
1
  e[9 3 11]                       / select from e; e is a list because c is a list
`u$`c`c`b                         / the result is also an enumeration
  u[k[9 3 11]]
`c`c`b
  c[9 3 11]
`c`c`b
```

Enumerated values can be used in the same ways as non-enumerated ones. For example, compare the following results for c and e.

```
  c=`a
000010110000001b
  e=`a
000010110000001b
  c in `a`b
010011110001011b
  e in `a`b
010011110001011b
```

Internally, an enumerated value is a pair of objects, the name of the domain, say `d, and an int list k of indices into d; the enumerated value is identical to d[k]. When an enumerated value is written to a file, the name of the domain and the index list k are actually written. For simple varchar lists, k is a simple int list. As a result, reading a file of an enumerated simple varchar list is nearly instantaneous. Note the domain d, which is not written to disk with the enumerated value, must be present in the workspace when the file is read.

Find

The function named Find is a dyadic function whose left argument is a list and right argument is any data object.

The result is the lowest index for which the right argument matches (using the Match function) an item of the left argument (the so-called "first occurrence"). If there is no match the result is the count of the left argument. For example,

```
  w:10 -8 3 5 -1 2 3
  w?-8
1
  w[1]
-8
 w?3                 / the first occurrence of 3
2
  w?17               / not found
7
  w[7]
0N
  "abcde"?"d"
3
```

Find is type-specific relative to the left argument. In the case of a simple list left argument, the right argument can also be a list whose atoms are all the same type as the left argument. The result corresponds to the left argument item-by-item. For example,

```
  rt:(10 5 -1;-8;3 17)
  i:w?rt
  i
(0 3 4;1;2 7)
  w[i]
(10 5 -1;-8;3 0N)
  r
(10 5 -1;-8;3 17)
```

If the left argument is a list of lists and the right argument is simple list, then items of the left argument are matched with the entire right argument. For example,

```
  u:("abcde";10 2 -6;(2 3;`ab))
  u?10 2 -6
1
  u?"abcde"
0
```

However, if the right argument is a general list then items of the left argument are matched with items of the right argument. For example,

```
  u?(2 3;`ab)
3 3
```

In this case Find matches items of the left argument with 2 3 and `ab, not (2 3;`ab) . In order to find (2 3;`ab), one has to explicitly check each component of u using the match function, that is

```
  where u~\:(2 3 ; `ab)
,2
```

If the left argument is a table then the right one must be a compatible record (dictionary or list) or table. That is, each column of the left argument, paired with the corresponding item of the right argument, must be valid arguments of Find. For example,

```
  sp?(`s1;`p4;200)
3
  sp?`s`p`qty!(`s2;`p5;450)
12
```

Reverse

Reverse applies to lists and gives their items in the opposite order

For example,

```
  reverse `a`b`c
`c`b`a
```

Reverse comes into play when it is more efficient to work with a list in its reverse order. For example, suppose that an application calls for many joins to the front of a long list l, as in

```
l:b,l
```

This expression causes a list of size (count b)+count l to be created and both b and l to be copied into it. That is, the list l must be copied for every join. However, when the joins go to the end of l, as in

```
l:l,b
```

then it is possible to do it so that (usually) only b is copied onto the end of a. This is due to the fact that the chunk of memory allocated for a list usually has unused space at the end. To specify this behavior use the form of amend that modifies l in place. The computation would proceed something like this:

```
  l:reverse l
  .[`l;();,;reverse b]
  ...
  .[`l;();,;reverse b]
  l:reverse l
```

Associative Arithmetic

Arithmetic is defined for associations with numeric value lists

For example,

```
  d:10 20 30!25 38.5 17
  e:10 30 45!11.5 24 -18
  d+e
10 20 30 45!36.50 38.50 41.00 -18.00
  d*e
10 20 30 45!287.50 38.50 408.00 -18.00
```

You can see that the keys of the sum are the union of the keys of the arguments. Also, if a key is in both arguments then the value for that key in the result is the sum of the values in the arguments. Otherwise, the value in the sum for that key is the value for that key in whichever argument it appears.

The other primitive atom functions apply in the same way. For example,

```
  d
```

Some applications such as sparse arrays may prefer the non-boolean items of the last result to be 0b. One way to do this is as follows.

```
  u:(key d)union key e
  i:(key d)inter key e
  v: u!u=-1
  v[i]:”b”$d[i]
```

The result of the expression u=-1 is 0000b; the assumption is that all keys of sparse arrays are non-negative ints.

IEEE NaNs and Infinities

The IEEE arithmetic NaN (not-a-number) for floats is a float denoted by 0n. Plus-infinity and Minus-infinity for floats are denoted by 0w and -0w. For example:

```
  0%0
0n
  1%0
0w
  -1%0
-0w
```

NaNs serve as nulls in q tables

Nulls

Working with nulls

Null values often represent missing values. Null values are used in tables to represent missing values. Some null values are exceptional values, such as the IEEE Nan for floats. Others are actual values, such as 0b for bools. Nulls for all datatypes can be found in the datatypes summary. Indexing a list with an out-of-range index produces the null value for the datatype of the list. For example,

```
  1 -2 12h[3]
0Nh
```

Three functions manipulate nulls: fill, fills and null.

Extract from Denis Shasha Primer:Operations on Atoms and Lists

```
x: 5
x / returns 5
y: 6
x+y / returns 11
x,: 9 2 -4 / returns a type error because x is not a list so concatentation
           / is not well defined.
x: enlist x
x / returns ,5 which means the list having the single element 5
x,: 9 2 -4  / now it works
x / returns 5 9 2 -4 (we don't need the initial comma because
  / any entity having several elements must be a list.
x+y / returns 11 15 8 2 (result of adding 6 to each element here)
z: x*y / form the product
z / returns 30 54 12 -24
z + 3 2 4 1 / gives 33 56 16 -23 by element by element addition
/ Operations are processed in right to left order.
/ Others think of operations as being evaluated in left OF right
/ order (by analogy to sum of f(x)).
/ Whichever way you think of it, there is no operator precedence,
/ only position precedence. Thus,
5 * 4 - 4 / returns 0, first do 4 - 4 and then multiply that result by 5.
```

Here are some other binary operations that can apply to atoms or lists in the same set (atom op atom; atom op list; list op atom; list op list if the same length).

a+b	Plus
a-b	Minus
a*b	Times
a%b	Divide
a=b	Equal
a>b	More

a

There are also a few other very common binary operators (you may have noticed that only the unary operators are English words; the binary ones are normally just keyboard characters).

~ asks whether two entities have the same contents.
, concatenates two lists.
_ drops elements of a list.
# selects elements of a list.
? finds the positions of elements of a list.
? is overloaded to generate random numbers.
bin supports binary search.

```
/ Note that the where operator finds the positions where there is 1b.
ii _ a / returns (" the";" people";" of";" the";" United";" States")
 / That is a list where each element is the part of the list between
 / one blank and the next.
Examples K: Binary operators
x: 2 4 3 5
y: 2 5 3 4
x = y / returns a boolean vector 1010b
x ~ y / returns 0b because these are not identical
z: 2 4 3 5
x ~ z / returns 1b because contents are identical
z,z,y,z / returns 2 4 3 5 2 4 3 5 2 5 3 4 2 4 3 5 the concatenation
w: z, (2*z), (3*y)
w / returns 2 4 3 5 4 8 6 10 6 15 9 12
5 _ w / returns 8 6 10 6 15 9 12 dropping the first 5 numbers
-5 _ w / returns 2 4 3 5 4 8 6 having dropped the last 5 numbers
a: "We the people of the United States"
a = " " / returns 0010001000000100100010000001000000b
ii: where a = " " / finds the positions of the blanks

 5 # a / returns "We th"
-5 # a / returns "tates" (the last 5)
7 # "abc" / returns "abcabca"
a ? "p" / returns 7 which is the first position with a "p"
a ? "ptb" / returns 7 3 34 because 7 is the first position of "p"
          / 3 is the first position of "t"
          / and "b" is never present so its position is the length of the string
          / which is 34.
/ When the left object is a scalar and the right number is a scalar
/ we can generate random numbers that can be float:
7 ? 5.2 / generates 7 numbers between 0 and 5.2
/ returns 0.9677783 4.321816 3.661838 2.394824 2.102985 0.9226833 2.556388
/ or whole numbers
9 ? 18  / returns 15 1 1 8 6 11 10 8 9
/ without replacement
-9 ? 18  / returns 13 9 10 2 3 0 15 1 7
/ Permutations
-15 ? 15 / 13 0 1 3 7 10 6 4 14 11 9 8 2 5 12

7 ? `cain`abel`job`isaac
/ returns `job`job`abel`abel`job`cain`isaac
evens: 2 * key 20
evens / returns 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
evens bin 4 / returns 2 because 4 is at position 2
evens bin 5 / also returns 2 because 5 < 6 which is at the next position
```

Extract from Denis Shasha Primer: Some Binary Operations on Dictionaries

```
schoolfriends: `bob`ted`carol`alice ! 23 28 30 24
rockfriends: `sue`steve`alice`bob`allan ! 19 19 24 23 34
/ multiplication applies to each element of the domain
2*rockfriends / returns sue`steve`alice`bob`allan!38 38 48 46 68
/ for each common element in the domain, the range parts are added.
schoolfriends+schoolfriends / returns `bob`ted`carol`alice!46 56 60 48
/ Same as above and in addition there is a union of the domains.
/ So bob and alice are doubled, but everyone else goes in as they were
/ in the base dictionaries.
schoolfriends+rockfriends
/ returns `bob`ted`carol`alice`sue`steve`allan!46 28 30 48 19 19 34
schoolfriends = rockfriends
/ returns `bob`ted`carol`alice`sue`steve`allan!1001000b
/ Note that in spite of the fact that bob and alice
/ are in different positions in the two dictionaries, the fact that
/ they have the same value is recognized.
```

Extract from Denis Shasha Primer: Type and Cast

The operator type determines the type of a q object.

```
type 100    / returns -6h
type 100 99 88 / returns 6h (in general lists of type T are positive
/ whereas a scalar of type T is the same negative value).
type 1.2 3.1 / returns 9h
```

Here are some further examples:

```
"x"$97                / cast the int 97 using the datatype name x for a byte
/ returns 0x61
/ representation of 97 in hex
"x"$"a"               / cast a char to a byte
/ return 0x61
/ "a" also happens to be encoded in hex 97
"c"$0x41              / cast a byte to a char
/ returns "A"
"d"$2003.03.23T08:31:53 / extract the date from a datetime value
/ returns 2003.03.23
/ the year.month.day value
"t"$2003.03.23T08:31:53
/ returns 08:31:53.000
```

Casting to and from strings is special. There one should use the upper case character corresponding to the type. Here are some examples.

```
"I"$"67" / returns 67
"S"$"abc 012" / returns `abc 012
"D"$"2003.03.23" / returns 2003.03.23
"D"$"2003-03-23"
"D"$"03/23/2003"
/ We have a means to create text from data too:
string "D"$"03/23/2003" / returns "2003.03.23"
```

Extract from Denis Shasha Primer: String identifiers

The operator like determines whether a string matches a pattern. The operator ss finds the positions of a pattern.

```
a: "We the people of the United States"
a like "people" / returns 0b because people doesn't match the whole string
a like "*people*" / returns 1b because the * is a wildcard.
a ss "the" / returns 3 17 which are the positions where the word "the" starts
```

There are a number of other important unary functions such as string,first, reverse, key and group.

The string function converts atoms (incl. symbols) to strings. The first function returns the first element of a list (or the domain of a dictionary or the first row of a table). The reverse function reverses a list. The key function is a way to generate a sequence of numbers. The group function on a vector returns a dictionary whose domain is the list of unique elements in the vector and whose range is a list of their positions.

```
string `ILoveNY / returns "ILoveNY"
reverse string `ILoveNY / returns "YNevoLI"
key 15 / returns 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
reverse key 15 / returns 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
z: 50+ key 5
z / returns 50 51 52 53 54
zdup: z, reverse z / Note that the comma is concatenation.
zdup / returns 50 51 52 53 54 54 53 52 51 50
group zdup / returns a directory with the unique elements
/ of zdup in  the domain and their positions in the range
/ 50 51 52 53 54!(0 9;1 8;2 7;3 6;4 5)
d:`name`salary! (`tom`dick`harry; 30 30 35)
first d / returns `tom`dick`harry
key d / returns `name`salary
e: flip d / names are domains and values are column
first e / returns `name`salary!(`tom;30)
`name xkey `e
key e / returns +(,`name)!,`tom`dick`harry
```

Functions

We have seen ways of manipulating atoms, lists and dictionaries. Some of these are in fact functions. This section introduces further functions.[1]

There are a number of different ways of classifying functions. For example they can be either:

- Atomic – these apply to atomic arguments and produce atomic results
- Aggregate (atom from list)
- Uniform (list from list) – these extend the concept of atom functions in that they apply to lists. The count of the argument list equals the count of the corresponding result list. Unlike an atom function, an item of a uniform function result is not solely dependent on the corresponding item of the argument. The relationship between the argument and result is more general.
- Other

Binary operations in mathematics are called dyadic functions in q, and unary operations are called monadic functions. For example, + is a dyadic function and floor is a monadic function.

<table>
<tr><th>Function</th><th>Details</th></tr>
<tr><td>**Absolute value**</td><td>abs x</td></tr>
<tr><td colspan="2">The Absolute Value function is a useful companion to the arithmetic functions.
Absolute Value leaves non-negative numbers unchanged and negates negative numbers. For example,
  abs -4.7
4.7</td></tr>
<tr><td>**acos**</td><td>acos x</td></tr>
<tr><td colspan="2">This is a unary function that returns the arcos (or inverse cosine) of x, with the result in radians.
  acos 0.1
1.470629
The arguments must be within -1<=x<1, otherwise the null value 0n is returned.</td></tr>
<tr><td>**aj**</td><td>Asof join</td></tr>
<tr><td colspan="2">Generally, this is used to get the prevailing quote (one table) **as of** the time of the trade (other table). The columns must be specified for this join to work correctly. In order to work with two columns the first must be either parted by (`p#) or grouped by (`g#) the key; if neither q will attempt to force conversion to `p#. Realtime data is sorted by time and `g# by sym and Historical data is `p# by sym.<br>Syntax:<br>  aj[`sym`time;trade;quote]   /Trade and quote are the names of the two tables here & and two columns are sym and time.</td></tr>
<tr><td>**all**</td><td>Logical AND (numeric min)</td></tr>
<tr><td colspan="2">This is an aggregate monadic function. For numeric types this function is equivalent to the **min** function.
  all 3 4 0 -2
-2</td></tr>
<tr><td>**And, Or and Logical Negation**</td><td>The logical functions () do not have separate symbols</td></tr>
<tr><td colspan="2">The logical functions () do not have separate symbols; they are, respectively, the restrictions of Min, Max and Not to bool arguments. For example,
  1b&1b   / And (Max)
1b
  1b|0b   / Or (Min)
1b
  not 1b / Logical Negate (Not)
0b</td></tr>
<tr><td>**any**</td><td>Logical OR (numeric max)</td></tr>
</table>

---

[1] Syntactically, q has verbs and functions. Verbs are primitive symbols and names that can be evaluated with infix notation, as in a+b . Functionally, verbs are dyadic functions that can be evaluated in the functional form f[a;b], but rarely are. For example, +[2;3] . Monadic primitive functions appear in the juxtaposition form of monadic function evaluation.

This is an aggregate monadic function. For numeric types this function is equivalent to the **max** function.

```
  any 9 2.0 -3.2 1.0
9f
```

asc
Order list ascending

This monadic function simply sorts a numeric list from smallest to largest. If a non-numeric is included the appropriate numeric null value is returned. Duplicates are allowed making this a uniform function.

```
  asc 1 4 7 -2 3
`s#-2 1 3 4 7
```

asin
asin x

This is a unary function that returns the arcsin (or inverse sine) of x, with the result in radians.

```
  asin -1.0
-1.570796
```

The arguments must be within -1<=x<=1, otherwise the null value 0n is returned.

atan
atan x

This is a unary function that returns the arctan (or inverse tangent) of x, with the result in radians.

```
  atan 3.14159%4
0.6657733
```

Valid arguments must be approximately of the order +10^9 to -10^9. If this is not the case then the function will not evaluate.

attr
attribute

The attr function gives the *attributes* of data, which describe how it is sorted. `s denotes fully sorted, e.g `s#2 3 3, `u denotes unique, e.g. `u#2 4 5 and `p and `g are used to refer to lists with repetition, with `p standing for parted and `g for grouped. As an example of their usefulness, `s#table marks a table to use binary search and marks its first column as sorted.

Attribute flags are descriptive, not prescriptive; **amends** and **appends** preserve flags iff the attribute is preserved. In other words, a sorted list remains sorted until an element is inserted in such a manner to disrupt the sort.

avg
Average

This function returns the arithmetic average of a list of numeric values. The numeric value can be a vector as well.

```
  avg (2  4 ; 3  6)
2.5  5
```

The syntax for a usual list is

```
  avg 3 5 7 9
6f
```

bin
Binary search

```
  a:asc key 10       /Create an ordered list
  a bin 5            / Perform a binary search on the list.
```

by
Groups rows in a table at given sym

The 'by' keyword allows rows to be grouped in a table at a given sym. It can make keyed or nested tables. For instance one can have

```
  select sum amount by sym from trade      /Finds total amount for each sym
```

ceiling
ceiling x

The Ceiling function is the complement to the **floor** function. Given a float or real the ceiling function always rounds up to the nearest integer, e.g.

```
  ceiling 2.7
3
  ceiling -3.4
-3
```

It is equivalent to 1+ Floor x.

cols
Lists columns

This keyword allows columns of a table to be listed, for example cols trade.

cor
correlation

This function gives the correlation between two numeric lists of the same length. If each list only has one member, the float null 0n is returned. The function output is between 1 and -1, with 1 denoting perfect correlation and -1 denoting perfect anti-correlation.

```
  cor[3 2 1 ; 2 -1 0]
0.6546537
```

cos
cos x

This unary function returns the cosine of x, where x is in radians.
Note that cos x repeats itself every 2*pi and consequently valid values of x are of the order + 10^9 to -10^9. For example:

```
  cos 3.14159          / 3.14159 is Pi to 5 dps
 -1f
```

count
Count list

This aggregate function counts the elements in a list and returns a single int value.

```
  count 1 2 3 4 5
5
  count “abcdefg”
7
  count `sym`trade 1 2 3      /If the list is not simple, i.e. all of the same type then the count only applies to the last type in the list
3
  count (`sym`trade;1;2;3)    /In this case, the list is heterogeneous, with 4 atoms not all of the same type.
4
```

cov
covariance

This function gives the covariance between two numeric lists of the same length. Attempting to find the covariance between two single numbers produces the null float value 0f.

```
  cov[1 2 ; -9 4]
3.25
```

cross — cross[x;y] returns the cartesian product

```
x: 3 4 5
y: `a`b`
cross[x;y] / returns (in the browser) the cartesian product:
((3;`a);(3;`b); (4;`a);(4;`b); (5;`a);(5;`b))
```

csv

This command allows queries in a browser to be exported to Excel by prefixing the query, such as
  http://localhost:5001/.csv?select from trade where sym=`IBM. Can be used with the port command \p as well as when saving and loading.

cut

This command allows a table or list to be cut at a certain point, with the syntax being (0 5 10)cut “safAasdfAasdf?”
The above example would equate to

```
  (0 5 10)cut “safAasdfAasdf?”
(“safAa”;”sdfAa”;”sdf?”)              /The argument is split at the 0’th 5’th and 10’th atoms (letters)
```

A single number such as

```
  5 cut “safAasdfAasdf?”
```

would result in the ‘word’ being split into 5 atom (letter) parts until its end, with the same result as above. Putting in a single negative number results in an error, whilst putting in **a single zero** leads to q shutting down. Putting in a number larger then the number of atoms results in the argument being returned as a single element list, for example:

```
  8 cut (1 2 3)
,1 2 3
```

delete — Delete rows/columns

This command enables rows or columns from a table to be deleted. The syntax is similar to that of a query, for instance

```
  delete from table where a<2      /Deleting rows where a>2
  delete cols from table            /Deleting cols
```

deltas — deltas a b c

The deltas function is another primitive uniform function, which produces the differences of neighboring items in its argument. For example,

```
  deltas 4 9 -5 1 2
4 5 -14 6 1
```

The ith item of the result of deltas v is v[i]-v[i-1] for i greater than 0. Note that the first item of the result is identical to the first item of the argument.

desc — Order list descending

This monadic function sorts a numeric list from largest to smallest. If a non-numeric is included the appropriate numeric null value is returned. Duplicates are allowed making this a uniform function.

```
  desc 1 4 7 -2 3
7 4 3 1 -2
```

dev — Deviation

This function gives the mathematical deviation for a list of numeric values, it is therefore an aggregate function. The numeric list needs to have at least 2 values or else the null 0f is returned.

```
  dev 4 8 3 -0 2.0 -4 3
3.410668
```

differ — Difference

This uniform function checks each member of a list to see if it differs from the preceding element. It returns a list of bool values with 1b indicating that the current value and the preceding one do differ whilst a 0b indicates the opposite. The first atom of a list returns 1b by default.

```
  differ 1 1 2 3 3 3 4 5 6
101100111b
  differ `mars`mars`jupiter`mars`earth`uranus`uranus
1011110b
  differ (1 2 ; 3 4 ; 3 4 ; 5 4 ; 5 4)
11010b
```

distinct

This monadic function returns the distinct elements of a list; that is, the output has no repetition.

```
  distinct 1 1 1 1 1 2 2 3
1 2 3
  distinct (1 2 ; 1 2 ; 1 2 ;`sym;`sym;”a”;”a”)
(1 2;`sym;”a”)
```

If it is used with a single positive number x of type *type*, then a random number, of type *type,* between 0 and x is generated (x-1 for an int or long).

```
  distinct 8
1            / An int produces an int result.
  distinct 0
67214319     /Produces a random long in the valid domain.
```

Divide — a%b

Divide is a dyadic arithmetic function.Note that Divide is denoted by %, not /. Numeric datatypes can be mixed in arithmetic expressions and required conversions from one to another are automatic. The arguments of % are converted to float before the operation is performed.

. For example,

```
  4%2
2.00
```

The result type is int if there are no occurrences of % and all the datatypes are bool, byte, short or int. Everything is converted to int before the operations are performed. For example,

```
  1b+1h
2
  10*0xab
1710
```

If long integers are included among the arguments and there are still no occurrences of % then the results are always of type long.

```
  1b+1h+1j
3j
  10j*0xab
1710j
```

Do
Do loop

The usual DO construct with syntax as follows

```
  Do[10;foo[]]          /Performs the foo function 10 times.
```

Each
‘

The each keyword (adverb), denoted by quote ‘, modifies dyadic functions and verbs to apply to the items of lists as opposed to the lists themselves. More information is in the Dictionaries and Associations section.

enlist

A monadic primitive function which enables one-item lists to be created.

```
  enlist 34.5
,34.5          /A one-item simple float list
  enlist (1;”as”;`sym)
, (1;”as”;`sym)         /A one item list is displayed with a leading comma.
```

Equal
a=b

This is a dyadic relational function. This function produces boolean values, where 1b means that the relation holds, or is true.The relational functions in q use a multiplicative tolerance when applied to floats. This makes arithmetic work better.

except

This function deletes items from a list.

```
  (2 3 4 5 6)except 2 4
3 5 6
  (“a”;”b”;”c”)except “a”
“bc”
```

exit

This keyword exits from a process, e.g. exit 0.

exp
Exponential (exp x)

This function returns e^x (e to the power of x) where ‘e’ is the natural number and is approximately equal to 2.718282.

```
  exp 1
2. 718282
```

fby

This function can take the place of ‘common by phrases’, thereby simplifying them – useful in where clauses.

For instance:

```
  select count I by sym from trade where price>(avg:price)fby sym
```

fill
used with nulls

There are three functions for processing null values. The dyadic function named fill replaces null values in the right argument with the atomic left argument. For example,

```
  0^1 2 3 0N
1 2 3 0
  100^1 2 -5 0N 10 0N
1 2 -5 100 10 100
  1.0^1.2 -4.5 0n 0n 15
1.20 -4.50 1.00 1.00 15.00
  `nobody^`tom`dick``harry
`tom`dick`nobody`harry
```

Integer left arguments are promoted when the right argument is float or real. For example,

```
  10^1 2 3 0n 4.5 0n
1.00 2.00 3.00 10.00 4.50 10.00
/ Suppose we have a sales vector and some days have an unknown number of sales.
/ We could fill in those unknown days with 0s:
sales: 45 21 0N 13 0N 11 34
adjsales: 0^sales
adjsales / returns 45 21 0 13 0 11 34
/ Or we could choose to fill in with the non-null values:
x: avg sales where not null sales
adjsales2: x ^ sales
adjsales2 / returns 45 21 24.8 13 24.8 11 34
```

fills
used with nulls

This function fills in nulls with the previous not null value.

```
   fills 1 0N 2 0N 0N 1 2 0N
1 1 2 2 2 1 2 1 2
```

first
First atom

This command returns the first atom of a list.

```
  first 1 2 3 4
```

```
1
  first (`first;1;”sad”;`candle;4)
`first
```

flip

Flip is a monadic primitive that applies to lists and associations. The effect is to interchange the top two levels of its argument. That is if we perform c:flip b, then b[i][j] and c[j][i] are identical for all valid indices i and j. In matrix terminology c is the transpose of b. In particular all items of a list must have the same count in order to be flipped.

```
  flip ((1 2;3);(4;5);(3 3 3;5 6))
((1 2;4;3 3 3);(3;5;5 6))
```

floor
floor x

The Floor function is a useful companion to the arithmetic functions. Floor applies to floats and reals and produces their integer parts as ints. For example,

```
  floor 2.7
2
  floor -3.4
-4
```

Floor is often used to truncate and round floats to a specified number of decimal places. For example, the first expression truncates the float 72.277 to 2 decimal places and the second expression rounds it to 2 decimal places.

```
  0.01*floor 100*72.277
72.27
  0.01*floor 0.5+100*72.277
72.28
```

from
from clause

The from clause as in ‘select from trade’.

Get

This allows one to memory map a file, as in get`:./2007.02.14/trade/price.

getenv
Get environment variable

Get an environment variable, for example getenv DISPLAY.

group
Group a list

This function returns an association of the **distinct** atoms of a list with their positions in the list.

```
  group 2 3 4 5 3 4 5
2 3 4 5 ! (,0;1 4;2 5;3 6)
  group “abcdddeeee”
“abcde” ! (,0;,1;,2;3 4 5;6 7 8 9)
```

gtime
GMT

Converts time to Greenwich Mean Time (GMT), such as gtime .z.Z

hclose
Handle close

This keyword closes an ipc connection or file handle. For example

```
  hclose h
```

hcount
Handle count

Handle count, also returns the size of a file in bytes. For example,

```
  hcount q.exe
143360
```

hdel
Handle delete

Handle delete, useful for files. For example,

```
  file:`:oracle.exe
  hdel file
```

hopen
Handle open

This keyword opens an ipc connection or file handle, such as

```
  H:hopen(`::2000)
```

hsym

This keyword converts a symbol to a file or process handle, such as

```
  hsym `file.txt
```

iasc
Index ascending

This is a uniform function which returns the indices of the ascended sorted list relative to the input list.

```
  iasc 5 4 3 6
2 1 0 3
```

idesc
Index descending

This is a uniform function which returns the indices of the descended sorted list relative to the input list.

```
  idesc 5 4 3 6
3 0 1 2
```

in
In a list

This dyadic function can be used to query lists (on the right-hand side) about their contents. The output is a bool list indicating whether or not a specified atom is in the queried list.

```
  (2 4)in 1 2 3
10b                  /2 is in the right-hand list, 4 isn’t.
  “me” in “team”
11b                 /Both “m” and “e” are in the right-hand list.
```

insert
Insert statement

This keyword is used to upload new data into a table. For instance

```
  `trade insert(2005.05.12;.z.z;`IBM;1.5;100)
```

inter
Intersection

This dyadic function returns the intersection of two lists; that is the atoms they have in common.

```
  1 2 3 inter 2 3
2 3
  “carthorse” inter “orchestra”
“carthorse”
```

inv
Matrix inverse

This monadic function returns the matrix inverse, but it should be noted that it only works on floats.

```
  inv(1.0 2.0;3.0 4.0)
(-2 1f;1.5 -0.5)
```

key

This keyword has three different functions, firstly it enumerates an int as follows:

```
  key 10
0 1 2 3 4 5 6 7 8 9
```

it can also obtain the contents of a directory, for example:

```
  key `:c:
```

and it can also be used to get the key of a table/dictionary

```
  key tbl
```

keys

This function obtains the keys of a table, for example,

```
  keys trade
,`price                    /Possible answer.
```

last
Last atom

This keyword returns the last atom in a list.

```
  last 1 2 3 4 5
5
last (`sym;”abc”;(1 2);9.0)
9f
```

Like

This function performs a string match, where list1[i] matches pattern in list2[i].

```
  list1[i] like list2[i]                    /The [i[ notation emphasises that the strins should match atom for atom.
```

Example

```
  select from stock where company like “M?c[rR][n-p]s*”   /? means any char and * denotes anything, so Microsoft would pass filter.
```

lj
Left join (x lj y)

The left join is a special case of the **asof join**, where either the y argument is keyed or both are. X must have a column of the same name as the key of y. As a result x fills out with any valid values from y and any rows not in y are filled with nulls.

load
Load data file

This keyword loads a q data file, such as

```
  load `:trade
```

log
Natural logarithm (log x)

This function returns the natural logarithm of a non-negative numeric value.

```
  log 1
0f
  log 0
-0w         /minus infinity
```

lower
Convert to lower case and floor

This function converts Upper case letters to lowercase in chars and varchars and applies the **floor** function to numeric values.

```
  lower(-2.3;”ABC”;`SYM;1.0 2.8 3.4)
(-3;”abc”;`sym;1 2 3)
```

lsq
Least squares

The least-squares approximation. The lsq notation can also form associations

```
  “abc” lsq 1 2 3
  “abc” ! 1 2 3
```

ltime
Local time

Converts to local time, as in ltime.z.z

Match
a ~ b

Match is not an atom function, but is related to Equal and is so useful when experimenting with the language that we'll define it here.

It often happens that you want to compare two results to determine whether or not they are identical. This is often done in this manual simply by looking at them, but that doesn't always work, e.g. when the console display doesn't contain all the details. Sometimes Equal can be used, but not always. The primitive function called Match is the function to use. For example,

```
  (1 2 3+4 5 6)~4 5 6+1 2 3
1b                         / the arguments are identical
  (1 2 3-4 5 6)~4 5 6-1 2 3
0b                         / these are not
  1 2 3 ~`a`b
0b                         / any two data objects can be compared
```

Comparison tolerance is used when matching floats.

Match depends on the datatype of the arguments, not just the values. For example,

```
  1~1h
0b
```

```
 3~3.0
0b
```

mavg

Moving Average (n mavg list)

Computes the n-number moving average of a list of numeric values. The function is uniform, so that the moving average builds up to the n-number one as the function goes along the list. The upshot of this is that the first atom of the output is the 1-number average of the input (first atom), the second is the 2-number average of the first and second atoms of the output and so forth and so on until the n-number moving average is reached.

```
  3 mvg 2 4 5 8
2 3 3.666667 5.666667           /2 is the average of 2, 3 is the average of 2 and 4, 3.666667 is the average of 2, 4 and 5 etc
```

If the count of the list is m, and we perform n mvg list with n>=m, the moving average just builds up in the usual manner, so the i'th atom of the output is the i-number moving average starting from the beginning of the input list.

Max and Min

a|b and a&b

The comparison functions are | for Max and & for Min.

```
  9|5  / Max
9
  9&5  / Min
5
```

The arguments can be of any numeric types, or both can be chars, or both syms. When the numeric types are different the result is one of the types, always the one to which the other argument can be safely converted. For example,

```
  9.0|5
9.00
  9.0e&5
5.00e
  0x18|1b
0x18
```

The Max of 2 chars is the one with the highest byte order. For example,

```
  "z"|"a"
"z"
```

The | gives the maximum no matter what the domain, so

```
x: "algebras"
y: "calculus"
x | y / returns "clleurus"
x & y / returns "aagcblas"
/ The max and min also apply to boolean results, e.g.
(x > y) | (x < y) / returns 11111110b
(x > y) & (x < y) / returns 00000000b
```

maxs

maxs a b c

The maxs function is a primitive uniform function. The nth result is the maximum of the first n items in the argument.

```
maxs 1 2 3 -4 5
1 2 3 3 5
```

md5

Encrypt

This function encrypts text using the "Message-Digest algorithm 5". For example

```
  md5”I know where you live”
0x8adc84de2ef391a352c5ca5366a8a450
```

mdev

Moving Standard Deviation (n mdev list)

This uniform function computes the moving standard deviation of a list of numeric values. The first atom of the output is always 0, as this is the measure of the deviation of a single number. The next atom of the output is the deviation of the first and second atoms of the input and so forth until the n'th atom of the input list is reached. After that, the first atom of the input is dropped from consideration and the next one is added etc.

```
  2 mdev 2 4 3 9 -3 2
0 1 0.5 3 6 2.5
```

med

Median value

This function returns the mathematical median of a list of numeric values.

```
  med 2 4 0 7
3f
```

meta

Displays table meta data.

```
  show meta `trade
```

mins

mins a b c

The mins function is a primitive uniform function. The nth result is the minimum of the first n items in the argument.

```
mins 1 2 3 -4 5
1 1 1 -4 -4
```

Minus

a-b

Minus is a dyadic arithmetic function. Numeric datatypes can be mixed in arithmetic expressions and required conversions from one to another are automatic.

The Sum, Product and Difference of two bools are ints.

```
  1b-1b
   0
```

The difference of two ints is always an int, the modulo 2^{32} value of the mathematical result.

Analogously the difference of two longs or a long and an int is always a long, the modulo 2^{64} value of the mathematical result.

mmax

Moving maximum (n mmax list)

This uniform function is the n-number moving maximum of the input list of numeric values. If n is greater than or equal to 'count list' then the output is the same as that of the **maxs** function. Until the n'th atom of the input list is reached the corresponding atom in the output is simply the corresponding maximum. After that the first atom of the input is dropped and the next one added, and the maximum of these numbers is the next atom of the output and so forth and so on.

