eveni Proof of Concept

---

https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/sql/DataFrameReader.html#jdbc-java.lang.String-java.lang.String-java.lang.String-long-long-int-java.util.Properties- (https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/sql/DataFrameReader.html#jdbc-java.lang.String-java.lang.String-java.lang.String-long-long-int-java.util.Properties-)

```
public Dataset<Row> jdbc(String url,
                         String table,
                         String columnName,
                         long lowerBound,
                         long upperBound,
                         int numPartitions,
                         java.util.Properties connectionProperties)
```

The problem set `eveni` is aiming to help out with relates selecting a column and number of partitions for the `columnName` and `numPartitions` parameter in above by querying the source table via Python. Advising on most efficient `lowerBound` and `upperBound` values is a possible future goal.

Simply put, there may be several candidate columns within a given source table that might serve as a partitioning key but the effectiveness and efficiency of the partioning operations will be reliant on the actual distribution of values across that column.

imports

- could have used `pyodbc` only but wanted to check out `turbodbc`, use former for metadata, latter for data inserts and reads

In [1]:
```python
import collections
import time

import pandas as pd
import numpy as np

import turbodbc
import pyodbc

from typing import NamedTuple
from faker import Faker

import bokeh
from bokeh.io import output_notebook
import holoviews as hv
hv.extension('bokeh')

print('numpy version: ', np.__version__) # 1.14.5
print('pandas version: ', pd.__version__) # 0.23.3
print('bokeh version: ', bokeh.__version__) # 0.13.0 = June, 2018
print('holoviews version: ', hv.__version__) # 1.10.7 = July, 2018
```

```
numpy version:  1.14.5
pandas version:  0.23.3
bokeh version:  0.13.0
holoviews version:  1.10.7
```

create/populate db metadata objects

- inital testing with:
  - `Vertica 8.1`
  - `Microsoft SQL Server 2016`
  - `PostgreSQL 10.4`
- each of the ODBC DSN below have been preconfigured, on a Windows 10 machine
- the schema and table name used throughout below = **eveni.tbl1**

In [2]:
```
TARGET_SCHEMA_NAME='eveni'
TARGET_TABLE_NAME='tbl1'

class DB(NamedTuple):
    name: str
    dsn: str
    dt_type: str = 'DATETIME'

db_vert = DB(name='vertica', dsn='vert_localhost')
db_mssql = DB(name='mssql', dsn='P71_mssql2016')
db_pg = DB(name='postgres', dsn='pg_localhost', dt_type='TIMESTAMP')


dbs = {}
dbs['vert'] = db_vert
dbs['mssql'] = db_mssql
dbs['pg'] = db_pg

dbs
```

Out[2]:
```
{'vert': DB(name='vertica', dsn='vert_localhost', dt_type='DATETIME'),
 'mssql': DB(name='mssql', dsn='P71_mssql2016', dt_type='DATETIME'),
 'pg': DB(name='postgres', dsn='pg_localhost', dt_type='TIMESTAMP')}
```

populate dataframe with Faker data, to serve as test data

- later step will be to insert as rows into each of the target dbs
- include three `int` columns with predefined distributions of values (uniform, normal, wald)

In [3]:
```
ROW_COUNT = 10000

fake = Faker()
fake.seed(55)
np.random.seed(55)

faker_columns = [
    'company',
    'bban',
    'tx_date',
    'updated_date',
    'int_uni',
    'int_normal',
    'float_normal',
    'int_wald',
]

rows = []
for _ in range(ROW_COUNT):
    fake.random
    row = [
        fake.company(),
        fake.bban(), # basic bank account number
        fake.date_time_this_decade(before_now=True, after_now=False, tzinfo=None),
        fake.date_time_this_year(before_now=True, after_now=False, tzinfo=None),
        np.random.randint(1,5000, dtype=np.int32),
        int(np.random.normal(loc=2500, scale=500)),
        np.random.normal(loc=2500, scale=1000),
        int(np.random.wald(mean=1000, scale=3000))
    ]
    rows.append(row)

df_faker = pd.DataFrame(data=rows, columns=faker_columns)

print(df_faker.shape)
df_faker.head()
```

```
(10000, 8)
```

Out[3]:

	company	bban	tx_date	updated_date	int_uni	int_normal	float_normal	int_wald
0	Travis and Sons	EXJC2415786686568	2010-09-10 01:49:47	2018-09-03 15:30:03	4558	2609	4273.298166	1803
1	Williams-Hall	WASO3063186333624	2015-08-15 08:55:07	2018-01-04 05:30:16	607	2153	2497.710137	821
2	Petersen Group	JKLT3372234827342	2017-01-23 10:16:00	2018-07-02 09:44:27	2092	1772	2860.859535	411
3	Morrison, Rodriguez and Roth	CIBW4454219677717	2014-06-18 01:43:32	2018-08-02 01:53:54	4589	2033	3801.351539	1633
4	Martinez-Walker	RNPS2403565580053	2018-04-26 23:51:01	2018-05-08 04:09:31	1598	2686	2138.588787	516

DROP and re-create the target table = eveni.tbl1, INSERT the Faker data

- using `turbobdc` here, mostly to play around with it
- even with this small amount of data performance is significantly better than `pyodbc`, though as the library maintainers point out the free postgresql ODBC is slow and there is only so much that can be done to speed it up

