package org.apache.hadoop.hbase;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

import org.apache.hadoop.hbase.io.ImmutableBytesWritable;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.WritableComparable;

/\*\*

\* An HColumnDescriptor contains information about a column family such as the

\* number of versions, compression settings, etc.

\*

\* It is used as input when creating a table or adding a column. Once set, the

\* parameters that specify a column cannot be changed without deleting the

\* column and recreating it. If there is data stored in the column, it will be

\* deleted when the column is deleted.

\*/

public class HColumnDescriptor implements WritableComparable {

// For future backward compatibility

// Version 3 was when column names become byte arrays and when we picked up

// Time-to-live feature. Version 4 was when we moved to byte arrays, HBASE-82.

// Version 5 was when bloom filter descriptors were removed.

// Version 6 adds metadata as a map where keys and values are byte[].

private static final byte COLUMN\_DESCRIPTOR\_VERSION = (byte)6;

/\*\*

\* The type of compression.

\* @see org.apache.hadoop.io.SequenceFile.Writer

\*/

public static enum CompressionType {

/\*\* Do not compress records. \*/

NONE,

/\*\* Compress values only, each separately. \*/

RECORD,

/\*\* Compress sequences of records together in blocks. \*/

BLOCK

}

public static final String COMPRESSION = "COMPRESSION"; //TODO: change to protected

public static final String BLOCKCACHE = "BLOCKCACHE"; //TODO: change to protected

public static final String LENGTH = "LENGTH"; //TODO: change to protected

public static final String TTL = "TTL"; //TODO: change to protected

public static final String BLOOMFILTER = "BLOOMFILTER"; //TODO: change to protected

public static final String FOREVER = "FOREVER"; //TODO: change to protected

public static final String MAPFILE\_INDEX\_INTERVAL = //TODO: change to protected

"MAPFILE\_INDEX\_INTERVAL";

/\*\*

\* Default compression type.

\*/

public static final CompressionType DEFAULT\_COMPRESSION =

CompressionType.NONE;

/\*\*

\* Default number of versions of a record to keep.

\*/

public static final int DEFAULT\_VERSIONS = 3;

/\*\*

\* Default maximum cell length.

\*/

public static final int DEFAULT\_LENGTH = Integer.MAX\_VALUE;

public static final Integer DEFAULT\_LENGTH\_INTEGER =

Integer.valueOf(DEFAULT\_LENGTH);

/\*

\* Cache here the HCD value.

\* Question: its OK to cache since when we're reenable, we create a new HCD?

\*/

private volatile Integer maxValueLength = null;

/\*\*

\* Default setting for whether to serve from memory or not.

\*/

public static final boolean DEFAULT\_IN\_MEMORY = false;

/\*\*

\* Default setting for whether to use a block cache or not.

\*/

public static final boolean DEFAULT\_BLOCKCACHE = false;

/\*\*

\* Default setting for whether or not to use bloomfilters.

\*/

public static final boolean DEFAULT\_BLOOMFILTER = false;

/\*\*

\* Default time to live of cell contents.

\*/

public static final int DEFAULT\_TTL = HConstants.FOREVER;

// Column family name

private byte [] name;

/\*\*

\* Default mapfile index interval.

\*/

public static final int DEFAULT\_MAPFILE\_INDEX\_INTERVAL = 128;

// Column metadata

protected Map<ImmutableBytesWritable,ImmutableBytesWritable> values =

new HashMap<ImmutableBytesWritable,ImmutableBytesWritable>();

/\*\*

\* Default constructor. Must be present for Writable.

\*/

public HColumnDescriptor() {

this.name = null;

}

/\*\*

\* Construct a column descriptor specifying only the family name

\* The other attributes are defaulted.

\*

\* @param familyName Column family name. Must be 'printable' -- digit or

\* letter -- and end in a <code>:<code>

\*/

public HColumnDescriptor(final String familyName) {

this(Bytes.toBytes(familyName));

}

/\*\*

\* Construct a column descriptor specifying only the family name

\* The other attributes are defaulted.

\*

\* @param familyName Column family name. Must be 'printable' -- digit or

\* letter -- and end in a <code>:<code>

\*/

public HColumnDescriptor(final byte [] familyName) {

this (familyName == null || familyName.length <= 0?

HConstants.EMPTY\_BYTE\_ARRAY: familyName, DEFAULT\_VERSIONS,

DEFAULT\_COMPRESSION, DEFAULT\_IN\_MEMORY, DEFAULT\_BLOCKCACHE,

Integer.MAX\_VALUE, DEFAULT\_TTL, false);

}

/\*\*

\* Constructor.

\* Makes a deep copy of the supplied descriptor.

\* Can make a modifiable descriptor from an UnmodifyableHColumnDescriptor.

\* @param desc The descriptor.

\*/

public HColumnDescriptor(HColumnDescriptor desc) {

super();

this.name = desc.name.clone();

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

desc.values.entrySet()) {

this.values.put(e.getKey(), e.getValue());

}

}

/\*\*

\* Constructor

\* @param familyName Column family name. Must be 'printable' -- digit or

\* letter -- and end in a <code>:<code>

\* @param maxVersions Maximum number of versions to keep

\* @param compression Compression type

\* @param inMemory If true, column data should be kept in an HRegionServer's

\* cache

\* @param blockCacheEnabled If true, MapFile blocks should be cached

\* @param maxValueLength Restrict values to &lt;= this value

\* @param timeToLive Time-to-live of cell contents, in seconds

\* (use HConstants.FOREVER for unlimited TTL)

\* @param bloomFilter Enable the specified bloom filter for this column

\*

\* @throws IllegalArgumentException if passed a family name that is made of

\* other than 'word' characters: i.e. <code>[a-zA-Z\_0-9]</code> and does not

\* end in a <code>:</code>

\* @throws IllegalArgumentException if the number of versions is &lt;= 0

\*/

public HColumnDescriptor(final byte [] familyName, final int maxVersions,

final CompressionType compression, final boolean inMemory,

final boolean blockCacheEnabled, final int maxValueLength,

final int timeToLive, final boolean bloomFilter) {

isLegalFamilyName(familyName);

this.name = stripColon(familyName);

if (maxVersions <= 0) {

// TODO: Allow maxVersion of 0 to be the way you say "Keep all versions".

// Until there is support, consider 0 or < 0 -- a configuration error.

throw new IllegalArgumentException("Maximum versions must be positive");

}

setMaxVersions(maxVersions);

setInMemory(inMemory);

setBlockCacheEnabled(blockCacheEnabled);

setMaxValueLength(maxValueLength);

setTimeToLive(timeToLive);

setCompressionType(compression);

setBloomfilter(bloomFilter);

}

private static byte [] stripColon(final byte [] n) {

byte [] result = new byte [n.length - 1];

// Have the stored family name be absent the colon delimiter

System.arraycopy(n, 0, result, 0, n.length - 1);

return result;

}

/\*\*

\* @param b Family name.

\* @return <code>b</code>

\* @throws IllegalArgumentException If not null and not a legitimate family

\* name: i.e. 'printable' and ends in a ':' (Null passes are allowed because

\* <code>b</code> can be null when deserializing).

\*/

public static byte [] isLegalFamilyName(final byte [] b) {

if (b == null) {

return b;

}

if (b[b.length - 1] != ':') {

throw new IllegalArgumentException("Family names must end in a colon: " +

Bytes.toString(b));

}

for (int i = 0; i < (b.length - 1); i++) {

if (Character.isLetterOrDigit(b[i]) || b[i] == '\_' || b[i] == '.') {

continue;

}

throw new IllegalArgumentException("Illegal character <" + b[i] +

">. Family names can only contain 'word characters' and must end" +

"with a colon: " + Bytes.toString(b));

}

return b;

}

/\*\*

\* @return Name of this column family

\*/

public byte [] getName() {

return name;

}

/\*\*

\* @return Name of this column family with colon as required by client API

\*/

public byte [] getNameWithColon() {

return HStoreKey.addDelimiter(this.name);

}

/\*\*

\* @return Name of this column family

\*/

public String getNameAsString() {

return Bytes.toString(this.name);

}

/\*\*

\* @param key The key.

\* @return The value.

\*/

public byte[] getValue(byte[] key) {

ImmutableBytesWritable ibw = values.get(new ImmutableBytesWritable(key));

if (ibw == null)

return null;

return ibw.get();

}

/\*\*

\* @param key The key.

\* @return The value as a string.

\*/

public String getValue(String key) {

byte[] value = getValue(Bytes.toBytes(key));

if (value == null)

return null;

return Bytes.toString(value);

}

/\*\*

\* @return All values.

\*/

public Map<ImmutableBytesWritable,ImmutableBytesWritable> getValues() {

return Collections.unmodifiableMap(values);

}

/\*\*

\* @param key The key.

\* @param value The value.

\*/

public void setValue(byte[] key, byte[] value) {

values.put(new ImmutableBytesWritable(key),

new ImmutableBytesWritable(value));

}

/\*\*

\* @param key The key.

\* @param value The value.

\*/

public void setValue(String key, String value) {

setValue(Bytes.toBytes(key), Bytes.toBytes(value));

}

/\*\* @return compression type being used for the column family \*/

public CompressionType getCompression() {

String value = getValue(COMPRESSION);

if (value != null) {

if (value.equalsIgnoreCase("BLOCK"))

return CompressionType.BLOCK;

else if (value.equalsIgnoreCase("RECORD"))

return CompressionType.RECORD;

}

return CompressionType.NONE;

}

/\*\* @return maximum number of versions \*/

public int getMaxVersions() {

String value = getValue(HConstants.VERSIONS);

if (value != null)

return Integer.valueOf(value).intValue();

return DEFAULT\_VERSIONS;

}

/\*\*

\* @param maxVersions maximum number of versions

\*/

public void setMaxVersions(int maxVersions) {

setValue(HConstants.VERSIONS, Integer.toString(maxVersions));

}

/\*\*

\* @return Compression type setting.

\*/

public CompressionType getCompressionType() {

return getCompression();

}

/\*\*

\* @param type Compression type setting.

\*/

public void setCompressionType(CompressionType type) {

String compressionType;

switch (type) {

case BLOCK: compressionType = "BLOCK"; break;

case RECORD: compressionType = "RECORD"; break;

default: compressionType = "NONE"; break;

}

setValue(COMPRESSION, compressionType);

}

/\*\*

\* @return True if we are to keep all in use HRegionServer cache.

\*/

public boolean isInMemory() {

String value = getValue(HConstants.IN\_MEMORY);

if (value != null)

return Boolean.valueOf(value).booleanValue();

return DEFAULT\_IN\_MEMORY;

}

/\*\*

\* @param inMemory True if we are to keep all values in the HRegionServer

\* cache

\*/

public void setInMemory(boolean inMemory) {

setValue(HConstants.IN\_MEMORY, Boolean.toString(inMemory));

}

/\*\*

\* @return Maximum value length.

\*/

public synchronized int getMaxValueLength() {

if (this.maxValueLength == null) {

String value = getValue(LENGTH);

this.maxValueLength = (value != null)?

Integer.decode(value): DEFAULT\_LENGTH\_INTEGER;

}

return this.maxValueLength.intValue();

}

/\*\*

\* @param maxLength Maximum value length.

\*/

public void setMaxValueLength(int maxLength) {

setValue(LENGTH, Integer.toString(maxLength));

this.maxValueLength = null;

}

/\*\*

\* @return Time-to-live of cell contents, in seconds.

\*/

public int getTimeToLive() {

String value = getValue(TTL);

return (value != null)? Integer.valueOf(value).intValue(): DEFAULT\_TTL;

}

/\*\*

\* @param timeToLive Time-to-live of cell contents, in seconds.

\*/

public void setTimeToLive(int timeToLive) {

setValue(TTL, Integer.toString(timeToLive));

}

/\*\*

\* @return True if MapFile blocks should be cached.

\*/

public boolean isBlockCacheEnabled() {

String value = getValue(BLOCKCACHE);

if (value != null)

return Boolean.valueOf(value).booleanValue();

return DEFAULT\_BLOCKCACHE;

}

/\*\*

\* @param blockCacheEnabled True if MapFile blocks should be cached.

\*/

public void setBlockCacheEnabled(boolean blockCacheEnabled) {

setValue(BLOCKCACHE, Boolean.toString(blockCacheEnabled));

}

/\*\*

\* @return true if a bloom filter is enabled

\*/

public boolean isBloomfilter() {

String value = getValue(BLOOMFILTER);

if (value != null)

return Boolean.valueOf(value).booleanValue();

return DEFAULT\_BLOOMFILTER;

}

/\*\*

\* @param onOff Enable/Disable bloom filter

\*/

public void setBloomfilter(final boolean onOff) {

setValue(BLOOMFILTER, Boolean.toString(onOff));

}

/\*\*

\* @return The number of entries that are added to the store MapFile before

\* an index entry is added.

\*/

public int getMapFileIndexInterval() {

String value = getValue(MAPFILE\_INDEX\_INTERVAL);

if (value != null)

return Integer.valueOf(value).intValue();

return DEFAULT\_MAPFILE\_INDEX\_INTERVAL;

}

/\*\*

\* @param interval The number of entries that are added to the store MapFile before

\* an index entry is added.

\*/

public void setMapFileIndexInterval(int interval) {

setValue(MAPFILE\_INDEX\_INTERVAL, Integer.toString(interval));

}

@Override

public String toString() {

StringBuffer s = new StringBuffer();

s.append('{');

s.append(HConstants.NAME);

s.append(" => '");

s.append(Bytes.toString(name));

s.append("'");

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

values.entrySet()) {

s.append(", ");

s.append(Bytes.toString(e.getKey().get()));

s.append(" => '");

s.append(Bytes.toString(e.getValue().get()));

s.append("'");

}

s.append('}');

return s.toString();

}

@Override

public boolean equals(Object obj) {

return compareTo(obj) == 0;

}

@Override

public int hashCode() {

int result = Bytes.hashCode(this.name);

result ^= Byte.valueOf(COLUMN\_DESCRIPTOR\_VERSION).hashCode();

result ^= values.hashCode();

return result;

}

// Writable

@SuppressWarnings("deprecation")

public void readFields(DataInput in) throws IOException {

int version = in.readByte();

if (version < 6) {

if (version <= 2) {

Text t = new Text();

t.readFields(in);

this.name = t.getBytes();

if (HStoreKey.getFamilyDelimiterIndex(this.name) > 0) {

this.name = stripColon(this.name);

}

} else {

this.name = Bytes.readByteArray(in);

}

this.values.clear();

setMaxVersions(in.readInt());

int ordinal = in.readInt();

setCompressionType(CompressionType.values()[ordinal]);

setInMemory(in.readBoolean());

setMaxValueLength(in.readInt());

setBloomfilter(in.readBoolean());

if (isBloomfilter() && version < 5) {

// If a bloomFilter is enabled and the column descriptor is less than

// version 5, we need to skip over it to read the rest of the column

// descriptor. There are no BloomFilterDescriptors written to disk for

// column descriptors with a version number >= 5

throw new UnsupportedClassVersionError(this.getClass().getName() +

" does not support backward compatibility with versions older " +

"than version 5");

}

if (version > 1) {

setBlockCacheEnabled(in.readBoolean());

}

if (version > 2) {

setTimeToLive(in.readInt());

}

} else {

// version 6+

this.name = Bytes.readByteArray(in);

this.values.clear();

int numValues = in.readInt();

for (int i = 0; i < numValues; i++) {

ImmutableBytesWritable key = new ImmutableBytesWritable();

ImmutableBytesWritable value = new ImmutableBytesWritable();

key.readFields(in);

value.readFields(in);

values.put(key, value);

}

}

}

public void write(DataOutput out) throws IOException {

out.writeByte(COLUMN\_DESCRIPTOR\_VERSION);

Bytes.writeByteArray(out, this.name);

out.writeInt(values.size());

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

values.entrySet()) {

e.getKey().write(out);

e.getValue().write(out);

}

}

// Comparable

public int compareTo(Object o) {

HColumnDescriptor other = (HColumnDescriptor)o;

int result = Bytes.compareTo(this.name, other.getName());

if (result == 0) {

// punt on comparison for ordering, just calculate difference

result = this.values.hashCode() - other.values.hashCode();

if (result < 0)

result = -1;

else if (result > 0)

result = 1;

}

return result;

}

}

/\*\*

\*

package org.apache.hadoop.hbase;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.NoSuchElementException;

import java.util.Random;

import java.util.TreeMap;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.client.HConnection;

import org.apache.hadoop.hbase.client.HConnectionManager;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Scanner;

import org.apache.hadoop.hbase.io.BatchUpdate;

import org.apache.hadoop.hbase.io.Cell;

import org.apache.hadoop.hbase.io.RowResult;

import org.apache.hadoop.hbase.regionserver.HLog;

import org.apache.hadoop.hbase.regionserver.HRegion;

import org.apache.hadoop.hbase.regionserver.InternalScanner;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.util.Writables;

/\*\*

\* A non-instantiable class that has a static method capable of compacting

\* a table by merging adjacent regions.

\*/

class HMerge implements HConstants {

static final Log LOG = LogFactory.getLog(HMerge.class);

static final Random rand = new Random();

/\*

\* Not instantiable

\*/

private HMerge() {

super();

}

/\*\*

\* Scans the table and merges two adjacent regions if they are small. This

\* only happens when a lot of rows are deleted.

\*

\* When merging the META region, the HBase instance must be offline.

\* When merging a normal table, the HBase instance must be online, but the

\* table must be disabled.

