

HBase For Developers

- NoSql Overview
- HDFS and HBase Architecture
- Installing and Configuring HDFS and HBase Cluster
- Using HBase Shell and Java APIs and Writing Advanced Filters
- Writing Co-processors and Hive Queries
- HBase Schema Design & Advanced Management Operations
- Performance Tuning & Sizing an HBase Cluster



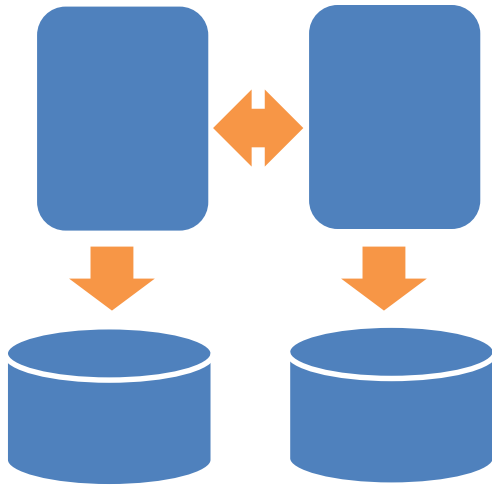
What is NoSQL?



Limitations of RDBMS

- Inability to handle several terabytes of data
- Inability to serve millions of read and writes concurrently
- Upgrading Server Capacity – Scale Up is not an option always
- Scale Out or Horizontal Scalability
- Flexibility in defining Schema
 - Everything can not be defined in terms of rows and columns
- Impedance Mismatch – Objects to Databases (ORM)
- Not big enough to handle big data

Deployment Models of RDBMS

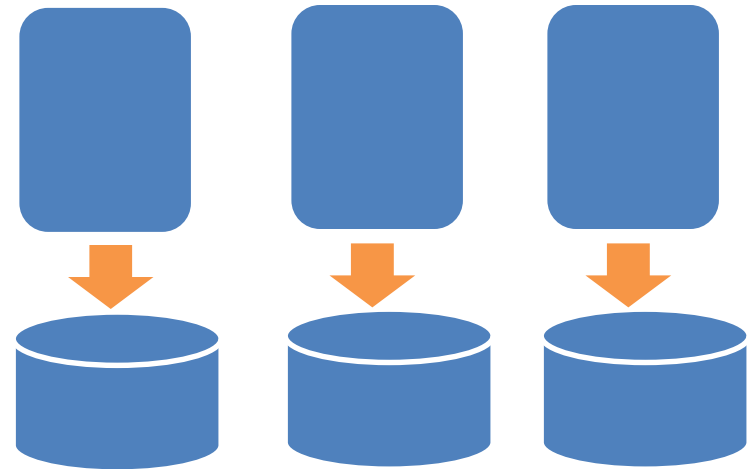


Clustering

- Load Balancing
- Availability
- Not very highly scalable solution considering the syncing of nodes becomes an overhead if scaled to hundreds of servers

Sharding

- Design time decision on number of shards
- If a shard becomes large, it need to be split again. An expensive task. No auto sharding
- Managing a large number of shards is a complex task
- Applications need to be aware of shards



Complex OR Mappings



Users



Videos, Files, Images

Name: _____
 Last First Middle

Address: _____
 Street City State Zip

Home Phone: _____ **Birthday:** _____

Email Address: _____

Employment: _____ **May we call you at work (Y or N)**

Marital Status: _____ **Work Phone** _____

Personal Information

00000021:000000a2:0003	LOP_LOCK_XACT	LCX_NULL	0000:00000544	0	0x0000	24	44
00000021:000000a2:0004	LOP_INSERT_ROWS	LCX_CLUSTERED	0000:00000544	0	0x0000	62	196
00000021:000000a2:0005	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:00000544	0	0x0000	62	164
00000021:000000a2:0006	LOP_INSERT_ROWS	LCX_CLUSTERED	0000:00000544	0	0x0000	62	132
00000021:000000a2:0007	LOP_INSERT_ROWS	LCX_CLUSTERED	0000:00000544	0	0x0000	62	128
00000021:000000a2:0008	LOP_INSERT_ROWS	LCX_INDEX_LEAF	0000:00000544	0	0x0000	62	116
00000021:000000a2:0009	LOP_MODIFY_ROW	LCX_CLUSTERED	0000:00000544	0	0x0000	62	1172

Transaction or Activity Logs

Key Concepts

- CAP
 - Consistency, Availability & Partition-tolerance
- Strong Vs Immediate consistency
- ACID Vs. BASE
 - BASE – Basic Availability, Soft State, Eventual Consistency
- Auto – sharding
 - Shards split itself if grows large
- Elastic – Linearly scalable
- Commodity Hardware
 - How to build a large scale database economically