```
  3 mmax 2 4 -1 9 0 4 7 8 12 4 -7 2
2 4 4 9 9 9 7 8 12 12 12 4
```

mmin

Moving minimum (n mmin list)

This uniform function is the n-number moving minimum of the input list of numeric values. If n is greater than or equal to ‘count list’ then the output is the same as that of the **mins** function. Until the n’th atom of the input list is reached the corresponding atom in the output is simply the corresponding minimum. After that the first atom of the input is dropped and the next one added, and the minimum of these numbers is the next atom of the output and so forth and so on.

```
  3 mmin 2 4 -1 9 0 4 7 8 12 4 -7 2
2 2 -1 -1 -1 0 0 4 7 4 -7 -7
```

mmu

Matrix multiplication (a mmu b)

This dyadic operator only works on matrices of type float. If ‘a’ is a **l x m** matrix and ‘b’ is a **m x n** matrix, then (a mmu b) is a **l x n** matrix. This function works for all matrices, except for the case of a column vector times a row vector – this is because both types of vectors are represented in the same way and **mmu** distinguishes between them according to the situation, that is:

```
  a:(2.0 3.0)              /Looks like a is assigned to be a 1 x 2 row vector.
  b:(2.0 3.0 ; 4.0 5.0)    /b is assigned to be a 2 x 2 matrix.
  a mmu b
16 21f      /A normal 1 x 2 by a 2 x 2 matrix calculation with the result, mathematically being a row vector.
  b mmu a
13 23f      /In this case q performs a 2 x 2 by a 2 x 1 matrix calculation (a is now a column vector!), technically giving a column
/vector output, though its representation is indistinguishable from that of a row vector.
```

Multiplying 2 vectors always results in the sum of the scalar product::

```
  (1.0 2.0 3.0) mmu (1.0 2.0 3.0)
14f         /In this case it is not possible to multiply a 3 x 1 vector by a 1x 3 one to get a 3 x 3 matrix, the operation is performed as
a 1 x 3 vector by a 3 x 1 one, to obtain a 1 x 1 matrix, or a scalar.
```

mod

m mod n returns the remainder of dividing m by n

26 mod5 / returns 1

More(Less)

a>b (a<b)

These are relational dyadic functions. These functions produce boolean values, where 1b means that the relation holds, or is true. For example,

```
  3>5      / 3 is not greater than 5
0b
  "z">"a"   / chars have byte order
1b
  `abc>`def
0b
```

The relational functions in q use a multiplicative tolerance when applied to floats. This makes arithmetic work better.

More examples;

```
x: 3 2 6 3
y: 7 1 4 6
x < y / returns 1001b because 3 < 7 (first positions of
      / x compared to first of y)
      / 2 < 1 is false, 6 < 4 is false and 3 < 6 is true
      / So true is 1b and false is 0b.
x: "algebras"
y: "calculus"
x > y / returns 01010100b because "a" > "c" is false, "l" > "a" is true etc.
```

msum

Moving sum (n msum list)

This uniform function is the n-number moving sum of the input list of numeric values. If n is greater than or equal to ‘count list’ then the output is the same as that of the **sums** function. Until the n’th atom of the input list is reached the corresponding atom in the output is simply the corresponding sum. After that the first atom of the input is dropped and the next one added, and the sum of these numbers is the next atom of the output and so forth and so on.

```
  3 msum 2 4 -1 9 0 4 7 8 12 4 -7 2
2 6 5 12 8 13 11 19 27 24 9 -1
```

neg

Negative

```
  3.0 * (neg 0.2)
-0.6
```

next

Next

Useful for selecting next `variable in a table for instance:

```
  select (price + next price)%2 by sym from trade
```

It can also be used to cycle through a list

```
  next 2 3 4
3 4 0N      /This use of next is uniform, with the first atom 2 being dropped and the null int being tacked on the end.
```

not

not x

There is one monadic relational function, named **not**. This function gives the relationship of all numeric values to zero: the result is 1b if the argument is identical to 0 and 0b if it is not. For example,

```
  not 34.56
0b
  not 0
1b
```

Less or Equal in q is **not a>b**, Greater or Equal is **not a<b** and Not Equal is **not a=b** .

```
x: "algebras"
```

```
y: "calculus"
x > y / returns 01010100b because "a" > "c" is false, "l" > "a" is true etc.
/ The expression x <= y can be simulated by not x > y.
not x > y / returns 10101011b
```

The relational functions in q use a multiplicative tolerance when applied to floats. This makes arithmetic work better.

null	null a

The monadic function named Null is an atomic function whose result has the same structure as the argument, with each atom replaced by 1b if the atom is a null, or 0b otherwise. For example,

```
  null 1 2 3 0N
0001b
  null 1 2 -5 0N 10 0N
000101b
  null 1.2 -4.5 0n 0n 15
00110b
  null `tom`dick``harry
0010b
```

This function is useful when replacing null values in a way other than that defined by Fill. For example, suppose you want to replace the null values with their previous values. The Where function used with Null, gives the indices of the null values.

```
  where null 1 2 3 0n 4.5 0n
3 5
```

The replacement can then be done as follows:

```
  v:1 2 3 0n 4.5 0n
  v -1+where null v
3.00 4.50
  v[i]:v -1+i:where null v
  v
1.00 2.00 3.00 3.00 4.50 4.50
```

There are two cases not handled by this expression. The first case is when the first item in v is null, and the second is when there are successive null values. The first case must be handled separately, depending on circumstances. For example, if you know the simple list will always be a float list and that a null first item should be replaced with 0.0, then the following expression will do.

```
  if[null first v;v[0]:0.0]
```

The second case can be handled in a While loop or with the kdb+ construct called Over. The while loop is used here. Let's first do all the iterative steps explicitly.

```
  f:{r:x;r[i]:r[-1+i:where null r];r}
  v:10 -3.1 0n 0.1 0n 0n 0n 3.4
  v:f v                              / fill the nulls
  v
10.00 -3.10 -3.10 0.10 0.10 0n 0n 3.40
  v:f v                              / fill the null
  v
10.00 -3.10 -3.10 0.10 0.10 0.10 0n 3.40
  v:f v                              / fill the last null
  v
10.00 -3.10 -3.10 0.10 0.10 0.10 0.10 3.40
```

The test we need for a While statement is whether or not any item of v is a null; if so, another pass through the loop must be made. Again, if we know that the simple list is a float list, then the test can be done with

```
0n in v
```

Here is the While statement.

```
  v:10 -3.1 0n 0.1 0n 0n 0n 3.4          / reset v
  while[0n in v;v:f v]
  v
10.00 -3.10 -3.10 0.10 0.10 0.10 0.10 3.40
```

peach	Parallel each

This function, whose name is a hybrid of ‘parallel’ and ‘each’, allows process across slaves. The syntax is

```
  foo peach list 1                    /foo is a function applied across the slaves named in list 1
```

Plus	a+b

Plus is a dyadic arithmetic function. Numeric datatypes can be mixed in arithmetic expressions and required conversions from one to another are automatic.

The Sum, Product and Difference of two bools are ints.

```
  1b+1b
  2
```

The sum, difference and product of two ints is always an int, the modulo 2^{32} value of the mathematical result. For example,

```
  a:256
  b:a+a*a
  b
65792
  c:65792j
  c*c
4328587264j                              / long result of long arithmetic
  b*b
33619968       / int result of int arithmetic
 (c*c)-b*b
```

4294967296j / 2^{32}, as a long

Analogously, the sum, difference and product of two longs or a long and an int is always a long, the modulo 2^{64} value of the mathematical result.

prd	Product (prd list)

This function produces the product of a numeric list.

```
  prd  1 2 3 4 5 6
720
```

prds

prds a b c

The prds function is a primitive uniform function. The nth result is the product of the first n items in the argument.

```
prds 1 2 3 -4 5
1.00 2.00 6.00 -24.00 -120.00
```

prev

Previous

The complement of the **next** function. Used with a numeric list, it pushes the list forwards, that is:

```
  prev 1 2 3 4
0N 1 2 3
```

It is also useful in lists

```
  select (price + prev price)%2 by sym from trade
```

rand

Pick an item from a list or generate a random number

The function rand with an atom works like 1? Atom, for example

```
  rand 1.0
0.6398249              /Random number between 0 and 1 (of type float)
```

whereas

```
  rand 3
2                      /Random integer between 0 and 2, importantly 3 cannot be returned!
```

However with a list it picks out one member at random

```
  rand `a`b`c`d
`a
  rand `a`b`c`d
`d
```

random

n ? list

kdb+ provides a convenient way to generate random test data, which is very useful when experimenting with the language or modeling applications. The primitive function called Rand produces random sequences of ints and floats. For example, the expression 20?5 produces an int vector of length 20 whose items are random ints between 0 and 4, as follows.

```
  20?5
4 3 3 4 1 2 2 0 1 3 1 4 0 2 2 1 4 4 2 4
```

If the right argument is a float, say 4.5, then the result is a simple float list whose items are random floats between 0.0 and 4.5. For example,

```
  10?4.5
3.13239 1.699364 2.898484 1.334554 3.085937 2.437705 2.540967 3.445748 1.838425 0.6240313
```

Sequences of random selections from a specific set of values can also be generated. To do that, first form a list of the values, say `Arthur`Steve`Dennis

The ints 0, 1 and 2 are the valid indices of this list. Consequently, 10?3 is a list of random indices into this list. A list of random selections from this list is generated as follows.

```
  v:`Arthur`Steve`Dennis
  v[10?count v]
`Dennis`Arthur`Steve`Arthur`Dennis`Steve`Arthur`Arthur`Steve`Steve
```

rank

Returns ranks of elements when they are known

This function returns the indices of the ordered list, therefore if the list is numeric the output matches that of the **iasc** function.

```
  rank `a`c`d`b
0 2 3 1
  rank “movie”
2 3 4 1 0
```

ratios

Pairwise ratios (a% prev a)

This is a uniform function which produces the ratio of an atom to the previous one in the list.

```
  ratios 2 3 7 8 4 6 100 2
2 1.5 2.333333 1.142857 0.5 1.5 16.666667 0.02
```

raze

Flatten a list of lists

This monadic function removes a layer of indexing from a list of lists, for instance:

```
  raze ((1 2;6 7) ; (“ads”;7 8);1 2)
(1 2;6 7;”a”;”d”;”s”;7;8;1;2)       /One layer of indexing removed
  raze  raze ((1 2;6 7) ; (“ads”;7 8);1 2)
(1;2;6;7;”a”;”d”;”s”;7;8;1;2)       /Final layer of indexing removed, further razes will perform no function.
```

read0

Read in a text file

This function can be used to read in a text file:

```
  read0`:file.txt
```

read1

Read in a q data file

This function can be used to read in a q data file:

```
  read1`:trade
```

reciprocal

invert x

```
x:7
reciprocal x / returns 0.1428571
```

reverse

Reverses a list

This uniform function simply reverses a list.

```
  reverse 3 1 4 1 5 9
9 5 1 4 1 3
  reverse “abracadabra”
“arbadacarba”
```

rload

A Kdb function to load a table to the q workspace, it works currently but will eventually be phased out.

rotate

n rotate z rotates z by n

```
z:`a`b`c`d`e`f`g`h`i
4 rotate z / returns the rotation `e`f`g`h`i`a`b`c`d
3 rotate z / returns the rotation `d`e`f`g`h`i`a`b`c
```

rsave

A Kdb function that saves a splayed table, for instance one partitioned by dates. It will eventually be phased out but currently works.

save

Save q data

This function allows q data to be saved.

```
  save `trade
```

select

Select statement

The select keyword is vital in querying tables via the **select .. by .. from ..** construct.

set

Set value of a variable

The set function can be used to assign a value to a variable

```
  `a set 2        /Assigns the value 2 to a.
```

It can also be used to save data to disk

```
  `:/data/2007.01.01/quote set quote                /Saves tbl as a single file.
  `:/data/2007.01.01/quote/ set quote               /Saves tbl splayed.
```

show

Display data

This allows a table to be shown in table format within the q console – similar to what would what be shown on a browser.

```
show `table
```

signum

Returns the sign of a number

This uniform function returns a numeric list whose atoms have the int value 1, -1 or 0. If the atom in the input list is greater than zero then the output value is 1, if the atom in the input list is less than zero then the output value is -1 and finally if the atom in the input list has value 0 then the corresponding output value is 0.

```
  signum 1 2 0 0 -3 -89999 23000
1 1 0 0 -1 -1 1
```

sin

Sin x

This unary function returns the sine of x, where x is in radians.

Note that sin x repeats itself every time its argument increases by 2*pi. Consequently valid values of x are approximately of the order + 10^9 to -10^9. For example:

```
  sin 3.14159        / 3.14159 is Pi to 5 dps.
2.65359e-006       /Sine of pi is zero, this approximation is close.
```

sqrt

Square root

This function returns the positive square root of a non-negative number. Attempting to find the square root of a negative number will produce the null float 0f.

```
  sqrt 2
1.414214
```

ssr

String search and replace

This function allows an input list to be probed for a specified ‘search string’, and then if it is found the search string is replaced with a specified ‘replace string’.

```
  ssr[“thecatinthehat” ; “cat” ; “mouse”]
“themouseinthehat”
```

Usefully, the third parameter can be a monadic function:

```
  ssr[“thecatinthehat”,”the”,reverse]
“ehtcatinehthat”
```

string

Converts to string

This function converts all types to a string format. For example

```
  string(1 2 3;`abc;”XYZ”;0b)
((,”1”;,”2”;,”3”) ; ”abc” ; (,”X”;,”Y”;,”Z”) ; ,”0”)
```

sublist

m n sublistz returns items m->n in z

This function produces a sublist of list x depending on condition y.

```
n sublist x          /First n elements of list x.
-n sublist x         /Last n elements of list x.
n m sublist x        /m elements of list x beginning at index n.
```

If the condition requests more elements than x has, the function will still work and return only the valid elements asked for.

```
  x:”abcdefghijklmnopqrstuvwxyz”
  9 10 sublist x
“jklmnopqrs”      /10 elements of x beginning at j.
  9 1000 sublist x
“jklmnopqrstuvwxyz”   /The valid 17 elements of x are given, the invalid other 983 elements are not given.
```

sum

sum list

This aggregate function adds all the elements of a numeric list and ouputs it in the appropriate type

```
  sum 3.0 3.0 4.0
10f
  sum 1 3 5
9
```

sums

sums a b c

This is a primitive uniform function.

```
sums 1 2 3 -4 5
1.00 3.00 6.00 2.00 7.00
```

This function is also called Running Sums. The relationship of result items to an argument list is that the nth result item is the sum of the first n items of the argument. For example, the third item, 6.00, is the sum of the first three items of the argument. Similarly, the fourth result item, 2.00, is the sum of the first four items of the argument.

sv

Scalar from vector

This function performs different tasks dependent on its arguments. It evaluates the base representation of numbers, which allows us to calculate the number of seconds in a month or convert a length from feet and inches to centimetres etc.

```
  24 60 60 sv 12 30 59
45059            /The number of seconds elapsed in a day at 12:30:59 in the afternoon.
  3 12 2.54 sv 5 10 0
177.8            /The approximate height in cm of someone who is 5'10".
```

When used with text the left argument is placed between each right argument except for the case when the left arg is `, then line breaks are replaced by newline chars ("\n").

```
  "|" sv ("abc";"def")
"abc|def"
  ` sv ("abc";"def")   /With the show keyword the output would look like:        abc
"abc\r\ndef\r\n"       /                                                          def
```

system

This allows a system command to be sent, for example

```
system "dir"
```

tables

Lists all tables

```
  tables `
```

tan

Tan x

This unary function returns the tangent of x, where x is in radians.

Note that tan x is approximately equal to tan (x+n*3.14159) where n is an integer and 3.14159 is Pi to 5 dps. Valid values of x are of the order + 10^9 to -10^9. For example:

```
  tan 3.14159
2.65359e-006      /Tan of pi is zero, this approximation is close to zero.
```

The tangent function however heads to plus or minus infinity at multiples of pi%2

```
  tan 3.14159%2
753696f           /If the approximation of pi was improved then the answer would tend closer to + infinity (0w).
```

Temporal Arithmetic

Working with dates

Ints representing day counts can be added to and subtracted from dates. As a result, dates can be added to and subtracted from dates to give day counts. For example,

```
  2001.11.21+3          / date +/- day count equals date
2001.11.24
  2001.11.21-23
2001.10.29
  2002.12.31-2001.11.21   / date +/- date equals day count
405
```

Ints representing milliseconds can be added to and subtracted from times. For example,

```
  05:30:20.100+15100  / time +/- milliseconds equals time
05:30:35.200
```

Floats can be added to and subtracted from datetime values. The integer part of a float represents a day count and the fractional part represents a fraction of a day. For example, 2 hours and a half can be added to a datetime as follows.

```
  2002.12.31T09:10:35.000+2.5%24  / % is Divide in q
2002.12.31T11:40:35.000
  2002.12.31T09:10:35.0+5.0      / Add 5 days to a datetime
2003.01.05T09:10:35.000
```

Dates and datetimes can be compared to one another, while times can only be compared to times.

```
  2002.12.31T09:10:35.005
```

til

Enumerate

On int's, **til** performs the same function as **key**

```
  til 10
0 1 2 3 4 5 6 7 8 9
```

Times

a*b

Times is a dyadic arithmetic function. Numeric datatypes can be mixed in arithmetic expressions and required conversions from one to another are automatic.

The Sum, Product and Difference of two bools are ints.

```
  1b*1b
  1
```

The product of two ints is always an int, the modulo 2^{32} value of the mathematical result.

Analogously, the product of two longs or a long and an int is always a long, the modulo 2^{64} value of the mathematical result.

trim

Eliminate string spaces

This function eliminates any trailing or leading spaces from a string.

```
  trim "      beard              "
"beard"
```

txf

Foreign key chasing

Deprecated function.

type

See **type** in the section entitled **Manipulating Atoms, Lists, Dictionaries and Verbs**.

ungroup
Flatten a nested table

This function can produce a flat table from a nested one, for example:

```
  ungroup select b by a from ([]a:1 2 1 2 3 3 1;b:1 2 3 4 5 6 7f)
+`a`b ! (`p#1 1 1 2 2 3 3 ; 1 3 7 2 4 5 6f)
```

union
Joins tables

This function allows us to join two tables together.

```
  tbl1 union tbl2
```

update
Update table data

This function can be used to update certain information in a table, for instance:

```
  update price:2 by sym from trade
```

See the **Dictionaries and Tables** section for more information on update.

upper
Converts to upper case

This function converts lower case letters to uppercase in chars and varchars.

```
  upper(”abc” ; ”sym”)       /Turning char list to uppercase
(”ABC” ; ”SYM”)
  upper ‘tim`tam             /Turning varchar to uppercase. No type mixing is allowed.
`TIM`TAM
```

uj
Union join

This function joins two tables together such that the columns in the new table are the union of the two sets of columns from the original tables. As such, no duplicates are allowed and if required the new table is filled out with null values.

Uj will not work with keyed tables unless both tables have the same key.

```
  tbl1 uj tbl2
```

value

The **value** function has two main uses, one is to execute a string such as

```
  value”a:2”          /’a’ now has the value 2.
```

and the other is the get the value of a dict

```
  value `a`b ! 1 2
1 2
```

var
Variance (var list)

This aggregate function computes the mathematical variance of a list of numbers.

```
  var 2 3 4 2 3
0.56
  var 2 2 2 2
0f
```

vs
Vector from scalar

This dyadic function produces a vector quantity from a scalar quantity, with the left-hand string being the identifier for the individual vector atoms.

```
  “|” vs “20050202|IBM|20.23”
(“20050202” ;” IBM” ; “20.23”)
```

xasc
Order table ascending

This dyadic function allows a table (right-hand argument) to be sorted such that one column (left-hand argument) is in ascending order. For example

```
  `price xasc trade              /Orders the table ‘trade’ such that the price column is ascending.
```

xbar
x xbar y

For ints and floats, this dyadic function acts as a scaled type of the **floor** function, such that (x xbar y) is the same as x*(floor y%x).

```
  2 xbar 1.7 5.0 6.3
0 4 6
```

xcol
Renames columns of a table

With this function the columns of a table can be renamed, for instance to rename the first two columns of table ‘tbl’ all one has to do is:

```
  `newCol1`newCol2 xcol tbl
```

and the first two columns are now called newCol1 and newCol2.

xcols
Reorders the columns of a table

This dyadic function allows the user to reorder the columns of a table, as well as repeat some columns if necessary. For instance, consider the following trade table:

```
time         sym  price size
----------------------------
09:30:00.000 MSFT 10.75 100
09:31:00.000 FT   20    400
```

and put the last two columns first by performing

```
  `price`size xcols trade
```

```
price size time         sym       /No actual assignment has been made to trade, so change is not permanent.
----------------------------
10.75 100  09:30:00.000 MSFT
20    400  09:31:00.000 FT
```

It is possible to repeat columns, but in that case the others are always put in, i.e.

```
  `size`size`size xcols trade
```

```
size size size time         sym  price
--------------------------------------
100  100  100  09:30:00.000 MSFT 10.75
400  400  400  09:31:00.000 FT   20
```

xdesc

Order table descending

This dyadic function allows a table (right-hand argument) to be sorted such that one column (left-hand argument) is in descending order. For example

  `price xdesc trade   /Orders the table ‘trade’ such that the price column is descending.

xexp

m xexp n raises m to the power of n

2 xexp5 / returns 32f (2 to the power 5)
5 xexp2 / returns 25f (5 to the power 2)
-2 xexp 3 /returns -8f
2.1 xexp 5.2 / returns 47.37403

xgroup

Creates nested table

This dyadic function allows the creation of a nested table, for instance
  `a xgroup ([]a:1 2 1 2 3 3 1;b:1 2 3 4 5 6 7f)
produces

```
a| b
-| -----
1| 1 3 7
2| 2 4
3| 5 6
```

xkey

Set key on table

This dyadic function allows a key to be set on a table, for instance
  `orderid key trade   /orderid is now a key, assuming it was a valid column in the first place.

xlog

x xlog y

This dyadic function computes the logarithm of y to the base x.
  10 xlog 100
2f
  3 xlog 100
4.191807

xprev

Vector shift

This dyadic function shifts a vector by n elements, for example
  select price-5 xprev price from trade where sym=`IBM   /5 tick delta – allows the user to track changes in the IBM price

xrank

Generalised rank (n xrank list/table etc)

Whereas **rank** gave each atom of a list, for example, its own *unique* index, xrank groups the entire list/table etc into n distinct groups. For instance, splitting the first ten natural numbers into 2 groups gives the unsurprising result
  2 xrank 1 2 3 4 5 6 7 8 9 10   /xrank[2;1 2 3 4 5 6 7 8 9 10]   -   different syntax but works the same
0 0 0 0 0 1 1 1 1 1
  10 xrank 1 2 3 4 5 6 7 8 9 10
0 1 2 3 4 5 6 7 8 9   /Forcing a list of 10 atoms into 10 distinct groups results in the same behaviour as **rank**.

It can also be useful in grouping data, for instance
  t:flip `val`name!((100?100);(100?(`MSFT`CSCO`ORCL)))
  show select MinVal:min val,MaxVal:max val, NumItemsInBucket:count I by xrank[4;val] from t

```
val| MinVal MaxVal NumItemsInBucket
---| -----------------------------
0  | 0      28     25
1  | 28     51     25
2  | 53     74     25                    /Because of the random generation of the table there is no guarantee that this
3  | 74     97     25                    /exact table will be replicated with the given code.
```

wavg

Weighted average

Can be used to compute a weighted average of a certain variable dependent on a certain condition. This obviously lends itself to table queries such as:
  select size wavg price by sym from trade   /Calculates weighted average of **price** conditional on **size**.

where

Result filter

This is used to filter a result, for example
  select from trade where sym=`IBM

within

Result filter

This is used as a filter on a range, as in
  select from trade where price within 1.5 1.7

wsum

Weighted Sum

This function computes the weighted sum of a certain variable dependent on a certain condition. This lends itself to table queries such as:
  select size wsum price by sym from trade   /Calculates weighted sum of **price** conditional on **size**.

Order of Evaluation

q is a rich, orthogonal language and therefore has no precedence among functions. Unlike SQL and conventional mathematical notation functions execute left OF right, e.g. write x*1+rate instead of x*(1+rate) . Uniform functions preserve length. They include the atomic functions, integrals (sums prds ..) and derivatives (deltas ratios ..) . Q has no ambivalence so we must use neg x (not -x) .

Extract from Denis Shasha Primer: Procedures

As we are starting to develop some expressions of non-trivial length, it is time to show how to define procedures that embody those expressions. Forming a procedure is like assigning an expression (in programming language terminology, a "lambda expression") to a variable. Because set manipulation occurs all over most application, we define set primitives first.

```
/ the [x;y] indicates the name of the arguments (the formal parameters)
/ The unary operator distinct removes duplicates from a list.
intersect:{[x;y] distinct x where x in y}
a: 4 2 6 4 3 5 1
b: 3 5 4 6 7 2 8 9
intersect[a;b] / returns 4 2 6 3 5
/ Next we define a function that determines whether one list
/ is a subset of the next.
/ The function min returns the minimum element of a vector.
/ If a boolean vector, then min will return 1 only if all boolean
/ elements are 1.
subset:{[x;y] min x in y}
subset[a;b] / returns 0b
subset[a;a] / returns 1b
c: `a`b`a`b`d`e / remember no spaces in symbol lists
d: `b`d`a`e`f
subset[c;d] / returns 1b
subset[d;c] / returns 0b
/ Set difference is the discovery of all elements in a first list
/ that are not in a second.
/ This function produces a boolean vector of the elements in x
/ that are in y (x in y).
/ Flips the bits so 1 goes to 0 and 0 goes to 1 (not x in y).
/ Then it finds the locations of those bits (where not x in y).
/ Finally, it indexes x with those locations.
difference:{[x;y] x where not x in y}
difference[c;d] /returns  0#` because there is no such element
difference[d;c] / returns ,`f because f alone is in d but not in c.
```

Now we resume our exploration of other functionality. Suppose we want to create a basic set of statistical procedures: average, variance, standard deviation, and correlation.

Examples H: Statistical functions

```
var:{[x] (avg[x*x]) - (avg[x])*(avg[x])}
std:{[x] sqrt[var[x]]}
a: 4 3 6 13 1 32 8
std[a] / returns 9.839529
```

The built in primitives avg and sqrt are joined by other familiar ones:
log exp count sum prd min max

Working with the Database and Database Design

Creating Tables

Tables can be created by using the syntactic form to initialise the table with empty columns and then populate it with applications of the insert function. Tables are collections of data whose items, or "columns", have names. Following the SQL convention, tables are created by naming the items and initializing the items as empty lists. Since all simple lists are also general lists, any type of data can be inserted in this empty table.

```
q) trade:([]time:();sym:();price:();size:())
q) trade
+`time`sym`price`size!(();();();())
```

Colon (:) denotes specification. The above expression defines a table named trade with items named time, sym,price and size. The square brackets are used to define primary keys. If there is nothing between the square brackets then there are no primary keys; as in the situation above. If there is more than one primary key then those definitions would be separated by semicolons. The parentheses pair () denotes an empty list, which means that each of the table items is defined to be an empty list. The columns time, sym, price and size are the data columns. Note that the definitions of these columns are separated by a semicolon.

At this point, the datatypes of the columns are unspecified, this can be seen this by using the meta function (which shows the datatypes of a table), as in;

```
q) show meta tradecol     | type0 fkey0 attr0
--------------- | -----------------
time            |
sym             |
price           |
size            |
```

The datatypes of the columns will be fixed the first time data is inserted into the table, which can be done with the insert function.

```
q) insert[`trade](09:30:01.000;`AIG;59.25;2000)
,0
```

An alternative notation, performing the exact same insertion is:

```
q) `trade insert(09:30:01.000;`AIG;59.25;2000)
,1
```

Notice that the output in this case is 1, as opposed to the earlier 0; this is because the output denotes the index of the inserted row (with the index beginning at zero).

Assuming now that only the first row was inserted gives:

```
q) trade
+`time`sym`price`size!(,09:30:01.000;,`AIG;,59.25;,2000)
```

The symbol `trade holds the name of the table.[2] Now the column datatypes are more specific and henceforth the data that can be inserted is limited in type. The insert function is a dyadic function, taking the table name and data to be inserted as arguments, (note that above we have used the insert function and table name to create the function projection insert[`trade], see Defined Functions). The return value of the insert function are the indices of the inserted rows .The new data for a column is automatically cast to the datatype of the column (e.g., an int will be converted to a float) or an error is reported if that is not possible. This type of

---

[2] In general, data objects in the q workspace can be referred to in two ways, simply by their names or by symbols whose contents are those names. In the case of insert, using the name instead of the symbol will produce a new table with the new data, while using the symbol causes the named object to be modified with the new data. The above insert statement modifies the table trade, which now has one row

insertion can only be done in place, a similar construct using ‘delete’ is not allowed. To delete certain rows the trade table would have to be overwritten with an amended version of itself either using the assignment operator ‘:’ or the Amend function. Similarly, updates in place cannot be performed by the ‘update’ keyword, a method like those in the previous sentence would have to be employed.

```
q) 0#trade
+`time`sym`price`size!(`time$();`symbol$();`float$();`int$())
```

If no data is available it may be best to initialize the tables with empty lists to indicate the appropriate datatypes. This is necessary for foreign keys. Null values can be used, as in

```
q) trade:([]time:0#0Nt;sym:0#`;price:0#0n;size:0#0N)
q) trade
+`time`sym`price`size!( `time$();`symbol$();`float$();`int$())
```

It is also possible to initialise and populate a table at the same time, as follows.

```
q) trade:([]time:09:30:01.000 10:15:02.000;sym:`AIG`IBM;price:65.0 88.0;size:100 200)
q) trade
+`time`sym`price`size!(09:30:01.000 10:15:02.000;`AIG`IBM;65.0 88.0;100 200)
```

Even so, this is usually done only for small tables. Large amounts of data are often read from files and organized in lists. For example, the data for the trade table might have been created as a list,

```
q) listofdata: ( 09:30:01.000 10:15:02.000;`AIG`IBM;65.0 88.0;100 200)
q) trade:flip `time`sym`price`size!listofdata
+`time`sym`price`size!(09:30:01.000 10:15:02.000;`AIG`IBM;65.0 88.0;100 200)
```

Above we have used the fact that a dictionary is the flip of a table, this is covered in more detail below.

Foreign Keys

A foreign key defines a mapping from the rows of the table in which it is defined to the rows of the table with the corresponding primary key. Foreign keys in SQL provide referential integrity. In other words, an attempt to insert a foreign key value that is not in the primary key will fail. This is also true in q. Consider the following example where an industry table is defined to record the industry to which a particular company or sym belongs. The sym is defined as a primary key as each sym only appears once in the table. The trade table is defined as before except that sym is defined as a foreign key relating it to the sym in the industry table.The two tables have been populated below with some valid data.

```
q) industry:([sym:()]ind:())
q) `industry insert(`AIG;`Insurance)
q) `industry insert(`IBM;`Consulting)
q) trade: ([]time:();sym:`industry$();price:();size:())
q) `trade insert(09:30:01.000;`AIG;59.25;2000)
q) `trade insert(09:30:02.000;`AIG;59.50;1000)
q) `trade insert(09:30:04.000;`IBM;54.25;200)
```

Attempting to insert a trade for a sym, which has no corresponding entry in the industry table, will fail. In the example below an entry for MSFT must first be inserted in the industry table. In SQL one says that a foreign key represents a reference from its host table to the primary key table; hence the term referential integrity. In q the reference is actually a mapping from one table to the other; it may be more accurate to use the term domain integrity here. The error message ‘cast simply indicates the problem mentioned at the start of the paragraph.

```
q) `trade insert(09:30:06.000;`MSFT;79.25;250)
‘cast
```

The foreign key expression `industry$() in the definition of trade, which denotes the mapping of trade to industry, is called an enumeration. The symbol `industry in the expression refers to the industry table. You can rename column sym in the industry table or column sym in the trade table and the expression `industry$() remains the same. However, if you change the name of the industry table to, say sector, then the foreign key expression must be changed to `sector$().

The mapping represented by a foreign key defines each corresponding column of the primary key table as a so-called virtual column in the foreign key table. For example, the ind column of the industry table is a virtual column of the trade table, as the following display attempts to illustrate:

```
time          sym   price  size   ind
-----------------------------------------------------------------------------
09:30:01.000  AIG   59.25  2000   Insurance
09:30:02.000  AIG   59.50  1000   Insurance
09:30:04.000  IBM   54.25  200    Consulting
```

The first 4 columns are identical to the trade table. You can see that wherever two items of the sym column are identical, the corresponding items of the industry column are identical. For example, according to the industry table the AIG is an Insurance stock, and Insurance appears in the above industry column wherever AIG is in the sym column.

q has a simple mechanism for specifying virtual columns in queries called dot notation. Consider the following example:

```
q) select sym.ind,size from trade
q) +`ind`size!(`Insurance`Insurance`Consultancy;2000 1000 200)
```

In a console (using the show keyword) the result appears as follows;

```
ind        size
----------------
Insurance   2000
Insurance   1000
Consulting   200
```

Note that the industry is accessed simply using the general notation select a.b from c

- a is the foreign key (sym)
- b is a field in the primary key table (ind)
- c is the foreign key table (trade)

In the case above, we used just one foreign key relationship, but if the ind column in the industry table had itself been a foreign key, we could have accessed columns in the corresponding primary key table through the dot notation as well as sym.ind.col.

Dictionaries and Tables

A table can be thought of as consisting of a dictionary or some dictionaries, known as record(s).This close relationship between dictionaries and tables is fortunate in that it allows us to take advantage of the structural primitives, such as indexing, which apply to lists.

Here are some examples.