\*

\* @param conf - configuration object for HBase

\* @param fs - FileSystem where regions reside

\* @param tableName - Table to be compacted

\* @throws IOException

\*/

public static void merge(HBaseConfiguration conf, FileSystem fs,

final byte [] tableName)

throws IOException {

HConnection connection = HConnectionManager.getConnection(conf);

boolean masterIsRunning = connection.isMasterRunning();

HConnectionManager.deleteConnectionInfo(conf, false);

if (Bytes.equals(tableName, META\_TABLE\_NAME)) {

if (masterIsRunning) {

throw new IllegalStateException(

"Can not compact META table if instance is on-line");

}

new OfflineMerger(conf, fs).process();

} else {

if(!masterIsRunning) {

throw new IllegalStateException(

"HBase instance must be running to merge a normal table");

}

new OnlineMerger(conf, fs, tableName).process();

}

}

private static abstract class Merger {

protected final HBaseConfiguration conf;

protected final FileSystem fs;

protected final Path tabledir;

protected final HLog hlog;

private final long maxFilesize;

protected Merger(HBaseConfiguration conf, FileSystem fs,

final byte [] tableName)

throws IOException {

this.conf = conf;

this.fs = fs;

this.maxFilesize =

conf.getLong("hbase.hregion.max.filesize", DEFAULT\_MAX\_FILE\_SIZE);

this.tabledir = new Path(

fs.makeQualified(new Path(conf.get(HBASE\_DIR))),

tableName.toString()

);

Path logdir = new Path(tabledir, "merge\_" + System.currentTimeMillis() +

HREGION\_LOGDIR\_NAME);

this.hlog =

new HLog(fs, logdir, conf, null);

}

void process() throws IOException {

try {

for(HRegionInfo[] regionsToMerge = next();

regionsToMerge != null;

regionsToMerge = next()) {

if (!merge(regionsToMerge)) {

return;

}

}

} finally {

try {

hlog.closeAndDelete();

} catch(IOException e) {

LOG.error(e);

}

}

}

protected boolean merge(final HRegionInfo[] info) throws IOException {

if(info.length < 2) {

LOG.info("only one region - nothing to merge");

return false;

}

HRegion currentRegion = null;

long currentSize = 0;

HRegion nextRegion = null;

long nextSize = 0;

for (int i = 0; i < info.length - 1; i++) {

if (currentRegion == null) {

currentRegion =

new HRegion(tabledir, hlog, fs, conf, info[i], null);

currentRegion.initialize(null, null);

currentSize = currentRegion.getLargestHStoreSize();

}

nextRegion =

new HRegion(tabledir, hlog, fs, conf, info[i + 1], null);

nextRegion.initialize(null, null);

nextSize = nextRegion.getLargestHStoreSize();

if ((currentSize + nextSize) <= (maxFilesize / 2)) {

// We merge two adjacent regions if their total size is less than

// one half of the desired maximum size

LOG.info("merging regions " + currentRegion.getRegionName()

+ " and " + nextRegion.getRegionName());

HRegion mergedRegion =

HRegion.mergeAdjacent(currentRegion, nextRegion);

updateMeta(currentRegion.getRegionName(), nextRegion.getRegionName(),

mergedRegion);

break;

}

LOG.info("not merging regions " + currentRegion.getRegionName()

+ " and " + nextRegion.getRegionName());

currentRegion.close();

currentRegion = nextRegion;

currentSize = nextSize;

}

if(currentRegion != null) {

currentRegion.close();

}

return true;

}

protected abstract HRegionInfo[] next() throws IOException;

protected abstract void updateMeta(final byte [] oldRegion1,

final byte [] oldRegion2, HRegion newRegion)

throws IOException;

}

/\*\* Instantiated to compact a normal user table \*/

private static class OnlineMerger extends Merger {

private final byte [] tableName;

private final HTable table;

private final Scanner metaScanner;

private HRegionInfo latestRegion;

OnlineMerger(HBaseConfiguration conf, FileSystem fs,

final byte [] tableName)

throws IOException {

super(conf, fs, tableName);

this.tableName = tableName;

this.table = new HTable(conf, META\_TABLE\_NAME);

this.metaScanner = table.getScanner(COL\_REGIONINFO\_ARRAY, tableName);

this.latestRegion = null;

}

private HRegionInfo nextRegion() throws IOException {

try {

RowResult results = getMetaRow();

if (results == null) {

return null;

}

Cell regionInfo = results.get(COL\_REGIONINFO);

if (regionInfo == null || regionInfo.getValue().length == 0) {

throw new NoSuchElementException("meta region entry missing " +

COL\_REGIONINFO);

}

HRegionInfo region = Writables.getHRegionInfo(regionInfo.getValue());

if (!Bytes.equals(region.getTableDesc().getName(), this.tableName)) {

return null;

}

checkOfflined(region);

return region;

} catch (IOException e) {

e = RemoteExceptionHandler.checkIOException(e);

LOG.error("meta scanner error", e);

metaScanner.close();

throw e;

}

}

protected void checkOfflined(final HRegionInfo hri)

throws TableNotDisabledException {

if (!hri.isOffline()) {

throw new TableNotDisabledException("Region " +

hri.getRegionNameAsString() + " is not disabled");

}

}

/\*

\* Check current row has a HRegionInfo. Skip to next row if HRI is empty.

\* @return A Map of the row content else null if we are off the end.

\* @throws IOException

\*/

private RowResult getMetaRow() throws IOException {

RowResult currentRow = metaScanner.next();

boolean foundResult = false;

while (currentRow != null) {

LOG.info("Row: <" + currentRow.getRow() + ">");

Cell regionInfo = currentRow.get(COL\_REGIONINFO);

if (regionInfo == null || regionInfo.getValue().length == 0) {

currentRow = metaScanner.next();

continue;

}

foundResult = true;

break;

}

return foundResult ? currentRow : null;

}

@Override

protected HRegionInfo[] next() throws IOException {

List<HRegionInfo> regions = new ArrayList<HRegionInfo>();

if(latestRegion == null) {

latestRegion = nextRegion();

}

if(latestRegion != null) {

regions.add(latestRegion);

}

latestRegion = nextRegion();

if(latestRegion != null) {

regions.add(latestRegion);

}

return regions.toArray(new HRegionInfo[regions.size()]);

}

@Override

protected void updateMeta(final byte [] oldRegion1,

final byte [] oldRegion2,

HRegion newRegion)

throws IOException {

byte[][] regionsToDelete = {oldRegion1, oldRegion2};

for (int r = 0; r < regionsToDelete.length; r++) {

if(Bytes.equals(regionsToDelete[r], latestRegion.getRegionName())) {

latestRegion = null;

}

table.deleteAll(regionsToDelete[r]);

if(LOG.isDebugEnabled()) {

LOG.debug("updated columns in row: " + regionsToDelete[r]);

}

}

newRegion.getRegionInfo().setOffline(true);

BatchUpdate update = new BatchUpdate(newRegion.getRegionName());

update.put(COL\_REGIONINFO,

Writables.getBytes(newRegion.getRegionInfo()));

table.commit(update);

if(LOG.isDebugEnabled()) {

LOG.debug("updated columns in row: "

+ newRegion.getRegionName());

}

}

}

/\*\* Instantiated to compact the meta region \*/

private static class OfflineMerger extends Merger {

private final List<HRegionInfo> metaRegions = new ArrayList<HRegionInfo>();

private final HRegion root;

OfflineMerger(HBaseConfiguration conf, FileSystem fs)

throws IOException {

super(conf, fs, META\_TABLE\_NAME);

Path rootTableDir = HTableDescriptor.getTableDir(

fs.makeQualified(new Path(conf.get(HBASE\_DIR))),

ROOT\_TABLE\_NAME);

// Scan root region to find all the meta regions

root = new HRegion(rootTableDir, hlog, fs, conf,

HRegionInfo.ROOT\_REGIONINFO, null);

root.initialize(null, null);

InternalScanner rootScanner =

root.getScanner(COL\_REGIONINFO\_ARRAY, HConstants.EMPTY\_START\_ROW,

HConstants.LATEST\_TIMESTAMP, null);

try {

HStoreKey key = new HStoreKey();

TreeMap<byte [], Cell> results =

new TreeMap<byte [], Cell>(Bytes.BYTES\_COMPARATOR);

while(rootScanner.next(key, results)) {

key.setHRegionInfo(HRegionInfo.ROOT\_REGIONINFO);

for(Cell c: results.values()) {

HRegionInfo info = Writables.getHRegionInfoOrNull(c.getValue());

if (info != null) {

metaRegions.add(info);

}

}

}

} finally {

rootScanner.close();

try {

root.close();

} catch(IOException e) {

LOG.error(e);

}

}

}

@Override

protected HRegionInfo[] next() {

HRegionInfo[] results = null;

if (metaRegions.size() > 0) {

results = metaRegions.toArray(new HRegionInfo[metaRegions.size()]);

metaRegions.clear();

}

return results;

}

@Override

protected void updateMeta(final byte [] oldRegion1,

final byte [] oldRegion2, HRegion newRegion)

throws IOException {

byte[][] regionsToDelete = {oldRegion1, oldRegion2};

for(int r = 0; r < regionsToDelete.length; r++) {

BatchUpdate b = new BatchUpdate(regionsToDelete[r]);

b.delete(COL\_REGIONINFO);

b.delete(COL\_SERVER);

b.delete(COL\_STARTCODE);

b.delete(COL\_SPLITA);

b.delete(COL\_SPLITB);

root.batchUpdate(b,null);

if(LOG.isDebugEnabled()) {

LOG.debug("updated columns in row: " + regionsToDelete[r]);

}

}

HRegionInfo newInfo = newRegion.getRegionInfo();

newInfo.setOffline(true);

BatchUpdate b = new BatchUpdate(newRegion.getRegionName());

b.put(COL\_REGIONINFO, Writables.getBytes(newInfo));

root.batchUpdate(b,null);

if(LOG.isDebugEnabled()) {

LOG.debug("updated columns in row: " + newRegion.getRegionName());

}

}

}

}

package org.apache.hadoop.hbase;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import org.apache.hadoop.hbase.io.HeapSize;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.io.WritableComparator;

/\*\*

\* A Key for a stored row.

\*/

public class HStoreKey implements WritableComparable<HStoreKey>, HeapSize {

/\*\*

\* Colon character in UTF-8

\*/

public static final char COLUMN\_FAMILY\_DELIMITER = ':';

private byte [] row = HConstants.EMPTY\_BYTE\_ARRAY;

private byte [] column = HConstants.EMPTY\_BYTE\_ARRAY;

private long timestamp = Long.MAX\_VALUE;

/\*

\* regionInfo is only used as a hack to compare HSKs.

\* It is not serialized. See https://issues.apache.org/jira/browse/HBASE-832

\*/

private HRegionInfo regionInfo = null;

/\*\*

\* Estimated size tax paid for each instance of HSK. Estimate based on

\* study of jhat and jprofiler numbers.

\*/

// In jprofiler, says shallow size is 48 bytes. Add to it cost of two

// byte arrays and then something for the HRI hosting.

public static final int ESTIMATED\_HEAP\_TAX = 48;

/\*\* Default constructor used in conjunction with Writable interface \*/

public HStoreKey() {

super();

}

/\*\*

\* Create an HStoreKey specifying only the row

\* The column defaults to the empty string, the time stamp defaults to

\* Long.MAX\_VALUE and the table defaults to empty string

\*

\* @param row - row key

\*/

public HStoreKey(final byte [] row) {

this(row, Long.MAX\_VALUE);

}

/\*\*

\* Create an HStoreKey specifying only the row

\* The column defaults to the empty string, the time stamp defaults to

\* Long.MAX\_VALUE and the table defaults to empty string

\*

\* @param row - row key

\*/

public HStoreKey(final String row) {

this(row, Long.MAX\_VALUE);

}

/\*\*

\* Create an HStoreKey specifying the row and timestamp

\* The column and table names default to the empty string

\*

\* @param row row key

\* @param hri

\*/

public HStoreKey(final byte [] row, final HRegionInfo hri) {

this(row, HConstants.EMPTY\_BYTE\_ARRAY, hri);

}

/\*\*

\* Create an HStoreKey specifying the row and timestamp

\* The column and table names default to the empty string

\*

\* @param row row key

\* @param timestamp timestamp value

\* @param hri HRegionInfo

\*/

public HStoreKey(final byte [] row, long timestamp, final HRegionInfo hri) {

this(row, HConstants.EMPTY\_BYTE\_ARRAY, timestamp, hri);

}

/\*\*

\* Create an HStoreKey specifying the row and timestamp

\* The column and table names default to the empty string

\*

\* @param row row key

\* @param timestamp timestamp value

\*/

public HStoreKey(final byte [] row, long timestamp) {

this(row, HConstants.EMPTY\_BYTE\_ARRAY, timestamp);

}

/\*\*

\* Create an HStoreKey specifying the row and timestamp

\* The column and table names default to the empty string

\*

\* @param row row key

\* @param timestamp timestamp value

\*/

public HStoreKey(final String row, long timestamp) {

this (row, "", timestamp, new HRegionInfo());

}

/\*\*

\* Create an HStoreKey specifying the row and column names

\* The timestamp defaults to LATEST\_TIMESTAMP

\* and table name defaults to the empty string

\*

\* @param row row key

\* @param column column key

\*/

public HStoreKey(final String row, final String column) {

this(row, column, HConstants.LATEST\_TIMESTAMP, new HRegionInfo());

}

/\*\*

\* Create an HStoreKey specifying the row and column names

\* The timestamp defaults to LATEST\_TIMESTAMP

\* and table name defaults to the empty string

\*

\* @param row row key

\* @param column column key

\*/

public HStoreKey(final byte [] row, final byte [] column) {

this(row, column, HConstants.LATEST\_TIMESTAMP);

}

/\*\*

\* Create an HStoreKey specifying the row, column names and table name

\* The timestamp defaults to LATEST\_TIMESTAMP

\*

\* @param row row key

\* @param column column key

\* @param regionInfo region info

\*/

public HStoreKey(final byte [] row,

final byte [] column, final HRegionInfo regionInfo) {

this(row, column, HConstants.LATEST\_TIMESTAMP, regionInfo);

}

/\*\*

\* Create an HStoreKey specifying all the fields

\* Does not make copies of the passed byte arrays. Presumes the passed

\* arrays immutable.

\* @param row row key

\* @param column column key

\* @param timestamp timestamp value

\* @param regionInfo region info

\*/

public HStoreKey(final String row,

final String column, long timestamp, final HRegionInfo regionInfo) {

this (Bytes.toBytes(row), Bytes.toBytes(column),

timestamp, regionInfo);

}

/\*\*

\* Create an HStoreKey specifying all the fields with unspecified table

\* Does not make copies of the passed byte arrays. Presumes the passed

\* arrays immutable.

\* @param row row key

\* @param column column key

\* @param timestamp timestamp value

\*/

public HStoreKey(final byte [] row, final byte [] column, long timestamp) {

this(row, column, timestamp, null);

}

/\*\*

\* Create an HStoreKey specifying all the fields with specified table

\* Does not make copies of the passed byte arrays. Presumes the passed

\* arrays immutable.

\* @param row row key

\* @param column column key

\* @param timestamp timestamp value

\* @param regionInfo region info

\*/

public HStoreKey(final byte [] row,

final byte [] column, long timestamp, final HRegionInfo regionInfo) {

// Make copies

this.row = row;

this.column = column;

this.timestamp = timestamp;

this.regionInfo = regionInfo;

}

/\*\*

\* Constructs a new HStoreKey from another

\*

\* @param other the source key

\*/

public HStoreKey(HStoreKey other) {

this(other.getRow(), other.getColumn(), other.getTimestamp(),

other.getHRegionInfo());

}

/\*\*

\* Change the value of the row key

\*

\* @param newrow new row key value

\*/

public void setRow(byte [] newrow) {

this.row = newrow;

}

/\*\*

\* Change the value of the column in this key

\*

\* @param c new column family value

\*/

public void setColumn(byte [] c) {

this.column = c;

}

/\*\*

\* Change the value of the timestamp field

\*

\* @param timestamp new timestamp value

\*/

public void setVersion(long timestamp) {

this.timestamp = timestamp;

}

/\*\*

\* Set the value of this HStoreKey from the supplied key

\*

\* @param k key value to copy

\*/

public void set(HStoreKey k) {

this.row = k.getRow();

this.column = k.getColumn();

this.timestamp = k.getTimestamp();

}

/\*\* @return value of row key \*/

public byte [] getRow() {

return row;

}

/\*\* @return value of column \*/

public byte [] getColumn() {

return this.column;

}

/\*\* @return value of timestamp \*/

public long getTimestamp() {

return this.timestamp;

}

/\*\* @return value of regioninfo \*/

public HRegionInfo getHRegionInfo() {

return this.regionInfo;

}

/\*\*

\* @param hri

\*/

public void setHRegionInfo(final HRegionInfo hri) {

this.regionInfo = hri;

}

/\*\*

\* Compares the row and column of two keys

\* @param other Key to compare against. Compares row and column.

\* @return True if same row and column.

\* @see #matchesWithoutColumn(HStoreKey)

\* @see #matchesRowFamily(HStoreKey)

\*/

public boolean matchesRowCol(HStoreKey other) {

return HStoreKey.equalsTwoRowKeys(getHRegionInfo(), getRow(), other.getRow()) &&

Bytes.equals(getColumn(), other.getColumn());

}

/\*\*

\* Compares the row and timestamp of two keys

\*

\* @param other Key to copmare against. Compares row and timestamp.

\*

\* @return True if same row and timestamp is greater than <code>other</code>

\* @see #matchesRowCol(HStoreKey)

\* @see #matchesRowFamily(HStoreKey)

\*/

public boolean matchesWithoutColumn(HStoreKey other) {

return equalsTwoRowKeys(getHRegionInfo(), getRow(), other.getRow()) &&

getTimestamp() >= other.getTimestamp();

}

/\*\*

\* Compares the row and column family of two keys

\*

\* @param that Key to compare against. Compares row and column family

\*

\* @return true if same row and column family

\* @see #matchesRowCol(HStoreKey)

\* @see #matchesWithoutColumn(HStoreKey)

\*/

public boolean matchesRowFamily(HStoreKey that) {

int delimiterIndex = getFamilyDelimiterIndex(getColumn());

return equalsTwoRowKeys(getHRegionInfo(), getRow(), that.getRow()) &&

Bytes.compareTo(getColumn(), 0, delimiterIndex, that.getColumn(), 0,

delimiterIndex) == 0;

}

@Override

public String toString() {

return Bytes.toString(this.row) + "/" + Bytes.toString(this.column) + "/" +

timestamp;

}

@Override

public boolean equals(Object obj) {

HStoreKey other = (HStoreKey)obj;

// Do a quick check.

if (this.row.length != other.row.length ||

this.column.length != other.column.length ||

this.timestamp != other.timestamp) {

return false;

}

return compareTo(other) == 0;

}

@Override

public int hashCode() {

int result = Bytes.hashCode(getRow());

result ^= Bytes.hashCode(getColumn());

result ^= getTimestamp();

return result;

}

// Comparable

public int compareTo(final HStoreKey o) {

return compareTo(this.regionInfo, this, (HStoreKey)o);

}

static int compareTo(final HRegionInfo hri, final HStoreKey left,

final HStoreKey right) {

// We can be passed null

if (left == null && right == null) return 0;

if (left == null) return -1;

if (right == null) return 1;

int result = compareTwoRowKeys(hri, left.getRow(), right.getRow());

if (result != 0) {

return result;

}

result = left.getColumn() == null && right.getColumn() == null? 0:

left.getColumn() == null && right.getColumn() != null? -1:

left.getColumn() != null && right.getColumn() == null? 1:

Bytes.compareTo(left.getColumn(), right.getColumn());

if (result != 0) {

return result;

}

// The below older timestamps sorting ahead of newer timestamps looks

// wrong but it is intentional. This way, newer timestamps are first

// found when we iterate over a memcache and newer versions are the

// first we trip over when reading from a store file.

if (left.getTimestamp() < right.getTimestamp()) {

result = 1;

} else if (left.getTimestamp() > right.getTimestamp()) {

result = -1;

}

// Because of HBASE-877, our BeforeThisStoreKey trick no longer works in

// mapfiles and so instead we need to do this weird check here below.

return result == 0 && left instanceof BeforeThisStoreKey? -1:

result == 0 && right instanceof BeforeThisStoreKey? 1:

result;

}

/\*\*

\* @param column

\* @return New byte array that holds <code>column</code> family prefix only

\* (Does not include the colon DELIMITER).

\* @throws ColumnNameParseException

\* @see #parseColumn(byte[])

\*/

public static byte [] getFamily(final byte [] column)

throws ColumnNameParseException {

int index = getFamilyDelimiterIndex(column);

if (index <= 0) {

throw new ColumnNameParseException("Missing ':' delimiter between " +

"column family and qualifier in the passed column name <" +

Bytes.toString(column) + ">");

}

byte [] result = new byte[index];

System.arraycopy(column, 0, result, 0, index);

return result;

}

/\*\*

\* @param column

\* @return Return hash of family portion of passed column.

\*/

public static Integer getFamilyMapKey(final byte [] column) {

int index = getFamilyDelimiterIndex(column);

// If index < -1, presume passed column is a family name absent colon

// delimiter

return Bytes.mapKey(column, index > 0? index: column.length);

}

/\*\*

\* @param family

\* @param column

\* @return True if <code>column</code> has a family of <code>family</code>.

\*/

public static boolean matchingFamily(final byte [] family,

final byte [] column) {

// Make sure index of the ':' is at same offset.

int index = getFamilyDelimiterIndex(column);

if (index != family.length) {

return false;

}

return Bytes.compareTo(family, 0, index, column, 0, index) == 0;

}

/\*\*

\* @param family

\* @return Return <code>family</code> plus the family delimiter.

\*/

public static byte [] addDelimiter(final byte [] family) {

// Manufacture key by adding delimiter to the passed in colFamily.

byte [] familyPlusDelimiter = new byte [family.length + 1];

System.arraycopy(family, 0, familyPlusDelimiter, 0, family.length);

familyPlusDelimiter[family.length] = HStoreKey.COLUMN\_FAMILY\_DELIMITER;

return familyPlusDelimiter;

}

/\*\*

\* @param column

\* @return New byte array that holds <code>column</code> qualifier suffix.

