##### **Git Repository**

<https://github.com/cmti95035/BigData/tree/master/analytics>

This is a private repository.

##### **Maven Modules**

groupId: com.cmti

artifactId:

analytics: master

cmti-base: common library

cmti-db: relational database common library

cmti-hbase: HBase common library

cmti-kafka: Kafka common library

cmti-redis: Redis common library

cmti-spark: Spark common library

cmti-storm: Storm common library

station-lib: Application Station common library

station-app: Application Station backend code

station-web: Application Station web code

tracking-lib: Application Highway Tracking common library

tracking-app: Application Highway Tracking backend code

tracking-web: Application Highway Tracking web code

##### **Eclipse Setup**

In project home directory, run "mvn eclipse:eclipse -Dwtpversion=2.0".

2 Eclipse files are generated in the directories: .project and .classpath. Use them to create your Eclipse project.

##### **Configuration**

For every running environment/site, we need to define a specific set of configurations, like database and HBase connections, log file locations, etc.

When we run our code in a site, says QA, we only need to add this VM arg to the java command: "-Dsite=qa". The code then looks up configurations from properties file \src\main\resources\app\app\_qa.properties.

All QA specific settings are stored in \src\main\resources\app\app\_qa.properties, which looks like:

*#Database*

*datasource.jdbc.driver = com.mysql.jdbc.Driver*

*datasource.jdbc.url = jdbc:mysql://quickstart.cloudera:3306/hbase*

*datasource.user=cmti*

*datasource.password=xxxxxxx*

*#HBase*

*hbase.config=hbase-site\_qa.xml*

As a developer, I created my own properties file \src\main\resources\app\app\_gmo.properties. When I test my code locally, I only need to add "-Dsite=gmo" in my VM arg.

Properties that are not site specific are put in \src\main\resources\app\share.properties

**How to use it in java code**:

\src\main\java\com\cmti\analytics\db\DataSources.java has some examples:

*Configuration config = Config.getConfig();*

*String jdbcDrive = config.getString("datasource.jdbc.driver" + dbSuffix);*

##### **Logging**

Similar to the configuration above, [Log4j2](http://logging.apache.org/log4j/2.x/) properties for QA is defined in \src\main\resources\log4j2\_qa.xml. Add this VM arg to the java command: "-Dlog4j.configurationFile=log4j2\_qa.xml" when running code in QA.

Log4j2 allows logs to be saved to DB. In our QA site, it is saved to table app\_log, which is defined in \src\main\resources\log4j2\_qa.xml.

##### **HBase Row to Java Domain Object Mapping**

Let's use 'Mr' object as an example. We manually define the HBase to Java mappings in MrMapping.java.

Here, among other things, we specify the table name, default column family, row keys, and each column's java type, java field name, Hbase column name, and column family.

Current supported java types: Long, Integer, Double, Float, Boolean, String, Date, byte[], and array([ ]) and ArrayList for String, Long, Integer, Double, Float. More can be added if needed.

All tracking mapping classes are in package com.cmti.analytics.app.tracking.hbase.domain.mapping.

Please note that all mapping classes extend HBaseObject.

##### **HBase Rowkey Design**

The 2 rowkeys for 'mr' table are imsi and time. See MrMapping.java. In our application, we study a individual’s tracking, so imsi is the first row key.

##### **Auto Generated Domain Bean**

Given a mapping definition, says MrMapping.java, we run a code generation tool to generate a subclass (bean) of it, com.cmti.analytics.app.tracking.hbase.domain.bean.MrBean, where getter and setter methods are provided, with dirty detect ability.

All tracking app bean classes are in package com.cmti.analytics.app.tracking.hbase.domain.bean.

To add business logic to a Bean, we create a subclass of it, com.cmti.analytics.app.tracking.hbase.domain.Mr. In other business logic code, we should use this class, instead of mapping and bean.

**Implementation details**: The tool contains 2 java files, both in package com.cmti.analytics.hbase.domaingen, and a few [Velocity](http://velocity.apache.org/) template files, located at \src\main\resources\domaingen.