```
 q) trade[1] / any record (row) with an atom index
`time`sym`price`size!(09:30:02.000;`industry$;`AIG;59.5;1000)
q) first trade / the first row, as a record
`time`sym`price`size!(09:30:01.000;` ;`industry$;`AIG;59.25;2000)
q) trade[0] / this also gives the first row
`time`sym`price`size!(09:30:01.000;` ;`industry$;`AIG;59.25;2000)
q) count trade / the number of rows
3
q)  trade[10] / An out-of-range index gives a record of the null values for each column
`time`sym`price`size!((); `industry$;();())
q) trade[2;`price] / indexing at depth applies to tables
54.25
q) trade[2][`price] / another syntactical alternative
54.25
  q) trade[1 2] / A simple list of indices gives a sub-table.
+`time`sym`price`size!(09:30:02.000 09:30:04.000;`industry$`AIG`IBM;59.5 54.25;100 200)
```

It is instructive to note that the console display of a table corresponds to the functional form for defining tables. We are now in a position to discuss this form. Everything to the right of the leading + is the definition of a dictionary. That dictionary has the column names of the trade table as its keys and the contents of the columns as its value list. The leading + denotes the flip function, or transpose. We have already observed that to convert from a dictionary of lists to a table, we use the unary (single argument) verb "flip".If you apply flip twice in succession you get back to where you started, which means that the flip of a table is a dictionary.

For example,

```
q) trade
+ `time`sym`price`size!( 09:30:01.000 09:30:02.00 009:30:04.000  ;`industry$`AIG….)
q) flip trade
`time`sym`price`size!( 09:30:01.000 09:30:02.00 009:30:04.000  ;`industry$`AIG….)
```

The distinction is very subtle when viewed in the console – the table is denoted by a leading +, whereas the dictionary or flipped table isn’t. The difference is more easily visualized using a browser. The first display shows the trade table whilst the second shows the flipped table/dictionary x. The word flip can be thought of as suggesting the swapping of columns for rows in the geometrical sense described here. If you are familiar with linear algebra think matrix transposition. The reason this "flipped" representation is convenient is that we can index the table as if were an array. There are other advantages as well. In addition, tables may have some columns (also known as fields) which are "key fields". These again form a domain for the non-key fields. So, these keyed tables are again dictionaries. In this case there is a functional relationship from key fields to non-key fields.

```
(trade)
time         sym  price  size
------------------------------
09:30:01.000  AIG  59.25  2000
09:30:02.000  AIG   59.5  1000
09:30:04.000  IBM  54.25   200

(dict:flip trade)
time   ..
sym    ..
price  ..
size   ..
```

Since dict is a dictionary it can be indexed with items from its key list. In particular,

```
q) dict[`sym]
`industry$`AIG`AIG`IBM \ or equivalently
q) dict[`sym;]
`industry$`AIG`AIG`IBM
```

Since dict is the flip of trade (and trade is not a keyed table), we can use the same indices of trade, but reversed.

```
q) trade[;`sym]
`industry$`AIG`AIG`IBM
```

Note that any dictionary can be flipped and a table will result, as long as the items of its value list are all lists with the same count. For example,

```
q)  flip 1 2 3 4!(1 2; 3.2 4;`x`y;"ab")
+1 2 3 4!(1 2;3.20 4.00;`x`y;"ab")
```

The result is a table with columns named 1, 2, 3 and 4. This is not a relational table, however; q select and update expressions only apply to tables that are the flips of dictionaries whose keys are valid simple names (all alphanumeric characters and a leading alphabetic).

Strictly speaking, a key table is not a table, but a pair of tables instead. The primary key columns form one table and the data columns the other. For example, the key table industry has the primary key ind and data column sym. There are two tables namely +(,‘sym)!,`AIG`IBM`AIG and +(,ind)!,`Insurance`Consultancy`Insurance. On the console industry displays as follows:

```
q) industry
(+(,‘sym)!,`AIG`IBM`AIG)!+(,ind)!,`Insurance`Consultancy`Insurance
```

To the left is the table of primary key(s) in parentheses, followed by the Xkey function (!) followed by data table. Since industry is a dictionary its primary key table can be extracted with the key function and its data table, with the value function.

```
  q) key industry
+(,‘sym)!,`AIG`IBM`AIG
 q) value industry
+(,ind)!,`Insurance`Consultancy`Insurance
```

We can index keyed tables with the primary key value as below

```
q) show industry`AIG
```

Extract from Denis Shasha Primer: Table Formation and Access

```
/ I have a dictionary where the left column is the key
/ and the right column is a list of two vectors (each having three elements)
d:`name`salary! (`tom`dick`harry;30 30 35)
/ This returns (if we look through the http browser, see appendix)
/ name    ..
/ salary  ..
e: flip d / list of dicts instead of dict of lists
/ Again looking through the http browser:
/ name   salary
/ -------------
/ tom        30
/ dick       30
/ harry      35
e[1] / returns `name`salary!(`dick;30) the second row of the table.
/ Now try:
select name from e
select sum salary from e
/ Now let's look at the implications of having a key field.
e2: e
`name xkey `e2 / creates a key.
/ (Note that salary cannot be a key because 30 appears twice.)
keys e2 / returns ,`name
cols e2 / returns `name`salary
edouble: e2+e2 / returns
edouble / (+(,`name)!,`tom`dick`harry)!+(,`salary)!,60 60 70
/ Note that all fields are updated.
d3:`name`salary!(`bob`alice`dick`ted;130 15 235 11)
e3: flip  d3
`name xkey `e3
ecombine: e3-e2
ecombine / returns
/ (+(,`name)!,`bob`alice`dick`ted`tom`harry)!+(,`salary)!,130 15 205 11 -30 -35
/ Now let us say that we append information to e and to e2.
`e insert(`ann;50)
```

```
e / +`name`salary!(`tom`dick`harry`ann;30 30 35 50)
/ Tables can be formed directly as well.
/ We will start with non-keyed tables (thus the []).
mytab:([]x: 1 2 3; y: 10 20 30)
mytab / returns +`x`y!(1 2 3;10 20 30) where the + here means flip
mytab.x / returns 1 2 3
mytab.y / returns 10 20 30
select x from mytab where y = 20 / returns +(,`x)!,,2
select x,y from mytab where y < 21 / returns +`x`y!(1 2;10 20)
/ These last two return tables, single and double column.
/ Sometimes we don't want to return a table, but some simpler type.
/ In that case, we use the verb exec
exec x from mytab where y < 21 / returns 1 2 directly
/ Here are two create table statements.
/ Both have columns a and b as keys, as indicated by the brackets.
/ The closing right bracket omits the need for a
/ semi-colon.
t:([a:2 3 4 5; b:30 40 30 40] c:10 20 30 40)
u:([a:2 3 2 3; b:30 30 40 40] c:4 2 8 9)
/ t returns (+`a`b!(2 3 4 5;30 40 30 40))!+(,`c)!,10 20 30 40
/ Thus it's like two tables with the left one being the domain.
t-u / returns
/ (+`a`b!(2 3 4 5 3 2;30 40 30 40 30 40))!+(,`c)!,6 11 30 40 -2 -8
/ The first entry in the result of substracting 10-4.
/ The second is 20-9. In both cases the keys match.
/ The next two entries (4;30) and (5;40) are present only in t.
/ The last two are present only in u.
z: 0!t / eliminates the keys from t. This is sometimes useful.
```

Insert and Upsert

Kdb+ has two distinct primitive datatypes to which q applies: tables and keyed tables. With regard to traditional RDBMS's treat both datatypes as tables except for the key check on inserts. Tables can be created functionally with q primitive functions or with the syntactic form ([PrimaryKeyCols]Cols).

Inserts have been dealt with previously and are similar to SQL92 type inserts. q adds the notion of upserts.If necessary it will cast incoming data to match the columns and it will check primary key violations.

The q statement

```
t,u
```

is insert for tables and upsert (update existing and insert new) for keyed tables t and u, the schemas of t and u must match in either case.

Suppose we have trade and industry as before i.e

```
q) show trade
time         sym price size
---------------------------
09:30:01.000 AIG 59.25 2000
09:30:02.000 AIG 59.5  1000
09:30:04.000 IBM 54.25 200
q>show industry
sym  | ind
-----  | ----------
AIG  | Insurance
IBM  | Consulting
```

And the new tables newtrade and newindustry

```
q) newtrade:([]time:(09:30:01.000;09:30:06.000);sym:(`industry$`AIG`IBM);price:(59.25;60.1);size:(2000 400)
q) newindustry:([sym:(`AIG`MSFT)];ind:(`Insurance`Software))
```

Then

```
q) show trade,newtrade
time                   sym price size
---------------------------------------------
09:30:01.000           AIG 59.25 2000
09:30:02.000           AIG 59.5  1000
09:30:04.000           IBM 54.25 200
09:30:01.000           AIG 59.25 2000
09:30:06.000           IBM 60.1  400
```

Notice that the first and fourth rows of this new table are identical since an insert was performed

```
q) show industry,newindustry
sym          | ind
-------------|----------
AIG          | Insurance
IBM          | Consulting
MSFT         | Software
```

This time an upsert was performed because the arguments to “,” were keyed tables.

Extract from Denis Shasha Primer: Modifying Tables

We've already seen how to build tables from dictionaries or from direct statements. SQL 92-style inserts, updates and deletes are also possible. Q adds the notion of upserts.

```
book:([book: ()] language: (); numprintings: ())
insert[`book](`forwhomthebelltolls; `english; 3)
insert[`book](`salambo; `french; 9)
book: update language:`French from book where book=`salambo
book2:([book: ()] language: (); numprintings: ())
/ An alternate insert notation
`book2 insert (`secretwindow; `english; 4)
`book2 insert (`salambo; `Fch; 9)
/ go back to the classic insert, just for fun
insert[`book2](`shining; `english; 2)
book3: book, book2 / book2 adds all rows where the key field is
/ new and replaces values where the keyfield is not present.
/ These upsert semantics (insert rows having new keys and replace
/ range values of exsiting keys) are due to the fact that book
/ and book2 are both keyed tables.
select language from book3 where book=`salambo
/ Returns +(,`language)!,,`Fch
count select from book3 / returns 4 because book has two rows, book2 has 3
/ rows but one of those rows has the key `salambo.
book3: delete from book3 where book=`secretwindow
count select from book3 / return 3
```

Updates and update aggregations

In SQL, update statements modify existing tables. This is also true in q, but q has also extended the traditional definition to include the creation of new columns. It also permits update expressions to produce new tables instead of modifying the source tables in place. In effect, update expressions are an alternative form of select expressions. Here is an example comparing an aggregation within a select expression to an aggregation within an update expression.

```
q) trade:([]time;sym;price;size)
q) `trade insert(09:30:00.000;`AIG;10.75;200)
q) `trade insert(09:30:01.000;`IBM;10.55;500)
q) `trade insert(09:30:02.000;`IBM;10.75;100)
q) `trade insert(09:30:03.000;`AIG;10.75;400)
q) `trade insert(09:30:04.000;`MSFT;10.45;100)
q) `trade insert(09:30:05.000;`IBM;10.35;100)
```

```
q) `trade insert(09:30:06.000;`MSFT;10.75;400)
q) avgpricetrades1:select ap:avg price by sym from trade
q) avgpricetrades2: update ap:avg price by sym from trade

q) show avgpricetrades1
sym      ap
----------------
AIG   10.75
IBM   10.55
MSFT   10.6


q) show avgpricetrades2
time              sym    price    size    ap
----------------------------------------------------
09:30:00.000  AIG   10.75   200  10.75
09:30:01.000  IBM   10.55   500  10.55
09:30:02.000  IBM   10.75   100  10.55
09:30:03.000  AIG   10.75   400  10.75
09:30:04.000  MSFT  10.45   100   10.6
09:30:05.000  IBM   10.35   100  10.55
09:30:06.000  MSFT  10.75   400   10.6
```

If you compare the ap columns of the two results you will see that the aggregations in the select result are distributed in the aggregation column of the update result according to the sym column. For example, the aggregation value for `AIG, which is 10.75, appears in every row of the update result where the sym value is `AIG.

Update expressions that include aggregations are called update aggregations, or update-by expressions.

Uniform functions are effective in update aggregations because of the way aggregation results are distributed. For example, the following aggregation computes the percent change in prices for each sym.

```
q) tradetbl3:update pct:100*(deltas price)%price by sym from trade

q) show tradetbl3
time          sym   price  size        pct
-----------------------------------------------------------
09:30:00.000  AIG   10.75   200        100
09:30:01.000  IBM   10.55   500        100
09:30:02.000  IBM   10.75   100   1.860465
09:30:03.000  AIG   10.75   400          0
09:30:04.000  MSFT  10.45   100        100
09:30:05.000  IBM   10.35   100  -3.864734
09:30:06.000  MSFT  10.75   400   2.790698
```

The pct value for the first or only occurrence of a sym is 100.00. Otherwise the value of pct is the change in the price from the previous price, expressed as a percentage of the current price. Taking the sym `IBM as an example; the value of pct in the third row is difference of the values of price in the second and third rows shown as a percentage of the value of price in the third row.

Stored Procedures

Server functions and procedures are written in q. For example, a server function to subtract a client table p from a server table P is f:{[p]P-:p}
For more detailed information on q functions see the section Developing Analytics in Q

This needs lots of things.
For a start, from the Q ref manual – section 22 – defined functions. We need at least that much.

Table Arithmetic

It is common to want to do arithmetic with tables. The simplicity of q's arithmetic is in stark contrast to the convoluted coalescing and outer joins needed in sql.

```
q) table1:([sym:`AIG`IBM]size:200 400)
q) table2:([sym:`AIG`MSFT]size:600 700)
q) table3:table1+table2
q) show table1
sym         | size
----------  | ----
AIG         | 200
IBM         | 400
q>show table2
sym         | size
----------  | ----
AIG         | 600
MSFT        | 700
```

table3

```
sym         | size
----------  | ----
AIG         | 800
IBM         | 400
MSFT           | 700
```

Similar results are obtained for the other arithmetic dyadic functions. q will attempt to perform the arithmetic operation on each of the non primary key fields in the tables, you should ensure that such operations make sense, for example you will not want to attempt addition of two varchar fields

Joins

Joins generally run at 10 to 100 million records per second. Joins do not require there to be foreign key relationships predefined, in that case you can use use virtual columns with dot notation as above in the section on foreign keys.

Simple Join

This is the most basic type of join, performed with a comma ‘,’. In this case the two tables have to be type conformant, in other words have the same columns in the same order, and the same key if necessary. This is also used for in- and upserting. Conveniently, ‘,:’ can be used to automatically reassign a table when joining, for example

```
table,:table2            /table is assigned the value of table2
```

Asof Join

As implied by the name Asof Join (aj) can be used to get the the value of a field in one table asof the time in another table, (the bid price asof the time of the last trade for example). Asof Join has been abstracted so that we can get the value of a field in one table asof the value of *some* field in another table. This is shown in the example below, where time does not come into it

```
q) table1:([]a:(1 1 3 1);b:(4 5 4 4);d:(7 8 9 6))
q) table2:([]a:(1 2 3);b:(4 5 6);c:(7 8 9))
q) show aj[`a`b;table1;table2]
a  b  d  c
------------
1  4  7  7
1  5  8  7
3  4  9
1  4  6  7
```

Notice how a null valued is filled in for c where table2 doesn’t contain a row with (3,4) as the a and b values. At present asof joins can only be performed on one or two columns

Left Join

Left Join (lj) is a special case of aj where the second argument is a keyed table and the first argument contains the columns of the right argument’s key. The result is a table with columns of both tables as in the following example

```
q) table1:([]a:(1 1 3 1);b:(4 5 4 4);d:(7 8 9 6))
q) table3:([a:(1 2 3);b:(4 5 6)];c:(7 8 9))
q) show  table3 lj table1          /or if you prefer    show lj[table1;table3]
```

```
a b d c
------------
1 4 7 7
1 5 8
3 4 9
1 4 6 7
```

Union Join

Union Join (uj) allows you to create a union of two tables with distinct schemas, this creates null values in columns in the result, as in the example below (where table1 and table2 are as above).

```
q) show table1 uj table2
a b d c
------------
1 4 7
1 5 8
3 4 9
1 4 6
1 4   7
2 5   8
3 6   9
```

if using uj on keyed tables the primary keys must match.

Need to explain dot notation first. It’s not really done anywhere in the doc. Include some basic examples

Joins in q are much easier to implement than the equivalent joins in SQL, as the folowing example from the kx.com website illustrates.

```
There are 8 tables. we've changed the names to reduce code mass.
 (c.. various comment fields)

 l - lineitem(o,p,s,v(flag),u(status),mode(h),shipdate(ds),commitdate(dc),
      receiptdate(dr),q(quantity),x(price),xd(discount),xt(tax))
 o - order([o]c,d,j(priority),k(clerk),i(shippriority))
 c - customer([c]name,n,x(acctbal),m(market),c...)
 p - part([p]name,b(brand),t(type),z(size),e(container),x(price),c...)
 s - supply([s]name,n,x(acctbal),c...)
 n - nation([n]name,r)
 r - region([r]name)
 ps - partsupply([p,s]x(cost))

tpcd queries can also be written in q.(execute at the same speed)

q is simpler/more expressive than sql, e.g., query 8 (8-way join)

revenue share of suppliers(s) in BRAZIL by order(o) year to customers(c) in region AMERICA
in 1995 and 1996 for parts(p) of type(t) 'ECONOMY ANODIZED STEEL'.

q:

select rev avg s.n=`BRAZIL by o.d.year from l where
 o.c.n.r=`AMERICA, o.d.year in 1995 1996, p.t=`$"ECONOMY ANODIZED STEEL"

sql:

select year,sum(case when name='BRAZIL' then rev else 0 end)/sum(rev) from(
  select extract(year from o.d)as year,l.x*(1-l.xd) as rev,n2.name
  from p,s,l,o,c,n n1,n n2,r
  where p.p=l.p and s.s=l.s and l.o=o.o and o.c=c.c and c.n=n1.n and n1.r=r.r and r.name='AMERICA'
   and s.n=n2.n and o.d between date'1995-01-01' and date'1996-12-31' and
  p.t='ECONOMY ANODIZED STEEL')t
```

```
group by year order by year
```

Parameters

Functions can have queries and queries can have functions, e.g. given the following table layouts for trade and quote, we can have:

Trade

```
date            sym price size
----------------------------------------------
2006.06.01      AIG 59.25 2000
2006.06.02      AIG 59.5  1000
2006.06.03      IBM 54.25 200
2006.06.04      AIG 59.25 2000
2006.06.05      IBM 60.1  400
```

Quote

```
date         sym   bid  ask
-------------------------------------
2006.06.01   AIG   59.4  60.9
2006.06.02   AIG   59.2  60.1
2006.06.03   IBM   52.7  55.0
2006.06.04   AIG   53.0  57.3
2006.06.05   IBM   59.2  60.6
```

```
q) mid:{[t;s]exec .5*(bid+ask)time bin t from quote where sym=s}
```

then

```
q) f:{[s]select sum price>mid[time;s]by date from trade where sym=s}
```

will calculate how many times per day the trade price was higher than the midpoint for sym s.

Extract from Denis Shasha primer: Q's Semantic Extensions to SQL

Q views tables as a set of named columns having order. (Because their data representation resembles arrays and their use of named columns suggests tables, I like to describe them using the neologism *arrables*, but I'll use table in the rest of this primer.) The order allows a class of very useful aggregates that are unavailable to the relational database programmer without the cumbersome and poorly performing temporal extensions.

Note:

In these examples, we make use of *uniform* functions, both built-in (deltas and mins) and user-created (myavgs). A uniform function applies to one or more lists of the same length L. The output list has length L. The arithmetic plus operator is uniform. *Atomic* operators like plus are special because the element at each position p of the output depends on the elements of the inputs at position p and on them alone. Uniform functions don't make that restriction. For example, consider the running minimum function the value of the running minimum function (mins) at position p depends on all elements of the input up to position p. Here is a table of the built-in uniform (and non-atomic operators in q) slightly simplified from Don Orth's manual:

```
Func    Example         Result
--------------------------------------------------------
sums    sums 1 2 3 -4 5    1 3 6 2 7
deltas  deltas 1 2 3 -4 5  1 1 1 -7 9
prds    prds 1 2 3 -4 5    1 2 6 -24 -120
ratios  ratios 1 2 3 -4 5  1.00 2.00 1.50 -1.33 -1.25
mins    mins 1 2 3 -4 5    1 1 1 -4 -4
maxs    maxs 1 2 3 -4 5    1 2 3 3 5
```

Copy the following to a file (e.g. frenchtrade.q) keeping indentations as is. Then invoke q on that file (e.g. q frenchtrade.q).

Examples

```
/ Create a list of French stocks where name is the key.
stock:([name:`Alcatel`Alstom`AirFrance`FranceTelecom`Snecma`Supra]
 industry:`telecom`engineering`aviation`telecom`aviation`consulting)
/ get a list of distinct stocks are there?
stocks: distinct exec name from stock
ns: count stocks
```

```
n:10000
/ stock is a foreign key foreign key.
/ We are taking n stocks at random.
/ Then n prices up to 100.0 at random, then n random amounts
/ then n random dates.
trade:([]stock:`stock$n?stocks;
price:n?100.0;amount:100*10+n?20;exchange:5+n?2.0;date:2004.01.01+n?449)
/ Sort these in ascending order by date.
/ We will need this to make the following queries meaningful.
`date xasc `trade
/ Now find the dates when the price of Snecma went up
/ regardless of time of day.
/ The deltas function subtracts the previous value in a column with
/ the current one.
select date from trade where stock=`Snecma, 0 < deltas price
/ Find the average price by day.
aa: select aprice: avg price by date from trade where stock=`Snecma
/ An aside: Find the weighted average price by day where prices associated with
/ bigger trades have more weight.
/ wavg is a binary function that takes two columns as arguments:
select wavg[amount;price] by date from trade where stock=`Snecma
/ Here is an infix form giving the same result.
select amount wavg price by date from trade where stock=`Snecma
/ Find the dates when the average price went up
/ compared with the previous day.
select date, aprice from aa where 0 < deltas aprice
/ Suppose we wanted to do the above but for every stock.
/ Basically we replace the where stock= part by putting stock
/ into the by clause.
/ First we get the average price for each stock and date.
aaall: select aprice: avg price by stock, date from trade
/ Now find dates having rising prices for each stock
/ Note that stock is the key of each row and there is a vector
/ of dates and aprice associated with each stock.
xx: select date, aprice by stock from aaall where 0 < deltas aprice
/ See which are those dates for Snecma (same as before)
select date from xx where stock = `Snecma
/ See which are those dates for Alcatel
select date from xx where stock = `Alcatel
/ Suppose that we do this on a monthly basis.
/ Note that the date arithmetic is very flexible and
/ that a field is created called month by default from the by clause.
aaallmon: select aprice: avg price by stock, date.month from trade
xxmon: select month, aprice by stock from aaallmon where 0 < deltas aprice
select month from xxmon where stock = `Snecma
/ Here we do a compound statement.
/ The idea is to find the profit of the ideal transactions for each stock.
/ An ideal transaction is a buy on a day x followed by a sell on day y (x < y)
/ such that the sell - buy price is maximal.
/ To do this, we want to find the time when the difference between
/ the actual price and the minimum of all previous prices is greatest.
/ Read this as follows: compute the running minimum of prices.
/ This gives a vector v1 of non-increasing numbers.
/ Then consider the vector of prices v2.
/ Find the value where v2-v1 is maximum.
bestprofit: select best: max price - mins price by stock from trade
/ One of the very powerful features of q (shared with KSQL) is that
/ a programmer can add his own procedures to the SQL and things just work.
/ Let's start with something simple:
mydouble:{[x] 2*x}
select mydouble price from trade where stock = `Alcatel
/ Some functions can take several arguments. Suppose that we were
/ interested in the moving average of double the prices.
myavg:{[x] (sum x) % (count x)}
/ Therefore, I invent a binary operator myavg and then take
/ the running averages ending at each point in the vector.
/ The vector is prefilled with n 0s.
/ Reasonable people may disagree about this initialization strategy.
myavgs:{[n;vec] x: (n # 0), vec; each[myavg]x[n+(key (count vec)) -\: key n]}
select myavgs[3; mydouble price] from trade where stock = `Alcatel
/ Or we could do this for every stock.
select myavgs[3; mydouble price] by stock from trade
/ Even better, these procedures can go into any clause.
select date by stock from trade where 70 < myavgs[3; mydouble price]
select date by stock, myavgs[3; mydouble price] from trade
/ Functions can contain sql. The only limitation is that you
/ may not include field names in an argument.
f:{[mytable] count select from mytable}
f[stock] / returns 6
f[trade] / returns 10000
/ The last feature we want to introduce is the fact that q
/ can store a vector in a field of a row.
/ This non-first normal form capability can contribute to performance.
t: select date, price, amount by stock from trade
/ In the browser window this gives:
/ stock         date  price  amount
/ ---------------------------------
/ Alcatel        ..    ..     ..
/ Alstom          ..    ..     ..
/ AirFrance       ..    ..     ..
/ FranceTelecom  ..    ..     ..
/ Snecma          ..    ..     ..
/ Supra           ..    ..     ..
/ This means that we get a vector of dates for Alcatel as well
/ as a vector of prices and amounts.
/ Now unary functions work using each meaning for each row.
select stock, each[first] price from t
select stock, first each price from t / these two are equivalent
/ Now suppose that for each stock, we want the volume weighted price.
/ That is a binary operator so the each becomes a '
select stock, amount wavg' price from t
select stock, wavg'[amount;price] from t / these two are equivalent
/ We can go to higher level of valences by using the second syntax.
f:{[x;y;z] (avg x)*y-z}
select stock, f'[price; amount; price] from t
```

Q as an extension of SQL

q is an the next generation of k and has been improved in a number of ways (e.g. the unary operators are no longer overloaded thus reducing ambiguity). Its major benefit is that it gives tight integration with table functionality. q provides a powerful dialect of SQL with additional unique and powerful features.

The following table illustrates how to perform common sql things in q. There is no reverse map since q is a much richer language than sql. (With respect to CJDate "The SQL Standard")

SQL	**Q**
create table s(s varchar(*)primary key,name varchar(*),status int,city varchar(*))	s:([s]name;status;city)

create table sp(s varchar(*)references s,p varchar(*)references p,qty int)	sp:([]s:`s$();p:`p$();qty:())
insert into s values('s1','smith',20,'london')	`s insert(`s1;`smith;20;`london)
update s set status=1+status where s='s1'	s:update 1+status from s where s=`s1
delete from s where s='s1'	s:delete from s where s=`s1
select * from s where s='s1'	select from s where s=`s1
select s,count(*)as x from sp group by s order by s	select count i by s from sp
select s,sum(qty)as qty from sp,s,p where sp.s=s.s and sp.p=p.p and p.city=s.city group s order s	select sum qty by s from sp where s.city=p.city
select distinct ..	select distinct ..
count sum min max avg	count sum min max avg prd first last wavg wsum
+ - * / < > = <= >= <> like between-and not and or	+ - * % < > = like within not & |

Extract from Denis Shaha Primer: Table Operations: the SQL dialect

Note that the first line of SQL statements must be outdented and the rest must be indented.
The basic syntax is:

```
select ..
 by ..
 from ..
 where ..
```

As of this writing (July 2004), the from clause contains only one table, but this does not prevent joins from happening, provided they are foreign key joins. To understand foreign keys, please consider the following example. A publisher has many authors. It holds author information in one table: author(author, name, address) where author is the key. Each author may have several books in print. bookauthor(book, author, ...) where book and author together are a key (so that we can handle the case that a book has several authors and that an author has written several books). Now, the publisher wants to send notices, checks, reviews and other good news about a book to its authors. For this to be possible, every author in the book table must have an entry in the author table. We say that "book.author is a foreign key for author". It's foreign because it's in another table and it's a key because author is a key of the author table (i.e. no two rows can have the same authorid). With that understanding, we can present the following script:

Examples N: Key tables

```
/ This example must be loaded as a file.
/ So copy it to a file, say foo.q.
/ Be careful to keep the indentations.
/ First line must not be indented; others must be.;
/ Then type
/ q foo.q
/ Reason: indents don't work within the interactive window as of July 2004.
/ Remember that keys are surrounded by brackets
author:([author:`king`hemingway`flaubert`lazere`shasha]
 address: `maine`expat`france`mamaroneck`newyork;
 area: `horror`suffering`psychology`journalism`puzzles)
book:([book:`forwhomthebelltolls`oldmanandthesea`shining`secretwindow`clouds`madambovary`salambo`ou
toftheirminds]
 language: `english`english`english`english`english`french`french`english;
 numprintings: 3 5 4 2 2 8 9 2)
/ Here we indicate that the author field is a foreign key to the table
/ author and book to the table book.
/ If we wished, we could also surround the two fields author and book
/ by brackets to indicate that they are keys.
```

```
bookauthor:([]
 author:`author$`hemingway`hemingway`king`king`king`flaubert`flaubert`shasha`lazere;

book:`book$`forwhomthebelltolls`oldmanandthesea`shining`secretwindow`clouds`madambovary`salambo`outoftheirminds`outoftheirminds;
 numfansinmillions: 20 20 50 50 30 60 30 0.02 0.02)
/ SQL 92: select * from bookauthor
select from bookauthor
/ SQL 92: identical to this except that we use the symbol notation
select numfansinmillions from bookauthor where book=`forwhomthebelltolls
select author,numfansinmillions from bookauthor where book=`forwhomthebelltolls
/ Implicit join via the foreign key.
/ SQL92:
/ select bookauthor.author, book.language
/ from bookauthor, book
/ where book.book = bookauthor.book
/ and numfansinmillions< 30
select author,book.language from bookauthor where numfansinmillions< 30
/ Same idea, but note the outdented first line followed
/ by the indented later lines.
/ SQL92:
/ select bookauthor.author, book.language, author.address
/ from bookauthor, book, author
/ where book.book = bookauthor.book
/ and author.author = bookauthor.author
/ and author.area = "psychology"
select author,book.language,author.address
 from bookauthor
 where author.area = `psychology
/ Notice that the distinct is to the left of the select.
/ SQL92:
/ select distinct bookauthor.author, book.language, author.address
/ from bookauthor, book author
/ where book.book = bookauthor.book
/ and author.author = bookauthor.author
/ and author.area = "psychology"
distinct select author,book.language,author.address
 from bookauthor
 where author.area = `psychology
/ Here we are doing implicit joins and also an implicit groupby
/ SQL92:
/ select book.language, sum(numfansinmillions)
/ from bookauthor, book
/ where bookauthor.book = book.book
/ group by book.language
select sum numfansinmillions by book.language from bookauthor
```

Database Administration

Database Layout

The optimum layout for a kdb+ database depends on the volume of data it will hold. Broadly speaking, they are divided into 3 categories; small, medium or large. The upper limits for each can be seen here:

	32 bit	**64 bit**	**Storage**	**Query Speed**
Small	1GB	10GB	Single files	10µs
Medium	32 million rows	512 million rows	Single directory	0.3ms
Large	Unlimited	Unlimited	Multiple partitions	10ms per date/sym

Small Databases

Tables or databases are saved as individual files. Query times are around 10µs. Tables can be nested (as in kdb+tick) or flat.

Medium Databases

Tables are stored splayed in a single directory.

Large Databases

Databases are:

trade `p#sym

quote `p#sym

Queries run from disk at 10ms per date/sym/field.

The partition field is date month year or int and is virtual.

Kdb+taq is more than 2500 partitions of daily trades and quotes. Each new day is more than 1GB. [Queries on partitioned databases can run in parallel.] To set a subview -- handy for test and development:

```
.Q.view 2#date / view two first partitions
.Q.view[]      / reset
```

Logs

For any recoverable application, it may be desirable to log update operations to disk. This can be achieved by starting the database with a –l (asynchronous logging) or –L (synchronous).

Synchronous Logging operates at approximately 100 records per second with SCSI disk and 10,000 times per second with SSD (Solid State Disk).

Asynchronous Logging achieves rates of around 100,000 transactions per second. It is useful if the storage is trusted or duplicated.

In either case, a single drive can commit more than 40MB of updates per second.

```
> q db –l
```

This starts the database in asynch logging mode. It loads up the file “db.q” and the database “db.qdb”. The log goes in a file called “db.log” in the same directory. Any of these can be empty.

All update messages are logged. To send a log message from the console use 0. (e.g. 0"v+:1") An error after a partial update will cause a rollback.

```
q) \l
```

This causes db.qdb to be checkpointed and db.log to be emptied.

Nested Databases

A database directory can have a number of tables and scripts.. The scripts are loaded after all the tables are loaded. Tables themselves can be single files or spread across a directory. (Although keyed indicative tables must be single files.)

The easiest way to start a nested database is just to pass the path and name of the directory holding the data and script files.

```
> q /home/data/dir / loads the contents of dir
```

Parallel Databases

Large databases
>q dir / loads parallel database and sets partition variable.
dir/{..|{date|month|year|int}/{tables}/{fields}}

There is one partition variable date|month|year|int and it is a virtual column in all the partition tables. There can be other date columns with different names. New partitions can be added without having to restart the database – send “\\l .”

Loading Tables

From Textfiles

The general form for loading from a textfile is:

```
(types;delimiter) 0: filename
```

or

```
(types;widths) 0: filename
```

Enlist the delimiter if the first row of the file is the column names.
Supply a one-character type for each field in the file. A blank tells the interpreter to skip a field; an asterix is used for a variable type. See datatypes table above for what each letter corresponds to.

Examples:

```
tbl:("BXHSMDZUVT* "; enlist ”|”) 0: `:file  / pipe-delimited text with column names in first row
tbl:("CCIISS  FF"; “,”) 0: `:file            / comma-separated text without column names
tbl:("CCIIF";10 10 4 4 7) 0: `:file                    / fixed-width textfile
```

From Data

```
trade:get`:trade                           / read in q datafile
trade:get`:trade/                          / map in splayed table
```

equivalent to:

```
load `trade                                / load table into memory
```

Currently, it is possible to use the rload function to ‘map in a splayed table,’ but its use is discouraged as it is a deprecated function.

```
rload `trade
```

```
L:("bxhijefcsmdzuvt* ";widths)1:F          / fixed-width data file
```

Syms(S) trim, blank skips and * is character vector. Reverse types and widths to load reverse endian data.

```
data: .[`:f[/];();[:,];..] !`:d .`:d/(defermap) stdout: -1".." stderr: -2".."
line: f 0:L:read0:f[,i,n] L:d 0:L:("*BXHIJEFCSMDZUVT ";[,]delim)0:{L|f[,i,n]}
byte: f 1:G:read1:f[,i,n] G:w 1:L:("*bxhijefcsmdzuvt ";   width)1:{G|f[,i,n]}
```

Saving Tables

Tables can be written as a single file or spread across a directory, e.g.