\* @see #parseColumn(byte[])

\*/

public static byte [] getQualifier(final byte [] column) {

int index = getFamilyDelimiterIndex(column);

int len = column.length - (index + 1);

byte [] result = new byte[len];

System.arraycopy(column, index + 1, result, 0, len);

return result;

}

/\*\*

\* @param c Column name

\* @return Return array of size two whose first element has the family

\* prefix of passed column <code>c</code> and whose second element is the

\* column qualifier.

\* @throws ColumnNameParseException

\*/

public static byte [][] parseColumn(final byte [] c)

throws ColumnNameParseException {

byte [][] result = new byte [2][];

int index = getFamilyDelimiterIndex(c);

if (index == -1) {

throw new ColumnNameParseException("Impossible column name: " + c);

}

result[0] = new byte [index];

System.arraycopy(c, 0, result[0], 0, index);

int len = c.length - (index + 1);

result[1] = new byte[len];

System.arraycopy(c, index + 1 /\*Skip delimiter\*/, result[1], 0,

len);

return result;

}

/\*\*

\* @param b

\* @return Index of the family-qualifier colon delimiter character in passed

\* buffer.

\*/

public static int getFamilyDelimiterIndex(final byte [] b) {

if (b == null) {

throw new NullPointerException();

}

int result = -1;

for (int i = 0; i < b.length; i++) {

if (b[i] == COLUMN\_FAMILY\_DELIMITER) {

result = i;

break;

}

}

return result;

}

/\*\*

\* Returns row and column bytes out of an HStoreKey.

\* @param hsk Store key.

\* @return byte array encoding of HStoreKey

\*/

public static byte[] getBytes(final HStoreKey hsk) {

return Bytes.add(hsk.getRow(), hsk.getColumn());

}

/\*\*

\* Utility method to compare two row keys.

\* This is required because of the meta delimiters.

\* This is a hack.

\* @param regionInfo

\* @param rowA

\* @param rowB

\* @return value of the comparison

\*/

public static int compareTwoRowKeys(HRegionInfo regionInfo,

byte[] rowA, byte[] rowB) {

if (regionInfo != null && regionInfo.isMetaRegion()) {

byte[][] keysA = stripStartKeyMeta(rowA);

byte[][] KeysB = stripStartKeyMeta(rowB);

int rowCompare = Bytes.compareTo(keysA[0], KeysB[0]);

if(rowCompare == 0)

rowCompare = Bytes.compareTo(keysA[1], KeysB[1]);

return rowCompare;

}

return Bytes.compareTo(rowA, rowB);

}

/\*\*

\* Utility method to check if two row keys are equal.

\* This is required because of the meta delimiters

\* This is a hack

\* @param regionInfo

\* @param rowA

\* @param rowB

\* @return if it's equal

\*/

public static boolean equalsTwoRowKeys(HRegionInfo regionInfo,

byte[] rowA, byte[] rowB) {

return rowA == null && rowB == null? true:

rowA == null && rowB != null? false:

rowA != null && rowB == null? false:

rowA.length != rowB.length? false:

compareTwoRowKeys(regionInfo,rowA,rowB) == 0;

}

private static byte[][] stripStartKeyMeta(byte[] rowKey) {

int offset = -1;

for (int i = rowKey.length - 1; i > 0; i--) {

if (rowKey[i] == HConstants.META\_ROW\_DELIMITER) {

offset = i;

break;

}

}

byte [] row = rowKey;

byte [] timestamp = HConstants.EMPTY\_BYTE\_ARRAY;

if (offset != -1) {

row = new byte[offset];

System.arraycopy(rowKey, 0, row, 0, offset);

timestamp = new byte[rowKey.length - offset - 1];

System.arraycopy(rowKey, offset+1, timestamp, 0,rowKey.length - offset - 1);

}

byte[][] elements = new byte[2][];

elements[0] = row;

elements[1] = timestamp;

return elements;

}

// Writable

public void write(DataOutput out) throws IOException {

Bytes.writeByteArray(out, this.row);

Bytes.writeByteArray(out, this.column);

out.writeLong(timestamp);

}

public void readFields(DataInput in) throws IOException {

this.row = Bytes.readByteArray(in);

this.column = Bytes.readByteArray(in);

this.timestamp = in.readLong();

}

public long heapSize() {

return getRow().length + Bytes.ESTIMATED\_HEAP\_TAX +

getColumn().length + Bytes.ESTIMATED\_HEAP\_TAX +

ESTIMATED\_HEAP\_TAX;

}

/\*\*

\* Passed as comparator for memcache and for store files. See HBASE-868.

\*/

public static class HStoreKeyWritableComparator extends WritableComparator {

private final HRegionInfo hri;

/\*\* @param hri \*/

public HStoreKeyWritableComparator(final HRegionInfo hri) {

super(HStoreKey.class);

this.hri = hri;

}

@Override

public int compare(final WritableComparable left,

final WritableComparable right) {

return compareTo(this.hri, (HStoreKey)left, (HStoreKey)right);

}

}

/\*\*

\* Pass this class into {@link org.apache.hadoop.io.MapFile}.getClosest when

\* searching for the key that comes BEFORE this one but NOT this one. This

\* class will return > 0 when asked to compare against itself rather than 0.

\* This is a hack for case where getClosest returns a deleted key and we want

\* to get the previous. Can't unless use use this class; it'll just keep

\* returning us the deleted key (getClosest gets exact or nearest before when

\* you pass true argument). TODO: Throw this class away when MapFile has

\* a real 'previous' method. See HBASE-751.

\*/

public static class BeforeThisStoreKey extends HStoreKey {

private final HStoreKey beforeThisKey;

/\*\*

\* @param beforeThisKey

\*/

public BeforeThisStoreKey(final HStoreKey beforeThisKey) {

super();

this.beforeThisKey = beforeThisKey;

}

@Override

public int compareTo(final HStoreKey o) {

int result = this.beforeThisKey.compareTo(o);

return result == 0? -1: result;

}

@Override

public boolean equals(@SuppressWarnings("unused") Object obj) {

return false;

}

@Override

public byte[] getColumn() {

return this.beforeThisKey.getColumn();

}

@Override

public byte[] getRow() {

return this.beforeThisKey.getRow();

}

@Override

public long heapSize() {

return this.beforeThisKey.heapSize();

}

@Override

public long getTimestamp() {

return this.beforeThisKey.getTimestamp();

}

@Override

public int hashCode() {

return this.beforeThisKey.hashCode();

}

@Override

public boolean matchesRowCol(HStoreKey other) {

return this.beforeThisKey.matchesRowCol(other);

}

@Override

public boolean matchesRowFamily(HStoreKey that) {

return this.beforeThisKey.matchesRowFamily(that);

}

@Override

public boolean matchesWithoutColumn(HStoreKey other) {

return this.beforeThisKey.matchesWithoutColumn(other);

}

@Override

public void readFields(DataInput in) throws IOException {

this.beforeThisKey.readFields(in);

}

@Override

public void set(HStoreKey k) {

this.beforeThisKey.set(k);

}

@Override

public void setColumn(byte[] c) {

this.beforeThisKey.setColumn(c);

}

@Override

public void setRow(byte[] newrow) {

this.beforeThisKey.setRow(newrow);

}

@Override

public void setVersion(long timestamp) {

this.beforeThisKey.setVersion(timestamp);

}

@Override

public String toString() {

return this.beforeThisKey.toString();

}

@Override

public void write(DataOutput out) throws IOException {

this.beforeThisKey.write(out);

}

@Override

public HRegionInfo getHRegionInfo() {

return this.beforeThisKey.getHRegionInfo();

}

@Override

public void setHRegionInfo(final HRegionInfo hri) {

this.beforeThisKey.setHRegionInfo(hri);

}

}

}

package org.apache.hadoop.hbase;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.util.Collection;

import java.util.Collections;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.client.tableindexed.IndexSpecification;

import org.apache.hadoop.hbase.io.ImmutableBytesWritable;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.io.WritableComparable;

/\*\*

\* HTableDescriptor contains the name of an HTable, and its

\* column families.

\*/

public class HTableDescriptor implements WritableComparable<HTableDescriptor> {

// Changes prior to version 3 were not recorded here.

// Version 3 adds metadata as a map where keys and values are byte[].

// Version 4 adds indexes

public static final byte TABLE\_DESCRIPTOR\_VERSION = 4;

private byte [] name = HConstants.EMPTY\_BYTE\_ARRAY;

private String nameAsString = "";

// Table metadata

protected Map<ImmutableBytesWritable, ImmutableBytesWritable> values =

new HashMap<ImmutableBytesWritable, ImmutableBytesWritable>();

public static final String FAMILIES = "FAMILIES";

public static final ImmutableBytesWritable FAMILIES\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(FAMILIES));

public static final String MAX\_FILESIZE = "MAX\_FILESIZE";

public static final ImmutableBytesWritable MAX\_FILESIZE\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(MAX\_FILESIZE));

public static final String READONLY = "READONLY";

public static final ImmutableBytesWritable READONLY\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(READONLY));

public static final String MEMCACHE\_FLUSHSIZE = "MEMCACHE\_FLUSHSIZE";

public static final ImmutableBytesWritable MEMCACHE\_FLUSHSIZE\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(MEMCACHE\_FLUSHSIZE));

public static final String IS\_ROOT = "IS\_ROOT";

public static final ImmutableBytesWritable IS\_ROOT\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(IS\_ROOT));

public static final String IS\_META = "IS\_META";

public static final ImmutableBytesWritable IS\_META\_KEY =

new ImmutableBytesWritable(Bytes.toBytes(IS\_META));

// The below are ugly but better than creating them each time till we

// replace booleans being saved as Strings with plain booleans. Need a

// migration script to do this. TODO.

private static final ImmutableBytesWritable FALSE =

new ImmutableBytesWritable(Bytes.toBytes(Boolean.FALSE.toString()));

private static final ImmutableBytesWritable TRUE =

new ImmutableBytesWritable(Bytes.toBytes(Boolean.TRUE.toString()));

public static final boolean DEFAULT\_IN\_MEMORY = false;

public static final boolean DEFAULT\_READONLY = false;

public static final int DEFAULT\_MEMCACHE\_FLUSH\_SIZE = 1024\*1024\*64;

private volatile Boolean meta = null;

private volatile Boolean root = null;

// Key is hash of the family name.

private final Map<Integer, HColumnDescriptor> families =

new HashMap<Integer, HColumnDescriptor>();

// Key is indexId

private final Map<String, IndexSpecification> indexes =

new HashMap<String, IndexSpecification>();

/\*\*

\* Private constructor used internally creating table descriptors for

\* catalog tables: e.g. .META. and -ROOT-.

\*/

protected HTableDescriptor(final byte [] name, HColumnDescriptor[] families) {

this.name = name.clone();

this.nameAsString = Bytes.toString(this.name);

setMetaFlags(name);

for(HColumnDescriptor descriptor : families) {

this.families.put(Bytes.mapKey(descriptor.getName()), descriptor);

}

}

/\*\*

\* Private constructor used internally creating table descriptors for

\* catalog tables: e.g. .META. and -ROOT-.

\*/

protected HTableDescriptor(final byte [] name, HColumnDescriptor[] families,

Collection<IndexSpecification> indexes,

Map<ImmutableBytesWritable,ImmutableBytesWritable> values) {

this.name = name.clone();

this.nameAsString = Bytes.toString(this.name);

setMetaFlags(name);

for(HColumnDescriptor descriptor : families) {

this.families.put(Bytes.mapKey(descriptor.getName()), descriptor);

}

for(IndexSpecification index : indexes) {

this.indexes.put(index.getIndexId(), index);

}

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> entry:

values.entrySet()) {

this.values.put(entry.getKey(), entry.getValue());

}

}

/\*\*

\* Constructs an empty object.

\* For deserializing an HTableDescriptor instance only.

\* @see #HTableDescriptor(byte[])

\*/

public HTableDescriptor() {

super();

}

/\*\*

\* Constructor.

\* @param name Table name.

\* @throws IllegalArgumentException if passed a table name

\* that is made of other than 'word' characters, underscore or period: i.e.

\* <code>[a-zA-Z\_0-9.].

\* @see <a href="HADOOP-1581">HADOOP-1581 HBASE: Un-openable tablename bug</a>

\*/

public HTableDescriptor(final String name) {

this(Bytes.toBytes(name));

}

/\*\*

\* Constructor.

\* @param name Table name.

\* @throws IllegalArgumentException if passed a table name

\* that is made of other than 'word' characters, underscore or period: i.e.

\* <code>[a-zA-Z\_0-9-.].

\* @see <a href="HADOOP-1581">HADOOP-1581 HBASE: Un-openable tablename bug</a>

\*/

public HTableDescriptor(final byte [] name) {

super();

setMetaFlags(this.name);

this.name = this.isMetaRegion()? name: isLegalTableName(name);

this.nameAsString = Bytes.toString(this.name);

}

/\*\*

\* Constructor.

\* <p>

\* Makes a deep copy of the supplied descriptor.

\* Can make a modifiable descriptor from an UnmodifyableHTableDescriptor.

\* @param desc The descriptor.

\*/

public HTableDescriptor(final HTableDescriptor desc) {

super();

this.name = desc.name.clone();

this.nameAsString = Bytes.toString(this.name);

setMetaFlags(this.name);

for (HColumnDescriptor c: desc.families.values()) {

this.families.put(Bytes.mapKey(c.getName()), new HColumnDescriptor(c));

}

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

desc.values.entrySet()) {

this.values.put(e.getKey(), e.getValue());

}

this.indexes.putAll(desc.indexes);

}

/\*

\* Set meta flags on this table.

\* Called by constructors.

\* @param name

\*/

private void setMetaFlags(final byte [] name) {

setRootRegion(Bytes.equals(name, HConstants.ROOT\_TABLE\_NAME));

setMetaRegion(isRootRegion() ||

Bytes.equals(name, HConstants.META\_TABLE\_NAME));

}

/\*\* @return true if this is the root region \*/

public boolean isRootRegion() {

if (this.root == null) {

this.root = isSomething(IS\_ROOT\_KEY, false)? Boolean.TRUE: Boolean.FALSE;

}

return this.root.booleanValue();

}

/\*\* @param isRoot true if this is the root region \*/

protected void setRootRegion(boolean isRoot) {

// TODO: Make the value a boolean rather than String of boolean.

values.put(IS\_ROOT\_KEY, isRoot? TRUE: FALSE);

}

/\*\* @return true if this is a meta region (part of the root or meta tables) \*/

public boolean isMetaRegion() {

if (this.meta == null) {

this.meta = calculateIsMetaRegion();

}

return this.meta.booleanValue();

}

private synchronized Boolean calculateIsMetaRegion() {

byte [] value = getValue(IS\_META\_KEY);

return (value != null)? new Boolean(Bytes.toString(value)): Boolean.FALSE;

}

private boolean isSomething(final ImmutableBytesWritable key,

final boolean valueIfNull) {

byte [] value = getValue(key);

if (value != null) {

// TODO: Make value be a boolean rather than String of boolean.

return Boolean.valueOf(Bytes.toString(value)).booleanValue();

}

return valueIfNull;

}

/\*\*

\* @param isMeta true if this is a meta region (part of the root or meta

\* tables) \*/

protected void setMetaRegion(boolean isMeta) {

values.put(IS\_META\_KEY, isMeta? TRUE: FALSE);

}

/\*\* @return true if table is the meta table \*/

public boolean isMetaTable() {

return isMetaRegion() && !isRootRegion();

}

/\*\*

\* Check passed buffer is legal user-space table name.

\* @param b Table name.

\* @return Returns passed <code>b</code> param

\* @throws NullPointerException If passed <code>b</code> is null

\* @throws IllegalArgumentException if passed a table name

\* that is made of other than 'word' characters or underscores: i.e.

\* <code>[a-zA-Z\_0-9].

\*/

public static byte [] isLegalTableName(final byte [] b) {

if (b == null || b.length <= 0) {

throw new IllegalArgumentException("Name is null or empty");

}

if (b[0] == '.' || b[0] == '-') {

throw new IllegalArgumentException("Illegal first character <" + b[0] +

">. " + "User-space table names can only start with 'word " +

"characters': i.e. [a-zA-Z\_0-9]: " + Bytes.toString(b));

}

for (int i = 0; i < b.length; i++) {

if (Character.isLetterOrDigit(b[i]) || b[i] == '\_' || b[i] == '-' ||

b[i] == '.') {

continue;

}

throw new IllegalArgumentException("Illegal character <" + b[i] + ">. " +

"User-space table names can only contain 'word characters':" +

"i.e. [a-zA-Z\_0-9-.]: " + Bytes.toString(b));

}

return b;

}

/\*\*

\* @param key The key.

\* @return The value.

\*/

public byte[] getValue(byte[] key) {

return getValue(new ImmutableBytesWritable(key));

}

private byte[] getValue(final ImmutableBytesWritable key) {

ImmutableBytesWritable ibw = values.get(key);

if (ibw == null)

return null;

return ibw.get();

}

/\*\*

\* @param key The key.

\* @return The value as a string.

\*/

public String getValue(String key) {

byte[] value = getValue(Bytes.toBytes(key));

if (value == null)

return null;

return Bytes.toString(value);

}

/\*\*

\* @return All values.

\*/

public Map<ImmutableBytesWritable,ImmutableBytesWritable> getValues() {

return Collections.unmodifiableMap(values);

}

/\*\*

\* @param key The key.

\* @param value The value.

\*/

public void setValue(byte[] key, byte[] value) {

setValue(new ImmutableBytesWritable(key), value);

}

/\*

\* @param key The key.

\* @param value The value.

\*/

private void setValue(final ImmutableBytesWritable key,

final byte[] value) {

values.put(key, new ImmutableBytesWritable(value));

}

/\*

\* @param key The key.

\* @param value The value.

\*/

private void setValue(final ImmutableBytesWritable key,

final ImmutableBytesWritable value) {

values.put(key, value);

}

/\*\*

\* @param key The key.

\* @param value The value.

\*/

public void setValue(String key, String value) {

setValue(Bytes.toBytes(key), Bytes.toBytes(value));

}

/\*\*

\* @return true if all columns in the table should be kept in the

\* HRegionServer cache only

\*/

public boolean isInMemory() {

String value = getValue(HConstants.IN\_MEMORY);

if (value != null)

return Boolean.valueOf(value).booleanValue();

return DEFAULT\_IN\_MEMORY;

}

/\*\*

\* @param inMemory True if all of the columns in the table should be kept in

\* the HRegionServer cache only.

\*/

public void setInMemory(boolean inMemory) {

setValue(HConstants.IN\_MEMORY, Boolean.toString(inMemory));

}

/\*\*

\* @return true if all columns in the table should be read only

\*/

public boolean isReadOnly() {

return isSomething(READONLY\_KEY, DEFAULT\_READONLY);

}

/\*\*

\* @param readOnly True if all of the columns in the table should be read

\* only.

\*/

public void setReadOnly(final boolean readOnly) {

setValue(READONLY\_KEY, readOnly? TRUE: FALSE);

}

/\*\* @return name of table \*/

public byte [] getName() {

return name;

}

/\*\* @return name of table \*/

public String getNameAsString() {

return this.nameAsString;

}

/\*\* @return max hregion size for table \*/

public long getMaxFileSize() {

byte [] value = getValue(MAX\_FILESIZE\_KEY);

if (value != null)

return Long.valueOf(Bytes.toString(value)).longValue();

return HConstants.DEFAULT\_MAX\_FILE\_SIZE;

}

/\*\*

\* @param maxFileSize The maximum file size that a store file can grow to

\* before a split is triggered.

