# *Kdb Programmers Manual*

Kdb Programmers Manual        Copyright © 2000 by Kx Systems, Inc.

# Contents

# 1 Introduction

The home page on the Kx website displays the following succinct description of their database products.

*High-performance analytic database servers*

This manual is an elaboration on that statement in technical terms. We begin with a closer look at the statement itself that will lay the groundwork for the technical chapters ahead.

## *RDBMS*

First of all, let us understand that *database server* implies, among other things, relational database server. Kdb supports ANSI SQL and the client interfaces that application programmers have come to expect: JDBC for Web clients and OBDC for Windows clients. Kdb databases can also be published and queried directly over the Web in HTML, XML and CSV format.

Kdb servers run on Linux, NT and UNIX.

## *High Performance*

Performance of database servers generally refers to through-put, of which there are several flavors. There is transaction processing, which generally means many (nearly) simultaneous, small interactions with a database. There is decision support, including OLAP, which involves substantial data analysis that may be either precomputed or real-time. And there are data warehouses, which hold vast amounts of data that can be mined (like an analytic server) or distributed to “retail centers”, databases that interact with clients. There are Kdb customers in all three areas that chose Kdb over the competition, in part because of its performance advantage.

How does Kdb performance excel in all three areas? While no single factor can account for everything, the most prominent one is also particularly relevant to this manual: from the beginning, the performance characteristics of modern computers and operating systems has been a major factor in the design and implementation of Kdb. We will see this in the organization of Kdb data, in the characteristics of the KSQL language and elsewhere throughout this manual. We will also see that Kdb performance is close to optimal.

## *Analytic Server*

An analytic database server is one where data analysis can routinely be performed on the server instead of on clients' workstations. The typical SQL database server is not an analytic server. SQL is simply not suitable for many computations and too slow for many others. However, Kdb extends SQL with

- order (in practice, usually time-dependent order),
- function-based access to indicative tables for roll-ups, and
- extensible queries using customer-defined analytics.

to create KSQL, a powerful computational language. KSQL, together with Kdb’s high performance, makes Kdb an analytic server.

Who needs an analytic server? Analyses of data on an SQL server that depend on order in the data – usually, date order – are typically done on clients' workstations. (For example, analyzing increments in sales from one day to the next requires data to be processed in chronological order). SQL does not recognize order. The most that a typical SQL server can do is return data to clients in a prescribed order, where the analyses is done.

There are many problems associated with performing analysis on the client side. First, client code is complicated enough managing user interactions and report generation without the additional complications posed by analytical code. Second, it is fairly typical that the analytical results are relatively small compared to the amount of data required to generate them. If the computations are done on a server then only the results must be sent to the client, while large amounts of data sent to clients for analysis can clog networks and workstations.

In contrast to a typical SQL database, Kdb tables are ordered and the KSQL language works with ordered tables. You can ignore the order and use SQL on Kdb databases, but you can also have an analytic server by using KSQL. In addition to order, KSQL is extensible. Customers can define their own specialized analytics that can be used in KSQL queries as if they are KSQL primitive functions. KSQL is SQL with order and extensibility.

## *Kdb Architecture*

There are three architectures for Kdb databases. The first level is *virtual memory* databases, or simple *memory* databases. Virtual memory databases are limited only by the practical limits for virtual memory on the server, about 1.5 gigabytes for NT and Linux and 3 gigabytes for UNIX. These databases require the least maintenance.

The second level of architecture is called *shuffle* because it generally marks the point where segments of the database are shuffled in and out of virtual memory on demand. Tables in Kdb are stored in column order; in particular, an integer column with N items requires 4N bytes of storage and a floating point column of that length requires 8N bytes. The shuffled database segments are columns of designated tables. There is no limit to the size of a shuffle database other than disk space. There is, however, a practical limit on the size of table columns: for optimal performance, all columns required for each computation must fit entirely in real memory[1]. This requirement still allows for very large tables in large shuffle databases. For example, one Kdb customer queries tables of 35,000,000 rows (and growing) and 200 columns. In this particular application, the shuffle architecture can accommodate a database that is 50 times larger than the memory architecture.

The third architectural level is called *parallel*. The only limit on the size of a parallel database is the amount of disk memory on the server. Conceptually, parallel databases are collections of virtual memory or shuffle databases.

Given modern high-speed sequential disk access, real memory caching and the inverted, column-oriented layout of Kdb tables, the performance when data on disk must be accessed can approach that of a memory database.

## *Kdb Flexibility*

Kdb database systems are programmable systems. Kdb databases have the standard client interfaces, JDBC and ODBC, for sending KSQL and SQL commands to servers and receiving results. Kdb also has its own interface, KDBC, for remote database management. Using KDBC, customized processes can be introduced into a database system that modify default Kdb server behavior such as transaction logging and client access. For example, a KDBC client that is also connected to a real-time security price feed can peri-

1. The optimal performance requirement applies to virtual memory databases as well.

odically collect prices from the feed and send them to a Kdb server as bulk updates to achieve a one hundred-fold increase in the rate of updates, typically one hundred thousand updates per second instead of one thousand per second.

Custom analytics and KDBC clients can be written in C or K, the implementation language of Kdb. KDBC clients can also be written in Java and Visual Basic.

## *Other Kdb Information*

The examples in this manual go into sufficient detail to give a comprehensive picture of the Kdb databases and what must be done to customize and maintain them. Related notes, comments, performance numbers and application kernels can be found on the web, at www.kx.com/a/. The web files are maintained by Arthur Whitney, the architect of Kdb and CTO of Kx Systems. This manual is essentially an elaboration of that material, but does not cover everything.

## *Evaluation Kdb*

You will see is that it is relatively easy to create and use sample databases, and test their performance as well. Hopefully this will induce you to try Kdb on your data. It is easy (and fast) to download Kdb for an evaluation period; see the Kx website, www.kx.com, and “Production Kdb”, below, for more details. Contact Kx Systems, the developer of Kdb, if you have special requirements.

## *Production Kdb*

Kdb is downloaded from the kx website and installed in a matter of seconds. That's all there is to it. It's the same process as downloading the evaluation version. The only difference between the evaluation version and the production version is the license. The installation process installs an evaluation license unless a production license is present. Contact Kx Systems at www.kx.com for information on production licenses.

When the installation completes you will see a window entitled *trade.t* and a text file named *Kdb.txt*. The *trade.t* window is the Kdb (database) Viewer. *Kdb.txt* contains information about the Viewer, an example of a query construction and an example of creating a Kdb Web Viewer.

Kdb is rock solid, but there are a few simple things that can be done to make sure that the installation went OK. For example, the Viewer window has a *queries* button. Press it and a query entry panel will be displayed with two sample queries. Click on either one to evaluate it. Or go through the examples in *Kdb.txt*. If everything looks OK, you're done.

The download takes a few seconds, depending on your internet hookup. The installation (execution of *kdbsetup.exe*) is nearly instananeous.

## *Who Should Read this Manual*

The first manual in this series, Introduction to Kdb, is for professionals who do not necessarily know how to program. It shows how to build (memory) databases from the Kdb Viewer and how to construct elementary queries. The second manual, KSQL Reference, defines the KSQL query language. The sample databases in that manual are also memory databases. Using only these two manuals, complex, substantial databases can be built, maintained and accessed with no previous programming knowledge.

This manual – the third – is directed at a different audience, database administrators and programmers. It's focus is what they must know and do to create powerful Kdb databases. The next chapter, "Design and Performance", describes the three Kdb architectures, their performance characteristics, and indicates what must be done to use them effectively. The rest of the manual gives the details.

# 2 Design and Performance

The chapter gives a technical overview of the three Kdb architectures. Other notes of interest on this subject by Arthur Whitney, the architect and principal implementer of Kdb, can be found on the Kx Systems website at www.kx.com/a/kdb/. Some of those notes are elaborated on here.

All performance numbers given here were taken from the author's desktop computer, which has one 450MHz Pentium III processor and 128 mb of RAM. The numbers are used to illustrate methodologies for estimating Kdb performance and not to suggest actual performance on database servers.

## Memory Architecture

A significant part of this manual is devoted to programmers who customize Kdb database systems that are larger than 2 gigabytes. The Kdb memory[1] architecture is the one to use for databases up to 2 gigabytes. Kdb memory databases are the easiest to build and require the least maintenance. The size restriction is due to the fact that memory databases are loaded entirely into virtual memory when they are activated. (In that regard, make sure that every page file on the host computer is large enough to hold the entire database; if not, a *workspace full* message will be reported if the operating system assigns too small a page file to the Kdb server process.) Optimal performance is achieved by matching the amount of real memory on the host computer to the execution requirements of the most active queries of the database.

Every table in a memory database is arranged in column order, both on disk and in memory. Every column is memory-mapped when a table is loaded. Integer and floating-point columns are loaded – that is, mapped – instantaneously. Varchar items are hashed as they are loaded and therefore take longer to load – about 1,000,000 items per second. Not only are tables stored in column order, but integer and floating-point columns are stored as 1-dimensional C-language arrays. KSQL computational functions are implemented directly in C. Consequently, KSQL statements execute at near-optimal speeds. The actual speed can be closely estimated for any KSQL or SQL statement.

1. The word "memory", when not modified by "real" or "virtual", means virtual memory.

An important characteristic of Kdb memory databases is that all but a negligible amount of processing power in executing a KSQL or SQL statement is directly attributable to the processing power required by the components of the statement: namely, the CPU cycles required to execute the statement are – very nearly – the sum of the CPU cycles required by the components. There are no hidden execution costs that are not attributable to the host operating system. For example, the execution costs depend only on the columns that appear in the statement and the ways they are used; the number of unused columns in the table doesn't matter.

