

# SQL in KDB-X Quickstart

---

*This quickstart walks you through creating, inserting into, querying, and dropping a table in KDB-X using SQL. You will use basic Data Definition Language (DDL) and Data Manipulation Language (DML) commands.*

## 1. Create a table

Use **CREATE TABLE** to define a new table in memory. The example below creates an empty table with two columns: **vendor** (string) and **fare** (float).

```
s)CREATE TABLE tripsFare (vendor varchar, fare float)
.s.e"SELECT * FROM tripsFare"
```

You can also create a more complex table with multiple data types and constraints:

```
s)CREATE TABLE cars (
  id int,
  Name varchar(250),
  "Miles_per_Gallon" smallint,
  "Cylinders" smallint,
  "Displacement" smallint,
  "Horsepower" smallint,
  "Weight_in_lbs" smallint NOT NULL,
  "Acceleration" smallint,
  "Year" date NOT NULL,
  "Origin" varchar(60)
);
.s.e"SELECT * FROM cars"
```

### Note:

Use ``char(1)`` to define a column of `q` type ``char``. Use ``char(n>1)`` or ``varchar(n)`` for longer strings.  
See [\[reference\]\(reference.md#data-types\)](#) for the full list of supported types.

## 2. Insert data

Use **INSERT** to add rows to an existing table. The following example inserts a single row into **tripsFare**.

```
s)INSERT INTO tripsFare(vendor,fare) VALUES ('CMT',100)
s)SELECT * FROM tripsFare
```

You can also insert multiple rows at once:

```
s)INSERT INTO tripsFare(vendor,fare) VALUES ('DDS',301),('CMT',589)
s)SELECT * FROM tripsFare
```

The next example inserts two rows into the `cars` table:

```
s)CREATE TABLE cars (
  id int, Name varchar(250), "Miles_per_Gallon" smallint,
  "Cylinders" smallint, "Displacement" smallint, "Horsepower" smallint,
  "Weight_in_lbs" smallint NOT NULL, "Acceleration" smallint,
  "Year" date NOT NULL, "Origin" character varchar(60)
);

s)INSERT INTO cars VALUES
  (1, 'chevrolet chevelle malibu', 18, 8, 307, 130, 3504, 12, '1970-01-01',
  'USA');

s)INSERT INTO cars VALUES
  (2, 'volkswagen 1131 deluxe sedan', 26, 4, 97, 46, 1835, 21, '1970-01-01',
  'Europe');
```

### 3. Query data

Use `SELECT` to retrieve rows. You can return all data or filter it with conditions.

```
s)SELECT * FROM tripsFare
```

### 4. Drop a table

Use `DROP TABLE` to delete a table from memory. The example below drops the `tripsFare` table if it exists.

```
s)DROP TABLE IF EXISTS tripsFare
```

Congratulations! You have now created, populated, queried, and dropped tables in KDB-X using SQL.

## SQL in KDB-X Reference

---

*This page describes the full SQL reference for KDB-X. It lists supported data types, operators, functions, statements, and compliance. All examples show SQL that KDB-X translates into q at runtime.*

### Data and literals

SQL in KDB-X supports a range of data types and literals. You can define tables using these types and write queries that recognize common literal values such as `true`, `false`, and `null`. This section describes how SQL types map to q datatypes and shows how to use string and type literals in queries.

Data type conversions

The table below shows how SQL types map to SQL in KDB-X types and q datatypes.

| SQL type               | KX SQL | q  | Name                                     |
|------------------------|--------|----|--|
| text                   | a      | 0  | list of character vectors                |
| varchar, char(n>1)     | s      | 11 | symbol                                   |
| char(1)                | c      | 10 | char                                     |
| guid                   | g      | 16 | guid                                     |
| boolean                | b      | 1  | boolean                                  |
| uuid                   | q      | 2  | guid                                     |
| tinyint                | x      | 4  | byte                                     |
| smallint               | h      | 5  | short                                    |
| integer                | i      | 6  | int                                      |
| bigint                 | j      | 7  | long                                     |
| real                   | e      | 8  | real                                     |
| float, double, numeric | f      | 9  | float                                    |
| date                   | d      | 14 | date                                     |
| datetime               | z      | 8  | datetime                                 |
| time                   | t      | 19 | time                                     |
| datetime               | n      | 8  | timespan                                 |
| long                   | m      | 4  | month                                    |
| long                   | u      | 4  | minute                                   |
| long                   | v      | 4  | second                                   |
| timestamp (w/o tz)     | p      | 12 | timestamp                                |
| varchar                | q      | 20 | enum                                     |
|                        | l      | 20 | link (enum used for linked table access) |
|                        | w      |    | 'null' SQL value                         |

Refer to [KX data types](#) for details of how q datatypes are defined.

