Key BQL Functions Explained

Important Concepts Help Working with BQL

Use Associated Columns

BQL returns the data table with associated columns.

For example, when requesting index constituents' data (ID), it returns columns including:

Query String:

GET(ID) FOR(MEMBERS('INDU Index'))

Object Model:

ticker = bq.data.id()

index = bq.univ.members('INDU Index')

request = bql.Request(index, ticker)

ID	Weights	Positions	ORIG_IDS	ID
Members Ticker	Index Weight	Index Position	Index Ticker	Members Ticker

If we only want to take one column for custom calculation, for example, if we want to take Weights to calculate weighted average of the return, we can use the following ways to do so:

Query String:

GET(ID().Weights) FOR(MEMBERS('INDU Index'))

Object Model:

weights = bq.data.id()['Weights']

index = bq.univ.members('INDU Index')

request = bql.Request(index, weights)

If we only want to take the actual data points without the associated columns, the special column to use is "VALUE". For example If we want to take EPS data and do not want any other associated columns such as AS_OF_DATE, CURRENCY and REVISION_DATE, we can use "VALUE" only.

Query String:

GET(IS_EPS().VALUE)

Object Model:

eps = bq.data.is_eps()['VALUE']

Function VALUE

There is a special and very useful function **VALUE** in BQL helping us to introduce the data outside the universe.

For example if we want to calculate the raw beta between stocks and index, we will put the stocks list in the universe part and use function **VALUE** within GET part to introduce the index data.

Object Model:

price = bq.data.px_last(start='-1y', per='w', ca_adj='full')

price_return = bq.func.pct_diff(price)

index_price_return = bq.func.value(price_return, bq.univ.list(['INDU Index']))

beta = bq.func.slope(index_price_return, price)

request = bql.Request(['IBM US Equity', 'MSFT US Equity'], beta)

The value function can also help us to link the data sets. For example if we would like to get the ultimate parent fundamental data with a bond, we can use the function **VALUE**.

Object Model:

debt_to_ebitda = bq.data.tot_debt_to_ebitda()

debt_to_ebitda = bq.func.value(debt_to_ebitda, bq.univ.parent(TYPE='ULTIMATE'), mapby='lineage')

reqeust = bql.Request(['AU654357 Corp'], debt_to_ebitda)

Or if we want to get the fundamental data for stocks that may not be the fundamental ticker, such as VOW3 GY, which is Preferred Stock.

Object Model:

eps = bq.func.value(bq.data.is_eps(),
bq.univ.translatesymbols(TARGETIDTYPE='FUNDAMENTALTICKER'), mapby='lineage')

Here because we are translating the ultimate parent ticker from the universe (bond ticker), i.e. introducing another multi security universe, we need to use the parameter “mapby” to decide how we want to join two universes and set it as “lineage”. The parameter is default to be “broadcast”.

Lineage – join by the ID

If we are requesting security A and B with the value function:

Ticker	Data Item
A	Data derived from the original ticker A
B	Data derived from the original ticker B

Broadcast – create pairs between two universes

If we are requesting security A and B with the value function:

Ticker	Data Item
A	Data derived from the original ticker A
A	Data derived from the original ticker B
B	Data derived from the original ticker A
B	Data derived from the original ticker B

* Typical use case are pair wise correlation

Functions GROUP and UNGROUP

One of the key functions of BQL is to group and compute the group numbers. For example, we want to compute the industry median PE Ratios or we want to calculate the relative PE Ratio between stocks and industry median. In those cases, GROUP function become extremely powerful.

In most of the cases, **GROUP** function need to combine with other statistical functions such as **MEDIAN**, **AVG** and **STD**, etc. For example if we need industry median PE Ratio.

Object Model:

pe = bq.data.pe_ratio()

industry_pe = bq.func.median(bq.func.group(pe, bq.data.gics_sector_name()))

Ticker	PE	Sector
A	PE[A]	Sector A
B	PE[B]	Sector A
C	PE[C]	Sector B
D	PE[D]	Sector B
E	PE[E]	Sector C
F	PE[F]	Sector C

Final Output

Sector A	Median (A, B)
Sector B	Median (C, D)
Sector C	Median (E, F)

* Note: the second parameter of **GROUP** function can be blank, which groups the whole universe

Function **GROUP** is good to produce aggregated level statistics, however in many screening case, we would like to use the aggregated level statistics with individual security level numbers. For example, if we want to calculate the relative PE Ratio against the industry's median. The above aggregated level table cannot be easily used and we need the function **UNGROUP** to join the aggregated level table back to individual security level.