```
`:c:/data/trade  set tbl      / write table as a single file called “trade”
`:c:/data//trade/ set tbl     / write as a splayed table in a directory called “trade”
```

or

```
save `tbl                     / save as a single file
```

Currently, it is also possible to use a similar rsave keyword, but its use is discouraged as it is deprecated.

```
rsave `tbl                    / save as a splayed table
```

Tables splayed across a directory must be fully enumerated (no varchars) and not keyed.

They can also be saved as csv’s or in text format:

```
save `tbl.csv
save `tbl.txt
```

Datetime txt and csv are written as: yyyy-mm-dd hh:mm:ss.ddd

Developing analytics in q

Defined Functions

The General Form

The general form of defined functions is

{[argument list] body}

The argument list is a sequence of names separated by semi-colons. The body is a list of expressions and control statements separated by semi-colons. The items in the body are executed strictly from left to right, as are items within control statements. For example, a function that computes the amount of interest paid in terms of principal, rate and time can be defined as follows:

```
q) interest:{[principal;rate;time] principal*rate*time}
q) interest[100;.05;.5]
2.5
```

This function has 3 arguments, principal, rate and time. The expression that follows does the evaluation.

The argument list for a monadic function is one name between square brackets (no semi-colons). It is possible to define a function with no arguments by specifying the empty list, []. The empty argument list can be omitted, but if so, default arguments may come into play.

Default Arguments

The argument list can be omitted when default arguments are used. The default arguments are x, y and z. If any of these names appears in a function definition that has no argument list then that name is a function argument. Moreover, z is always the third argument, whether or not x or y is present in the function definition. Similarly, y is always the second argument and x is always the first. For example,

```
q) f:{z*z}
 q) f[1;2;3]     / f is a function of 3 arguments
9
 q) f[`a;"b";3]  / the first two arguments aren't used; they can have any values
9
```

Whether or not you use default arguments is a matter of taste. In general, they are useful in general utility functions whose arguments can't be named in meaningful ways.

Default Result

The value of the right-most expression in a function body is the default result if there is no semi-colon between it and the closing}. However, functions always have some kind of results. Even in a case below where nothing is displayed when the function is evaluated, such as

```
  f:{.cx.global:0;}
  f[]
  _    / the default prompt
r:f[]  / this expression will not fail
```

Consequently, we say that this function has no explicit result.

Localization

All unqualified names in a function definition in which there is at least one simple assignment of the form name:value are local to that function. A local name has no value in references before an assignment is made, and any assignment to a global with that name, e.g. name::value, is treated like the local assignment name:value . For example,

```
q) a:5         / global a
q) f:{[a;b]a+b}
q) f[3;9]        / a is local
12
q) a
5               /global a remains unchanged
q) g:{[a,b]a*b}
q) g[a::7;9]   /a is now global
63
q) a          / global a is 7 and not 5
7
```

Projections

For functions of valence at least two, when any of the argument positions are left blank in an evaluation expression, i.e. when those arguments are not specified, the effect is to create a new function in which the specified arguments have those fixed values. The new function is called a projection onto those fixed values. For example:

```
q) f:{x+y+z}
q) f[7;9;11]
27
q) g:f[7;;9] / function is now effectively 7 + y + 9
```

The monadic function g is the projection of f onto its first and third arguments. The function g is formed by fixing the first and third arguments of f to be 1 and 3, respectively. That leaves one argument unspecified, which is the argument of the function g. Continuing,

```
q) g[65]  / apply g to the argument 2
81
q) g[1]  /apply g to the argument 1
17
```

Trailing arguments that are left unspecified can be left out of the projection expression. For example, the function h, which has valence 2, is equivalent to f[1;;]

```
q) h:f[1]
q) h[2;3]
6
```

Projections of projections are equivalent to projections of the original function. For example, e, which has valence 1, is equivalent to f[1;;3]

```
q) e:h[;3]
q) e[2]
6
```

Function projections retain their definitions even when their source functions are redefined. For example,

```
q) g 2
18
q) f:{x}
 q)g 2
18
```

It is also possible to project onto the left argument of a verb. For example, 3+ is a monadic function.

```
q) (3+)10
13
```

In this example 10 is the argument to the monadic function 3+; function evaluation is by juxtaposition, which in this case is equivalent to (3+)[10]. It is not possible to project on the right argument of + in infix form, but it can done if function evaluation notation can be used for +.

```
 q) +[;10]3
13
```

Projections in Practice

Projections are useful when many different evaluations of a function have some number of common arguments. For example, in kdb+ the trade table was constructed with a series of table inserts of the form

```
insert[`trade](09:30:01.000;`aaa;53.75;1200)
```

Insert is a dyadic function whose first argument is the table that receives the data in the second argument. That is, the following is an equivalent expression.

```
insert[`trade;(09:30:01.000;`aaa;53.75;1200)]
```

In the first expression, the subexpression insert[`trade] is a projection of the insert function onto the first argument. The expression is the application of the monadic function insert[`trade] to the argument (09:30:01.000;`aaa;53.75;1200) using juxtaposition. Whether to use the style of the first or second expression is a matter of taste. Some programmers find the first expression easier to read.

Parameterized Queries

Select, update and delete expressions can be evaluated in defined functions. Every name except column names can be a function argument, local variable or global variable (think of column names as local within the phrases). For example, using the tables from earlier examples:

```
q) industry:([sym]ind:())
q) `industry insert(`AIG;`Insurance)
q) `industry insert(`IBM;`Consulting)
q) trade: ([]time:();sym:`industry$();price:();size:())
q) `trade insert(09:30:01.000;`AIG;59.25;2000)
q) `trade insert(09:30:02.000;`AIG;59.50;1000)
q) `trade insert(09:30:04.000;`IBM;54.25;200)
q) trade
q) vol:300
q) f:{[tbl] select from tbl where size>vol}
 q)  f[trade]
+`time`sym`price`size!( 09:30:01.00009:30:02.000;`industry$`AIG`AIG;59.225 59.50; 200 1000)
```

Default arguments cannot be used in parameterized queries.

Execution Control

There are various methods of contolling execution. These are described with examples in the table below.

Construct	Details and Example
Conditional Evaluation	$[condexp;truexp;falsexp]

The simplest form of Conditional evaluation is denoted by $[condexp;truexp;falsexp] . For example,

```
  f:{$[x
```

Using a function in this example was simply a matter of convenience. The conditional expression can have any atomic integer value (e.g., bool or int); truexp is executed if that value is not 0 and otherwise falsexp is executed. The result of the conditional expression is the result of whichever sub expression is executed.

There are two extended forms of conditional evaluation. In one case there can be more than one conditional expression. The general form is

```
$[condexp1;truexp1;...;condexpn;truexpn;falsexp]
```

For example,

```
  f:{$[x<10;`abc;x
```

More generally, any "true expression" or "false expression" in the above forms can be replaced a block of expressions enclosed in square brackets. For example,

```
  f:{$[x
```

The individual expressions in a bracketed set such as [t:5;u:7;t+u] are executed left to right and the rightmost one gives the result of the set.

Protected Evaluation

Errors do not halt execution

It is possible to execute a function in such a way that an error does not halt execution. This is particularly useful when executing user input, such as ad hoc queries. Like Amend, there is a Dot version and an At version.

The primitive value function is useful for executing user input. Here is an example of a protected execution of value.

```
  f:{@[value;x;`fail]      /The 1st argument is the function, the 2nd is its arguments and the third is the returned argument.
  f "2+3"
5
  f "2 2+3 3 3"
`fail
```

Protected evaluation can be used when the atoms are not known to be convertible. The general form of the previous expression was

```
c[i]:(type c)$v
```

A protected evaluation version is

```
c[i]:.[$;(type c;v);c -1]
```

If the cast fails, the result of the protected evaluation will be the result of an out-of-range indexing selection from c, which is the null value for the type of c. Repeating the previous example,

```
  c:10 345 -20 11
  c[2]:.[$;(type c;0xab);c -1]
  c[0]:.[$;(type c;23h);c -1]
  c
23 345 171 11
```

Continuing with an atom that cannot be cast to an int,

```
  .[$;(type c;`abc);c -1]
0N
  c[1]:.[$;(type c;`abc);c -1]
  c
23 0N 171 11
```

Do statements

do[count; expr]
do[count; expr1;...; exprN]

```
do[count; expr]
do[count; expr1;...; exprN]
```

The first Do statement executes the expression count times. The second do statement executes the expression list count times. The expressions in the expression list are executed left to right. If count is an expression then it is executed first and that value determines the number of times the other expressions are executed. Do statements do not have explicit results.

If statements

while[cond; expression]
while[cond; expression1;...; expressionN]

```
if[cond; expr]
if[cond; expr1;...; exprN]
```

The conditional expression is like the one in Conditional Evaluation. If its result is not 0 then the expression or expression list is executed. In the second case the expressions are executed left to right. If statements do not have explicit results.

While statements

```
while[cond; expression]
while[cond; expression1;...; expressionN]
```

The conditional expression is like the one in Conditional Evaluation. As long as its value remains not equal to 0, the expression or expression list is executed. When there are more than one expressions they are executed left to right. While statements do not have explicit results.

An example earlier in the context of nulls provides an example of a While statement. In that section, the function

```
f:{r:x;r[i]:r[-1+i:where null r];r}
```

was applied repeatedly to a list to replace all nulls with the nearest non-null value to the left. The example list was

```
v:10 -3.1 0n 0.1 0n 0n 0n 3.4
```

To test for nulls in v, we can use max and the Null function, as follows.

```
  null v
00101110b
  max null v
1b
```

The interation in Processing Nulls can then be carried out as follows.

```
  while[max null v;v:f v]
  v
10.00 -3.10 -3.10 0.10 0.10 0.10 0.10 3.40
```

return

r' exits a function with a result value

This is a primitive function which alters the default execution flow in a defined function by exiting with a result value. Return is denoted by :r . For example, the following function returns the sym atom `other if the argument is not an atom.

```
  f:{if[not 0>type x;:`other];x}
  f 2 3
`other
```

signal — 'r exits a function with an error

This is a primitive function which alters the default execution flow in a defined function by exiting with an error signal. This function is denoted by 'r. For example,

```
  f:{if[not 0>type x;'`other];x}
  f 2 3
{if[not 0>type x;'`other];3}
'other
> _
```

Signal is useful for debugging and processing user input. In the above example, the reason for the signal can be analyzed by using debugging techniques. The function f could also be called with user input under protected execution by another function. The signal from f would indicate to the caller that the message "other" was an error message, which could be reported to the user.

Debugging — Example guidelines

If a defined function fails you will see a console display of the failed primitive function and it's argument(S). The console prompt becomes a "> ". You can examine the arguments and local variables of the function that failed. For example,

```
  f:{[a] b:`x`y;b[0]+a*2}
  f 4                      / a type error is generated
{[a] b:`x`y;b[0]+a*2}      / the function in which the failure occurred
'type                      / the error type
+                          / the function that failed
`x                         / the left argument
8                          / the right argument
> _
```

The underbar in the last line indicates a blinking cursor. You can see from the console display that the failure is a type error of the function +, applied to the arguments `x and 8. In this simple example you can see immediately what the problem is, but it's not always that easy. When it's not, you can examine the arguments and locals in the function. For example,

```
>  a
4
>  b
`x`y
> _
```

Enter a back-slash to abort the execution. You will then see the default cursor.

```
> \
 _
```

Let's look at a slightly more complicated example.

```
  f:{[a;b] b[0]+a*2}
  g:{[c] d:`a`b;f[c;d]}
  g 3
{[a;b] b[0]+a*2}
'type
+
`a
6
> _
```

Again, we see the definition of the failed function. We can examine its arguments.

```
>  a
3
>  b
`a`b
> _
```

The problem is clearly the argument b. We can signal up with quote to see where this function is called.

```
>  '                 / signal up
{[c] d:`a`b;f[c;d]}  / clearly, the function in which the failure occurred is f
'                    / the error type (signal)
{[a;b] b[0]+a*2}     / the function that failed
3                    / the first argument
`a`b                 / the second argument
> _
```

We can examine the argument and the local variable in g.

```
>  c
3
>  d
`a`b
> _
```

We can now see that the local d is the problem. Enter \ to abort or ' to signal up and out and fix the definition of function g.

Extract from Denis Shsha Primer: Execution Control

Every programming language since Algol has offered if-then, if-then-else, and while. Q is no exception, though good q programmers tend not to need these functions (especially looping functions) as much as programmers in scalar languages.

```
/ $[condexp1;truexp1;...;condexpn;truexpn;falsexp]
/ if then-else. The expressions can be surrounded by []
/ to connote a block.
val: 35
$[val > 60; val; val < 30; 0; [x: neg val; 2*x]] / returns -70
/ This can be rendered by several if statements.
/ Personally, I like these because they make the assumptions clear.
if[val > 60; out: val]
if[val < 30; out: 0]
if[not (val > 60) | (val < 30); out: 2 * neg val]
out / returns -70
/ compute the square of each number up to n then subtract 1
squareminusone:{[n] out: (); i: 0; while[i < n; out,: (i*i)-1; i:i+1]; out}
squareminusone[5] / -1 0 3 8 15
/ The same can be done without a loop
/ This will normally be faster.
squareminusonealt:{[n] ((key n) * (key n)) - 1}
squareminusonealt[5] / returns -1 0 3 8 15
```

Inter-Process Communication

A Kdb+ process is a server. The port is specified with the -p option in the startup command. For example, the following process will have the port 5001.

```
q sp.q -p 5001
```

The port can also be specified in a script or in the console with the command \p; see Commands.

Kdb+ Data Client

We have already used the port in Kdb+ for executing q queries and displaying results in a web browser. We will now look at Kdb+ data clients.

Opening and Closing a Connection

A kdb+ client can connect to a kdb+ server process on the same computer, on the same network, or remotely. The client uses the Hopen function to connect. For example, suppose the server listening on port 5001 is named srvr and is on the same network as the client. In that case the client connects to the server as follows.

```
h:hopen`:srvr:5001
```

The symbol `:srvr:5001 is called the communication handle; the int h is called the connection handle. There is no need to identify the server if the client is running on the same machine. The communication handle in that case can simply be `::5001. A remote server is identified by its IP address, say 66.108.242.81, in which case the communication handle is `:66.108.242.81:5001.
An open connection is closed with the hclose function, as in

```
hclose h
```

Asynchronous and Synchronous Messages

There are two ways to send messages to a server, asynchronously and synchronously. An asynchronous message does not return a result. The expression that sends an asynchronous message completes as soon as the expression is executed (which may be a little sooner than when the message is sent). Asynchronous messages are also called set messages because typically they cause changes in the state of the server. For example, a delete or insert statement can be sent asynchronously.

A synchronous message expects a response from the server. The expression that sends a synchronous message waits for the response and returns it as its result. For example, a select statement must be sent as a synchronous message. Synchronous messages are also called get messages, since they expect a response.

There are two forms of messages. One is a character string holding an executable expression. For example,

```
"select avg price by time.u from trade"
"insert[`trade](10:30:01;`dd;88.5;1625)"
```

The second form of a message is a list. The first item of the list is a char string or sym atom holding the name of a function (i.e., stored procedure) to be executed on the server. The other items are the arguments of the function call, in order. For example, the above insert message can also be phrased as a remote procedure call, as follows.

```
(`insert;`trade;(10:30:01;`dd;88.5;1625))
```

A synchronous message is sent by executing the expression processhandle message and an asynchronous message is sent by

(-processhandle)message

The following examples send synchronous character string messages to the communication partner.

```
h"select avg price by time.u from trade"

u      price
-----------------
09:30  77.92
10:00  76.31
```

```
h"insert[`trade](10:30:01;`dd;88.5;1625)"
`trade
```

Since the insert message was sent synchronously the result (the name of the modified table) is returned. This confirms that the insert was sucessful. However, if the client doesn't require a response, the insert message can be sent asynchronously, as in the follow example.

```
(-h)"insert[`sp](`s1;`p1;400)"
```

Execution of this message returns immediately with no result.

The function-arguments list form of this insert message is

```
(-h)("insert";`sp;(`s1;`p1;400))
```

In most realistic situations the data to be inserted is not constant, but is either generated algorithmically or received from an external source. Consequently, the function-arguments message format is the more generally useful one because it does not require formatting the data into char strings.

Message Filters

The default behavior of a kdb+ server is to apply the Value function to incoming messages. This behavior can be modified with message filters. Message filters are monadic, user-defined functions with reserved names .z.ps for asynchronous (set) messages and .z.pg for synchronous (get) messages. The reason for two filters is that the asynchronous message filter is not expected to return a result - and even it did, the result would not be sent to the sender - while the synchronous filter is expected to return a result. The argument to each function is the complete message received from the sender.

There are many different uses for message filters. In a server that processes ad hoc queries the filters can use protected execution to avoid stopping on query errors. In so-called gateway servers, message filters route client messages to other servers. For example, select statements with a date specification in the where phrase can be routed to a real-time server or historical server. (How can the message filter know the date? The message can be in a stored procedure call, with a parameterized query and the date as arguments). Message filters can also maintain state while a sequence of related messages is processed.

Whenever a message is received the variable .z.w is automatically given the value of the server's connection handle to the sender. This value can be referenced in the message filters. The value can be used for various purposes. For example, if you maintain a list of connected partners and the current value of .z.w is not in the list then you know this is a new user, in which case authorization and initialization functions can be executed. For example,

```
if[not .z.w in cp;cp,:.z.w;init .z.w]
```

Evaluating Messages with the Value Primitive

The primitive Value function can evaluate either form of message. It applies to a simple char list holding a valid kdb+ expression, evaluates the expression as if it had been entered at a console, and returns the result of the evaluation, if any. For example,

```
 value"3+4"
7               / the value of 3+4

 value"m:2 3e"  / no value is produced in this case
 m              / this is the next input line; display the value of m
2 3e
```

The Close Handler

Either communicating partner can close a connection. A message indicating the close is printed in the console of the other partner unless that partner has a close handler. A close handler is a monadic function with the name .z.pc that is automatically called with the connection handle to the partner that closed the connection.

If a client closes a connection then the server's .z.pc can use the argument to remove that client from its client list. If the server unexpectedly closes a connection (the server crashes) then the client may try to reconnect. Typically this is done with a timer so that reconnection is attempted repeatedly until successful, or perhaps up to some maximum number of attempts.

In the following example, .z.pc resets the connection handle to 0 and sets the timer to 1 second. Note that the way to execute a command within a defined function is to apply the Value function to a char string holding the command text. The function .z.ts will then be called automatically every second. On each call the client attempts to reconnect. If successful, the connection handle will be assigned a positive int value. In addition, the if statement in .z.ts resets the timer to 0 if the reconnection attempt is successful.

```
.z.pc:{o::0;value"\\t 1000"}
.z.ts:{o::hopen`:srvr:5001;if[o>0;value"\\t 0"]}
```

Kdb+ HTTP Server

There is also a message filter for HTTP messages, with the reserved name .z.ph.

HTTP messages are always synchronous, so there is only one filter. This filter, which manages the Kdb+ Web Viewer, is the only one with a default definition. It can be replaced with a customized version.

Working with Files

Kdb+ distinguishes three types of files, kdb+ data files, text files and all others, which we will call binary files. Files are identified by file handles, which are sym atoms of the form

```
`:[path]name
```

For example,

```
`:c:/kdbtestdata is the file handle.
```

File handles, like communication handles, are sym atoms that begin with the character ":".

Kdb+ Data Files

Data in the Kdb+ workspace is self-describing, which is accomplished by appending the descriptive information to the actual data. A kdb+ data file is a binary file that is the image of a kdb+ workspace data object, including the descriptive information.

Kdb+ data can be written to kdb+ data files with Dot Amend. For example,

```
  .[`:c:/kdbdata;();:;1 3 -4 5 10 11]
`:c:/kdbdata
```

It is also possible to append to an existing file with Amend. For example,

```
  .[`:c:/kdbdata;();,;1 3 -4 5 10 11]
`:c:/kdbdata
```

Note the comma (Join) in place of colon (Assign). The file can be read by applying the Value function to the file handle, as follows.

```
  value`:c:/kdbdata
1 3 -4 5 10 11 10 23 -56
```

Each of these operations opens (or creates) the file, applies the operation, and closes the file. It is also possible to explicitly open a file, in which case it stays open until you close it. A file is opened and an open file handle is created in the same way as a connection handle. For example,

```
  h:hopen`:c:/kdbdata
```

Data can be appended to the open file with the same syntax for sending messages, i.e. either h[data] or h data .

```
  h 100 101
700
```

The returned value, which may be different when you repeat this example, is the value of h. This is analogous to the above Amend statement returning the file handle as its result. Any number of appends can be done with this open file handle, which can then be closed with the hclose function.

```
  h 1050 1100 1125
  hclose h
  value`:c:/kdbdata
1 3 -4 5 10 11 10 23 -56 100 101 1050 1100 1125
```

Kdb+ data files are used for transaction logging. When a transaction is appended to the log file with Amend, the file write is immediately synchronized to disk because the file is closed after the write. If an open file handle is used then the file write is not synchronized, which permits many more logging operations per second.

Tables

The above Amend expression, when applied to tables, writes an entire table as one file. It is also possible to write the table to disk as a directory whose file contents are the individual columns. That is, there is one file per column in the directory. This format of a table on disk is called splayed format. The Amend statement for writing a splayed table differs from the one above only in the file handle.

```
.[`:trade/;();:;trade]
```

It is also possible to write the entire contents of a database context to one file. In particular, the default context can be written to file namedcurrentdb as follows.

```
.[`:currentdb;();:;.`.]
```

Text Files

Text files are not Kdb+ data files and therefore the above applications of Amend do not apply. The primitive verb denoted by 0: takes a file handle as its left argument and a list of char strings as it right argument and writes the list to the text file with that handle. For example,

```
  `:f.txt 0:("abc";"defg")
`:f.txt
```

As in the Amend cases, the result is the file handle. New rows can be appended to an open text file in the same way as binary files.

```
  h:hopen `:f.txt
  h "hijkl";
  hclose h
```

The monadic primitive function read0 is used to read a text file.

```
  read0`:f.txt
("abc";"defg";"hijkl")
```

Binary Files

Binary files are managed in the same way as text files, using the verb 1: instead of 0:. From the Kdb+ point or view, binary files are lists of byte items. For example,

```
  `:f.bin 1: 0xabcdef
`:f.bin
```

creates the binary file f.bin with contents 0xabcdef . The following appends to the file,

```
  h:hopen`:f.bin
  h 0x01020304;
  close h
```

THe following reads the file.

```
  read1`:f.bin
0xabcdef01020304
```

Text files can also be read and written with read1 and 1: and managed as binary or char data. For example,

```
  read0`:f.txt
("abc";"defg";"hijkl")
  read1`:f.txt
0x6162630d0a646566670d0a68696a6b6c
  "c"$read1`:f.txt              / cast the data to char type
"abc\r\ndefg\r\nhijkl"
```

Specifying Field Types when Reading Files

There are two primitives for doing this, one for text files (0:) and one for binary files (1:). The field types are designated by letters, like the q datatypes. The letters are same for text files as q data, only capitalized. For binary files, the letters are lower case.

Text Files(0)	Binary Files(1)	Type	Data(1)	Text(0)
blank	skip			
B	b	bool	1	[1tTyY]
X	x	byte	1	
H	h	short	2	[0-9a-fA-F][0-9a-fA-F]
I	i	int	4	
J	j	long	8	
E	e	real	4	
F	f	float	8	
C	c	char	1	
S	s	sym	N	
M	m	month	4	[yy]yy[?]mm
D	d	date	4	[yy]yy[?]mm[?]dd or [m]m/[d]d/[yy]yy
Z	z	datetime	8	date?time
U	u	minute	4	hh[:]mm
V	v	second	4	hh[:]mm[:]ss
T	t	time	4	hh[:]mm[:]ss[[.]ddd]
* as is chars				

For example, ("IFC D";4 8 10 6 4) specifies a file of fixed width fields consisting of a 4 byte int field, an 8 byte float field, a field of 10 chars, a field of width 6 that will not appear in the loaded data, and a 4 byte date field. Note that the sum of the field widths must equal the record width. For example, if there is one blank between fields then the specification would have to be ("IFC D";5 9 11 7 4) . Text files to be read as fixed width fields must have a newline character at the end.
A text file with fixed-width fields is read by

```
(types;widths)0:f
```

and a binary file is read by

```
(types;widths)1:f
```

The right argument f is either a file name as a sym atom or a 3-item list of the form (file name;I;L) . L specifies the number of bytes to be read and I specifies the starting index (the index of the first character in the file is 0). A file can only be read from the start of one record to the end of another. This form is the one to use when a very large file must be read incrementally.

The result in both cases is a general list of lists with an item for each field.

Variable width, delimited text files can also be read. The left argument in this case is a pair of the form (T;D) . T is a simple char list of type letters, as above. D is either a field-delimiting character or the enlist of one. In the case of a field delimiter, all rows are read as data and the result is a list, as above. For the enlist of a delimiter, it assumed that the first row contains field names, which are read as a simple field list; the remaining rows are read as data. The result is a table.

For example, suppose that the following display are the rows of a csv file named test.csv.

```
abc,def,ghi,jkl
1050,1.234,abcdef,G
234,1e50,gqw,X
```

If this file read by the expression

```
  ("IFSC";",")0:`test.csv
```

then the result is the list

```
(0N 1050 234;0n 1.23 1.000e+050;`ghi`abcdef`gqw;" GX")
```

You can see that the text "abc" on the first row of the file becomes the int 0N, "def" becomes the float 0n, and "jkl" becomes the null char " " (because it has more than one character). However, if the file is read with an enlisted delimiter,

```
("IFSC";enlist ",")0:`test.csv
```

the result is the table

```
+`abc`def`ghi`jkl!(1050 234;1.23 1.000e+050;`abcdef`gqw;"GX")
```

Input/Output to Files

Extract from Denis Shasha primer: Input/Output to Files
Files are identified as symbols beginning with a colon:

```
`:[path]name
```

The path need not be specified if the file is in the current directory.
Files are text, byte or q data. Let us start by reading and writing to text files:

```
myfile: `:foofile
/ can write directly to the file
myfile 0: ("i love";"rock and roll")
/ Can append to files, by opening the file and then talking to the file.
h: hopen myfile
h "But rock concerts are too loud."
/ The function read0 reads the file
newlist: read0 myfile
newlist / returns ("i love";"rock and roll";"But rock concerts are too loud.")
```

Byte files are for non-string data. For such data we use 1: instead of 0:.

```
mybinfile: `:foofilebin
mybinfile 1: 0x6768
/ To append, we open as before.

h2: hopen mybinfile
h2 0x6364
read1 mybinfile / returns 0x67686364
```

Q files are for q data.

```
myfile:`:tmpfoo
.[myfile;();:;`a`b`c]  / set
.[myfile;();,;`d`e`f]  / append
value`:tmpfoo  / returns `a`b`c`d`e`f
```

Very often we want to parse data that we get from external text files. Sometimes that data comes in fixed format. Suppose for example we have the file fooin with implicit schema employeeid, name, salary, age.

```
312 smith    3563.45 24
23  john     5821.19 32
9   curtiss  9821.19 51
```

(Note: no blank line at the end.)
We notice that we have a 4 character integer (including the trailing blank), a 9 character name, a 8 character float and a 2 digit integer. So we use the standard table to get the types :

We can then bring this data in as

```
myfooin: `:fooin
x: ("ISFI"; 4 9 8 2) 0: myfooin
x
returns (312 23 9;`smith`john`curtiss;3563.45 5821.19 9821.19;24 32 51)
/ Now can create a dictionary
d: `id`name`salary`age!x
/ And then a table
myemp: flip d
select name from myemp where salary > 5000 / returns +(,`name)!,`john`curtiss
```

We can write entire tables and databases to disk.

```
/ We write the stock table as follows
.[`:stock;();:;stock]
/ We read the table back with
value `:stock
```

12.5 Interprocess Communication

When running a q script, you can specify a port and then clients can talk to that port. Let's start with the script trade.q:

```
/ Create a list of French stocks where name is the key.
stock:([name:`Alcatel`Alstom`AirFrance`FranceTelecom`Snecma`Supra]
 industry:`telecom`engineering`aviation`telecom`aviation`consulting)
/ get a list of distinct stocks are there?
stocks: distinct exec name from stock
ns: count stocks
n:10000
/ stock is a foreign key foreign key.
/ We are taking n stocks at random.
/ Then n prices up to 100.0 at random, then amounts then dates.
trade:([]stock:`stock$n?stocks;
 price:n?100.0;amount:100*10+n?20;exchange:5+n?2.0;date:1998.01.01+n?449)
`date xasc`trade
/ An example function.
f:{[mytable] count select from mytable}
f[stock]
f[trade]
/ A somewhat less trivial function does volume-weighted
/ rollups by week (an abridged example from Tom Ferguson)
/ over a certain time period.
weekrollup:{[daterange]
 select first date, last date, amount wavg price by stock, date.week
  from trade where date within daterange }
weekrollup[2004.02.31 2004.03.31]
/ In the browser, the first few rows are (with my random generation)
stock         week        date        date          price
---------------------------------------------------------
Alcatel       2004.03.01  2004.03.02  2004.03.07  47.35833
Alcatel       2004.03.08  2004.03.08  2004.03.14  52.90358
Alcatel       2004.03.15  2004.03.15  2004.03.21  45.02117
Alcatel       2004.03.22  2004.03.22  2004.03.28  48.47076
Alcatel       2004.03.29  2004.03.29  2004.03.31  55.88028
Alstom        2004.03.01  2004.03.02  2004.03.07  54.75892
Alstom        2004.03.08  2004.03.08  2004.03.14  46.97044
Alstom        2004.03.15  2004.03.15  2004.03.21  46.22599
Alstom        2004.03.22  2004.03.22  2004.03.28  45.22842
Alstom        2004.03.29  2004.03.29  2004.03.31  44.11812
AirFrance     2004.03.01  2004.03.02  2004.03.07  49.62135
AirFrance     2004.03.08  2004.03.08  2004.03.14  61.13775
AirFrance     2004.03.15  2004.03.15  2004.03.21  50.15903
AirFrance     2004.03.22  2004.03.22  2004.03.28  48.55051
AirFrance     2004.03.29  2004.03.29  2004.03.31  67.37883
FranceTelecom 2004.03.01  2004.03.02  2004.03.07  44.48418
FranceTelecom 2004.03.08  2004.03.08  2004.03.14  62.79708
```

Let's run trade.q at some port, here 5001.

```
q trade.q -p 5001
```

The process now running is the server. It is waiting for commands as we will see.
Now open up a new window on the same machine (or a different one) and type simply q. This is the client. The client then connects to this server by opening the connection, communicating, and then closing the connection.

```
/ If client on same machine, then please type in the console window for client:
h: hopen`::5001 / open a connection to a process on the local machine.
/ If client on different machine and server on machine foobar
/ then please type in the client console window:
h: hopen`:foobar:5001
/ Please type in the client console:
h"select avg price by stock from trade" / send a message as a string
/ and gets a result.
/ Define in the server.
/ Please type in the server console:
myfunc:{[x] 3*x}
/ Then the client can send a message to this function.
/ Please type this in the client console:
h"myfunc[4]" / and get the result 12
/ Alternatively, please type this in the client console:
h"n: 4; myfunc[n]" / returns 12
/ Also n is now defined as 4 in the server.
/ Alternatively, please type this in the client console:
h"z: 4"
h"myfunc[z]"
/ Sometimes the server chooses to allow certain operations to be
/ executed on its site, so uses a message filter.
/ There are two: .z.pg for synchronous and .z.ps for asynchronous messages
/ We are dealing with synchronous only for now.
/ Suppose we define the function .z.pg to count the number of characters.
/ Please type this in the server console:
.z.pg:{[x] count x}
/ Please type in the client console:
h"myfunc[z]" / returns 9 (because there are 9 characters)
```

Note that you can talk to this process from other places as well. For example, from a browser you can type. (Note however: it is very very important that you refresh from time to time if you think you have changed your data.)

```
http://localhost:5001/?select avg price by stock from trade
/ if server is local
http://hogwarts:5001/?select avg price by stock from trade
/if server is on machine hogwarts
```

There are also hooks from excel, perl, java, and C but please check the documentation elsewhere on the kx site for those.

Handles

```
`v set 2  / indirect assignment
  get`v
```

Files

File handles are of the form `:PATH. Kdb+ files can be read and written just like any other variable, e.g.

```
`:f set 2
   get`:f
```

Non kdb+ files are data(bytes) or text(lines).

```
h:hopen`:f.dat
h 0x2324
hclose h
h:hopen`:f.txt
h ` sv("asdf";"qwer")
hclose h
```

Sockets

```
h:hopen`:host:port
```

```
h"f[2;3]"          / execute
h(`f;2;3)          / func args
(-h)"a:2"          / asynch
hclose h           / close
.z.pg              / get callback(default: value)
.z.ps              / set callback(default: value)
.z.w               / who (socket handle)
```

NB: hclose and exit flush pending messages. to chase use: h""

Interfacing with Other Programmes

Kdb+ can interface with a number of other common technologies including:

- Java
- C#
- C
- C++

General Notes

The code gives some general notes on the various methods of accessing kdb+ databases

```
clients (see http://www.studio4kdb.com/ExcelRTDServer.htm)

the easiest way to programmatically call the kdb+servers
is with c.java and c.cs (memory managed and 1-1 datatypes).

java, .net, c/c++ and q clients connect and execute.

java/.net: (c.java/c.cs)
c c=new c("host",port);        // connect
r=c.k(["string"[,x[,y[,z]]]]); // remote call/read

c/c++: (l32/c.o s32/c.o w32/c.dll(c.lib))
int c=khp("host",port);
r=k(c[,"string"[,x[,y[,z]]]],(K)0);

q:
c:hopen`:host:port
c("string"[;x[;y[;z]]])

in java/.net

c.k("string"); // execute
c.k("func",x,y); // func[x;y]

to pass more than 3 args put them in list(s), e.g.

// build (time;sym;price;size) record
Object[]x={new Time(System.currentTimeMillis()%86400000),"xx",new
Double(93.5),new Integer(300)};
c.k("insert","trade",x);       // insert[`trade;x]

clients can also read incoming messages:
r=c.k();

for example, kdb+tick clients do:
String[]syms={"IBM","MSFT",..};
c.k(".u.sub","trade",syms);    // subscribe
while(1){Object r=c.k();..     // process incoming

parameters(x,y,z) and results(r) are arbitrarily
nested arrays of arrays and atoms.
(Integer, Double, int[], DateTime etc.)
```

```
java/.net(like q) have memory management
and self-describing objects so they are very easy to use.
the java/.net programmer passes and receives java/.net data.