Info:

Always use uppercase characters when you define a list.

Literals

SQL in KDB-X supports the following literals:

```
false
null
true
```

String literals

SQL in KDB-X converts string literals automatically to the following types when possible:

```
date
time
timestamp
```

Example

```
s)select * from t where date in ('2001-01-01','2002-02-02') and time>'12:03'
```

Type literals

You can use type literals directly. For example:

```
date '2001-01-01'
```

Operators

Operators let you compare values and build expressions in SQL statements.

Arithmetic operators

Use arithmetic operators to perform basic calculations.

| Operator | Description |
|----------|-------------|
| +        | Add         |
| -        | Subtract    |

| Operator | Description              |
|----------|--------------------------|
| *        | Multiply                 |
| /        | Divide                   |
| =        | Equals to                |
| >        | Greater than             |
| <        | Less than                |
| >=       | Greater than or equal to |
| <=       | Less than or equal to    |
| <>       | Not equal to             |

## Examples

- **Add** two columns and create a new column::

```
.s.e"SELECT fare,tip,fare+tip as new_total FROM trips"
```

- **Subtract** one column from another:

```
.s.e"SELECT month,month-1 as prev_month FROM trips"
```

- **Multiply** one column from another and create a new column:

```
// multiply *  
.s.e"SELECT distance,5280*distance as distance_feet FROM trips"
```

- **Divide** a column by a fixed value and create a new column:

```
// divide /  
.s.e"SELECT fare, fare/100 as fare_cents FROM trips"
```

- Filter rows using **equality**:

```
// equal to =  
.s.e"SELECT * FROM trips where passengers=2"
```

- Apply **inequalities**:

```
// equal to =
.s.e"SELECT * FROM trips where passengers=2"

// greater than >=
.s.e"SELECT fare,tip,fare+tip as total FROM trips WHERE tip>0"

// less than <=
.s.e"SELECT * FROM trips WHERE fare<5"

// greater than or equal to >=
.s.e"SELECT * FROM trips WHERE distance>=12"

// less than or equal to <=
.s.e"SELECT * FROM trips WHERE fare<=5"
```

- Exclude rows with **not equal**:

```
// not equal to <>
.s.e"SELECT * FROM trips WHERE passengers<>1"

// not equal to !=
.s.e"SELECT * FROM trips WHERE passengers!=1"
```

Logical operators

Logical operators combine conditions in a **WHERE** clause.

| Operator | Description  |
|----------|--|
| AND      | TRUE if all the conditions separated by AND are TRUE         |
| OR       | TRUE if any of the conditions separated by OR is TRUE        |
| NOT      | Displays a record if the condition(s) is NOT TRUE            |
| BETWEEN  | TRUE if the operand is within the range of comparisons       |
| IN       | TRUE if the operand is equal to one of a list of expressions |
| LIKE     | TRUE if the operand matches a pattern                        |
| IS NULL  | TRUE if the value is NULL                                    |

Examples

- **AND** only returns records that meet ALL criteria:

```
// AND
.s.e"SELECT * FROM trips WHERE passengers=3 AND vendor='DDS'"
```

- **OR** returns records that meet ONE or MORE of the criteria:

```
// OR
.s.e"SELECT * FROM trips WHERE tip>20 OR fare>100"
```

- **IN** specifies multiple values in a **WHERE** clause, which is a shorthand for multiple OR clauses:

```
.s.e"SELECT * FROM trips WHERE payment_type IN ('CASH', 'CREDIT');"
```

- **NOT** returns records that DO NOT meet the criteria:

```
// NOT
.s.e"SELECT * FROM trips WHERE NOT passengers=1"
```

**Info:** This option is equivalent to **<>** and **!=**.

## Between

Use **between** to select values within a given range. The values can be numbers, text, dates or datetimes.

## Examples

- Return records where values fall between two numbers:

```
.s.e"SELECT * FROM trips WHERE fare BETWEEN 10 AND 12;"
```

- Return records that fall alphabetically between two text values:

```
.s.e"SELECT * FROM trips WHERE payment_type BETWEEN 'CASH' AND 'DISPUTE';"
```

- Return records that fall between two datetimes:

```
.s.e"SELECT * FROM trips WHERE pickup_time BETWEEN '2009-01-01 00:30:00' AND  
'2009-01-01 00:35:00';"
```

- Return records for a single day:

```
.s.e"SELECT * FROM trips WHERE date BETWEEN '2009-01-01' AND '2009-01-02';"
```

## Like

Use `like` in a `WHERE` clause to search for a specified pattern in a column. As in standard SQL syntax the following two wildcards are supported:

- The percent sign `%` represents zero, one, or multiple characters.
- The underscore sign `_` represents one, single character.

