Introduction to Parquet in KDB-X

This page introduces how KDB-X integrates with the Parquet file format at a technical level. It covers the key concepts behind Parquet support, the architecture of the `pq` module, and current limitations.

Supported features

See the Apache Parquet documentation for details on the various features.

Data types

The following logical data types are supported:

Parquet type	q type
`string`	`char[]`
`binary`	`byte[]`
`bool`	`boolean`
`uuid`	`guid`
`int8`	`short`
`int16`	`short`
`int32`	`int`
`int64`	`long`
`uint8`	`byte`
`uint16`	`int`
`uint32`	`long`
`uint64`	`long`
`date`	`date`
`timestamp`	`timestamp`
`interval`	`timespan`
`duration`	`long`
`null`	`short`

Compression

The snappy, gzip, brotli, lz4 and zstd compression algorithms are supported.

API

To read Parquet files, first import the `pq` module:

```
q).pq:use`kx.pq
```

This will provide three functions:

- `pq[path]` maps the Parquet file from the path specified as a file symbol. It returns a virtual table with a virtual column `RG__` for the row group.
- `op[path]` opens a Parquet file, reads the metadata, and returns a data structure that is subject to change.
- `rd[data;row;col]` takes the structure returned by `op`, a row group index and a column index, and returns the corresponding column chunk.

Virtual tables

The virtual table module is a submodule of `pq`, therefore can be loaded with a multi-part module name:

```
q).pq.t:use`kx.pq.t
```

The virtual table API provides the following functions:

- `tt[tbl]`: wraps a kdb+ table in a virtual table. This allows combining Parquet and kdb+ tables as partitions in a virtual table.
- `mkP[map]`: creates a partitioned virtual table from a key table and a list of virtual tables.

Virtual tables support `select` and `exec` queries.

Examples

Example files for testing can be created using `pandas` in Python (requires `pyarrow` to be installed):

```
import pandas as pd
f=pd.DataFrame(data={'col0': [True,True,False,True,False,False,True,False,False],
    'col1': [1,2,3,4,5,6,7,8,9],
    'col2': ['asd','bsd','csd','asd','bsd','csd','asd','bsd','csd']})
f.to_parquet('types.parquet',compression='lz4',use_dictionary=True,data_page_versi
on='1.0')
```

To load the file in KDB-X:

```
q).pq:use`kx.pq
q)t:.pq.pq`:types.parquet
q)select from t
col0 col1 col2
---------------
```



```
1    1    "asd"
1    2    "bsd"
0    3    "csd"
1    4    "asd"
0    5    "bsd"
0    6    "csd"
1    7    "asd"
0    8    "bsd"
0    9    "csd"
q)meta t
c   | t f a
----| -----
col0| b
col1| j
col2| C
```

To demonstrate row group functionality, we can save a Parquet file with a forced row group size:

```
import pandas as pd
import pyarrow as pa
import pyarrow.parquet

f=pd.DataFrame(data={
    'c_bool': [True,True,False,True,False,False,True,False,False],
    'c_int8': pd.Series([1,2,3,4,5,6,7,8,9],dtype="int8"),
    })
pa.parquet.write_table(pa.Table.from_pandas(f), 'rowgrp.parquet',
row_group_size=3)
```

The row group index can be accessed using the hidden column `RG__`:

```
q).pq:use`kx.pq
q)t:.pq.pq`:rowgrp.parquet
q)select c_int8, RG__ from t
c_int8 RG__
-----------
1      0
2      0
3      0
4      1
5      1
6      1
7      2
8      2
9      2
```

The low-level functions `op` and `rd` can be used to extract a given column chunk given the row group and column index:


```
q)f:.pq.op`:rowgrp.parquet
q).pq.rd[f;0;0]
110b
q).pq.rd[f;1;0]
100b
q).pq.rd[f;2;0]
100b
q).pq.rd[f;0;1]
1 2 3h
q).pq.rd[f;1;1]
4 5 6h
q).pq.rd[f;2;1]
7 8 9h
```