In [4]:
```
SQL_TBL1_CREATE = """
CREATE TABLE {schema_name}.{table_name} (
    company varchar(255),
    bban varchar(30),
    tx_date {dt_type},
    updated_date {dt_type},
    int_uni INT,
    int_normal INT,
    float_normal FLOAT,
    int_wald INT
)
"""

SQL_TBL1_DROP = """
    DROP TABLE IF EXISTS {schema_name}.{table_name}
"""

SQL_INSERT = """
INSERT INTO {schema_name}.{table_name}
VALUES (
    ?, ?, ?, ?,
    ?, ?, ?, ?
)
"""

DROP_EXISTING_TABLE=True

for db in dbs.values():
    with turbodbc.connect(dsn=db.dsn) as conn:
        start = time.time()
        cur = conn.cursor()
        drop_existing = True
        if DROP_EXISTING_TABLE:
            print('DROPng in {}'.format(db.name))
            cur.execute(SQL_TBL1_DROP.format(schema_name=TARGET_SCHEMA_NAME, table_name=TARGET_TABLE_NAME))
        cur.execute(SQL_TBL1_CREATE.format(schema_name=TARGET_SCHEMA_NAME, table_name=TARGET_TABLE_NAME, dt_type=db.dt_
type))
        conn.commit()

        print('INSERT to {}, {} rows'.format(db.name, df_faker.shape[0]))
        cur.executemany(SQL_INSERT.format(schema_name=TARGET_SCHEMA_NAME,
                                          table_name=TARGET_TABLE_NAME),
                        df_faker.astype(str).values.tolist())
        conn.commit()
        print('{} took {:.4f} seconds'.format(db.name, time.time() - start))
        print('-'*77)
```

```
DROPng in vertica
INSERT to vertica, 10000 rows
vertica took 0.3780 seconds
-----------------------------------------------------------------------------
DROPng in mssql
INSERT to mssql, 10000 rows
mssql took 0.5036 seconds
-----------------------------------------------------------------------------
DROPng in postgres
INSERT to postgres, 10000 rows
postgres took 14.3784 seconds
-----------------------------------------------------------------------------
```

pull the test data back out again, from each db into one master dataframe

- of course in a real-world scenario only one of these db systems would be targeted but continue with all three for cross-db testing purposes

In [5]:
```python
SQL_SELECT = 'SELECT * FROM {schema_name}.{table_name}'

df_out = pd.DataFrame()
db_rows = []
for db in dbs.values():
    with turbodbc.connect(dsn=db.dsn) as conn:
        cur = conn.cursor()
        df_db = pd.read_sql(SQL_SELECT.format(schema_name=TARGET_SCHEMA_NAME, table_name=TARGET_TABLE_NAME), conn)
        df_db = pd.concat([df_db], keys=[db.name], names=['db'])
        db_rows.append(df_db)

df_out = pd.concat(db_rows)
print('rows for each source database:')
print(df_out.index.get_level_values('db').value_counts())
print('\npeek at the round-tripped data')
df_out.head()
```

```
rows for each source database:
postgres    10000
vertica     10000
mssql       10000
Name: db, dtype: int64

peek at the round-tripped data
```

Out[5]:

		company	bban	tx_date	updated_date	int_uni	int_normal	float_normal	int_wald
db									
vertica	**0**	Abbott Ltd	ZLOX7369311506540	2012-11-03 04:33:58	2018-02-23 15:23:15	4199	2388	4180.308311	946
	1	Abbott PLC	BWVE0123462705448	2012-04-03 19:56:03	2018-07-23 18:34:59	689	2396	3644.904434	1519
	2	Abbott PLC	OCJF6874397666635	2012-03-20 12:46:12	2018-09-02 08:26:18	1223	2743	2450.941803	469
	3	Abbott, Mckay and Orozco	TJXW0748945688912	2017-09-13 09:20:36	2018-04-05 01:50:28	977	3373	4210.720389	522
	4	Abbott, Meyer and Reeves	ZQZV0305326524875	2010-01-28 19:04:12	2018-06-06 01:18:48	4165	2887	3798.249319	647

get candidate columns (INT only for first pass)

- inspect `eveni.tbl1` in each of the source systems and pull out only those that are some form of `INTEGER` datatype
- Spark documentation refers to using a column of "integral type" but the final SQL queries indicate any numeric type would work
- at the least an integer value could be parsed from a different datatype, e.g. pull the minute value out of a timestamp
- either way, for POC only going to consider those columns that are truly of a flavor = `INTEGER`

In [6]:
```
import enum
class SqlCategory(enum.Enum):
    numerical = 0
    floaty = 1
    inty = 2
    datish = 3
    datetimey = 4

schema_name = 'eveni'
table_name = 'tbl1'

cands = []
for db in dbs.values():
    with pyodbc.connect(dsn=db.dsn) as conn:
        # print('db: {}'.format(db))
        cur = conn.cursor()
        for row in cur.columns(schema=schema_name, table=table_name):
            # run below to get guts of row,e.g. ('table_cat', <class 'str'>, None, 128, 128, 0, True)
            # print(row.cursor_description)
            row_info = 'column_name: {} sql_data_type: {}'.format(
                row.column_name,
                row.sql_data_type
            )
            #print(row_info)


            # TODO: look at pyodbc/src/getdata.cpp to help finish this off
            # group into: numeric / floaty / inty / date / time etc.
            category = None
            if row.sql_data_type in (-6,-5,4,5):
                 category = SqlCategory.inty
    #         if row.sql_data_type in (2,3):
    #             category = SqlCategory.numerical
    #         elif row.sql_data_type in (6,7):
    #             cateogry = SqlCategory.floaty
    #         elif row.sql_data_type in (sql_data_type):
    #             category = SqlCategory.datish

            if category:
                cand = {
                    'db_name': db.name,
                    'schema_name': schema_name,
                    'table_name': table_name,
                    'column_name': row.column_name,
                    'category_name': category.name,
                    'category_value': category.value,
                    'native_type_name': row.type_name,
```

```
                'native_sql_data_type': row.sql_data_type
            }
            cands.append(cand)

print('candidate columns in all of the source databsaes')
df_cands = pd.DataFrame.from_records(cands).set_index(keys='db_name', append=True).swaplevel()
df_cands
```

```
candidate columns in all of the source databsaes
```

Out[6]:

		category_name	category_value	column_name	native_sql_data_type	native_type_name	schema_name	table_name
db_name								
vertica	**0**	inty	2	int_uni	-5	Integer	eveni	tbl1
	1	inty	2	int_normal	-5	Integer	eveni	tbl1
	2	inty	2	int_wald	-5	Integer	eveni	tbl1
mssql	**3**	inty	2	int_uni	4	int	eveni	tbl1
	4	inty	2	int_normal	4	int	eveni	tbl1
	5	inty	2	int_wald	4	int	eveni	tbl1
postgres	**6**	inty	2	int_uni	4	int4	eveni	tbl1
	7	inty	2	int_normal	4	int4	eveni	tbl1
	8	inty	2	int_wald	4	int4	eveni	tbl1