**How to run the tool**: *java -cp C:\git\cmti2\cmti\analytics\tracking-app\target\tracking-app-1.0-SNAPSHOT.jar -Ddir=C:\git\cmti2\cmti\analytics\tracking-lib\src\main\java\com\cmti\analytics\app\tracking\hbase -Dpackage=com.cmti.analytics.app.tracking.hbase com.cmti.analytics.hbase.domaingen.GenSource Mr MrOnRoad Road RoadCell RoadTest RoadTestData*

##### **HBase Generic Dao**

All DAOs extend HBaseGenericDao.java.

All subclass DAOs are lightweight, since HBaseGenericDao implements all the functions. Actually, all DAOs extend ExportDao, which extends HBaseGenericDao and provides common data export features, to be discussed in the next section.

For tracking app, all DAOs are in package com.cmti.analytics.app.tracking.hbase.dao.

**How to use a DAO:** com.cmti.analytics.app.tracking.task.scan.scanhandler.DriveTestRoadCellHandler has an insert example.

For retrieving data from HBase, com.cmti.analytics.app.tracking.hbase.dao.RoadCellDao has an example.

Basically HBaseGenericDao does all HBase data access operations like update, insert, and delete. It is recommended that one reads through HBaseGenericDao.java to see what it can do and how it does it.

**Implementation details**: HBaseGenericDao uses a MappingMetaData to store all the mapping info defined in a mapping file (e.g. MrMapping.java). For a given java domain object, MappingMetaData knows how to extract data from the object and construct corresponding HBase row info to update HBase. See MappingMetaData.getUpdate() for an example. For retrieving data from HBase, MappingMetaData knows how to parse a HBase row into a java domain object. See MappingMetaData.parse() for an example.

MappingMetaData itself stores column mapping info in mappedColumns (a List), and key mapping info in mappedKey. A key can be a single key, (see SingleKey.java), or a composite key, (see CompoKey.java). A column is a single property column only, (see SingleColumn.java).

SingleKey/CompoKey/SingleColumn obtains all info about a field defined in the mapping through java annotation and java refection. It then relies on a (java type specific) mapper to convert between java value (like a String, int, etc.) to byte[] that HBase uses.

All mappers are in package com.cmti.analytics.hbase.mapping.mapper. \com\cmti\analytics\hbase\mapping\mapper\Mapper.java is the super class of all mappers. We have one mapper for every supported java type: Long, Integer, Double, Float, Boolean, String, Date, byte[], String[], ArrayList<String>. CompositeMapper is used to support CompoKey.

All mappers use ByteFormatter to do the dirty object <-> byte[] conversion work.

##### **Database Generic Dao**

We use a third party library [hibernate-generic-dao](https://code.google.com/p/hibernate-generic-dao/). JPAGenericDao is a wrapper on GenericDAOImpl (from 3rd party library) that allows Spring to inject EntityManager and JPASearchProcessor. All our Daos extend JPAGenericDao.

In station app, for OLAP database access, all domain-to-table mappings are in package com.cmti.analytics.app.station.olapdatabase.domain, and all Daos are in package com.cmti.analytics.app.station.olapdatabase.dao. Service is a layer on top of Daos. All services are in package com.cmti.analytics.app.station.olapdatabase.service.

OlapUserCountMRHandler.doReduce() has an example of how the services are used.

For each domain-to-table mapping, you need to add an entry to resources/META-INF/persistence.xml, a standard JPA configuration file.

##### **Spring Framework Configuration Settings**

In resources/spring folder, 2 config files are loaded by Spring: config\_qa.xml and app.xml. (\_qa is for QA site, see Configuration section).

config\_qa.xml loads site specific configurations from db\_qa.properties. Currently all configurations are database related.

app.xml includes all other xml files: db.xml, service.xml, repository.xml.

##### **Multitasking MapReduce Framework**

In a MapReduce job, a table scan is expensive, especially if the table is huge and it is a full scan. We want to do as many this-table-related tasks as possible within one MapReduce job/scan.

Package com.cmti.analytics.hbase.task.mapreduce provides such a framework. An application example can be found in MrMR.java.