The performance effect of row group pruning can be demonstrated by writing the same data with different row group sizes:

```
import pandas as pd
import pyarrow as pa
import pyarrow.parquet
ROWS=50000000
f=pd.DataFrame(data={'time': pd.date_range("2012-01-01 00:00", periods=ROWS,
freq=pd.Timedelta("00:00:01")),
    'sym': (ROWS//4)*['aaa','bbb','ccc','ddd'],
    'price': (ROWS//10)*[1,2,3,4,5,6,7,8,9,10]})
pa.parquet.write_table(pa.Table.from_pandas(f), 'rowgrp1.parquet',
row_group_size=1000)
pa.parquet.write_table(pa.Table.from_pandas(f), 'rowgrp2.parquet',
row_group_size=ROWS//10)
```

```
q).pq:use`kx.pq
q)\ts f1:.pq.pq`:rowgrp1.parquet
601 371102256
q)\ts f2:.pq.pq`:rowgrp2.parquet
2 88656
q)\ts select from f1 where time<2012.10.16
3273 1425051600
q)\ts select from f2 where time<2012.10.16
1962 1811943440
```

Filtering on row groups

Given the following Parquet file:

```
import pandas as pd
import pyarrow as pa
```



```
import pyarrow.parquet

f2=pd.DataFrame(data={
    'c_bool': [True,True,False,True,False,False,True,False,False],
    'c_int8': pd.Series([1,2,3,4,5,6,7,8,9],dtype="int8"),
    })
pa.parquet.write_table(pa.Table.from_pandas(f2), 'rowgrp.parquet',
row_group_size=3)
```

The table is split into three row groups. If there is a condition on a particular column, querying will consider the minimum and maximum value in each column chunk to determine whether it satisfies the condition. Any row groups that don't match the condition are not scanned.

```
q)select from .pq.pq`:rowgrp.parquet where c_int8<3
c_bool c_int8
-------------
1      1
1      2
0      3
```

The chunks of column `c_int8` are checked against the condition `<3`, and by checking the minimum and maximum values, it is determined that only row group 0 may have matching rows. Therefore the chunk for row groups 1 and 2 is not read for the column `c_bool`.

Partitioning across multiple files

The virtual table API allows combining multiple Parquet files into a single virtual table using virtual partition columns.

Given the following Parquet files:

```
import pandas as pd
import pyarrow as pa
import pyarrow.parquet

fd1=pd.DataFrame(data={
    'c_bool': [True,True,False,True],
    'c_int8': pd.Series([1,2,3,4],dtype="int8"),
    })
fd2=pd.DataFrame(data={
    'c_bool': [False,False,True,False,False],
    'c_int8': pd.Series([5,6,7,8,9],dtype="int8"),
    })
pa.parquet.write_table(pa.Table.from_pandas(fd1), 'part1.parquet',
row_group_size=3)
pa.parquet.write_table(pa.Table.from_pandas(fd2), 'part2.parquet',
row_group_size=3)
```


we can map the files as partitions in a virtual table, using one or more **virtual columns**:

```
q).pq.t:use`kx.pq.t
q)t:.pq.t.mkP([]date:2001.01.01 2001.01.02;a:1 -1)!.pq.pq
each`:part1.parquet`:part2.parquet
q)select from t
date       a  c_bool c_int8
--------------------------
2001.01.01 1  1      1
2001.01.01 1  1      2
2001.01.01 1  0      3
2001.01.01 1  1      4
2001.01.02 -1 0      5
2001.01.02 -1 0      6
2001.01.02 -1 1      7
2001.01.02 -1 0      8
2001.01.02 -1 0      9
```

Like with partitioned kdb+ tables, filtering on the virtual columns avoids reading the data for partitions that don't match the filtering condition:

```
q)select from t where a=1
date       a c_bool c_int8
-------------------------
2001.01.01 1 1      1
2001.01.01 1 1      2
2001.01.01 1 0      3
2001.01.01 1 1      4
```

Because the column `a` has the value of `-1` in the second partition, no data is loaded from the second file. Virtual tables also support map-reduce-style operations. These are equivalent to and subject to the same restrictions as for partitioned tables.

```
q)select avg c_int8 by date from t
date      | c_int8
----------| ------
2001.01.01| 2.5
2001.01.02| 7
```

Metadata

The built-in function `meta` returns the usual column information for virtual tables:

```
q)meta t
c     | t f a
```



```
------| -----
date  | d
a     | j
c_bool| b
c_int8| h
```

Performance considerations

- Large files with multiple row groups perform better than many small files.

Planned improvements

- Nested types, such as `interval` in pandas, or `MAP`, `STRUCT` and `UNION` in DuckDB.
- Support for nested (non-atomic) virtual columns such as strings.
- `hive` and `iceberg` layouts.
- Object storage support.