C clients, e.g.  http://kx.com/q/c/c.c (requires gcc 3.0 or later)

must call khp before generating k data.
link with l32/c.o, s32/c.o or w32/c.dll(c.lib)
send asynch messages with: k(-c,..)
read asynch messages with: r=k(c,(K)0);

int c=khp("host",port);
K x=knk(4,kt(1000*(time(0)%86400)),ks("xx"),kf(93.5),ki(300)); //
time,sym,price,size
k(-c,"insert",ks("trade"),x,(K)0);

we use the K struct to hold data:

 K is atom(-19..-1) list(0 1..19) flip(98) dict(99)

atoms are type(t),value(ghijefs)      x->g     x->h     x->i     etc.
lists are type(t),count(n),values(G)  kG(x)[i] kH(x)[i] kI(x)[i] etc.

 r(ref) t(type) atom(ghijefs) list(u(supg) n(count) G(data)) flip(+k)
dict(kK(x)[0]!kK(x)[1])
 atom(kb,kg,..kt) list(ktn(t,n) knk(1-7,..) kp(S) kpn(S,I))  xT(xD(keys,values))
T:ktd(D|T)

 time(t) is milliseconds. datetime(d/z) is days from 2000.01.01 e.g.
 I x=time(0),t=1000*(x%86400),d;F z=x/8.64e4-10957;d=(I)z;

 r1(inc) r0(dec)
 r=k(c,s,x,y,z,(K)0); decrements(r0) x,y,z. eventually program must do r0(r);
 if one of the parameters is reused you must increment, e.g.

 K x=ks("trade");
 k(-c,s,r1(x),..,(K)0);
 k(-c,s,r1(x),..,(K)0);
 ..
 r0(x);

 r=k(c,..) error is r->t==-128 and Ks(r) is the error string

 k(-c,..) just returns non-zero on success.

 if k(..) returns 0 the connection is broken

servers, e.g. http://kx.com/q/c/a.c

c shared objects (link with q.lib) can be dynamically loaded
into q servers (q can call c and c can call q:  r=k(0,..))

 krr(return error)
 sockets: sd1(d,callback) sd0(close)

BULK INSERTS FROM C
at some point the records need to be assigned to different tables(types).
for one table(easiest case):

either grow (using ja) or set(better) if you know how many messages.
(remember the data is all by column)
suppose n columns and m rows then:
```

```
send all available data at once.(bulk)
if you know there are m records and n columns:

x=ktn(0,n); // n columns

e.g. (sym;price;size)

a=kK(x)[0]=ktn(KS,m);  // symbol vector
b=kK(x)[1]=ktn(KF,m); // float(64bit) vector
c=kK(x)[2]=ktn(KI,m); // integer vector

for(i=0;i<m;++i){ // populate all the cells
 kS(a)[i]=..; // set row i of column a
 kF(b)[i]=..; // set row i of column b
 ..}

k(c,".u.upd",ks("trade"),x,(K)0); // insert


GROW
..=ktn(kS,0);..

while(..){ja(a,&sym);ja(b,&price);ja(c,&size);}
```

Dynamically Linked C Functions

This code extract provides an example of dynamically linking C functions to q.

```
// q/w32>cl /LD ..\c\a.c a.def q.lib
// q/l32>/usr/local/gcc-3.3.2/bin/gcc -shared ../c/a.c -o a.so
// q/s32>/usr/local/gcc-3.3.2/bin/gcc -G      ../c/a.c -o a.so

#include"k.h"
K f(K x){return ki(x->i+1);}  // q calls c
K g(K x){return k(0,"1+",r1(x),0);} // c calls q


/*
f:`a 2:(`f;1)
g:`a 2:(`g;1)
f 2
g 3
*/
```

This code sample gives an example of how to connect, query and update kdb+ databases using C.

```
//>q trade.q -p 5001
#include"k.h"
#define Q(e,s) {if(e)return printf("?%s\n",s),-1;} // error
int main(){K x,y;int c=khp("localhost",5001);Q(c<0,"connect")
 Q(!k(-c,"`trade insert(12:34:56.789;`xx;93.5;300)",0),"err") // statement insert
 Q(!(x=k(c,"select sum size by sym from trade",0)),"err")    // statement select
 Q(!(x=ktd(x)),"type")                  // flip from keyedtable(dict)
  y=x->k;                               // dict from flip
  y=kK(y)[1],printf("%d cols:\n",y->n);  // data from dict
  y=kK(y)[0],printf("%d rows:\n",y->n);  // column 0
  printf("%s\n",kS(y)[0]);               // sym 0
  r0(x);                                 // release memory

 x=knk(4,kt(1000*(time(0)%86400)),ks("xx"),kf(93.5),ki(300)); // data record
// DO(10000,Q(!k(-c,"insert",ks((S)"trade"),r1(x),0),es)) // 10000 asynch inserts
// k(c,"",0); // synch chase
// return 0;
 Q(!k(-c,"insert",ks("trade"),x,0),"err")                     // parameter insert
 Q(!(x=k(c,"{[x]select from trade where size>x}",ki(100),0)),"err") // parameter select
  r0(x);
 close(c);
 return 0;}

/*
gcc ../c/c.c c.o -lsocket
gcc ../c/c.c c.o -lsocket -lnsl
cl  ../c/c.c c.lib ws2_32.lib
*/
```

Kdb+/C# API

This code illustrates the kdb+/C# API.

```
// 2005.04.26 s.Close()
// 2005.03.09 readwrite 0Nd 0Nz
// 2004.10.12 usr:pwd
http://msdn.microsoft.com/library/default.asp?url=/library/en-
us/cpguide/html/cpcontemplatefiles.asp
using System;using System.IO; //csc c.cs  given >q trade.q -p 5001
class c:System.Net.Sockets.TcpClient{public static void Main(string[]args){
 c c=new c("localhost",5001);
 Flip r=td(c.k("select sum price by sym from trade"));O("cols:
"+n(r.x));O("rows: "+n(r.y[0]));
// object[]x=new object[4];x[0]=t();x[1]="xx";x[2]=(double)93.5;x[3]=300;
// tm();for(int i=0;i<1000;++i)c.ks("insert", "trade", x);tm();
// c c=new
c("localhost",5010);c.k(".u.sub[`trade;`MSFT.O`IBM.N]");while(true){object
r=c.k();O(n(at(r,2)));}
 c.Close();}
byte[]b,B;int j,J;bool a;Stream s;public c(string h,int
p):this(h,p,System.Environment.UserName){}
public new void Close(){base.Close();s.Close();}
public c(string h,int p,string u):base(h,p){s=this.GetStream();B=new
byte[1+u.Length];J=0;w(u);s.Write(B,0,J);if(1!=s.Read(B,0,1))throw new
Exception("access");}
static TimeSpan t(){return DateTime.Now.TimeOfDay;}static TimeSpan v;static void
tm(){TimeSpan u=v;v=t();O(v-u);}
static void O(object x){Console.WriteLine(x);}static string i2(int i){return
String.Format("{0:00}",i);}
static bool dt(DateTime d){return d==d.Date;}static int DT(DateTime[]d){int
i=0;for(;i<n(d);++i)if(dt(d[i]))return 15;return 14;}
static double zd(DateTime d){return d.Ticks==0?Double.NaN:(d.Ticks-
o)/8.64e11;}static long o=(long)8.64e11*730119;
public class Dict{public object x;public object y;public Dict(object X,object
Y){x=X;y=Y;}}
```

```
public class Flip{public string[]x;public object[]y;public Flip(Dict
X){x=(string[])X.x;y=(object[])X.y;}}
public class Month{public int i;public Month(int x){i=x;}public override string
ToString(){int m=24000+i,y=m/12;return i2(y/100)+i2(y%100)+"-"+i2(1+m%12);}}
public class Minute{public int i;public Minute(int x){i=x;}public override
string ToString(){return i2(i/60)+":"+i2(i%60);}}
public class Second{public int i;public Second(int x){i=x;}public override
string ToString(){return new Minute(i/60).ToString()+':'+i2(i%60);}}
public static Flip td(object X){if(t(X)==98)return(Flip)X;Dict d=(Dict)X;Flip
a=(Flip)d.x,b=(Flip)d.y;int m=c.n(a.x),n=c.n(b.x);
 string[]x=new string[m+n];Array.Copy(a.x,0,x,0,m);Array.Copy(b.x,0,x,m,n);
 object[]y=new
object[m+n];Array.Copy(a.y,0,y,0,m);Array.Copy(b.y,0,y,m,n);return new Flip(new
Dict(x,y));}
static int t(object x){return x is bool?-1:x is byte?-4:x is short?-5:x is int?-
6:x is long?-7:x is float?-8:x is double?-9:x is char?-10:
 x is string?-11:x is Month?-13:x is DateTime?(dt((DateTime)x)?-14:-15):x is
Minute?-17:x is Second?-18:x is TimeSpan?-19:
 x is bool[]?1:x is byte[]?4:x is short[]?5:x is int[]?6:x is long[]?7:x is
float[]?8:x is double[]?9:x is char[]?10:
 x is DateTime[]?DT((DateTime[])x):x is TimeSpan[]?19:x is Flip?98:x is
Dict?99:0;}
static int[]nt={0,1,0,0,1,2,4,8,4,8,1,0,0,4,4,8,4,4,4,4}; // x.GetType().IsArray
static int n(object x){return x is Dict?n(((Dict)x).x):x is
Flip?n(((Flip)x).y[0]):((Array)x).Length;}
static int nx(object x){int i=0,n,t=c.t(x),j;if(t==99)return
1+nx(((Dict)x).x)+nx(((Dict)x).y);if(t==98)return
3+nx(((Flip)x).x)+nx(((Flip)x).y);
 if(t<0)return t==-11?2+((string)x).Length:1+nt[-
t];j=6;n=c.n(x);if(t==0)for(;i<n;++i)j+=nx(((object[])x)[i]);else
j+=n*nt[t];return j;}
public static bool qn(object x){int t=-c.t(x);return
t==5?(short)x==Int16.MinValue:t==6?(int)x==Int32.MinValue:t==7?(long)x==Int64.Mi
nValue:

t==8?Single.IsNaN((float)x):t==9?Double.IsNaN((double)x):t==14||t==15?0L==((Date
Time)x).Ticks:t==19?qn(((TimeSpan)x).Ticks):false;}
public static object at(object x,int i){object r=((Array)x).GetValue(i);return
qn(r)?null:r;}

void w(bool x){B[J++]=(byte)(x?1:0);}bool rb(){return 1==b[j++];}void w(byte
x){B[J++]=x;}byte rx(){return b[j++];}
void w(short h){B[J++]=(byte)h;B[J++]=(byte)(h>>8);}short rh(){int
x=b[j++],y=b[j++];return(short)(a?x&0xff|y<<8:x<<8|y&0xff);}
void w(int i){w((short)i);w((short)(i>>16));}int ri(){int x=rh(),y=rh();return
a?x&0xffff|y<<16:x<<16|y&0xffff;}
void w(long j){w((int)j);w((int)(j>>32));}long rj(){int x=ri(),y=ri();return
a?x&0xffffffffL|(long)y<<32:(long)x<<32|y&0xffffffffL;}
void w(float e){byte[]b=BitConverter.GetBytes(e);foreach(byte i in b)w(i);}float
re(){byte c;float e;

if(!a){c=b[j];b[j]=b[j+3];b[j+3]=c;c=b[j+1];b[j+1]=b[j+2];b[j+2]=c;}e=BitConvert
er.ToSingle(b,j);j+=4;return e;}
void w(double f){w(BitConverter.DoubleToInt64Bits(f));}double rf(){return
BitConverter.Int64BitsToDouble(rj());}
void w(char c){w((byte)c);}char rc(){return(char)(b[j++]&0xff);}void w(string
s){foreach(char i in s)w(i);B[J++]=0;}
string rs(){int i=0,k=j;for(;b[k]!=0;)++k;char[]s=new char[k-
j];for(;j<k;)s[i++]=(char)(0xFF&b[j++]);++j;return new string(s);}
void w(Month m){w(m.i);}Month rm(){int i=ri();return qn(i)?null:new Month(i);}
void w(Minute u){w(u.i);}Minute ru(){int i=ri();return qn(i)?null:new
Minute(i);}
```

```
void w(Second v){w(v.i);}Second rv(){int i=ri();return qn(i)?null:new
Second(i);}
void w(DateTime d){w((int)zd(d));} DateTime rd(){int i=ri();return new
DateTime(qn(i)?0:(long)(8.64e11*i)+o);}
void W(DateTime z){w(zd(z));}   DateTime rz(){double f=rf();return new
DateTime(qn(f)?0:(long)(8.64e11*f)+o);}
void w(TimeSpan t){w((int)(t.Ticks/10000));}TimeSpan rt(){return new
TimeSpan(10000L*ri());}
void w(object x){int i=0,n,t=c.t(x);w((byte)t);if(t<0)switch(t){case-
1:w((bool)x);return;case-4:w((byte)x);return;
 case-5:w((short)x);return;case-6:w((int)x);return;case-
7:w((long)x);return;case-8:w((float)x);return;case-9:w((double)x);return;
 case-10:w((char)x);return;case-11:w((string)x);return;case-
13:w((Month)x);return;case-17:w((Minute)x);return;case-18:w((Second)x);return;
 case-14:w((DateTime)x);return;case-15:W((DateTime)x);return;case-
19:w((TimeSpan)x);return;}
 if(t==99){Dict r=(Dict)x;w(r.x);w(r.y);return;}B[J++]=0;if(t==98){Flip
r=(Flip)x;B[J++]=99;w(r.x);w(r.y);return;}
 w(n=c.n(x));for(;i<n;++i)if(t==0)w(((object[])x)[i]);else
if(t==1)w(((bool[])x)[i]);else if(t==4)w(((byte[])x)[i]);
 else if(t==5)w(((short[])x)[i]);else if(t==6)w(((int[])x)[i]);else
if(t==7)w(((long[])x)[i]);
 else if(t==8)w(((float[])x)[i]);else if(t==9)w(((double[])x)[i]);else
if(t==10)w(((char[])x)[i]);
 else if(t==14)w(((DateTime[])x)[i]);else if(t==15)W(((DateTime[])x)[i]);else
if(t==19)w(((TimeSpan[])x)[i]);}
object r(){int i=0,n,t=(sbyte)b[j++];if(t<0)switch(t){case-1:return rb();case-
4:return b[j++];case-5:return rh();
 case-6:return ri();case-7:return rj();case-8:return re();case-9:return
rf();case-10:return rc();case-11:return rs();
 case-13:return rm();case-14:return rd();case-15:return rz();case-17:return
ru();case-18:return rv();case-19:return rt();}
 if(t>99){j++;return null;}if(t==99)return new Dict(r(),r());j++;if(t==98)return
new Flip((Dict)r());n=ri();switch(t){
  case 0:object[]L=new object[n];for(;i<n;i++)L[i]=r();return L;  case
1:bool[]B=new bool[n];for(;i<n;i++)B[i]=rb();return B;
  case 4:byte[]G=new byte[n];for(;i<n;i++)G[i]=b[j++];return G;   case
5:short[]H=new short[n];for(;i<n;i++)H[i]=rh();return H;
  case 6:int[]I=new int[n];for(;i<n;i++)I[i]=ri();return I;       case
7:long[]J=new long[n];for(;i<n;i++)J[i]=rj();return J;
  case 8:float[]E=new float[n];for(;i<n;i++)E[i]=re();return E;   case
9:double[]F=new double[n];for(;i<n;i++)F[i]=rf();return F;
 case 10:char[]C=new char[n];for(;i<n;i++)C[i]=rc();return C;    case
11:String[]S=new String[n];for(;i<n;i++)S[i]=rs();return S;
 case 13:Month[]M=new Month[n];for(;i<n;i++)M[i]=rm();return M;  case
14:DateTime[]D=new DateTime[n];for(;i<n;i++)D[i]=rd();return D;
 case 17:Minute[]U=new Minute[n];for(;i<n;i++)U[i]=ru();return U;case
15:DateTime[]Z=new DateTime[n];for(;i<n;i++)Z[i]=rz();return Z;
 case 18:Second[]V=new Second[n];for(;i<n;i++)V[i]=rv();return V;case
19:TimeSpan[]T=new TimeSpan[n];for(;i<n;i++)T[i]=rt();return T;}return null;}
void w(int i,object x){int n=nx(x)+8;B=new
byte[n];B[0]=1;B[1]=(byte)i;J=4;w(n);w(x);s.Write(B,0,n);}
public object k(){s.Read(b=new byte[8],0,8);a=b[0]==1;j=4;int i=0,m=ri()-8;b=new
byte[m];
 for(;i<m;i+=j)j=s.Read(b,i,m-i);if(b[0]==128){j=1;throw new
Exception(rs());}j=0;return r();}
public object k(object x){w(1,x);return k();}
public object k(string s){return k(cs(s));}char[]cs(string s){return
s.ToCharArray();}
public object k(string s,object x){object[]a={cs(s),x};return k(a);}
public object k(string s,object x,object y){object[]a={cs(s),x,y};return k(a);}
public object k(string s,object x,object y,object
z){object[]a={cs(s),x,y,z};return k(a);}
```

```
public void ks(String s){w(0,cs(s));}
public void ks(String s,Object x){Object[]a={cs(s),x};w(0,a);}
public void ks(String s,Object x,Object y){Object[]a={cs(s),x,y};w(0,a);}
}
```

Kdb+/C# Sample Interface

The process to display a q table in a C# windows forms application is as follows.

1. Open a connection to a Q database
2. Query the q database
3. Populate a DataGridView object with the result of the query.

All database interaction is done through the standard c.cs supplied by kx.
The main functionality used by this simple demo is as follows. (extracts from the code below)

```
qConn = new c(host, port);
```

The constructor for c allows us to connect to the database specified by host and port. In this case we already have a q process running on the same machine as the java process, on port 5001.

```
object o = qConn.k(sSql);
```

The function c.k(String) is used to query the database. It returns an object. This can be any of the q types (as specified in c.w and c.t function). Usually the returned object should be checked for expected type.

```
c.Flip d = c.td(o);
```

The above function attempts to convert the Object into a c.Flip type. The Flip type is the one that most resembles a table. Flip.x is the column names and Flip.y is the Data. This function will only work with dictionaries and flips

```
int cols = c.n(d.x);
int rows = c.n(d.y[0]);
```

The above counts the columns and rows from the data.

```
for(int i = 0; i < cols; i ++)
      dataGridView1.Columns.Add(d.x[i], d.x[i]);
```

The above code adds the appropriate columns to the table.

```
dataGridView1.Rows.Add(rows);
```

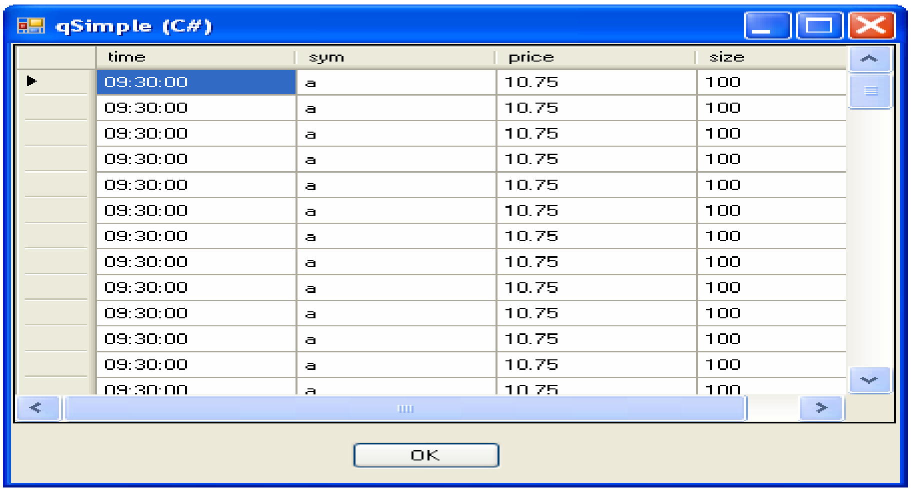
The above code adds the correct amount of rows.

```
for (int r = 0; r < rows; r++)
{
  for (int cc = 0; cc < cols; cc++)
    dataGridView1.Rows[r].Cells[cc].Value = c.at(d.y[cc], r);
}
```

c.at attempts to properly extract the data from the Array, whilst checking types and indices.

Simply start a Windows Application project in VS.NET and drop a DataGridView and a Button onto the form.
Then cut and paste this code over the Sample Code.

```
using System;
using System.Windows.Forms;

namespace qSimple
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            qConn = new c(host, port);
            populateModel();
        }

        string host = "localhost";
        int port = 5001;
        string sSql = "100#select from trade";
        c qConn = null;


        private void populateModel()
        {
            // Do Query
            object o = qConn.k(sSql);
            c.Flip d = c.td(o);

            int cols = c.n(d.x);
        int rows = c.n(d.y[0]);

        //add the column names
        for(int i = 0; i < cols; i ++)
        dataGridView1.Columns.Add(d.x[i], d.x[i]);

        //add the correct number of rows
        dataGridView1.Rows.Add(rows);


      //populate the values
        for (int r = 0; r < rows; r++)
        {
            for (int cc = 0; cc < cols; cc++)
              dataGridView1.Rows[r].Cells[cc].Value = c.at(d.y[cc], r);
        }

      } //end of populateModel()

        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void Form1_Resize(object sender, EventArgs e)
        {
            dataGridView1.Width = this.Width - 11;
            dataGridView1.Height = this.Height - 116;
        }
    }
}
```



	time	sym	price	size
▶	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100
	09:30:00	a	10.75	100

Kdb+/Java API

This code illustrates the kdb+/Java API.

```
// 2005.02.08 tzo

import java.net.*;import java.io.*;import java.sql.*;import java.text.*;import java.lang.reflect.Array;

public class c{public static void main(String[]args){try{

//c c=new c(new ServerSocket(5010));while(true)c.w(2,c.k());

//c c=new c("localhost",5010);Object[]x={"GE",new Double(2.5),new Integer(23)};c.k(".u.upd","trade",x);

 c c=new c("localhost",5001);Object[]x={new Time(t()),"xx",new Double(93.5),new Integer(300)};

 c.ks("insert","trade",x);

 Flip t=td(c.k("select sum size by sym from trade"));

 O(n(t.x));O(n(t.y[0]));O(at(t.y[0],0)); //cols rows data

 c.close();}catch(Exception e){e.printStackTrace();}}

DataInputStream i;OutputStream o;byte[]b,B;int j,J;boolean a;void io(Socket s)throws IOException{i=new
DataInputStream(s.getInputStream());o=s.getOutputStream();}

public void close(){if(i!=null)try{i.close();o.close();}catch(IOException e){}finally{i=null;o=null;}}

public c(Socket s)throws IOException{io(s);i.read(b=new byte[99]);o.write(b,0,1);}

public c(ServerSocket s)throws IOException{this(s.accept());}

public c(String h,int p,String u)throws KException,IOException{io(new Socket(h,p));B=new
byte[1+ns(u)];J=0;w(u);o.write(B);if(1!=i.read(B,0,1))throw new KException("access");}

public c(String h,int p)throws KException,IOException{this(h,p,System.getProperty("user.name"));}

public static class Dict{public Object x;public Object y;public Dict(Object X,Object Y){x=X;y=Y;}}

public static class Flip{public String[]x;public Object[]y;Flip(Dict X){x=(String[])X.x;y=(Object[])X.y;}}

public static class Month{public int i;Month(int x){i=x;}public String toString(){int m=i+24000,y=m/12;return
i2(y/100)+i2(y%100)+"-"+i2(1+m%12);}}

public static class Minute{public int i;Minute(int x){i=x;}public String toString(){return i2(i/60)+":"+i2(i%60);}}

public static class Second{public int i;Second(int x){i=x;}public String toString(){return new
Minute(i/60).toString()+':'+i2(i%60);}}

static String i2(int i){return new DecimalFormat("00").format(i);}

public static Flip td(Object X){if(t(X)==98)return(Flip)X;Dict d=(Dict)X;Flip a=(Flip)d.x,b=(Flip)d.y;int
m=n(a.x),n=n(b.x);

 String[]x=new String[m+n];System.arraycopy(a.x,0,x,0,m);System.arraycopy(b.x,0,x,m,n);

 Object[]y=new Object[m+n];System.arraycopy(a.y,0,y,0,m);System.arraycopy(b.y,0,y,m,n);return new
Flip(new Dict(x,y));}

static java.util.TimeZone tz=java.util.TimeZone.getDefault();static int tzo(long x){return tz.getOffset(x);}

static long lg(long x){return x+tzo(x);}static long gl(long x){long t=x-tzo(x);return t+x-lg(t);}

//object.getClass().isArray()   t(int[]) is .5 isarray is .1 lookup .05
```

```
static int t(Object x){return x instanceof Boolean?-1:x instanceof Byte?-4:x instanceof Short?-5:x instanceof Integer?-6:x instanceof Long?-7:

 x instanceof Float?-8:x instanceof Double?-9:x instanceof Character?-10:x instanceof String?-11:x instanceof Month?-13:

 x instanceof Date?-14:x instanceof Timestamp?-15:x instanceof Minute?-17:x instanceof Second?-18:x instanceof Time?-19:

 x instanceof boolean[]?1:x instanceof byte[]?4:x instanceof short[]?5:x instanceof int[]?6:x instanceof long[]?7:

 x instanceof float[]?8:x instanceof double[]?9:x instanceof char[]?10:x instanceof String[]?11:x instanceof Month[]?13:

 x instanceof Date[]?14:x instanceof Timestamp[]?15:x instanceof Minute[]?17:x instanceof Second[]?18:x instanceof Time[]?19:

 x instanceof Flip?98:x instanceof Dict?99:0;}

static int ni=Integer.MIN_VALUE;static long nj=Long.MIN_VALUE;static double nf=Double.NaN;

static int[]nt={0,1,0,0,1,2,4,8,4,8,1,0,0,4,4,8,0,4,4,4};static int ns(String s){return s.length();}

static Object[]NULL={null,null,null,null,null,new Short(Short.MIN_VALUE),new Integer(ni),new Long(nj),new Float(nf),new Double(nf),

 new Character(' '),"",null,new Month(ni),new Date(nj),new Timestamp(nj),null,new Minute(ni),new Second(ni),new Time(nj)};

public static boolean qn(Object x){int t=-t(x);return t>4&&x.equals(NULL[t]);}

public static Object at(Object x,int i){return qn(x=Array.get(x,i))?null:x;}

public static void set(Object x,int i,Object y){Array.set(x,i,null==y?NULL[t(x)]:y);}

static int n(Object x){return x instanceof Dict?n(((Dict)x).x):x instanceof Flip?n(((Flip)x).y[0]):Array.getLength(x);}

static int nx(Object x){int i=0,n,t=t(x),j;if(t==99)return 1+nx(((Dict)x).x)+nx(((Dict)x).y);if(t==98)return 3+nx(((Flip)x).x)+nx(((Flip)x).y);

 if(t<0)return t==-11?2+ns((String)x):1+nt[-t];j=6;n=n(x);if(t==0||t==11)for(;i<n;++i)j+=t==0?nx(((Object[])x)[i]):1+ns(((String[])x)[i]);else j+=n*nt[t];return j;}



void w(byte x){B[J++]=x;}void w(boolean x){w((byte)(x?1:0));}void w(short h){w((byte)(h>>8));w((byte)h);}

void w(int i){w((short)(i>>16));w((short)i);}void w(long j){w((int)(j>>32));w((int)j);}

void w(float e){w(Float.floatToIntBits(e));}void w(double f){w(Double.doubleToLongBits(f));}

void w(char c){w((byte)c);}void w(String s){int i=0,n=ns(s);for(;i<n;)w(s.charAt(i++));B[J++]=0;}

void w(Date d){long l=d.getTime();w(nj==l?ni:(int)(lg(l)/86400000-10957));}

void w(Timestamp z){long l=z.getTime();w(nj==l?nf:lg(l)/8.64e7-10957);}

void w(Time t){long l=t.getTime();w(nj==l?ni:(int)(lg(l)%86400000));}
void w(Month m){w(m.i);}void w(Minute u){w(u.i);}void w(Second v){w(v.i);}

void w(Object x){int i=0,n,t=t(x);w((byte)t);if(t<0)switch(t){case-1:w(((Boolean)x).booleanValue());return;

  case-4:w(((Byte)x).byteValue());return;      case-5:w(((Short)x).shortValue());return;

  case-6:w(((Integer)x).intValue());return;    case-7:w(((Long)x).longValue());return;

  case-8:w(((Float)x).floatValue());return;    case-9:w(((Double)x).doubleValue());return;

 case-10:w(((Character)x).charValue());return; case-11:w((String)x);return;

 case-13:w((Month)x);return;case-14:w((Date)x);return;case-15:w((Timestamp)x);return;

 case-17:w((Minute)x);return;case-18:w((Second)x);return;case-19:w((Time)x);return;}

 if(t==99){Dict r=(Dict)x;w(r.x);w(r.y);return;}B[J++]=0;if(t==98){Flip r=(Flip)x;B[J++]=99;w(r.x);w(r.y);return;}

 w(n=n(x));for(;i<n;++i)if(t==0)w(((Object[])x)[i]);else if(t==1)w(((boolean[])x)[i]);else if(t==4)w(((byte[])x)[i]);

 else if(t==5)w(((short[])x)[i]);else if(t==6)w(((int[])x)[i]);else if(t==7)w(((long[])x)[i]);

 else if(t==8)w(((float[])x)[i]);else if(t==9)w(((double[])x)[i]);else if(t==10)w(((char[])x)[i]);

 else if(t==11)w(((String[])x)[i]);else if(t==13)w(((Month[])x)[i]);else if(t==14)w(((Date[])x)[i]);

 else if(t==15)w(((Timestamp[])x)[i]);else if(t==17)w(((Minute[])x)[i]);else if(t==18)w(((Second[])x)[i]);

 else w(((Time[])x)[i]);}


boolean rb(){return 1==b[j++];}short rh(){int x=b[j++],y=b[j++];return(short)(a?x&0xff|y<<8:x<<8|y&0xff);}

int ri(){int x=rh(),y=rh();return a?x&0xffff|y<<16:x<<16|y&0xffff;}long rj(){int x=ri(),y=ri();return
a?x&0xffffffffL|(long)y<<32:(long)x<<32|y&0xffffffffL;}

float re(){return Float.intBitsToFloat(ri());}double rf(){return Double.longBitsToDouble(rj());}char
rc(){return(char)(b[j++]&0xff);}

String rs(){int i=j;for(;b[j++]!=0;);return new String(b,i,j-1-i);}

Time rt(){int i=ri();return new Time(i==ni?nj:gl(i));}Date rd(){int i=ri();return new
Date(i==ni?nj:gl(86400000L*(i+10957)));}

Timestamp rz(){double f=rf();return new Timestamp(Double.isNaN(f)?nj:gl((long)(.5+8.64e7*(f+10957))));}

Minute ru(){return new Minute(ri());}Month rm(){return new Month(ri());}Second rv(){return new Second(ri());}

Object r(){int i=0,n,t=b[j++];if(t<0)switch(t){case-1:return new Boolean(rb());case-4:return new
Byte(b[j++]);case-5:return new Short(rh());

  case-6:return new Integer(ri());case-7:return new Long(rj());case-8:return new Float(re());

  case-9:return new Double(rf());case-10:return new Character(rc());case-11:return rs();

  case-13:return rm();case-14:return rd();case-15:return rz();case-17:return ru();case-18:return rv();case-
19:return rt();}

 if(t>99){j++;return null;}if(t==99)return new Dict(r(),r());j++;if(t==98)return new Flip((Dict)r());n=ri();switch(t){
```

```
 case 0:Object[]L=new Object[n];for(;i<n;i++)L[i]=r();return L; case 1:boolean[]B=new
boolean[n];for(;i<n;i++)B[i]=rb();return B;

 case 4:byte[]G=new byte[n];for(;i<n;i++)G[i]=b[j++];return G; case 5:short[]H=new
short[n];for(;i<n;i++)H[i]=rh();return H;

 case 6:int[]I=new int[n];for(;i<n;i++)I[i]=ri();return I; case 7:long[]J=new long[n];for(;i<n;i++)J[i]=rj();return
J;

 case 8:float[]E=new float[n];for(;i<n;i++)E[i]=re();return E; case 9:double[]F=new
double[n];for(;i<n;i++)F[i]=rf();return F;

 case 10:char[]C=new char[n];for(;i<n;i++)C[i]=rc();return C; case 11:String[]S=new
String[n];for(;i<n;i++)S[i]=rs();return S;

 case 13:Month[]M=new Month[n];for(;i<n;i++)M[i]=rm();return M; case 14:Date[]D=new
Date[n];for(;i<n;i++)D[i]=rd();return D;

 case 17:Minute[]U=new Minute[n];for(;i<n;i++)U[i]=ru();return U;case 15:Timestamp[]Z=new
Timestamp[n];for(;i<n;i++)Z[i]=rz();return Z;

 case 18:Second[]V=new Second[n];for(;i<n;i++)V[i]=rv();return V;case 19:Time[]T=new
Time[n];for(;i<n;i++)T[i]=rt();return T;}return null;}

void w(int i,Object x)throws IOException{int n=nx(x)+8;B=new
byte[n];B[0]=0;B[1]=(byte)i;J=4;w(n);w(x);o.write(B);}

public void ks(String s)throws IOException{w(0,cs(s));} char[]cs(String s){return s.toCharArray();}

public void ks(String s,Object x)throws IOException{Object[]a={cs(s),x};w(0,a);}

public void ks(String s,Object x,Object y)throws IOException{Object[]a={cs(s),x,y};w(0,a);}

public void ks(String s,Object x,Object y,Object z)throws IOException{Object[]a={cs(s),x,y,z};w(0,a);}

public static class KException extends Exception{KException(String s){super(s);}}

public synchronized Object k()throws KException,IOException{i.readFully(b=new byte[8]);a=b[0]==1;j=4;

 i.readFully(b=new byte[ri()-8]);if(b[0]==-128){j=1;throw new KException(rs());}j=0;return r();}

public Object k(Object x)throws KException,IOException{w(1,x);return k();}

public Object k(String s)throws KException,IOException{return k(cs(s));}

public Object k(String s,Object x)throws KException,IOException{Object[]a={cs(s),x};return k(a);}

public Object k(String s,Object x,Object y)throws KException,IOException{Object[]a={cs(s),x,y};return k(a);}

public Object k(String s,Object x,Object y,Object z)throws
KException,IOException{Object[]a={cs(s),x,y,z};return k(a);}

static void O(Object x){System.out.println(x);}static void O(int x){System.out.println(x);}static void O(boolean
x){System.out.println(x);}

static void O(long x){System.out.println(x);}static void O(double x){System.out.println(x);}

static long t(){return System.currentTimeMillis();}static long t;static void tm(){long u=t;t=t();if(u>0)O(t-u);}

}
```

```
/*

String rs(){int k=j;for(;b[k]!=0;)++k;char[]s=new char[k-j];for(int i=0;j<k;)s[i++]=(char)(0xFF&b[j++]);++j;return new String(s);}

// java(unlike .net) messed up datetime, e.g. new Time(0);

static String sd(String s,java.util.Date x){SimpleDateFormat f=new SimpleDateFormat(s);f.setTimeZone(java.util.TimeZone.getTimeZone("GMT"));return f.format(x);}

static String sd(Date x){return sd("yyyy.MM.dd",x);}static String sd(Time x){return sd("HH:mm:ss.SSS",x);}

static String sd(Timestamp x){return sd("yyyy.MM.dd HH:mm:ss.SSS",x);}

public static class KDate extends Date{public KDate(long date){super(date);}public String toString(){return sd(this);}}

public static class KTime extends Time{public KTime(long time){super(time);}public String toString(){return sd(this);}}

public static class KTimestamp extends Timestamp{public KTimestamp(long time){super(time);}public String toString(){return sd(this);}}

Time rt(){int i=ri();return new KTime(i==ni?nj:i);}Date rd(){int i=ri();return new KDate(i==ni?nj:86400000L*(i+10957));}

Timestamp rz(){double f=rf();return new KTimestamp(Double.isNaN(f)?nj:(long)(.5+8.64e7*(f+10957)));}



static Class[]c={int[].class,boolean[].class,null,null,byte[].class,short[].class,int[].class};

static int foo(Class x){if(x.isArray())for(int i=0;i<9;++i)if(c[i]==x)return i;return 0;}

boolean b;int[]q=new int[2];tm();for(int i=0;i<1000000;++i)foo(q.getClass());tm();



5000 roundtrip

Object: 4+16Integer/Double|24Date/String + strings

c c=new c((new ServerSocket(2000)).accept());for(;;)c.w(2,c.k());

*/
```

Kdb+/Java interface example

The process to display a q table in a Swing Java screen is as follows.

4. Open A Connection to a Q database
5. Query the q Database
6. Create a Swing Table Model with the data returned from the q query inside it
7. Apply the model to a table and place the table in a Swing GUI.

All Database interaction is done through the standard c.java supplied by kx.
The main functionality used by this simple demo is as follows. (extracts from the code below)

```
qConn = new c(host,port);
```

The constructor for c allows us to connect to the database specified by host and port. In this case we already have a q process running on the same machine as the java process, on port 5001.