Types of NoSql Databases

- Key Value Store
 - A key and plain value
 - Amazon Dynamo
- Document Store
 - Stored as key and value, where the value can be a structure, like a JSON object. Each "document" can have all, some, or none of the same elements as another
 - Mongo DB
- Column Families
 - Each row (addressed by a key) contains one or more "columns". Columns are themselves key-value pairs. The column names need not be predefined, i.e. the structure isn't fixed.
 - Columns are grouped as set of families and stored against a key
 - Big Table, HBase

What is Big Data

- IBM's definition – Big Data Characteristic
 - <http://www-01.ibm.com/software/data/bigdata/>

Volume

12 terabytes of Tweets
created each day

Velocity

Scrutinize 5 million trade
events created each day to
identify potential fraud

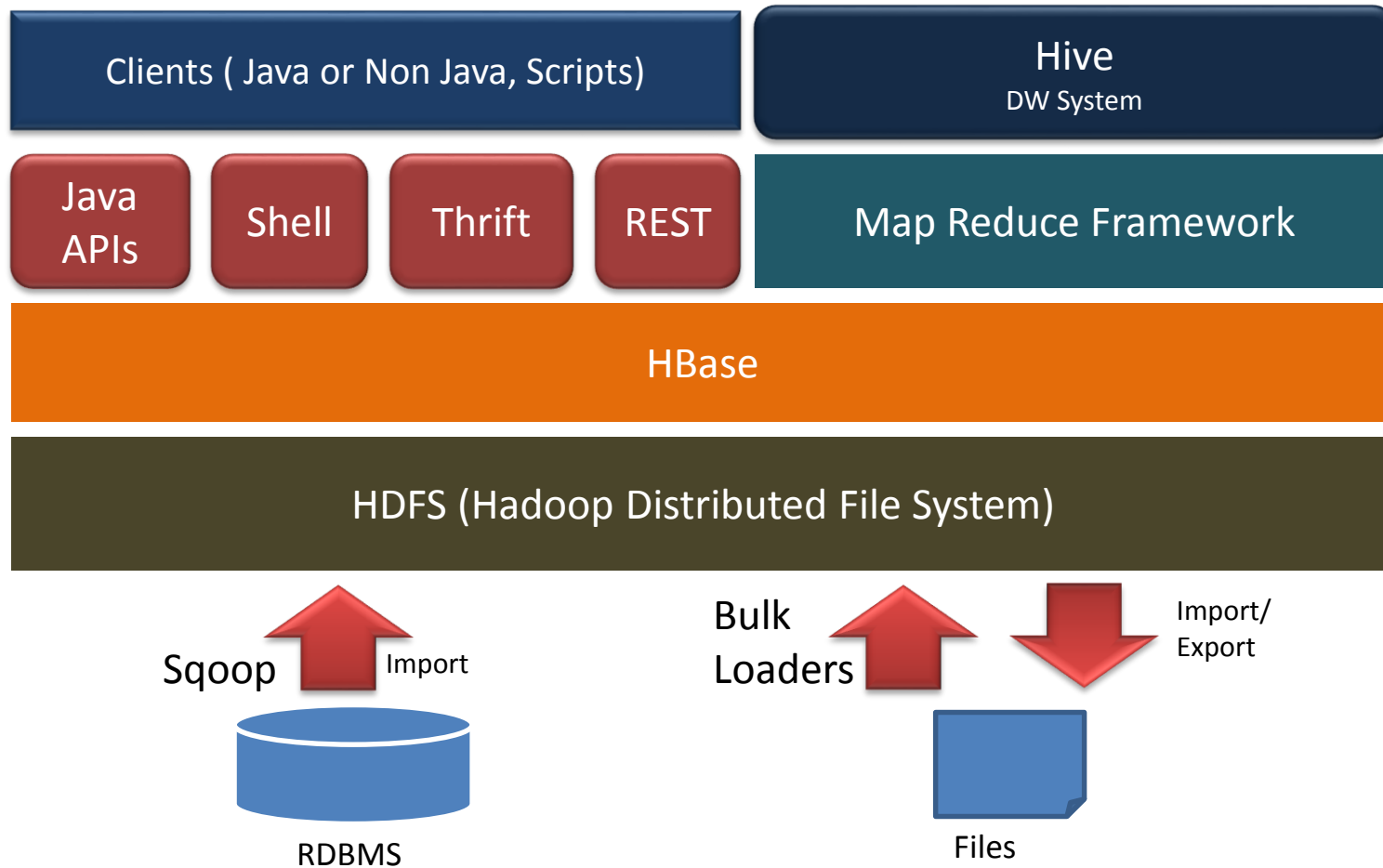
Variety

Sensor data, audio, video,
click streams, log files and
more

HBase

- Open Source Apache Project
- Consistency and Partition Tolerance (CP)
- Can host large tables with billions of rows and thousands of columns
- Auto sharding of tables when it grows
- Can be deployed on hundreds of commodity hardware
- Automatic Failovers
- Who is using it?
 - Facebook – messaging platform
 - Twitter – Read/write backup of all mysql tables in Twitter's production backend
 - Yahoo, Adobe, StumpleUpon, Ning etc.