The practical upper limit on memory databases is 1.5 gigabytes on NT and Linux and 3 gigabytes for UNIX. This estimate includes space for temporary storage requirements while statements are being evaluated and transaction logs, if any. The size of temporary results is most important because all columns required for each computation must fit entirely in real memory for optimal performance. The objective is to be CPU bound.

The section "Estimating Performance in Memory Databases" on page 31 describes methodologies for estimating performance, temporary storage requirements and real memory requirements of KSQL analytical statements, e.g. **select** statements. The example is the following aggregation on two table columns of a table with 1 million rows.

*select sum amount by stock.industry, date.month from trade*

This statement evaluates in 1 second. The storage size of the 1 million row **trade** table is 20 mb. Both the temporary storage requirement and real memory requirement are 16 mb per million rows. As the size of the table is increased incrementally, the execution times of the simple component statements increase linearly. The execution times of the statement itself start out linear, but are eventually overwhelmed by memory swapping when the available real memory is exhausted by the **by** phrase evaluation. This example illustrates the general rule: for optimal performance, the real memory requirement is that no table column used in computations should exceed 25% of RAM. (RAM is a relatively cheap route to a trouble-free, high performance database server. Load up!)

An individual Kdb server is single-threaded (see "Multi-Threading" on page 43 for a discussion of this design choice). Single-threaded means that user requests are handled serially, which makes it easy to estimate through-put. Specifically, execution costs are additive, i.e. the time for a server to execute a suite of statements is the sum of the times for the individual statements. It is typical that most of the time is spent on repeated execution of relatively few different statements. In this case the approximate average execution time for a request is simply a weighted average of the execution times for those statements by their relative occurrence frequencies. Various through-put measures can be calculated from this average.

## *Shuffle Architecture*

The shuffle architecture is one of the two choices available when a database exceeds the limits of a memory database. In this framework, selected tables are not loaded into memory when the database is activated. Instead, their columns are "shuffled" in and out of memory, as needed. Columns that are not needed at a particular time are not necessarily in memory. The shuffle architecture depends on the fact that only a few columns of any table are ever needed at exactly the same time. Tables that are not shuffled are loaded into virtual memory. Consequently we speak of *shuffle tables* and *memory tables*.

Both shuffle and memory tables are arranged in column order. The only difference is that the columns of shuffle tables must be stored separately on disk. Tables whose columns are stored separately on disk are said to be *splayed*. Shuffle tables are always splayed tables. However, tables in memory databases can be splayed on disk or not.

Tables with many columns are the best candidates for shuffling because the bulk of these tables will always be out of memory. In the case where only a few columns account for most of a table's references in queries, it may be possible to isolate those columns into a separate memory table and leave the remainder as a shuffle table.

The methodology for estimating performance in memory databases ("Estimating Performance in Memory Databases" on page 31) can be used for shuffle databases, but with an additional consideration. The items of a varchar column are not simply mapped into memory; they are also pre-processed to optimize searching and storage. Pre-processing occurs only once for a memory database, when it is first activated, and consequently, pre-processing effects do not appear in performance estimates. However, pre-processing a column of a shuffle table happens every time the column is mapped. Pre-processing can be avoided by using *enumeration*, which makes varchar columns behave like integer columns; see "Enumeration" on page 16.

Column items are stored contiguously on disk and laid out contiguously in real memory when accessed. This organization gives optimal disk access performance. (Disk access of an integer column on a high performance server can be on the order of 100-200 milliseconds.) However, the operating system caches accessed shuffle columns in real memory.

Consequently, if the memory hit-rate (so-called locality) is high, the performance is real memory performance.

The methodology for temporary storage requirements in memory databases (“Estimating Temporary Storage Requirements” on page 34) also applies to shuffle databases, but with one additional point. The storage requirements for shuffled columns must be added to the memory database estimates whenever a shuffled column is referenced.

Real memory requirements for optimal performance (“Estimating Real Memory Requirements” on page 35) are the same as for memory databases. In particular, there is the same size constraint on table columns, shuffled or not: the real memory requirement is that no table column used in computations should exceed 25% of RAM.

## *Parallel Architecture*

The parallel architecture is both an alternative to the shuffle architecture and the choice for databases that exceed the limits of a shuffle database. There are no size limits to parallel databases other than disk space. A parallel database is a family of memory or shuffle databases. That is, a source database that is physically partitioned so that each segment satisfies the virtual and real memory requirements. In principle, there can be as many segments as necessary. The segments can all run on one host computer or be distributed among several.

### *Distributed Queries*

SQL and KSQL statements can be executed in a distributed manner. Consider a detail table **A** that is partitioned into two detail tables **B** and **C**, so that every row of **A** is either in **B** or in **C**, but not in both. For example, consider the **select** statement

*select price, quantity from T where quantity>1000*

Apply this statement to **A** (in place of **T**) and let **R** be the result table. Then apply it, separately, to both **B** and **C** and let **S** be the union of those two result tables. **R** and **S** are identical SQL tables. However, as KSQL tables, their row orders may differ, depending on how **B** and **C** were extracted from **A**.

Aggregations require an additional step. For example,

*select sum quantity by stock from T where quantity>1000*

Create **R** and **S** as before. If a stock appears only in **B** or only in **C** then the rows for that stock in **R** and **S** are identical. Otherwise, there are two rows in **S** whose quantity values sum to the corresponding value in **R**. Apply the aggregation statement (without the **where** phrase) a third time, to **S**, to produce a result equivalent to **R**.

As of this writing, Kdb support for parallel databases requires the distributed processing language K. A dedicated Kdb parallel module is in the planning stage. It is not, however, difficult for programmers to deal with parallel databases and distributed queries, as we will see in “Gateway Servers” on page 30. The example in that section supports the performance discussion that follows.

## *Performance of Parallel Databases*

Each partition in a parallel database has its own Kdb server. All servers are running when the database is active and each one must have its own page file. That is, there must be as many page files of sufficient size as active servers. Clients can be organized so that a request is sent to a particular server when the database segment in that server is known to have all the required data, or to a *gateway server* that receives all Kdb requests, distributes them to the parallel servers and consolidates results. Each single server request is a normal database request, which has already been discussed. The focus here is on distributed requests.

In a distributed request, the same statement goes to every server. Every database segment is a memory database and therefore the MRPS (million rows per second) estimate for this statement (see “Estimating Performance in Memory Databases” on page 31) is meaningful for each segment. Using that estimate, we know how many seconds are required to evaluate the statement in every server. The sum of those numbers is the total evaluation time (although the elapsed time is the maximum value if all servers use different CPUs). Note that the sum equals the evaluation time when the entire reference table fit in one memory database.

Additional elapsed time goes to operating system costs to manage multiple servers, network costs to distribute requests and send back results, and dispatcher time to consolidate results. Modern operating systems are very good at running multiple processes and modern disk technology deals efficiently with large page files. There are no network costs if all servers, including the gateway server, run on the same host computer. The execution time for table unions can be estimated like any other KSQL statement, but is negligible for commonly-sized results. If the distributed query is an aggregation then an additional aggregation must be done in the gateway server. However, the execution time for this additional aggregation is negligible because it is applied to a union of aggregation results, which is relatively small.

In the example in “Gateway Servers” on page 30, the total elapsed time of the distributed query was 5% more than the evaluation time. That is, the elapsed time for the distributed query, which includes all overhead as well as the total evaluation times on the individual memory databases, is 5% more than the evaluation time when the same query is run on a single memory database.

Parallel databases can take advantage of multiple CPUs on the host computer. For example, if there are two CPUs and the load of a distributed query is equally balanced over the two, then statement evaluation time, viewed as elapsed time, is halved. In other words, through-put is doubled.

To summarize, when a parallel database runs on one host computer, the measured evaluation time for distributed requests is approximately the same as a single request in a single memory database. The measured elapsed time is approximately 5% more. If there are multiple CPUs and processing is well-balanced, the apparent evaluation time (elapsed time) is reduced by a factor nearly equal to the number of CPUs.

## Transaction Processing

Consistent and fail-safe ways to update databases is not a topic for this manual. Instead, we must concentrate here on the tools for implementing update strategies. There is a very useful paper on the kx website, *High Volume Transaction Processing*, that is relevant to transaction processing in Kdb databases. The paper predates the Kdb product. It is written in terms of K, the implementation language of Kdb. However, most of the paper is about databases and the reader who is familiar with this chapter will recognize the different Kdb architectures. The paper also contains performance numbers based on standard benchmarks.

There are two types of transactions in Kdb, individual and bulk. An example of an individual transaction is a bank account deposit and withdrawal. Bulk transactions are inserts or updates of large number of rows, all at once. For example, databases that support analyses of telecommunication or security trading activities are generally updated at regular intervals with bulk transactions instead of continuously with individual transactions. In Kdb a bulk transaction is a single message to the database server, not a sequence of individual update statements.

The bank account benchmark application provides an interesting example of how individual transactions can be processed collectively as bulk transactions to improve performance (see http:www.kx.com/a/kdb/examples/tpcb.t). In this example there are 100 bank branches, 10,000 tellers and 1,000,000 accounts. A transaction increments or decrements an amount from the specified account, from the teller's total and the branch's total. This author timed 50,000 transactions on the database server process, in various arrangements. If the transactions arrive one a time, the server evaluates about 2000/sec. However, if they arrive in batches of 5 each, the server evaluates about 6000/sec., and if there are 10 in each batch, about 7200/sec. One batch of 50,000 evaluates in about 200 milliseconds. See "The tpcb Example" on page 40 for the way these timings are done.

Knowing the arrival rate of individual transactions at the server tells us how they should be grouped in batches. Grouping transactions at each branch can cause unacceptable delays. However, a gateway server that receives all transactions can send them in batches to the database server and distribute the results.