```
Object o = qConn.k(sSql);
```

The function c.k(String) is used to query the database. It returns an object. This can be any of the q types (as specified in c.w and c.t function). Usually the returned object should be checked for expected type.

```
c.Flip d = c.td(o);
```

The above function attempts to convert the Object into a c.Flip type. The Flip type is the one that most resembles a table. Flip.x is the column names and Flip.y is the Data. This function will only work with dictionaries and flips

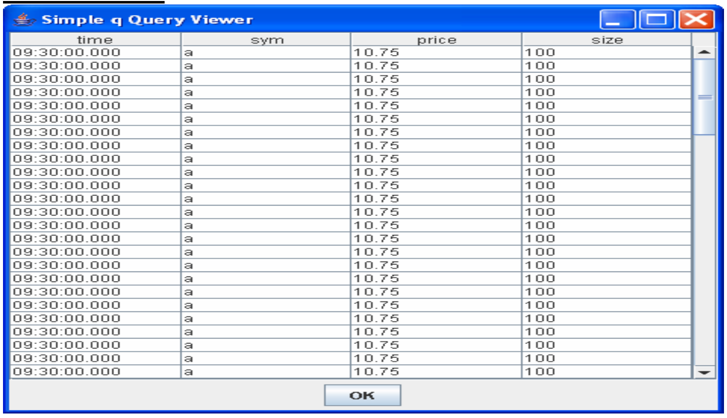
```
int cols = c.n(d.x);
int rows = c.n(d.y[0]);
```

The above counts the columns and rows from the data.

```
oo[r][cc] = c.at(d.y[cc], r);
```

c.at attepts to properly extract the data from the Array, whilst checking types and indices.

Screen Shot



time	sym	price	size
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100
09:30:00.000	a	10.75	100

Sample Code

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.io.IOException;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import c;

public class qSimple extends JFrame {
        String host = "localhost";
        int port = 5001;
        String sSql = "100#select from trade";
        c qConn = null;
        DefaultTableModel mdl = new DefaultTableModel();

        public qSimple() throws c.KException, IOException{
            // 1 - Open Connection to the Database
            qConn = new c(host,port);
            // 2 - Query Database & 3 - populateTableModel
            populateModel();
            // 4 - Place Model in Table and On Screen
            initScreen();
      }
```

```
	private void populateModel() throws c.KException, IOException {
		// Do Query
		Object o = qConn.k(sSql);
		c.Flip d = c.td(o);

		int cols = c.n(d.x);
		int rows = c.n(d.y[0]);

		Object[][] oo = new Object[rows][cols];
		for (int r = 0; r < rows; r++) {
			for (int cc = 0; cc < cols; cc++)
				oo[r][cc] = c.at(d.y[cc], r);
		}

		// Populate Table Model
		mdl.setDataVector(oo, d.x);
	}

	private void initScreen(){
		JButton btnOK = new JButton();
		JPanel pnlSouth = new JPanel();
		this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
	    this.setTitle("Simple q Query Viewer");
	    this.getContentPane().setLayout(new BorderLayout());
	    btnOK.setText("OK");
	    btnOK.addActionListener(new AbstractAction(){
			public void actionPerformed(ActionEvent e) {
				btnOK_actionPerformed(e);
			}
	    });
	    JScrollPane scrlMain = new JScrollPane();
	    this.getContentPane().add(pnlSouth,  BorderLayout.SOUTH);
	    pnlSouth.add(btnOK, null);
	    scrlMain = new JScrollPane(new JTable(mdl));
	    this.getContentPane().add(scrlMain,  BorderLayout.CENTER);
	}

	  void btnOK_actionPerformed(ActionEvent e) {
		    this.dispose();
		  }

	  public static void main(String args[]){
		  try{
			  JFrame frm = new qSimple();
			  frm.pack();
			  frm.setVisible(true);
		  }catch(Exception e){

		  }
	  }
}
```

kdb+/C++ API

This code extract illustrates the kdb+/C++ API.

```
#define nh ((I)0xFFFF8000)
#define wh ((I)0x7FFF)
#define ni ((I)0x80000000)
#define wi ((I)0x7FFFFFFF)
#ifdef WIN32
```

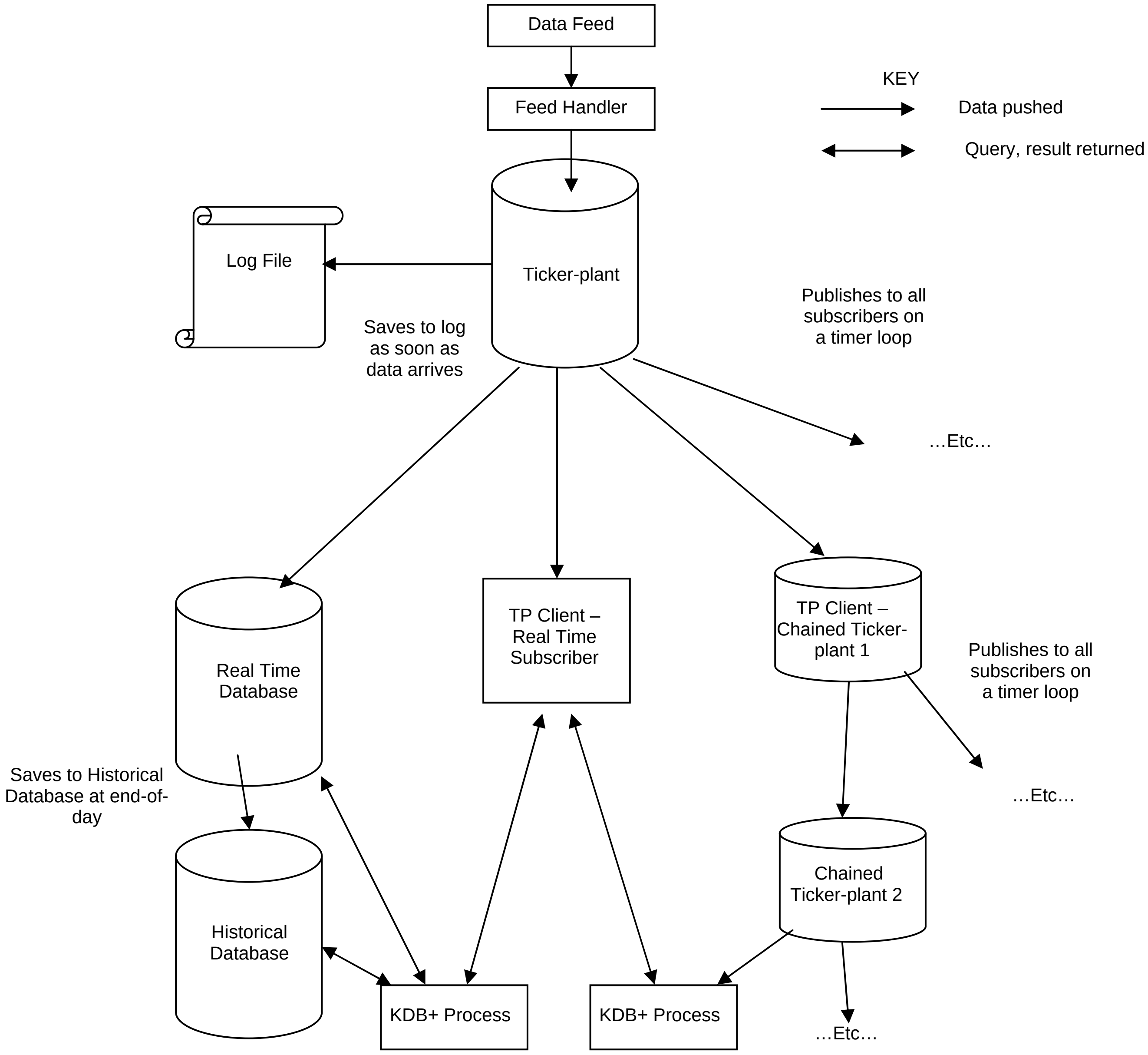
```
#define nj ((J)0x8000000000000000)
#define wj ((J)0x7FFFFFFFFFFFFFFF)
#define nf (sqrt(-1.0))
#define wf (-log( 0.0))
#define isnan _isnan
#define finite _finite
typedef __int64 J;
#else  //gcc3.0 or later for anonymous unions and anonymous structs
#define nj 0x8000000000000000LL
#define wj 0x7FFFFFFFFFFFFFFFLL
#define nf (0/0.0)
#define wf (1/0.0)
#define closesocket(x) close(x)
typedef long long J;
#endif //KBGHIJEFCSMDZUVT*  basetypes: GHIJEFS (G:BC)(I:MDUVT)(F:Z)
typedef int I;typedef unsigned char*A,G,*S,C;typedef short H;typedef float
E;typedef double F;typedef void V;
typedef struct k0{I r;H t,u;union{G g;H h;I i;J j;E e;F f;S s;struct
k0*k;struct{I n;G G0[1];};};}*K;

#ifdef __cplusplus
extern"C"{
#endif
extern I khpu(char*,I,char*),khp(char*,I),ymd(I,I,I);extern F sqrt(F);extern V
r0(K),sd0(I);extern S ss(S);
extern K
ka(I),kb(I),kg(I),kh(I),ki(I),kj(J),ke(F),kf(F),kc(I),ks(S),kd(I),kz(F),kt(I),sd
1(I,K(*)()),

ktn(I,I),knk(I,...),kp(S),kpn(S,I),ja(K*,A),js(K*,S),jk(K*,K),k(I,char*,...),xT(
K),xD(K,K),ktd(K),r1(K),krr(S);
#ifdef __cplusplus
}
#endif

// vector accessors, e.g. kF(x)[i] for float&datetime
#define kG(x)  ((x)->G0)
#define kH(x)  ((H*)kG(x))
#define kI(x)  ((I*)kG(x))
#define kJ(x)  ((J*)kG(x))
#define kE(x)  ((E*)kG(x))
#define kF(x)  ((F*)kG(x))
#define kS(x)  ((S*)kG(x))
#define kK(x)  ((K*)kG(x))

// vector types
#define KB 1
#define KG 4
#define KH 5
#define KI 6
#define KJ 7
#define KE 8
#define KF 9
#define KC 10
#define KS 11
#define KM 13
#define KD 14
#define KZ 15
#define KU 17
#define KV 18
#define KT 19

// table,dict
```

```
#define XT 98 //K
#define XD 99 //KK

// make c readable
#define O printf
#define R return
#define Z static
#define P(x,y) {if(x)R(y);}
#define U(x) P(!(x),0)
#define SW switch
#define CS(n,x)case n:x;break;
#define CD default
#define DO(n,x){I i=0,_i=(n);for(;i<_i;++i){x;}}

// remove more clutter
#define K1(f) K f(K x)
#define K2(f) K f(K x,K y)
#define TX(T,x) (*(T*)((G*)(x)+8))
#define xr x->r
#define xt x->t
#define xu x->u
#define xn x->n
#define xk TX(K,x)
#define xx xK[0]
#define xy xK[1]
#define xg TX(G,x)
#define xh TX(H,x)
#define xi TX(I,x)
#define xj TX(J,x)
#define xe TX(E,x)
#define xf TX(F,x)
#define xs TX(S,x)
#define xB ((G*)xG)
#define xG x->G0
#define xH ((H*)xG)
#define xI ((I*)xG)
#define xJ ((J*)xG)
#define xE ((E*)xG)
#define xF ((F*)xG)
#define xC ((C*)xG)
#define xS ((S*)xG)
#define xK ((K*)xG)

#define ZA Z A
#define ZV Z V
#define ZK Z K
#define ZH Z H
#define ZI Z I
#define ZJ Z J
#define ZE Z E
#define ZF Z F
#define ZC Z C
#define ZS Z S
```

Tick, Taq and Tow

Kdb+/tick Architecture

The diagram below gives a generalized outline of a typical Kdb+/tick architecture, followed by a brief explanation of the various components and the through-flow of data.


Data Feed
Feed Handler
KEY
Data pushed
Query, result returned
Log File
Ticker-plant
Saves to log as soon as data arrives
Publishes to all subscribers on a timer loop
...Etc...
Real Time Database
TP Client – Real Time Subscriber
TP Client – Chained Ticker-plant 1
Publishes to all subscribers on a timer loop
...Etc...
Saves to Historical Database at end-of-day
Historical Database
Chained Ticker-plant 2
KDB+ Process
KDB+ Process
...Etc...

- The Ticker-plant, Real-Time Database and Historical Database are operational on a 24/7 basis.
- The data from the data feed is parsed by the feed handler.

- The feed handler pushes the parsed data to the ticker-plant.
- Immediately upon receiving the parsed data, the ticker-plant writes the new data to the log file.
- The ticker-plant publishes the received data to its clients. This is either done immediate or on a timer loop, depending on configuration.
- The ticker-plant captures intra-day data but does not store it.
- The real-time database is a specialised ticker-plant subscriber which holds the intra-day data and responds to queries.
- In general, clients who need immediate updates of data (for example custom analytics) will subscribe directly to the ticker-plant (becoming a real-time subscriber). Clients who don't require immediate updates, but need a view the intra-day data will query the real-time database.
- A real-time subscriber can also be a chained ticker-plant. In this case it receives updates from a ticker-plant (which could itself be a chained ticker-plant) and publishes to its subscribers. This structure can be used to push data to clients as soon as it is available.
- At day end, the ticker-plant creates a new log file and sends an end-of-day message to each client.
- Upon receiving the end-of-day message, the real-time database saves all it's data to disk, sends a reload message to the historic database and purges its tables.
- The historic database reads the new data from the disk.

Components of kdb+/tick

Feed Handler

The feed handler is a process which connects to a data feed, parses incoming messages and pushes them into the ticker-plant. A ticker-plant can handle input from multiple feed handlers.

Feed handlers can be written in any language with an interface to kdb+. Kx Systems provide standard feed handlers for both Reuters and Bloomberg data feeds. However, feed handlers have been developed for many other data feeds.

The most commonly used feed handler is the Reuters Triarch feed handler. For more information on this see sections **Feed Handler Configuration** and **Reuters** Feedhandler Customisation.

Ticker-plant

The core component of Kdb+/tick is the ticker-plant, a specialized Kdb+ process that operates in a publish & subscribe configuration. The ticker-plant acts as a gateway, between a data feed and a number of subscribers.

As the ticker-plant logs all incoming updates to disk, any data which reaches to ticker-plant is recoverable following intra-day failure.

Subscriptions to the ticker-plant are on a per table and symbol basis. The ticker-plant passes on updates of interest to each client.

The update messages have a specific format. They are of the form

(`upd;`table_name;table_data)

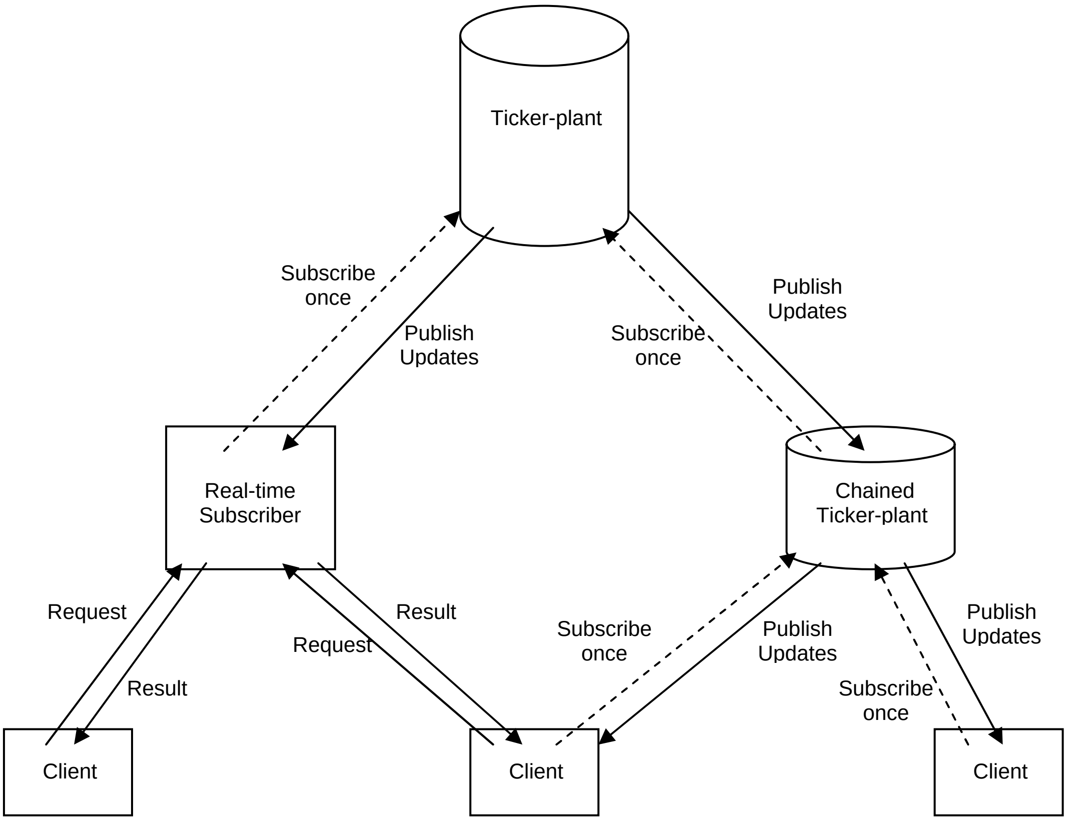
Each ticker-plant client must define a upd function which takes two parameters (table name and table data). The upd function defines what the client does with each update.

At day end (defined as midnight of the time zone in which the ticker-plant process runs) the ticker-plant sends out a message to all subscribers. This has the form

(`.u.end; current_date)

Subscribers can use this function call to invoke special purpose end-of-day functionality. A good example of this is the RTD, which uses .u.end as the signal to save to disk, reload the hdb, and flush its tables.

Real-Time Subscribers



Real-time subscribers are processes that subscribe to the ticker-plant and receive updates on the requested data. Typical real-time subscribers are kdb+ processes that process the data received from the ticker-plant and/or store them in local tables. The operation of a real-time subscriber can be easily customised.

kdb+/tick includes a set of default real-time databases, which are in-memory kdb+ databases that can be queried in real-time, taking full advantage of the powerful analytical capabilities of the *q* language and the speed of kdb+. Each real-time subscriber to the ticker-plant can support hundreds of clients and still deliver query results in milliseconds. Clients can connect to a real-time subscriber using one of the many interfaces available on Kdb+, including C/C++, C#, Java and the embedded HTTP server, which can format query results in HTML, XML, TXT, and CSV.

Multiple real-time processes subscribing to the ticker-plant may be used, for example, to off-load queries that employ complex, special-purpose analytics.

Real-time subscribers are not necessarily kdb+ databases. Using one of the interfaces above or TCP/IP socket programming, custom subscribers can be created using virtually any programming language, running on virtually any platform.

Real-Time Database

The real-time database (RTD) is a specialised real-time subscriber. It contains all the data received for the day so far.

When the RTD starts up it makes a call to the ticker-plant. In this call it makes a synchronous subscription for all data from all tables. The synchronous subscription returns the schema of all tables defined in the ticker-plant, which the RTD initializes itself with. The RTD also retrieves from the ticker-plant the location of the log and the number of messages observed so far.

It replays the log of messages and receives all subsequent updates. After this operation is complete the RTD has and maintains a complete history of all data observed on that day. As the RTD subscribes and retrieves the log count in the same call it will not miss or duplicate any updates.

When the RTD receives the end-of-day message (.u.end) it saves all data to disk and flushes its tables. It then sends a reload message to the historic database.

Chained Ticker-plants

Real-time subscribers can also be chained ticker-plants. This means that they have subscribers to which they publish updates.

Chained ticker-plants can be very simple. A common example is one which reduces subscription load to the main ticker-plant – one which subscribes to all data from the main ticker-plant and accepts subscription requests to the client. This reduces load on the main ticker-plant as it doesn’t have to deal with new clients subscribing and/or dropping subscriptions.

A more advanced use of chained ticker-plants would be as analytics engines – receiving the real-time (raw) data and performing some calculations to produce derived data, which it then publishes on to clients.

Historic Database

At day end, the RTD saves (splays) its data to disk. This data on disk forms the historic database (HDB). The HDB is a kdb+ process which reads this data from the disk in response to queries.

An HDB is a partitioned database composed of a collection of independent segments, any subset of which comprise a valid HDB. The database segments can all be stored within one directory on a disk, or distributed over multiple disks.

The HDB is partitioned by date, and each database segment is a directory on disk whose name is the date corresponding to the unique date on all data in that segment. When the RTD saves to disk it creates a new partition for the current day’s data, then sends a reload message to the HDB process which reads the new partition from the disk.

Customising kdb+/tick

All of the above is standard, out-of-the box operation of kdb+/tick. However, each process described is a kdb+ process, so can be customized in whichever way is required.

Implementing kdb+/tick

The table below outlines the main steps in a standard kdb+/tick implementation with cross references to other parts of this manual. It is not exhaustive but should give an indication of the main areas to consider.

Task	**Details and Manual References**
Install kdb+ and kdb+/tick	**Installation**
Configure the ticker-plant	Define the database schema and define and activate the connection to the (various) datafeed(s). Kdb+/tick comes with a number of predefined configuration scripts including two basic equity ticker plants (TAQ and SYM), the Level 2 ticker-plant and a fx ticker-plant. (**Configuration**) The default handler of kdb+/tick is Reuters ssl but custom feeds and schema can be built. (**Feed Handler Configuration**)
Managing the ticker-plant in production	Personnel tasked with managing the ticker-plants should get some understanding of how the database is partitioned and some of the conventions used (**The Ticker-plant**). Further consideration will need to be given to issues of scheduling startups and performance optimization (**Performance**).
Real-time database subscribers	Kdb+/tick can be configured to update a number of real-time subscribers. (**Real-time subscribers**)
Historical Database Issues	**Historical Database** In order to make queries to the HDB as efficient as possible, they should normally be constructed 'by date' first to take advantage of the construction of the database.
Making use of the ticker-plant	The installation of kdb+/tick is normally designed to take advantage of the power of q. There may be some requirement to use analytics or interfaces in other languages such as C++ or .net.
Multiple ticker-plants	**Using multiple ticker-plants** Data captured using multiple tickers should be divided up in such a fashion so as to keep queries as simple as possible; for instance in the same time-zone or currency.

Installation

To install kdb+/tick you need to have a valid licensing agreement with Kx Systems. The installation and license files for Kdb+/tick must be obtained directly from Kx Systems. The license file **‘k4.lic’** must be copied into the KDB+ installation directory. The Kdb+/tick distribution file is called **‘tick.zip’**, and contains the ticker-plant core and the configuration scripts for a variety of ticker-plant, real-time, and historical databases.

To install, simply extract the contents of the zip archive under the q/ directory. Prior to installing kdb+/tick, kdb+ must also be installed on the system. On windows the default installation directory is “C:\q” and under Solaris or Linux is “$HOME/q”. This location can be controlled via the “QHOME” environment variable. For the remainder of this document the q home directory will be donated as “QHOME”, and the path separator will be donated as “\” (back-slash).

The files should be unzipped and placed in the following directories:

Directory/file	**Purpose**
QHOME\tick	This contains all the q feedhandler and client code. The code within this folder may need to be modified for a number of purposes e.g., • taq.txt/sym.txt could be modified to capture different Reuters fields • the actual schema scripts sym.q/taq.q which defines the table structure may also need to be changed. • ssl.q may need to be modified to provide for different feedhandlers. • It may also be necessary to add to some of the default subscribers
QHOME\tick.k	This is the module containing all the ticker-plant functionality.

The path from which the commands are executed will be indicated as *the working directory*.

A Brief Description of the Scripts

The following scripts are included in the kdb+/tick package.

- **tick.k**
  The main ticker-plant script which defines the operation of the ticker-plant. The operation of the ticker-plant varies depending on the parameters with which the script is started. The command line parameters dictate whether or not the ticker-plant logs data, and whether it publishes its subscribers on a timer loop or immediately. See section **The Ticker-plant** System for more details.

- **r.k**
  This is the real-time database (RTD) which maintains a complete view of the intra day data. On subscription the RTD loads the days tick data up to that point from the log on disk, and then continues to receive updates via TCP/IP. In this way RTD can subscribe at any time during the day without overloading or delaying the plant.

- **u.k**
  This script contains the definitions for the publisher/subscriber functions. Any process which loads this script can become a tickerplant itself, making use of the pub/sub functions.

The following scripts are a subset of those available from the Kx Systems website.

- **ssl.q (Reuters ssl feedhandler)**
  This script receives the raw data from the feed, parses it and sends it to the ticker-plant. It connects to the feed by dynamically loading a c library containing functions for subscribing to Reuters. This can be configured for many different types of feed by making a few changes to the parsing rules.

- The scripts define the schema for the ticker-plant and just contain the table definitions.
  taq: trade and quote data
  sym: simplified trade and quote data
  fx: Forex data
  lvl2: level2 data

- **c.q**

  This script contains numerous easily configured sample ticker-plant subscribers.

The Ticker-plant System

Starting the Ticker-plant

A ticker-plant system usually has the ticker-plant, real-time db, historical db, one or more feeds and several clients. The file **test.q** included in the tick directory contains a script to start a ticker-plant system. (Note: This can be used only on Windows. For Solaris and Linux is should be changed to reflect a proper terminal starting command.)

```
\start q tick.k sym .      -p 5010
\start q tick/r.k 5010     -p 5011
\start q ./sym             -p 5012
\start q tick/c.q vwap     5010
\start q tick/ssl.q sym    5010
```

Explanation

q tick.k sym . -p 5010

This line starts the ticker-plant, using the table schema tick/sym.q. The general form of it is

q tick.k SRC DST [-p 5010] [-t 1000] [-o hours]

SRC specifies the schema to be loaded – a .q file located in the tick directory. sym refers to tick/sym.q.

DST specifies the location of the log file. The log will have the path **DST\symYYYY.MM.DD**, i.e. the schema type with the current date appended, at the location specified by **DST**. If **DST** is not specified the ticker-plant will not log the incoming messages. The "." in the line specified in test.q refers to the current directory.

The **–p <port>** option enables a q-IPC server listening for incoming subscriptions on the specified TCP/IP port. If no q-IPC port is specified the default port of 5010 is used

The **-t <int>** option sets the update interval used by the ticker-plant. This value defines the frequency in seconds of how often the ticker-plant publishes data to its real-time subscribers. If this is not set the ticker-plant defaults to zero-latency mode where it publishes updates immediately to subscribers.

The **–o <int>** is the offset in hours from GMT. This defaults to 0.

q tick/r.k 5010 -p 5011

This line starts the **RTD** on the port specified by –p. The parameter 5010 specifies the port on which the ticker-plant is running and tells the **RTD** which port to connect to receive real-time updates.

q ./sym -p 5012

This starts the historical db on the port specified by –p. The general form of it is

q DST/SRC –p 5012

Both DST and SRC should be the same as those specified in the ticker-plant start up line. This is because the **RTD** will save the historic data to this location, so the historic db should run from the data found in this location.

q tick/c.q vwap 5010

This starts up a real time client to the tickerplant using the script c.q. The client in this case is configured to calculate VWAP.

q tick/ssl.q sym 5010

This starts the feedhandler. The first parameter, sym, is the name of the file which contains the universe to subscribe to. If this parameter is set to “fx”, “taq” or “lvl2” it will configure the feedhandler to publish messages which fit the associated schema (see below).

Configuration

Kdb+/tick comes with a number of pre-defined ticker-plant schemas, including two basic equity ticker-plants (taq.q and sym.q), the level 2 ticker-plant and an FX ticker-plant. The default feed handler script of kdb+/tick (ssl.q) is used by all four configurations, and is configured with a command line parameter.

The Schema File

The schema file loaded by the ticker-plant determines the tables which the ticker-plant is expecting to receive, and the fields which those tables contain. This is the first parameter to the ticker-plant on start up, and is the starting point for defining a kdb+/tick database.

Ticker-plant Configuration

TAQ ticker-plant

The TAQ ticker-plant is the most widely used ticker-plant configuration as it is fully compatible with the other Kx Systems database product, Kdb+/taq, which allows the transfer of TAQ data from the NYSE-issued TAQ CDs to the TAQ historical database (see Kdb+/taq section for more details).

The TAQ ticker-plant can be used to subscribe to NYSE, AMEX and OTC symbols using the Reuters Triarch feed. The Reuters Triarch software must be installed on the system and the user authorized to access the data feed.

The database schema is as follows:

```
quote:([]time:`time$();sym:`symbol$();bid:`float$();ask:`float$();bsize:`int$();asize:`int$();mode:`char$();ex:`char$())
```

```
trade:([]time:`time$();sym:`symbol$();price:`float$();size:`int$();stop:`boolean$();cond:`char$();ex:`char$())
```

SYM ticker-plant

The SYM ticker-plant is a simplified version of the TAQ ticker-plant that stores trades and quotes for generic markets. The database schema is defined as follows:

```
quote:([]time:`time$();sym:`symbol$();bid:`float$();ask:`float$();bsize:`int$();asize:`int$())
```

```
trade:([]time:`time$();sym:`symbol$();price:`float$();size:`int$())
```

In the SYM ticker-plant symbols are stored exactly as received from the feed (i.e. including the exchange, if present, thus the missing ex column). The cond and mode columns are also not included in the schema.

This is the most commonly used configuration for markets which don't conform to TAQ specifications.

Level 2 ticker-plant

The Level 2 ticker-plant is designed to handle NASDAQ Level 2 quotes.

```
quote:([]time:`time$(); sym:`symbol$(); mm:`symbol$(); bid:`float$(); ask:`float$(); bsize:`float$(); asize:`float$())
```

where the sym column contains the combined stock symbol and the mm column contains the market maker identifier.

FX ticker-plant

The FX ticker-plant is used for subscribing to foreign exchange symbols.

The ticker-plant has the following schema :

```
quote:([]time:`time$();sym:`symbol$();bid:`float$();ask:`float$())
trade:([]time:`time$();sym:`symbol$();price:`float$();buy:`boolean$())
```

Custom ticker-plants

It is possible to define a customized ticker-plant using the Reuters or custom feed handler. In order to do so, a configuration script for the ticker-plant must be created, which defines the database schema and the connection to the data feed. As usual, the name of the configuration script will be automatically assigned as the name of the database.

Schema

The database schema must be defined so that the first two columns of all the tables are the time and sym columns, i.e. all tables must be of the form:

```
([]time:`time$(),sym:`symbol$(),…)
```

Valid tables have this form because the ticker-plant automatically fills in the time column with the current time when the update data is received, and subscriptions to the ticker-plant are on a table and symbol basis.

Feed Handler Configuration

Reuters Feed Handler Configuration

The current Reuters feed handler is written with the *SSL API* and will work with the old Triarch systems or the newer RMDS architecture. All that is required is that a sink distributor be available.

The only configuration issues should be:

- Create or add the appropriate entry to the *ipcroute* file;
- Ensure that the user account under which the ticker-plant will run has the correct permissions i.e. DACS.

