# The Kerf Database and Administration

Kerf can be viewed as a programming language that features convenient database-like syntax and functionality, or it can be viewed as a powerful column-oriented database which happens to include a general purpose scripting language for writing more complex queries and applications. In either case, Kerf's functionality overlaps with SQL, but there are many important differences between Kerf and, say, MySQL.

This guide is intended to provide a quick introduction to Kerf for users who have some familiarity with conventional SQL-based relational databases and who wish to use Kerf in a similar context. For more detailed explanations of Kerf features and syntax, refer to *The Kerf Programming Language*.

## 1 Launching Kerf

To start Kerf, open a terminal and type `kerf -q`. This will drop you into the Kerf Read-Evaluate-Print Loop (REPL). The REPL allows you to type expressions interactively with immediate results. This is the preferred means of interacting with Kerf. In all the following examples, blue text indicates something typed by a user at the Kerf REPL, and orange text occasionally indicates a comment- no need to type those. Everything else is a result printed by Kerf itself:

```
> ./kerf -q
KeRF> 2 + 3
  5

KeRF> range 10
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
KeRF>
```

If you're having trouble launching Kerf, refer to the installation guide in *The Kerf Programming Language*.

## 2 Loading Data

The first step in designing any SQL database is the creation of a *schema* which describes the structure, names, datatypes and constraints of one or more tables. For example, we might declare a table which stores customer information like so:

```
CREATE TABLE Customers(
        account_number integer     PRIMARY KEY,
        name           text        NOT NULL,
        ticker_code    varchar(5)  NOT NULL,
        created_on     date        NOT NULL
);
```

Kerf table columns can store elements of any datatype, and tables can be easily created, serialized or destroyed on the fly. This is particularly helpful when using Kerf to pull together potentially messy data from a variety of sources. If a column consists entirely of a uniform simple datatype, Kerf will automatically *vectorize* the column, allowing it to be stored compactly and manipulated efficiently. It is only necessary to explicitly specify column datatypes when importing data from text-based serialization formats, as we will see shortly.

Defining an empty table in Kerf to represent similar information to the above requires no more than stating the names of columns. Keep in mind that unlike SQL, Kerf tables are first-class values. Kerf tables need not be global and they can be passed to and returned from functions just like numbers, lists or dictionaries.

```
KeRF> Customers: {{account_number, name, ticker_code, created_on}}

|account_number|name|ticker_code|created_on|
|              |    |           |          |
```

In SQL, we add data to a table by using the INSERT statement:

```
INSERT INTO Customers VALUES (1, 'Hudsucker Industries', 'HUD', NOW());
```

Kerf accepts very similar syntax:

```
KeRF> INSERT INTO Customers VALUES [1, 'Hudsucker Industries', 'HUD', now()]
  "Customers"

KeRF> Customers

|account_number|name                |ticker_code|created_on             |
|             1|Hudsucker Industries|        HUD|2016.05.24T21:38:28.448|
```

Kerf also has a variety of helpful built-in functions for data import and export. For example, we can easily populate a table from a CSV file `customers.csv`:

```
account_number,name,ticker_code,created_on
1,Hudsucker Industries,HUD,24-May-16 21:38:28
2,Kerf Inc.,KERF,24-May-16 21:39:03
3,Weyland-Yutani Corp.,WAYU,24-May-16 21:39:36
4,Omni Consumer Products,OCP,24-May-16 21:53:40
```

```
KeRF> // "ISSZ" specifies imported column types as int, string, string, timestamp:
KeRF> Customers: read_table_from_csv("customers.csv", "ISSZ", 1)

|account_number|name                  |ticker_code|created_on             |
|             1|  Hudsucker Industries|        HUD|2016.05.24T21:38:28.000|
|             2|             Kerf Inc.|       KERF|2016.05.24T21:39:03.000|
|             3|  Weyland-Yutani Corp.|       WAYU|2016.05.24T21:39:36.000|
|             4|Omni Consumer Products|        OCP|2016.05.24T21:53:40.000|
```

With a bit of data loaded, we can explore it interactively from the REPL. Observe how we can conveniently access columns or rows from a table and that Kerf has correctly vectorized its columns:

```
KeRF> Customers[1]
┌──────────────┬──────────┬───────────┬───────────────────────┐
│account_number│name      │ticker_code│created_on             │
├──────────────┼──────────┼───────────┼───────────────────────┤
│             2│Kerf Inc. │       KERF│2016.05.24T21:39:03.000│
└──────────────┴──────────┴───────────┴───────────────────────┘

KeRF> Customers['ticker_code']
  ["HUD", "KERF", "WAYU", "OCP"]

KeRF> Customers['created_on']['minute']
  [38, 39, 39, 53]

KeRF> count Customers
  4

KeRF> kerf_type_name Customers['created_on']
  "stamp vector"

KeRF> kerf_type_name mapdown Customers
  ["integer vector", "list", "list", "stamp vector"]

KeRF> xkeys Customers
  ["account_number", "name", "ticker_code", "created_on"]
```