## Examples

- Use `%` to return all records where a field begins with a letter:

```
.s.e"SELECT * FROM trips WHERE payment_type LIKE 'C%';"
```

- Use `_` to search just leaving out one single character:

```
.s.e"SELECT * FROM trips WHERE payment_type LIKE 'C_EDIT';"
```

## Conditional expressions

Conditional expressions let you return values based on conditions.

```
case [a] when b then x ... else y
coalesce
nullif
```

### Note:

Both the general and 'simple' comparison forms of ``case`` are supported.

## Functions

SQL in KDB-X provides built-in functions you can use to transform, aggregate, and analyze data in queries. These functions cover string manipulation, datetime operations, mathematical calculations, and type casting.

### String functions

SQL in KDB-X supports the following string functions:

|                    |  |
|--------------------|--|
| <code>  </code>    | <code>position(x in y)</code>          |
| <code>left</code>  | <code>position(x,y)</code>             |
| <code>right</code> | <code>substring(x from y)</code>       |
| <code>lower</code> | <code>substring(x from y for z)</code> |

|        |                  |
|--------|------------------|
| upper  | substring(x,y,z) |
| length | concat           |
| trim   | ltrim            |
| rtrim  |                  |

**Note:** "Substring"

There is no pattern matching on `substring`.

## Datetime functions

SQL in KDB-X supports the following datetime functions:

|                       |                   |
|-----------------------|-------------------|
| extract(field from x) | current_date      |
| current_time          | current_timestamp |
| localtime             | localtimestamp    |
| date_trunc            | now               |
| unnest                | xbar              |

## Examples

- Use `extract`:

```
s)select extract(hour from timestamp '2002-09-17 19:27:45')
extract
-----
19
```

- Use `date_trunc`:

```
s)select date_trunc('hour', timestamp '2017-03-17 02:09:30');
date_trunc
-----
2017.03.17D02:00:00.000000000
```

- Use `xbar`:

```
s)select xbar(10,x) from qt('([1 12 23]')
s)select xbar('0D00:10',x) from qt('([0D+10:21 11:32 13:43]')
```

## Math functions

SQL in KDB-X supports the following math functions:

```
div
floor
power
round
stddev
trunc
```

### Examples

- **round:**

```
t:([ a:1.123 1.456 2.532)
s)select round(a) from t

round
-----
1
1
3
```

## Cast

Use **cast** to convert from one data type to another.

```
cast(x as typename) x::typename
```

## Select statements

Combine SQL operations with the **SELECT** statement. The following operations apply:

| Operator | Description   |
|----------|---|
| DISTINCT | Return only distinct (different) values                 |
| LIMIT    | Select a limited number of records                      |
| AS       | Give a table, or a column in a table, a temporary name  |
| ORDER BY | Sort the result set in ascending or descending order    |
| GROUP BY | Groups rows that have the same values into summary rows |
| JOIN     | Combine rows from two or more tables                    |

| Operator | Description   |
|----------|---|
| WHERE    | Filters records                                       |
| HAVING   | Used in conjunction with GROUP BY aggregate functions |

```
select [distinct] ns from t (left|right|inner|cross)
join ..join q on f(t)=g(q)
where c
group by 1,g
having h
order by 1,o asc|desc
limit n
```

Distinct

Use `select distinct` to return only distinct (different) values.

Examples

- Return a unique list of vendor names in the trips table:

```
.s.e"SELECT DISTINCT vendor FROM trips;"
```

Limit

Use `limit` to select a limited number of records, partition-by-partition. This is useful on large tables to reduce the impact on performance.

**Warning:** Using `limit` with an `offset` is not yet available.

Examples

- Return only the first 5 records from a table:

```
// LIMIT
.s.e"SELECT * FROM TRIPS LIMIT 5;"
```

As

Use `as` to give a table, or a column in a table, a temporary name.

Examples

- Renaming the column:

```
.s.e"SELECT vendor as company FROM trips"
```

## Order by

Use **order by** to sort the result set in ascending or descending order.

This keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### Examples

- Sort the records in ascending order:

```
.s.e" SELECT * FROM trips ORDER BY payment_type;"
```

- Sort the records in descending order:

```
.s.e" SELECT * FROM trips ORDER BY payment_type DESC;"
```

**Info:** "Expression types" **order by** expressions may be column names, ordinal numbers of the output columns, or arbitrary expressions.

## Group by

Use **group by** to groups rows that have the same values for one or more columns or expressions into summary rows.

This statement is often used with **aggregate** functions.

### Examples

- Return a count of the number of trips by a column value:

```
.s.e"SELECT payment_type, COUNT(payment_type) AS count_per_type FROM trips  
GROUP BY payment_type"
```

**Info:** "Expression types" **group by** expressions may be column names, ordinal numbers of the output columns, or arbitrary expressions.