The example bank account database is a small memory database with plenty of room for a transaction log (see "The transaction logging flag -l" on page 25). Realistically, grouping transactions is most likely not necessary, even if ATM machines are included. For larger databases, bulk updates go well with the shuffle architecture because every column of a shuffle table, which is shuffled in and out of memory, must be saved to disk immediately after it is modified. It is best that a column contains a significant number of changes every time it is saved to disk. Very large databases that can be partitioned for updates – where the data required for any transaction is entirely in one partition – fit the parallel architecture and require very little customizing. See the paper referenced at the beginning of this section for a general approach in other situations.

## *The K Programming Language*

The implementation language of Kdb is K, which is also a product of Kx Systems. K is a high-performance, high-level array language. K is implemented in C. With regard to arithmetic and relational primitives, K is like a C macro language for arrays. K also has highly-optimized advanced primitive functions, such as binary search. Additionally, like Java, K memory management is transparent.

Kdb databases can be enhanced with functions defined in K or C. K functions are defined in K and Kdb scripts and loaded into the server. C functions are linked into the server via K and Kdb scripts and behave like K functions. Database enhancements, i.e. custom analytics and transaction definitions (so-called *stored procedures*), require no K knowledge by C programmers.   However, C programmers do need the interface between C and K; see the chapter “The C - K Interface”.

Kdb clients and gateway servers can be written Java, C and K; clients can also be written in Visual Basic.  A bit of K knowledge is required of  C programmers here; see “K Language Essentials” on page 35.

# 3 Creating and Managing Kdb Databases

## Database Organization on Disk

We have already dealt with the performance characteristics of Kdb databases for accessing and manipulating data (see chapter “Design and Performance”). The question remains as to how the saved database should be arranged on disk for the best performance when writing modified data to disk.

There are two ways to arrange a Kdb database on disk, either as a single database file (extension .kdb) or in *splayed* format, i.e. as a directory whose entries are tables (the directory is the database). When a database is splayed into individual tables, each table can also be arranged in one of two ways, either as a single file (extension .l) or splayed as a directory whose entries are files containing individual columns (extension .l).

A memory database can be arranged in any way that conforms to these rules. How it is done is largely a matter of convenience when building the database and whether or not default logging will be used, which requires a single file arrangement (see “The transaction logging flag -l” on page 25). However, if performance is in issue when saving a particular table, then that table should be splayed on disk when the its size exceeds half the available RAM.

A shuffle database must be splayed because every shuffle table must be splayed. Every splayed table is a shuffle table.

## Creating Databases

A Kdb database can be initialized in the conventional way using SQL **create** statements in a SQL script (extension .s) or KSQL table initialization statements (see “Empty Lists and Empty Tables” in the KSQL Reference Manual). Small tables can also be populated at the same time they are created using KSQL table initialization.

A database can also be created by starting a server with a KSQL script in which **load** statements bring in tables from files of any type that appear in “The data source name f” on page 24. The **load** statements can also be entered by hand in the Kdb console after the server is started.

Once the database is create it must be saved. It can be saved in one file as follows.

```
save 'c:/dbs/test'
```

The saved database is **c:/dbs/test.kdb**. Each table must be saved separately to create a splayed database, in which case each table can be splayed or not. For example, if the customer table is to be saved as one file, use **save** dyadically, as follows.

```
'c:/dbs/stest' save 'customer'
```

The saved table is **c:/dbs/stest/customer.i**. Use **rsave** to save the order table as a splayed table.

```
'c:/dbs/stest' rsave 'order'
```

The saved table is the directory **c:/dbs/stest/order**; the columns are files in that directory.

## *Enumeration*

Every time a **varchar** column is loaded into memory it is pre-processed to optimize storage and searching. This cost can be avoided for a column that is loaded frequently by *enumerating* the column. An enumerated **varchar** column is an integer column whose items are indices into a *reference* **varchar** column. Typically, the reference column is a primary key column and the enumerated column is a foreign key.

Enumerated columns are like **date** columns. For example,

```
trans:([n:('cart', 'bike', 'roadster')] w:(4, 2, 4))
o:([] n:('cart', 'roadster', 'bike', 'cart', 'cart', 'bike'),
   d:('10/5/1998', '4/7/1997', '11/14/1998', '1/5/1999', '8/15/1998', '3/10/1997'))
```

Even though column **d** is intended to be a **date** column, it is simply a **varchar** column whose items look like dates. To make **d** a column of dates, the relationship to dates must be made explicit.

```
o:([] n:('cart', 'roadster', 'bike', 'cart', 'cart', 'bike'),
   d:date('10/5/1998', '4/7/1997', '11/14/1998', '1/5/1999', '8/15/1998', '3/10/1997'))
```

The entries of column **d** are now integers, the Kdb internal representation of dates. Analogously, to enumerate column **o** on the primary key of **trans**, the relationship must be made explicit.

```
o:([] n:trans('cart', 'roadster', 'bike', 'cart', 'cart', 'bike'),
   d:date('10/5/1998', '4/7/1997', '11/14/1998', '1/5/1999', '8/15/1998', '3/10/1997'))
```

The entries of column **n** are now indices into column **n** of table **trans**. Column **n** is now an enumerated column.

A text representation of dates can be used in **where** phrases, and this is also true of enumerated columns. For example, the following statement is valid.

```
select from o where n ='cart'
```

Also like dates, enumerations must be explicit in inserts.

```
'o'  insert (trans('bike'), date('04/02/1999'))
```

## *Managing User Access*

There are two reserved tables in a Kdb database, **access** and **user**. The **access** table holds information on which tables can be seen or modified by which users. The **user** table holds user names and passwords. If a **user** table is defined then every connection to the database requires a user name and password. Consequently, if access to the database must be limited, the prudent thing to do is include the database administrators' names and passwords in the **user** table's initialization when the database is created. The administrators' can then give access to others.

### *The* **user** *Table*

The user table is managed like any other table. It has columns named **user** and **password** and can be initialized as follows.

```
user: ([user:()] password:())
```

Both fields are varchar fields. An administrator (or any user) can be added.

```
'user' insert ('admin', 'pw310')
```

Alternatively, administrators (or users) can be included when the table is defined.

```
user: ([user: enlist 'admin'] password: enlist 'pw310')
```

## *The* **access** *Table*

The **access** table has columns **access**, **var** and **user**. The **access** column holds the type of access allowed to the data object named in **var** for the user named in **user**. Note that data objects are not just base tables, but include views and any other data in the database. The **select** entries for a base table are inherited by any view for which that base table is the reference table. On the other hand, users can have **select** access to a view without having **select** access to its reference table.

The access table is initialized as follows.

*access:([access: (), var: (), user: ()])*

All three fields are varchar fields. Every entry of the **access** column is one of 'delete', 'insert', 'select', and 'update'. Like the **user** table, the **access** table can also be managed like any other table. In addition, permissions can be specified with the SQL **grant** statement. For example,

*grant all on all to admin*

gives this administrator complete access to all data in the database. Note that “$” must be appended to the front of this statement if it appears in a KSQL script. Other examples are

*grant select on all to all*

lets everyone access and modify all tables, including **user** and **access**,

*grant all on t to all*

lets everyone access and modify the table **t**,

*grant insert on all to tom*

lets Tom insert data in any table, and

*grant update, delete on t, u, v to mary, andrew*

lets Mary and Andrew update and delete tables **t**, **u** and **v**.

Kdb does not support the *with grant option* on a **grant** statement.

# *Customizing Kdb Databases*

## *Column Attributes*

Every table column has a set of attributes that are defined in the column's attribute dictionary (see "Attributes" on page 37). Attributes are defined using a syntax that is carried over from K. The column width attribute, for example, is denoted by **W** and is defined for column name of table customer as follows.

*customer.name..W:4*

Any customer name with more than 4 characters will now display as "****". This attribute does not have a default value because Kdb estimates the width from the data. Attributes with default values do not necessarily appear in attribute dictionaries.

The **T** attribute specifies the column type. This may be the data type, i.e. *'int'*, *'float'*, *'varchar'* or *'varbinary'*. The column type can also be any **date**, **time** or **timestamp** type, such as *'date'*, *'day'*, *'month'*, *'time'* and *'timestamp'*. It can also be the name of the table by which the column is enumerated. For example, in "Enumeration" on page 16, column **o.n** has type *'trans'*. Finally, the type of a foreign key is the (enlisted) name of the table holding the primary key. For example, if the primary key table is customer then the type attribute of the foreign key column is **enlist 'customer'**.

The value of the **T** attribute need not be set for tables defined by KSQL or SQL. However, it may be necessary to do so for imported tables. Note that setting the attribute does not automatically force conversion of the data to that type.

The **P** attribute specifies the number of decimal digits to be displayed in floating-point columns. Its default value is 2.

The **N** attribute indicates whether or not the column can contain null values (1 for yes, 0 for no). The default value is 1.

The **D** attribute defines the default value to be used when a column item is needed but not specified. The default is the null value for that column.

The **K** attribute indicates whether or not a column is a primary key (1 if yes). The value need not be set for tables defined by KSQL or SQL. However, it may be necessary to do so for imported tables. Note that all primary key columns must appear first in the default column order, i.e. the order of a **select** statement where no columns are specified.

The **S** attribute indicates whether or not a column is sorted in ascending order (1 if sorted). This value is used by Kdb to determine whether or not optimized searches can be done. The value must be explicitly set. It is up to the user to ensure that the column actually is sorted.

When a splayed is saved the attributes are saved in the table directory as the file named ".l". That is, the name consists of only the .l extension.

## *Custom Analytics and Stored Procedures*

Functional enhancements to Kdb servers are generally in one of two areas, either custom analytics or stored procedures. Custom analytics typically take 1-dimensional array arguments – i.e., table columns – and return conformable array results. They can be embedded directly in KSQL statements as if they are KSQL primitives. For example, a data-smoothing function that applies to two columns, **price** and **date**, can be used as follows.