## 3 Searching and Sorting

Kerf provides equivalents to a number of SQL's basic statements for extracting data:

```
KeRF> People: read_from_path("people.bin")

┌────────────────┬───┬──────┬─────────────────────┐
│name            │age│gender│job                  │
├────────────────┼───┼──────┼─────────────────────┤
│Hamilton Butters│ 37│     M│              Janitor│
│       Emma Peel│ 29│     F│         Secret Agent│
│ Jacques Maloney│ 48│     M│Private Investigator│
│   Renee Smithee│ 31│     F│           Programmer│
│   Karen Milgram│ 16│     F│              Student│
│   Chuck Manwich│ 29│     M│              Janitor│
│ Steak Manhattan│ 18│     M│         Secret Agent│
│ Tricia McMillen│ 29│     F│        Mathematician│
└────────────────┴───┴──────┴─────────────────────┘

KeRF> SELECT name, job AS occupation FROM People WHERE age > 30

┌────────────────┬────────────────────┐
│name            │occupation          │
├────────────────┼────────────────────┤
│Hamilton Butters│             Janitor│
│ Jacques Maloney│Private Investigator│
│   Renee Smithee│          Programmer│
└────────────────┴────────────────────┘

KeRF> SELECT name FROM People GROUP BY gender

┌──────┬────────────────────────────────────────────────────────────────────┐
│gender│name                                                                │
├──────┼────────────────────────────────────────────────────────────────────┤
│     M│[Hamilton Butters, Jacques Maloney, Chuck Manwich, Steak Manhattan]│
│     F│         [Emma Peel, Renee Smithee, Karen Milgram, Tricia McMillen]│
└──────┴────────────────────────────────────────────────────────────────────┘
```

Basic collective operations such as sum and count can be used in familiar SQL syntax, but they are also general-purpose built-in functions. There are usually several approaches for solving a given problem in Kerf:

```
KeRF> SELECT count(job) AS employed FROM People GROUP BY job
┌────────────────────┬────────┐
│job                 │employed│
├────────────────────┼────────┤
│             Janitor│       2│
│        Secret Agent│       2│
│Private Investigator│       1│
│          Programmer│       1│
│             Student│       1│
│       Mathematician│       1│
└────────────────────┴────────┘

KeRF> count mapdown part People["job"]
        [2, 2, 1, 1, 1, 1]
```

Subqueries can be arbitrarily nested and there's no requirement to give them names:

```
KeRF> SELECT job FROM (SELECT count(job) AS n FROM People GROUP BY job) WHERE n = 1

┌─────────────────────┐
│job                  │
├─────────────────────┤
│Private Investigator │
│           Programmer│
│              Student│
│        Mathematician│
└─────────────────────┘
```

Kerf has its own approach for equivalents to SQL `ORDER BY` and `LIMIT` clauses. The `ascend` function produces a permutation vector which can be used to reorder a list or table into ascending value. For descending order, use `descend`:

```
KeRF> ascend SELECT age FROM People
  [4, 6, 1, 5, 7, 3, 0, 2]

KeRF> // equivalent to "SELECT * FROM People ORDER BY age ASC"
KeRF> People[ascend SELECT age FROM People]

┌────────────────┬───┬──────┬────────────────────┐
│name            │age│gender│job                 │
├────────────────┼───┼──────┼────────────────────┤
│   Karen Milgram│ 16│     F│             Student│
│  Steak Manhattan│ 18│     M│        Secret Agent│
│       Emma Peel│ 29│     F│        Secret Agent│
│   Chuck Manwich│ 29│     M│             Janitor│
│  Tricia McMillen│ 29│     F│       Mathematician│
│   Renee Smithee│ 31│     F│          Programmer│
│Hamilton Butters│ 37│     M│             Janitor│
│  Jacques Maloney│ 48│     M│Private Investigator│
└────────────────┴───┴──────┴────────────────────┘
```

As for `LIMIT`, we can define a custom function:

```
KeRF> limit_rows: {[n, t] first(min(count t, n), t)};

KeRF> // equivalent to "SELECT name, age FROM People LIMIT 3"
KeRF> limit_rows(3, SELECT name, age FROM People)

┌────────────────┬───┐
│name            │age│
├────────────────┼───┤
│Hamilton Butters│ 37│
│       Emma Peel│ 29│
│  Jacques Maloney│ 48│
└────────────────┴───┘
```