For each of the ticker-plant configurations listed in the previous section there is a corresponding feedhandler. In each case the feedhander should be started with the name of the configuration as the first command line parameter, e.g.

```
q tick/ssl.q taq 5010 –p 5009
```

The universe of symbols to which the feedhandler subscribes is contained in the tick directory. This file must contain one symbol per line. Symbols must include the exchange specification.

This file is only read by the feedhandler when the feedhander starts-up. If changes are made to the file they will not be picked up until the feedhandler restarts. However, it is possible to force the feedhandler to subscribe to new symbols during the trading activity by calling the sub function. From the moment this function is called, the newly subscribed symbols will be received through the Reuters feed.

TAQ Feedhandler

The file which contains the universe of symbols is called 'taq.txt'. This will look like

```
MSFT.O
IBM.N
GOOG.O
```

The trade and quote tables store only the symbol in the sym column, while the specific exchange of each trade or quote is stored in the ex column.

sym Feedhandler

The sym configuration is the default configuration for ssl.q. The list of symbols must be specified in the 'sym.txt' file or in the file specified as a start up parameter. These will be symbols similar to the example given for 'taq.txt', e.g.

```
VOD.L
IBM.N
ALVG.DE
```

Lvl2 feehdandler

The list of symbols must be specified in the 'lvl2.txt' file in the tick directory. Symbols must be valid NASDAQ symbols the exchange specification, e.g.

```
0#MSFT.O
0#INTC.O
```

At start-up, the Level 2 feedhandler "chases" the linked list of Reuters Market Maker pages. The feedhandler uses these pages to find and subscribed to the complete list of market makers for each symbol.

fx feedandler

The fx feedhandler reads from a file called 'fx.txt', e.g.

```
EUR=
GBP=D2
```

The fx configuration works by modifying the FIDs which the feedhandler parses, rather than modifying the parsing functions themselves, as the lvl2 and taq configurations do.

Customising the Reuters Feed Handler

The standard Reuters feed handler passes the complete message back from the C library to q, where the required fields are parsed out and inserted into the tables. It is therefore possible to change the fields captured by either modifying the FIDs which the feedhandler inspects, or by modifying the relevant q functions. See **Reuters** Feedhandler Customisation for more details.

Bulk Inserts & Buffering

Bulk inserts can be used to maximize performance. kdb+/tick can handle up to 1 million singleton inserts per second, but many more records per second if the inserts are sent in bulk. Inserting records in bulk can improve performance greatly.

Inserting Data from Non-kdb+ Processes

Feedhandlers can be written in any language which can interface with kdb+. The following are simple examples in Java and C.

They both insert into a trade table with schema

```
([] time:`time$(); sym:`sym$(); price:`float$(); size:`int$())
```

Java

The example below will insert data into a tickerplant on a timer loop. The data will be the same each time.

```
import java.util.*;
import java.sql.*;

public class feed{

       static long t(){return System.currentTimeMillis()%86400000;}
       static void tm(){System.out.println(System.currentTimeMillis()%86400000);}
       int n=10000;
       int timer=100;
       Object[]Sym,sym,Price,price,Size,size,a;

public feed(){init();}

// Creates initial sym,price and size buffers
private void init(){
       Sym=new Object[n];
       Price=new Object[n];
       Size=new Object[n];}

// Copies sym,price and size to an object array for insertion to the tickerplant
private void copy(){
       a = new Object[3];
       a[0]=sym;
       a[1]=price;
       a[2]=size;}

private void tick(){
try{
       c c=new c("localhost",5010);                    // Connects to the tickerplant or
   // c c=new c("localhost",5010,"username:password"); // ... with username and password.
       long st=t();                                    // timer
       while(true){                // blocking loop to read from your stream

       for(int i=0;i<n;i++){
              Sym[i]  =new String("IBM");      //fake data – insert read function here
              Price[i]=new Double(93.5);
              Size[i] =new Integer(1001);

       // The next line ensures we process as many messages as possible
       // It checks whether the number of messages received is = to a certain predefined
       // number or whether or not a predefined time period has elapsed
       // note that we can also just do this tick by tick and the implementation
       // is even easier in this case
        // c.k(".u.upd","trade",Object[]x={"IBM",new Double(92.5),new Integer(1001)});

       if((i==n-1)||(timer<t()-st)){
                     if (i<(n-1)){      // if the time constraint has been surpassed we
                                        // only insert the number of records received
                            sym = new Object[i];             //create smaller sym array
                            System.arraycopy(Sym,0,sym,0,i); //copy existing Sym to sym –
                                                             // same for the other vectors
```

```
                          price=new Object[i];
                          System.arraycopy(Price,0,price,0,i);
                          size=new Object[i];
                          System.arraycopy(Size,0,size,0,i);
                          i=n;}

                   else{        // otherwise we use all records-i.e.the full vectors
                          sym = Sym;price=Price;size= Size;}

             Object[]x={sym,price,size};
             c.k(".u.upd","trade",x);                // does the update into the trade
table in our database
             init();                                       // re-initialise the counters
             st=t();}                                      // re-initialise the timer
             }}}

catch(Exception e)
       {e.printStackTrace();}}

public static void main(String[]args){feed f=new feed();f.tick();}}
```

C

```
/*******************************************************************************
*
* This program demonstrates the C - Q API.
*
* It behaves as a feed sending data to a kdb+ TickerPlant
*
* Assumes TP with trade table
*
* trade:([]time:`time$();sym:`symbol$();price:`float$();size:`int$();ex:())
*
* Written by A.Galiotos. 10/2006
*
* agaliotos@firstderivatives.com
*
********************************************************************************
/

#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include "k.h"

// Globals
int handle;
int max_rows = 100;     // maximum number or rows generated for bulk insert
char *syms[] = {"SPW","SSE","AA"};
char *exs[]  = {"L","PA"};

// feed frequency (in seconds)
const int freq = 1;
// TP port
char host[] = "localhost";
int port    = 5010;


void insert();
int q_time();

int main ()
{
      long timer;
      handle = khp(host,port);

      timer=clock();
      while(1)
      {
            insert();

            // insert every 'freq' seconds
            while ((clock()/CLOCKS_PER_SEC)<((timer/CLOCKS_PER_SEC)+ freq)){}
            timer=clock();
      }

}

void insert()
{
      int nrows,i;
      K data,time_col,sym_col,price_col,size_col,ex_col,res;

      nrows= rand() % max_rows;

      // create vector of type 0 and n elements. This is a list of the table
cols
      //data = ktn(0,5);
      data = knk(5,KT,KS,KF,KI,0);
      // feed doesnt give time
      // data = knk(5,KS,KF,KI,0);

      // define columns
      time_col    = kK(data)[0] = ktn(KT,nrows);
      sym_col     = kK(data)[1] = ktn(KS,nrows);
      price_col   = kK(data)[2] = ktn(KF,nrows);
      size_col    = kK(data)[3] = ktn(KI,nrows);
      ex_col      = kK(data)[4] = ktn(0,nrows);


      // enter rows of data for each column
      for (i=0;i<nrows;i++)
      {
            kI(time_col)[i]   = q_time();
            kS(sym_col)[i]    = ss(syms[i%3]);        // interned string
            kF(price_col)[i]  = 100*(rand()/(double)RAND_MAX);
            kI(size_col)[i]   = rand() % 10000;
            kK(ex_col)[i]     = kp(exs[i%2]);               // char list
      }

      // Make async call
      res = k(-handle,".u.upd",ks("trade"),data,0);

      // Async Call. Error is only thrown if handle is broken
      if(res == NULL){
            printf("CONNECTION BROKEN!!\nExiting...");
            exit(1);
      }

      printf("Inserted %d rows\n",nrows);

}

// use localtime function to construct q time (time in milliseconds)
int q_time()
{
      time_t * nulltime = 0;
      time_t curtime;
      int sec,min,hour;

      curtime = time(nulltime);
      sec    = 1000*                  localtime(&curtime)->tm_sec;
      min    = 1000*60*         localtime(&curtime)->tm_min;
      hour   = 1000*60*60*      localtime(&curtime)->tm_hour;
      return sec+min+hour;
}
```

Reuters Feedhandler Customisation

The script ssl.q, and the associated library (ssl.so or ssl.dll depending on operating system), will subscribe to a Reuters service and parse trade and quote date. The default Reuters service is IDN_SELECTFEED.

There are two main functions which are worth further consideration in ssl.q

1. f – the function called from the ssl library;
2. k – the function used to parse the message.

The other main function is g. This function parse the raw reuters message into a dictionary mapping Reuters FIDs (integers) to value. Generally it is not necessary to modify this function.

The Feedhandler functions

f function

This function is called from the ssl library when a new message arrives from the triarch service.

In ssl.q, f has this form :

```
f:{k each x where"6"=x[;3];
  if[count t;h(".u.upd";`trade;flip t)];
  if[count q;h(".u.upd";`quote;flip q)];
  t::q::()}
```

Line by line, this means :

```
k each x where"6"=x[;3];
```

Parse each message which is an update. On Triarch, a 316 message is an update and a 340 message is a snapshot. There are other message types, but these two are the types of most relevance at this stage.

```
if[count t;h(".u.upd";`trade;flip t)];
if[count q;h(".u.upd";`quote;flip q)];
```

If trades have been observed, held in t, or quotes, held in q, format them as a .u.upd message and send to the tickerplant.

The first parameter of .u.upd is the name of the table, and the second is the values.

```
t::q::()
```

Flush any messages which have just been published.

k function

In ssl.q, k has the form :

```
k:{s:`$sf x:g x;
```

```
 if[   tj in key x;t,:enlist s,tf@'x ti];
 if[any qj in key x;q,:enlist s,qf@'x qi]}
```

Line by line :

```
s:`$sf x:g x;
```

The g function is called on x, which converts the update message into a dictionary of fids mapped to values. This is then re-assigned back to x.

The sf function takes the first value of the dictionary x and drops of the first three characters. This is then cast to a symbol to create the variable s. s is the RIC code of the update.

This is the usual way to retrieve the RIC code. However, it is not required under the ssl api for an update to contain a RIC. See **Diadic Initialisation** for more details.

```
if[tj in key x;t,:enlist s,tf@'x ti];
```

At this point, s is the RIC code and x is a dictionary of FIDs to values. This line will parse a trade update.

ti is a list of the fids required to define a trade – by default this is 6 and 178, where 6 is the trade price, and 178 is the trade size. tj is the first element of ti, i.e. 6. So

```
tj in key x
```

is checking if the update dictionary contains a trade price. If it does, then

```
t,:enlist s,tf@'x ti
```

is executed.

```
x ti
```

will get the values from the dictionary x which are necessary for a trade.

```
tf@’x ti
```

will apply the corresponding parse function to each of the values retrieved from x ti. These functions are usually casts.

```
t,:enlist s,tf@’x ti
```

joins s (RIC) to the front of the list of values, enlists it, then adds this to t. As we saw previously, the f function then publishes the contents of t to the ticker-plant.

The next line, for quotes,

```
if[any qj in key x;q,:enlist s,qf@'x qi]
```

is similar to above.

Adding FIDs

If, for example, we would also like to retrieve the trade time (FID 379) from the update messages, we have to do several things.

1. Add a new column to the tickerplant schema.
2. Kill the tickerplant, feedhandler and RTD.
3. Remove the log file (or the RTD will fail to load with the new schema)
4. Re-start the tickerplant.
5. Modify the tickerplant code to parse the new FID. As 379 already has a parse function defined in fi, simply add 379 into the definition of ti.

If the FID is not already in fi, a parse function should be defined.

Customising the feedhandler

The feedhandler can be customised by changing (or adding) the parse functions defined in fi.

More advanced customisations should be undertaken by modifying the f and k functions. Some possible examples are:

1. insert into tables other than trade and quote;
2. separate trade information into different tables, e.g. on-exchange and off-exchange trades for European exchanges;
3. stamp an intra-millisecond ordering onto each update message so the stream of data can be reproduced exactly/

Any sort of customisation is possible using the power of q.

Diadic Initialisation

It is possible to initialise the Reuters ssl library so as to return two values on each callback to the f function. The first value is the RIC code of the update, taken from the header of the message received from Triarch. The second value is the body of the message. This should be used if the message body doesn't contain the RIC code. The k function should be modified to expect two parameters, the RIC code and the message.

To make use of this functionality, the library should be initialized with a string :

```
(`ssl 2:(`init;1))""
```

the g function should be changed :

```
g:(!)."I\037\036"0:
```

and the call of k in the f function should then be modified to expect two parameters :

```
f:{x k'g each y; etc
```

Filling in the blanks

When an update is received it does not contain all FIDs. It only contains the FIDs which have changed. It is sometimes necessary to fill in the previously observed values. For example, if the bid of a quote is updated the ask should be filled into the quote record. There is code to do this in ssl.q

```
/ maintain previous bid/ask state and fill if necessary
bid:ask:()!()
```

```
as:{[s;x]$[null x;ask s;ask[s]:x]}
bs:{[s;x]$[null x;bid s;bid[s]:x]}
```

The line in the k function for parsing quotes should then be changed to

```
if[any qj in key x;q,:enlist s,(as[s];bs[s];::;::)@'qf@'x qi]
```

This differs from the standard version by adding

```
(as[s];bs[s];::;::)@'
```

which is a list of functions which are then applied to each value parsed from the update message. as[s;x] and bs[s;x] will return either the new value for that sym or the previous value if the new value is null. The other two functions simply return the passed in value.

Filling in the blanks using dictionaries

An alternative approach to filling in the blanks is by using dictionaries to store the previous values. The idea is to create a dictionary of dictionaries which stores the previously observed values. This dictionary is keyed by RIC, and each sub dictionary contains the previously observed values, keyed by FID.

When an update arrives, create a dictionary of the updated values. Then update the stored dictionary with the new values and publish.

```
quote:()!()

/- in the k function
if[count qj:qi inter key x;
       /- FIDs which we are interested in have been observed (qj)
       /- Use these FIDs to update the stored values
       /- s1 is the RIC
       new_quote: quote[s1] ,: qj!qf[qj]@'x qj;
       /- At to the list of values to publish
       q,::enlist s1,value new_quote];
```

When adopting this approach, the dictionary quote will take the form

```
`VOD.L`BARC.L!((22 25!120.5 121);(22 25!650.5 651.0))
```

Other Functions and Variables within the Feedhandler

Callbacks from the C library

dis : function called when a the feedhandler is disconnected by Triarch.

rec : function called with the feedhandler re-connects to Triarch.

stt : function called by Triarch when a RIC is requested which is not available / is not permissioned for that user.

Functions and Variables

d : schema

sym : the RICS that will be subscribed to

sub : used to subscribe to Triarch.

cond / mode - dictionary for formatting condition and mode codes. Used by the taq tickerplant only.

Database Customisation

Message Handlers

kdb+ processes have several pre-defined message handlers. These are then only way to communicate remotely with a kdb+ process – all messages must pass through a message handler.

Message handlers are important for configuring a database. Examples of usage include:

- Logging - log incoming messages, results of calls, timings of calls;
- Security - allow/disallow access to the database, certain function calls etc. based on username/ip address etc.;
- Handle connections/disconnections from other processes

As message handlers are function calls, then can be configured to allow any required behaviour. The message handlers which are most commonly customised are listed below.

- .z.pg: synchronous message handler. Executed when a remote process makes a synchronous call. Parameter is the string/function call to be executed.
- .z.ps: asynchronous message handler. Executed when a remote process makes an asynchronous call. Parameter is the string/function call to be executed.
- .z.po: connection open handler. Executed when a remote process opens a connection.
- .z.pc: close connection handler. Executed when a remote process closes a connection. Parameter passed in is the handle of the process which has closed the connection.
- .z.pw: validation connection handler. Executed when a remote process opens a connection, and before .z.po. Used to validate connections.

Other message handlers are .z.ph (web server), .z.pp (http post) and .z.pi (calls made from qcon utility program).

RTD Customisation

Customising r.k

The script which defines the RTD is r.k. When customising this script, it is advisable to maintain the customisations in a separate file, and create a wrapper script which loads r.k followed by the customisations. The two main reasons for maintaining customisations in a separate file are

1. if a new version of r.k is released, this can simply be dropped in place of the previous version.
2. if another script requires to load the customisations (e.g. a log replay utility script), then it can load the customisations without loading r.k, thus avoiding starting as an rtd.

.u.end

At midnight the tickerplant calls the .u.end function in all of it's subscribers. It calls this function with one parameter, which is the date which has just passed. In the RTD standard configuration, .u.end will save down (splay) each of the top level RTD tables to the historic database under the partition defined by the parameter passed in (the previous date).

However, more functionality can be added. Some common requirements include :

1. Delete a table which doesn't need to be saved to the HDB;
2. Manipulate a table which doesn't need to be splayed. Some tables may contain semi-static data which would lend itself to being saved to a q data table at the top level of the HDB, rather than being splayed;
3. Create some summary details on a per instrument basis into a separate table which is also splayed to disk, e.g. open, close, high, low, vwap etc.

As .u.end is simply a function call, any logic can be implemented.

Loading files

Stored procedures should not be defined directly in the RTD customization script. Instead, they should be logically broken up into separate files.

These files should then be loaded into the RTD. One option would be to have a load line for each script. However this means that when a new stored procedure file is to be released a load line has to be added into the RTD customization script.

A better option is to separately maintain a directory of q files to be loaded by the RTD. This will make releases of new stored procedures easier, as the new script can simply be added to this directory. The loading of files should be error trapped so that loading a bad file does not cause the RTD to fail. Below is some example code to do this.

```
/- directory listing function – redefine for particular os
ls : "\\ls "

load_files : {
      /- get the list of files in the load directory
      files : @[system; ls,x; {[x;y] -1"Directory ",x," not found"; ()}[x]];
      if[count files;
            /- If files have been found, prepend the directory path
            files : x ,/: files;
            /- Then load each one in an error trap
            {@[{1"Loading file ",x,"..."; value"\\l ",x; 1" Successful\n"}; x;
      {1" Failure - ",x,"\n"}]} each files];

      -1"Load complete";}

/- run the function
load_files "rtd_sprocs/european_sprocs/"
```

HDB Customisation

Database Structure

In a standard tick setup the HDB is a set of directories on disk, each of which contains the data for one day. The top level of the database contains a sym (enumeration) file and any other files which should be loaded on start up.

Not all tables lend themselves to being splayed to disk. For example, a table of static/rarely changing data does not need to be saved down to disk every day. This data can be saved to the top level of the database, either as a splayed directory or as a single kdb+ table (binary format). This static table can be amended/appended each day by the RTD during the end-of-day save.

Loading Files

Any q file stored at the top level of a historic database directory will be loaded when the historic database starts up. These q files can be used to define stored procedures required by users of the database. However, if one of these scripts contains a bug, then the historic database will fail to load.

Similarly to the RTD, a directory of q files to be loaded by the HDB can be maintained separately. Again, the loading should be error trapped. This directory can be loaded by keeping one q file at the top level of the HDB which loads the stored procedure directory.

Real-Time Subscriber and Chained Ticker-plant Design

Kdb+/tick includes a number of default real-time subscribers contained in the script c.q, which are in-memory q databases updated in real-time by the ticker-plant. Although it is possible to create ticker-plant subscribers in practically any programming language, non-q ticker-plant subscribers offer almost no practical advantage over a software module directly connecting to the data feed.

kdb+ subscribers, on the other hand, are extremely easy to implement, can be queried in real-time from client applications using any of the several supported interfaces, and offer all the advantages of relational databases extended with the powerful q time-series and analytical capabilities. Moreover, q databases can be designed to alert clients upon specific conditions, such as when certain updates are received, or when a custom analytical query returns a certain value of interest.

If a real-time subscriber also accepts subscription requests and publishes updates, it is know as a chained ticker-plant. Chained ticker-plants (CTP) should be written in q. Both CTPs and RTSs will subscribe to a ticker-plant and receive updates upon subscription. Both will define a function (upd) which acts on update data. Generally, both will perform some calculations on this data, although a CTP can act simply as a relay, and a RTS can simply store data – the Real-Time Database is an RTS. Both can be queried, or set up to send messages or alerts to other processes. The only difference is that a CTP will publish updates to its subscribers whereas a RTS will not.

Discussion follows of the design and development of CTPs in q, although several of the issues are relevant to RTSs also.

When to use which

Use a CTP when any of the following are true:

- There are a dynamic number of clients;
- Clients require the ability to change the data they are listening for during the day (i.e. change their subscription);

Use a RTS when any of the following are true :

- Clients are not prepared to accept update messages (i.e. clients don’t subscribe)
- Clients require data infrequently or irregularly (i.e. they can just query a RTS).

Writing a Chained Tickerplant

A CTP could be used simply as a relay for data. This would be used to isolate and take load off the main tickerplant in an environment where there are a dynamic number of subscribers which are subscribing and un-subscribing throughout the day.

However, CTPs will generally do some processing on the data and then publish this processed data - the CTP will add value. It is quite common for a CTP to publish the raw data as well as the processed, so that clients receive all their data from one source and do not have to deal with timing issues associated with receiving raw and processed data from multiple sources.

A good example of a CTP is one which publishes VWAP data when new trades are published. This is a simple example but contains must of the features required by more complex CTPs.

Programming Considerations

It is usually important that a non-trivial CTP should publish its calculated data as fast as possible. The following guidelines should be observed when writing a CTP.

1. Cut down the number of tables and symbols being subscribed to as much as possible – this limits the amount of data flowing into the CTP.
2. Use groupings, sorts etc on tables wherever appropriate.
3. Do operations on bulked data wherever appropriate. This is more complicated with the zero latency ticker-plant as the messages which arrive tend to be quite small.
4. Whichever bit of data is most time-critical should be calculated and published first.

To implement a simple CTP, the following should be done:

1. Define tables which are to be published in the top level name space.
2. Define tables which aren't to be published in a different name space.
3. Define the upd function. This is the function which is called when an update is received and can be as complicated as required.
4. Load u.k and call .u.init[]. This initialises the process as a CTP and makes all top level tables publishable.
5. Subscribe to the main tickerplant.

As an example, the following script defines a CTP which acts as a zero latency data relay. Data relay tickerplants can also be implemented directly using tick.k.

```
/- load in u.k
\l tick/u.k

/- set the update function - simply publish
upd : {[t;x]
        .u.pub[t;x];}

/- open a handle to the main tickerplant
h : hopen `::5010

/- subscribe to the tickerplant synchronously
/- and save the result (the schema of the tables)
schema : h(`.u.sub;`;`);
```

```
/- set each of the schema tables at the top level
{.[x 0; (); :; x 1]} each schema

/- initialise the CTP
.u.init[];
```

A VWAP Publisher

A VWAP publisher is more complicated. It requires data previously observed to be stored, and to use this to calculate the VWAPs. One approach would be to store all the trades previously observed and calculate the VWAPs freshly every time. However, this would take up a lot of memory. A better approach is to store the sum of the size traded and the sum of the size multiplied by the price traded for each symbol. The following script will calculate and publish VWAPs (more examples, including a more concise vwap publisher, are available at http://www.kx.com/q/tick/c.q).

```
/- Load in the u.k
\l tick/u.k

/- Define the vwap table for publishing
vwap:([] time:`time$(); sym:`symbol$(); vwap:`float$())

/- Define a table, keyed on sym, to store the sum of the size
/- and the sum of (size*price)
.store.vwap:([sym:`symbol$()] sumsize:`int$(); sumsizeprice:`float$())

/- Define update function
/- Publish the table which comes in
/- Calculate the vwap if the table is a trade.
upd:{[t;x]
        .u.pub[t;x];
        if[t=`trade;
                /- update the vwap table
                .store.vwap +: select sumsize:sum size, sumsizeprice:size wsum price by
sym from x;
                /- publish out new vwap values for this syms which have traded
                .u.pub[`vwap; select time:`time$.z.z, sym, vwap : sumsizeprice % sumsize
from .store.vwap where sym in x[`sym]]];}

/- Initialise the CTP
.u.init[];

/- Connect and subscribe to the tickerplant
h : hopen`::5010
h(`.u.sub;`trade;`)
```

The upd function

The upd function is always called with two parameters – the name of the table being updated (first parameter) and the table itself (second parameter). There isn't any reason why the upd function shouldn't call other functions. Below is a common structure for upd:

```
upd:{[t;x]
        if[t=`trade;
                do_something[]];
        if[t=`quote;
                do_something_else[]];
        .u.pub[t;x]};
```

Subscriptions

The previous examples cover subscriptions where a CPT is interested in all tables and all symbols. If a subset of tables or symbols is required, the following structures could be used

```
/- open the connection
h:hopen`::5010

/- the subsets
tabs:`trade`quote`depth
syms:`VOD.L`BARC.L`ENI.MI

/- all tables, all symbols
h(`.u.sub;`;`)

/- all tables, subset of symbols in each table
h(`.u.sub;`; syms)

/- some tables, all symbols
{h(`.u.sub;x;`)} each tabs

/- some tables, some symbols
{h(`.u.sub;x;syms)} each tabs
```

It is possible to subscribe to symbols that don't exist in a table. The client will receive the updates for the subset of symbols which exist.

Subscriptions overwrite each other. If a new subscription is to be added, the previous subscriptions need to be re-sent. For example :

```
/- subscribe to some symbols in the trade table
h(`.u.sub; `trade; `VOD.L`BARC.L)

/- We want to add a new subscriptions to RDSa.AS and NOK1V.HE, but
/- h(`.u.sub; `trade; `RDSa.AS`NOK1V.HE)
/- will not work as will no longer be subscribed to VOD.L and BARC.L
/- We must do
h(`.u.sub;`trade;`VOD.L`BARC.L`ENI.MI`NOK1V.HE)

/- resubscribing to all will overwrite individual table subscriptions
/- this will subscribe to all tables and syms
h(`.u.sub;`;`)

/- but then re-subscribing to an individual table will leave all other
/- tables still subscribed.
/- this will leave us subscribed to all symbols and all tables, except
/- quote where we're only subscribed to STL.OL and ABBY.L
h(`.u.sub;`quote;`STL.OL`ABBY.L)
```

Subscribing to more than one tickerplant

The above subscription method only allows connections to be made to one tickerplant. There is no reason why a CTP cannot subscribe to more than one tickerplant, provided the table definitions don't differ. Problems could arise if a subscription is made to two tickerplants both with a table called quote where quote is defined differently on each.

```
/- subscribe to more than one tickerplant
/- all tables, all symbols.
subscribe_to_tp:{
        h:@[hopen;x;0];
        if[h=0; -1"Subscription failed - couldn't connect to TP on ",string x; :()];
        h(`.u.sub;`;`);}

subscribe_to_tp each `::5010`::8010
```

If different symbols and/or table combinations are required, an approach similar to that outlined in the previous section could be used.

Priming

Sometimes it may be necessary for a CTP to prime itself from either the RTD or HDB (or both). This can be especially useful in the case of disaster recovery where a CTP may have to re-sync itself from the RTD after a crash.

The following code could be used by the VWAP example given in section **A VWAP Publisher**

```
/- Connect to an rtd, initialise vwap table for all syms
initialise:{[rtd]
        h:@[hopen; rtd; 0];
        if[h=0; -1"Initialisation failed - couldn't connect to RTD."; :()];
        .store.vwap,:h"select sumsize:sum size, sumsizeprice:sum size*price by sym from
trade";}

/- Connect to an rtd, initialise vwap table for some syms
initialise_syms:{[rtd;syms]
        h:@[hopen; rtd; 0];
        if[h=0; -1"Initialisation failed - couldn't connect to RTD."; :()];
        .store.vwap,:h({select sumsize:sum size, sumsizeprice:sum size*price by sym from
trade where sym in x};syms);}

/- initalise from the supplied rtd
initialise[`::5011]
```

Modifying .u.sub

The standard way subscribe to a CTP is by calling the .u.sub function. However, it may be necessary to do other things when a new subscription call is made. Two things (amongst others) which might be useful are

1. Publishing a snapshot of data;
2. Allow the CTP to check the subscription requests and prime and initialise itself for any symbols it has not subscribed to.

Modifying .u.sub directly isn't a good idea. A better approach is to define a wrapper subscription function which handles any extra functionality required, meaning that other subscribers who do not require the extra functionality can remain unaffected.

Publishing a snapshot

The following structure could be used in the VWAP example for publishing a snapshot of VWAP data whenever a subscriber calls subscribe. In this example, VWAP is the only table for which snapshots are possible as no data is stored for the other tables.

In this example, subscribe function rather than .u.sub should be called by the client.

```
subscribe:{[t;x]
         /- set up subscription as before
        .u.sub[t;x];
         /- if the client is interested in vwap, send a snapshot
        if[t in `vwap`;
             (neg .z.w)(`upd;`vwap; select time:`time$.z.z, sym, vwap : sumsizeprice %
sumsize from .store.vwap where sym in x]];}
```

Updating subscription lists

This sort of functionality is useful for reducing the amount of processing by cutting down the subscription list. The idea is to define a list of symbols statically which will (generally speaking) be the universe of interest, but to still have the ability to extend the universe dynamically throughout the day, for example when a new stock is being traded or monitored.

The subscription function could be used to check that the CTP has already subscribed to the main tickerplant for all the symbols which the subscription request is for. If there are missing symbols, the CTP will initialise itself from the RTD for these symbols (assuming a suitable function is defined as seen in section **Priming**) and re-subscribe to the main tickerplant.

```
/- The list of symbols of interest initially
sub_syms:`VOD.L`BARC.L

/- Connect and subscribe to the tickerplant
h : hopen`::5010
h(`.u.sub;`;sub_syms)

/- function to update a subscription list
/- if any of the symbols in syms aren't in sub_syms,
/- then intialise the data from the rtd, and re-subscribe.
update_subscription:{[syms]
        diff_syms: syms except sub_syms;
        if[not () ~ diff_syms;
                sub_syms ,:: diff_syms;
                initialise_syms[diff_syms];
                h(`.u.sub;`;sub_syms)]}

.u.subscribe:{[t;x]
        update_subscription[x];
        .u.sub[t;x];}
```

.z.pc

When loading u.k, .z.pc (close connection handler) is overwritten. .z.pc is defined to remove subscriptions from clients which drop their connections.

If custom logic is required in .z.pc, the u.k logic should be maintained, otherwise the CTP will not handle client disconnections properly. For example:

```
/- Close connection handler - drop the handle from the signal_handles list
/- .z.pc is already defined in tick.k so must be careful not to
/- overwrite the definition - must amend it instead

.tick.pc : .z.pc;
.z.pc : {custom_logic[x];  .tick.pc[x];}
```

Real-Time Subscribers contained in c.q

A Kdb+ real-time subscriber can be started from c.q using the following command:

q tick/c.q {config} [host]:port[:usr:pwd] –p [port]

The **config** parameter indicates the subscriber to use from the script c.q. This configuration defines how tables are updated when update messages are received from the ticker-plant. The **[host]:port** option specifies the host and TCP/IP port that is used to subscribe to the ticker-plant, and also if the ticker-plant is password protected we must also include the usr name and password. As usual the –p option specifies the tcp/ip port that a client must use in order to connect to the subscriber through either through the web interface or from another q process or external applications.

A few example subscribers for both the taq and sym ticker-plants are included in kdb+/tick installation in the script c.q. These show how to implement specialized subscribers that only use the received data to update summary tables or specific analytics.

Subscribers can be started as described above:

q tick/c.q all :5010	All the data-like **RTD**
q tick/c.q last :5010	Last tick for each sym
q tick/c.q last5 :5010	Last tick for each 5 minute bucket
q tick/c.q tq :5010	All trades with then current quote
q tick/c.q vwap :5010	VWAP for each sym
q tick/c.q vwap1 :5010	Minutely VWAP
q tick/c.q move :5010	Moving Vwap
q tick/c.q hlcv :5010	High Low Close Volume
q tick/c.q lvl2 :5010	lvl2 book for each sym
q tick/c.q nest :5010	nested data, for arbitrary trend analysis
q tick/c.q vwap2 :5010	vwap last 10 ticks
q tick/c.q vwap3 :5010	vwap last minute

The script c.q can be easily extended to create further customized subscribers.

Failure Management

Backup and Recovery

kdb+ databases are stored as files and directories on disk. This makes handling databases extremely easy because database files can be manipulated as operating system files. Backing up a kdb+ database can be implemented by using any standard file system backup utility. This is a key difference from traditional databases, which have their own back-up utilities and do not allow direct access to the database files. The kdb+ use of the native file system is also reflected in the way it uses standard operating system features for accessing data (memory mapped files), whereas traditional databases use proprietary techniques in an effort to speed up the reading and writing processes.

Kdb+ databases are easily restored by retrieving the relevant files from the backup system. Restored databases can be loaded just like any others because they are simply file system entities.

However, for high availability, an active-active backup (hot standby) setup is advised.

Active-Active Backup

The aim is of an active-active backup is to maintain two complete systems which are identical. If one machine goes down, the other one can be used instead. There is no real alternative when a high availability solution is required.

The fact that an active-active back-up is being used should be kept transparent from the user. A good way to do this is to provide connection methods to the user which send a query to one database, and upon failure sends a query to the other database and returns the results. kdb+ clients can use the .z.pc callback as a notification when a connection drops.

As both systems are identical, clients could use either machine as primary. However, in the event of an intra-day data feed failure it is important to disable any affected process (RTD, ticker-plant clients etc), as it is not desirable for the production servers to be providing different data.

Most users of kdb+ employ an active-active backup.

Failure Recovery

Failure recovery is possible to different degrees, depending on which process fails.

Best Effort Recovery Strategy

In an active-active setup, one machine crashes, a best efforts recovery solution can be deployed to attempt to bring the machines back into line. This is outlined below.

The recovery is implemented from a separate q process. This process is responsible for creating two files – a recovery script (in q) and a recovery log. The recovery script contains a set of commands to be run on the recovering RTD and the recovery log contains the missing data. One of the commands in the recovery script will be to load the recovery log.

1. Machine A crashes, losing all kdb+ processes.
2. Several minutes / hours later, machine A recovers.

3. All the kdb+ process are restarted on Machine A. The RTD in Machine A now has a set of information with a gap in it, but will receive all subsequent data.
4. Machine A RTD is queried for the minutes where it has in-complete information.
5. Deletes are written to the recovery script which ensure all the data in the missing minutes are removed, and including several buffer minutes at either end of the crash period.
6. Query Machine B RTD for the missing minutes worth of data, for every table that has a time and sym column (i.e. came from the ticker-plant).
7. Write this missing data to the recovery log.
8. Write a command to the recovery script to load the recovery log.
9. Run the recovery script in Machine A RTD – this will load in all the missing data.
10. Machine A now has two logs – the normal ticker-plant log and the recovery log. These are both required until after the data has been successfully saved down at the end of the day.
11. If Machine B now crashes, Machine A has a reasonable set of data. When Machine B comes back up, it can use Machine A for recovery.