\*/

public void setMaxFileSize(long maxFileSize) {

setValue(MAX\_FILESIZE\_KEY, Bytes.toBytes(Long.toString(maxFileSize)));

}

/\*\*

\* @return memory cache flush size for each hregion

\*/

public int getMemcacheFlushSize() {

byte [] value = getValue(MEMCACHE\_FLUSHSIZE\_KEY);

if (value != null)

return Integer.valueOf(Bytes.toString(value)).intValue();

return DEFAULT\_MEMCACHE\_FLUSH\_SIZE;

}

/\*\*

\* @param memcacheFlushSize memory cache flush size for each hregion

\*/

public void setMemcacheFlushSize(int memcacheFlushSize) {

setValue(MEMCACHE\_FLUSHSIZE\_KEY,

Bytes.toBytes(Integer.toString(memcacheFlushSize)));

}

public Collection<IndexSpecification> getIndexes() {

return indexes.values();

}

public IndexSpecification getIndex(String indexId) {

return indexes.get(indexId);

}

public void addIndex(IndexSpecification index) {

indexes.put(index.getIndexId(), index);

}

/\*\*

\* Adds a column family.

\* @param family HColumnDescriptor of familyto add.

\*/

public void addFamily(final HColumnDescriptor family) {

if (family.getName() == null || family.getName().length <= 0) {

throw new NullPointerException("Family name cannot be null or empty");

}

this.families.put(Bytes.mapKey(family.getName()), family);

}

/\*\*

\* Checks to see if this table contains the given column family

\* @param c Family name or column name.

\* @return true if the table contains the specified family name

\*/

public boolean hasFamily(final byte [] c) {

return hasFamily(c, HStoreKey.getFamilyDelimiterIndex(c));

}

/\*\*

\* Checks to see if this table contains the given column family

\* @param c Family name or column name.

\* @param index Index to column family delimiter

\* @return true if the table contains the specified family name

\*/

public boolean hasFamily(final byte [] c, final int index) {

// If index is -1, then presume we were passed a column family name minus

// the colon delimiter.

return families.containsKey(Bytes.mapKey(c, index == -1? c.length: index));

}

/\*\*

\* @return Name of this table and then a map of all of the column family

\* descriptors.

\* @see #getNameAsString()

\*/

@Override

public String toString() {

StringBuffer s = new StringBuffer();

s.append('{');

s.append(HConstants.NAME);

s.append(" => '");

s.append(Bytes.toString(name));

s.append("'");

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

values.entrySet()) {

s.append(", ");

s.append(Bytes.toString(e.getKey().get()));

s.append(" => '");

s.append(Bytes.toString(e.getValue().get()));

s.append("'");

}

s.append(", ");

s.append(FAMILIES);

s.append(" => ");

s.append(families.values());

s.append(", ");

s.append("INDEXES");

s.append(" => ");

s.append(indexes.values());

s.append('}');

return s.toString();

}

@Override

public boolean equals(Object obj) {

return compareTo((HTableDescriptor)obj) == 0;

}

@Override

public int hashCode() {

int result = Bytes.hashCode(this.name);

result ^= Byte.valueOf(TABLE\_DESCRIPTOR\_VERSION).hashCode();

if (this.families != null && this.families.size() > 0) {

for (HColumnDescriptor e: this.families.values()) {

result ^= e.hashCode();

}

}

result ^= values.hashCode();

return result;

}

// Writable

public void readFields(DataInput in) throws IOException {

int version = in.readInt();

if (version < 3)

throw new IOException("versions < 3 are not supported (and never existed!?)");

// version 3+

name = Bytes.readByteArray(in);

nameAsString = Bytes.toString(this.name);

setRootRegion(in.readBoolean());

setMetaRegion(in.readBoolean());

values.clear();

int numVals = in.readInt();

for (int i = 0; i < numVals; i++) {

ImmutableBytesWritable key = new ImmutableBytesWritable();

ImmutableBytesWritable value = new ImmutableBytesWritable();

key.readFields(in);

value.readFields(in);

values.put(key, value);

}

families.clear();

int numFamilies = in.readInt();

for (int i = 0; i < numFamilies; i++) {

HColumnDescriptor c = new HColumnDescriptor();

c.readFields(in);

families.put(Bytes.mapKey(c.getName()), c);

}

indexes.clear();

if (version < 4) {

return;

}

int numIndexes = in.readInt();

for (int i = 0; i < numIndexes; i++) {

IndexSpecification index = new IndexSpecification();

index.readFields(in);

addIndex(index);

}

}

public void write(DataOutput out) throws IOException {

out.writeInt(TABLE\_DESCRIPTOR\_VERSION);

Bytes.writeByteArray(out, name);

out.writeBoolean(isRootRegion());

out.writeBoolean(isMetaRegion());

out.writeInt(values.size());

for (Map.Entry<ImmutableBytesWritable, ImmutableBytesWritable> e:

values.entrySet()) {

e.getKey().write(out);

e.getValue().write(out);

}

out.writeInt(families.size());

for(Iterator<HColumnDescriptor> it = families.values().iterator();

it.hasNext(); ) {

HColumnDescriptor family = it.next();

family.write(out);

}

out.writeInt(indexes.size());

for(IndexSpecification index : indexes.values()) {

index.write(out);

}

}

// Comparable

public int compareTo(final HTableDescriptor other) {

int result = Bytes.compareTo(this.name, other.name);

if (result == 0) {

result = families.size() - other.families.size();

}

if (result == 0 && families.size() != other.families.size()) {

result = Integer.valueOf(families.size()).compareTo(

Integer.valueOf(other.families.size()));

}

if (result == 0) {

for (Iterator<HColumnDescriptor> it = families.values().iterator(),

it2 = other.families.values().iterator(); it.hasNext(); ) {

result = it.next().compareTo(it2.next());

if (result != 0) {

break;

}

}

}

if (result == 0) {

// punt on comparison for ordering, just calculate difference

result = this.values.hashCode() - other.values.hashCode();

if (result < 0)

result = -1;

else if (result > 0)

result = 1;

}

return result;

}

/\*\*

\* @return Immutable sorted map of families.

\*/

public Collection<HColumnDescriptor> getFamilies() {

return Collections.unmodifiableCollection(this.families.values());

}

/\*\*

\* @param column

\* @return Column descriptor for the passed family name or the family on

\* passed in column.

\*/

public HColumnDescriptor getFamily(final byte [] column) {

return this.families.get(HStoreKey.getFamilyMapKey(column));

}

/\*\*

\* @param column

\* @return Column descriptor for the passed family name or the family on

\* passed in column.

\*/

public HColumnDescriptor removeFamily(final byte [] column) {

return this.families.remove(HStoreKey.getFamilyMapKey(column));

}

/\*\*

\* @param rootdir qualified path of HBase root directory

\* @param tableName name of table

\* @return path for table

\*/

public static Path getTableDir(Path rootdir, final byte [] tableName) {

return new Path(rootdir, Bytes.toString(tableName));

}

/\*\* Table descriptor for <core>-ROOT-</code> catalog table \*/

public static final HTableDescriptor ROOT\_TABLEDESC = new HTableDescriptor(

HConstants.ROOT\_TABLE\_NAME,

new HColumnDescriptor[] { new HColumnDescriptor(HConstants.COLUMN\_FAMILY,

10, // Ten is arbitrary number. Keep versions to help debuggging.

HColumnDescriptor.CompressionType.NONE, false, true,

Integer.MAX\_VALUE, HConstants.FOREVER, false) });

/\*\* Table descriptor for <code>.META.</code> catalog table \*/

public static final HTableDescriptor META\_TABLEDESC = new HTableDescriptor(

HConstants.META\_TABLE\_NAME, new HColumnDescriptor[] {

new HColumnDescriptor(HConstants.COLUMN\_FAMILY,

10, // Ten is arbitrary number. Keep versions to help debuggging.

HColumnDescriptor.CompressionType.NONE, false, true,

Integer.MAX\_VALUE, HConstants.FOREVER, false),

new HColumnDescriptor(HConstants.COLUMN\_FAMILY\_HISTORIAN,

HConstants.ALL\_VERSIONS, HColumnDescriptor.CompressionType.NONE,

false, false, Integer.MAX\_VALUE, HConstants.WEEK\_IN\_SECONDS, false)});

}

package org.apache.hadoop.hbase;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.util.Comparator;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.hadoop.hbase.io.HeapSize;

import org.apache.hadoop.hbase.io.hfile.HFile;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.util.ClassSize;

import org.apache.hadoop.io.RawComparator;

import org.apache.hadoop.io.Writable;

/\*\*

\* An HBase Key/Value.

\*

\* <p>If being used client-side, the primary methods to access individual fields

\* are {@link #getRow()}, {@link #getFamily()}, {@link #getQualifier()},

\* {@link #getTimestamp()}, and {@link #getValue()}. These methods allocate new

\* byte arrays and return copies so they should be avoided server-side.

\*

\* <p>Instances of this class are immutable. They are not

\* comparable but Comparators are provided. Comparators change with context,

\* whether user table or a catalog table comparison context. Its

\* important that you use the appropriate comparator comparing rows in

\* particular. There are Comparators for KeyValue instances and then for

\* just the Key portion of a KeyValue used mostly in {@link HFile}.

\*

\* <p>KeyValue wraps a byte array and has offset and length for passed array

\* at where to start interpreting the content as a KeyValue blob. The KeyValue

\* blob format inside the byte array is:

\* <code>&lt;keylength> &lt;valuelength> &lt;key> &lt;value></code>

\* Key is decomposed as:

\* <code>&lt;rowlength> &lt;row> &lt;columnfamilylength> &lt;columnfamily> &lt;columnqualifier> &lt;timestamp> &lt;keytype></code>

\* Rowlength maximum is Short.MAX\_SIZE, column family length maximum is

\* Byte.MAX\_SIZE, and column qualifier + key length must be < Integer.MAX\_SIZE.

\* The column does not contain the family/qualifier delimiter.

\*

\* <p>TODO: Group Key-only comparators and operations into a Key class, just

\* for neatness sake, if can figure what to call it.

\*/

public class KeyValue implements Writable, HeapSize {

static final Log LOG = LogFactory.getLog(KeyValue.class);

/\*\*

\* Colon character in UTF-8

\*/

public static final char COLUMN\_FAMILY\_DELIMITER = ':';

public static final byte[] COLUMN\_FAMILY\_DELIM\_ARRAY =

new byte[]{COLUMN\_FAMILY\_DELIMITER};

/\*\*

\* Comparator for plain key/values; i.e. non-catalog table key/values.

\*/

public static KVComparator COMPARATOR = new KVComparator();

/\*\*

\* Comparator for plain key; i.e. non-catalog table key. Works on Key portion

\* of KeyValue only.

\*/

public static KeyComparator KEY\_COMPARATOR = new KeyComparator();

/\*\*

\* A {@link KVComparator} for <code>.META.</code> catalog table

\* {@link KeyValue}s.

\*/

public static KVComparator META\_COMPARATOR = new MetaComparator();

/\*\*

\* A {@link KVComparator} for <code>.META.</code> catalog table

\* {@link KeyValue} keys.

\*/

public static KeyComparator META\_KEY\_COMPARATOR = new MetaKeyComparator();

/\*\*

\* A {@link KVComparator} for <code>-ROOT-</code> catalog table

\* {@link KeyValue}s.

\*/

public static KVComparator ROOT\_COMPARATOR = new RootComparator();

/\*\*

\* A {@link KVComparator} for <code>-ROOT-</code> catalog table

\* {@link KeyValue} keys.

\*/

public static KeyComparator ROOT\_KEY\_COMPARATOR = new RootKeyComparator();

/\*\*

\* Get the appropriate row comparator for the specified table.

\*

\* Hopefully we can get rid of this, I added this here because it's replacing

\* something in HSK. We should move completely off of that.

\*

\* @param tableName The table name.

\* @return The comparator.

\*/

public static KeyComparator getRowComparator(byte [] tableName) {

if(Bytes.equals(HTableDescriptor.ROOT\_TABLEDESC.getName(),tableName)) {

return ROOT\_COMPARATOR.getRawComparator();

}

if(Bytes.equals(HTableDescriptor.META\_TABLEDESC.getName(), tableName)) {

return META\_COMPARATOR.getRawComparator();

}

return COMPARATOR.getRawComparator();

}

// Size of the timestamp and type byte on end of a key -- a long + a byte.

public static final int TIMESTAMP\_TYPE\_SIZE =

Bytes.SIZEOF\_LONG /\* timestamp \*/ +

Bytes.SIZEOF\_BYTE /\*keytype\*/;

// Size of the length shorts and bytes in key.

public static final int KEY\_INFRASTRUCTURE\_SIZE =

Bytes.SIZEOF\_SHORT /\*rowlength\*/ +

Bytes.SIZEOF\_BYTE /\*columnfamilylength\*/ +

TIMESTAMP\_TYPE\_SIZE;

// How far into the key the row starts at. First thing to read is the short

// that says how long the row is.

public static final int ROW\_OFFSET =

Bytes.SIZEOF\_INT /\*keylength\*/ +

Bytes.SIZEOF\_INT /\*valuelength\*/;

// Size of the length ints in a KeyValue datastructure.

public static final int KEYVALUE\_INFRASTRUCTURE\_SIZE = ROW\_OFFSET;

/\*\*

\* Key type.

\* Has space for other key types to be added later. Cannot rely on

\* enum ordinals . They change if item is removed or moved. Do our own codes.

\*/

public static enum Type {

Minimum((byte)0),

Put((byte)4),

Delete((byte)8),

DeleteColumn((byte)12),

DeleteFamily((byte)14),

// Maximum is used when searching; you look from maximum on down.

Maximum((byte)255);

private final byte code;

Type(final byte c) {

this.code = c;

}

public byte getCode() {

return this.code;

}

/\*\*

\* Cannot rely on enum ordinals . They change if item is removed or moved.

\* Do our own codes.

\* @param b

\* @return Type associated with passed code.

\*/

public static Type codeToType(final byte b) {

for (Type t : Type.values()) {

if (t.getCode() == b) {

return t;

}

}

throw new RuntimeException("Unknown code " + b);

}

}

/\*\*

\* Lowest possible key.

\* Makes a Key with highest possible Timestamp, empty row and column. No

\* key can be equal or lower than this one in memstore or in store file.

\*/

public static final KeyValue LOWESTKEY =

new KeyValue(HConstants.EMPTY\_BYTE\_ARRAY, HConstants.LATEST\_TIMESTAMP);

private byte [] bytes = null;

private int offset = 0;

private int length = 0;

/\*\* Here be dragons \*\*/

// used to achieve atomic operations in the memstore.

public long getMemstoreTS() {

return memstoreTS;

}

public void setMemstoreTS(long memstoreTS) {

this.memstoreTS = memstoreTS;

}

// default value is 0, aka DNC

private long memstoreTS = 0;

/\*\* Dragon time over, return to normal business \*/

/\*\* Writable Constructor -- DO NOT USE \*/

public KeyValue() {}

/\*\*

\* Creates a KeyValue from the start of the specified byte array.

\* Presumes <code>bytes</code> content is formatted as a KeyValue blob.

\* @param bytes byte array

\*/

public KeyValue(final byte [] bytes) {

this(bytes, 0);

}

/\*\*

\* Creates a KeyValue from the specified byte array and offset.

\* Presumes <code>bytes</code> content starting at <code>offset</code> is

\* formatted as a KeyValue blob.

\* @param bytes byte array

\* @param offset offset to start of KeyValue

\*/

public KeyValue(final byte [] bytes, final int offset) {

this(bytes, offset, getLength(bytes, offset));

}

/\*\*

\* Creates a KeyValue from the specified byte array, starting at offset, and

\* for length <code>length</code>.

\* @param bytes byte array

\* @param offset offset to start of the KeyValue

\* @param length length of the KeyValue

\*/

public KeyValue(final byte [] bytes, final int offset, final int length) {

this.bytes = bytes;

this.offset = offset;

this.length = length;

}

/\*\* Temporary constructors until 880/1249 is committed to remove deps \*/

/\*\*

\* Temporary.

\*/

public KeyValue(final byte [] row, final byte [] column) {

this(row, column, HConstants.LATEST\_TIMESTAMP, null);

}

public KeyValue(final byte [] row, final byte [] column, long ts) {

this(row, column, ts, null);

}

public KeyValue(final byte [] row, final byte [] column, long ts,

byte [] value) {

this(row, column, ts, Type.Put, value);

}

public KeyValue(final byte [] row, final byte [] column, long ts, Type type,

byte [] value) {

int rlength = row == null ? 0 : row.length;

int vlength = value == null ? 0 : value.length;

int clength = column == null ? 0 : column.length;

this.bytes = createByteArray(row, 0, rlength, column, 0, clength,

ts, type, value, 0, vlength);

this.length = this.bytes.length;

this.offset = 0;

}

/\*\* Constructors that build a new backing byte array from fields \*/

/\*\*

\* Constructs KeyValue structure filled with null value.

\* Sets type to {@link KeyValue.Type#Maximum}

\* @param row - row key (arbitrary byte array)

\* @param timestamp

\*/

public KeyValue(final byte [] row, final long timestamp) {

this(row, timestamp, Type.Maximum);

}

/\*\*

\* Constructs KeyValue structure filled with null value.

\* @param row - row key (arbitrary byte array)

\* @param timestamp

\*/

public KeyValue(final byte [] row, final long timestamp, Type type) {

this(row, null, null, timestamp, type, null);

}

/\*\*

\* Constructs KeyValue structure filled with null value.

\* Sets type to {@link KeyValue.Type#Maximum}

\* @param row - row key (arbitrary byte array)

\* @param family family name

\* @param qualifier column qualifier

\*/

public KeyValue(final byte [] row, final byte [] family,

final byte [] qualifier) {

this(row, family, qualifier, HConstants.LATEST\_TIMESTAMP, Type.Maximum);

}

/\*\*

\* Constructs KeyValue structure filled with null value.

\* @param row - row key (arbitrary byte array)

\* @param family family name

\* @param qualifier column qualifier

\*/

public KeyValue(final byte [] row, final byte [] family,

final byte [] qualifier, final byte [] value) {

this(row, family, qualifier, HConstants.LATEST\_TIMESTAMP, Type.Put, value);

}

/\*\*

\* Constructs KeyValue structure filled with specified values.

\* @param row row key

\* @param family family name

\* @param qualifier column qualifier

\* @param timestamp version timestamp

\* @param type key type

\* @throws IllegalArgumentException

\*/

public KeyValue(final byte[] row, final byte[] family,

final byte[] qualifier, final long timestamp, Type type) {

this(row, family, qualifier, timestamp, type, null);

}

/\*\*

\* Constructs KeyValue structure filled with specified values.

\* @param row row key

\* @param family family name

\* @param qualifier column qualifier

\* @param timestamp version timestamp

\* @param value column value

\* @throws IllegalArgumentException

\*/

public KeyValue(final byte[] row, final byte[] family,

final byte[] qualifier, final long timestamp, final byte[] value) {

this(row, family, qualifier, timestamp, Type.Put, value);

}

/\*\*

\* Constructs KeyValue structure filled with specified values.

\* @param row row key

\* @param family family name

\* @param qualifier column qualifier

\* @param timestamp version timestamp

\* @param type key type

\* @param value column value

\* @throws IllegalArgumentException

\*/

public KeyValue(final byte[] row, final byte[] family,

final byte[] qualifier, final long timestamp, Type type,

final byte[] value) {

this(row, family, qualifier, 0, qualifier==null ? 0 : qualifier.length,

timestamp, type, value, 0, value==null ? 0 : value.length);

}

/\*\*

\* Constructs KeyValue structure filled with specified values.

\* @param row row key

\* @param family family name

\* @param qualifier column qualifier

\* @param qoffset qualifier offset

\* @param qlength qualifier length

\* @param timestamp version timestamp

\* @param type key type

\* @param value column value

\* @param voffset value offset

\* @param vlength value length

\* @throws IllegalArgumentException

\*/

public KeyValue(byte [] row, byte [] family,

byte [] qualifier, int qoffset, int qlength, long timestamp, Type type,

byte [] value, int voffset, int vlength) {

this(row, 0, row==null ? 0 : row.length,

family, 0, family==null ? 0 : family.length,

qualifier, qoffset, qlength, timestamp, type,

value, voffset, vlength);

}

/\*\*

\* Constructs KeyValue structure filled with specified values.

