# SQI: The Scalable Query Interface

Wes Kendall

May 30, 2010

## 1 Introduction

Query-driven visualization is a popular research topic that has shown success in a variety of scientific fields. The intuitiveness of a range query gives the scientist flexibility to describe areas of interest in the dataset. Furthermore, it has also been shown that qualitative concepts, such as "the beginning of Spring", can be constructed with queries [1]. Succinctly describing complex features becomes much easier to the user with this ability.

With scientific datasets pushing the limits of I/O systems, the Scalable Query Interface (SQI) was developed to closely integrate I/O and analysis into query-driven methods. SQI is quite similar to a Map-Reduce programming model. The data is read in using sophisticated I/O techniques. The mapping function is then the Boolean range queries that are issued, and the reduction function may be applied for further analysis of the data after sorting. These techniques are outlined in [2] and use SQI to discover climatic trends on over a terabyte of satellite data.

A typical program using SQI will call $SQI\_Init$ and give it a data configuration file. This file describes how items, i.e. each element to be queried, are set up before querying. The user may place variables from netCDF files into the items and then query these variables in their code. They are also given the ability to query dimensions and then generate additional dimensions based on which file the data is coming from. Once $SQI\_Init$ has finished, the user then calls $SQI\_Query$ as many times as they need. The results are accumulated in memory. Once querying is finished, $SQI\_Query\_sort$ may be called to sort the items in parallel so analysis may be performed. This whole process may be started over again by first calling $SQI\_Query\_reset$ and then performing querying.

The API specification is listed in Section 2. An example configuration file and code are shown in Section 3. The code is freely available at the Seelab website with the sample code provided and a dataset for testing. Further questions and bug reports about the interface may be sent to Wes Kendall. Suggestions for future direction of the interface are welcome. If anyone intends to use this interface for their research, please reference [2].

# 2 Interface

## 2.1 SQI_Init

```
int32_t
SQI_Init
  (MPI_Comm world_comm,
   const char *config_file_name,
   SQI_User_Defined_Vars_Func user_defined_vars_func)
```

**Synopsis** Initializes the interface. The data is read into memory based on the specification in the data configuration file. An example data configuration file is shown in Section 3. The data is also distributed for load balancing and the querying structures are built.

**Return** 1 on success or 0 on failure.

**Parameters**

world_comm
: The MPI communicator for SQI.

config_file_name
: The name of the configuration file.

user_defined_vars_func
: A function that takes a $void^*$ and an $int64\_t$. This function is called once for each item after being read in. The $void^*$ parameter points to the item and the $int64\_t$ parameter indicates which item it is. This function is useful for the user to define their own variables they wish to query on. The user can use existing information in the item and generate new querying information. An example would be computing differences between time steps.

## 2.2 SQI_Finalize

```
int32_t
SQI_Finalize
  ()
```

**Synopsis** Finalizes the interface and deletes all the internal structures.

**Return** 1 on success or 0 on failure.

## 2.3 SQI_Query_reset

```
int32_t
SQI_Query_reset
  ()
```

**Synopsis** Resets the internal querying structures. If this function is not called, the query results will be aggregated together each time *SQI_Query* is called.

**Return** 1 on success or 0 on failure.

## 2.4 SQI_Query

```
int32_t
SQI_Query
  (const SQI_Value *mins,
   const SQI_Value *maxs,
   const int32_t num_attrs,
   const int32_t *attr_ret)
```

**Synopsis** Submits a query to SQI. The user creates two arrays of minimum and maximum ranges of values to return. If the values of an item fit in these ranges, the item will be aggregated into a buffer. The elements of the item that will be aggregated are specified by *attr_ret* and may be returned in any order.

**Return** 1 on success or 0 on failure.

**Parameters**

mins
: The minimum values for the ranges of each element of the item. Each value is of type $SQI\_Value$, which is a union representing each possible type. The type you used must match with the type you supplied in the configuration file.

maxs
: The maximum values for the ranges of each element of the item. The type of these is similar to the *mins* parameter.

num_attrs
: The number of attributes in an item that will be stored if the item matches.

attr_ret
: Which attributes in your item will be returned. These may be listed in any order and they will be returned in that order.