HBase Ecosystem



Hbase Architecture



HBase Data Model - Regions

4000001			
4000002			
4000003			
4000004			
4000005			
4000006			
4000007			
4000008			
4000009			
4000010			
4000011			
4000012			
4000013			
4000014			
4000015			
4000016			
4000017			
4000018			
4000019			
4000020			
4000021			
4000022			



4000001			
4000002			
4000003			
4000004			
4000005			
4000006			
4000007			
4000008			
4000009			
4000010			
4000011			
4000012			
4000013			
4000014			
4000015			
4000016			
4000017			
4000018			
4000019			
4000020			
4000021			
4000022			

Region 1

Region 2

Region 3

Row Key

Each Shard is called a Region

HBase Architecture

HBase
Master

- Manages the cluster
- Allocates the regions to region server
- Interface for all metadata changes

Zoo
Keeper

- Redirects clients to appropriate region servers for reads and writes

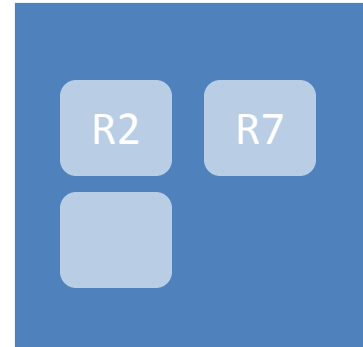
Region
Server 1



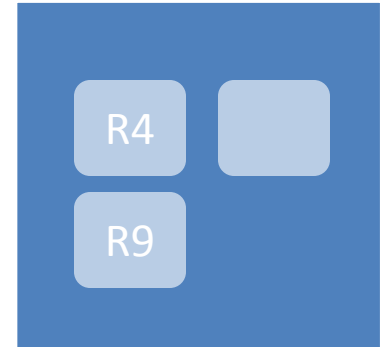
Region
Server 2



Regions
Server 3

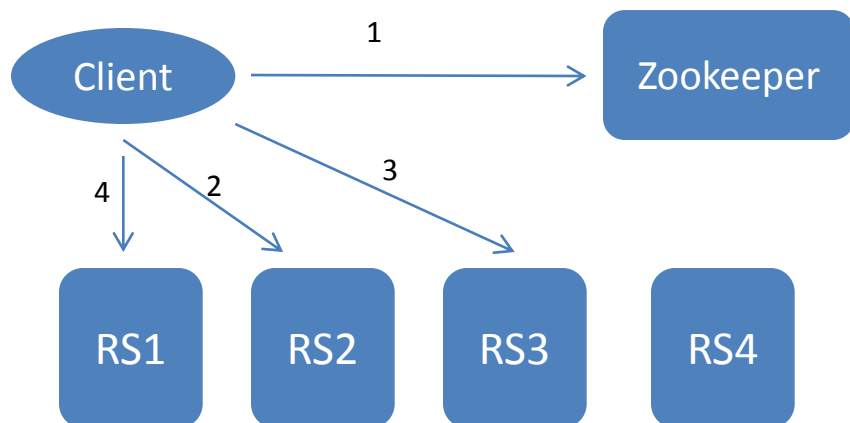


Region
Server 4



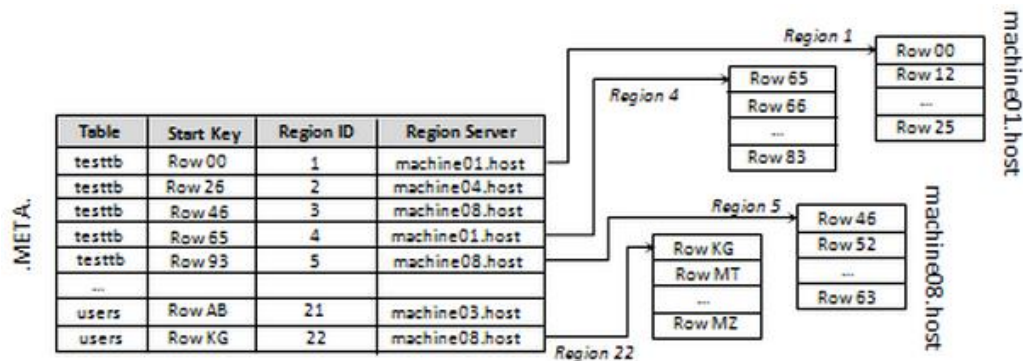
- Manages data
- Serves the read and write operations for the regions it is managing