*select price, date, smooth[price,date] from trade where amount>1000*

Custom analytics can be written in K, the implementation language of Kdb, and C. In particular, C mathematical library routines can be linked into Kdb as custom analytics; see "Linking to C Functions in K" on page 50.

Stored procedures usually define transactions, i.e. collections of table modifications that must be completed without intervening modifications from other sources, thereby maintaining database consistency. Stored procedures can also be written in K or C.

Kdb provides the entry point **.d.r** for evaluating KSQL and SQL statements within stored procedures. For example,

*.d.r"update qty:2*qty from 'OrderItems' where orderid=6099"*

is a K expression that updates the **OrderItems** table. This expression can be evaluated in a C function using the API function ksk; see See "Example: Evaluating KSQL Statements" on page 55 for the way to evaluate KSQL statements within C functions.

Custom analytics are embedded in KSQL statements, where they are called with KSQL syntax, as in

*smooth[price,date]*

Stored procedure evaluations usually make up the entire statement in which they appear. As a result, they can be called in a KSQL statement, as in

*update_save['t', ('abc',27, 3.24)]*

or in a KDBC remote procedure call statement; see “KDBC” on page 28.

Stored procedures are generally thought of in regard to transaction processing. They are also provide an effective way to manage Kdb gateway servers. See “Gateway Servers” on page 30.

# 4 *Starting and Managing Kdb Servers*

The simplest way to start Kdb from a console is with the command

```
k db
```

The console then becomes a Kdb console, as indicated by the prompt

```
t>
```

Databases and individual tables can be loaded (see the **load** command in the KSQL Reference manual). SQL and KSQL statements can be evaluated (enter a statement at the prompt and press **Return**). The active tables can be displayed in the Kdb Viewer (see the **show** command) and, from the Viewer, the Kdb process can become a web server or a TCP/IP server. The Kdb console is closed with double back-slash.

```
t>\\
```

Loading data and making the Kdb process a database server can also be done from the command line, and more. No matter what start command options are specified, the console still becomes a Kdb console where statements can be entered and evaluated.

## *The Kdb Startup Command*

The general form of the Kdb command that starts a fully functional server is as follows.

```
k db [-dsn] f [-P[n]] [-p[n]] [-l] [-r[h]p]
```

Square brackets mean that the item within need not be present; the brackets themselves do not appear in actual commands. Every command must start with "`k db`". The meanings of the items to the right of this prefix are listed below.

### *The ODBC flag* `-dsn`

If this flag is present, then the name immediately to its right is a file DSN or user DSN, which designates the data target of an OBDC connection. See the Microsoft NT documentation for details. After that, see www.kx.com/a/kdb/connect/odbc.txt for specific Kdb information.

## *The data source name* `f`

If `-dsn` is not present then `f` holds the name of one of the following:

- A Kdb database file (file extension `.kdb`)
- A directory whose entries make up a Kdb database (see “Database Organization on Disk” on page 15).
- A KSQL script (file extension `.t`). A KSQL script initializes the database. It can contain both KSQL and SQL statements, but the two statement types must be distinguished. This is done by putting “$” in front of SQL statements; KSQL statements are the default in KSQL scripts.
- An SQL script (file extension `.s`). An SQL script serves the same purpose as a KSQL script. The only difference is that KSQL and SQL statements are distinguished by putting “$” in front of KSQL statements; SQL statements are the default in SQL scripts.
- A Microsoft Access database file (extension `.mdb`). Each Access table becomes a Kdb table.
- A Microsoft SQL Server database (extension `.mdf`). The name of the database is given, not the database file (they are usually the same, but not necessarily). Even so, the file extension `.mdf` for a SQL Server database file must be used. For example, if `pr.dbf` is a SQL Server database file whose contents are the database named `prices`, then the parameter `f` to the Kdb start command must be `prices.mbf`. Each database table becomes a Kdb table.
- A dBase file (extension `.dbf`). These files result in one-table databases.
- An Excel file (extension `.xls`). Each worksheet in an Excel file contributes a table to the database.
- A comma-separated-value file (extension `.csv`) or a text file with tab-delimited fields (extension `.txt`). These files generate databases with only one table.

## *The Web server port* `-P[n]`

Specify this option to create a web server. Use `-P` to create a web server having the default port number or `-P 2080` to create a web server listening on port 2080. Any authorized port number can be used. The default port number is 80. A Kdb server can be both a web server and a TCP/IP server.

## *The TCP/IP port* `-p[n]`

Specify this option to create a TCP/IP server. Use `-p` to create a TCP/IP server having the default port number or `-p 2001` to create a TCP/IP server listening on port 2001. Any authorized port number can be used. The default port number is 2001. A Kdb server can be both a TCP/IP server and a web server.

## *The transaction logging flag* `-l`

If this flag is included then a transaction log is maintained and saved to disk after every transaction, i.e. every request that changes the data. Log items are triples containing the timestamp of when the request was received, the user who sent the request and the actual message. The log is replayed by doing the following for each log item, in the order in which the original transactions were received: set **current_time** to the timestamp in the log item, set **current_user** to the user in the log item and execute the transaction. Consequently, any reference to **current_time** and **current_user** will be the same as when the transaction was originally evaluated.

This default logging applies only to one-file memory databases (file extension `.kdb`). The log file is part of the database, but only the log file is saved to disk as part of transaction processing.

The database is saved to disk (and the log is reset to the empty log) whenever the following Kdb **save** statement is executed.

*save''*

Since the log is part of the database it is important to execute this statement on a regular schedule, depending on the transaction rate, so that the log does not get too large. See "Managing Transaction Logs" on page 30 for a way to do this automatically.

### *Rollbacks*

Rollbacks occur because the database is in an inconsistent state. That is, a stored procedure is executing a transaction when, after some modifications have been made, an error occurs. If any modifications have not been done at this point then those that have been completed must be undone. To guarantee a consistent state the transaction log is applied to saved database, thereby returning the database to the state before this stored procedure was executed. This can be very time consuming when the transaction log is large. Rollbacks should be avoided wherever possible, and the way to do that is with careful checking in stored procedures to anticipate potential errors. If a stored procedure cannot proceed with the expected updates then the recommended exit is with a 'rollback' error message, indicating to the sender that the updates have failed.

## *Replication server location* `-r[h]p`

Use this flag to designate the host and port of a replication server, which maintains a copy of the active database. Every message sent to the main server (the one whose start-up command contains the `-r` flag) that causes the database to be modified is sent asynchronously to the replications server. As a consequence, updates on the replication server are done after the main server.

If the host computer is not specified then it is the one on which the main server is running. (Running a replication server on the same host can be useful when the host has two CPUs). The main server and the replication server must be started with identical databases. If both run on the same computer then both can be started with the same database file because their in-memory copies will be different and the replication server never saves the database.

The replication server should be started first, as a TCP/IP server. The main server is then started with a `-r` option value identical to the `-p` option value in the replication server startup command.

## *Starting a Shuffle Database*

According to “Database Organization on Disk” on page 15, there is nothing in the way a splayed database is organized on disk to distinguish it as a memory or shuffle database. The distinction is made when the Kdb server is initialized. If the database is splayed then the `-s` indicates that the splayed tables in the database are to be shuffled. Otherwise, the database is treated like a memory database.

# *Connecting to a Kdb Server*

This section describes the ways clients can connect to Kdb database servers. Kdb supports both JDBC and ODBC, the well-known, standard interfaces. There is also an interface that is specific to Kdb called KDBC. Both KSQL and SQL statements can be sent to Kdb servers by all three interfaces. The only difference is that SQL statements are the default for JDBC and ODBC clients, while KSQL is the default for KDBC clients. Statements of the non-default kind must start with “$” at the front.

Java and Visual Basic are good implementation languages for Kdb clients because they and K, the implementation language of Kdb, share the same basic self-describing data types; see Table 4.1, below.

TABLE 4.1 **Data Types**

| **K[a]/Kdb[b]** | **Java** | **Visual Basic/Excel** |
|---|---|---|
| 1 (int) | Integer | Long |
| 2 (float) | Double | Double |
| 3 (byte) | Byte | Byte |
| 4 (symbol/varchar) | String | String |
| 6 (null) | null | Null |
| 0 general list | Object[] | Variant() |
| -1 int list/column | int[] | Long() |
| -2 float list/column | double[] | Double() |
| -3 (byte list/varbinary) | byte[] | Byte() |
| -4 varchar list/column | String[] | String() |

a. See “Data Types” on page 46 for more information on the K data types.

b. The Kdb Date type corresponds to K (julian) int, while Time and Timestamp correspond to float.

KDBC is simpler than JDBC and ODBC because it has just one entry point, is richer in the kinds of messages that can be sent and is faster because large amounts of data can be sent in a single message. Both JDBC and KDBC use KDBC in their implementations. The KDBC interface is available to Java and C clients as an alternative to JDBC and ODBC.

## *JDBC*

The Kdb JDBC driver is a type-IV, pure Java, driver. The driver can be found on the Kx website, at www.kx.com/a/kdb/connect/jdbc. Download the file `jdbc.zip` and unzip its contents into the k subdirectory of CLASSPATH. The Kx website directory also contains examples.

## *ODBC*

The Kdb ODBC driver is a level I driver. See the Kx website for information on ODBC front ends, in particular the files at www.kx.com/a/kdb/connect/odbc/ and the tutorial at www.kx.com/technical/tutorials/excel/.