\* <p>

\* Column is split into two fields, family and qualifier.

\* @param row row key

\* @param roffset row offset

\* @param rlength row length

\* @param family family name

\* @param foffset family offset

\* @param flength family length

\* @param qualifier column qualifier

\* @param qoffset qualifier offset

\* @param qlength qualifier length

\* @param timestamp version timestamp

\* @param type key type

\* @param value column value

\* @param voffset value offset

\* @param vlength value length

\* @throws IllegalArgumentException

\*/

public KeyValue(final byte [] row, final int roffset, final int rlength,

final byte [] family, final int foffset, final int flength,

final byte [] qualifier, final int qoffset, final int qlength,

final long timestamp, final Type type,

final byte [] value, final int voffset, final int vlength) {

this.bytes = createByteArray(row, roffset, rlength,

family, foffset, flength, qualifier, qoffset, qlength,

timestamp, type, value, voffset, vlength);

this.length = bytes.length;

this.offset = 0;

}

/\*\*

\* Write KeyValue format into a byte array.

\*

\* @param row row key

\* @param roffset row offset

\* @param rlength row length

\* @param family family name

\* @param foffset family offset

\* @param flength family length

\* @param qualifier column qualifier

\* @param qoffset qualifier offset

\* @param qlength qualifier length

\* @param timestamp version timestamp

\* @param type key type

\* @param value column value

\* @param voffset value offset

\* @param vlength value length

\* @return The newly created byte array.

\*/

static byte [] createByteArray(final byte [] row, final int roffset,

final int rlength, final byte [] family, final int foffset, int flength,

final byte [] qualifier, final int qoffset, int qlength,

final long timestamp, final Type type,

final byte [] value, final int voffset, int vlength) {

if (rlength > Short.MAX\_VALUE) {

throw new IllegalArgumentException("Row > " + Short.MAX\_VALUE);

}

if (row == null) {

throw new IllegalArgumentException("Row is null");

}

// Family length

flength = family == null ? 0 : flength;

if (flength > Byte.MAX\_VALUE) {

throw new IllegalArgumentException("Family > " + Byte.MAX\_VALUE);

}

// Qualifier length

qlength = qualifier == null ? 0 : qlength;

if (qlength > Integer.MAX\_VALUE - rlength - flength) {

throw new IllegalArgumentException("Qualifier > " + Integer.MAX\_VALUE);

}

// Key length

long longkeylength = KEY\_INFRASTRUCTURE\_SIZE + rlength + flength + qlength;

if (longkeylength > Integer.MAX\_VALUE) {

throw new IllegalArgumentException("keylength " + longkeylength + " > " +

Integer.MAX\_VALUE);

}

int keylength = (int)longkeylength;

// Value length

vlength = value == null? 0 : vlength;

if (vlength > HConstants.MAXIMUM\_VALUE\_LENGTH) {

throw new IllegalArgumentException("Valuer > " +

HConstants.MAXIMUM\_VALUE\_LENGTH);

}

// Allocate right-sized byte array.

byte [] bytes = new byte[KEYVALUE\_INFRASTRUCTURE\_SIZE + keylength + vlength];

// Write key, value and key row length.

int pos = 0;

pos = Bytes.putInt(bytes, pos, keylength);

pos = Bytes.putInt(bytes, pos, vlength);

pos = Bytes.putShort(bytes, pos, (short)(rlength & 0x0000ffff));

pos = Bytes.putBytes(bytes, pos, row, roffset, rlength);

pos = Bytes.putByte(bytes, pos, (byte)(flength & 0x0000ff));

if(flength != 0) {

pos = Bytes.putBytes(bytes, pos, family, foffset, flength);

}

if(qlength != 0) {

pos = Bytes.putBytes(bytes, pos, qualifier, qoffset, qlength);

}

pos = Bytes.putLong(bytes, pos, timestamp);

pos = Bytes.putByte(bytes, pos, type.getCode());

if (value != null && value.length > 0) {

pos = Bytes.putBytes(bytes, pos, value, voffset, vlength);

}

return bytes;

}

/\*\*

\* Write KeyValue format into a byte array.

\* <p>

\* Takes column in the form <code>family:qualifier</code>

\* @param row - row key (arbitrary byte array)

\* @param roffset

\* @param rlength

\* @param column

\* @param coffset

\* @param clength

\* @param timestamp

\* @param type

\* @param value

\* @param voffset

\* @param vlength

\* @return The newly created byte array.

\*/

static byte [] createByteArray(final byte [] row, final int roffset,

final int rlength,

final byte [] column, final int coffset, int clength,

final long timestamp, final Type type,

final byte [] value, final int voffset, int vlength) {

// If column is non-null, figure where the delimiter is at.

int delimiteroffset = 0;

if (column != null && column.length > 0) {

delimiteroffset = getFamilyDelimiterIndex(column, coffset, clength);

if (delimiteroffset > Byte.MAX\_VALUE) {

throw new IllegalArgumentException("Family > " + Byte.MAX\_VALUE);

}

} else {

return createByteArray(row,roffset,rlength,null,0,0,null,0,0,timestamp,

type,value,voffset,vlength);

}

int flength = delimiteroffset-coffset;

int qlength = clength - flength - 1;

return createByteArray(row, roffset, rlength, column, coffset,

flength, column, delimiteroffset+1, qlength, timestamp, type,

value, voffset, vlength);

}

// Needed doing 'contains' on List. Only compares the key portion, not the

// value.

public boolean equals(Object other) {

KeyValue kv = (KeyValue)other;

// Comparing bytes should be fine doing equals test. Shouldn't have to

// worry about special .META. comparators doing straight equals.

boolean result = Bytes.BYTES\_RAWCOMPARATOR.compare(getBuffer(),

getKeyOffset(), getKeyLength(),

kv.getBuffer(), kv.getKeyOffset(), kv.getKeyLength()) == 0;

return result;

}

//---------------------------------------------------------------------------

//

// KeyValue cloning

//

//---------------------------------------------------------------------------

/\*\*

\* Clones a KeyValue. This creates a copy, re-allocating the buffer.

\* @return Fully copied clone of this KeyValue

\*/

public KeyValue clone() {

byte [] b = new byte[this.length];

System.arraycopy(this.bytes, this.offset, b, 0, this.length);

return new KeyValue(b, 0, b.length);

}

//---------------------------------------------------------------------------

//

// String representation

//

//---------------------------------------------------------------------------

public String toString() {

if (this.bytes == null || this.bytes.length == 0) {

return "empty";

}

return keyToString(this.bytes, this.offset + ROW\_OFFSET, getKeyLength()) +

"/vlen=" + getValueLength();

}

/\*\*

\* @param k Key portion of a KeyValue.

\* @return Key as a String.

\*/

public static String keyToString(final byte [] k) {

return keyToString(k, 0, k.length);

}

/\*\*

\* Use for logging.

\* @param b Key portion of a KeyValue.

\* @param o Offset to start of key

\* @param l Length of key.

\* @return Key as a String.

\*/

public static String keyToString(final byte [] b, final int o, final int l) {

if (b == null) return "";

int rowlength = Bytes.toShort(b, o);

String row = Bytes.toStringBinary(b, o + Bytes.SIZEOF\_SHORT, rowlength);

int columnoffset = o + Bytes.SIZEOF\_SHORT + 1 + rowlength;

int familylength = b[columnoffset - 1];

int columnlength = l - ((columnoffset - o) + TIMESTAMP\_TYPE\_SIZE);

String family = familylength == 0? "":

Bytes.toStringBinary(b, columnoffset, familylength);

String qualifier = columnlength == 0? "":

Bytes.toStringBinary(b, columnoffset + familylength,

columnlength - familylength);

long timestamp = Bytes.toLong(b, o + (l - TIMESTAMP\_TYPE\_SIZE));

byte type = b[o + l - 1];

// return row + "/" + family +

// (family != null && family.length() > 0? COLUMN\_FAMILY\_DELIMITER: "") +

// qualifier + "/" + timestamp + "/" + Type.codeToType(type);

return row + "/" + family +

(family != null && family.length() > 0? ":" :"") +

qualifier + "/" + timestamp + "/" + Type.codeToType(type);

}

//---------------------------------------------------------------------------

//

// Public Member Accessors

//

//---------------------------------------------------------------------------

/\*\*

\* @return The byte array backing this KeyValue.

\*/

public byte [] getBuffer() {

return this.bytes;

}

/\*\*

\* @return Offset into {@link #getBuffer()} at which this KeyValue starts.

\*/

public int getOffset() {

return this.offset;

}

/\*\*

\* @return Length of bytes this KeyValue occupies in {@link #getBuffer()}.

\*/

public int getLength() {

return length;

}

//---------------------------------------------------------------------------

//

// Length and Offset Calculators

//

//---------------------------------------------------------------------------

/\*\*

\* Determines the total length of the KeyValue stored in the specified

\* byte array and offset. Includes all headers.

\* @param bytes byte array

\* @param offset offset to start of the KeyValue

\* @return length of entire KeyValue, in bytes

\*/

private static int getLength(byte [] bytes, int offset) {

return (2 \* Bytes.SIZEOF\_INT) +

Bytes.toInt(bytes, offset) +

Bytes.toInt(bytes, offset + Bytes.SIZEOF\_INT);

}

/\*\*

\* @return Key offset in backing buffer..

\*/

public int getKeyOffset() {

return this.offset + ROW\_OFFSET;

}

public String getKeyString() {

return Bytes.toStringBinary(getBuffer(), getKeyOffset(), getKeyLength());

}

/\*\*

\* @return Length of key portion.

\*/

public int getKeyLength() {

return Bytes.toInt(this.bytes, this.offset);

}

/\*\*

\* @return Value offset

\*/

public int getValueOffset() {

return getKeyOffset() + getKeyLength();

}

/\*\*

\* @return Value length

\*/

public int getValueLength() {

return Bytes.toInt(this.bytes, this.offset + Bytes.SIZEOF\_INT);

}

/\*\*

\* @return Row offset

\*/

public int getRowOffset() {

return getKeyOffset() + Bytes.SIZEOF\_SHORT;

}

/\*\*

\* @return Row length

\*/

public short getRowLength() {

return Bytes.toShort(this.bytes, getKeyOffset());

}

/\*\*

\* @return Family offset

\*/

public int getFamilyOffset() {

return getFamilyOffset(getRowLength());

}

/\*\*

\* @return Family offset

\*/

public int getFamilyOffset(int rlength) {

return this.offset + ROW\_OFFSET + Bytes.SIZEOF\_SHORT + rlength + Bytes.SIZEOF\_BYTE;

}

/\*\*

\* @return Family length

\*/

public byte getFamilyLength() {

return getFamilyLength(getFamilyOffset());

}

/\*\*

\* @return Family length

\*/

public byte getFamilyLength(int foffset) {

return this.bytes[foffset-1];

}

/\*\*

\* @return Qualifier offset

\*/

public int getQualifierOffset() {

return getQualifierOffset(getFamilyOffset());

}

/\*\*

\* @return Qualifier offset

\*/

public int getQualifierOffset(int foffset) {

return foffset + getFamilyLength(foffset);

}

/\*\*

\* @return Qualifier length

\*/

public int getQualifierLength() {

return getQualifierLength(getRowLength(),getFamilyLength());

}

/\*\*

\* @return Qualifier length

\*/

public int getQualifierLength(int rlength, int flength) {

return getKeyLength() -

(KEY\_INFRASTRUCTURE\_SIZE + rlength + flength);

}

/\*\*

\* @return Column (family + qualifier) length

\*/

public int getTotalColumnLength() {

int rlength = getRowLength();

int foffset = getFamilyOffset(rlength);

return getTotalColumnLength(rlength,foffset);

}

/\*\*

\* @return Column (family + qualifier) length

\*/

public int getTotalColumnLength(int rlength, int foffset) {

int flength = getFamilyLength(foffset);

int qlength = getQualifierLength(rlength,flength);

return flength + qlength;

}

/\*\*

\* @return Timestamp offset

\*/

public int getTimestampOffset() {

return getTimestampOffset(getKeyLength());

}

/\*\*

\* @param keylength Pass if you have it to save on a int creation.

\* @return Timestamp offset

\*/

public int getTimestampOffset(final int keylength) {

return getKeyOffset() + keylength - TIMESTAMP\_TYPE\_SIZE;

}

/\*\*

\* @return True if this KeyValue has a LATEST\_TIMESTAMP timestamp.

\*/

public boolean isLatestTimestamp() {

return Bytes.compareTo(getBuffer(), getTimestampOffset(), Bytes.SIZEOF\_LONG,

HConstants.LATEST\_TIMESTAMP\_BYTES, 0, Bytes.SIZEOF\_LONG) == 0;

}

/\*\*

\* @param now Time to set into <code>this</code> IFF timestamp ==

\* {@link HConstants#LATEST\_TIMESTAMP} (else, its a noop).

\* @return True is we modified this.

\*/

public boolean updateLatestStamp(final byte [] now) {

if (this.isLatestTimestamp()) {

int tsOffset = getTimestampOffset();

System.arraycopy(now, 0, this.bytes, tsOffset, Bytes.SIZEOF\_LONG);

return true;

}

return false;

}

//---------------------------------------------------------------------------

//

// Methods that return copies of fields

//

//---------------------------------------------------------------------------

/\*\*

\* Do not use unless you have to. Used internally for compacting and testing.

\*

\* Use {@link #getRow()}, {@link #getFamily()}, {@link #getQualifier()}, and

\* {@link #getValue()} if accessing a KeyValue client-side.

\* @return Copy of the key portion only.

\*/

public byte [] getKey() {

int keylength = getKeyLength();

byte [] key = new byte[keylength];

System.arraycopy(getBuffer(), getKeyOffset(), key, 0, keylength);

return key;

}

/\*\*

\* Returns value in a new byte array.

\* Primarily for use client-side. If server-side, use

\* {@link #getBuffer()} with appropriate offsets and lengths instead to

\* save on allocations.

\* @return Value in a new byte array.

\*/

public byte [] getValue() {

int o = getValueOffset();

int l = getValueLength();

byte [] result = new byte[l];

System.arraycopy(getBuffer(), o, result, 0, l);

return result;

}

/\*\*

\* Primarily for use client-side. Returns the row of this KeyValue in a new

\* byte array.<p>

\*

\* If server-side, use {@link #getBuffer()} with appropriate offsets and

\* lengths instead.

\* @return Row in a new byte array.

\*/

public byte [] getRow() {

int o = getRowOffset();

short l = getRowLength();

byte [] result = new byte[l];

System.arraycopy(getBuffer(), o, result, 0, l);

return result;

}

/\*\*

\*

\* @return Timestamp

\*/

public long getTimestamp() {

return getTimestamp(getKeyLength());

}

/\*\*

\* @param keylength Pass if you have it to save on a int creation.

\* @return Timestamp

\*/

long getTimestamp(final int keylength) {

int tsOffset = getTimestampOffset(keylength);

return Bytes.toLong(this.bytes, tsOffset);

}

/\*\*

\* @return Type of this KeyValue.

\*/

public byte getType() {

return getType(getKeyLength());

}

/\*\*

\* @param keylength Pass if you have it to save on a int creation.

\* @return Type of this KeyValue.

\*/

byte getType(final int keylength) {

return this.bytes[this.offset + keylength - 1 + ROW\_OFFSET];

}

/\*\*

\* @return True if a delete type, a {@link KeyValue.Type#Delete} or

\* a {KeyValue.Type#DeleteFamily} or a {@link KeyValue.Type#DeleteColumn}

\* KeyValue type.

\*/

public boolean isDelete() {

int t = getType();

return Type.Delete.getCode() <= t && t <= Type.DeleteFamily.getCode();

}

/\*\*

\* @return True if this KV is a {@link KeyValue.Type#Delete} type.

\*/

public boolean isDeleteType() {

return getType() == Type.Delete.getCode();

}

/\*\*

\* @return True if this KV is a delete family type.

\*/

public boolean isDeleteFamily() {

return getType() == Type.DeleteFamily.getCode();

}

/\*\*

\* Primarily for use client-side. Returns the column of this KeyValue in the

\* deprecated format: <i>family:qualifier</i>, and in a new byte array.<p>

\*

\* If server-side, use {@link #getBuffer()} with appropriate offsets and

\* lengths instead.

\* @return Returns column. Makes a copy. Inserts delimiter.

\*/

public byte [] getColumn() {

int fo = getFamilyOffset();

int fl = getFamilyLength(fo);

int ql = getQualifierLength();

byte [] result = new byte[fl + 1 + ql];

System.arraycopy(this.bytes, fo, result, 0, fl);

result[fl] = COLUMN\_FAMILY\_DELIMITER;

System.arraycopy(this.bytes, fo + fl, result, fl + 1, ql);

return result;

}

/\*\*

\* Primarily for use client-side. Returns the family of this KeyValue in a

\* new byte array.<p>

\*

\* If server-side, use {@link #getBuffer()} with appropriate offsets and

\* lengths instead.

\* @return Returns family. Makes a copy.

\*/

public byte [] getFamily() {

int o = getFamilyOffset();

int l = getFamilyLength(o);

byte [] result = new byte[l];

System.arraycopy(this.bytes, o, result, 0, l);

return result;

}

/\*\*

\* Primarily for use client-side. Returns the column qualifier of this

\* KeyValue in a new byte array.<p>

\*

\* If server-side, use {@link #getBuffer()} with appropriate offsets and

\* lengths instead.

\* Use {@link #getBuffer()} with appropriate offsets and lengths instead.

\* @return Returns qualifier. Makes a copy.

\*/

public byte [] getQualifier() {

int o = getQualifierOffset();

int l = getQualifierLength();

byte [] result = new byte[l];

System.arraycopy(this.bytes, o, result, 0, l);

return result;

}

//---------------------------------------------------------------------------

//

// KeyValue splitter

//

//---------------------------------------------------------------------------

/\*\*

\* Utility class that splits a KeyValue buffer into separate byte arrays.

\* <p>

\* Should get rid of this if we can, but is very useful for debugging.

\*/

public static class SplitKeyValue {

private byte [][] split;

SplitKeyValue() {

this.split = new byte[6][];

}

public void setRow(byte [] value) { this.split[0] = value; }

public void setFamily(byte [] value) { this.split[1] = value; }

public void setQualifier(byte [] value) { this.split[2] = value; }

public void setTimestamp(byte [] value) { this.split[3] = value; }

public void setType(byte [] value) { this.split[4] = value; }

public void setValue(byte [] value) { this.split[5] = value; }

public byte [] getRow() { return this.split[0]; }

public byte [] getFamily() { return this.split[1]; }

public byte [] getQualifier() { return this.split[2]; }

public byte [] getTimestamp() { return this.split[3]; }

public byte [] getType() { return this.split[4]; }

public byte [] getValue() { return this.split[5]; }

}

public SplitKeyValue split() {

SplitKeyValue split = new SplitKeyValue();

int splitOffset = this.offset;

int keyLen = Bytes.toInt(bytes, splitOffset);

splitOffset += Bytes.SIZEOF\_INT;

int valLen = Bytes.toInt(bytes, splitOffset);

splitOffset += Bytes.SIZEOF\_INT;

short rowLen = Bytes.toShort(bytes, splitOffset);

splitOffset += Bytes.SIZEOF\_SHORT;

byte [] row = new byte[rowLen];

System.arraycopy(bytes, splitOffset, row, 0, rowLen);

splitOffset += rowLen;

split.setRow(row);

byte famLen = bytes[splitOffset];

splitOffset += Bytes.SIZEOF\_BYTE;

byte [] family = new byte[famLen];

System.arraycopy(bytes, splitOffset, family, 0, famLen);

splitOffset += famLen;

split.setFamily(family);

int colLen = keyLen -

(rowLen + famLen + Bytes.SIZEOF\_SHORT + Bytes.SIZEOF\_BYTE +

Bytes.SIZEOF\_LONG + Bytes.SIZEOF\_BYTE);

byte [] qualifier = new byte[colLen];

System.arraycopy(bytes, splitOffset, qualifier, 0, colLen);

splitOffset += colLen;

split.setQualifier(qualifier);

byte [] timestamp = new byte[Bytes.SIZEOF\_LONG];

System.arraycopy(bytes, splitOffset, timestamp, 0, Bytes.SIZEOF\_LONG);

splitOffset += Bytes.SIZEOF\_LONG;

split.setTimestamp(timestamp);

byte [] type = new byte[1];

type[0] = bytes[splitOffset];

splitOffset += Bytes.SIZEOF\_BYTE;

split.setType(type);

byte [] value = new byte[valLen];

System.arraycopy(bytes, splitOffset, value, 0, valLen);

split.setValue(value);

return split;

}

//---------------------------------------------------------------------------

//

// Compare specified fields against those contained in this KeyValue

//

//---------------------------------------------------------------------------

/\*\*

\* @param family

\* @return True if matching families.