FUTURE: consider parsability

- to show how non-numerical columns could be used as partitioning columns, demonstrate pulling minute values out of `DATETIME` fields
- putting aside any performance concerns, something similar could performed on almost any datatype assuming structure of source data can be guaranteed , e.g. use substring or regex functions to extract numeric values from `VARCHAR` data

In [7]:

```
for db in dbs.values():
    with turbodbc.connect(dsn=db.dsn) as conn:
        cur = conn.cursor()
        if db.name == 'mssql':
            SQL_TO_MINUTE = "SELECT TOP 10 DATEPART(MINUTE,updated_date) AS updatedMinute FROM eveni.tbl1"
        else:
            SQL_TO_MINUTE = "SELECT DATE_PART('minute', updated_date) as updatedMinute FROM eveni.tbl1 LIMIT 10"
        df_minutes = pd.read_sql(SQL_TO_MINUTE.format(schema_name=TARGET_SCHEMA_NAME, table_name=TARGET_TABLE_NAME), co
nn)
        print('\n10 parsed minute values from {}:'.format(db.name))
        print(df_minutes)
```

```
10 parsed minute values from vertica:
   updatedMinute
0             23
1             34
2             26
3             50
4             18
5             49
6              5
7             40
8             19
9             20

10 parsed minute values from mssql:
   updatedMinute
0             30
1             30
2             44
3             53
4              9
5              1
6             53
7             15
8             26
9             28

10 parsed minute values from postgres:
   updatedminute
0           30.0
1           30.0
2           44.0
3           53.0
4            9.0
5            1.0
6           53.0
7           15.0
8           26.0
9           28.0
```

define the function to make the partition SQL

- an earlier version in Spark, I think < 2.0, did not include the lower `IS NULL` condition, but it is there now
- either way, fewer NULLs (and/or non-nullable column to begin) will result in less data skew in that first partition
- end off with a quick test of `get_filters` to confirm results for a given set of args are as expected

In [8]:

```python
# https://github.com/apache/spark/blob/47d84e4d0e56e14f9402770dceaf0b4302c00e98/sql/core/src/main/scala/org/apache/spar
k/sql/execution/datasources/jdbc/JDBCRelation.scala

def get_filters(column_name, lower_bound, upper_bound, num_partitions):
    stride = int(upper_bound/num_partitions) - int(lower_bound/num_partitions)

    i = 0
    curr_value = lower_bound
    where_clauses = []
    while i < num_partitions:
        lbound_value = str(curr_value)
        lbound = '{} >= {}'.format(column_name, lbound_value) if i != 0 else None
        curr_value += stride
        ubound_value = str(curr_value)
        ubound = '{} < {}'.format(column_name, ubound_value) if i != (num_partitions - 1) else None
        if not ubound:
            where_clause = lbound
        elif not lbound:
            where_clause = '{} OR {} IS NULL'.format(ubound, column_name)
        else:
            where_clause = '{} AND {}'.format(lbound, ubound)
        where_clauses.append(where_clause)
        i = i + 1

    return where_clauses


def test_get_filters():
    column_name = 'some_column'
    lower_bound = 0
    upper_bound = 100
    num_partitions = 10
    filters = get_filters(column_name, lower_bound, upper_bound, num_partitions)
    for f in filters:
        print(f)
    expected = [
        'some_column < 10 OR some_column IS NULL',
        'some_column >= 10 AND some_column < 20',
        'some_column >= 20 AND some_column < 30',
        'some_column >= 30 AND some_column < 40',
        'some_column >= 40 AND some_column < 50',
        'some_column >= 50 AND some_column < 60',
        'some_column >= 60 AND some_column < 70',
        'some_column >= 70 AND some_column < 80',
        'some_column >= 80 AND some_column < 90',
        'some_column >= 90'
    ]
```

```
    assert(filters == expected)

print('test (and print) results of get_filters():\n')
test_get_filters()
```

```
test (and print) results of get_filters():

some_column < 10 OR some_column IS NULL
some_column >= 10 AND some_column < 20
some_column >= 20 AND some_column < 30
some_column >= 30 AND some_column < 40
some_column >= 40 AND some_column < 50
some_column >= 50 AND some_column < 60
some_column >= 60 AND some_column < 70
some_column >= 70 AND some_column < 80
some_column >= 80 AND some_column < 90
some_column >= 90
```

assemble the (Vertica) df to hold the relevant partitioning queries

- in order to better emulate the expected use cases for `eveni`, makes sense to leave off cross-db testing and narrow things down to only one, going to proceed with Vertica
- optimal number of partitions may depend on a outside factors like size of Spark cluster, resources of the source db, file format of persisted data, etc.
    - choose an arbitrary range of numPartitions values: `10,20,50,100`
- for each of the `INTEGER` candidate columns, determine min/max values and feed those + each numPartitions into `get_filters()`
- wind up with a dataframe holding a series of "predicate suffixes", one row for each partitionColumn/numPartitions combo

In [9]:
```
SQL_MIN_MAX = """
SELECT MIN({column_name}), MAX({column_name})
FROM {schema_name}.{table_name}
"""

# predefined list of multiple numbers-of-partitions
num_partitions_cands = [10,20,50,100]

vert_meta = {}
with pyodbc.connect(dsn=dbs['vert'].dsn) as conn:
    cur = conn.cursor()
    for row in df_cands.loc['vertica'].iterrows():
        # row is a tuple of index, Series objects (index itself is tuple if multindex)
        #print(row[1].column_name)
        idx, srs = row
        min_max_query = SQL_MIN_MAX.format(schema_name=srs.schema_name,
                                           table_name=srs.table_name,
                                           column_name = srs.column_name)
        cur.execute(min_max_query)
        value_min, value_max = cur.fetchone()

        for num_partitions in num_partitions_cands:
            filters = get_filters(srs.column_name, value_min, value_max, num_partitions)

            d = {
                'value_min': value_min,
                'value_max': value_max,
                'partitioning_queries': filters,
            }
            vert_meta[(srs.schema_name, srs.table_name, srs.column_name, num_partitions)] = d

df_vert_meta = pd.DataFrame.from_dict(vert_meta, orient='index')
df_vert_meta.index = df_vert_meta.index.rename(names=['schema','table','column','num_partitions'])

df_vert_meta
```