MrMR extends FullScanMR<Mr>, and provides minimal setups: a scan ( how to scan table 'mr'), a few 'mr' handlers. Each handler is for one task. Every handler extends BaseMRHandler<MR> (a dummy handler), and provides necessary doMap, doReduce, doCombine. A handler example is TrackingMRHandler.java. Handlers are enabled/disabled in app.properties like:

com.cmti.analytics.app.tracking.task.mapreduce.mr.mrhandler.TrackingMRHandler=true

##### **Multitasking Table Scan Framework**

Similar to MapReduce framework, we have a framework for just the table scan task, no MapReduce involved. The implementation is in package com.cmti.analytics.hbase.task.scan.

Example usage can be found in ScanDriveTestData.java, where only one handler is setup.

##### **OLAP**

run \src\main\scripts\olap\_time\_dimension.sql to create time dimension. Other dimensions and fact table schema are in /tracking-lib/src/main/script/schema/db.

OlapUserCountMRHandler.java is plugged in to the RecordSigMR(see above Multitasking MapReduce Framework). When doing a full table scan, it populate the fact table and update the dimension tables if a new dimension row is found.

##### **Saiku Setup and Deployment**

To be written.

##### **Code Deployment**

In project home directory, run "mvn package", a fat jar file is generated: \target\analytics-1.0-SNAPSHOT.jar. Here [maven-shade-plugin](http://maven.apache.org/plugins/maven-shade-plugin/) is used to pack all dependencies into this single jar file. See \pom.xml for details.

We ftp the jar file to QA ~/analytics-1.0-SNAPSHOT.jar.

Here are the commands to run all the tasks on QA:

Data Import

hadoop jar ~/tracking-app-1.0-SNAPSHOT.jar com.cmti.analytics.app.tracking.task.importer.HdfsMrXmlBulkLoader /data/input/mrxml /data/output/mrxml -D mapreduce.map.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml" -D mapreduce.reduce.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml"

MapReduce job

export HADOOP\_OPTS="-Dsite=qa -Dlog4j.configurationFile=log4j2\_qa.xml"

hadoop jar tracking-app-1.0-SNAPSHOT.jar com.cmti.analytics.app.tracking.task.mapreduce.mr.MrMR -D mapreduce.map.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml" -D mapreduce.reduce.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml"

##### **Database and HBase Schema**

DB schema for log4j2 logging table and DBValue is in \src\main\scripts\db\_schema.sql. The db connection info is defined in resources/app/app\_prod.properties.

OLAP star schema is in /tracking-lib/src/main/script/schema/db.

HBase schema is stored in \src\main\scripts\hbase\_schema.txt.

##### **Hive Mapping**

##### Need to map some HBase tables to Hive tables, for other teams to explore our data.

Mapping schemas are in /tracking-lib/src/main/script/schema/hive.

##### **Data Import**

To load MR and drive test data to HBase, the data files are first loaded to HDFS like:

hdfs dfs -put \*.zip /data/input/mrxml

Then we run MapReduce job to read the data files from HDFS and generates HBase’s HFile.

hadoop jar ~/tracking-app-1.0-SNAPSHOT.jar com.cmti.analytics.app.tracking.task.importer.HdfsMrXmlBulkLoader /data/input/mrxml /data/output/mrxml -D mapreduce.map.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml" -D mapreduce.reduce.java.opts="-Dsite=gmo -Dlog4j.configurationFile=log4j2/log4j2\_prod.xml"

Later we use hbase shell tool to load the HFile into HBase table:

hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles /data/output/mrxml mr

**AWS EMR setup**

To configure a HBase cluster, using these parameters:

Hadoop distribution: Amazon

AIM version: 3.8.0

Additional applications: HBase

EC2 key pair: gmo

EMR role: iamRole

EC2 instance profile: ec2

Using default for the remaining parameters.

**Switch between AWS and VM**

AWS’ Hadoop and HBase versions are different from those of my local VM. These files need to modify when switching between them:

1. /analytics/pom.xml
2. /cmti-hbase/pom.xml
3. /tracking-app/pom.xml

**Deploy web war file**

183.221.242.172