\*/

public boolean matchingFamily(final byte [] family) {

if (this.length == 0 || this.bytes.length == 0) {

return false;

}

int o = getFamilyOffset();

int l = getFamilyLength(o);

return Bytes.compareTo(family, 0, family.length, this.bytes, o, l) == 0;

}

/\*\*

\* @param qualifier

\* @return True if matching qualifiers.

\*/

public boolean matchingQualifier(final byte [] qualifier) {

int o = getQualifierOffset();

int l = getQualifierLength();

return Bytes.compareTo(qualifier, 0, qualifier.length,

this.bytes, o, l) == 0;

}

/\*\*

\* @param column Column minus its delimiter

\* @return True if column matches.

\* @see #matchingColumn(byte[])

\*/

public boolean matchingColumnNoDelimiter(final byte [] column) {

int rl = getRowLength();

int o = getFamilyOffset(rl);

int fl = getFamilyLength(o);

int l = fl + getQualifierLength(rl,fl);

return Bytes.compareTo(column, 0, column.length, this.bytes, o, l) == 0;

}

/\*\*

\* @param column Column with delimiter

\* @return True if column matches.

\*/

public boolean matchingColumn(final byte [] column) {

int index = getFamilyDelimiterIndex(column, 0, column.length);

int rl = getRowLength();

int o = getFamilyOffset(rl);

int fl = getFamilyLength(o);

int ql = getQualifierLength(rl,fl);

if(Bytes.compareTo(column, 0, index, this.bytes, o, fl) != 0) {

return false;

}

return Bytes.compareTo(column, index + 1, column.length - (index + 1),

this.bytes, o + fl, ql) == 0;

}

/\*\*

\*

\* @param family column family

\* @param qualifier column qualifier

\* @return True if column matches

\*/

public boolean matchingColumn(final byte[] family, final byte[] qualifier) {

int rl = getRowLength();

int o = getFamilyOffset(rl);

int fl = getFamilyLength(o);

int ql = getQualifierLength(rl,fl);

if(Bytes.compareTo(family, 0, family.length, this.bytes, o, family.length)

!= 0) {

return false;

}

if(qualifier == null || qualifier.length == 0) {

if(ql == 0) {

return true;

}

return false;

}

return Bytes.compareTo(qualifier, 0, qualifier.length,

this.bytes, o + fl, ql) == 0;

}

/\*\*

\* @param left

\* @param loffset

\* @param llength

\* @param lfamilylength Offset of family delimiter in left column.

\* @param right

\* @param roffset

\* @param rlength

\* @param rfamilylength Offset of family delimiter in right column.

\* @return The result of the comparison.

\*/

static int compareColumns(final byte [] left, final int loffset,

final int llength, final int lfamilylength,

final byte [] right, final int roffset, final int rlength,

final int rfamilylength) {

// Compare family portion first.

int diff = Bytes.compareTo(left, loffset, lfamilylength,

right, roffset, rfamilylength);

if (diff != 0) {

return diff;

}

// Compare qualifier portion

return Bytes.compareTo(left, loffset + lfamilylength,

llength - lfamilylength,

right, roffset + rfamilylength, rlength - rfamilylength);

}

/\*\*

\* @return True if non-null row and column.

\*/

public boolean nonNullRowAndColumn() {

return getRowLength() > 0 && !isEmptyColumn();

}

/\*\*

\* @return True if column is empty.

\*/

public boolean isEmptyColumn() {

return getQualifierLength() == 0;

}

/\*\*

\* Splits a column in family:qualifier form into separate byte arrays.

\*

\* @param c The column.

\* @return The parsed column.

\*/

public static byte [][] parseColumn(byte [] c) {

final byte [][] result = new byte [2][];

final int index = getDelimiter(c, 0, c.length, COLUMN\_FAMILY\_DELIMITER);

if (index == -1) {

// If no delimiter, return <code>c</code> as family and null qualifier.

result[0] = c;

result[1] = null;

return result;

}

result[0] = new byte [index];

System.arraycopy(c, 0, result[0], 0, index);

final int len = c.length - (index + 1);

result[1] = new byte[len];

System.arraycopy(c, index + 1 /\*Skip delimiter\*/, result[1], 0,

len);

return result;

}

/\*\*

\* @param b

\* @return Index of the family-qualifier colon delimiter character in passed

\* buffer.

\*/

public static int getFamilyDelimiterIndex(final byte [] b, final int offset,

final int length) {

return getRequiredDelimiter(b, offset, length, COLUMN\_FAMILY\_DELIMITER);

}

private static int getRequiredDelimiter(final byte [] b,

final int offset, final int length, final int delimiter) {

int index = getDelimiter(b, offset, length, delimiter);

if (index < 0) {

throw new IllegalArgumentException("No " + (char)delimiter + " in <" +

Bytes.toString(b) + ">" + ", length=" + length + ", offset=" + offset);

}

return index;

}

static int getRequiredDelimiterInReverse(final byte [] b,

final int offset, final int length, final int delimiter) {

int index = getDelimiterInReverse(b, offset, length, delimiter);

if (index < 0) {

throw new IllegalArgumentException("No " + delimiter + " in <" +

Bytes.toString(b) + ">" + ", length=" + length + ", offset=" + offset);

}

return index;

}

/\*\*

\* @param b

\* @param delimiter

\* @return Index of delimiter having started from start of <code>b</code>

\* moving rightward.

\*/

public static int getDelimiter(final byte [] b, int offset, final int length,

final int delimiter) {

if (b == null) {

throw new NullPointerException();

}

int result = -1;

for (int i = offset; i < length + offset; i++) {

if (b[i] == delimiter) {

result = i;

break;

}

}

return result;

}

/\*\*

\* Find index of passed delimiter walking from end of buffer backwards.

\* @param b

\* @param delimiter

\* @return Index of delimiter

\*/

public static int getDelimiterInReverse(final byte [] b, final int offset,

final int length, final int delimiter) {

if (b == null) {

throw new NullPointerException();

}

int result = -1;

for (int i = (offset + length) - 1; i >= offset; i--) {

if (b[i] == delimiter) {

result = i;

break;

}

}

return result;

}

/\*\*

\* A {@link KVComparator} for <code>-ROOT-</code> catalog table

\* {@link KeyValue}s.

\*/

public static class RootComparator extends MetaComparator {

private final KeyComparator rawcomparator = new RootKeyComparator();

public KeyComparator getRawComparator() {

return this.rawcomparator;

}

@Override

protected Object clone() throws CloneNotSupportedException {

return new RootComparator();

}

}

/\*\*

\* A {@link KVComparator} for <code>.META.</code> catalog table

\* {@link KeyValue}s.

\*/

public static class MetaComparator extends KVComparator {

private final KeyComparator rawcomparator = new MetaKeyComparator();

public KeyComparator getRawComparator() {

return this.rawcomparator;

}

@Override

protected Object clone() throws CloneNotSupportedException {

return new MetaComparator();

}

}

/\*\*

\* Compare KeyValues. When we compare KeyValues, we only compare the Key

\* portion. This means two KeyValues with same Key but different Values are

\* considered the same as far as this Comparator is concerned.

\* Hosts a {@link KeyComparator}.

\*/

public static class KVComparator implements java.util.Comparator<KeyValue> {

private final KeyComparator rawcomparator = new KeyComparator();

/\*\*

\* @return RawComparator that can compare the Key portion of a KeyValue.

\* Used in hfile where indices are the Key portion of a KeyValue.

\*/

public KeyComparator getRawComparator() {

return this.rawcomparator;

}

public int compare(final KeyValue left, final KeyValue right) {

return getRawComparator().compare(left.getBuffer(),

left.getOffset() + ROW\_OFFSET, left.getKeyLength(),

right.getBuffer(), right.getOffset() + ROW\_OFFSET,

right.getKeyLength());

}

public int compareTimestamps(final KeyValue left, final KeyValue right) {

return compareTimestamps(left, left.getKeyLength(), right,

right.getKeyLength());

}

int compareTimestamps(final KeyValue left, final int lkeylength,

final KeyValue right, final int rkeylength) {

// Compare timestamps

long ltimestamp = left.getTimestamp(lkeylength);

long rtimestamp = right.getTimestamp(rkeylength);

return getRawComparator().compareTimestamps(ltimestamp, rtimestamp);

}

/\*\*

\* @param left

\* @param right

\* @return Result comparing rows.

\*/

public int compareRows(final KeyValue left, final KeyValue right) {

return compareRows(left, left.getRowLength(), right,

right.getRowLength());

}

/\*\*

\* @param left

\* @param lrowlength Length of left row.

\* @param right

\* @param rrowlength Length of right row.

\* @return Result comparing rows.

\*/

public int compareRows(final KeyValue left, final short lrowlength,

final KeyValue right, final short rrowlength) {

return getRawComparator().compareRows(left.getBuffer(),

left.getRowOffset(), lrowlength,

right.getBuffer(), right.getRowOffset(), rrowlength);

}

/\*\*

\* @param left

\* @param row - row key (arbitrary byte array)

\* @return RawComparator

\*/

public int compareRows(final KeyValue left, final byte [] row) {

return getRawComparator().compareRows(left.getBuffer(),

left.getRowOffset(), left.getRowLength(), row, 0, row.length);

}

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return getRawComparator().compareRows(left, loffset, llength,

right, roffset, rlength);

}

public int compareColumns(final KeyValue left, final byte [] right,

final int roffset, final int rlength, final int rfamilyoffset) {

int offset = left.getFamilyOffset();

int length = left.getFamilyLength() + left.getQualifierLength();

return getRawComparator().compareColumns(left.getBuffer(), offset, length,

left.getFamilyLength(offset),

right, roffset, rlength, rfamilyoffset);

}

int compareColumns(final KeyValue left, final short lrowlength,

final KeyValue right, final short rrowlength) {

int lfoffset = left.getFamilyOffset(lrowlength);

int rfoffset = right.getFamilyOffset(rrowlength);

int lclength = left.getTotalColumnLength(lrowlength,lfoffset);

int rclength = right.getTotalColumnLength(rrowlength, rfoffset);

int lfamilylength = left.getFamilyLength(lfoffset);

int rfamilylength = right.getFamilyLength(rfoffset);

return getRawComparator().compareColumns(left.getBuffer(), lfoffset,

lclength, lfamilylength,

right.getBuffer(), rfoffset, rclength, rfamilylength);

}

/\*\*

\* Compares the row and column of two keyvalues

\* @param left

\* @param right

\* @return True if same row and column.

\*/

public boolean matchingRowColumn(final KeyValue left,

final KeyValue right) {

short lrowlength = left.getRowLength();

short rrowlength = right.getRowLength();

if (!matchingRows(left, lrowlength, right, rrowlength)) {

return false;

}

return compareColumns(left, lrowlength, right, rrowlength) == 0;

}

/\*\*

\* @param left

\* @param right

\* @return True if rows match.

\*/

public boolean matchingRows(final KeyValue left, final byte [] right) {

return compareRows(left, right) == 0;

}

/\*\*

\* @param left

\* @param right

\* @return True if rows match.

\*/

public boolean matchingRows(final KeyValue left, final KeyValue right) {

short lrowlength = left.getRowLength();

short rrowlength = right.getRowLength();

return matchingRows(left, lrowlength, right, rrowlength);

}

/\*\*

\* @param left

\* @param lrowlength

\* @param right

\* @param rrowlength

\* @return True if rows match.

\*/

public boolean matchingRows(final KeyValue left, final short lrowlength,

final KeyValue right, final short rrowlength) {

int compare = compareRows(left, lrowlength, right, rrowlength);

if (compare != 0) {

return false;

}

return true;

}

public boolean matchingRows(final byte [] left, final int loffset,

final int llength,

final byte [] right, final int roffset, final int rlength) {

int compare = compareRows(left, loffset, llength,

right, roffset, rlength);

if (compare != 0) {

return false;

}

return true;

}

/\*\*

\* Compares the row and timestamp of two keys

\* Was called matchesWithoutColumn in HStoreKey.

\* @param right Key to compare against.

\* @return True if same row and timestamp is greater than the timestamp in

\* <code>right</code>

\*/

public boolean matchingRowsGreaterTimestamp(final KeyValue left,

final KeyValue right) {

short lrowlength = left.getRowLength();

short rrowlength = right.getRowLength();

if (!matchingRows(left, lrowlength, right, rrowlength)) {

return false;

}

return left.getTimestamp() >= right.getTimestamp();

}

@Override

protected Object clone() throws CloneNotSupportedException {

return new KVComparator();

}

/\*\*

\* @return Comparator that ignores timestamps; useful counting versions.

\*/

public KVComparator getComparatorIgnoringTimestamps() {

KVComparator c = null;

try {

c = (KVComparator)this.clone();

c.getRawComparator().ignoreTimestamp = true;

} catch (CloneNotSupportedException e) {

LOG.error("Not supported", e);

}

return c;

}

/\*\*

\* @return Comparator that ignores key type; useful checking deletes

\*/

public KVComparator getComparatorIgnoringType() {

KVComparator c = null;

try {

c = (KVComparator)this.clone();

c.getRawComparator().ignoreType = true;

} catch (CloneNotSupportedException e) {

LOG.error("Not supported", e);

}

return c;

}

}

/\*\*

\* Creates a KeyValue that is last on the specified row id. That is,

\* every other possible KeyValue for the given row would compareTo()

\* less than the result of this call.

\* @param row row key

\* @return Last possible KeyValue on passed <code>row</code>

\*/

public static KeyValue createLastOnRow(final byte[] row) {

return new KeyValue(row, null, null, HConstants.LATEST\_TIMESTAMP, Type.Minimum);

}

/\*\*

\* Create a KeyValue that is smaller than all other possible KeyValues

\* for the given row. That is any (valid) KeyValue on 'row' would sort

\* \_after\_ the result.

\*

\* @param row - row key (arbitrary byte array)

\* @return First possible KeyValue on passed <code>row</code>

\*/

public static KeyValue createFirstOnRow(final byte [] row) {

return createFirstOnRow(row, HConstants.LATEST\_TIMESTAMP);

}

/\*\*

\* Creates a KeyValue that is smaller than all other KeyValues that

\* are older than the passed timestamp.

\* @param row - row key (arbitrary byte array)

\* @param ts - timestamp

\* @return First possible key on passed <code>row</code> and timestamp.

\*/

public static KeyValue createFirstOnRow(final byte [] row,

final long ts) {

return new KeyValue(row, null, null, ts, Type.Maximum);

}

/\*\*

\* @param row - row key (arbitrary byte array)

\* @param c column - {@link #parseColumn(byte[])} is called to split

\* the column.

\* @param ts - timestamp

\* @return First possible key on passed <code>row</code>, column and timestamp

\* @deprecated

\*/

public static KeyValue createFirstOnRow(final byte [] row, final byte [] c,

final long ts) {

byte [][] split = parseColumn(c);

return new KeyValue(row, split[0], split[1], ts, Type.Maximum);

}

/\*\*

\* Create a KeyValue for the specified row, family and qualifier that would be

\* smaller than all other possible KeyValues that have the same row,family,qualifier.

\* Used for seeking.

\* @param row - row key (arbitrary byte array)

\* @param family - family name

\* @param qualifier - column qualifier

\* @return First possible key on passed <code>row</code>, and column.

\*/

public static KeyValue createFirstOnRow(final byte [] row, final byte [] family,

final byte [] qualifier) {

return new KeyValue(row, family, qualifier, HConstants.LATEST\_TIMESTAMP, Type.Maximum);

}

/\*\*

\* @param row - row key (arbitrary byte array)

\* @param f - family name

\* @param q - column qualifier

\* @param ts - timestamp

\* @return First possible key on passed <code>row</code>, column and timestamp

\*/

public static KeyValue createFirstOnRow(final byte [] row, final byte [] f,

final byte [] q, final long ts) {

return new KeyValue(row, f, q, ts, Type.Maximum);

}

/\*\*

\* @param b

\* @param o

\* @param l

\* @return A KeyValue made of a byte array that holds the key-only part.

\* Needed to convert hfile index members to KeyValues.

\*/

public static KeyValue createKeyValueFromKey(final byte [] b, final int o,

final int l) {

byte [] newb = new byte[b.length + ROW\_OFFSET];

System.arraycopy(b, o, newb, ROW\_OFFSET, l);

Bytes.putInt(newb, 0, b.length);

Bytes.putInt(newb, Bytes.SIZEOF\_INT, 0);

return new KeyValue(newb);

}

/\*\*

\* Compare key portion of a {@link KeyValue} for keys in <code>-ROOT-<code>

\* table.

\*/

public static class RootKeyComparator extends MetaKeyComparator {

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

// Rows look like this: .META.,ROW\_FROM\_META,RID

// LOG.info("ROOT " + Bytes.toString(left, loffset, llength) +

// "---" + Bytes.toString(right, roffset, rlength));

final int metalength = 7; // '.META.' length

int lmetaOffsetPlusDelimiter = loffset + metalength;

int leftFarDelimiter = getDelimiterInReverse(left,

lmetaOffsetPlusDelimiter,

llength - metalength, HRegionInfo.DELIMITER);

int rmetaOffsetPlusDelimiter = roffset + metalength;

int rightFarDelimiter = getDelimiterInReverse(right,

rmetaOffsetPlusDelimiter, rlength - metalength,

HRegionInfo.DELIMITER);

if (leftFarDelimiter < 0 && rightFarDelimiter >= 0) {

// Nothing between .META. and regionid. Its first key.

return -1;

} else if (rightFarDelimiter < 0 && leftFarDelimiter >= 0) {

return 1;

} else if (leftFarDelimiter < 0 && rightFarDelimiter < 0) {

return 0;

}

int result = super.compareRows(left, lmetaOffsetPlusDelimiter,

leftFarDelimiter - lmetaOffsetPlusDelimiter,

right, rmetaOffsetPlusDelimiter,

rightFarDelimiter - rmetaOffsetPlusDelimiter);

if (result != 0) {

return result;

}

// Compare last part of row, the rowid.

leftFarDelimiter++;

rightFarDelimiter++;

result = compareRowid(left, leftFarDelimiter,

llength - (leftFarDelimiter - loffset),

right, rightFarDelimiter, rlength - (rightFarDelimiter - roffset));

return result;

}

}

/\*\*

\* Comparator that compares row component only of a KeyValue.

\*/

public static class RowComparator implements Comparator<KeyValue> {

final KVComparator comparator;

public RowComparator(final KVComparator c) {

this.comparator = c;

}

public int compare(KeyValue left, KeyValue right) {

return comparator.compareRows(left, right);

}

}

/\*\*

\* Compare key portion of a {@link KeyValue} for keys in <code>.META.</code>

\* table.