Object Model:

pe = bq.data.pe_ratio()

industry_pe = bq.func.median(bq.func.group(pe, bq.data.gics_sector_name()))

relative _pe = pe / bq.func.ungroup(*industry_pe*)

Sector	Median PE
Sector A	Median (A, B)
Sector B	Median (C, D)
Sector C	Median (E, B)

UNGROUP:

Join Back According to the Sector of the Security

Ticker	PE	Sector	Sector PE	Relative PE
A	PE[A]	Sector A	PE[Sector A]	PE[A] / PE[Sector A]
B	PE[B]	Sector A	PE[Sector A]	PE[B] / PE[Sector A]
C	PE[C]	Sector B	PE[Sector B]	PE[C] / PE[Sector B]
D	PE[D]	Sector B	PE[Sector B]	PE[D] / PE[Sector B]
E	PE[E]	Sector C	PE[Sector C]	PE[E] / PE[Sector C]
F	PE[F]	Sector C	PE[Sector C]	PE[F] / PE[Sector C]

In BQL we also have **GROUP[X]** functions such as **GROUPAVG**, **GROUPSTD** and **GROUPRANK**, etc. those are the pre-defined functions doing group and ungroup in one go. For example, our above request can be re-write to the following:

Object Model:

pe = bq.data.pe_ratio()

industry_pe = bq.func.groupmedian(pe, bq.data.gics_sector_name())

relative_pe = pe / *industry_pe*

Universe Function FILTER

Function **FILTER** is for screening with conditions. We can define the conditions with logical functions such as **AND**, **OR** and **NOT** as well as matching the text using "==" and function **IN**. Because Python has same keywords, in BQL we usually add "_" at the end, such as bq.data.and_().

We can also use function FILTER in a nested way, which helps to make the request faster, because every layer of the filtering will reduce the universe size for the next filtering. Here is an Example:

Object Model:

start_universe = bq.univ.members('SPX Index')

security_type_condition = bq.data.security_typ() == 'Common Stock'

sector_condition = bq.func.not_(bq.func.in_(bq.data.gics_sector_name(), ['Financials', 'Utilities']))

top_ten_pe = bq.func.grouprank(bq.data.pe_ratio(), ORDER='ASC') <= 10

screen = bq.univ.filter(start_universe, bq.func.and_(security_type_condition, sector_condition))

screen = bq.univ.filter(screen, top_ten_pe)

request = bql.Request(screen, bq.data.id())

The above screening starts with SPX Index. It finds all common stocks in the index, which are not in the GICS sector Financials and Utilities. After the first layer of the filtering, it then finds the top ten value stocks by ranking and finding the smallest 10 PE stocks.

Important Database Query Behaviour and Function MATCHES

Like all other database query language, BQL computes the GET part after FOR part, i.e. the data computed within BQL is based on the ultimate universe, if there is filtering with the original universe.

For example, we have original universe 500 stocks. We calculate the relative PE Ratio against the sector level median, and then we take top 50 stocks who has smallest relative PE Ratio. Finally, we request the relative PE Ratio for those 50 stocks.

In the case above, the reason that we request for the relative PE Ratio is that we want to check the actual number used to rank and pick 50 stocks. However, due to the database query language behaviour, those requested relative PE Ratio are actually based only on the final 50 stocks, which are very different from the original relative PE Ratio based on the original 500 stocks.

The solution this such issue is using the function MATCHES. Please see the example below:

Object Model:

```
index = bq.univ.members('SPX Index')

pe = bq.data.pe_ratio()

relative_pe = pe / bq.func.groupmedian(pe, bq.data.gics_sector_name())

screen = bq.univ.filter(index, bq.func.grouprank(relative_pe, ORDER='ASC') <= 10)

request = bql.Request(screen, bq.data.id())

response = bq.execute(request)

top_stocks = response[0].df().index.tolist()


top_stock_relative_pe = bq.func.matches(relative_pe, bq.func.in_(bq.data.id(), top_stocks))

request = bql.Request(index, top_stock_relative_pe)

response = bq.execute(request)

top_stocks_with_original_data = response[0].df().dropna()
```

Notice that we do the first request to find all of the top stocks by using the screen as the universe in our request. Then in our second request, we use the index (original index) as our universe, so that all lines will be calculated with original universe, and function **MATCHES** will only pick up the lines within our top stocks list. Finally, when we take the data table, we will use function **DROPNA** to only display the top stocks data.