## *KDBC*

KDBC consists of entry points to open and close connections to Kdb servers and a single entry point for sending messages. Every KDBC message is in the form

*("function",arg1,arg2,...)*

where there are as many *argn*'s as there are function arguments. The function is a stored procedure defined on the server. Consequently KDBC messages are *remote procedure calls*. Note that a KSQL or SQL statement can be used in place of *"function"* because such a statement is executable; and therefore can be considered a function with no arguments. For example,

*("select sum qty by p from sp")*

The parentheses can be dropped in this case.

The file www.kx.com/a/kdb/connect/kdbc.txt contains both a Java and C version of a client accessing a Kdb database with ODBC. The C version can also be found in “Example: Accessing a Kdb Server with KDBC” on page 54. In the Java version a Java class named **c** is created with a constructor that takes the host and port of the Kdb server as arguments. KSQL statements can then be sent to the server, for example

```
Object[]r = (Object[])c.k("select sum qty by p from sp");
```

The result **r** contains three lists of column names, column data, and column types. The connection to the server is closed with `c.close()`. SQL statements can also be sent, but since this is a KSQL interface, they must begin with “$”.

The C version is organizationally the same, except that K functions are called to connect to the server and close the connection, and a K function is defined for executing KSQL expressions on the server and receiving the results. See “K Language Essentials” on page 35 and chapter “The C - K Interface”.

## *Remote Procedure Calls*

The Java method k used to evaluate the KSQL expression above is one of several that together implement a Java remote procedure call (RPC, for short). For example, if `foo` is a stored procedure on the Kdb server that takes to arguments, and if those arguments are created in the Java client as **A** and **B**, then **foo** can be executed as follows.

```
(Object[])c.k("foo",A,B);
```

See “Example: A Remote Procedure Call” on page 55 for making remote procedures calls from a C client.

### *Bulk Updates*

Individual updates to databases are done with KSQL and SQL **update** statements. Bulk updates are done with the KSQL **insert** function, which can be executed with the following KDBC remote procedure call.

*("insert","table",bulk_data)*

The insert function takes two arguments, the name of the table (*"table"*) and the data to be inserted (*bulk_data*). The bulk data is a rectangular arrangement of data with one item for each column in the named table. The items must be in the same order as the table columns, i.e. the order when the table was defined or, equivalently, the order in which they appear as a result of a KSQL *select from table* statement. If the RPC is sent from a K or C, the bulk data can be organized as a K dictionary whose entry names match the table column names, in which the column order of the data is not relevant.

### *Managing Transaction Logs*

If default logging is in effect on the Kdb server then the database is saved to disk (and the log is reset to the empty log) be executing the following Kdb **save** statement *save''* (“The transaction logging flag -l” on page 25). The corresponding remote procedure call is

*("save","")*

A dedicated client can execute this RPC as a scheduled task with frequency depending on the level of activity on the database.

## *Gateway Servers*

A gateway server is a intermediate process that receives client requests as if it is the database server and passes them along to an actual database server. There are many reasons for using a gateway server. For example, a gateway server can require that all requests are by way of remote procedure calls, so that all requests can be monitored and modified as needed. (Even ad hoc queries can be submitted as arguments to stored procedures). A gateway server can provide custom management of transaction logs or, as a Kdb server itself, provide default logging for a shuffle or parallel database. Finally, to keep the list short, a Kdb gateway server can employ a secure memory database for access and distribution control for other servers, using its own **access** and **user** tables for initial screening (see “Managing User Access” on page 17).

See “Inter-process Communication” on page 37 for sending requests to Kdb servers from a Kdb gateway server.

Kdb gateway servers provide customized control over Kdb database systems.

# 5 Kdb Topics

## Estimating Performance in Memory Databases

The way to estimate Kdb query performance is as follows. Every KSQL query statement can be broken down into a collection of *base* statements, each of which uses one and only one computation in the source statement. The performance of each base statement is estimated by timing its evaluation. These estimates are summed to produce a performance estimate for the statement. The example in this section is a **select** statement. but the methodology applies to other queries as well, in particular **update - by** statements (see the KSQL Reference manual).

An obvious question comes to mind: Why not simply time the source statements themselves? The answer is that base statements are *generic*. For example, if *quantity>100* appears in one KSQL statement and its base statement is timed, then that number will be the same for any other statement containing a relational function applied to an integer column. Consequently, timings for a comprehensive set of base statements can be collected in order to estimate the performance of any source statement of interest.

### An Example

The example is from the **trade** database defined by the Kdb script **trade.t** that comes with the Kdb download. Open that script in an editor and you will see (as of this writing) that the integer **n**, which defines the number of rows in the trade table, is set to 100,000. For convenience, that value has been changed to 1 million here, even though the new value may be too large for the evaluation version of Kdb. The following is the test statement.

*select sum amount by stock.industry, date.month from trade*

There are three computations in this statement, a two-dimensional aggregation (**industry**, **month**), a table join (**stock.industry**) and a field extraction from a date column (**date.month**). As a result, there are three base statements isolating these computations.

*select sum amount by stock, date from trade*

*select stock.industry from trade*

*select date.month from trade*

Note that the first base statement is also an aggregation on two columns, but the columns require no computations for their formation. The last two statements isolate the column computations in the **by** phrase of the test statement.

KSQL statements can be timed by putting a colon to the left of the statement and evaluating it. The CPU time (in milliseconds) used in the evaluation is then displayed below the statement. For example, time the first base statement as follows.

*:select sum amount by stock, date from trade*

The result is 640 milliseconds. (As in chapter “Design and Performance”, all timings were done on the author’s desktop computer. The numbers given here are the result of doing six independent evaluations, throwing out the smallest and largest and averaging the remaining four.) The results for the second and third base statements are 120 and 260 milliseconds, respectively.

The estimated CPU time required for the test statement is the sum of the CPU times of its base statements. In this case, it is 640+120+260 = 1020 milliseconds. (That the estimate is almost exactly 1 second is purely coincidental). (A quick check on this number is obtained by timing the test statement in the same way as the base statements, which gives 1010 milliseconds).

Note that execution time depends only on the computations in a statement, not on peripheral issues such as the number of columns in the reference table (i.e., the table named in the **from** phrase).

## *Base Statements are Generic*

These three base statements in the above example provide further support for the idea that base statements are generic. That is, 640 milliseconds applies to any other statement with an aggregation on two columns, 120 milliseconds to any table join (dot notation evaluation) based on a varchar column and 260 milliseconds to any date field extraction.

## *Other Base Statements*

Timings of other base statements using the 1 million row **trade** table are as follows: 110 milliseconds for aggregations on one column, 50 milliseconds for arithmetic functions, relational functions, logical functions and string (varchar) match, and 60 milliseconds for a **select - where** selection in which relatively few rows are chosen and 50 milliseconds +

110 milliseconds per column (220 milliseconds for floating-point columns) where most rows are chosen.

## *Search Phrases*

Searches require special attention. KSQL searches are table selections defined by **where** phrases that are simple **equals** expressions, as in

*select from trade where stock='aaa'*

and table indexing expressions, as in

*stock['aaa'].industry*

Search evaluations are optimized when the reference table is sorted in ascending order of the search column, as in

*'stock' asc 'trade'*

Once this sorting is done, the search takes only one or two milliseconds, no matter how many rows there are in the reference table.

## **where** *Phrases*

A **where** phrase that uses the punctuation symbol “,” in place of **and** is executed differently than the phrase with **and**. For example, if the **where** phrase is

*(stock='aaa')and date>date'06/01/98'*

then the search expression *stock='aaa'* takes 50 milliseconds, the relational expression takes 50 milliseconds and the logical **and** expression takes another 50. Add to those the 60 milliseconds for a selection in which only a few rows are chosen, and the estimate is 220 milliseconds per million rows for a selection based on this **where** phrase.

Note that this number does not change when the **trade** table is sorted by **stock** because the same evaluations must be done.

Now consider the **where** phrase

*stock='aaa', date>date'06/01/98'*

which is a cascading **where** phrase. First the selection based on *stock='aaa'* is executed, and then, on that result, the selection based on *date>date’06/01/98'* is done. If the **trade** table is sorted by **stock** then the (optimized) selection based on *stock='aaa'* takes only 1 or 2 milliseconds. The result of this selection has about 150,000 rows, to which the selection based on *date>date'06/01/98'* is applied. The computational effort is somewhat less than selecting relatively few rows from a million, giving a total cost estimate of 60 milliseconds.

## *The Performance Unit MRPS*

A performance measurement unit that combines execution time with what is actually computed is million-rows-per-second, or MRPS. The rows refer to the reference table of the KSQL statement.

CPU milliseconds can be converted to MRPS by the KSQL expression

*N/(1000*MS)*

where N is the number of rows in the reference table and MS is milliseconds. In the above example, N is 1000000 and the MRPS values for the three base expressions are (approximately) 1.6, 8.3 and 3.8. The estimated MRPS value for the test statement is then given by

*1/sum 1/(1.6, 8.3, 3.8)*

which is approximately 0.99 (a purely coincidental closeness to 1). The result is an estimated 1 million rows per second for the test statement.

## *SQL Performance*

Kdb ANSI-SQL is translated to KSQL for evaluation and consequently has comparable performance.

## *Estimating Temporary Storage Requirements*

The methodology for estimating performance can be adapted to temporary storage requirements, that is, the virtual memory needed to create and hold temporary results that exist during statement execution. Since virtual memory is not unlimited and memory databases also occupy virtual memory, temporary space requirements could be significant for tables with very long columns.

In the base statements of the above example, both *stock.industry* and *date.month* are temporary integer arrays with 1 million items each. Both are temporary results that must be available for the **by** phrase. In addition, the **by** phrase requires a temporary integer array for its result and, presumably, another for internal computational use. This means that four integer arrays with 1 million items each are required during the **by** phrase evaluation. When the **by** phrase evaluation is complete only the result must be maintained, leaving one temporary integer array of 1 million items. The aggregation result, in the very worst case, occupies another integer array with 1 million items. Consequently, at most four temporary integer arrays are required at any one time, which is 16 megabytes of storage per million rows.

