Function:Troilkatt/Pipeline

From FunctionWiki

A Troilkatt **pipeline** is used to do a series of processing steps on data downloaded from public repositories. This document describes the programming interface for the pipelines. For the full system design description see main design document.

See also: HEFaLMp wiki (http://gtrac-hefalmp.princeton.edu/trac/wiki/WikiStart) .

See also: Function:Troilkatt/Troilkatt Stages

Contents

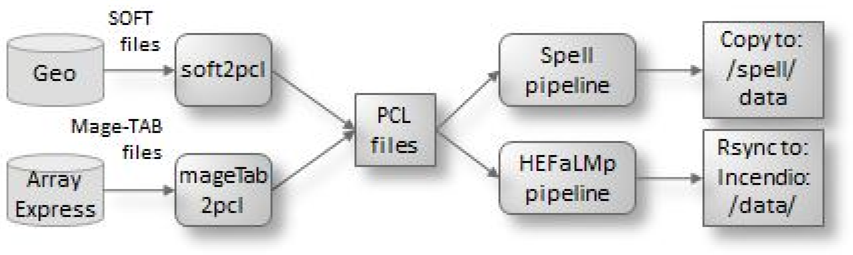
Version History

Version 0.1 [1] (http://incendio.princeton.edu/functionwiki/index.php?title=Function:Troilkatt/Pipeline&oldid=4101)

Overview

The Troilkatt system provides automated download, management, and processing of a wide variety of genomics data. To use Troilkatt the administrator configures *downloaders*, *pipelines*, and *exporters*. A downloader downloads data from a public repository and saves the downloaded files in the Troilkatt storage. A pipeline converts, integrates, and does other processing on the downloaded data. Finally, an exporter copies the processed data to a tool specific location.

The figure below shows an example configuration with two downloaders, four pipelines, and two exporters:



To add a file to a compendium the file is sent through one or more pipelines with several computation stages that each implement a data cleanup, transformation, or integration algorithm. In the pipeline the first stage is always source and the last is a sink. Each stage takes lists of new, updated, deleted, does some calculation on the all files and produces similar lists of files as output. A stage can be executed in parallel, using MapReduce, by having each mapper doing calculation on one file.

The processing pipelines has been designed to be backward compatible with previously written scripts and tools. The pipelines move all the scripts, meta-data files, and knowledge about the processing order from individual developers personal computers to a central repository.

Requirements

The Troilkatt system must provide:

- Automatic data retrieval of all datasets in GEO and ArrayExpress.
- Support for extending the data retrieval to include other data sources.
- Reliable storage of meta-data and data to a allow data compendium be rebuilt by re-downloading, re-computing, and re-integrating datasets.
- Save all previous versions of a the data in a compendium and allow users to access an old version of a compendium.
- More than 3x compression ratio for all data, version based compression for compendium, duplicate elimination, and de-compression throughput > 10 Mbps/s.
- User-friendly tools for manual addition, deletion, and updating of a dataset in a compendium.
- Automatic conversion of the downloaded microarray data to a format useful for the Spell (processed PCL), and HEFaLMp (dab) tools.
- Support for extending the data processing to include other processing tools and data formats.
- Fault-tolerant, scalable data processing.
- Automated logging and error reporting for download, conversion, and integration of datasets.
- User-friendly tools for accessing log files.
- Automated data export of final version of processed data to be used in the Spell and HEFaLMp.

Goals

The goals for Troilkatt are:

- Provide largest up-to-date collection of integrated microarray data.
- Scalability to handle storage and compute requirements of next generation data, algorithms, and tools.
- Provide the data processing for the Spell, HEFaLMp, and "function prediction" tools.

Dependencies

Troilkatt is implemented using the Hadoop software stack:

- The hadoop file system, to leverage distributed storage.
- Hadoop MapReduce for fault tolerant parallel processing.

It should be quite easy to port the system to another filesystem and job system.

The data processing done by Troilkatt has many dependencies, including:

- ruby
- python2.6
- R, including libraries from Bioconductor.
- Sleipnir

Deliverables

- Deployed automatic download system.
- Compendium of all GEO GSD (dataset), GEO GSE (series), GEO supplementary (cel, etc), and ArrayExpress files.
- Data processing pipeline for the Spell, HEFaLMp, and function prediction projects.

Open Issues

Detailed Description

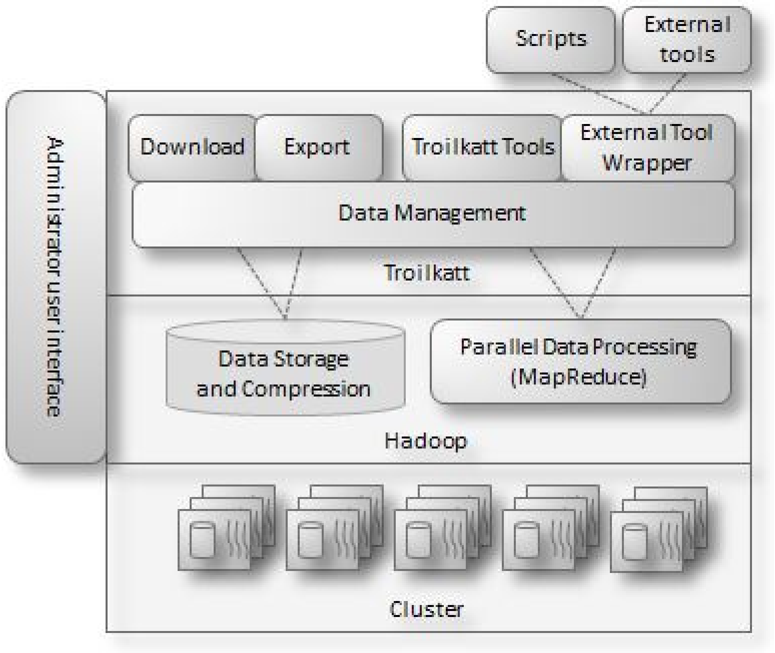
This section gives a high-level view of Troilkatt. The following two sections describe the programming interface and administrator interface.