It is still possible that Machine A will have some bad data (either missing information or duplicates) at either end of the recover period. This is due to the two machine clocks not being exactly in sync. One option would be to re-point clients to Machine B RTD for the rest of the day.

kdb+ clients which run on the same machine will also have failed. These clients can be designed to re-start intra-day if required, by querying the RTD or HDB for any data which has been lost.

If the recovery procedure outlined above is used after a failure of Machine A, and there isn't a subsequent failure of Machine B, the complete log file for the day should be copied from Machine B to Machine A after day-end. This log should be replayed to ensure that Machine A has an accurate set of historic information. See section **Replaying a Log After Day End** for details.

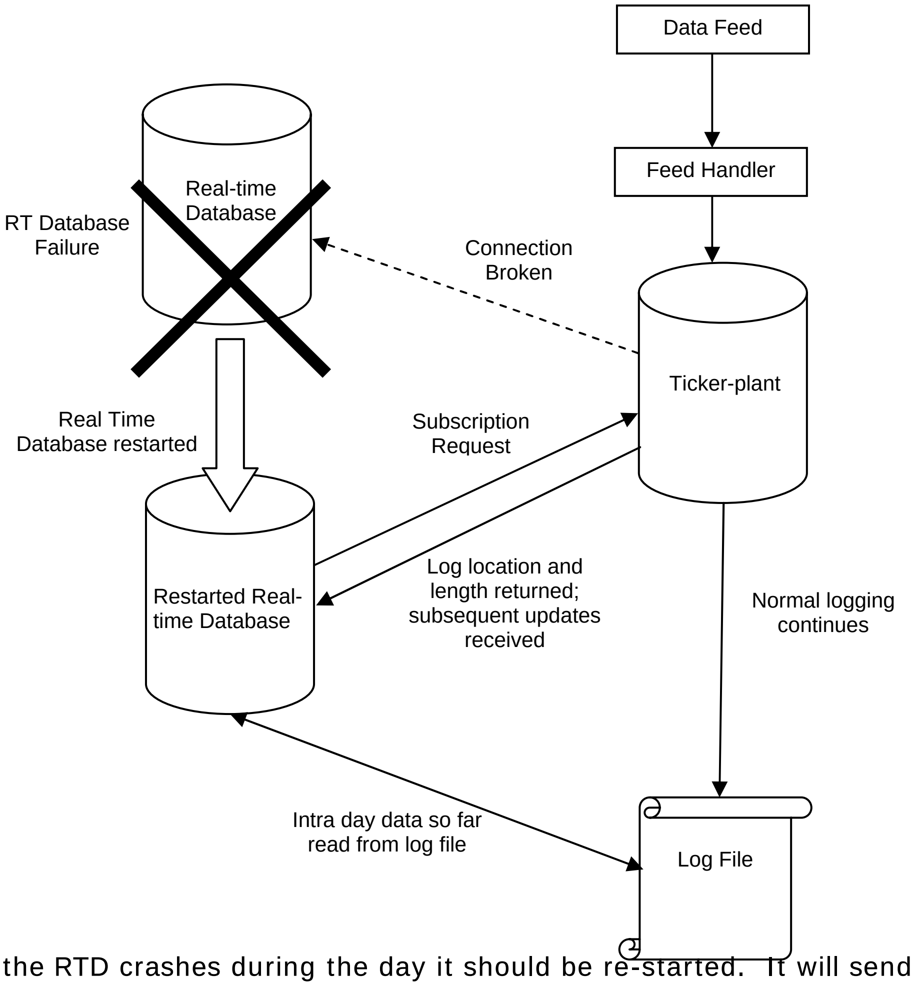
Ticker-plant Failure

If the ticker-plant fails, there will not be any information propagated to the rest of the system. It may be possible for clients to switch over to the back-up ticker-plant. As all connections to and from the ticker-plant will have been broken, the cleanest approach may be to re-start the entire system.

A recovery of the data would be possible, in a similar fashion to that outlined in section **Best Effort Recovery Strategy**.

Real Time Database Failure

The RTD can easily recover from an intra-day failure, with the only knock-on effect being the unavailability of database for a short period of time.


Data Feed
Feed Handler
Real-time Database
RT Database Failure
Connection Broken
Ticker-plant
Real Time Database restarted
Subscription Request
Restarted Real-time Database
Log location and length returned; subsequent updates received
Normal logging continues
Intra day data so far read from log file
Log File

If the RTD crashes during the day it should be re-started. It will send a subscription message to the Ticker-plant, which will return the location of the log file and the number of lines to read. The RTD will then replay the log file, and receive all subsequent updates from the ticker-plant, ensuring it has a complete set of information.

Towards the end of the trading day the RTD may take a few minutes to re-start as the data captured for the day may extend to many gigabytes.

It is important that the RTD is running at day end, as the RTD process is responsible for saving down to the Historic Database. If the RTD end-of-day save is not completed, the HDB will be missing one day's worth of data. It is easy to replay and re-save this data from the ticker-plant log file, but until this is done the Historic Database should be disabled as it has an in-complete set of information.

Historic Database Failure

Similarly to the Real Time Database, any intra-day failure of the Historic Database will only have the effect that clients will not be able to retrieve information.

If the Historic Database fails it should be re-started. This should be instantaneous at any time of the day.

Feed Handler Failure

If a feed handler fails, the data which it subscribes to and parses from the data feed will not be published to the ticker-plant. The feed handler should be re-started, and a recovery procedure similar to that outlined in section **Best Effort Recovery Strategy** may be possible.

Failing that, the log should be replayed and re-saved at the end of the day on the machine with the incomplete set of data.

In the event of a feed handler failure, clients will still be able to connect to the RTD and run queries. Therefore the RTD should be disabled until the missing information is retrieved. Any clients which connect directly to the ticker-plant should also be disabled.

Machine Failure

If a machine fails, it should be re-booted and all processes re-started. The ticker-plant log will have a complete set of data up until the time of the failure, so a recovery similar to that outlined in section 1.2 should be possible.

No clients will be able to retrieve or receive information during this time.

Network Failure

If the network fails, no processes will be able to query the database, and no new data from Reuters will be captured by the database. This should be treated in the same way as a machine failure.

Replaying a Log After Day End

In an active-active configuration it is important for both machines to have the same set of history. Occasionally the data collect for one day can differ between machines. This could be because of:

1. Development changes made to the database schema
2. Changes made to the universe of instruments sourced.
3. New data being captured.
4. An intra-day failure.

In this situation it is necessary to replay the log file.

NOTE: Date partitions should not be directly copied from one database to the other. All columns of type varchar held in the database are enumerated against the sym file held in the top level of the HDB. In a active-active set up it is extremely difficult to guarantee that these sym files (and therefore the enumerations) will be the same, so date partitions should nerver be copied directly.

Replaying a log is a fairly simple operation. It involves reading the loading up the schema file, setting the upd function, reading and executing the log using –11!, then saving to disk. The following is a small example:

```
save_date : 2006.10.01
logfile : `:tic/schema2006.10.01
hdb_dir : `:tic/schema
hdb_port : `:5012
```

```
/- load the schema file
\l tick/schema.q

/- set upd
upd : insert

/- streaming execute the log
-11!logfile

/- use .Q.hdpf for save
.Q.hdpf[hdb_port; hdb_dir; save_date; `sym]
```

However, it is common for the RTD to contain custom logic. If this is the case, it would be beneficial to maintain the customisations in a separate file, so it is easily loaded by any replay script. See section **Customising r.k** for details.

Recovering a Corrupt Log

An intra-day ticker-plant failure can sometimes corrupt the log file (for example, if the server crashes as the ticker-plant is in the process of writing to the log file). A corrupt log will not be replay-able by either the RTD or replay script.

Sometimes it is possible to recover the log. A log can be recovered as follows.

1. Create a upd function which increments a counter and writes the counter value out to a file on disk.
2. Replay the log using –11!. When the q process hits the corrupt record it will not write the counter to disk and will crash out.
3. Create a new upd function which reads each log message and saves it in memory.
4. Replay the log again, up until the last value of the counter. This can be done with :

    ```
    -11!(counter_value; `:log2006.10.01)
    ```

5. Write the non-corrupt log messages out to a new log file.

This new log file can then be replayed in the normal way.

Other Considerations

Performance

The performance of the kdb+/tick varies with the characteristics of the system such as the processor's speed, the platform-specific TCP/IP implementation, the database schema, and the feed handler.

As an approximation, on modern production servers, each ticker-plant and RTD can handle 1,000,000 individual messages per second - more than enough to deal with all trades, quotes, level2 quotes or options. However, if the records are bulked together either by the feed or the ticker-plant, this number increases dramatically.

The ticker-plant can handle many real-time subscribers, and the RTD can handle many clients.

Queries on in-memory data are done at up to 10,000,000 records per second.

For disk-based data, querying is carried out at 1,000,000 ticks per second

Most queries execute in milliseconds on the kdb+ real-time databases. It is possible to time the query evaluation by preceding the query statement with "\t ", as in

```
\t select avg size by sym from trade
```

Using multiple ticker-plants

One ticker-plant should be sufficient to capture data from multiple market data streams. However, sometimes it is necessary or convenient to capture different data into different ticker-plants.

The data captured should be divided up in whichever way is most convenient for query development. For example, if the intention is to capture data from global equities markets, it may be beneficial to have three distinct ticker-plant instances, for (Austral)Asian, European and American exchanges, all of which run in their own timezones.

The associated RTDs / HDBs are then typically queried either individually or through the use of a gateway server.

Memory Usage

The purpose of the RTD is to capture everything and write out the tables at the end of the day. It holds this data in memory, so the RTD will use a lot of memory. The capture and end-of-day processing will take 4 to 6 times the size of the log.

Assuming the RTD is started on a Monday morning and runs 24 hours a day, the RTD memory usage will grow on the first day of the week to a peak and remain at about the same level for the rest of the week. If a day of larger volumes occurs, the memory usage will increase again. The memory usage will not automatically decrease after the end-of-day save as kdb+ does not release the free memory. However, it is possible that memory will be returned to the OS if swapping occurs.

The kdb+ ticker-plant doesn't hold any data in memory, so memory usage will be very small.

The HDB will use very little when it starts up. The amount of memory allocated to the HDB will depend on the queries being run on the database and how they are implemented. Generally the HDB will use less memory than the RTD.

Memory usage of other processes e.g. real-time subscribers, feedhandlers etc., will vary depending on implementation.

It is highly recommended that all implementations of kdb+/tick are on a 64 bit system as 32 bits systems have a memory addressability limit of 3 to 4 Gb. On most production systems where kdb+ is deployed this is simply not enough.

Appendices

Appendix A: Troubleshooting Kdb+/tick and Kdb+/taq

Memory

To ensure optimal use of the system, ensure that:

- no swapping is taking place
- no process is close to using up all available addressability

From the q console it is possible to check the amount of memory being used by typing \w. The first number returned represents memory in use. The second number returned represents total memory allocated. The third number is the amount of memory mapped data and the fourth number is the maximum amount of memory used so far at any one time.

CPU

CPU usage is primarily dependant on the number of ticks being captured. It will also be affected by the use of logging or the number of real-time subscribers etc.

In general, ticker-plants that capture the main US Equities can operate at less than 10% of one CPU with peaks of up to 30% at market open. Any regular peaks higher than this, or a tendency for CPU usage to increase during the course of the day, could indicate a problem.

Disk IO

File-write speed is critical if transaction logging is being used in the ticker-plant. File read speed is often the dominant factor in the time taken by queries against the historical database. It is therefore important to ensure that the drives being used are sufficient for these tasks and investing in fast hard drives will provide substantial benefits when using kdb+/tick and kdb+/taq.

The minimum recommendation is a hard-drive capable of 100MB/s. In general it is difficult to test read speeds due to caching, but write speeds can be tested with the commands below. If transaction logging is to be used it is also worth testing that file append operations do not degrade as file size increases, since the log file can be hundreds of megabytes in size by the end of the day and slow logging could result in ticks being dropped at market close.

```
\t .[`:c:/foo;();:;key 25000000]        /write 100MB
\t .[`:c:/foo;();,;key 25000000]        /append 100MB
```

Errors

Most feed handlers will generate error logs when a problem occurs. With the Reuters feed handler, a file with a name of the form **SSL_elog5418** will be created in the ticker-plant's current directory (the number at the end is the process id of the process which created the file).

Useful error information may also be available through the sink distributor or data source.

Additionally, it is generally useful to redirect standard output and standard error from the ticker-plant to capture any messages generated by kdb+.

Messages

The best way to monitor messages being received by the kdb+ databases is to override the message handlers .z.ps, .z.pg and .z.ph. These can then be used to log all messages received or sent.

Another useful place to add a trace (using 0N!) is in the function f in ssl.q when using the Reuters feed handler – this function will receive all of the raw messages from the feed (this can be a lot of output though).

kdb+ Licence

An error message of

```
abort: k4.lic
```

indicate a problem with the license or its location. kdb+/tick and kdb+/taq will not function correctly without a valid license file ‘k4.lic’.

A full license is provided by Kx Systems when the product is purchased; a temporary one is supplied for an agreed evaluation period or Proof of Concept. This file must be located in the directory that kdb+ home directory. This is usually $HOME/q under Linux and Solaris, or c:\q under Windows.

The q home directory can be specified by setting QHOME, and add the q license file directory can be specified with QLIC. The default for QLIC is QHOME.

The license owner and expiry data should be printed out when kdb+ is started.

Appendix B: Technical Implementation of Ticker-plant

The source code for the ticker-plant is provided with the distribution, to allow customization of its behavior as required. The 2 core files are 'tick.k' and 'u.k', which should be present in the directory. tick.k loads u.k.

Variables

.u.d: stores the date at start-up. This is the value inserted into the date column when the data is saved and will be used for the name of the new partition.

.u.L: the name of the current log file.

.u.d: today's date in local time.

.u.i: the count of the log file, i.e. the number of messages of the form (`upd;t;x) that has been appended to the log file.

.u.l: this is the handle to the log file and is used to append messages to it.

.u.t: all tables in the current ticker-plant process.

.u.w: this is a dictionary containing the connection handles and sym subscription lists for each table in .u.t for all subscribers.

Functions contained in u.k

.u.init[]: initializes the ticker-plant. An entry is created in .u.w for each table contained in the top level namespace of the tickerplant process.

.u.del[handle;table_name]: deletes a connection handle from the subscription list .u.w for the supplied table name.

.z.pc[x]: calls .u.del with the supplied handle for each table in .u.t.

.u.sel[table;symlist]: gets the data from a table for a specific sym list. If the sym list is `, then all data is retrieved.

.***u.pub[table_name; data]***: publishes the data to each of the connection handles specified in .u.w which have subscribed to table_name. It only publishes to clients the (subsets of) symbols that they have subscribed to. It uses the connection handles to call the client upd functions.

.u.sub[table_name;symlist]: when a client subscribes it (asynchronously) calls this function with the table and sym list as arguments. This function then adds the process handle of whoever called it together with the sym list to the subscription list .u.w. It also immediately returns to the caller a two element list, consisting of the name of the table they subscribed to plus the table schema. The caller will then be updated asynchronously via the pub function.

.u.end[date]: this is the message sent from the ticker-plant to each of its subscribers and the RTD at day-end. The default functionality of a processing loading u.k is to propagate the end-of-day message to each of its clients.

Functions contained in tick.k

.u.endofday[]: defines the end of day behaviour. This calls .u.end with the current date, increments .u.d, and creates a new log if logging is being used. The name of the new log is stored in .u.ld.

.u.ts[datetime]: checks if the supplied datetime has a date component greater the .u.d. If so, .u.endofday is called.

.u.upd[table_name; data]: all inserts to the ticker-plant should be passed in through this function. It firstly checks if day end has occurred - if so it calls .z.ts immediately. Otherwise it performs the following steps:

1. If there is no time column on the incoming data it adds one;
2. An insert message is created;
3. The message is appended to the log and the count of the log incremented.

If the ticker-plant is running on a timer, it will then insert the incoming data into its table. If running in zero-latency mode, it will immediately publish the data.

.z.ts[datetime]: is a timer called every heartbeat. The definition of this function depends on whether the ticker-plant is running as zero-latency or not.

If zero latency, .z.ts is defined to call .u.ts every second.

If not zero latency, .z.ts is defined to publish the contents of each table, flush each table, then call .u.ts. The frequency depends on \t.

.u.ld[date]: this is the logging function which is called with .u.d as argument. It creates the log (named destianation/schemaDATE) if is not there and initializes .u.i to 0. If the log already exists, it sets .u.i to be the count of it. It then returns a handle to the log file for appending.

.u.tick[schema; destination]: is the main function call to initialise the ticker-plant.

Firstly, it calls .u.init. It then checks that each table in the top level namespace of the process contains a time and a sym column. If not, a ‘timesym error is thrown.

Next, it applies a grouping on the sym column to each table. .u.d is then intialised to the current date.

If the second parameter, destination, is supplied, it sets .u.L to be the log file name (destination/schema..........). It then calls .u.ld for the current date and assigns the returned handle to .u.l.

Appendix C: Bloomberg Ticker-plant

The Bloomberg ticker-plant is designed to handle Bloomberg equity quotes. The Bloomberg ticker-plant can be started using the following command line (with the usual options):

```
q tick/bb.q [host]:port[:usr:pwd]
```

In standard configuration it connects to a ticker-plant running on port 5010.

The default schema is:

```
trade:([]time:(),sym:(),price:(),size:())
```

```
bid:([]time:(),sym:(),bid:(),bsize:())
ask:([]time:(),sym:(),ask:(),asize:())
a:([]time:(),sym:(),value:(),type:())
```

The Bloomberg feedhandler uses bb.dll.

Unfortunately the Bloomberg feedhandler is much more difficult to customise than the Reuters Triarch feedhandler. As such, it will not be discussed further in this document.

If a customisation of the Bloomberg feedhandler is required, contact First Derivatives.

Appendix D: The Reuters Feed Handler

The 4 standard Reuters ticker-plants all work in much the same way with regards to the feed handler.

There is one q script for specifying the fields to be captured-ssl.q and the schema of the ticker-plant is determined from the command line arguments, for example to start a taq feed the following command would be issued

```
q tick/ssl.q taq 5010
```

The script **ssl.q** makes use of the C library **ssl.dll** or **ssl.so** which must be located in the folder q/OS i.e. in the case of windows this would be in c:/q/w32

The complete Reuters messages are passed back from the C library to K (as the argument to the function **f**) and it is therefore possible to modify the fields captured without requiring any changes to the C code.

ssl.q

Callbacks from the C library
close
dis
rec
stt

These keep state and notifiy of disconnections etc.

Functions and Variables

d - schema
fi - map of reuters fids to the corresponding formatting functions
h - handle to the ticker-plant process
qf - quote formatting functions and are obtained from fi@qi
qi - quote identifiers-list of fids to capture from the feed
tf - trade formatting functions obtained from fi@ti
ti - trade identifiers-list of fid to capture from the feed
qj / tj – allows the differentiation between trades and quote
sym - the RICS that will be subscribed to
f - default callback for c library
g - map of fid to values from string received from the reuters feed
k - function that parses the data
sf - for taq-gets sym from RIC
sub - K function mapping **ssl** entry point to C library-sends the subscription message to **reuters** - dynamically loaded from ssl library
cond/mode - dictionary for formatting condition and mode codes

Kdb+/taq – Historical Database

What is Kdb+/taq?

Since 1993 the volume of NYSE TAQ data has grown twenty-fold. Where one CD used to hold a month of data, now it holds only one day's worth. Not surprisingly, the tools most traders use to load and analyze this valuable NYSE data are struggling to keep up with higher volumes. Extracting the multiple end-of-month CDs & DVDs to a relational database can take hours or even days to finish. If you're extracting to flat files, your analytical abilities are severely restricted. Kdb+/taq solves the problem of loading, extracting and analyzing NYSE TAQ data.

With kdb+/taq it takes about 15 minutes to load a month's worth of NYSE TAQ data. At present, daily TAQ data files contain approximately 5GB of quote data (45-55 million quotes), and 500MB of trade data (5-8 million trades) in their unzipped format, capturing 8000+ different symbols. With default schema, this is a little over 1.8GB of storage in Kdb+ database (or over 35GB per month). Once stored, data can be queried at 1 million ticks per second per CPU. In addition kdb+/taq customers can FTP the NYSE TAQ data directly into kdb+/taq at the end of every trading day in minutes, enabling them to perform relational time-series analysis on billions of NYSE TAQ data records and obtain results in seconds.

Kdb+/taq is built on kdb+, the ultra high-performance database from Kx Systems. Kdb+ provides extreme database efficiency and performance. It uses inverted (column-based) storage for high search efficiency, and its query language, **q**, includes support for time-series analysis. If your analytics have been slowed down by multiple-joins and other database overhead, you will find that kdb+/taq returns results in seconds - even when searching a billion records.

A test version of the historical database can be generated for evaluation purposes using the script **‘tq.q’** available at http://www.kx.com/k4/taq

Some additional information and useful scripts are available at http://www.kx.com/k4

Hardware Requirements

o/s: solaris64, linux64(opteron or nocona)
cpu: 2+ (2 sets of disk arrays per cpu)
ram: 16GB+ (8GB+ per cpu)
disk: 2TB+ (U320SCSI 10000rpm+ drives)

With the historical database the hard drive can be the most significant factor in query execution times. 10000+ RPM SCSI would be a minimal recommendation. EMC drives are another common choice for storing the database. In general, a good specification should provide read-write speeds of around 40MB/s.

The complete TAQ history is currently about 25 billion+ trades and quotes (>1TB) and is growing at a rate of 50 million records per day (>2GB).

The kdb+ storage factor is about 1. Get 2 times as much disk for raid5, staging and scratch space. For example the following set-up could be used for the full TAQ database (2004):

HP DL585 with a 2 CPU AMD Opteron with 16 GB RAM, running 64 bit Linux

2 MSA500 G2 arrays with (7+7)*146GB 10k RPM drives, providing 3.5TB useable space in 2 raid5 arrays.

Installation

Prior to installing kdb+/taq, kdb+ must be installed on the system. The installation of kdb+ creates the k4 directory in $HOME/ (UNIX) or in C:/ (Windows). For simplicity, both these directories will be referred to as k4/ throughout this manual.

The installation and license files for kdb+/taq must be obtained directly from Kx Systems (send e-mail to tech@kx.com). The license file must be copied into the k4/ directory. The kdb+/taq distribution file taq.k can be obtained from either Kx Systems or First Derivatives.

Running the Kdb+ TAQ loader

1. Copy the data files from CD or DVD (monthly data) or download them using FTP (daily files) onto a directory (**SRC**) on the server.

These files should consist of:

Monthly: One dividend and one master file per month (e.g. **'D200202.TAB'** and **'M200202.TAB'**). A number of trade and quote files – these should be in pairs with matching IDX and BIN files e.g. '**T200202A.BIN'** and **'T200202A.IDX'**). It is best not to rename these files as the loader makes a number of assumptions based on the file name.
Data before and after 2000 should be loaded separately as there were some format changes. The default loader will also only load 1 month's dividend and master files at a time.

Daily: Zipped files containing **'taqtradeYYYYMMDD.txt'**, **'taqquoteYYYYMMDD.txt'** and **'masterYYYYMMDD.txt'**. The master file is a separate download and must be purchased separately if required. There are 2 available downloads for the quote data - a ZIP archive containing the single **'.txt'** file or an archive with multiple files. This is because the single quote file increased to more than 2GB in size, which cause problems with some applications. The daily files must be unzipped prior to loading.

2. Execute the load command.

```
>q taq.k SRC DST
```

where **SRC** is the directory where the raw data files reside and **DST** is the directory where the data will be written to. This command will load all the files it finds in **SRC** to **DST/taq.**
Kdb+ loads, indexes and stores about 1GB per minute per cpu.

If you want kdb+ to run in parallel:

a) put a list of directories(different drive arrays) in DST/taq/par.txt
b) q DST/taq -s N   where N is equal to number of drive arrays

There will be N slaves each reading their own drive array -- no contention. 2 disk arrays per cpu is about right (e.g. 2 cpu's and 4 array's above). Days are round-robin allocated. Multi-day queries run in parallel.

3. Start the database.

Once the load has completed, the historical database can be started with the command:

```
>q DST/taq –p 5014
```

This starts the historical database running on tcp/ip and http port 5014 and the data can be viewed through the web-viewer at http://localhost:5014. This can run forever as the load can send reset messages

4. Options

There are some options available with the loader:

```
q taq.k SRC DST [-corr] [-host host:port [host:port ..]]
```

-corr : load the rearranged incorrect records (default delete)

-host : send reset message to server(s)

Queries

The schema of the taq database is as follows:

```
trade:([]date;sym;time;price;size;stop;cond;ex)
quote:([]date;sym;time;bid;ask;bsize;asize;mode;ex)
mas:([]sym;date;name;cusip;wi;ex;uot)
```

The trade tables holds all the trades made and should be updated daily.
The quote table holds all the quotes made and should be updated daily.
The mas table holds the master information about each symbol and should be updated daily.

The data is indexed by symbol for one disk seek per day*sym*field. This yields about one million prices per second per processor (at 10,000 ticks per sym per day). In general query time can be estimated using the following simple formula,

T=10ms(6seek+4read) * (days/drives) * syms * fields

e.g. 10*1day*2sym*3fields=60ms for the following vwap query,

```
select size wavg price by sym from trade where date=2000.10.02,sym in`A`IBM
```

Data cached in memory is much faster. We always aim to restrict dates, symbols and fields as much as possible -- read as little as possible, for example

```
select time,price from trade where date=2000.10.02,sym=`IBM
```

is faster than

```
select from trade where date=2000.10.02,sym=`IBM
```

Also try and move as little data as possible - calculate in the server.

From java:

```
r=k("select size wavg price from trade where date=2000.10.02,sym=`IBM");
```

is much faster than pulling the data and calculating locally:

```
r=vwap(k("select size,price from trade where date=2000.10.02,sym=`IBM"));
```

To retrieve for sym.exchange, e.g. `AA.N

```
f:{[d;s]x:string s;update sym:s from select from trade where date=d,sym=`$-
2_x,ex=last x}
f[2000.10.02;`AA.N]
```

Corporate Actions

The TAQ CDs come with a basic corporate action detail containing details of symbol changes (e.g. mergers) and stock dividends. These are loaded into the historical database as the mas and div tables. Since this data changes on a daily basis, most people opt to store the raw trade and quote information and apply these changes, when necessary, at query time. This avoids having to potentially rewrite hundreds of gigabytes of data, as well as allowing queries to execute without the corporate actions where they are irrelevant.
One possible way of using this data is contained in

http://www.kx.com/k4/taq/adj.q

This script creates a number of new tables and functions that allow the symbols to be mapped and prices adjusted as required. The script makes use of binary searching to apply the changes to millions of records per second. The master is the last know value of the sym. Resulting queries are then of the form:

```
r:select last price,sum size by date,mas:ms sym from trade where sym=sm`CUZ
update AMD[mas;date]*price,size%AMD[mas;date]from r
```

sm gets the sym at a given date from the master and **ms** gets the master from the sym.

So the query

```
r:select last price,sum size by date,mas:ms sym from trade where sym=sm`CUZ
```

takes the master (latest) symbol `CUZ, and at each date finds the corresponding symbol, taking the last price and summing the size. Notice that the symbol is converted back to the master.

If price adjustments are done, a table mapping the (master) symbol and the date of the adjustment is created. This is called **AMD** in the adj.q script example. Therefore the statement

```
update AMD[mas;date]*price,size%AMD[mas;date]from r
```

will adjust the prices and sizes in the table r accordingly. The adjustment is done on the master symbol. There is further information in the script file itself.

Basic corporate action data is also available with the daily FTP download. However, the data that is supplied by NYSE is at present incomplete and many people therefore choose to acquire this data from other sources - it is then possible to develop a script similar to **‘adj.q’** to work with this data. Alternatively, it may be possible to purchase and store pre-adjusted data (though this is more difficult to maintain).

Handling Other Sources of Historical Data

In some cases it is necessary to write a custom script to create a TAQ type historical database based on alternative data sources. First Derivatives have developed a loader for the historical data provided by Tick Data and can also provide examples and advise for developing new scripts.

It is recommended that any new loader should create a structure similar to the TAQ loader in order to be consistent with new partitions created by the ticker-plant, as well as to optimize performance. This involves the following:

1. Partition data by date, with directory names of the form 2003.01.02 (kdb+ is optimized to recognize and take advantage of this type of partition, allowing it to disregard any directories irrelevant to any query which uses date as the first part of a where clause).

2. Each partition should be sorted by **sym** and then **time** (which means applying the sorting in the reverse order). Ensure that the **`p#** attribute is set on the **sym** columns (this attribute is how kdb+ detects that a column is sorted and indexed and that binary search can therefore be used to optimize query performance).

Kdb+/Tow – Replay Module

The basic aim of the replay module is to facilitate rapid replay of historical data (i.e. data in a kdb+ database residing on disk for example NYSE TAQ data). For the moment the module can only replay 1 days worth of data at a time, the reason for this is that symbols (ticker symbols that identify stocks) are not constant overtime and as such can and do change from day to day (although not very often). So if we try and replay multiple days of data for a particular stock and the stock symbol has changed within the timeframe of interest then the replay will obviously be wrong. We can try and get around this by using a master table with which we can build symbol chains but for now we will just look at the basic single day replay.

The replay module works by getting record handles to the required data. By doing this it means that a query need only be performed once- we store the record-handle indices and then use these iteratively to 'pull' or 'push' the chunks of data.

There are two options for configuring the replay server:

1. We can 'pull' the data from the server which means we get the total number of time buckets for the date, table and symbols we are interested in (N, say), get our record handles and then call pull N times where each time it is called we are returned a table with the required data. In the case of replaying both trades and quotes we would get a list containing a segment of trades and a segment of quotes.

or

2. We can actually 'push' the data through the kdb+/tick application one chunk at a time-in this way we can mimic a feed and also have multiple different subscribers receiving the data doing whatever we wish. This method is very useful for testing strategies and also stress testing subscribers.

In kdb+ we can do both of these at extremely high speeds- up to a million records a second. However, if using java/c as our engine for the strategies then we slow down a bit, so if possible the aim would be to try and write the strategies/clients in q and use these.

Implementing the Replay

1. Load the script

```
>q tow.k SRC
```

where SRC is the directory where the data to be replayed resides.
So in the case of the data resinding in c:/k4/data/taq this would be:

```
>q tow.k c:/k4/data/taq
```

2. Variables and Initialisation

a: start_time- default is 09:30:00
b: end_time- default is 15:00:00
m: time_slice- this gives the granularity of the replayed data, i.e. the ‘time chunk to be replayed’. 1 means that we will be replayed 1 second of data each time.

To initialize the server the following function is called. If you just want the default values there is no need to call this function.

timeinit[start_time;end_time;time_slice]

e.g:

```
timeinit[09:30:00;14:00:00;5]
```

This does not return anything- it simply sets the variables on the server.

3. Doing the Replay

doReplay[date;table;sym]

e.g.

```
doReplay[2001.10.01;`trade;`IBM`MSFT]
```

This call starts the replay. It firstly returns the number of records to be replayed, and the number of time buckets for our arguments. For a 1 second time_slice with start_time of 09:30:00 and end_time of 16:00:00, this will return 23400. This corresponds to the number of seconds between the start_time and the end_time

It then iterates through the records, either pushing them through kdb+/tick or just pulling them back one at a time.

Note that to push through kdb+/tick the tickerplant must be started on its usual port-5010 with no dst for the logfile and with a correct schema. For example:

```
>q tick.k taqR –p 5010
```

where taqR contains an identical schema as the historical data that is to be replayed. In the case of taq this is:

```
quote:([]time:`time$();sym:`symbol$();bid:`real$();ask:`real$();bsize:`int$();asize:`int$();mode:`char$();ex:`char$())
trade:([]time:`time$();sym:`symbol$();price:`real$();size:`int$();stop:`boolean$();cond:`char$();ex:`char$())
```

Then all subscribers work in the usual fashion. This is described in more detail in the First Derivatives Tickerplant manual.

Index

First Derivatives & Kx Systems

About First Derivatives

First Derivatives plc is a trading and risk management consulting firm specializing in providing services to both financial software vendors and large financial institutions.

Their team of consultants has substantial business knowledge of the capital markets sector combined with extensive experience in the development, implementation and support of large-scale trading and risk management systems. The company has particular expertise in the development of on-line trading and risk management software.

Partnership

First Derivatives has been working with Kx technology since 1998 and is one of two accredited partners of Kx Systems.

First Derivatives offers a complete range of Kx technology services:

- Evaluation Workshops

Training
Systems Architecture & Design

- K, KSQL development resources
- Kdb/tick implementation and customisation
- Database Migration
- Production Support

Products

Kdb+ a next-generation relational database that handles time-ordered data and financial analytics at real-time speeds.

Kdb+tick captures and analyzes billions of trades and quotes in milliseconds.

Kdb+taq solves the problem of loading, extracting and analyzing NYSE TAQ data.

About Kx Systems

Kx Systems (www.kx.com) provides ultra high performance database technology, enabling innovative companies in finance, insurance and other industries to meet the challenges of acquiring, managing and analyzing massive amounts of data in real-time.

Their breakthrough in database technology addresses the widening gap between what ordinary databases deliver and what today's businesses really need.

Kx Systems offers next-generation products built for speed, scalability, and efficient data management.

Kx Technology

Kx products offer a unique combination of:

- high-speed analysis of streaming real-time and massive historical data
- relational database capabilities for real-time and historical data
- fast querying of data - a billion records analyzed in seconds
- time-series functions built in
- open interfaces
- ability to combine streaming and historical data into one database for analysis and storage

Contact Michael O'Neill

First Derivatives House
Kilmorey Business Park
Newry, Co.Down, N.Ireland

Tel: +44 28 3025 2242
Fax: +44 28 3025 2060

Email: moneill@firstderivatives.com
Web: www.firstderivatives.com

Building Faster Solutions

in association with kxsystems