Out[9]:

				value_min	value_max	partitioning_queries
schema	**table**	**column**	**num_partitions**			
eveni	**tbl1**	**int_normal**	**10**	802	4211	[int_normal < 1143 OR int_normal IS NULL, int_...
			20	802	4211	[int_normal < 972 OR int_normal IS NULL, int_n...
			50	802	4211	[int_normal < 870 OR int_normal IS NULL, int_n...
			100	802	4211	[int_normal < 836 OR int_normal IS NULL, int_n...
		int_uni	**10**	1	4999	[int_uni < 500 OR int_uni IS NULL, int_uni >= ...
			20	1	4999	[int_uni < 250 OR int_uni IS NULL, int_uni >= ...
			50	1	4999	[int_uni < 100 OR int_uni IS NULL, int_uni >= ...
			100	1	4999	[int_uni < 50 OR int_uni IS NULL, int_uni >= 5...
		int_wald	**10**	104	4825	[int_wald < 576 OR int_wald IS NULL, int_wald ...
			20	104	4825	[int_wald < 340 OR int_wald IS NULL, int_wald ...
			50	104	4825	[int_wald < 198 OR int_wald IS NULL, int_wald ...
			100	104	4825	[int_wald < 151 OR int_wald IS NULL, int_wald ...

run the filter queries, get back bin counts

- this is the meat of it, where we simulate an actual Spark run and get back the number of rows that would be in each partition
    - the simulation part is that we are doing simple `COUNT(*)` instead of retreiving actual data
- for illustrative purposes print out the query used to find the number of rows for the first partition of each `numPartitions` value (`10,20,50,100`) for `int_normal` column
- storing results in a new column of same df... next cell reconfigures into more digestable format but would be better to handle that here, good enough for now

In [10]:

```python
SQL_BIN_COUNT = """
SELECT COUNT(*)
FROM {schema_name}.{table_name}
WHERE {filter_sql}
"""

def run_queries(queries, conn):
    cur = conn.cursor()
    bin_counts = []
    for i, query in enumerate(queries):
        full_query = SQL_BIN_COUNT.format(schema_name=schema_name,
                                          table_name=table_name,
                                          filter_sql=query)
        cur.execute(full_query)
        cnt = cur.fetchone()[0]
        bin_counts.append(cnt)
        if i == 0 and 'int_normal' in query:
            print('>> SQL query: {}'.format(full_query))
            print('>> results: {}'.format(cnt))

    return bin_counts

with pyodbc.connect(dsn=dbs['vert'].dsn) as conn:
    df_vert_meta['partition_counts'] = df_vert_meta['partitioning_queries'].apply(run_queries, conn=conn)

df_vert_meta
```

```
>> SQL query:
SELECT COUNT(*)
FROM eveni.tbl1
WHERE int_normal < 1143 OR int_normal IS NULL

>> results: 24
>> SQL query:
SELECT COUNT(*)
FROM eveni.tbl1
WHERE int_normal < 972 OR int_normal IS NULL

>> results: 7
>> SQL query:
SELECT COUNT(*)
FROM eveni.tbl1
WHERE int_normal < 870 OR int_normal IS NULL

>> results: 3
>> SQL query:
SELECT COUNT(*)
FROM eveni.tbl1
WHERE int_normal < 836 OR int_normal IS NULL

>> results: 1
```

Out[10]:

				value_min	value_max	partitioning_queries	partition_counts
schema	**table**	**column**	**num_partitions**				
eveni	**tbl1**	**int_normal**	**10**	802	4211	[int_normal < 1143 OR int_normal IS NULL, int_...	[24, 161, 654, 1639, 2463, 2574, 1655, 656, 15...
			20	802	4211	[int_normal < 972 OR int_normal IS NULL, int_n...	[7, 17, 51, 107, 212, 437, 666, 957, 1137, 131...
			50	802	4211	[int_normal < 870 OR int_normal IS NULL, int_n...	[3, 2, 2, 8, 9, 15, 20, 29, 43, 51, 65, 95, 12...
			100	802	4211	[int_normal < 836 OR int_normal IS NULL, int_n...	[1, 2, 0, 2, 2, 0, 4, 4, 2, 7, 3, 12, 13, 7, 1...
		int_uni	**10**	1	4999	[int_uni < 500 OR int_uni IS NULL, int_uni >= ...	[1016, 963, 981, 994, 978, 958, 971, 1063, 104...
			20	1	4999	[int_uni < 250 OR int_uni IS NULL, int_uni >= ...	[490, 523, 483, 475, 489, 498, 516, 471, 490, ...
			50	1	4999	[int_uni < 100 OR int_uni IS NULL, int_uni >= ...	[203, 198, 182, 206, 220, 190, 179, 197, 196, ...
			100	1	4999	[int_uni < 50 OR int_uni IS NULL, int_uni >= 5...	[101, 102, 105, 86, 90, 95, 94, 105, 108, 115,...
		int_wald	**10**	104	4825	[int_wald < 576 OR int_wald IS NULL, int_wald ...	[2372, 4047, 2042, 896, 383, 149, 70, 23, 15, 3]
			20	104	4825	[int_wald < 340 OR int_wald IS NULL, int_wald ...	[407, 1965, 2243, 1804, 1225, 817, 547, 349, 2...
			50	104	4825	[int_wald < 198 OR int_wald IS NULL, int_wald ...	[16, 159, 488, 785, 903, 920, 879, 838, 736, 6...
			100	104	4825	[int_wald < 151 OR int_wald IS NULL, int_wald ...	[3, 13, 58, 101, 221, 267, 354, 431, 451, 452,...