Operational Scenario

The system is to be deployed on a 66-node cluster. It will weekly download data from GEO, ArrayExpress, Entrez Gene, and other data repositories. The data will be converted to a common format, and converted to a format that can be used by data analysis and exploration tools such as Spell or HEFaLMp. The final processed data will then be exported either to a directory on the local filesystem or copied to a remote filesystem.

Architecture

Troilkatt provides infrastructure services for automated genomics compendium management. It consists of five main components. First, the large genomics compendium maintained by Troilkatt are stored and processed on a cluster. Second, Troillkatt leverages the Hadoop software stack for reliable storage and scalable fault-tolerant data processing (including the Hadoop distributed file system (HDFS) that is similar to Google’s global file system, Hadoop MapReduce, and HBase that us similar to Google’s BigTable). Third, the Troilkatt runtime system provides data management including versioning and a library of tools for downloading and processing data. Fourth, a large collection of external tools and scripts for various data processing that can be executed by Troilkatt. Finally, a graphical user interface provides the administrator for controlling the compendium content and data processing.



Troilkatt is designed to be provide just the minimum of services required for genomics compendium management. These include:

- Data and meta-data organization and compression.
- Framework for implementing downloaders.
- Framework for specifying how data should be processed.
- Support for using existing scripts and programs for the data processing.
- Logging and error reporting.

Troilkatt is implemented on top of the Hadoop stack in order to get:

- A distributed file system.
- Scalable fault-tolerant data analysis.
- Framework for implementing compression.

The Troilkatt user interface tools are implemented as client tools built using Troilkatt libraries.

Execution Model

Data processing in Troilkatt is specified by a set of pipelines. Each pipeline has a configuration file that specifies a source, a set of data processing stages, and a sink.

The main loop in Troilkatt executes one pipeline, and one pipeline stage at a time. A pipeline consists of a source and a set of stages executed sequentially. There is one stage instance for each input file to the stage. The stage instances are independent of each other and can therefore be executed in parallel without any synchronization. For improved fault-tolerance the data produced by each stage is written to persistent storage after execution. Pseudo code for the main loop in Troilkatt is therefore as follows:

```
readGlobalMetaData()                            # Read global meta data
for p in pipelines:
  inputFiles = p.source.retrieve()              # Get list of files to process
  for s in p.stages:
    for inputFile in inputFiles execute in parallel:
      inputData = read(inputFile)               # Read input data or copy input file to local FS
      outputData, logData, metaData = s.process(inputData, timestamp)
      writeToTmp(outputData, logData, metaData) # Save output, log, and meta data in temporary directory
    write(tmpDir, outputDir)                    # Move files from temporary directory to persistent storage
    inputFiles = outputFiles                    # The next stage process the files output by the previous stage
  sink(outputFiles)                             # Last set of output files are written to the sink
writeGlobalMetaData()                           # Save global meta data
```

Each stage in the pipeline should read data from an input file, do some processing on the data, write the output data to a specified output file, and save the log information in a specific log file (it is also possible to use stdin, stdout, and pipes). Troilkatt takes care of providing the filenames, parallel execution, fault-tolerance, and to either set up a MapReduce job or move the files to a location where a non-MapReduce program can access it.

Data Storage and Compression

Troilkatt uses three different filesystems. First, HDFS is used to save all downloaded, intermediate, stage specific log-files, and processed files. We choose to use HDFS since it provides a reliable and stable file system for a cluster, and since it allows using MapReduce for scalable data processing. All intermediate files created during data processing are saved in HDFS. These are useful for the system maintainer (or developer) to check in case a file could not be converted or integrated. However, these files can periodically be deleted for example after 2-3 months.

Second, the local filesystem on each cluster node is used to temporarily save files downloaded from public repositories, and files being processed by external scripts and programs which typically require a standard POSIX filesystem interface which currently is not supported by HDFS (although there is an alternative in the fuse tool provided by Cloudera). A common operation during the data processing is therefore to move files to and from HDFS. But, this should be fast to do since MapReduce usually runs a task on the node with the input data, hence the file copy can be optimized by the operating system.

Third, NFS is used to store Troilkatt the main log-files, meta-data files, and executables. For all of these it having a single filesystem is practical and since these do not require high performance and are have small size.

HDFS provides a replication factor of three (can be configured), thus the data is relatively safe. However, catastrophic failures may occur due to operator errors or bugs in the relatively stable code. It is not practical to backup all the data in HDFS. We do not have the capacity to reliably back up everything. In case of a catastrophic failure it is possible to re-download files from the public repositories and re-compute the compendiums. However, it will be very hard to do the computation under the exactly same conditions since there may be changes to the scripts and tools, meta-data, and libraries. We therefore back up the final output of the compendium on reliable storage. The compendium can then be restored in case of catastrophic failure, ensuring that searches on the old versions return identical results. Neither the source or intermediate files are saved, so to update the compendium all source files must be re-downloaded and re-integrated. An exception are source files or intermediate files changed by the maintainer. These are also saved reliability and later added to the compendia.

There are a few compression methods supported by HDFS (gzip and bzip). We have extended HDFS with the X-Y compression method (used in BigTable) to provide efficient compression of the timestamped files.

Versioning

HDFS do not provide version control of the files. Troilkatt therefore adds a timestamp to the filename of each file it stores in HDFS. The timestamp is set when the main processing loop starts such that all files downloaded, all processing intermediate files, and all compendium has the same timestamp.

Fault Tolerance

The two important design goals for Troilkatt are fault-tolerance and scalability. In this section we describe the former, while the following section describes the latter.

There are five types of failures that may occur in Troilkatt:

1. One of the pipeline stages crashes or fails, including external programs.
2. The Troilkatt code crashes.
3. One of the cluster nodes running Troilkatt crashes, including loosing one or more of its disks.
4. HDFS or MapReduce crashes.
5. Catastrophic failure with many cluster nodes being destroyed, such that most of the data on the cluster is lost.