A **where** phrase in a **select** statement may reduce temporary storage requirements. For example, consider the cascading **where** phrase from above.

*stock='aaa', date>date'06/01/98'*

Executing *stock='aaa'* requires at most a 1 million row integer array for its result. However, all other temporary storage requirements are now reduced by more than 80% in this example. The total temporary storage requirement is now less than 8 megabytes per million rows.

### *Estimating Real Memory Requirements*

Having enough real memory to avoid page swapping during statement execution is a significant performance issue. In order to run at optimal speed, every computation must fit in real memory. In particular, the arguments and result of every function execution must all fit in memory. The real memory requirement to evaluate a statement is the same as the temporary virtual memory requirement; the temporary virtual memory simply overlays the real memory.

# *K Language Essentials*

## *Data Types*

K has the same basic data types as Kdb, integer, floating-point, symbol (varchar) and character (varbinary). K vectors do not require surrounding parentheses and comma-separators between items; spaces are used for separating numeric items. For example,

```
x:1 3 43 5
```

defines an integer vector with four items,

```
y:1 3.4 5.5
```

defines a floating-point vector with three items, and

```
z:`abc `xz.s34 `"s*4/5"
```

defines a symbol vector with three items (the spaces between items are for readability only). Symbols in K are denoted differently than KSQL varchar items. A symbol is written with a leading backquote followed by its contents which, in general, must be surrounded by double-quotes. For example, `` `"s*4/5" ``. The double-quotes are not part of the data. Symbols that represent valid K names do not need double-quotes, as in `` `xz.s34 `` and `` `abc ``.

Finally, a character vector is written the same as a KSQL varbinary, as in

```
w:"1224 w 34th, New York"
```

The *items* of vectors are called *atoms*, which are valid data objects. Vectors are special forms of lists whose items are all atoms of the same data type. In general, the items of lists can be atoms of mixed types or other lists or any type of K data object, including functions. The items of a general list are separated by semi-colons in K, as in

```
(1 2 3; (`a`b;4.127)
```

which is a two-item list whose first item is the integer vector `1 2 3` and whose second item is also a two-item list, consisting of the symbol vector `` `a`b `` and the floating-point atom `4.127`.

Note that the results of expressions executed in a Kdb session are displayed in K data format, not Kdb format.

## *The K Tree*

All data in an active K environment is organized hierarchically in a tree. Every node of the tree is a data object called a *dictionary.* Because of similarities with file systems, dictionaries are also called directories. A dictionary that plays a special role is often called a *context*. For example, any dictionary can be made the *active context*, which means that references to data within that node can be made relative to it and do not need full path specifications.

The top level dictionary is denoted simply by dot (`.`). There are several dictionaries within `dot` that appear by default when a K or Kdb session is started. For example, the `.k` dictionary is the default active context. In a Kdb session, this dictionary holds the active database. Kdb tables are K dictionaries. A Kdb table called **trade** can be referenced in K by `.k.trade` or `.k[`trade]`.

### *Attributes*

Every data object on the K tree has an associated dictionary called its *attribute dictionary.* Some attributes have default meanings; all others can be application specific. For example, the attribute named `t` is the default *trigger* attribute. If defined, the value is a character vector holding a K expression that is automatically executed whenever the value of the owner changes. For example, the attribute dictionary of a K object `abc` is denoted by `abc.` (i.e., `abc-dot` with no spaces) while reference to a specific attribute requires two dots, as in `abc..t` for the trigger on `abc`. The object `abc` is the *owner* of its attribute dictionary. See “Column Attributes” on page 19 for the default attributes of Kdb tables.

A dictionary is displayed as a series of triples, one for each item consisting of the item name as a symbol, its value and its attribute dictionary).

## *Inter-process Communication*

A client can send a synchronous or an asynchronous message to a gateway or Kdb server. Synchronous messages are called *get* messages and asynchronous messages are called *set* messages. The client sends a synchronous message when a result is expected and an asynchronous message otherwise. In effect, a synchronous message corresponds to executing a function on the server and waiting for the result. For example, a client should send a KSQL **select** statement synchronously because a result is expected. Asynchronous messages are appropriate when no response is expected.

### *Connecting*

The monadic primitive function denoted by `3:` is used to connect to a gateway or Kdb server. The argument to this function is a list `p` of connection parameters that is described in the chapter “Starting and Managing Kdb Servers”. The result of the function is called a *handle*, which is used whenever messages are sent to that server. For example,

*h :3: p*

defines the handle **h**. The connection is successful if no error is reported.

### *Sending and Receiving*

The dyadic primitive functions `3:` and `4:` send asynchronous and synchronous messages, respectively. The left argument of either one is the handle for the connection and the right argument is the message. The message can be any K object that is meaningful to the recipient. In the following example, the K object `r` holds the result of a **select** statement sent to a Kdb server.

```
r: h 4: "select avg price by stock from trade"
```

The result `r` is a triple containing a list of column names as symbols, a list of values and a list of column data types as symbols. A stored procedure `sp` on the server with arguments `A`, `B` and `C` can be evaluated as follows.

```
s: h 4: (‘sp;(A;B;C))
```

An asynchronous message can only be sent by K clients.

### *Closing a Connection*

If h is a handle created by monadic 3: (see “Connecting”, above), then

```
3: h
```

closes the connection.

## *File Management*

Any K object that does not contain function definitions can be saved to disk and loaded (i.e., mapped) from disk. The expression to save the object **o** at path location **p** is

```
p 1:o
```

The expression to map the K object on disk into the K workspace as the K object **o** is

```
o: 1: p
```

The path location variable **p** is a character vector, as in

```
p:"c:/k/data/item2"
```

Slashes can always be used in path specifications, even when back-slashes must be used outside K. In that case you can also use back-slashes in K, but you must always use double back-slashes, as in *"c:\\k\\data\\item2"*.

A K object **o** can be appended to an existing K object stored on disk with *5:*. For example, if the character vector **p** defines the path to a K list with **n** items is stored on disk, and if **o** is a K list with **m** items, then

*p 5: o*

appends **o** to the stored list to create a stored list of *n+m* items.

## *Functions*

A K function definition is a series of K statements surrounded by braces. Statements on the same line are separated by semi-colons. Statements on successive lines are separated by a new-line. The result of the function is – in the default case – the result of the last statement in the definition, i.e. the statement immediately followed by the right brace that marks the end of the definition.

This examples in this manual are one-statement functions, which in K are sufficient to define complex computations. The update function in “The tpcb Example”, below, which can be found in the referenced Kdb script on the kx website, has several lines. See the K Reference Manual on the kx website for the definitions of the K primitives used in that function.

The function *{+/x}* sums the items of its argument **x** and *{x+y}* sums its arguments **x** and **y**, item-by-item if either argument is a list. The name **x** is the default argument of a monadic function (i.e., one argument), while **x** and **y** are the default argument names of a dyadic function. Other names can also be used, as in *{[aa] +/aa}* and *{[al;ar] al+ar}*. Functions are given names with ordinary assignment, as in *agg:{+/x}* and *sum:{x+y}*. A monadic function can be evaluated by *agg[a]* for a numeric list **a**, or simple *agg a*. Evaluation of a dyadic function is of the form *sum[a;b]*.

## *K and Kdb*

K functions can be defined in K scripts (file extension `.k`) and loaded into Kdb by loading K scripts within Kdb scripts. They can also be defined in Kdb scripts by preceding K definitions with the back-slash *escape* character. The following example illustrates the use of back-slash for indicating K definitions in Kdb scripts.

## *The tpcb Example*

The script http:www.kx.com/a/kdb/examples/tpcb.t from the kx website is used in “Transaction Processing” on page 12 to illustrate the effect of grouping transactions in batches. Various timings are given there for various batch sizes. In this section we will see how those timings are done.

First of all, download the script tpcb.t. This script is loaded into a timing script (say time.t) with the Kdb load statement

```
load'tpcb.t'
```

(see the KSQL Reference manual). The update function defined in the script is the K function named **up**. It takes four integer vector arguments called account id, teller id, branch id and the amount by which the corresponding quantities are increased (positive amount) or decreased (negative amount). Fifty thousand random transactions can be defined by the following KSQL statements:

```
ai:50000 rand account.id
ti:50000 rand teller.id
bi:50000 rand branch.id
x:50000 rand 100.0
```

Then

```
:up[ai, ti, bi, x]
```

gives the CPU time for fifty thousand transactions done in one batch.

We can simulate five thousand batches of ten transactions each as follows. The K **reshape** function is denoted by #. The K expression

```
\AI:5000 10#ai
```

reshapes the items in **ai** into a list **AI** with 5000 items, each of which is an integer vector with 10 items. The leading back-slash is an *escape character* that is required in a KSQL or SQL script to indicate that what follows is a K expression. Do the same thing for the other vectors.

```
\TI:5000 10#ti
\BI:5000 10#bi
```

*\X:5000 10#x*

Each set of corresponding items in the new lists, say *AI[i]*, *TI[i]*, *BI[i]* and *X[i]* for an index *i*, represents a batch of ten transactions and

*up[AI[i], TI[i], BI[i], X[i]]*

evaluates the *i*th batch of ten transactions. All five thousand batches can be evaluated (and timed) in a single KSQL statement as follows.

*:up Each[AI, TI, BI, X]*

That is, **Each** is an operator that applies **up** to every set of corresponding items in **AI**, **TI**, **BI** and **X**, that is, to every transaction batch. Use 10000 5 to reshape for five transactions per batch and 50000 1 for one transaction per batch.