## 2.5 SQI_Query_sort

```
int32_t
SQI_Query_sort
  ()
```

**Synopsis** After queries have been issued, the results are stored internally in the order specified by the user when querying. This function is used to sort all the queried items in parallel. The ordering of the items is determined by the first variable in the items. If the first variable of two items is equal, the order is determined by the second variable and so forth.

**Return** 1 on success or 0 on failure.

## 2.6 SQI_Query_print

```
void
SQI_Query_print
  ()
```

**Synopsis** Prints all the queried items. A token is passed along processes to guarantee that the output is ordered by process.

## 2.7 SQI_Get_query

```
const void *
SQI_Get_query
  ()
```

**Synopsis** Returns the queried results in item format. The item format is a packed ordering of the returned and perhaps sorted result. If the variables you chose to return from your query are not byte aligned, you will have to be sure to unpack the individual variables in a buffer for byte alignment. The buffer returned is a pointer to SQI's query buffer and it should not be freed.

**Return** Buffer of all queried items on process.

## 2.8 SQI_Get_timing

```
void
SQI_Get_timing
  ()
```

**Synopsis** Prints timing information about SQI on process 0.

# 3 Code Examples

## 3.1 ipcc_data_config.sqic

**Synopsis** An example data configuration file for reading in variables from the IPCC climate dataset. This configuration is included in the SQI package.

```
// information about the dataset. the "TYPE" specifies which type
// of data is being read in. currently SQI only supports netCDF.
// the "name" field is not significant but could
// be used in future versions of SQI. "files" points to the directory
// where the netCDF files are located. the second parameter to "files"
// tells SQI to only read the files ending in ".nc"
// "files_per_item" specifies how many files are in a single item. for
// example, if you have 2 variables in an item and these variables are
// located in different files, you will have to set "files_per_item" to 2.
// similarly, if you want the same variable stored across many files, you
// have to update the "files_per_item" parameter accordingly
data
{
    type NETCDF
    name ipcc
    files ./ .nc
    files_per_item 1
}
// the first dimension that will be in the items you query. the type
// must be specified. for this variable, the "generate" flag is specified.
// this means that the dimension is actually not present in the netCDF files
// and that we are going to assign it a number depending on which file this
// item comes from. the number is based on the alphabetical ordering of
// the files. if the user had files from multiple timesteps saved, this
// would be the way to include time information in the items you query
dim
{
    name T
    type UINT8
    generate
}
// the second dimension in the item. the name of the dimension should match
// the name that is in the netCDF file. also, although it is not necessary
// to specify a start and a size, it you may specify smaller blocks of
// data from your file to read in
dim
{
    name lat
    type UINT8
    start 0
    stop 128
}
// the third dimension in the item. this is similar to the "lat" dimension
```

```
dim
{
    name lon
    type UINT8
    start 0
    stop 256
}
// the fourth dimension in the item. since the variables in the IPCC files
// are defined among this dimension, we have to use it when reading in
// the netCDF files and include it in our item
dim
{
    name time
    type UINT8
    start 0
    stop 1
}
// the first variable and fifth overall element in our item that we can
// query. the name has to match up the name in the netCDF file and the
// type has to match as well. fill values may be specified and will be
// filtered upon being read. you may use the "eq" "neq" "gt" "gte" "lt"
// and "leq" flags when defined fill values. you may use as many of these
// flags as necessary on the same line and they will automatically combined
// with the OR operator
var
{
    name ELAI
    type FLOAT32
    fill eq 1e+36
}
// the last element in the item and similar to the "ELAI" variable
var
{
    name TSNOW
    type FLOAT32
    fill eq 1e+36
}
```