## Aggregates

The following aggregates are supported:

|          |       |
|----------|-------|
| sum      | total |
| avg      | last  |
| count    | max   |
| count(*) | min   |
| first    | sum   |

## Examples

- Using `sum`, `avg`, `count`, `min`, `max`:

```
t:([ s:`AAPL`GOOG`BARC`VOD; p:13.0 12.1 4.0 5.4; a:1 2 3 4)
s)select avg(p), sum(a), min(s), max(s) from t

p      a  s
-----
8.625 10 AAPL
```

- Using `distinct`:

```
q:([ s:`AAPL`AAPL`GOOG`GOOG; p:4?10.0)
s)select count(distinct s) from q

s
-
2
```

**Note:** "Aggregates and distinct" `sum`, `avg`, `count`, `min`, `max` are all supported with `distinct`.

## Join

Use the `join` clause to combine rows from two or more tables, based on a related column between them.

Join types supported include `left`, `right`, `inner`, and `cross`.

**Warning:** Joins may be nested, but `natural`, `using` and `lateral` are not yet implemented.

### Left join

Returns all records from the left-hand table, and the matched records from the right-hand table.

## Examples

- Join two tables based on matching values in a column:

```
.s.e"SELECT * FROM trips LEFT JOIN cash_credit ON trips.payment_type =
cash_credit.payment_type;"
```

**Info:** "No matching records" If no matches exist, records in the left-hand table are returned but joined fields are blank.

**Right join**

Returns all records from the right-hand table, and the matching records from the left-hand table.

**Examples**

- Join two tables based on matching values in a column, returning all records in the right-hand table:

```
.s.e"SELECT * FROM trips RIGHT JOIN cash_credit ON trips.payment_type = cash_credit.payment_type;"
```

**Inner join**

Returns records that have matching values in both tables.

**Examples**

- Join two tables based on matching values in a column, only returning records that are ones in both tables:

```
.s.e"SELECT * FROM trips INNER JOIN cash_credit ON trips.payment_type = cash_credit.payment_type;"
```

**Cross join**

Use **cross join** to return a result set which is the number of rows in the first table multiplied by the number of rows in the second table, if no WHERE clause is used along with **cross join**. This result is a **Cartesian Product**.

**Examples**

- Cross join two tables:

```
cross_tab:.s.e"SELECT * FROM cash_credit CROSS JOIN vendors;"
```

Each row from the first table joins with each row of the second table such that:

| Table       | Number rows |
|-------------|-------------|
| cash_credit | x           |
| vendors     | y           |

| Table                          | Number rows |
|--------------------------------|-------------|
| cash_credit CROSS JOIN vendors | x*y         |

**Note:** "CTE/WITH SELECT"

SQL in KDB-X supports common table expressions (CTEs). Write them in the form:  
  
with t1 as (select...), t2 as (select... t1) select... t2  
  
The system materializes the results unless it can express them as a union of operations on individual partitions.

Subquery

SQL in KDB-X supports scalar subqueries and correlated subqueries. The following operators accept subquery arguments:

< [not] in  
> [not] exists  
=

Combined queries

SQL in KDB-X supports the following operators for combining queries:

union[all]  
intersect[all]  
except[all]

SQL compliance

View ANSI SQL compliance details.

The following table summarizes ANSI SQL features and whether KDB-X supports them:

| Function | Description        | Status | Notes |
|----------|--------------------|--------|-------|
| E011     | Numeric data types |        |       |

|         |  |     |
|---------|--|-----|
| E011-01 | INTEGER and SMALLINT data types (including all spellings)                          |     |
| Yes     |  |     |
| E011-02 | REAL, DOUBLE PRECISION, and FLOAT data types                                       |     |
| Yes     |  |     |
| E011-03 | DECIMAL and NUMERIC data types   |     |
| No      |  |     |
| E011-04 | Arithmetic operators   |     |
| Yes     |  |     |
| E011-05 | Numeric comparison   |     |
| Yes     |  |     |
| E011-06 | Implicit casting among the numeric data types                                      |     |
| Yes     |  |     |
|         |  |     |
|         |  |     |
| E021    | Character string types   |     |
|         |  |     |
| E021-01 | CHARACTER data type (including all its spellings)                                  |     |
| Yes     |  |     |
| E021-02 | CHARACTER VARYING data type (including all its spellings)                          |     |
| Yes     |  |     |
| E021-03 | Character literals   |     |
| Yes     |  |     |
| E021-04 | CHARACTER_LENGTH function  |     |
| Yes     |  |     |
| E021-05 | OCTET_LENGTH function  |     |
| Pending |  |     |
| E021-06 | SUBSTRING function   |     |
| Yes     |  |     |
| E021-07 | Character concatenation  |     |
| Yes     |  |     |
| E021-08 | UPPER and LOWER functions  |     |
| Yes     |  |     |
| E021-09 | TRIM function  |     |
| Pending |  |     |
| E021-10 | Implicit casting among the fixed-length and variable-length character string types | Yes |
|         |  |     |
| E021-11 | POSITION function  |     |
| Yes     |  |     |
| E021-12 | Character comparison   |     |
| Yes     |  |     |
|         |  |     |
|         |  |     |
| E031    | Identifiers  |     |
|         |  |     |
| E031-01 | Delimited identifiers  |     |
| Yes     |  |     |
| E031-02 | Lower case identifiers   |     |
| Yes     |  |     |
| E031-03 | Trailing underscore  |     |
| Yes     |  |     |