\*/

public static class MetaKeyComparator extends KeyComparator {

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

// LOG.info("META " + Bytes.toString(left, loffset, llength) +

// "---" + Bytes.toString(right, roffset, rlength));

int leftDelimiter = getDelimiter(left, loffset, llength,

HRegionInfo.DELIMITER);

int rightDelimiter = getDelimiter(right, roffset, rlength,

HRegionInfo.DELIMITER);

if (leftDelimiter < 0 && rightDelimiter >= 0) {

// Nothing between .META. and regionid. Its first key.

return -1;

} else if (rightDelimiter < 0 && leftDelimiter >= 0) {

return 1;

} else if (leftDelimiter < 0 && rightDelimiter < 0) {

return 0;

}

// Compare up to the delimiter

int result = Bytes.compareTo(left, loffset, leftDelimiter - loffset,

right, roffset, rightDelimiter - roffset);

if (result != 0) {

return result;

}

// Compare middle bit of the row.

// Move past delimiter

leftDelimiter++;

rightDelimiter++;

int leftFarDelimiter = getRequiredDelimiterInReverse(left, leftDelimiter,

llength - (leftDelimiter - loffset), HRegionInfo.DELIMITER);

int rightFarDelimiter = getRequiredDelimiterInReverse(right,

rightDelimiter, rlength - (rightDelimiter - roffset),

HRegionInfo.DELIMITER);

// Now compare middlesection of row.

result = super.compareRows(left, leftDelimiter,

leftFarDelimiter - leftDelimiter, right, rightDelimiter,

rightFarDelimiter - rightDelimiter);

if (result != 0) {

return result;

}

// Compare last part of row, the rowid.

leftFarDelimiter++;

rightFarDelimiter++;

result = compareRowid(left, leftFarDelimiter,

llength - (leftFarDelimiter - loffset),

right, rightFarDelimiter, rlength - (rightFarDelimiter - roffset));

return result;

}

protected int compareRowid(byte[] left, int loffset, int llength,

byte[] right, int roffset, int rlength) {

return Bytes.compareTo(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* Compare key portion of a {@link KeyValue}.

\*/

public static class KeyComparator implements RawComparator<byte []> {

volatile boolean ignoreTimestamp = false;

volatile boolean ignoreType = false;

public int compare(byte[] left, int loffset, int llength, byte[] right,

int roffset, int rlength) {

// Compare row

short lrowlength = Bytes.toShort(left, loffset);

short rrowlength = Bytes.toShort(right, roffset);

int compare = compareRows(left, loffset + Bytes.SIZEOF\_SHORT,

lrowlength,

right, roffset + Bytes.SIZEOF\_SHORT, rrowlength);

if (compare != 0) {

return compare;

}

// Compare column family. Start compare past row and family length.

int lcolumnoffset = Bytes.SIZEOF\_SHORT + lrowlength + 1 + loffset;

int rcolumnoffset = Bytes.SIZEOF\_SHORT + rrowlength + 1 + roffset;

int lcolumnlength = llength - TIMESTAMP\_TYPE\_SIZE -

(lcolumnoffset - loffset);

int rcolumnlength = rlength - TIMESTAMP\_TYPE\_SIZE -

(rcolumnoffset - roffset);

// if row matches, and no column in the 'left' AND put type is 'minimum',

// then return that left is larger than right.

// This supports 'last key on a row' - the magic is if there is no column in the

// left operand, and the left operand has a type of '0' - magical value,

// then we say the left is bigger. This will let us seek to the last key in

// a row.

byte ltype = left[loffset + (llength - 1)];

byte rtype = right[roffset + (rlength - 1)];

if (lcolumnlength == 0 && ltype == Type.Minimum.getCode()) {

return 1; // left is bigger.

}

if (rcolumnlength == 0 && rtype == Type.Minimum.getCode()) {

return -1;

}

// TODO the family and qualifier should be compared separately

compare = Bytes.compareTo(left, lcolumnoffset, lcolumnlength, right,

rcolumnoffset, rcolumnlength);

if (compare != 0) {

return compare;

}

if (!this.ignoreTimestamp) {

// Get timestamps.

long ltimestamp = Bytes.toLong(left,

loffset + (llength - TIMESTAMP\_TYPE\_SIZE));

long rtimestamp = Bytes.toLong(right,

roffset + (rlength - TIMESTAMP\_TYPE\_SIZE));

compare = compareTimestamps(ltimestamp, rtimestamp);

if (compare != 0) {

return compare;

}

}

if (!this.ignoreType) {

// Compare types. Let the delete types sort ahead of puts; i.e. types

// of higher numbers sort before those of lesser numbers

return (0xff & rtype) - (0xff & ltype);

}

return 0;

}

public int compare(byte[] left, byte[] right) {

return compare(left, 0, left.length, right, 0, right.length);

}

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return Bytes.compareTo(left, loffset, llength, right, roffset, rlength);

}

protected int compareColumns(

byte [] left, int loffset, int llength, final int lfamilylength,

byte [] right, int roffset, int rlength, final int rfamilylength) {

return KeyValue.compareColumns(left, loffset, llength, lfamilylength,

right, roffset, rlength, rfamilylength);

}

int compareTimestamps(final long ltimestamp, final long rtimestamp) {

// The below older timestamps sorting ahead of newer timestamps looks

// wrong but it is intentional. This way, newer timestamps are first

// found when we iterate over a memstore and newer versions are the

// first we trip over when reading from a store file.

if (ltimestamp < rtimestamp) {

return 1;

} else if (ltimestamp > rtimestamp) {

return -1;

}

return 0;

}

}

// HeapSize

public long heapSize() {

return ClassSize.align(ClassSize.OBJECT + ClassSize.REFERENCE +

ClassSize.align(ClassSize.ARRAY + length) +

(2 \* Bytes.SIZEOF\_INT) +

Bytes.SIZEOF\_LONG);

}

// Writable

public void readFields(final DataInput in) throws IOException {

this.length = in.readInt();

this.offset = 0;

this.bytes = new byte[this.length];

in.readFully(this.bytes, 0, this.length);

}

public void write(final DataOutput out) throws IOException {

out.writeInt(this.length);

out.write(this.bytes, this.offset, this.length);

}

}

package org.apache.hadoop.hbase;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

import java.nio.ByteBuffer;

import org.apache.hadoop.hbase.io.HeapSize;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.util.Writables;

import org.apache.hadoop.io.RawComparator;

import org.apache.hadoop.io.Writable;

import org.apache.hadoop.io.WritableComparable;

import org.apache.hadoop.io.WritableComparator;

import org.apache.hadoop.io.WritableUtils;

/\*\*

\* A Key for a stored row.

\* @deprecated Replaced by {@link KeyValue}.

\*/

public class HStoreKey implements WritableComparable<HStoreKey>, HeapSize {

/\*\*

\* Colon character in UTF-8

\*/

public static final char COLUMN\_FAMILY\_DELIMITER = ':';

/\*\*

\* Estimated size tax paid for each instance of HSK. Estimate based on

\* study of jhat and jprofiler numbers.

\*/

// In jprofiler, says shallow size is 48 bytes. Add to it cost of two

// byte arrays and then something for the HRI hosting.

public static final int ESTIMATED\_HEAP\_TAX = 48;

private byte [] row = HConstants.EMPTY\_BYTE\_ARRAY;

private byte [] column = HConstants.EMPTY\_BYTE\_ARRAY;

private long timestamp = Long.MAX\_VALUE;

private static final HStoreKey.StoreKeyComparator PLAIN\_COMPARATOR =

new HStoreKey.StoreKeyComparator();

private static final HStoreKey.StoreKeyComparator META\_COMPARATOR =

new HStoreKey.MetaStoreKeyComparator();

private static final HStoreKey.StoreKeyComparator ROOT\_COMPARATOR =

new HStoreKey.RootStoreKeyComparator();

/\*\* Default constructor used in conjunction with Writable interface \*/

public HStoreKey() {

super();

}

/\*\*

\* Create an HStoreKey specifying only the row

\* The column defaults to the empty string, the time stamp defaults to

\* Long.MAX\_VALUE and the table defaults to empty string

\*

\* @param row - row key

\*/

public HStoreKey(final byte [] row) {

this(row, Long.MAX\_VALUE);

}

/\*\*

\* Create an HStoreKey specifying only the row

\* The column defaults to the empty string, the time stamp defaults to

\* Long.MAX\_VALUE and the table defaults to empty string

\*

\* @param row - row key

\*/

public HStoreKey(final String row) {

this(Bytes.toBytes(row), Long.MAX\_VALUE);

}

/\*\*

\* Create an HStoreKey specifying the row and timestamp

\* The column and table names default to the empty string

\*

\* @param row row key

\* @param timestamp timestamp value

\*/

public HStoreKey(final byte [] row, final long timestamp) {

this(row, HConstants.EMPTY\_BYTE\_ARRAY, timestamp);

}

/\*\*

\* Create an HStoreKey specifying the row and column names

\* The timestamp defaults to LATEST\_TIMESTAMP

\* and table name defaults to the empty string

\*

\* @param row row key

\* @param column column key

\*/

public HStoreKey(final String row, final String column) {

this(row, column, HConstants.LATEST\_TIMESTAMP);

}

/\*\*

\* Create an HStoreKey specifying the row and column names

\* The timestamp defaults to LATEST\_TIMESTAMP

\* and table name defaults to the empty string

\*

\* @param row row key

\* @param column column key

\*/

public HStoreKey(final byte [] row, final byte [] column) {

this(row, column, HConstants.LATEST\_TIMESTAMP);

}

/\*\*

\* Create an HStoreKey specifying all the fields

\* Does not make copies of the passed byte arrays. Presumes the passed

\* arrays immutable.

\* @param row row key

\* @param column column key

\* @param timestamp timestamp value

\*/

public HStoreKey(final String row, final String column, final long timestamp) {

this (Bytes.toBytes(row), Bytes.toBytes(column), timestamp);

}

/\*\*

\* Create an HStoreKey specifying all the fields with specified table

\* Does not make copies of the passed byte arrays. Presumes the passed

\* arrays immutable.

\* @param row row key

\* @param column column key

\* @param timestamp timestamp value

\*/

public HStoreKey(final byte [] row, final byte [] column, final long timestamp) {

// Make copies

this.row = row;

this.column = column;

this.timestamp = timestamp;

}

/\*\*

\* Constructs a new HStoreKey from another

\*

\* @param other the source key

\*/

public HStoreKey(final HStoreKey other) {

this(other.getRow(), other.getColumn(), other.getTimestamp());

}

public HStoreKey(final ByteBuffer bb) {

this(getRow(bb), getColumn(bb), getTimestamp(bb));

}

/\*\*

\* Change the value of the row key

\*

\* @param newrow new row key value

\*/

public void setRow(final byte [] newrow) {

this.row = newrow;

}

/\*\*

\* Change the value of the column in this key

\*

\* @param c new column family value

\*/

public void setColumn(final byte [] c) {

this.column = c;

}

/\*\*

\* Change the value of the timestamp field

\*

\* @param timestamp new timestamp value

\*/

public void setVersion(final long timestamp) {

this.timestamp = timestamp;

}

/\*\*

\* Set the value of this HStoreKey from the supplied key

\*

\* @param k key value to copy

\*/

public void set(final HStoreKey k) {

this.row = k.getRow();

this.column = k.getColumn();

this.timestamp = k.getTimestamp();

}

/\*\* @return value of row key \*/

public byte [] getRow() {

return row;

}

/\*\* @return value of column \*/

public byte [] getColumn() {

return this.column;

}

/\*\* @return value of timestamp \*/

public long getTimestamp() {

return this.timestamp;

}

/\*\*

\* Compares the row and column of two keys

\* @param other Key to compare against. Compares row and column.

\* @return True if same row and column.

\* @see #matchesWithoutColumn(HStoreKey)

\* @see #matchesRowFamily(HStoreKey)

\*/

public boolean matchesRowCol(final HStoreKey other) {

return HStoreKey.equalsTwoRowKeys(getRow(), other.getRow()) &&

Bytes.equals(getColumn(), other.getColumn());

}

/\*\*

\* Compares the row and timestamp of two keys

\*

\* @param other Key to copmare against. Compares row and timestamp.

\*

\* @return True if same row and timestamp is greater than <code>other</code>

\* @see #matchesRowCol(HStoreKey)

\* @see #matchesRowFamily(HStoreKey)

\*/

public boolean matchesWithoutColumn(final HStoreKey other) {

return equalsTwoRowKeys(getRow(), other.getRow()) &&

getTimestamp() >= other.getTimestamp();

}

/\*\*

\* Compares the row and column family of two keys

\*

\* @param that Key to compare against. Compares row and column family

\*

\* @return true if same row and column family

\* @see #matchesRowCol(HStoreKey)

\* @see #matchesWithoutColumn(HStoreKey)

\*/

public boolean matchesRowFamily(final HStoreKey that) {

final int delimiterIndex = getFamilyDelimiterIndex(getColumn());

return equalsTwoRowKeys(getRow(), that.getRow()) &&

Bytes.compareTo(getColumn(), 0, delimiterIndex, that.getColumn(), 0,

delimiterIndex) == 0;

}

/\*\*

\* @see java.lang.Object#toString()

\*/

@Override

public String toString() {

return Bytes.toString(this.row) + "/" + Bytes.toString(this.column) + "/" +

timestamp;

}

/\*\*

\* @see java.lang.Object#equals(java.lang.Object)

\*/

@Override

public boolean equals(final Object obj) {

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final HStoreKey other = (HStoreKey)obj;

// Do a quick check.

if (this.row.length != other.row.length ||

this.column.length != other.column.length ||

this.timestamp != other.timestamp) {

return false;

}

return compareTo(other) == 0;

}

/\*\*

\* @see java.lang.Object#hashCode()

\*/

@Override

public int hashCode() {

int c = Bytes.hashCode(getRow());

c ^= Bytes.hashCode(getColumn());

c ^= getTimestamp();

return c;

}

// Comparable

/\*\*

\* @param o

\* @return int

\* @deprecated Use Comparators instead. This can give wrong results.

\*/

@Deprecated

public int compareTo(final HStoreKey o) {

return compareTo(this, o);

}

/\*\*

\* @param left

\* @param right

\* @return

\* @deprecated Use Comparators instead. This can give wrong results because

\* does not take into account special handling needed for meta and root rows.

\*/

@Deprecated

static int compareTo(final HStoreKey left, final HStoreKey right) {

// We can be passed null

if (left == null && right == null) return 0;

if (left == null) return -1;

if (right == null) return 1;

int result = Bytes.compareTo(left.getRow(), right.getRow());

if (result != 0) {

return result;

}

result = left.getColumn() == null && right.getColumn() == null? 0:

left.getColumn() == null && right.getColumn() != null? -1:

left.getColumn() != null && right.getColumn() == null? 1:

Bytes.compareTo(left.getColumn(), right.getColumn());

if (result != 0) {

return result;

}

// The below older timestamps sorting ahead of newer timestamps looks

// wrong but it is intentional. This way, newer timestamps are first

// found when we iterate over a memcache and newer versions are the

// first we trip over when reading from a store file.

if (left.getTimestamp() < right.getTimestamp()) {

result = 1;

} else if (left.getTimestamp() > right.getTimestamp()) {

result = -1;

}

return result;

}

/\*\*

\* @param column

\* @return New byte array that holds <code>column</code> family prefix only

\* (Does not include the colon DELIMITER).

\* @throws ColumnNameParseException

\* @see #parseColumn(byte[])

\*/

public static byte [] getFamily(final byte [] column)

throws ColumnNameParseException {

final int index = getFamilyDelimiterIndex(column);

if (index <= 0) {

throw new ColumnNameParseException("Missing ':' delimiter between " +

"column family and qualifier in the passed column name <" +

Bytes.toString(column) + ">");

}

final byte [] result = new byte[index];

System.arraycopy(column, 0, result, 0, index);

return result;

}

/\*\*

\* @param column

\* @return Return hash of family portion of passed column.

\*/

public static Integer getFamilyMapKey(final byte [] column) {

final int index = getFamilyDelimiterIndex(column);

// If index < -1, presume passed column is a family name absent colon

// delimiter

return Bytes.mapKey(column, index > 0? index: column.length);

}

/\*\*

\* @param family

\* @param column

\* @return True if <code>column</code> has a family of <code>family</code>.

\*/

public static boolean matchingFamily(final byte [] family,

final byte [] column) {

// Make sure index of the ':' is at same offset.

final int index = getFamilyDelimiterIndex(column);

if (index != family.length) {

return false;

}

return Bytes.compareTo(family, 0, index, column, 0, index) == 0;

}

/\*\*

\* @param family

\* @return Return <code>family</code> plus the family delimiter.

\*/

public static byte [] addDelimiter(final byte [] family) {

// Manufacture key by adding delimiter to the passed in colFamily.

final byte [] familyPlusDelimiter = new byte [family.length + 1];

System.arraycopy(family, 0, familyPlusDelimiter, 0, family.length);

familyPlusDelimiter[family.length] = HStoreKey.COLUMN\_FAMILY\_DELIMITER;

return familyPlusDelimiter;

}

/\*\*

\* @param column

\* @return New byte array that holds <code>column</code> qualifier suffix.

\* @see #parseColumn(byte[])

\*/

public static byte [] getQualifier(final byte [] column) {

final int index = getFamilyDelimiterIndex(column);

final int len = column.length - (index + 1);

final byte [] result = new byte[len];

System.arraycopy(column, index + 1, result, 0, len);

return result;

}

/\*\*

\* @param c Column name

\* @return Return array of size two whose first element has the family

\* prefix of passed column <code>c</code> and whose second element is the

\* column qualifier.

\* @throws ColumnNameParseException

\*/

public static byte [][] parseColumn(final byte [] c)

throws ColumnNameParseException {

final byte [][] result = new byte [2][];

final int index = getFamilyDelimiterIndex(c);

if (index == -1) {

throw new ColumnNameParseException("Impossible column name: " + Bytes.toString(c));

}

result[0] = new byte [index];

System.arraycopy(c, 0, result[0], 0, index);

final int len = c.length - (index + 1);

result[1] = new byte[len];

System.arraycopy(c, index + 1 /\*Skip delimiter\*/, result[1], 0,

len);

return result;

}

/\*\*

\* @param b

\* @return Index of the family-qualifier colon delimiter character in passed

\* buffer.

\*/

public static int getFamilyDelimiterIndex(final byte [] b) {

return getDelimiter(b, 0, b.length, COLUMN\_FAMILY\_DELIMITER);

}

static int getRequiredDelimiterInReverse(final byte [] b,

final int offset, final int length, final int delimiter) {

int index = getDelimiterInReverse(b, offset, length, delimiter);

if (index < 0) {

throw new IllegalArgumentException("No " + delimiter + " in <" +

Bytes.toString(b) + ">" + ", length=" + length + ", offset=" + offset);

}

return index;

}

/\*

\* @param b

\* @param delimiter

\* @return Index of delimiter having started from end of <code>b</code> moving

\* leftward.

\*/

static int getDelimiter(final byte [] b, int offset, final int length,

final int delimiter) {

if (b == null) {

throw new NullPointerException();

}

int result = -1;

for (int i = offset; i < length + offset; i++) {

if (b[i] == delimiter) {

result = i;

break;

}

}

return result;

}

/\*

\* @param b

\* @param delimiter

\* @return Index of delimiter

\*/

private static int getDelimiterInReverse(final byte [] b, final int offset,

final int length, final int delimiter) {

if (b == null) {

throw new NullPointerException();

}

int result = -1;

for (int i = (offset + length) - 1; i >= offset; i--) {

if (b[i] == delimiter) {

result = i;

break;

}

}

return result;

}

/\*\*

\* Utility method to check if two row keys are equal.