The first case is to be expected during a normal run since the input data is processed using a let-us-hope-it-works approach. The data and biological heterogeneity of the input data will therefore often fail or crash a stage due to unsupported input data format, corrupted input data (for example by the previous stage), unsupported dataset size, or bugs in the code. Troilkatt assumes that the pipeline stages or external programs handle all exceptions hence it isolates and tolerates failures.

Failure isolation handles: (i) programs that use too much memory causing a program to start swapping, and (ii) programs that run too long causing a job to hang. To restrict the amount of memory that can be used by a single stage we use the Java Virtual Machine maximum heap size for pipeline stages implemented in Troilkatt (including extensions), and a minimal-container that limits the memory usage of an external program using the setrlimit mechanism in Linux. Maximum runtime monitoring is provided by MapReduce. Note that the failure isolation assumes that the administrator has verified that the programs are not malicious and that they do not have bugs that cause serious damage to the system.

Fault-tolerance requires detecting failures, clean-up in case of a failure, and re-execution of failed stages. All processing is deterministic, so in case of a crash the processing can be re-run to produce the missing results. Failure detection and stage (task) re-execution is provided by MapReduce. Clean-up is provided by Troilkatt by deleting all temporal data after each execution.

The second case is less common, but may occur if the Troilkatt job is killed (for example if the cluster is needed for other higher priority jobs). Since a Troilkatt update may take a long time to run, we avoid re-executing all pipeline stages by maintaing a status that is updated before and after the execution of a stage. In case of a Troilkatt crash, the status can be checked such that execution can processed at the stage under execution at the time of the crash (see following section for details).

If a slave node in the cluster crashed re-execution of the tasks and re-distribution of the data will be handled by MapReduce and HDFS. If the node running the Troilkatt master process crashes, Troilkatt can be restarted as described in the previous paragraph. If, HDFS or MapReduce crashes Troilkatt will also crash. In that case Troilkatt will detect that these systems are not present and it will shutfown. If a significant amount of cluster nodes crash, some of the data stored in HDFS may become unavailable. This will cause an exception which will shut down troilkatt. However, it may also cause the MapReduce processing to run extremely slow. This is an condition which must be detected by the administrator who must then kill Troilkatt manually.

Catastrophic failure requires rebuilding the compendium as described in Data Storage.

Recovery

Troilkatt provides recovery in order to avoid redoing expensive computation steps. No other recovery is necessary since Troilkatt does not maintain any meta-data structures that may become corrupt. To provide for such recovery, Troilkatt maintains a file with the status of a Troilkatt execution that is used for recovery. The status files is used as follows:

```
timestamp = getTimestamp()                        # Timestamp for this execution
setStatus(timestamp, 'Troilkatt', 'start')        # Record the start of an execution
for p in pipelines:
  setStatus(timestamp, p.getSource().name(), 'start') # Set source status to started
  inputFiles = p.getSource().retrive(timestamp)       # Retrieve files
  setStatus(timestamp, p.getSource().name(), 'done')  # Set source status to done
  for s in p.stages:
    setStatus(timestmap, s.name(), 'start')       # Set stage status to start
    for inputFile in inputFiles execute in parallel:
      outputData, logData, metaData = s.process(inputData)
      writeToTmp(outputData, logData, metaData) # Save output, log, and meta data in temporary directory
    writeToStorage(tmpDir, outputDir)           # Move output, log, and meta data to output directory
    setStatus(timestmap, s.name(), 'done')      # Update stagestatus
  setStatus(timestamp, p.getSink().name(), 'start') # Set sink status to started
  p.getSink().sink(outputFiles)                     # Last set of output files are written to the sink
  setStatus(timestamp, p.getSink().name(), 'done')  # Set source status to started
setStatus(timestamp, 'Troilkatt', 'done')       # Record the end of an execution
```

To implement crash recovery Troilkatt will before a new execution check the status file. If the last status is 'Troilkatt done', the last execution completed and a new can start. Otherwise, Troilkatt enters recovery mode. First, the status file will be scanned to find the start entry of the previous execution, and the pipelines that were to be executed in the last execution. Then the following pseudo code is executed:

```
for p in recoverPipelines:
  source = p.getSource()
  if getStatus(recoverTimestamp, source.name()) != 'done':   # Source file retrieval did not complete in last e
    setStatus(timestmap, source.name(), 'start')
    inputFiles = source.retrieve()                           # Re-run retrieve function
  else:                                                      # Source function complete, so attempt to recover
    setStatus(timestmap, source.name(), 'recover')
    inputFiles = source.reocver(recoverTimestamp)            # The recovery function is source dependent
  setStatus(timestmap, source.name(), 'done')
  for s in p.stages:
    lastStatus = getStatus(recoverTimestamp, s.name())
    if lastStatus != 'done':                                 # This stage crashed in last execution
      setStatus(timestmap, s.name(), 'start')
      for inputFile in inputFiles execute in parallel:
        s.process(inputData, recoverTimestamp)               # Re-execute stage
        ...
    else:                                                    # This process function completed in the last iter
      setStatus(timestmap, s.name(), 'recover')              # so attempt to recover list of output files
      for inputFile in inputFiles execute in parallel:
        s.recover(inputData, recoverTimestamp)               # Timestamp files with recovery timestamp
        ...
    setStatus(timestmap, s.name(), 'done')
 sink = p.getSink()
 if getStatus(recoverTimestamp, sink.name()) != 'done':      # Sink did not complete in last execution
    setStatus(timestmap, sink.name(), 'start')
    sink.sink()                                              # Re-run sink function
    setStatus(timestmap, sink.name(), 'done')
 for p in pipelines:
   do normal execution
```

During recovery Troilkatt avoids re-executing stages that completed successfully. To re-execute a stage it uses the versioning support in Troilkatt to get a list of files produced in the previous iteration. Note that after recovery the pipelines are executed again. Also note that Troilkatt accept changes to a dataset configuration file between the crash and recovery. Such changes are often necessary since many crashes are due to errors in the configuration file. Finally, note that Troilkatt does not guarantee that two execution produce identical results since external programs, meta-data, and data repositories may updated between executions.