|         |   |  |
|---------|---|--|
|         |   |  |
| E051    | Basic query specification                             |  |
| E051-01 | SELECT DISTINCT                                       |  |
| Yes     |   |  |
| E051-02 | GROUP BY clause                                       |  |
| Yes     |   |  |
| E051-04 | GROUP BY can contain columns Not in <select-list>     |  |
| Yes     |   |  |
| E051-05 | Select list items can be renamed                      |  |
| Yes     |   |  |
| E051-06 | HAVING clause   |  |
| Yes     |   |  |
| E051-07 | Qualified * in select list                            |  |
| Yes     |   |  |
| E051-08 | Correlation names in the FROM clause                  |  |
| Yes     |   |  |
| E051-09 | Rename columns in the FROM clause                     |  |
| No      |   |  |
|         |   |  |
| E061    | Basic predicates and search conditions                |  |
| E061-01 | Comparison predicate                                  |  |
| Yes     |   |  |
| E061-02 | BETWEEN predicate                                     |  |
| Yes     |   |  |
| E061-03 | IN predicate with list of values                      |  |
| Yes     |   |  |
| E061-04 | LIKE predicate  |  |
| Partial | Uses q-like syntax, replacing % with *: No underscore |  |
| E061-05 | LIKE predicate: ESCAPE clause                         |  |
| No      |   |  |
| E061-06 | NULL predicate  |  |
| Yes     |   |  |
| E061-07 | Quantified comparison predicate                       |  |
| Pending |   |  |
| E061-08 | EXISTS predicate                                      |  |
| Yes     |   |  |
| E061-09 | Subqueries in comparison predicate                    |  |
| Yes     |   |  |
| E061-11 | Subqueries in IN predicate                            |  |
| Yes     |   |  |
| E061-12 | Subqueries in quantified comparison predicate         |  |
| Pending |   |  |
| E061-13 | Correlated subqueries                                 |  |
| Yes     |   |  |
| E061-14 | Search condition                                      |  |
| Yes     |   |  |
|         |   |  |

|         |   |     |  |
|---------|---|-----|--|
|         |   |     |  |
| E071    | Basic query expressions   |     |  |
|         |   |     |  |
| E071-01 | UNION DISTINCT table operator   |     |  |
| Yes     |   |     |  |
| E071-02 | UNION ALL table operator  |     |  |
| Yes     |   |     |  |
| E071-03 | EXCEPT DISTINCT table operator  |     |  |
| Yes     |   |     |  |
| E071-05 | Columns combined via table operators need Not have exactly the same data type | Yes |  |
|         |   |     |  |
| E071-06 | Table operators in subqueries   |     |  |
| Yes     |   |     |  |
|         |   |     |  |
|         |   |     |  |
| E081    | Basic Privileges  |     |  |
| No      |   |     |  |
| E081-01 | SELECT privilege at the table level   |     |  |
| No      |   |     |  |
| E081-02 | DELETE privilege  |     |  |
| No      |   |     |  |
| E081-03 | INSERT privilege at the table level   |     |  |
| No      |   |     |  |
| E081-04 | UPDATE privilege at the table level   |     |  |
| No      |   |     |  |
| E081-05 | UPDATE privilege at the column level  |     |  |
| No      |   |     |  |
| E081-06 | REFERENCES privilege at the table level                                       |     |  |
| No      |   |     |  |
| E081-07 | REFERENCES privilege at the column level                                      |     |  |
| No      |   |     |  |
| E081-08 | WITH GRANT OPTION   |     |  |
| No      |   |     |  |
| E081-09 | USAGE privilege   |     |  |
| No      |   |     |  |
| E081-10 | EXECUTE privilege   |     |  |
| No      |   |     |  |
|         |   |     |  |
|         |   |     |  |
| E091    | Set functions   |     |  |
|         |   |     |  |
| E091-01 | AVG   |     |  |
| Yes     |   |     |  |
| E091-02 | COUNT   |     |  |
| Yes     |   |     |  |
| E091-03 | MAX   |     |  |
| Yes     |   |     |  |
| E091-04 | MIN   |     |  |
| Yes     |   |     |  |
| E091-05 | SUM   |     |  |