## *Each Operators*

The K **each** operator is denoted by single-quote. The K expression corresponding to the above KSQL **Each** expression is

*\\t up'[AI; TI; BI; X]*

The leading back-slash indicates that this is a K expression (assuming we're still in a KSQL or SQL script). It is followed by `\t`, the K command to time the expression to its right (replacing the leading colon in the KSQL expression). The expression to the right is `up'[AI; TI; BI; X]`, which is the equivalent to the KSQL statement `up Each[AI, TI, BI, X]`.

There are variations of each called *each-left* and *each-right*; they are denoted `\:` and `/:`, respectively, in K and named **Eachleft** and **Eachright** in KSQL. For example, connection handles are integers (see "Connecting", above). If **H** is a vector of connection handles to more than server, the following K expression sends the message **m** to every server whose connection is in **H**.

*H 4:\: m*

# 6 Database Topics

## Multi-Threading

There are two ways for a database product to execute multiple requests simultaneously: multiple threads and multiple servers. Threads are so-called light-weight processes that require less system resources than ordinary processes. A database server that simultaneously executes simultaneous multiple requests uses multiple threads. The server itself is an ordinary process; multiple servers executing simultaneous requests means multiple processes, not multiple threads.

Assuming that everything is running on one computer, neither multiple threads nor multiple servers have much effect on through-put unless there are multiple CPU units. When there are multiple CPUs, many database products (including Kdb) recommend multiple servers (one per CPU) where, for applications doing updates or inserts, all changes to any particular table are routed through one server to avoid costly synchronization of simultaneous modifications.

Kdb servers do not employ multiple threads. The reason is that a high through-put rate o a single CPU, together with the multiple server alternative for multiple CPUs, place multiple threads at a low priority. That is, Kdb installations strive to make the servers CPU-bound (hence the high through-put rate), in which case multiple threads on a single CPU have a negligible effect on through-put. The example below illustrates this point.

This is not to say that multiple threads could never be useful in Kdb servers and that Kdb servers will never be multi-threaded. Since active Kdb databases consist of mapped files and multiple threads can share mapped files, there are situations where multiple threads could provide an easy and effective way to exploit multiple CPUs with a single server.

### Example

This example consists of two simultaneous requests of a single database server running on a host computer with one CPU. Suppose the two requests would use, respectively, $X$ and $Y$ CPU seconds if executed independently. On a single-threaded server the two requests take $X+Y$ total CPU seconds and one goes first while the other waits. Thus “elapsed time” looks like $X$ or $Y$ seconds for the one that goes first and $X+Y$ seconds for the one that waits.

On a multi-threaded server they also take *X+Y* total CPU seconds, but they run at the same time. Assuming that they equally share the CPU while both are running, elapsed time looks like *2*min(X,Y)* for one request and *max(X,Y)+min(X,Y)* for the other. In particular, if *X* equals *Y* the elapsed time of both requests is *2*X* with multiple threads, while in the single-thread case only the one executed last has elapsed time *2*X*; the one executed first has elapsed time *X*. On the other hand, if *X* is much greater than *Y* the elapsed time of the shorter-running request is only *2*Y* seconds with multiple threads, but if the longer-running request goes first in the single-thread case, it is the much greater *X+Y* seconds. One can generally conclude that elapsed time is more uniformly correlated to resource consumption when multiple threads are used. However, when requests are executed fast enough then other factors dominate total elapsed time (for example, network transport time), the different effects between single-threadedness and multi-threadedness become negligible.

# 7 The C - K Interface

## Introduction

This chapter defines the API for calling C functions from K and K functions from C. The C functions that make up the API are listed in the table below, together with the sections in which they are described. C functions called from K must manage K arguments and produce K results. C programs that call K functions must create K arguments and manage K results. The API functions for managing K data are listed in the first 5 rows of the table. The next-to-last row lists API functions for calling K functions and accessing K data from C. The mechanism for calling C functions from K is part of the K language and therefore does not appear in this table; see “Linking to C Functions in K”.

**K <-> C Interface Functions**

| **C Function** | **Section Reference** |
|---|---|
| `gi`, `gf`, `gc`, `gs`, `gn`, `sp` | Creating K Atoms, Data Types |
| `gtn`, `gnk`, `gp`, `gpn`, `gsk` | Creating K Lists |
| `Ki`, `Kf`, `Kc`, `Ks` | Accessing and Modifying K Atoms |
| `KI`, `KF`, `KC`, `KS`, `KK`, `kap` | Accessing and Modifying K Lists |
| `dj`, `jd` | Date Conversion |
| `kerr` | Signalling a K Error |
| `sdf`, `scd` | Registering K Event Loop Callbacks |
| `ksk`, `sfn` | Calling K From C |
| `cd`, `ci` | Managing Reference Counts |

## Compilation

C functions that will be called from a K program must be defined as entry points in an NT DLL (file extension `.dll`) or a Linux SO (file extension `.so`).

## *Header and Lib files*

These files can be found at www.kx.com/a/k/connect. They are `K20.lib`, `K20.h` and `K20x.h`. Include `K20x.h` in your C files and use `K20.lib` for linking. There are brief comments in `K20.h` and `K20x.h` summarizing the contents of this document.

## *The C Structure of the K Data Object*

The internal format of K data objects is defined in `K20.h` by the recursive C-structure named `K`. The members of the K structure are

- `c` – reference count of the object
- `t` – the data type of the object
- `n` – the number of data items when the object is a list or dictionary

The structure members are primarily for reference. However, there are occasions when you must manage the reference count (see "Managing Reference Counts").

## *Data Types*

The data types of K objects are represented by integer values, as follows.

| | |
|---|---|
| `6` | – atomic nil |
| `5` | – dictionary |
| `4` | – symbol atom, i.e. `sp`(null-terminated character string)[1] |
| `3` | – character atom (unsigned) |
| `2` | – double atom |
| `1` | – integer atom |
| `0` | – general list whose items are other K objects |
| `-1` | – integer list (vector) |
| `-2` | – double list (vector) |
| `-3` | – character list (vector) |
| `-4` | – symbol list (vector), i.e. each item is `sp`(null-terminated character string)[1] |

1. The API function `sp` internalizes its character string argument in K for optimized searches, but does not create a K object.

## *Creating K Atoms*

There are atomic constructors for each type of K atom. They are

- `gi` – generate an integer atom, as in `gi(3)` or `gi(i)` for `int i`;
- `gf` – generate a floating-point atom, as in `gf(3.5)` or `gf(a)` for `double a`;
- `gc` – generate a character atom, as in `gc('c')` or `gc(a)` for `unsigned char a`;
- `gs` – generate a symbol atom, as in `gs(sp("price"))` or `gs(sp(s))` for `char *s`;
- `gn` – generate the nil atom, as in `gn()`.

## *Creating K Lists*

There are several list constructors. The most general is `gtn(type,count)`. For example, `gtn(-1,5)` creates an integer vector of length 5. Valid types are 0, -1, -2, -3, -4. Valid counts are non-negative integers.

The constructor `gnk` creates a list from its arguments. It is useful for small lists, particularly for building argument lists to K functions called from C. The first argument to `gnk` is the number of arguments that follow, which can be from zero to eight. It is also the length of the result list. The remaining arguments are the items of the result, in order. For example,

```
gnk(5,gi(2),gf(3.4),gc('a'),gs(sp("abc")),gn());
```

is a list of 5 items. An argument other than first can be any K object.

### *Creating Character Vectors*

A K character vector can be created from a null-terminated string using the constructor `gp`, as in

```
gp("abcd");
```

The constructor `gpn` is used to select a specific number of characters from the front of a C character vector or string, as in

```
gpn(cv,10);
```

The result is a K character vector of length 10.

### *Creating Dictionaries*

A K dictionary is a list of type 5 consisting of symbol-value pairs. A dictionary can be created with the API function `gtn` for creating lists; `gtn(5,n)` is a dictionary with `n` entries.

The API function `gsk` is a useful tool for creating symbol-value pairs. For example,

```
   gsk("abc",gf(2.71));
```

The access function for general lists, KK, can be used to access and replace dictionary items, just as if the dictionary is a list of type 0. See “The Access Function for a General List”. The API function for appending to a general list, `kap`, can be used to append a symbol-value pair to a dictionary. See “Appending to a K List”.

## *Accessing and Modifying K Atoms*

If `x` is an integer atom then `Ki(x)` is a C `int`.

If `x` is a floating-point atom then `Kf(x)` is a C `double`.

If `x` is a character atom then `Kc(x)` is a C `unsigned char`.

If `x` is a string atom (symbol) the `Ks(x)` is a C `char*`.

The value of an atom can also be modified, as in

```
   Ki(x)=2;
```

and

```
   Ks(x)=sp("abc");
```

## *Accessing and Modifying K Lists*

If `x` is an integer vector then the `i`th item `KI(x)[i]` is a C `int`.

If `x` is a floating-point vector then the `i`th item `KF(x)[i]` is a C `double`.

If `x` is a character vector then the `i`th item `KC(x)[i]` is a C `unsigned char`.

If `x` is a string (symbol) vector the `i`th item `KS(x)[i]` is a C `char*`.

A list item can also be modified, as in

```
   KF(x)[2]=3.5;
```

## *The Access Function for a General List*

The access function for a general list is denoted by `KK`. It applies to K objects of type 0. The `i`th item of a K object `x` of type 0, `KK(x)[i]`, is also a K object. If, for example, that item is an integer vector, then its items can be accessed by `KI(KK(x)[i])[j]`.

## *Appending to a K List*

The API function for appending to a K list is `kap`. Use this function if the length of the result is not known when the list is created. Start with an empty list, as in

```
x=gtn(-1,0);
```

which is an empty integer vector. Whenever a new item to be appended is available, append it to `x` with `kap`. For example, append the value of C `int a` to `x` as follows.

```
kap(&x,&a);
```

If `y` is a general list (type 0) then the second argument of `kap` can be a pointer to any other K object; that K object becomes a new item of `y`. `kap` is item-to-list append, not list-to-list.