-ROOT- and .META. file



- ROOT and META files are also HBase tables and stored as regions
- ROOT table is never split and stored as one region only
- META file can be split into multiple regions as it grows
- Whereabouts of ROOT file is stored in Zookeeper

1. Client asks zookeeper: where is – ROOT- file
2. Client goes to appropriate regions server containing -ROOT- file and ask which .META. Region can tell whereabouts of a row key in a table
3. Client goes to the .META. region and asks about the location of a key
4. Client goes to the region server and asks for the row



Row, Column Families and Column Qualifiers

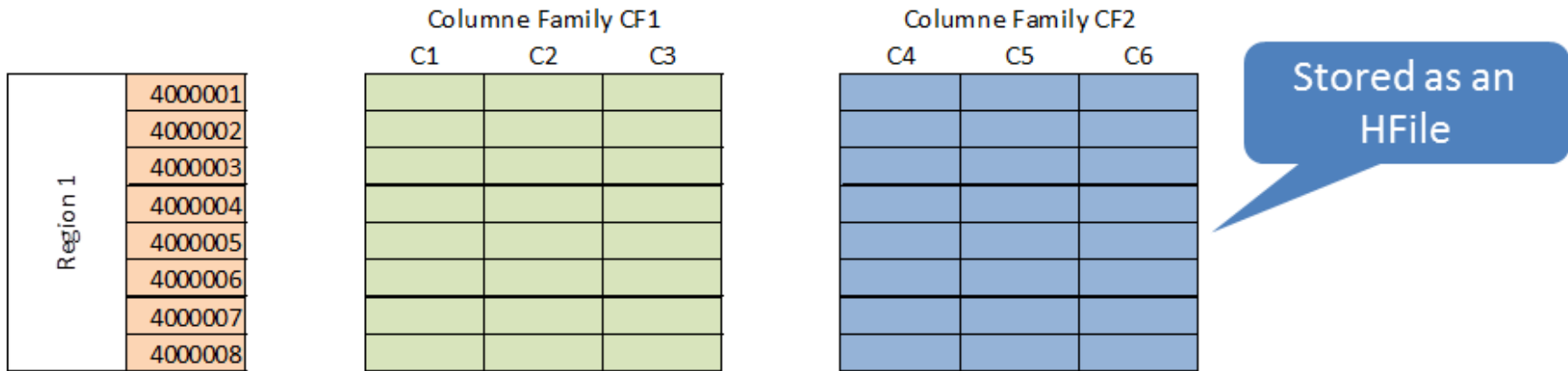
		Column Family CF1			Column Family CF2		
		C1	C2	C3	C4	C5	C6
Region 1	4000001						
	4000002						
	4000003						
	4000004						
	4000005						
	4000006						
	4000007						
	4000008						
Region 2	4000009						
	4000010						
	4000011						
	4000012						
	4000013						
	4000014						
	4000015						
	4000016						
Region 3	4000017						
	4000018						
	4000019						
	4000020						
	4000021						
	4000022						

Stored as an
HFile

CF1 contains
customer
information like
first name, last
name etc.

CF2 contains
transaction
information like
transaction id,
product name,
amount spent
etc.