|         |  |
|---------|--|
| Yes     |  |
| E091-06 | ALL quantifier                               |
| Pending |  |
| E091-07 | DISTINCT quantifier                          |
| Yes     |  |
|         |  |
|         |  |
| E101    | Basic data manipulation                      |
|         |  |
| E101-01 | INSERT statement                             |
| Yes     |  |
| E101-03 | Searched UPDATE statement                    |
| No      |  |
| E101-04 | Searched DELETE statement                    |
| No      |  |
|         |  |
|         |  |
| E111    | Single row SELECT statement                  |
| Yes     |  |
|         |  |
|         |  |
| E121    | Basic cursor support                         |
| No      |  |
| E121-01 | DECLARE CURSOR                               |
| No      |  |
| E121-02 | ORDER BY columns need Not be in select list  |
| Yes     |  |
| E121-03 | Value expressions in ORDER BY clause         |
| Yes     |  |
| E121-04 | OPEN statement                               |
| No      |  |
| E121-06 | Positioned UPDATE statement                  |
| No      |  |
| E121-07 | Positioned DELETE statement                  |
| No      |  |
| E121-08 | CLOSE statement                              |
| No      |  |
| E121-10 | FETCH statement: implicit NEXT               |
| No      |  |
| E121-17 | WITH HOLD cursors                            |
| No      |  |
| E131    | Null value support (nulls in lieu of values) |
| Partial | See 'nulls' in compatibility Notes           |
|         |  |
|         |  |
| E141    | Basic integrity constraints                  |
| No      |  |
| E141-01 | NoT NULL constraints                         |
| No      |  |
| E141-02 | UNIQUE constraints of NoT NULL columns       |
| No      |  |

|         |  |    |
|---------|--|----|
| E141-03 | PRIMARY KEY constraints  |    |
| No      |  |    |
| E141-04 | Basic FOREIGN KEY constraint with the No ACTION default for both referential delete action and referential update action | No |
| E141-06 | CHECK constraints  |    |
| No      |  |    |
| E141-07 | Column defaults  |    |
| No      |  |    |
| E141-08 | NoT NULL inferred on PRIMARY KEY   |    |
| No      |  |    |
| E141-10 | Names in a foreign key can be specified in any order   |    |
| No      |  |    |
|         |  |    |
| E151    | Transaction support  |    |
| No      |  |    |
| E151-01 | COMMIT statement   |    |
| No      |  |    |
| E151-02 | ROLLBACK statement   |    |
| No      |  |    |
| E152    | Basic SET TRANSACTION statement  |    |
| No      |  |    |
| E152-01 | SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause   |    |
| No      |  |    |
| E152-02 | SET TRANSACTION statement: READ ONLY and READ WRITE clauses  |    |
| No      |  |    |
|         |  |    |
| E*      | Other  |    |
|         |  |    |
| E153    | Updatable queries with subqueries  |    |
| No      |  |    |
| E161    | SQL comments using leading double minus  |    |
| Pending |  |    |
| E171    | SQLSTATE support   |    |
| No      |  |    |
| E182    | Host language binding (previously "Module Language")   |    |
| Yes     | Called from q, can call q  |    |
|         |  |    |
| F021    | Basic information schema   |    |
|         |  |    |
| F021-01 | COLUMNS view   |    |
| Pending |  |    |
| F021-02 | TABLES view  |    |
| Yes     |  |    |
| F021-03 | VIEWS view   |    |
| Pending |  |    |
| F021-04 | TABLE_CONSTRAINTS view   |    |
| No      |  |    |