# *Calling K from C*

K can be started from a C program, K scripts can be loaded and K expressions can be executed with results returned to C. The API function that does all this is `ksk`.

First of all, K must be initialized with `ksk("",0)`. Note: this function always returns a result. If the result is meaningless it can be immediately freed using the API function `cd`; see “Managing Reference Counts”. You will often see C statements, like the following, that call `ksk` and immediately free the result with `cd`.

```
cd(ksk("",0));
```

The character string argument can hold any valid K expression or command. For example,

```
r=ksk("2+3",0);
```

or

```
cd(ksk("\\l script.k",0));
```

The character string can contain a K function definition or the name of an existing K function, in which case the second argument must be K list of the function arguments. For example,

*ksk("{x+y}",gnk(2,gi(2),gi(3)));*

## *C calling K calling C*

It is conceivable that an application has a C main program that calls K functions that, in turn, call C functions. In this case it is not necessary to compile a separate library that is linked into K. The functions to be called from K can be compiled with the main program and registered as callbacks with K from the main program.

For example, suppose the C function is

```
K f(K x,K y){Ki(x)+Ki(y);}
```

`f` can be registered in K using the API function `sfn`, as follows.

```
sfn("g",f,2);
```

where `g` is the name by which `f` is called from K and 2 is the number of arguments of `f`. The K function can now be called with `ksk`.

```
ksk("g[2;3]",0);
```

or

```
ksk("g",gnk(2,gi(2),gi(3)));
```

# *Linking to C Functions in K*

The primitive dyadic K function `2:` defines links to C functions. A result of `2:` is a K function that, when called, calls the C function to which it is linked. The left argument to `2:` names the DLL or SO file (with path) in which the C function is found. The right argument is a pair; its first item is a character atom or vector holding the name of the C function and its second argument is the number of arguments in the C function.

For example, the C function

```
K f(K x,K y){....}
```

is linked into K with

```
g:obj 2: ("f"; 2)
```

The K program that calls `f` is named `g`.

It is up to the programmer to make sure that the arguments to the K function exactly match the arguments of the C function.

## *Signalling a K Error*

A K error can be signalled by a C function called from K with API function `kerr`. For example,

```
if(0>=x->t)return kerr("x must be an atom");
```

The effect, which is to signal an error in K with the message “x must be an atom”, is the same as if the error was signalled by a K function. Note that it may be necessary to clean up work in progress just before an error is signalled; see “Managing Reference Counts”.

## *Managing Reference Counts*

Reference counting is a standard technique in data management to avoid making unnecessary copies of data. When writing C programs that create K objects, there are circumstances when you must manage the reference counts of those objects.

Every K object has reference count 1 when it is created. If a K atom or vector is created in C function that is called from K and returned as that function’s result, it will be managed by K from the point of return onwards. However, if temporary K atoms and vectors that are not part of the result have been constructed, or if a K atom or vector result is under construction when an error is signalled, then the reference counts of those objects must be decremented before the function returns. Their reference counts then decrease by 1 to 0, indicating that the storage allocated to these objects can be freed.

Similarly, when a K function is called from a C program its result is returned to the C program. The reference count of this result should be decremented when it is no longer in use.

Reference counts are decremented by the API function `cd`; for example, `cd(x)` decrements the reference count of the K object `x`.

Knowing when to decrement reference counts is analogous to knowing when to free temporary storage allocated with `malloc()`, but trickier because `cd` is recursive. For example, every item in a general K list (type 0) is also a K object with its own reference count. If a K object `x` is created and then inserted in the general K list `y`, and if the reference count of `y` is subsequently decremented, the reference count of `x` will be decremented automatically and therefore should not also be decremented explicitly.

Reference counts can also incremented, with the API function `ci`. Typically, the reference count of a K object must be incremented once for every independent use after the first use. (An *independent use* is one that may cause the object's reference count to be decremented once in the future.) For example, if a K atom or vector is created and then inserted twice into a general k list, its reference count must be explicitly incremented once.

For convenience, the function `ci` returns its reference-count-incremented argument as its result. For example, suppose that the K object `r` is the result of a C function called from K and `r` is not created in that function. The reference count of `r` can be incremented in the `return` statement, as follows.

```
   return ci(r);
```

It is best to avoid complicated reference count situations and leave memory management to K by moving K objects to the K side of the interface and referencing them there. For example, the K object `bonddata` can be moved to a global variable in the `.u` directory of K with the same name by

```
   t=gnk(1,bonddata), r=ksk("{.u.bonddata::x}",t), cd(t)
```

The K object `.u.bonddata` can now be used in any K function, in particular, ones called from C.

## *Date Conversion*

Both `jd` and `dj` take a C `int` argument and return an C `int` result. The argument to `jd` is an integer of the form `yyyymmdd` and the result is a Julian day count. The argument to `dj` is a day count and the result is a `yyyymmdd` integer. These functions are useful for date arithmetic. For example, to add 5 days to a date, first convert to Julian days with `jd`, add 5 to the result and convert back with `dj`.

## *Registering K Event Loop Callbacks*

It is possible to send and receive non-K IPC messages in a K application by managing the non-K connection in C functions and registering the socket callbacks in the K event loop with the API function `sdf`. This function takes two arguments, the socket id (for an *accept* callback) and the callback function, as in

```
   sdf(sockid,fn);
```

Use the negative of the socket id to establish a *read* callback, as in

```
   sdf(-sockid,gn);
```

The callback functions `fn` and `gn` both take one argument, which is the socket id. Close the socket with the API function `scd`, e.g. `scd(sockid)`.

## *Example: Summing Two K objects*

The following C function illustrates straightforward manipulation of K objects by summing two K integer objects, both of which are either an atom or vector. The function header is for a DLL. The best way to sum two K objects is, of course, using K, as in

```
   a = gtn(2,x,y); s = ksk("+",a); cd(a);
```

```
__declspec(dllexport) K my_sum(K x,K y)
   {  K z;
      int i;
      // case: both x and y are atoms
      if(1==x->t&&1==y->t) return gi(Ki(x)+Ki(y));
      // case: x is an atom and y is a vector
      if(1==x->t&&-1==y->t){
         K z=gtn(-1,y->n);  // z is the same length as y
         for(i=0;i<y->n;i++)KI(z)[i]=Ki(x)+KI(y)[i];
         return z;
      }
      // other cases: vector x, atom y and vectors x, y
   }
```

It is left to the reader to complete this example.

## *Example: Accessing a Kdb Server with KDBC*

The following example can be found in www.kx.com/a/kdb/connect/kdbc.txt. It illustrates communication between a C program and a Kdb database server. The Kdb server, listening on port 2001, is started with the following command, which also creates the database from the SQL script `sp.s` (in the Kdb download from the kx website).

```
k db sp.s -p 2001
```

The C program connects to the database server, sends it an SQL query and processes the result. Note that the K result returned by the Kdb server is a 3-item general list holding the column names of the result table, the data in column order and the data types.

```
#include "k20x.h"
extern printf(S s,...),gets(S);
main()
{  K q,r,n,d,t; // query,result,names,data,types
   cd(ksk("h:3:(`;2001)",0)); // connect
   cd(ksk("k:{h 4:x}",0)); // a remote exec function
   q=gp("select sum qty by p from sp"); // KSQL query
   r=ksk("k",q),cd(q); // result, free query
   n=KK(r)[0],d=KK(r)[1],
   t=KK(r)[2]; // names, inverted data, types
      printf("columns: %d\n",n->n); // number of columns
   {I i=0;for(;i<n->n;++i)
      printf("%s %s\n",KS(n)[i],KS(t)[i]);} // name&type
   printf("rows: %d\n",KK(d)[0]->n); // number of rows
   printf("%s %d\n",KS(KK(d)[0])[0],
      KI(KK(d)[1])[0]); // first row(int,varchar)
   cd(r); // free result
   cd(ksk("3:h",0)); // close connection
   {C b[1];printf("\ndone ... ");gets(b);} // prompt
   return 0;}
```

## *Example: Evaluating KSQL Statements*

KSQL stored procedures written in C use the Kdb entry point **.d.r** to evaluate KSQL statements (see “Custom Analytics and Stored Procedures” on page 20). For example, the following C character string holds a KSQL **update** statement.

```
char s[]="update qty:2*qty from 'OrderItems' where \
     orderid=6099"
```

This constant is placed in a K character vector as follows.

```
K v;
v=gtn(-3,(strlen(s));
memcpy(KC(v), s, strlen(s));
```

The function **.d.r** is then called as follows.

```
K a;
a=gtn(1,t);
cd(ksk(".d.r",a));
cd(a);
```

The result of `ksk` is immediately freed with `cd` because this **update** statement does not produce a needed result. Freeing the K argument list `a` also frees the character vector `v`.

## *Example: A Remote Procedure Call*

The KDBC message format for a bulk update is

*("insert","table",bulk_data)*

(see “Bulk Updates” on page 29). In this case the remote procedure is **insert**. The corresponding K message has a slightly different arrangement. First, the arguments to the remote procedure, which in this example is **insert,** must be grouped. Also, **insert** requires that its table name be a symbol, and therefore *"table"* must be replaced with a symbol. Assuming the bulk data has already been constructed as the K object `bd`, the **insert** message can be constructed as follows.

```
msg = gnk(2,gp("insert"),gnk(2,gs(sp("table")),bd))
```

The message can be sent to the Kdb server using the K function **k** defined in “Example: Accessing a Kdb Server with KDBC” on page 54).

```
   cd(ksk("k",msg));
```

The reference count of the `ksk` result is decremented immediately because it will not be used.  The reference count of the message should also be decremented.

```
   cd(msg);
```