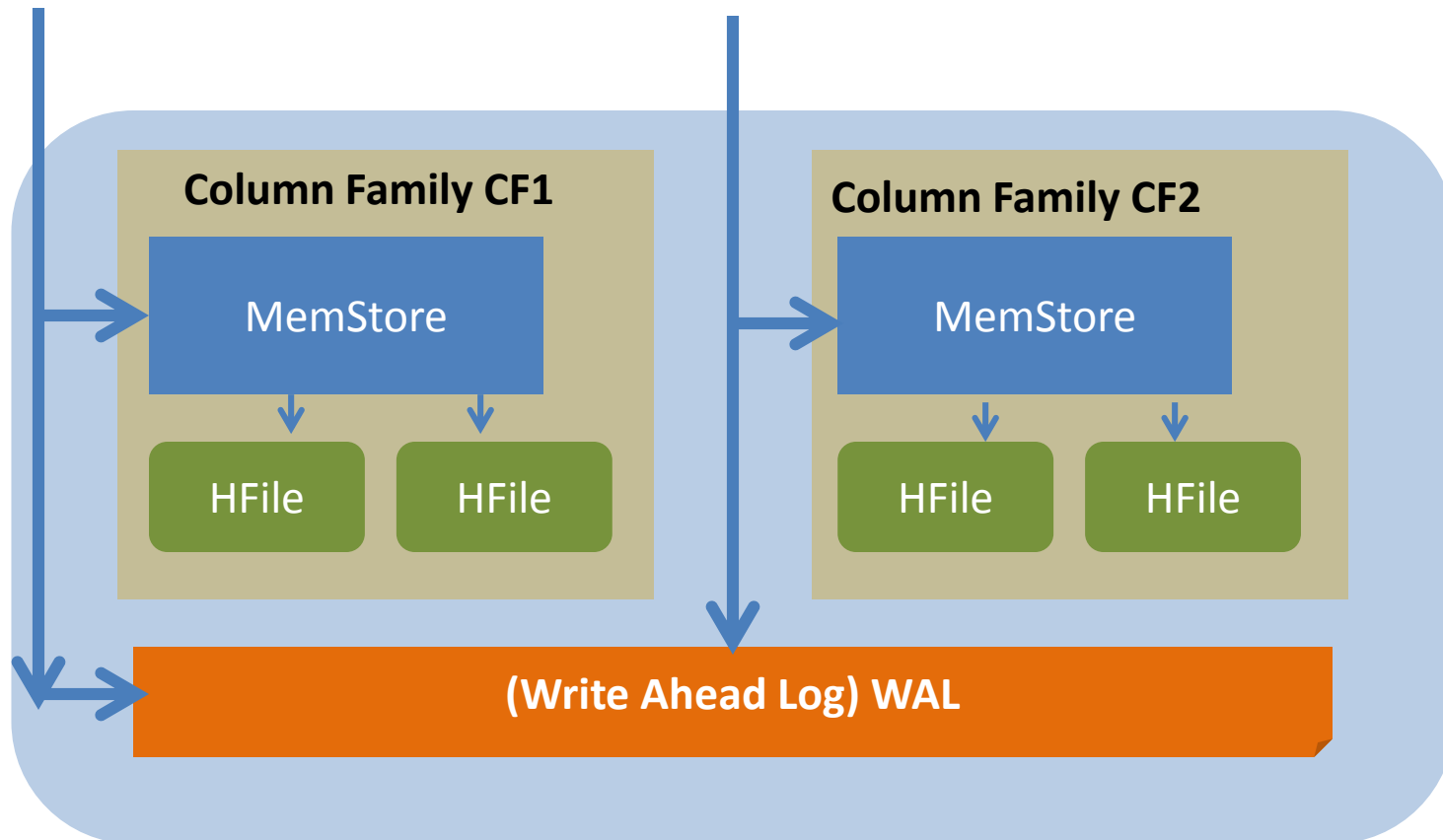
Physical Storage and Versioned Data



Column Family: personalinfo			
row key	column name	timestamp	value
8099099	first name	t3	Alex
8099099	last name	t2	Chen
8099099	age	t2	30
8099099	profession	t2	Teacher
8099099	age	t1	28

- The row key is treated by HBase as an array of bytes but it must have a string representation.
- It supports only Bytes for column values, so all types must be converted to bytes before storing in column values.
- Each column value is stored as a separate row along with row-key, column qualifier name, timestamp and value.

Writing Data



- MemStore is flushed to disk periodically to new HFiles.
- Every write is also updated in WAL, in case the MemStore crashes before it gets flushed to disk. So, the updates can be retrieved from WAL.
- One HFile is created for each column family.