collapse the two columns (list of queries, list of counts) into tidy version

- **TODO:** better solution when recording counts in the first place

In [11]:
```
df_q = df_vert_meta.apply(lambda x: pd.Series(x['partitioning_queries']),axis=1).stack().rename('queries')
df_c = df_vert_meta.apply(lambda x: pd.Series(x['partition_counts']),axis=1).stack().rename('counts')

df_qc = pd.concat([df_q, df_c], axis=1)
print(df_qc.loc[('eveni','tbl1')].head())
print(df_qc.loc[('eveni','tbl1')].tail())
```

```
                                                           queries  counts
column     num_partitions
int_normal 10             0   int_normal < 1143 OR int_normal IS NULL    24.0
                          1  int_normal >= 1143 AND int_normal < 1484   161.0
                          2  int_normal >= 1484 AND int_normal < 1825   654.0
                          3  int_normal >= 1825 AND int_normal < 2166  1639.0
                          4  int_normal >= 2166 AND int_normal < 2507  2463.0
                                                      queries  counts
column   num_partitions
int_wald 100            95  int_wald >= 4569 AND int_wald < 4616     0.0
                        96  int_wald >= 4616 AND int_wald < 4663     0.0
                        97  int_wald >= 4663 AND int_wald < 4710     1.0
                        98  int_wald >= 4710 AND int_wald < 4757     0.0
                        99                    int_wald >= 4757     2.0
```

calculate standard deviations on a per-numPartitions basis

- the bin-count results could be displayed as-is but want to add some "advisory" aspect before doing so
- first step is to calc standard deviations on the bin-counts
    - for a given `numPartitions` lower is better, indicating a more even distribution of counts across bins
    - at num_partitions=10, column holding values derived from uniform distribution has lowest standard deviation, as we would expect

In [31]:

```
# add standard deviation for number of rows in each column/pnum_partitions group
df_qc['partitions_std'] = df_qc.groupby(level=['schema','table','column','num_partitions'])['counts'].transform('std')
print('print all rows for each column where num_partitions=10')
df_qc.xs(10, level=3, drop_level=False)
```

print all rows for each column where num_partitions=10

Out[31]:

					queries	counts	partitions_std	column_avg_std
schema	**table**	**column**	**num_partitions**					
eveni	**tbl1**	**int_normal**	**10**	**0**	int_normal < 1143 OR int_normal IS NULL	24.0	1001.450393	219.331685
				1	int_normal >= 1143 AND int_normal < 1484	161.0	1001.450393	219.331685
				2	int_normal >= 1484 AND int_normal < 1825	654.0	1001.450393	219.331685
				3	int_normal >= 1825 AND int_normal < 2166	1639.0	1001.450393	219.331685
				4	int_normal >= 2166 AND int_normal < 2507	2463.0	1001.450393	219.331685
				5	int_normal >= 2507 AND int_normal < 2848	2574.0	1001.450393	219.331685
				6	int_normal >= 2848 AND int_normal < 3189	1655.0	1001.450393	219.331685
				7	int_normal >= 3189 AND int_normal < 3530	656.0	1001.450393	219.331685
				8	int_normal >= 3530 AND int_normal < 3871	155.0	1001.450393	219.331685
				9	int_normal >= 3871	19.0	1001.450393	219.331685
		int_uni	**10**	**0**	int_uni < 500 OR int_uni IS NULL	1016.0	36.493531	20.934106
				1	int_uni >= 500 AND int_uni < 999	963.0	36.493531	20.934106
				2	int_uni >= 999 AND int_uni < 1498	981.0	36.493531	20.934106
				3	int_uni >= 1498 AND int_uni < 1997	994.0	36.493531	20.934106
				4	int_uni >= 1997 AND int_uni < 2496	978.0	36.493531	20.934106
				5	int_uni >= 2496 AND int_uni < 2995	958.0	36.493531	20.934106
				6	int_uni >= 2995 AND int_uni < 3494	971.0	36.493531	20.934106
				7	int_uni >= 3494 AND int_uni < 3993	1063.0	36.493531	20.934106
				8	int_uni >= 3993 AND int_uni < 4492	1041.0	36.493531	20.934106
				9	int_uni >= 4492	1035.0	36.493531	20.934106
		int_wald	**10**	**0**	int_wald < 576 OR int_wald IS NULL	2372.0	1380.488078	317.890889
				1	int_wald >= 576 AND int_wald < 1048	4047.0	1380.488078	317.890889
				2	int_wald >= 1048 AND int_wald < 1520	2042.0	1380.488078	317.890889
				3	int_wald >= 1520 AND int_wald < 1992	896.0	1380.488078	317.890889
				4	int_wald >= 1992 AND int_wald < 2464	383.0	1380.488078	317.890889

					queries	counts	partitions_std	column_avg_std
schema	**table**	**column**	**num_partitions**					
				5	int_wald >= 2464 AND int_wald < 2936	149.0	1380.488078	317.890889
				6	int_wald >= 2936 AND int_wald < 3408	70.0	1380.488078	317.890889
				7	int_wald >= 3408 AND int_wald < 3880	23.0	1380.488078	317.890889
				8	int_wald >= 3880 AND int_wald < 4352	15.0	1380.488078	317.890889
				9	int_wald >= 4352	3.0	1380.488078	317.890889

similar to above but numPartitions=100, print first bin of each only

- as expected `int_uni` has lowest standard deviation at this `numPartitions`, though in absolute terms all sd are lower with a higher number of bins

In [13]:
```
df_qc.xs(100, level=3, drop_level=False).groupby('column').first()
```

Out[13]:

	queries	counts	partitions_std
column			
int_normal	int_normal < 836 OR int_normal IS NULL	1.0	97.677887
int_uni	int_uni < 50 OR int_uni IS NULL	101.0	21.048285
int_wald	int_wald < 151 OR int_wald IS NULL	3.0	144.358490

calculate a per-column weighted average of standard deviation values

- no magic here but want to come up with a per-column measure of variability in rows-per-partition, primarily for making recommendations
- `partition_std` values are calculated as an average across all rows/queries for that column, so the value for `num_partitions` = 100 is weighted 10x that of `num_partitions` = 10

In [32]:
```python
# weighted avg std, i.e. 100 partition's std weighted 10x the 10-num_partitions, good enough
df_qc['column_avg_std'] = (df_qc.groupby(level=['schema','table','column','num_partitions'])['counts'].transform('std')
    .groupby(level=['schema','table','column'])
 .transform('mean')
)
df_qc.head()

print('observe reasonably arbitrary column_avg_std values for int_normal column')
df_qc.xs('int_normal', level=2, drop_level=False).groupby('num_partitions').first()
```

```
observe reasonably arbitrary column_avg_std values for int_normal column
```