|         |  |     |
|---------|--|-----|
| F021-05 | REFERENTIAL_CONSTRAINTS view   |     |
| No      |  |     |
| F021-06 | CHECK_CONSTRAINTS view   |     |
| No      |  |     |
|         |  |     |
| F031    | Basic schema manipulation  |     |
|         |  |     |
| F031-01 | CREATE TABLE statement to create persistent base tables  |     |
| Partial | No persistence   |     |
| F031-02 | CREATE VIEW statement  |     |
| No      |  |     |
| F031-03 | GRANT statement  |     |
| No      |  |     |
| F031-04 | ALTER TABLE statement: ADD COLUMN clause   |     |
| No      |  |     |
| F031-13 | DROP TABLE statement: RESTRICT clause  |     |
| Partial | No restrict  |     |
| F031-16 | DROP VIEW statement: RESTRICT clause   |     |
| No      |  |     |
| F031-19 | REVOKE statement: RESTRICT clause  |     |
| No      |  |     |
|         |  |     |
| F041    | Basic joined table   |     |
|         |  |     |
| F041-01 | Inner join (but Not necessarily the INNER keyword)   |     |
| Yes     |  |     |
| F041-02 | INNER keyword  |     |
| Yes     |  |     |
| F041-03 | LEFT OUTER JOIN  |     |
| Yes     |  |     |
| F041-04 | RIGHT OUTER JOIN   |     |
| Yes     |  |     |
| F041-05 | Outer joins can be nested  |     |
| Yes     |  |     |
| F041-07 | The inner table in a left or right outer join can also be used in an inner join                    | Yes |
|         |  |     |
| F041-08 | All comparison operators are supported (rather than just =)  |     |
| No      |  |     |
|         |  |     |
|         |  |     |
| F051    | Basic date and time  |     |
|         |  |     |
| F051-01 | DATE data type (including support of DATE literal)   |     |
| Yes     |  |     |
| F051-02 | TIME data type (including support of TIME literal) with fractional seconds precision of at least 0 | Yes |
|         |  |     |
| F051-03 | TIMESTAMP data type (including support of TIMESTAMP literal) with                                  |     |

|  |   |     |  |
|--|---|-----|--|
| fractional seconds precision of at least 0 and 6 |   | Yes |  |
|  |   |     |  |
| F051-04  | Comparison predicate on DATE, TIME, and TIMESTAMP data types                |     |  |
| Yes  |   |     |  |
| F051-05  | Explicit CAST between datetime types and character string types             |     |  |
| Yes  |   |     |  |
| F051-06  | CURRENT_DATE  |     |  |
| Yes  |   |     |  |
| F051-07  | LOCALTIME   |     |  |
| Yes  |   |     |  |
| F051-08  | LOCALTIMESTAMP  |     |  |
| Yes  |   |     |  |
| F081   | UNION and EXCEPT in views   |     |  |
| No   |   |     |  |
|  |   |     |  |
|  |   |     |  |
| F131   | Grouped operations  |     |  |
| No   |   |     |  |
| F131-01  | WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views |     |  |
|  | No  |     |  |
| F131-02  | Multiple tables supported in queries with grouped views                     |     |  |
| No   |   |     |  |
| F131-03  | Set functions supported in queries with grouped views                       |     |  |
| No   |   |     |  |
| F131-04  | Subqueries with GROUP BY and HAVING clauses and grouped views               |     |  |
| No   |   |     |  |
| F131-05  | Single row SELECT with GROUP BY and HAVING clauses and grouped views        |     |  |
| No   |   |     |  |
|  |   |     |  |
|  |   |     |  |
| F*   | Other   |     |  |
|  |   |     |  |
| F181   | Multiple module support   |     |  |
| No   |   |     |  |
| F201   | CAST function   |     |  |
| Yes  |   |     |  |
| F221   | Explicit defaults   |     |  |
| No   |   |     |  |
|  |   |     |  |
|  |   |     |  |
| F261   | CASE expression   |     |  |
|  |   |     |  |
| F261-01  | Simple CASE   |     |  |
| Yes  |   |     |  |
| F261-02  | Searched CASE   |     |  |
| Yes  |   |     |  |
| F261-03  | NULLIF  |     |  |
| Yes  |   |     |  |
| F261-04  | COALESCE  |     |  |
| Yes  |   |     |  |

|         |  |  |
|---------|--|--|
|         |  |  |
|         |  |  |
| F311    | Schema definition statement                                      |  |
| No      |  |  |
| F311-01 | CREATE SCHEMA  |  |
| No      |  |  |
| F311-02 | CREATE TABLE for persistent base tables                          |  |
| No      |  |  |
| F311-03 | CREATE VIEW  |  |
| No      |  |  |
| F311-04 | CREATE VIEW: WITH CHECK OPTION                                   |  |
| No      |  |  |
| F311-05 | GRANT statement  |  |
| No      |  |  |
| F471    | Scalar subquery values   |  |
| Yes     |  |  |
| F481    | Expanded NULL predicate  |  |
| No      |  |  |
|         |  |  |
|         |  |  |
| F501    | Features and conformance views                                   |  |
| No      |  |  |
| F501-01 | SQL_FEATURES view  |  |
| No      |  |  |
| F501-02 | SQL_SIZING view  |  |
| No      |  |  |
| F501-03 | SQL_LANGUAGES view   |  |
| No      |  |  |
| F812    | Basic flagging   |  |
| No      |  |  |
| S011    | Distinct data types  |  |
| No      |  |  |
| S011-01 | USER_DEFINED_TYPES view  |  |
| No      |  |  |
|         |  |  |
|         |  |  |
| T321    | Basic SQL-invoked routines                                       |  |
|         |  |  |
| T321-01 | User-defined functions with No overloading                       |  |
| Yes     | q fns can be converted to SQL fns with `s.fs` and added to `s.F` |  |
| T321-02 | User-defined stored procedures with No overloading               |  |
| No      |  |  |
| T321-03 | Function invocation  |  |
| Yes     |  |  |
| T321-04 | CALL statement   |  |
| No      |  |  |
| T321-05 | RETURN statement   |  |
| No      |  |  |
| T321-06 | ROUTINES view  |  |
| No      |  |  |
| T321-07 | PARAMETERS view  |  |

|      |                                    |
|------|------------------------------------|
| No   |                                    |
| T631 | IN predicate with one list element |
| Yes  |                                    |