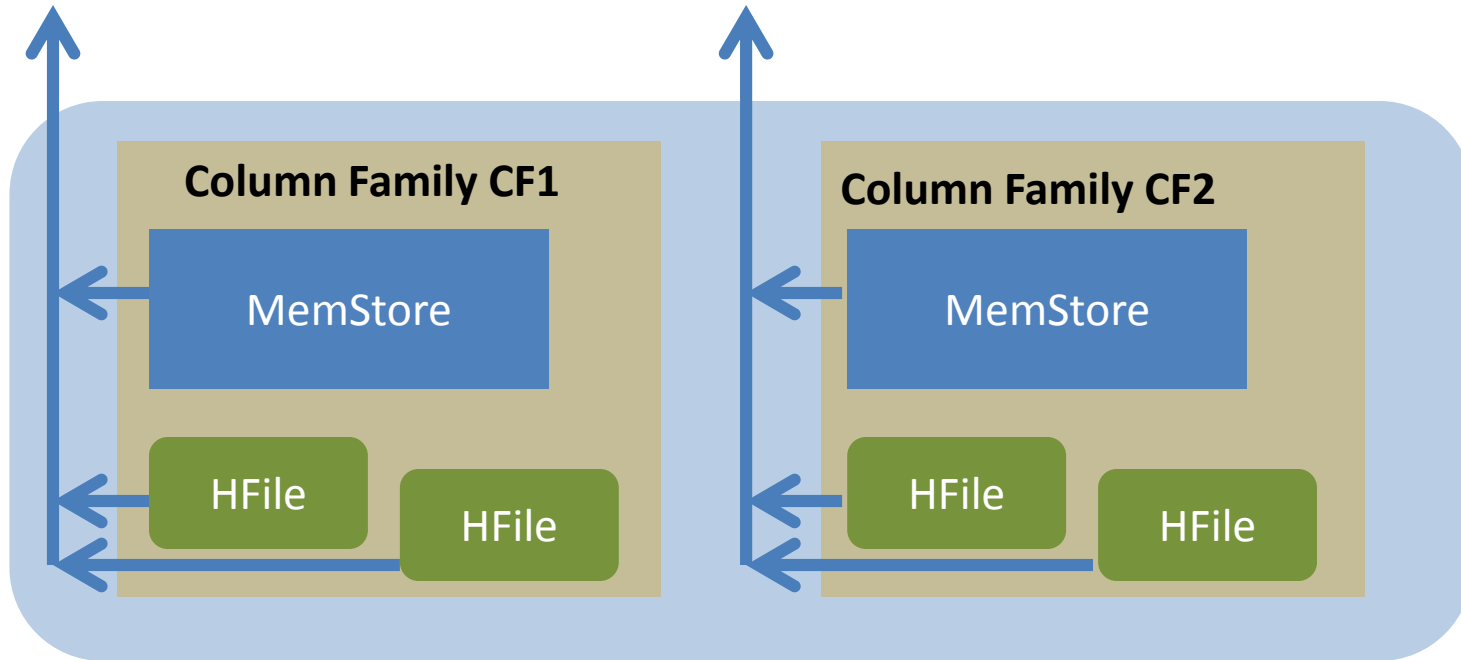
MemStore & HFile

- Data is always written to MemStore first and then an entry is made in WAL (required to recover from failures)
- One MemStore per column family
- MemStore is flushed to disk based on size
 - **hbase.hregion.memstore.flush.size** (default is 64 MB)
- By default 40% of region server heap size is allocated for MemStore
- It uses snapshot technique to write flush to disk while it is still serving reads and writes
- Data is stored in BLOCKs in HFiles (64 KB blocks)
 - HFile maintains an index of which key resides in which BLOCK
 - Create larger BLOCKs for sequential reads
 - Create smaller blocks for Random Reads

Writing Data

- Update to rows create new cell values with newer timestamp
- Delete are only marked for later removal
- Minor Compaction
 - Combines smaller HFiles and merges them
- Major Compaction
 - Rewrites the storage files – creates single HFile for column family per region
 - Deletes are dropped, values exceeding TTL are also dropped
 - Triggered by time threshold and can also be triggered manually
- The WAL is called HLog
- Writing to WALs can be deferred by writing log in memory and then flushing it later to disk.
 - This will improve write performance
 - but at the cost of availability, in case the region server crashes before it gets flushed to disk

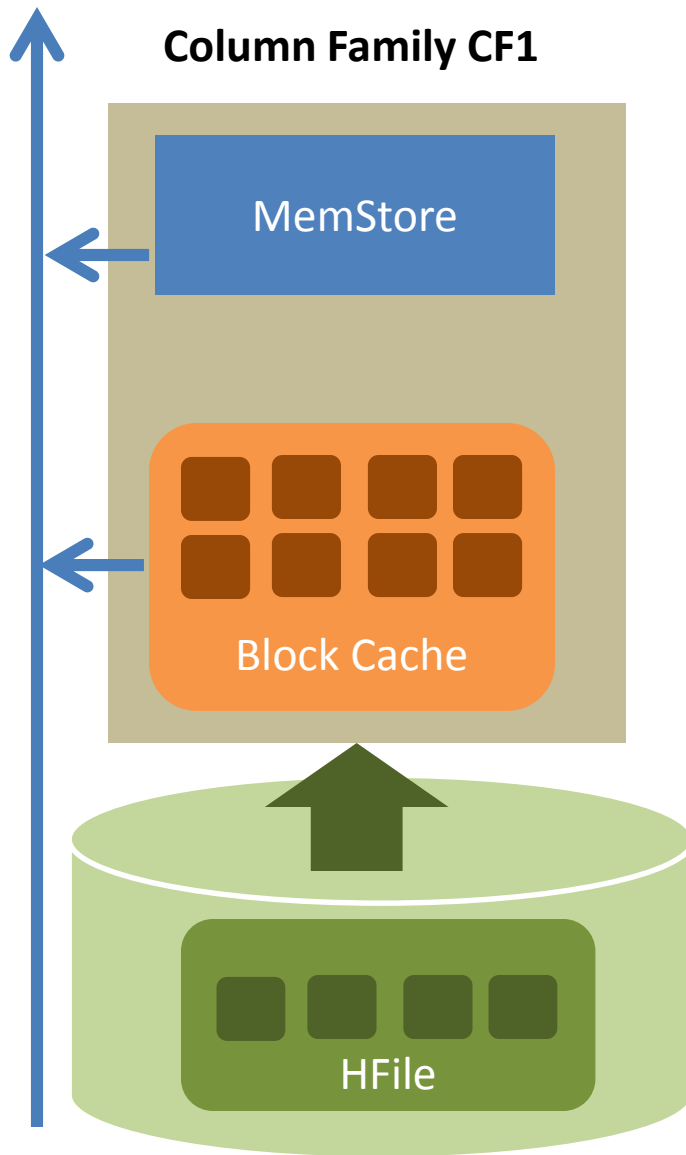
Reading Data



Different columns for a row-key might be present in different HFiles and MemStore depending on when they were written

After reading all the HFiles and MemStore, it consolidates everything for the row key and returns the results.