Out[32]:

	queries	counts	partitions_std	column_avg_std
num_partitions				
10	int_normal < 1143 OR int_normal IS NULL	24.0	1001.450393	219.331685
20	int_normal < 972 OR int_normal IS NULL	7.0	495.730827	219.331685
50	int_normal < 870 OR int_normal IS NULL	3.0	195.655883	219.331685
100	int_normal < 836 OR int_normal IS NULL	1.0	97.677887	219.331685

numPartitions=100, print first bin of each only

- when aggregating over all partition counts `int_uni` is still the winner, as expected

In [15]:
```python
df_qc.xs(100, level=3, drop_level=False).groupby('column').first()
```

Out[15]:

	queries	counts	partitions_std	column_avg_std
column				
int_normal	int_normal < 836 OR int_normal IS NULL	1.0	97.677887	219.331685
int_uni	int_uni < 50 OR int_uni IS NULL	101.0	21.048285	20.934106
int_wald	int_wald < 151 OR int_wald IS NULL	3.0	144.358490	317.890889

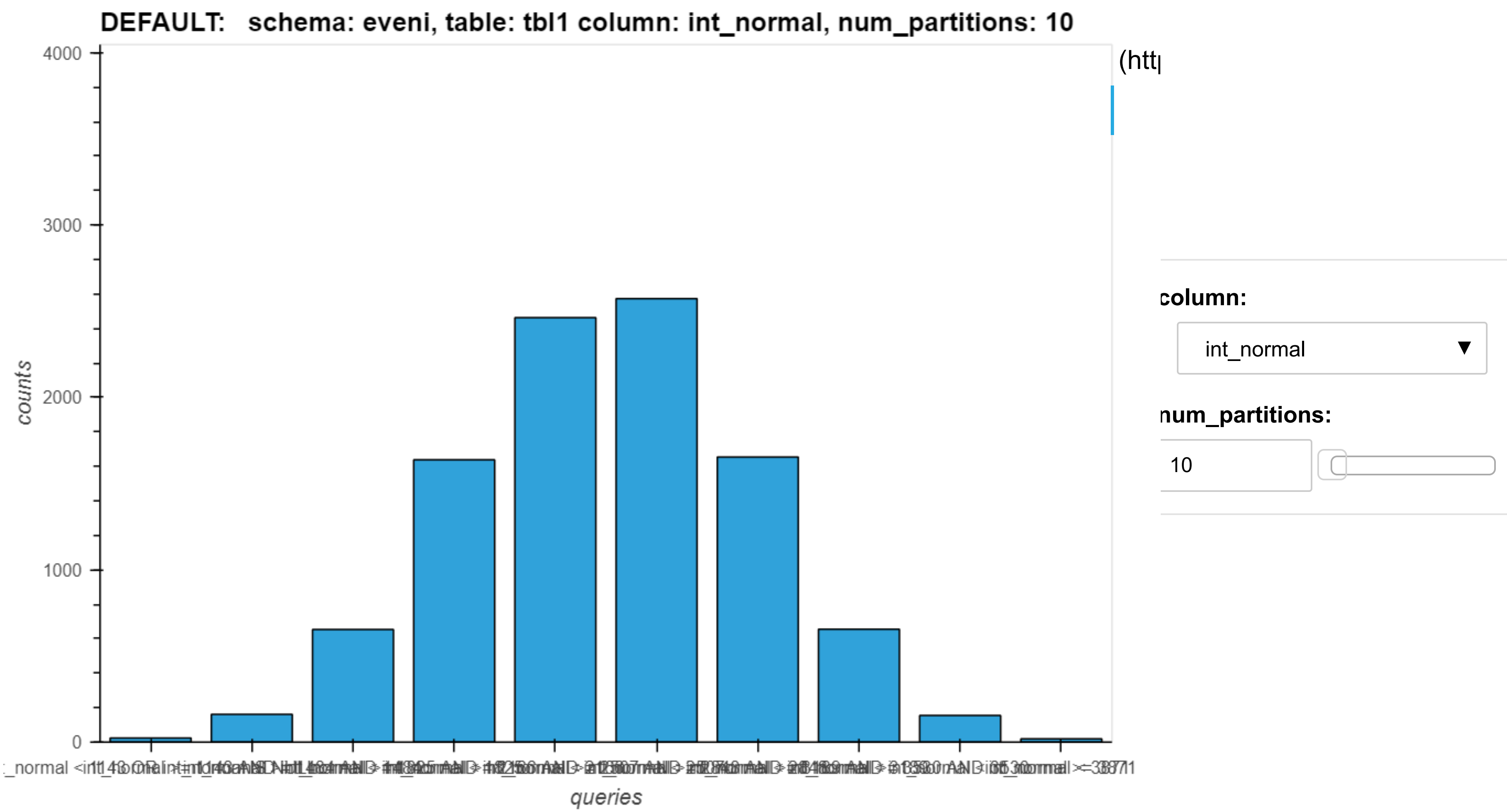
DEFAULT: dropdown (columns) and slider (num_partitions) for available choices

- default sort etc.

In [16]:
```
%%opts Bars [width=700 height=500 legend_position='top_left']

ds_vert = hv.Dataset(df_qc)
bars = ds_vert.to(hv.Bars,
                 vdims=['counts'],
                groupby=['schema','table','column','num_partitions'])
bars.relabel(group='DEFAULT:  ')
```

Out[16]:



SORTED: add ranked_column => dropdown sorted by standard deviation ASC

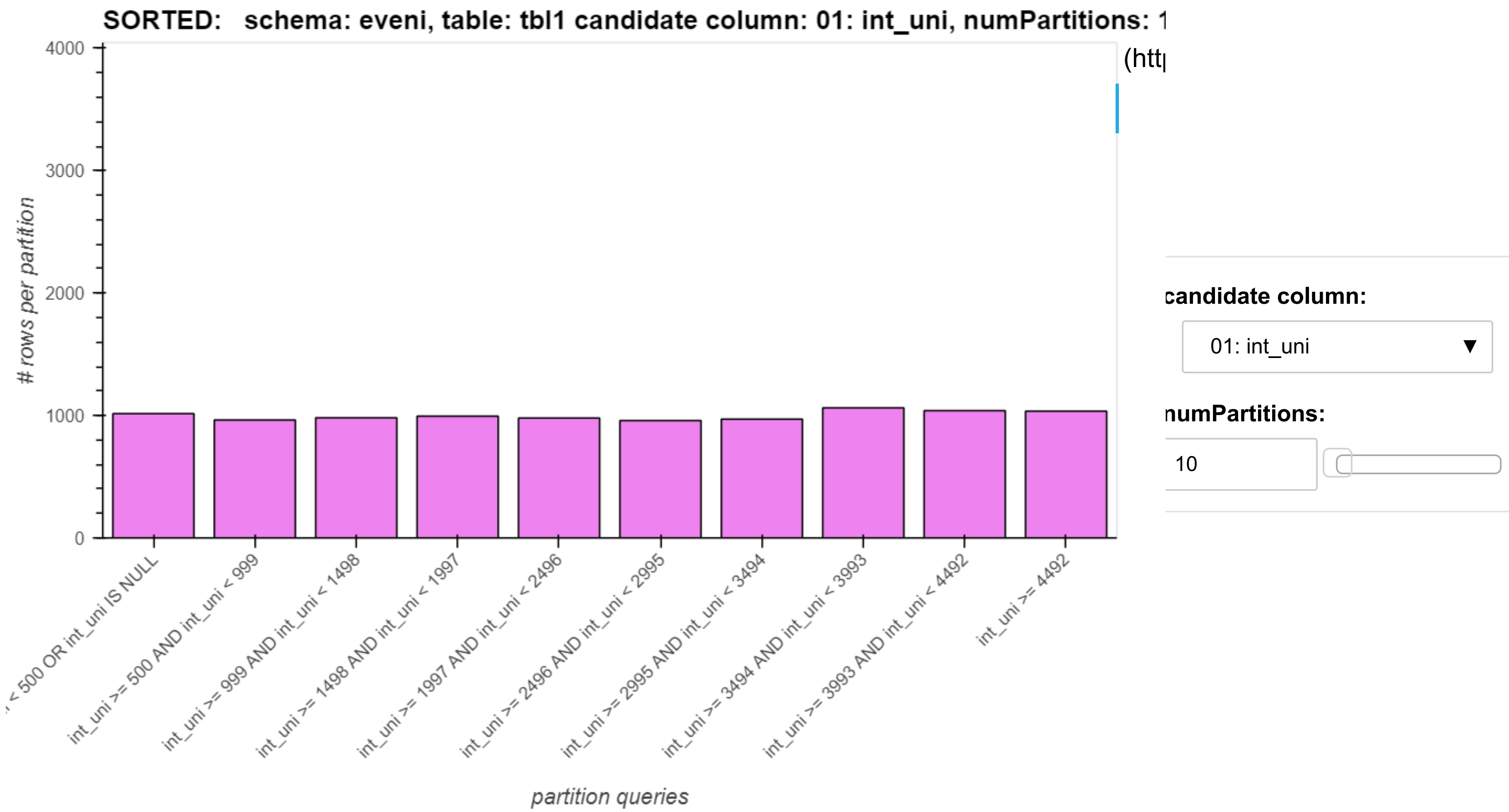
- each column is associated with a weighted average of per-column-and-num-partitions standard deviation value
- dropdown will now be sorted such that those columns with lowest standard deviation appear first
    - column listed first is likely to have most even distribution, taking all partition counts into account
- some minor aesthetic improvements, relabel axes, tilt x axis values
- **TODO**: is there some other way of sorting the dropdown in HoloViews w/o needing a new calculated column?

In [17]:
```
%%opts Bars [width=700 height=500 xrotation=45 legend_position='top_left'] ( color='violet' )

# up to 99 columns, create a string version of column name that will be alpha sorted as desired
srs_ranked_column = pd.Series(df_qc['column_avg_std'].rank(method='dense').astype('str').str.zfill(4).str.slice(0,2)
                 + ': '
                 + df_qc.index.get_level_values('column'), name='ranked_column')

ds_vert = hv.Dataset(pd.concat([df_qc, srs_ranked_column], axis=1))
bars = ds_vert.to(hv.Bars,
                  vdims=['counts'],
                 groupby=['schema','table','ranked_column','num_partitions'])
renamed = bars.redim(counts='# rows per partition', queries='partition queries',
                     ranked_column='candidate column', num_partitions='numPartitions')
renamed = renamed.relabel(group='SORTED:  ')
renamed
```

Out[17]:



above is the most useful I've come up with for POC, som other attempts follow

EVERYTHING: all distributions on the same chart

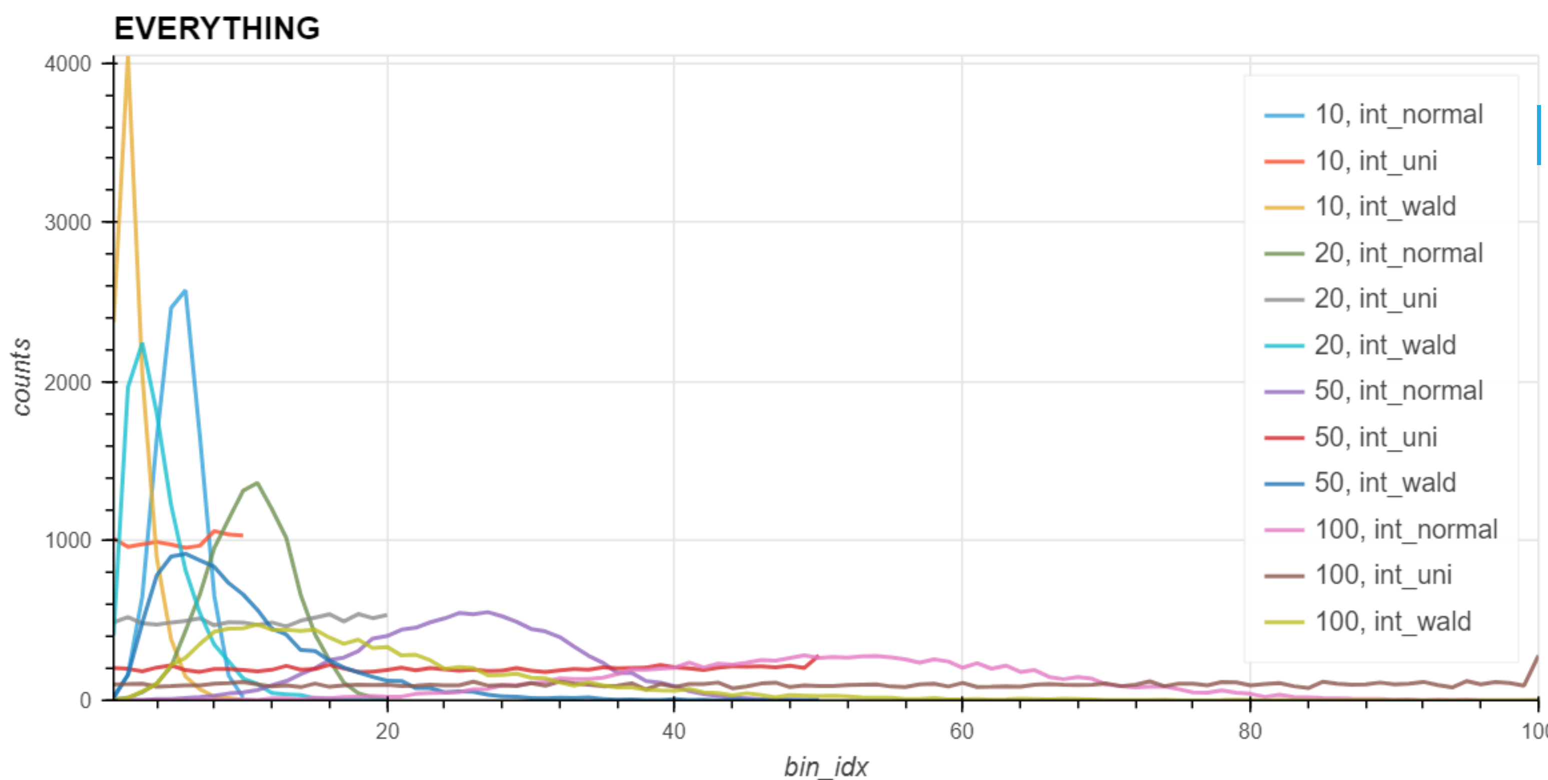
- too busy

In [18]:
```
%%opts Curve [width=800 height=400 show_grid=True ] ( alpha=0.8, line_width=2)

srs_bin_idx = pd.Series(df_qc.groupby(['schema','table','column','num_partitions']).cumcount() + 1, name='bin_idx')
ds_vert = hv.Dataset(pd.concat([df_qc, srs_bin_idx], axis=1).reset_index()[['column','num_partitions','counts','bin_id
x']])
curves = ds_vert.to(hv.Curve, kdims='bin_idx', vdims=['counts'], groupby=['num_partitions','column']).overlay()

curves.relabel(group='EVERYTHING')
```

Out[18]:



(htt

GROUPED: display series of charts for first n columns, selectable num_partitions

- nice to see all three columns at once but is this (dynamically) scalable? i.e. if there were 5 columns can we tell HoloViews to only display three columns horizontall and add a second row for columns 4 and 5?

In [19]:

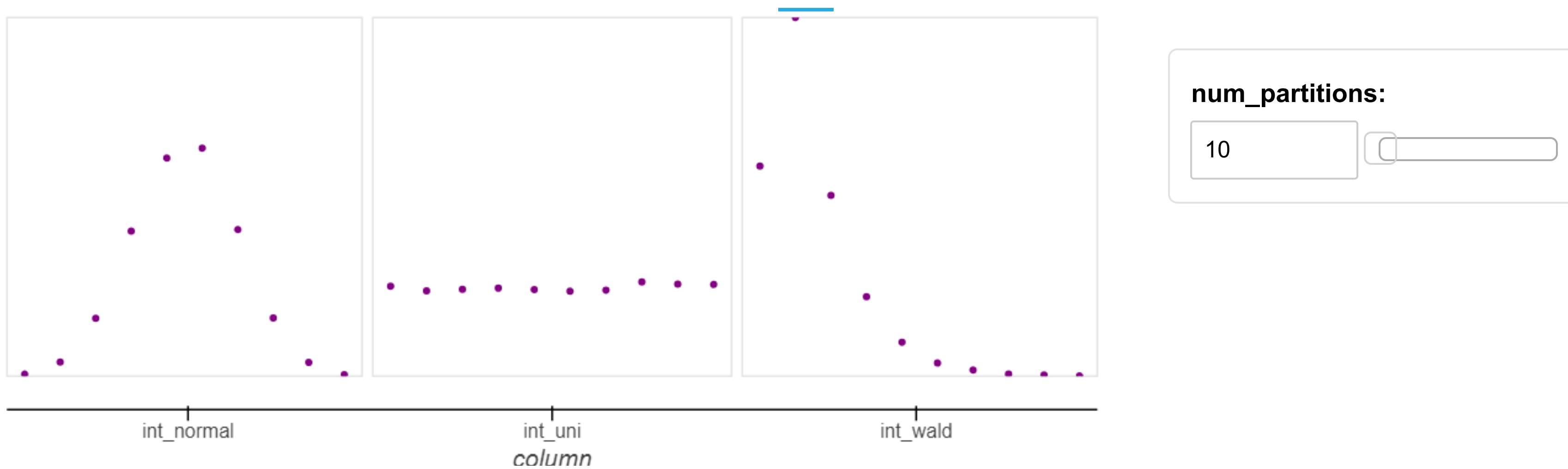
```
%%opts Scatter [width=200 height=200 ] (size=3 color='purple')

ds_vert = hv.Dataset(df_qc.reset_index(level=['num_partitions','column'])[['column','counts','queries','num_partition
s']])
grouped = ds_vert.select('column').to(hv.Scatter, 'queries', ['counts'])
grouped.grid('column').relabel('GROUPED')
```

Out[19]:

GROUPED
num_partitions: 10

(https://bokeh.pydata.org/)



Fin