Scalability and Performance Considerations

Troilkatt is designed to scale with respect to data compendia size, data processing, and pipeline manageability. Scalable storage is required to save all available expression data, multiple versions of each compendium, and a large volume of intermediate files. HDFS and the cluster provides the required storage capability. Scalable processing for all the data is provided by MapReduce. Finally Troilkatt provides an approach for configuring a pipeline, writing data processing stages, controlling the execution, and log analysis.

Most of the data download and processing done by Troilkatt is not performance critical since the compendium are only updated once a week. Also, often the compendium are only incremented by adding the new datasets. However when building a new compendium for example for a new tool, then it is usually necessary to process thousands of datasets. In addition changes to the gene-name meta-data may require re-buidling the compendium.

The typical data processing pipeline in Troilkatt reads a set of files from a directory, runs the file through one or more processing stages, and writes the output files to another directory. Most MapReduce programs therefore consist of one mapper per input file, which does all the processing. However, Troilkatt also has MapReduce programs (such as gene name counting) that does the processing one row at a time.

Scaling Troilkatt jobs is challenging due to:

1. The large number of datasets to be processed, hence several terabytes of data must be processed at each stage in the pipeline.
2. Some of the datasets are tens of gigabyte in size, while others are only a few hundred kilobytes.
3. Many pipeline stages work on one dataset at a time, hence the memory requirements are large for the biggest datasets and the jobs often have a load imbalance.

User Interfaces

Troilkatt has two types of users:

Maintainer is (are) persons responsible for maintaining the system, by using tools that provide:

1. Error logs for downloaded datasets that could not be converted, processed, or integrated.
2. GUI tools for viewing detailed logs for the processing of each dataset.
3. Scripts for adding, deleting, or updating a dataset in a compendium.
4. Scripts for manually adding a dataset to be processed and integrated.

Model developers are persons using the datasets and integrated data to develop new integration models. These use:

1. The programming API described below.
2. Scripts for downloading files maintained by Troilkatt.

Programming Interface

This section describes how to add new sources, pipeline stages, MapReduce programs, or sinks to Troilkatt.

Code Organization

The troilkatt code is in the SVN repository: svn://gen-svn.princeton.edu/hefalmp/ under trunk troilkatt.

The code is organized as an Eclipse Java Project, with the following directories:

- src/ for the Java source code
- conf/ for the Troilkatt configuration and datasets file.
- pipelines/ for the pipeline specification files.
- scripts/ for Troilkatt scripts and various other data processing scripts.

The Java source code is divided into the following packages:

- edu.princeton.function.troilkatt: contains the main class file Troilkatt.java.
- edu.princeton.function.troilkatt.clients: troilkatt clients.
- edu.princeton.function.troilkatt.mapreduce: MapReduce programs.
- edu.princeton.function.troilkatt.pipeline: pipeline stages.
- edu.princeton.function.troilkatt.sink: pipeline sinks.
- edu.princeton.function.troilkatt.sources: pipeline sources including data downloaders.
- edu.princeton.function.troilkatt.tools: various tools for processing pipeline data or meta-data.
- edu.princeton.function.troilkatt.utils: various libraries.

The scripts/ directory has the following structure:

- (root): Troilkatt scripts.
- array_utils/: various microarray processing scripts.
- array_utils/spell/: microarray processing scripts from the Spell tool.
- R/: R scripts.

Pipelines

Data processing in Troilkatt is specified by a set of pipelines. Each pipeline has a configuration file that specifies a source of files to process, a set of data processing stages, and a sink where the final data is stored.

Source

A source is a module that outputs a list of files to be processed. The list of files may be the content of a directory, the files in a directory added in this Troilkatt executoin, or the files not already downloaded from a remote repository. The parameters for a sink are:

- Name which combined with the pipeline name forms a unique identifier.
- The sink type (such as directory-reader, GEO-downloader, etc).
- Sink type specific arguments.
- Output directory in HDFS (relative to the Troilkatt root directory).
- Compression format to be used for the output data.
- Persistent storage time (intermediate data can for example be deleted after a week).

All source classes inherit from the Source superclass. The crawler class has three main methods:

- retrieve2(): is the method called by the Troilkatt main loop in order to execute this class.
- retrieve(): does the actual downloading and/or construction of the output file list. The different source implementations must override this function.
- recover(): is a method called by the Troilkatt main loop during recovery. Stage specific reocovery operations may be implemented here.

Pseudo code for the retrieve2() method is:

```
retrive2(timestamp):
  tmpOutput = createHDFSTmpDirectiory()
  createLocalFileSystemDirectories()
  downloadMetadataFiles()
  fileList[] = retrieve(timestamp, tmpOutput)
  saveOutputDirFiles()
  saveLogDirFiles()
  saveMetadataFiles()
  output(fileList)
  moveHDFSFiles(tmpOutput, hdfsOutputDir)
```

Pseudo code for a recover() method may be:

```
recover(recoverTimestamp):
    return getOutputFiles(recoverTimestamp) # Get files created in last iteration
```

Example pseudo code for the retrieve function() for a source that downloads a large volume of data may be:

```
retrieve(currentTimestamp):
  retrivedFiles = []                     // list of new files
  hdfsFiles[] = getFilelist(outputDir)   // get list of files in HDFS output directory
  repositoryFiiles[] = repository.ls()   // for example do an FTP ls()
  for f in repositoryFiles:
    if f not in hdfsFiles:                 // is a new file
      downloadedFile = repository.download(f)
      moveToHDFS(outputDir, downloaddFile) // move file to HDFS (tmp) output directory
      retrievedFiles.add(downloadedFile)
  return retrievedFiles
```

Stage

A pipeline stage specifies the processing to be done for the input data. Stage parameters are:

- A pipeline wide unique name.
- The stage type (such as KNNImputer, Dab2Dat converter, etc).
- Stage type specific arguments.
- Output directory in HDFS (relative to the Troilkatt root directory).
- Compression format to be used for the output data.
- Persistent storage time (intermediate data can for example be deleted after a week).