## 4 Joins

In Kerf, joins are provided by built-in functions. `left_join` is similar to the familiar SQL `LEFT JOIN` construct. It combines every row of the left table with the rows of the right table that match on a particular column, substituting appropriate `null` values where there is no corresponding row in the right table.

```
KeRF> a: {{name: ["Bob Johnson", "Kurt Hardfist", "Beef Manly"], age: 31 28 47}}

┌─────────────┬───┐
│name         │age│
├─────────────┼───┤
│  Bob Johnson│ 31│
│Kurt Hardfist│ 28│
│   Beef Manly│ 47│
└─────────────┴───┘

KeRF> b: {{name: ["Kurt Hardfist", "Madeline Potter", "Bob Johnson"],
>          net_worth: 43052.10 98100.5 12.75}}

┌───────────────┬─────────┐
│name           │net_worth│
├───────────────┼─────────┤
│  Kurt Hardfist│  43052.1│
│Madeline Potter│  98100.5│
│    Bob Johnson│    12.75│
└───────────────┴─────────┘

KeRF> left_join(a, b, "name")

┌─────────────┬───┬─────────┐
│name         │age│net_worth│
├─────────────┼───┼─────────┤
│  Bob Johnson│ 31│    12.75│
│Kurt Hardfist│ 28│  43052.1│
│   Beef Manly│ 47│      nan│
└─────────────┴───┴─────────┘
```

It is also possible to require matches on several columns or to specify that differently-named columns across two tables should be compared for equality.

A less familiar type of join is `asof_join`, which can be used to perform "fuzzy joins". The first three arguments behave identically to `left_join`. The fourth argument specifies a column or columns which will match if the values in the right table are less than or equal to the values they're compared with in the left table. `asof_join` is particularly useful in time-series database queries, as it allows you to easily normalize data. For example, associating timestamped transactions with the the closest preceding event.

## 5 The External World

We've already seen how Kerf can import data from files on disk. Kerf provides helper functions for handling a wide variety of common data interchange formats, including fixed-column formats, delimited formats like CSV, and JSON. It is also equipped with a custom high-performance disk serialization format. If a table is disk-backed, it will automatically be serialized and persisted across sessions as it is changed. Kerf will efficiently page portions of disk-backed tables in and out of working memory, allowing you to work with very large datasets:

```
KeRF> c: create_table_from_csv("Customers.bin", "Customers.csv", "ISSZ", 1)

┌──────────────┬──────────────────────┬───────────┬───────────────────────┐
│account_number│name                  │ticker_code│created_on             │
├──────────────┼──────────────────────┼───────────┼───────────────────────┤
│             1│  Hudsucker Industries│        HUD│2016.05.24T21:38:28.000│
│             2│             Kerf Inc.│       KERF│2016.05.24T21:39:03.000│
│             3│  Weyland-Yutani Corp.│       WAYU│2016.05.24T21:39:36.000│
│             4│Omni Consumer Products│        OCP│2016.05.24T21:53:40.000│
└──────────────┴──────────────────────┴───────────┴───────────────────────┘

KeRF> INSERT INTO c VALUES [5, "Water and Power", "TANKG", now()]
  "c"

KeRF> \\

> ./kerf -q
KeRF> Customers: open_table("Customers.bin")

┌──────────────┬──────────────────────┬───────────┬───────────────────────┐
│account_number│name                  │ticker_code│created_on             │
├──────────────┼──────────────────────┼───────────┼───────────────────────┤
│             1│  Hudsucker Industries│        HUD│2016.05.24T21:38:28.000│
│             2│             Kerf Inc.│       KERF│2016.05.24T21:39:03.000│
│             3│  Weyland-Yutani Corp.│       WAYU│2016.05.24T21:39:36.000│
│             4│Omni Consumer Products│        OCP│2016.05.24T21:53:40.000│
│             5│       Water and Power│      TANKG│2016.05.24T21:57:32.385│
└──────────────┴──────────────────────┴───────────┴───────────────────────┘
```

Kerf features a simple Inter-Process Communication (IPC) mechanism, allowing you to network remote instances of the interpreter. Supposing we had a second Kerf process on `localhost` listening on port 1234,

```
KeRF> c: open_socket("localhost", "1234")
  4
KeRF> send_async(c, "data: $1",[range 10])
  1
KeRF> send_sync(c,"2 * data")
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Kerf's documentation demonstrates how the same system can be used to easily interoperate between Kerf and other popular languages such as Java or Python.

If there's special functionality you need which isn't built into Kerf already, you can use Kerf's Foreign Function Interface (FFI) to produce wrappers for any existing C codebase:

```
#include <stdio.h>
#include "kerf_api.h"

KERF foreign_function_example(KERF argument) {
        int64_t value = argument->i;
        printf("Hello from C! You gave me a %d.\n", (int)value);
        return 0;
}
```

```
KeRF> f: dlload("example.dylib", "foreign_function_example", 1)
  {OBJECT:foreign_function_example}
KeRF> f(42)
Hello from C! You gave me a 42.
```

Refer to *The Kerf Programming Language* for several practical examples of using the FFI, including an extended discussion of the creation of a custom wrapper for `libcurl` and using it to perform HTTP requests against pre-existing web APIs.