DistributedWebDB

More About WebDB

Michael Cafarella February 15, 2004

This is the second of two documents that describe the Nutch WebDB. Part one covered the single-processor WebDB; we now turn to the distributed version.

We would like the distributed WebDB to be updated across multiple disks and machines simultaneously. The completed WebDB can exist across many disks, but we assume a single machine can mount and read all of them. The distributed WebDB exists so we can distribute the burdens of update over many machines.

The machines in a distributed WebDBWriter installation are connected either by NFS or by ssh/scp access to one another. (Our NFS implementation is a little more advanced right now.)

The distributed WebDBReader just loads in files from multiple locations. We aren't too worried about distributed reading; it can be implemented by WebDBReader clients instead of in the db itself.

Multiple WebDBWriter processes will each receive separate, possibly overlapping, sets of WebDBWriter API calls to apply to the db. It's OK for the WebDB to know in advance how many WebDBWriter? processes will be involved in any update operation. All of these processes will be run concurrently before the db update is finished.

Since we know the number of writers ahead of time, it's easy to partition the WebDB keyspace into k regions, one for each WebDBWriter process. MD5 keys can be evenly partitioned according to the first (log k) bits. We derived from data a set of good breakpoint URLs that evenly divide the URL keys.

It would be great if each WebDBWriter process could simply work on just its own part of the db, wholly ignorant of the other processes. Unfortunately, that's not possible. A WebDBWriter can receive an API call that might touch a record in another Writer's keyspace partition. Even worse, remember that every stage of WebDBWriter?

execution can emit edits to be processed in later stages. Each of these edits might involve a record in a different Writer's keyspace partition.

Take a moment to recall how the single-processor WebDBWriter applies changes to the WebDB:

1. Every API call that modifies the db results in writing an "edit" to disk. No edits are applied until the WebDBWriter is closed. An edit might apply to one or more of the WebDB's tables: PagesByURL, PagesByMD5?, LinksByURL?, LinksByMD5?. 2. Upon close, sort the PagesByURL edits. Now read in two sorted streams: the sorted edits and the preexisting PagesByURL database. Emit a single stream: the new, sorted, and correct PagesByURL? db. As a side effect, also emit edits that might apply to other tables. 3. Sort the PagesByMD5 edits. Merge with preexisting PagesByMD5 table, emitting the brand-new PagesByMD5 table. Also emit edits to LinksByMD5 or LinksByURL. 4. Apply same process to LinksByMD5. Emit any edits to LinksByURL. 5. Apply same process to LinksByURL. No more edits will be emitted. (Actually, this isn't strictly true. In certain circumstances we may need to re-emit the LinksByMD5 edits, and then redo step 4. Don't worry about this for now.)

So a distributed WebDBWriter cannot labor in blissful ignorance. It needs to communicate with other writers, particularly when writing out edits to be performed. We modify the above single-processor algorithm to account for this inter-writer protocol.

The Distributed DB system works as follows:

1. As before, every API call that a single WebDBWriter receives results in writing an "edit" to disk. However, there are now k different disk locations, corresponding to each of the k partitions of the web db. Each of the k WebDBWriters is writing to k different files, so there are k^2 edits files all told. Say that a WebDBWriter has finished writing out all its edits. The WebDBWriter now communicates each partition's edits file to the corresponding WebDBWriter? process. (That communication happens via our distributed filesystem, NutchFS?, but that's not critical to the algorithm here.) 2. Having been closed, each WebDBWriter should apply all relevant edits to its partition of the db. At this point, the Writer waits for each of (k-1) edits files to arrive from other machines. (Of course, one of the k partitions was generated locally.) After files for all k partitions are available, the WebDBWriter sorts the edits. The sorted edit stream is applied to the local sorted db segment, resulting in a brand-new db segment for the current partition. This process is carried out by all k WebDBWriters. The PagesByURL table is now correct. The data exists across k separate db segments, and the work was distributed across k machines. But as before, applying the PagesByURL edits does not just generate a brand-new PagesByURL table. It also generates a new set of edits, which may be applicable to any record in the entire db. Thus, as each PagesByURL? table segment is being generated, the next set of edits are generated. When each PagesByURL? table segment is complete, that Writer's emitted PagesByMD5? edits are complete as well, and so are sent to each of the (k-1) other Writers. 3. Each of the k Writers waits for its incoming set of (k-1) PagesByMD5 edit files. When they arrive, a Writer sorts the full set of k files and applies the sorted edits. It thus generates a new PagesByMD5 table segment and a full set of k LinksByMD5? edits files. 4. The same process happens across k processes for the LinksByMD5 table. Emit any edits to k edits files for LinksByURL. 5. The same process happens across k processes for the LinksByURL table. However, this time no more edits are generated. 6. The job is now done. The last step is for one of the WebDBWriters to total up the full size of the new WebDB by looking at each of the k brand-new segments. When this last WebDBWriter is finished, the entire update is complete.

Reading from a distributed WebDB is comparatively simple. The WebDB now exists in k separate pieces. When the distributed WebDBReader needs to examine a certain key, it simply checks the relevant segment first. When asked to enumerate all entries, it enumerates each of the k segments in order.

It's worthwhile to spend a little time on how the distributed WebDBWriter's inter-process communication works. No process ever opens a socket to communicate directly with another. Rather, all data is communicated via files. However, since the WebDBWriters exist over many machines and filesystems, these files need to be copied back and forth.

We do that via the NutchDistributedFileSystem. Really, "filesystem" is overstating the case a little bit. Rather it's a mechanism for a shared file namespace, with some automatic file copying between machines that announce interest in that namespace.

Every object under control of a NutchDistributedFileSystem machine group is represented with a "NutchFile" object. A NutchFile is named using three different parameters:

- "dbName" indicates the overall database that the file belongs to. This exists to enable multiple Nutch instances simultaneously on the same machine set. All files within a given instance will have the same dbName.
- "shareGroupName" is used to control where the NutchFile will be copied. Clients of the NutchDistributedFileSystem ask for NutchFiles via sharegroup. Each machine in a NutchDistributedFileSystem machine group is configured so that it knows the entire (sharegroup<->machine) mapping. For the purposes of the distributed WebDB, we create a sharegroup for each partition of the webdb. When a single WebDBWriter is emitting k separate edits files, it is writing to files in k different share groups. Having written everything out, the Writer demands to see any files meant for its particular segment's sharegroup. (We will also create a "master" sharegroup to contain the final db output.)
- "name" is just an arbitrary slash-separated filename. It describes a directory/filename hierarchy for the NutchFile in question.

A NutchFile object can be resolved to a "real-world" disk File with the help of a local NutchDistributedFileSystem object. Each machine in a NutchDistributedFileSystem machine group has a NutchDistributedFileSystem object that handles configuration and other services. One such config value is the place on a local disk where the "root" of the NutchDistributedFileSystem is found. The disk File embodiment of a NutchFile is a combination of that root, the shareGroupName, and the name.

(Of course, not all sharegroups' files will be present on each machine. That depends on the (sharegroup<->machine) mapping.)

The NutchDistributedFileSystem also takes care of file moves, deleted, locking, and guaranteed atomicity.

It should be clear that the NutchDistributedFileSystem can be implemented across any group of machines that have mutual remote-access rights. It can also be used across machines that share mutual Network File System mounts.