All source classes inherit from the Stage superclass. The stage class has three main methods:

- process2(): is the method called by the Troilkatt main loop in order to execute this class.
- process(): does the actual processing of the output file list. The different stsge implementations must override this function.
- recover(): is a method called by the Troilkatt main loop during recovery.

Pseudo code for process2() is:

```
process2(inputFiles, timestamp):
  tmpOutput = createHDFSTmpDirectiory()
  createLocalFileSystemDirectories()
  downloadMetadataFiles()
  for f in inputFiles:
    process2(f)
  saveOutputDirFiles()
  saveLogDirFiles()
  saveMetadataFiles()
  output(fileList)
  moveHDFSFiles(tmpOutput, hdfsOutputDir)
```

A process() function for a gene name mapping stage may consist of :

```
process(inputFile):
  inputStream = open(inputFile)
  outputStream = createOutputFile(inputFile)
  for line in inputStream:
    mappedLine = mapGeneNames(line)
    outputStream.write(mappedLine)
```

MapReduce Stage

A stage may be implemented as a MapReduce program for parallel execution.

Pseudo code for such a program is:

```
mapRecords[] = getInputFilenames()
for m in mapRecords:
  inputFilename = m.getInputValue()
  inputData = readFromHDFS(inputFilename)
  for s in p.stages:
    outputFilename, outputData = s.execute(inputData)
    outputPath = hdfsTmpDir + outputFilename // attempt specific
    outputFile = writeToHDFS(outputPath, outputData)
    output("files", outputFilename)
    inputData = outputData

for r in reduceRecords:
  for tmpFilename in r.getInputValues():
    move(tmpFilename, outputDir)
```

The program has a input formatter that splits the input records where each record is one file. The mappers receive just the filename to reduce the memory size of the mapper process (otherwise the entire file content had to be in memory). If a mapper is used to start external programs it first copies the file from HDFS and then executes the programs by forking of a child process. After each step the output and log files are moved to am attempt specific temporal directory in HDFS. Finally the path of the output file is sent to the reducer, which moves the temporal file to the actual output directory. The extra move is necessary to ensure that there are no race conditions in case multiple mappers are started for a given input file.

Troilkatt MapReudce applications have speculative execution disabled to prevent long running jobs to be restarted on multiple nodes.

To implement a non-script based map-reduce job that does processing on one file at a time the PerFile class can be sub-classed. The PerFile super-class takes care of creating the input-file list and saving the meta-data after the job has completed.

Script Stage

A stage can also be implemented by implementing a sub-class for the TroilkattScript Python object that overrides the run() function. Refer to code comments in scripts/troilkatt_script.py for additional details.

Sink

The sink specifies a script to be run in order to export the output data to another file system or to a remote file system.

All sink classes inherit from the Sink superclass which has the sink() method that must be overridden. Pseudo code for a sink that copies the files to a directory on the local filesystem may be:

```
sink(outputFiles, timestamp):
  localFSDir = getArgument("localFSDir")
  for f in outputFiles:
    copy(f, localFSDir + f)
```

Data Management and Organization

Troilkatt automatically organizes data in the different filesystems and moves data between the filesystems. The data is organized based on the pipelines, where each source and stage has six directories:

1. Input directory that contains the data files to be processed (a source does not have an input directory).
2. Output directory where the source or stage output files are written.
3. Temporal file directory that can be used to save temporal files that should not be archived.
4. Log directory where all logfiles are written.
5. Meta-data directory that contains stage specific meta-data files.
6. Global meta-data directory which is a directory shared between all stages that contains meta-data used by several stages.

HDFS

Permanent storage for the input, output, and log files is provided by HDFS. The meta-data and global meta-data are stored in NFS. During execution a stage may also have an input, output, and log directory on the local filesystem. Troilkatt will move the input files to from the HDFS to the local file system, and it will move the output files from the output and log directory.

Troilkatt has a root 'troilkatt' directory in HDFS that is split into:

- data/ for all the data files. The organization into sub-directories is specified by the pipeline configuration files (the output-dir parameters).
- log/ for all log files. There is one sub-directory for each pipeline that has one sub-directory for each stage with one sub-directory per execution (named using the timestamp)
- meta/ which contains an archive of the meta data directory content after each Troilkatt execution.
- global-meta/ which contains an arhcive of the global-meta data content after each Troilkatt execution.

This gives the following structure on HDFS:

```
<troilkatt-HDFS-dir>/
  data/
    <as configured in dataset files>
  log/
    <pipeline>/<stage number>-<stage name>/<timestamp>/
  meta/
    <pipeline>/<stage number>-<stage name>/<timestamp>/
  global-meta/
    <timestamp>/
```

All HDFS filenames are of the type: filename.filetype.timestamp.compression

For most files there is a file with the same filename in all processing or repository directory (GSE_X or GSD_Y for files from geo, and a five letter ID for files from ArrayExpress). However, for the latter the filenames may not use a common ID (especially in ArrayExpress). The timestamp is added when the file was created and all files created during the same main iteration has the same timestamp. Compression defines the compression format used.

Local Filesystem

For external programs that require a POSIX API, Troilkatt saves temporary files on the local filesystem. The file structure for a stage instance is:

```
<troilkatt-tmp-dir>/<pipeline>/<stage number>-<stage name>/<input filename>/
  input/
  log/
  meta/
  output/
  tmp/
<global-meta-dir>/
```

The input and meta data directories contains files copied by Troilkatt before the stage execution. The stage program writes output files to the output directory, updates the meta-data files in the meta directory, and writes log-files to the log directory. The files in these three directories are then moved to HDFS when the stage has executed. In addition there is a tmp directory where temporal files are saved.

The content of the troilkatt-tmp-directory is deleted once all stages in a pipeline have executed and all data has been moved to HDFS.

NFS

A global filesystem such as NFS is used to hold the Troilkatt binaries, external programs, and global-meta data.