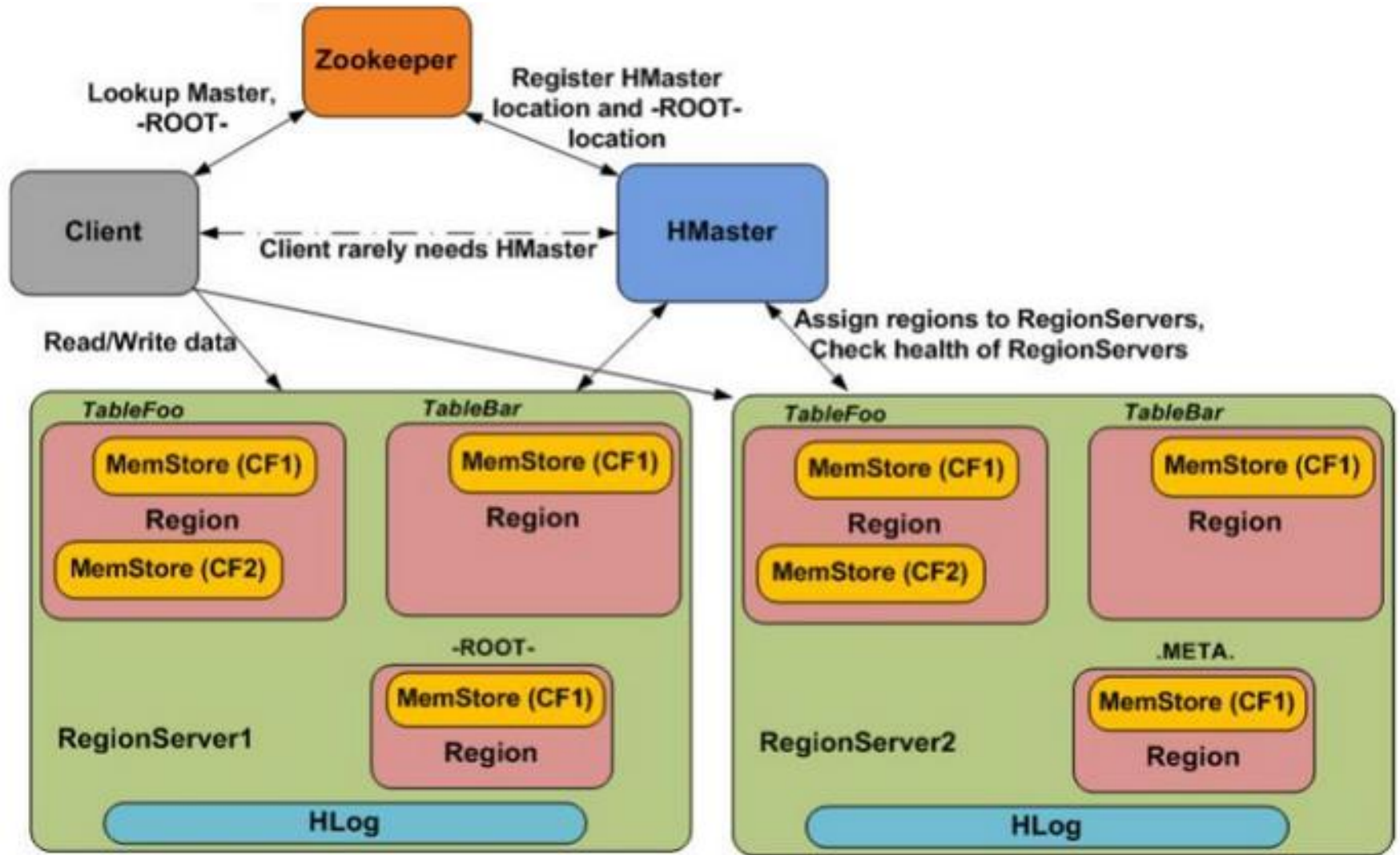
BLOCK Cache



Single Access	Multi Access	In Memory
---------------	--------------	-----------

- Block Cache is LRU cache for faster reads
- One Block Cache per Column Family
- BLOCKs are read from HFile and stored in BLOCK Cache
- Frequently Accessed blocks are kept in Block Cache for longer durations
- By Default 20% of Region Server Heap is allocated for Block Cache