\* This is required because of the meta delimiters

\* This is a hack

\* @param rowA

\* @param rowB

\* @return if it's equal

\*/

public static boolean equalsTwoRowKeys(final byte[] rowA, final byte[] rowB) {

return ((rowA == null) && (rowB == null)) ? true:

(rowA == null) || (rowB == null) || (rowA.length != rowB.length) ? false:

Bytes.compareTo(rowA, rowB) == 0;

}

// Writable

public void write(final DataOutput out) throws IOException {

Bytes.writeByteArray(out, this.row);

Bytes.writeByteArray(out, this.column);

out.writeLong(timestamp);

}

public void readFields(final DataInput in) throws IOException {

this.row = Bytes.readByteArray(in);

this.column = Bytes.readByteArray(in);

this.timestamp = in.readLong();

}

/\*\*

\* @param hsk

\* @return Size of this key in serialized bytes.

\*/

public static int getSerializedSize(final HStoreKey hsk) {

return getSerializedSize(hsk.getRow()) +

getSerializedSize(hsk.getColumn()) +

Bytes.SIZEOF\_LONG;

}

/\*\*

\* @param b

\* @return Length of buffer when its been serialized.

\*/

private static int getSerializedSize(final byte [] b) {

return b == null? 1: b.length + WritableUtils.getVIntSize(b.length);

}

public long heapSize() {

return getRow().length + Bytes.ESTIMATED\_HEAP\_TAX +

getColumn().length + Bytes.ESTIMATED\_HEAP\_TAX +

ESTIMATED\_HEAP\_TAX;

}

/\*\*

\* @return The bytes of <code>hsk</code> gotten by running its

\* {@link Writable#write(java.io.DataOutput)} method.

\* @throws IOException

\*/

public byte [] getBytes() throws IOException {

return getBytes(this);

}

/\*\*

\* Return serialize <code>hsk</code> bytes.

\* Note, this method's implementation has changed. Used to just return

\* row and column. This is a customized version of

\* {@link Writables#getBytes(Writable)}

\* @param hsk Instance

\* @return The bytes of <code>hsk</code> gotten by running its

\* {@link Writable#write(java.io.DataOutput)} method.

\* @throws IOException

\*/

public static byte [] getBytes(final HStoreKey hsk) throws IOException {

return getBytes(hsk.getRow(), hsk.getColumn(), hsk.getTimestamp());

}

/\*\*

\* @param row Can't be null

\* @return Passed arguments as a serialized HSK.

\* @throws IOException

\*/

public static byte [] getBytes(final byte [] row)

throws IOException {

return getBytes(row, null, HConstants.LATEST\_TIMESTAMP);

}

/\*\*

\* @param row Can't be null

\* @param column Can be null

\* @param ts

\* @return Passed arguments as a serialized HSK.

\* @throws IOException

\*/

public static byte [] getBytes(final byte [] row, final byte [] column,

final long ts)

throws IOException {

// TODO: Get vint sizes as I calculate serialized size of hsk.

byte [] b = new byte [getSerializedSize(row) +

getSerializedSize(column) + Bytes.SIZEOF\_LONG];

int offset = Bytes.writeByteArray(b, 0, row, 0, row.length);

byte [] c = column == null? HConstants.EMPTY\_BYTE\_ARRAY: column;

offset = Bytes.writeByteArray(b, offset, c, 0, c.length);

byte [] timestamp = Bytes.toBytes(ts);

System.arraycopy(timestamp, 0, b, offset, timestamp.length);

return b;

}

/\*\*

\* @param bb ByteBuffer that contains serialized HStoreKey

\* @return Row

\*/

public static byte [] getRow(final ByteBuffer bb) {

byte firstByte = bb.get(0);

int vint = firstByte;

int vintWidth = WritableUtils.decodeVIntSize(firstByte);

if (vintWidth != 1) {

vint = getBigVint(vintWidth, firstByte, bb.array(), bb.arrayOffset());

}

byte [] b = new byte [vint];

System.arraycopy(bb.array(), bb.arrayOffset() + vintWidth, b, 0, vint);

return b;

}

/\*\*

\* @param bb ByteBuffer that contains serialized HStoreKey

\* @return Column

\*/

public static byte [] getColumn(final ByteBuffer bb) {

// Skip over row.

int offset = skipVintdByteArray(bb, 0);

byte firstByte = bb.get(offset);

int vint = firstByte;

int vintWidth = WritableUtils.decodeVIntSize(firstByte);

if (vintWidth != 1) {

vint = getBigVint(vintWidth, firstByte, bb.array(),

bb.arrayOffset() + offset);

}

byte [] b = new byte [vint];

System.arraycopy(bb.array(), bb.arrayOffset() + offset + vintWidth, b, 0,

vint);

return b;

}

/\*\*

\* @param bb ByteBuffer that contains serialized HStoreKey

\* @return Timestamp

\*/

public static long getTimestamp(final ByteBuffer bb) {

return bb.getLong(bb.limit() - Bytes.SIZEOF\_LONG);

}

/\*

\* @param bb

\* @param offset

\* @return Amount to skip to get paste a byte array that is preceded by a

\* vint of how long it is.

\*/

private static int skipVintdByteArray(final ByteBuffer bb, final int offset) {

byte firstByte = bb.get(offset);

int vint = firstByte;

int vintWidth = WritableUtils.decodeVIntSize(firstByte);

if (vintWidth != 1) {

vint = getBigVint(vintWidth, firstByte, bb.array(),

bb.arrayOffset() + offset);

}

return vint + vintWidth + offset;

}

/\*

\* Vint is wider than one byte. Find out how much bigger it is.

\* @param vintWidth

\* @param firstByte

\* @param buffer

\* @param offset

\* @return

\*/

static int getBigVint(final int vintWidth, final byte firstByte,

final byte [] buffer, final int offset) {

long i = 0;

for (int idx = 0; idx < vintWidth - 1; idx++) {

final byte b = buffer[offset + 1 + idx];

i = i << 8;

i = i | (b & 0xFF);

}

i = (WritableUtils.isNegativeVInt(firstByte) ? (i ^ -1L) : i);

if (i > Integer.MAX\_VALUE) {

throw new IllegalArgumentException("Calculated vint too large");

}

return (int)i;

}

/\*\*

\* Create a store key.

\* @param bb

\* @return HStoreKey instance made of the passed <code>b</code>.

\* @throws IOException

\*/

public static HStoreKey create(final ByteBuffer bb)

throws IOException {

return HStoreKey.create(bb.array(), bb.arrayOffset(), bb.limit());

}

/\*\*

\* Create a store key.

\* @param b Serialized HStoreKey; a byte array with a row only in it won't do.

\* It must have all the vints denoting r/c/ts lengths.

\* @return HStoreKey instance made of the passed <code>b</code>.

\* @throws IOException

\*/

public static HStoreKey create(final byte [] b) throws IOException {

return create(b, 0, b.length);

}

/\*\*

\* Create a store key.

\* @param b Serialized HStoreKey

\* @param offset

\* @param length

\* @return HStoreKey instance made of the passed <code>b</code>.

\* @throws IOException

\*/

public static HStoreKey create(final byte [] b, final int offset,

final int length)

throws IOException {

byte firstByte = b[offset];

int vint = firstByte;

int vintWidth = WritableUtils.decodeVIntSize(firstByte);

if (vintWidth != 1) {

vint = getBigVint(vintWidth, firstByte, b, offset);

}

byte [] row = new byte [vint];

System.arraycopy(b, offset + vintWidth,

row, 0, row.length);

// Skip over row.

int extraOffset = vint + vintWidth;

firstByte = b[offset + extraOffset];

vint = firstByte;

vintWidth = WritableUtils.decodeVIntSize(firstByte);

if (vintWidth != 1) {

vint = getBigVint(vintWidth, firstByte, b, offset + extraOffset);

}

byte [] column = new byte [vint];

System.arraycopy(b, offset + extraOffset + vintWidth,

column, 0, column.length);

// Skip over column

extraOffset += (vint + vintWidth);

return new HStoreKey(row, column, Bytes.toLong(b, offset + extraOffset));

}

/\*\*

\* Passed as comparator for memcache and for store files. See HBASE-868.

\* Use this comparing keys in the -ROOT\_ table.

\*/

public static class HStoreKeyRootComparator extends HStoreKeyMetaComparator {

@Override

protected int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return compareRootRows(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* Passed as comparator for memcache and for store files. See HBASE-868.

\* Use this comprator for keys in the .META. table.

\*/

public static class HStoreKeyMetaComparator extends HStoreKeyComparator {

@Override

protected int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return compareMetaRows(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* Passed as comparator for memcache and for store files. See HBASE-868.

\*/

public static class HStoreKeyComparator extends WritableComparator {

public HStoreKeyComparator() {

super(HStoreKey.class);

}

@Override

@SuppressWarnings("unchecked")

public int compare(final WritableComparable l,

final WritableComparable r) {

HStoreKey left = (HStoreKey)l;

HStoreKey right = (HStoreKey)r;

// We can be passed null

if (left == null && right == null) return 0;

if (left == null) return -1;

if (right == null) return 1;

byte [] lrow = left.getRow();

byte [] rrow = right.getRow();

int result = compareRows(lrow, 0, lrow.length, rrow, 0, rrow.length);

if (result != 0) {

return result;

}

result = left.getColumn() == null && right.getColumn() == null? 0:

left.getColumn() == null ? -1:right.getColumn() == null? 1:

Bytes.compareTo(left.getColumn(), right.getColumn());

if (result != 0) {

return result;

}

// The below older timestamps sorting ahead of newer timestamps looks

// wrong but it is intentional. This way, newer timestamps are first

// found when we iterate over a memcache and newer versions are the

// first we trip over when reading from a store file.

if (left.getTimestamp() < right.getTimestamp()) {

result = 1;

} else if (left.getTimestamp() > right.getTimestamp()) {

result = -1;

}

return result; // are equal

}

protected int compareRows(final byte [] left, final int loffset,

final int llength, final byte [] right, final int roffset,

final int rlength) {

return Bytes.compareTo(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* StoreKeyComparator for the -ROOT- table.

\*/

public static class RootStoreKeyComparator

extends MetaStoreKeyComparator {

private static final long serialVersionUID = 1L;

@Override

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return compareRootRows(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* StoreKeyComparator for the .META. table.

\*/

public static class MetaStoreKeyComparator extends StoreKeyComparator {

@Override

public int compareRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

return compareMetaRows(left, loffset, llength, right, roffset, rlength);

}

}

/\*

\* @param left

\* @param loffset

\* @param llength

\* @param right

\* @param roffset

\* @param rlength

\* @return Result of comparing two rows from the -ROOT- table both of which

\* are of the form .META.,(TABLE,REGIONNAME,REGIONID),REGIONID.

\*/

protected static int compareRootRows(byte [] left, int loffset, int llength,

byte [] right, int roffset, int rlength) {

// Rows look like this: .META.,ROW\_FROM\_META,RID

// System.out.println("ROOT " + Bytes.toString(left, loffset, llength) +

// "---" + Bytes.toString(right, roffset, rlength));

int lmetaOffsetPlusDelimiter = loffset + 7; // '.META.,'

int leftFarDelimiter = getDelimiterInReverse(left, lmetaOffsetPlusDelimiter,

llength - lmetaOffsetPlusDelimiter, HRegionInfo.DELIMITER);

int rmetaOffsetPlusDelimiter = roffset + 7; // '.META.,'

int rightFarDelimiter = getDelimiterInReverse(right,

rmetaOffsetPlusDelimiter, rlength - rmetaOffsetPlusDelimiter,

HRegionInfo.DELIMITER);

if (leftFarDelimiter < 0 && rightFarDelimiter >= 0) {

// Nothing between .META. and regionid. Its first key.

return -1;

} else if (rightFarDelimiter < 0 && leftFarDelimiter >= 0) {

return 1;

} else if (leftFarDelimiter < 0 && rightFarDelimiter < 0) {

return 0;

}

int result = compareMetaRows(left, lmetaOffsetPlusDelimiter,

leftFarDelimiter - lmetaOffsetPlusDelimiter,

right, rmetaOffsetPlusDelimiter,

rightFarDelimiter - rmetaOffsetPlusDelimiter);

if (result != 0) {

return result;

}

// Compare last part of row, the rowid.

leftFarDelimiter++;

rightFarDelimiter++;

result = compareRowid(left, leftFarDelimiter, llength - leftFarDelimiter,

right, rightFarDelimiter, rlength - rightFarDelimiter);

return result;

}

/\*

\* @param left

\* @param loffset

\* @param llength

\* @param right

\* @param roffset

\* @param rlength

\* @return Result of comparing two rows from the .META. table both of which

\* are of the form TABLE,REGIONNAME,REGIONID.

\*/

protected static int compareMetaRows(final byte[] left, final int loffset,

final int llength, final byte[] right, final int roffset,

final int rlength) {

// System.out.println("META " + Bytes.toString(left, loffset, llength) +

// "---" + Bytes.toString(right, roffset, rlength));

int leftDelimiter = getDelimiter(left, loffset, llength,

HRegionInfo.DELIMITER);

int rightDelimiter = getDelimiter(right, roffset, rlength,

HRegionInfo.DELIMITER);

if (leftDelimiter < 0 && rightDelimiter >= 0) {

// Nothing between .META. and regionid. Its first key.

return -1;

} else if (rightDelimiter < 0 && leftDelimiter >= 0) {

return 1;

} else if (leftDelimiter < 0 && rightDelimiter < 0) {

return 0;

}

// Compare up to the delimiter

int result = Bytes.compareTo(left, loffset, leftDelimiter - loffset,

right, roffset, rightDelimiter - roffset);

if (result != 0) {

return result;

}

// Compare middle bit of the row.

// Move past delimiter

leftDelimiter++;

rightDelimiter++;

int leftFarDelimiter = getRequiredDelimiterInReverse(left, leftDelimiter,

llength - (leftDelimiter - loffset), HRegionInfo.DELIMITER);

int rightFarDelimiter = getRequiredDelimiterInReverse(right,

rightDelimiter, rlength - (rightDelimiter - roffset),

HRegionInfo.DELIMITER);

// Now compare middlesection of row.

result = Bytes.compareTo(left, leftDelimiter,

leftFarDelimiter - leftDelimiter, right, rightDelimiter,

rightFarDelimiter - rightDelimiter);

if (result != 0) {

return result;

}

// Compare last part of row, the rowid.

leftFarDelimiter++;

rightFarDelimiter++;

result = compareRowid(left, leftFarDelimiter,

llength - (leftFarDelimiter - loffset),

right, rightFarDelimiter, rlength - (rightFarDelimiter - roffset));

return result;

}

private static int compareRowid(byte[] left, int loffset, int llength,

byte[] right, int roffset, int rlength) {

return Bytes.compareTo(left, loffset, llength, right, roffset, rlength);

}

/\*\*

\* RawComparator for plain -- i.e. non-catalog table keys such as

\* -ROOT- and .META. -- HStoreKeys. Compares at byte level. Knows how to

\* handle the vints that introduce row and columns in the HSK byte array

\* representation. Adds

\* {@link #compareRows(byte[], int, int, byte[], int, int)} to

\* {@link RawComparator}

\*/

public static class StoreKeyComparator implements RawComparator<byte []> {

public StoreKeyComparator() {

super();

}

public int compare(final byte[] b1, final byte[] b2) {

return compare(b1, 0, b1.length, b2, 0, b2.length);

}

public int compare(final byte [] b1, int o1, int l1,

final byte [] b2, int o2, int l2) {

// Below is byte compare without creating new objects. Its awkward but

// seems no way around getting vint width, value, and compare result any

// other way. The passed byte arrays, b1 and b2, have a vint, row, vint,

// column, timestamp in them. The byte array was written by the

// #write(DataOutputStream) method above. See it to better understand the

// below.

// Calculate vint and vint width for rows in b1 and b2.

byte firstByte1 = b1[o1];

int vint1 = firstByte1;

int vintWidth1 = WritableUtils.decodeVIntSize(firstByte1);

if (vintWidth1 != 1) {

vint1 = getBigVint(vintWidth1, firstByte1, b1, o1);

}

byte firstByte2 = b2[o2];

int vint2 = firstByte2;

int vintWidth2 = WritableUtils.decodeVIntSize(firstByte2);

if (vintWidth2 != 1) {

vint2 = getBigVint(vintWidth2, firstByte2, b2, o2);

}

// Compare the rows.

int result = compareRows(b1, o1 + vintWidth1, vint1,

b2, o2 + vintWidth2, vint2);

if (result != 0) {

return result;

}

// Update offsets and lengths so we are aligned on columns.

int diff1 = vintWidth1 + vint1;

o1 += diff1;

l1 -= diff1;

int diff2 = vintWidth2 + vint2;

o2 += diff2;

l2 -= diff2;

// Calculate vint and vint width for columns in b1 and b2.

firstByte1 = b1[o1];

vint1 = firstByte1;

vintWidth1 = WritableUtils.decodeVIntSize(firstByte1);

if (vintWidth1 != 1) {

vint1 = getBigVint(vintWidth1, firstByte1, b1, o1);

}

firstByte2 = b2[o2];

vint2 = firstByte2;

vintWidth2 = WritableUtils.decodeVIntSize(firstByte2);

if (vintWidth2 != 1) {

vint2 = getBigVint(vintWidth2, firstByte2, b2, o2);

}

// Compare columns.

// System.out.println("COL <" + Bytes.toString(b1, o1 + vintWidth1, vint1) +

// "> <" + Bytes.toString(b2, o2 + vintWidth2, vint2) + ">");

result = Bytes.compareTo(b1, o1 + vintWidth1, vint1,

b2, o2 + vintWidth2, vint2);

if (result != 0) {

return result;

}

// Update offsets and lengths.

diff1 = vintWidth1 + vint1;

o1 += diff1;

l1 -= diff1;

diff2 = vintWidth2 + vint2;

o2 += diff2;

l2 -= diff2;

// The below older timestamps sorting ahead of newer timestamps looks

// wrong but it is intentional. This way, newer timestamps are first

// found when we iterate over a memcache and newer versions are the

// first we trip over when reading from a store file.

for (int i = 0; i < l1; i++) {

int leftb = b1[o1 + i] & 0xff;

int rightb = b2[o2 + i] & 0xff;

if (leftb < rightb) {

return 1;

} else if (leftb > rightb) {

return -1;

}

}

return 0;

}

/\*\*

\* @param left

\* @param right

\* @return Result comparing rows.

\*/

public int compareRows(final byte [] left, final byte [] right) {

return compareRows(left, 0, left.length, right, 0, right.length);

}

/\*\*

\* @param left

\* @param loffset

\* @param llength

\* @param right

\* @param roffset

\* @param rlength

\* @return Result comparing rows.

\*/

public int compareRows(final byte [] left, final int loffset,

final int llength, final byte [] right, final int roffset, final int rlength) {

return Bytes.compareTo(left, loffset, llength, right, roffset, rlength);

}

}

/\*\*

\* @param hri

\* @return Compatible comparator

\*/

public static WritableComparator getWritableComparator(final HRegionInfo hri) {

return hri.isRootRegion()?

new HStoreKey.HStoreKeyRootComparator(): hri.isMetaRegion()?

new HStoreKey.HStoreKeyMetaComparator():

new HStoreKey.HStoreKeyComparator();

}

/\*\*

\* @param hri

\* @return Compatible raw comparator

\*/

public static StoreKeyComparator getRawComparator(final HRegionInfo hri) {

return hri.isRootRegion() ? ROOT\_COMPARATOR :

hri.isMetaRegion() ? META\_COMPARATOR : PLAIN\_COMPARATOR;

}

/\*\*

\* @param tablename

\* @return Compatible raw comparator

\*/

public static HStoreKey.StoreKeyComparator getComparator(final byte [] tablename) {

return Bytes.equals(HTableDescriptor.ROOT\_TABLEDESC.getName(), tablename)?

ROOT\_COMPARATOR:

(Bytes.equals(HTableDescriptor.META\_TABLEDESC.getName(),tablename))?

META\_COMPARATOR: PLAIN\_COMPARATOR;

}

}