Logging

Log files output by each step are saved in the logfile directory specified in the Troilkatt configuration file. The directory structure is:

```
<tmpDir>/
  log/
    <pipeline name>/<stage number>-<stage name>/<timestamp>/
```

Status File

The Troilkatt status file is a text file with the following format:

```
<timestamp>: <id>: <status>
```

The timestamp is the time at the start of the troilkatt main loop, and it will be the same for all statuses in an execution. Id is the unique pipeline or stage name (<pipeline>-<stage number>-<stage name>). Status can be either: 'start', 'recover' and 'end', where start is set at the start of a pipeline and stage execution, end is set at the end of a pipeline and stage execution, and recover is set at the start of a pipeline or stage recovery.

Administrator Interface

This section describes how to configure Troilkatt sources, pipelines, sinks, and other parameters.

Configuration Files

Troilkatt has two main configuration files. The Troilkatt configuration file specifies environment specific parameters such as directories to use in HDFS and the local filesystem, path to external programs, and meta-data directories. An example configuration file can be found in *conf/troilkatt.xml*. This file contains all parameters that can be set, including a description of each parameter.

The set of pipelines to run, and the order which they are run, is specified in *conf/datsets* file. This file contains a list of XML filenames with the specification for each pipeline.

Pipeline Configuration

The downloaded files are processed to convert the data to a format that can be integrated. The processing pipeline consists of a set of stages that are run in sequence for an input file. Each stage consist of a Java class or external program that takes as input a file and writes the output as a file. The output files are then used as input files for the next stage in the pipeline. Troilkatt organizes the directories for the input, output, and log files. However, for each stage it is necessary to specify stage-specific arguments and how long the output data should be kept before being deleted. This information is specified in a dataset configuration file in the datasets directory. This file is in XML format and has three main elements: source, pipeline, and sink.

In the **source** element the code to retrieve the data to be processed is specified. The **pipeline** element specifies a list of **stage** elements executed in the specified order. The source and stage elements have both the following sub-elements:

- *name* is the stage name which is used in the log files and to in the data organization.
- *type* is the downloader or stage class called to retrieve or porcess the data. The download and stage classes implemented in Troilkatt are described below.

- *arguments* are the arguments sent to the download or stage class instance. Troilkatt provides many symbols that can be used to specify input file, output directory, log directory, etc. These are described below.
- *output-directory* is the directory in HDFS where the output files are saved. The directory is relative to the Troilkatt root directory.
- *compression-format* specifies the compression algorithm to be used for the output data.
- *storage-time* specifies the number of days to keep the data. If set to -1, the data is never deleted.
- *description* can be used to add a description for the downloader or stage.

The **sink** elements specifies a exporter class used to move data from HDFS to a tool specific directory either on a local or remote filesystem. A sink element has the following sub-elements:

- *name* is a name which is used in the log files and to in the data organization.
- *type* is the sink class called to retrieve or porcess the data. The sink classes implemented in Troilkatt are described below.
- *arguments* are the arguments sent to the sink class instance.

Troilkatt symbols

Troilkatt symbols can be used in program arguments to specify various directories and such. These are replaced by Troilkatt before executing the program.

The directory and file symbols are unique for each stage:

- TROILKATT.INPUT_DIR: a directory on the local filesystem with the input files for this stage.
- TROILKATT.OUTPUT_DIR: a directory on the local filesystem where the output files for this stage are stored.
- TROILKATT.LOG_DIR: a directory on the local filesystem where the log files for this stage are stored.
- TROILKATT.META_DIR: a directory on the local file system where the meta-data files are stored.

Some symbols are file and iteration specific:

- TROILKATT.FILE: the basename of the file being processed.
- TROILKATT.FILE_NOEXT: the basename of the file being processed with all extensions removed.
- TROILKATT.TIMESTAMP: the timestamp for the current Troilkatt iteration.

The program, script, and data directories are specified in the troilkatt configuration file:

- TROILKATT.DIR: the Troilkatt root directory.
- TROILKATT.BIN: directory with various binaries.
- TROILKATT.UTILS: directory with microarray data processing scripts and binaries.
- TROILKATT.SCRIPTS: directory with Troilkatt scripts.
- TROILKATT.GLOBALMETA_DIR: directory with shared meta-data files.

In addition the following symbols are supported to allow using command line arguments within command line arguments:

- TROILKATT.REDIRECT_OUTPUT: ">"
- TROILKATT.REDIRECT_ERROR: "2>"
- TROILKATT.REDIRECT_INPUT: "<"
- TROILKATT.SEPERATE_COMMAND: ";"

Implemented Sources

The main purpose of a source is to provide a list of files to be processed by the pipeline. Thee files may already be in Troilkatt or they may be downloaded from a remote repository. Such downloaders must often be tailored to a specific repository. However, Troilkatt provides a source superclass that can be extended to implement a new downloader. In addition the ScriptSource can be used to quickly implement a downloader in a scripting language.

The list of implemented sources is:

- Source: source superclass that contains useful functions for implementing a downloader. See below for a description about how to implement a downlaoder.
- NullCrawler: a stage that does not return any files. Used for debugging only.
- ListDir: outputs a recursive list of files in a HDFS directory.
- ListDirNew: outputs a recursive list of files in a directory that have been added since the last Troilkatt iteration.
- ListDirDiff: compares the content of two directories to find the files in the source directory that do not have a corresponding file in the destination directory.
- FileSource: read in a list of files from a file.
- ExecuteSource: retrieves the files by executing a script that outputs files to a directory specified in the arguments.
- ScriptSource: retrieves the files by executing a Troilkatt Python script specified in the argument. A script typically downloads new data files, uncompresses these, and saves them in a local directory. The content of this directory is then moved to HDFS by the ScriptDownloader.
- GeoGSDMirror: maintains a mirror of all GEO GSD Dataset SOFT files. It outputs the files downloaded since the last execution.
- GeoGSEMirror: maintains a mirror of all GEO GSE Series SOFT files, and it outputs the files downloaded since the last execution.
- GeoRawMirror: maintains a mirror of all GEO supplementary files. It also outputs the files downloaded since the last execution.
- ArrayExpressMirror: maintains a local repository of all ArrayExpress raw, MAGE-TAB, and meta-data files.

