History

Date created: Monday, April 28, 2014

Last modified: DATE \@ "dddd, MMMM d, y" Monday, April 28, 2014

Long before your time, in a land far far away, an idea to change compute and storage forever was conceived. This is the story of that idea. It all started with…

| Date | Author | Summary |
| --- | --- | --- |
| Monday, April 28, 2014 | Vikram, Mike, Jerene, Brent | First pass at architecture doc |
|  |  |  |
|  |  |  |
|  |  |  |

Overall

Xcalar consists of the following components:

1. CLI / WebGUI
2. libXcalarApis
3. MicroKernel
4. InputStream
5. BTree

APIs provided by libXcalarApis can be invoked by the users via the CLI or the WebGUI.

libXcalarApis, MicroKernel, InputStream and BTree are linked together as one binary.

CLI / WebGUI invokes Apis provided by libXcalarApis via RPC

libXcalarApis

Here are the initial APIs provided. More will be added to this list in future:

1. Handle createObject(char \*path)
2. Handle objectLookup(char \*path)
3. Status publishObject(Handle handle, Function hashFn, Range addrRange)
4. void bulkLoad(Handle handle, InputStream inputStream, Function callbackFn, Key primaryKey)
5. int aggregate(Handle handle, Key field)

**Example**

To illustrate how the APIs are to be used, here’s a toy example.

Suppose the user wants to run some analysis on Wikipedia. The entire corpus of wikipedia resides in <hdfs://toy-example/wikipedia>.

Suppose we are running on an n-node cluster.

Suppose the wikipedia corpus is really a collection of JSON objects with the following partial structure:

{

articleId : int,

title: String,

author: String,

…

}

From the CLI, the user enters:

Xcalar> load <hdfs://toy-example/wikipedia> Wikipedia articleId

“<hdfs://toy-example/wikipedia>” loaded into handle “Wikipedia” indexed by “articleId”

Here’s step-by-step what happens:

1. CLI first checks if the object named “Wikipedia” exists via objectLookup(“Wikipedia”)
2. It doesn’t. So CLI goes ahead and executes hdl = createObject(“Wikipedia”)
3. Next, CLI does publishObject(hdl, some default mod, 0 - 2 ^ 64). (The value of publishObject will only be more obvious in future when there is a need to publish the same object multiple times in the global namespace. At the moment, it is not obvious why there is a need to publish the same object multiple times.)
4. Finally CLI calls bulkLoad(hdl, inputStream, myCallback, “articleId”)
   1. On each node, the microkernel performs the equivalent of inputStream.load(“<hdfs://toy-example/wikipedia>”, nodeId, hdl, “articleId”)
   2. The effect of inputStream.load, is that all the nodes are, in parallel, loading from the data source. For example, on HDFS, you can achieve this effect with nodeID and a global path to the hdfs file (e.g. <hdfs://path/to/file>), and query the namenode for only the blocks of the file that are local to the node)
   3. inputStream.load is also given the global Handle hdl and the primary key to index by. This means inputStream has access to the global hash function.
   4. On each node, inputStream.load will need to populate n buffers, where n is the number of nodes in the cluster. Each buffer corresponds to the destination node (e.g. buffer 0 will ultimate land on node 0)
   5. Once the buffers are populated, the microkernel will take these n buffers and ship them to their respective node. This will happen in parallel on all the nodes.
   6. Once the buffer arrives at the destination, the microkernel will insert these records into the BTree via a call to BTreeBulkLoad(hdl, buffer, length). Again, this will happen in parallel on all the nodes.