## 3.2 sqi_query_test.c

**Synopsis** Example code for using SQI. This code uses the *ipcc_data_config.sqic* configuration file and sets up parameters for querying the dataset. This example is also included in the SQI code release.

```
#include "sqi.h"

// the ordering of different attibutes in an item. "T" is a number
// representing which file the data came from. "TIME" is a dimension
// from the IPCC files that is simply always zero, but we read it in
// anyways
#define T 0
#define LAT 1
#define LON 2
#define TIME 3
#define ELAI 4
#define TSNOW 5

void
user_defined_variables(uint8_t *item, int64_t which_item)
{
  // in here, you would  define a function that takes data that was
  // read in and then use it to compute other attributes that were flagged
  // as "USER_DEFINED" in your configuration file
  // this function is not used in this simple example, but it is
  // shown for completeness.
}

int32_t
main(int32_t argc, char **argv)
{
  if (argc != 2)
  {
    fprintf(stderr, "usage: read_netcdf config_file\n");
    exit(1);
  }

  // initialize MPI
  MPI_Init(NULL, NULL);

  // initialize SQI.
  // first arg: the world that SQI is operating in.
  // second arg: the name of the configuration file.
  // third arg: a function pointer. after items are read in, this function
  // is called for every single item. it allows the user to compute additional
  // variables on the fly. lets say I want to compute a difference between
  // timesteps and then be able to query on it. this is where it happens. this
  // function does nothing in this example.
  SQI_Init(MPI_COMM_WORLD, argv[1], user_defined_variables);
```

```
// these are your min and max values for querying. the max values are
// inclusive. SQI_Value is a union.
// if the first attribute in your item is a uint8, then you will have to
// set mins[].u8. play with these parameters and print off the results.
// you will see that the printed results match these parameters.
SQI_Value mins[6], maxs[6];
// T
mins[T].u8 = 0;   maxs[T].u8 = 3;
// lat
mins[LAT].u8 = 0;   maxs[LAT].u8 = 127;
// lon
mins[LON].u8 = 0;   maxs[LON].u8 = 255;
// time
mins[TIME].u8 = 0;    maxs[TIME].u8 = 0;

// even though we filter out the fill values in the config file, there is
// a chance that some of the variables might have a fill value. if one
// point has a fill value for ELAI and a regular value for TSNOW, then it
// won't be filtered out. we need to be aware of this and query for the ranges
// below the fill value (1e+36)
// ELAI
mins[ELAI].f32 = -FLT_MAX;  maxs[ELAI].f32 = FLT_MAX;
// TSNOW
mins[TSNOW].f32 = -FLT_MAX; maxs[TSNOW].f32 = FLT_MAX;

// these are the attributes you want returned from the query. you can
// also change the order of them too. if you only want ELAI returned,
// you would say attr_ret[1] = {4} since ELAI is the fourth attribute
// in the item
int32_t attr_ret[6] = {T, LAT, LON, TIME, ELAI, TSNOW};

// this call is not necessary here, but if you want to perform multiple
// queries, you will have to reset your query buffer or else each query
// will be accumulated
SQI_Query_reset();
// issue the query here. the results are aggregated in a buffer each
// time SQI_Query is called. to reset the buffer, call SQI_Query_reset
// first and second args: mins and maxs
// third arg: amount of attributes to return
// fourth arg: array of attributes to return
SQI_Query(mins, maxs, 6, attr_ret);

// sort the query results in parallel. it will be sorted based on the
// returned attributes. in this case, it is sorted by T first, lat
// second, lon third... etc.
SQI_Query_sort();

// print all the items on each process
SQI_Query_print();
```

```
  // if you need to work with items directly from the query buffer, this
  // is how you obtain it. the attributes of each item are packed, meaning
  // that if your attibutes aren't byte aligned, you will have to unpack
  // them yourself
  //const void *query_buffer = SQI_Get_query();

  // finalize SQI
  SQI_Finalize();
  // finalize MPI
  MPI_Finalize();

  return 0;
}
```

# 4 Future Work / Known Limitations

SQI has some primary know limitations that are currently being worked on. First of all, it only supports the netCDF file format. We plan to adopt raw and HDF formats in the near future as we update BIL, the main I/O library for SQI. Second, SQI only handles datasets that can be stored in memory. We are currently working on out-of-core methods for SQI to address this.

In the first alpha release of SQI, we plan to have support for regular expression queries as outlined in [1] along with more powerful options for neighborhood and statistical analysis.

# References


[1] M. Glatter, J. Huang, S. Ahern, J. Daniel, and A. Lu. Visualizing temporal patterns in large multivariate data using textual pattern matching. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1467–1474, 2008.

[2] W. Kendall, M. Glatter, J. Huang, T. Peterka, R. Latham, and R. Ross. Terascale data organization for discovering multivariate climatic trends. In *SC '09: Proceedings of the ACM/IEEE Supercomputing Conference*, 2009.