**\*Legend:**

Yes = fully supported

Partial = supported with limitations

Pending = under development

No = not supported

## SQL in KDB-X Examples

---

*These examples show how to run SQL queries in KDB-X. They include query patterns, execution methods, parameter usage, integration with q, and error handling.*

### Run SQL

You can invoke SQL in several ways:

- `s)` prompt
- `.s.e` function
- `pgwire` interface

#### Use `s)`

Use the `s)` prompt to enter SQL interactively. Prefix your query with `s)` to run SQL instead of `q`.

```
q)t:([a: til 3) // create a table t with a column a
s)select a from t
```

#### Example

```
q)trips:([date:2#.z.D;city:`ldn`ny)
s)SELECT * FROM trips WHERE date=.z.D
```

#### Use `.s.e`

Use `.s.e` to run SQL in the KDB-X process command line. Prefix the SQL query with `.s.e` and wrap it in double quotes.

```
q)query:"select * from t"
q)result.s.e query
q)result
a
```

```
-
0
1
2
```

| Parameter | Description                             |
|-----------|---|
| result    | Result set from the query.              |
| query     | SQL statement wrapped in double quotes. |

This example demonstrates how to store the results of one SQL query in a variable and reuse it in another query within KDB-X:

```
// select two fields and save them in variable x
x:.s.e"SELECT vendor, distance FROM trips WHERE distance<20"

// reuse x in another SQL statement
.s.e"SELECT * FROM x"
```

Parameters

Use parameters to provide values to predefined queries.

Execute directly

Use `.s.sp` to inject q type parameters into SQL queries. Use `$n` notation in the query.

```
parsedquery:.s.sp[query](parameter list)
```

This example shows how to execute a parameterized SQL query in KDB-X by injecting q values into placeholders using `.s.sp`. Here, the query selects rows from table `t` where `s` matches either `AAPL` or `GOOG` and `p` is greater than `12.34`:

```
result:.s.sp["select a from t where s in $1 and p>$2"](`AAPL`GOOG;12.34)
```

Single parameter queries

Convert a single parameter into a list because `.s.sp` always expects a list.

```
.s.sp["SELECT vendor, fare FROM trips WHERE fare>$1"](enlist 50)
```

Prepare and execute

Use `.s.sq` to parse and prepare a parameterized query once. Then use `.s.sx` to execute it multiple times with different parameters.

```
parsedquery:.s.sq["select a from t where s in $1 and p>$2"](`;0n);
r1:.s.sx[parsedquery](`AAPL`GOOG;12.34)
r2:.s.sx[parsedquery](`MSFT`VOD;56.78)
```

## SQL parse tree

Use `.s.prx` to display the SQL parse tree.

```
.s.prx"select * from trade where date='2021.11.23' and symbol in ('XBTUSD')"
```

## Integrate with q

You can run q functions inside SQL statements or create SQL functions from q functions.

Use `qt()`

Use `qt()` to call a q function that returns a table. You can only call `qt()` in the `FROM` clause.

```
s)select a from qt(f,x,y...) where ...
```

This example shows how to query the result of a q function directly from SQL using `qt()`:

```
s)select a from qt('{gettable[`$x;"D"$y;z]}' , 'AAPL' , '2001.01.01' , 1)
```

Use `q()`

Use `q()` to call q functions anywhere in a statement and return any q data type.

```
s)q(t,f,x,y..)
```

This example demonstrates how to use the `q()` function within an SQL statement to invoke q code directly and return its result:

```
s)select q('J','test',10,col) from t
```

## Convert a q function into a SQL function

Call `.s.F` to wrap a q function as a SQL function.

```
.s.F[`functionname]:.s.fx{code}
```

This example shows how to expose a q function so it can be called directly from SQL queries in KDB-X:

```
.s.F[`fun]:.s.fx{x+1}  
s)select fun(x) from qt('([1 2 3]')
```

## Error handling

SQL in KDB-X raises errors when you reference columns that do not exist or use unsupported syntax.

```
SELECT non_existing_column FROM trades;  
-- Error: column not found
```