Implemented Stages

The list of implemented pipeline stages is:

- Stage: pipeline stage superclass that can be extended to implement a new pipeline stage.
- NullStage: a stage that does not do anything beside returning the list of input files (for debugging and testing).
- ExecutePerFile: execute an external program specified in the arguments for each each input file. Troilkatt symbols should be used to specify the input file and output files for the program.
- ExecutePerDir: execute one instance of an external program specified in the arguments once per iteration. The program will typically do some processing on all files in the directory.Troilkatt symbols should be used to specify the input and output directory for the program.
- ScriptPerFile: execute a Troilkatt script once per input file. The input and outout files are specified using Troilkatt symbols.
- ScriptPerDir: execute a Troilkatt script once per iteration. The script must read input files from a directory, and write output files to a directory, specified using Troilkatt symbols.
- Filter: selects files to send to the next stage as specified by a regexp string given as argument.
- MapReduce: execute the stage as a MapReduce job (see below).
- MapReduceStage: execute the stage in parallel as a MapReduce job(see below).

In addition there are stages implemented as MapReduce programs:

- MapReduceStage: superclass for MapReduce stages.
- MapReduceExecutePerFile: execute a MapReduce where each mapper executes an external program as specified in the arguments.
- MapReduceApp: executes a MapReduce job as specified in the arguments.
- MetaSplitSoft: splits meta data from SOFT files.
- MetaMatchSoft: selects files based on the SOFT file meta-data fields.
- PCLStats: calculates various statistics for PCL files.
- GeoDatasetParser: convert a GEO GSD SOFT file to the PCL format, and extracts useful meta-data for further processing of the data.
- GeoSeriesParser: convert a GEO GSE SOFT file to the PCL format, and that extracts meta-data.

In addition there are stages to process SOFT, CEL, and PCL filed such that these can be integrated into an compendium (refer to Function:Troilkatt/Troilkatt Stages for details):

- Cel2Pcl: convert a GEO supplementary archive file with CEL files to the PCL format, and to extract meta-data.
- GSESplit: split a GEO "superset" series file into the individual subsets.
- InsertMissingValues: set low quality scores to missing.
- KNNImpute: fill in missing values.
- MapGeneNames: map gene names to the Entrez Gene ID.
- AverageDuplicates: average together duplicate genes.
- NumericCleanup: do various numeric cleanup and consolidation.
- pcl2qdab: calculate gene-to-gene correlation, convert the gene distance files to a format that is more efficient to process, and compress the file by quantifying the weights.

Implemented Sinks

The sinks implemented by Troilkatt are:

- CopyToLocal: copy files to the local filesystem.
- Rsync: do an rsync to a remote (or local) directory.

Troilkatt Scripts

Troilkatt provides a Python template that can be used to implement a script that can utilize the Troilkatt data management and also the Troilkatt symbols.

The current list of implemented script is:

- troilkatt_script: the Troilkatt script superclass.
- troilkatt_downloader: a Troilkatt script that can be extended to implement a downloader.
- meta_downloader: a script that downloads various meta-data files.

Client Programs

Several small client programs are provided to ease the administration of Troilkatt. This section describes these.

Data management

The data, log-files, and meta-data can be viewed and accessed using the hadoop tools. However, Troilkatt provides some tools that simplify frequently used operations:

- TroilkattClient: a superclass that can be used to implement additional clients.
- GetFiles: copies the most recent version of all files in an HDFS directory to a local file system, uncompresses the files, and removes the timestamp.
- GetFile: copy a file from HDFS to the local file system.
- PutFiles: add new data files.
- PutFile: add a new data file.
- DeleteDir: delete a data directory.
- DeleteFile: remove a data file.
- CleanupDir: delete old versions for files with multiple versions in a directory.
- ReCompressDir: add or change the compression algorithm for the files in an HDFS directory.
- TimestampDir: convert the filenames in a directory to the Troilkatt filename format.

Log analysis

The main task of the system maintainer is to check the logfiles to ensure that all downloaded data could be processed and integrated, verify that errors have not been introduced to the compendium, and to take actions in case of errors. To help with this task we provide some tools:

- FileViewer: creates a table that shows the size of a file at each pipeline stage. This tool is useful to detect which stage caused errors for a specific file in the pipeline.
- PipelineStatistics: creates a table with statistics for each pipeline, such as number of files in each stage.
- CompendiaTimelime: creates a timeline that shows the files in a comepndia.
- LogViewer: allows downloading timestamped logfiles for eitehr all stages, for a stage name, or for a filename throughout the pipeline.
- RowCounter: counts the number of genes in the output files for each stage of a pipeline (only works with PCL files currently).
- ColCounter: counts the number of columns in the output files for each stage of a pipeline (only works with PCL files currently).

Extending Troilkatt

To extend Troilkatt with new data sources, pipeline stages, sinks, or MapReduce programs it is usually only necessary to write a subclass that implements a few functions as described in the programming interface section.

Data Sources and Formats

See also Additional Data Sources and Data Formats See also File Formats

This section describes the data sources supported by the Princeton *c8* installation of Troilkatt. It also described the data format for the data downloaded from these. The most important data sources for genomics data are a few large data repositories that contain most of the published experiment data. There are standards (MIAME, MINSEQE) for what the data should contain In addition each data repository has a standard file formats, but these differ among the repositories. We currently mirror all experiment data uploaded to NCBI GEO and EBI ArrayExpress.

Gene Expression Omnibus (GEO)

See also: Function:Troilkatt/GEO

GEO contains gene expression experiment data both in normalized (SOFT) format and the raw data.

From GEO we mirror all:

- Series (GSExxx.soft) files.
- Dataset (GSDxxx.soft) files.
- *Full'* datasets (GSDxxx_full.soft) files.
- Series raw data (GSExxx_RAW.tar) files.

The series, dataset, and full dataset files are in the SOFT file format. A SOFT file contains meta data and gene expression values. Troilkatt converts all SOFT files to the PCL format (described in soft to pcl conversion), and extracts meta-data before processing the data further. Refer to [2] (http://www.ncbi.nlm.nih.gov/geo/info/soft2.html) for additional details about the SOFT file format.

The raw data is platform specific and may contain for example CEL files for datasets produced by the AffyMetrix Microarray instruments. The SOFT files are generated based on the raw data, but the processing may not be optimal and the quality of the data can be improved by re-normalizing the raw data. For this we use platform specific scripts as described in raw data normalization.

ArrayExpress

See also: Function:Troilkatt/ArrayExpress

TODO: update

Entrez Gene

See also: [3] (ftp://ftp.ncbi.nih.gov/gene/README)

Entrez Gene [4] (http://www.ncbi.nlm.nih.gov/gene) contains frequently update gene name mappings. We download the info files for each organism from the NCBI FTP site. Included in this file are for each organism:

- Taxonomy ID used to uniquely identify an organism.
- Entrez Gene ID, an integer used to uniquely identify a gene over all organisms.
- Symbol (or systematic name, or official name) with the name of the gene.
- Synonyms (or alias) for the gene.
- Database synonyms that contains cross references to other databases.
- Type of gene.

The above are used to generate a mapping of most gene names to a unique ID (the Entrez Gene ID) for all genes in our datasets, using the following rules:

1. The taxonomy ID for the row should match the taxonomy ID for the organism.
2. The type of gene is "protein-coding".
3. For rows with duplicate gene ID, the gene ID of the latter is changed to the gene ID of the first
4. Add all synonyms that map to a single gene ID.
5. Add all database synonyms that map to a single gene ID.
6. Add the symbol systematic name to gene ID mapping.
7. Add the systematic name to gene ID mapping.
8. Add the identity gene ID to gene ID mapping.
9. Add additional synonym to gene ID mappings read from additional files, but filter out all that do not map to a valid gene ID.

The mapping of each alias to the systematic gene name is stored in a file: <KEGG ID>.map with the following format:

```
alias or common name or systematic name<tab>systematic name 1|systematic name 2|...|systematic name N<newline>
```

PCL File Format

The format of the processed PCL files used as input to the data integration algorithms is:

Processed PCL File Content.

ID	NAME	GWEIGHT	Description 1	Description 2	...	Description E
EWEIGHT			**1**	**1**	**...**	**1**
Gene1	**Gene1**	**1**	**f.ff**	**f.ff**	**...**	**f.ff**
Gene2	**Gene2**	**1**	**f.ff**	**f.ff**	**...**	**f.ff**
...					**...**	
GeneN	**GeneN**	**1**	**f.ff**	**f.ff**	**...**	**f.ff**

The identifiers in the first *ID* columns have been mapped to a common namespace (Entrez Gene ID), such that there are no aliases and an identifier uniquely identifies a gene (note that it does not necessarily be a gene). The *NAME* column is typically identical to the *ID* column. The *descriptions* used as column headers for the data headers may be long and descriptive.

The, *GWEIGHT* column values have also been set to 1 during preprocessing, usually be calculating the average for genes occurring multiple times. Similarly, the values in the *EWEIGHT* row has also been set to 1.

Sleipnir DAT, DAB, and QDAB Format

TODO

Exported Data

This section describes the data collections exported by Troilkatt. This section also describes tools that can be used to access the data.

GEO

For GEO the c8 deployment of Troilkatt export the following:

- Series SOFT files.
- Series meta data extracted from the SOFT files.
- Series raw data.
- (Series pre-mapping PCL files: series files converted to the PCL format and imputed to insert missing values.)
- "Final" series PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- QDAB files created using the final PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- Raw series data.
- Raw PCL files: re-normalized raw data.
- "Final" raw series PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- QDAB files created using the final raw series PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- Dataset SOFT files.
- Dataset meta data extracted from the SOFT files.
- (Dataset pre-mapping PCL files: series files converted to the PCL format and imputed to insert missing values.)
- "Final" dataset PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- QDAB files created using the final dataset PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold.
- Full dataset SOFT files.
- Combined raw series and series PCL files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold: all raw series PCL files, and the series PCL files for the series without a raw PCL file.
- Combined raw series and series QDAB files for human, mouse, rat, arabidopsis, yeast, worm, fly, frog, and slime-mold: all raw series QDAB files, and the series QDAB files for the series without a raw QDAB file.

TODO: tools for downloading/exporting these datasets.

ArrayExpress

TODO

Spell

TODO: what should be exported.

Function Project

For the function project the combined raw series and series QDAB files are exported.

TODO: describe sink.

Pilgrim

For Pilgrim the raw series PCL, series PCL, and dataset PCL files are exported.

TODO: describe sink.

Deployment

TODO: innstalation instructions

Testing

Status

Unit Testing

Integration Testing

System Testing

Regression Testing

Stress Test

Evaluation

<To be updated>

Result and good (Spell), parallel system in 5 seconds, but on how many machines (effeciency). Linear scalability.

Downloading: correctness. But user can select updates.

Space usage: is part of effeciency.

Node failures.

Compendium studies:

- Number of zero elements in Spell distance matrices.

Performance microbenchmarks:

- Hbase read latency for a row with 1500 4 byte values for different number of columns.
- Hbae read latency when varying row size.
- Time to create a lookup table from DAT files.
- Perturbation of searches when updating lookup table with new datasets.
- Read performance of "bined" lookup table when varying bin size
- Write performance for "bined" lookup table
- Storage overhead of "bined" lookup table
- Overhead of starting a MapReduce job

Storage an integration:

- Ratio of datasets that could not be integrated

Search query part:

- Correctness: make sure the results for a query is identical to sequential version on yeast data.
- Interactive performance:
  - Time from "click on search" until results are displayed.
  - Use queries described in Spell paper.
  - Scalability test with regards to number query genes.
  - Scalability test with regards to number of datasets.
  - Scalability test with regards to number of genes in genome.

Performance

Space requirements

Fault-tolerance

Previous Work

Future Directions