HBase Architecture



<http://blog.csdn.net/macyang/article/details/6300277>

Installing & Configuring Hbase



Configure \$HBASE_HOME/conf/hbase-site.xml

Property	Value	Description
hbase.rootdir	hdfs://localhost:8020/hbase	The hdfs directory shared by RegionServers, where hbase will store all it's files including hfiles
hbase.cluster.distributed	true	The mode the cluster will be in. Possible values are false: standalone true: distributed
hbase.zookeeper.quorum	localhost	Comma separated list of servers in the ZooKeeper Quorum. Typical cluster deployment will have 3 zookeeper servers running.
hbase.tmp.dir	/home/notroot/lab/hbase/tmp	
hbase.local.dir	/home/notroot/lab/hbase/local	

Configure \$HBASE_HOME/conf/hbase-site.xml

Property	Value	Description
hbase.zookeeper. property.dataDir	/home/notroot/lab/zookeeper	Property from ZooKeeper's config zoo.cfg. The directory where the snapshot is stored.
hbase.zookeeper. property.clientPort	2181	Property from ZooKeeper's config zoo.cfg. The port at which the clients will connect.
hbase.master.port	16000	The port the HBase Master should bind to.
hbase.regionserver.port	16020	The port the HBase RegionServer binds to.

Configure hadoop-env.sh and hbase-env.sh

- Configure slaves in regionservers file
- Set JAVA_HOME in \$HADOOP_HOME/conf/hadoop-env.sh

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
```
- Set JAVA_HOME and ZooKeeper settings in \$HBASE_HOME/conf/hbase-env.sh

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64  
export HBASE_MANAGES_ZK=true
```
- Link hdfs-site.xml file from hbase conf directory to hadoop conf directory

In -s \$HADOOP_HOME/conf/hdfs-site.xml \$HBASE_HOME/conf/hdfs-site.xml

Format NameNode and Start All Services

- Format the namenode
 - `hadoop namenode -format`
- Start HDFS services
 - `$ <HADOOP_INSTALL/bin> ./start-dfs.sh`
- Start HBase Services
 - `$ <HBASE_HOME/bin> ./start-hbase.sh`
- Verify if all services are running
 - `jps`

```
notroot@ubuntu:~/lab/software/hbase-0.94.7/bin$ jps
3868 NameNode
4359 SecondaryNameNode
5526 HRegionServer
5302 HMaster
5726 Jps
5241 HQuorumPeer
4110 DataNode
notroot@ubuntu:~/lab/software/hbase-0.94.7/bin$
```

Using HBase Shell



Basic hbase shell commands

- Start hbase Shell
 - *hbase shell <enter>*
- List all tables
 - *list*
- Show hbase version
 - *version*
- Status of the cluster
 - *status*
 - *status "detailed"*
- Lists tools or commands for hbase operations
 - *tools*
- Shutdown the cluster
 - *shutdown*

Managing tables

- Create a table with a column family
 - *create 'emp', 'personalinfo'*
- Describe a table
 - *describe 'emp'*
- Drop a table (First disable and then drop the table)
 - *disable "emp"*
 - *drop "emp"*
- Recreate a table after deleting all data
 - *truncate "emp"*
- Add a new column family
 - *disable "emp"*
 - *alter 'emp', 'courses'*
 - *enable 'emp'*

Shell - CRUD Operations

- Add rows and columns
 - `put 'emp', '001', 'personalinfo:name', 'John'`
 - `put 'emp', '001', 'personalinfo:age', '35'`
- Update columns
 - `put 'emp', '001', 'personalinfo:age', '37'`
- Get all rows of the table
 - `scan 'emp'`
- Get selected rows or specific column families or columns
 - `get 'emp', '001' // all columns for all column families`
 - `get 'emp', '001', 'personalinfo' // all columns of a column family`
 - `get 'emp', '001', 'personalinfo:age' // a particular column`

Shell - CRUD Operations

- Get only limited number of rows starting from a specified key

```
scan 'custs',{ LIMIT => 2, STARTROW => '4000004' }
```

- Selecting rows whose columns have specific values

```
scan 'custs', { COLUMNS => 'personalinfo:age', FILTER => "ValueFilter( >, 'binary:60' )" }
```

```
scan 'custs', { COLUMNS => 'personalinfo:profession', LIMIT => 10, FILTER  
=> "ValueFilter( >, 'binary:art' )" }
```

- All data model operations HBase return data in sorted order.
 - First by row, then by ColumnFamily, followed by column qualifier, and finally timestamp (sorted in reverse, so newest records are returned first)

Shell - CRUD Operations

- Delete a cell or column value for an row

delete 'custs', '4000090', 'personalinfo:age'

- Deleting all roows

deleteall 'custs', '4000090'

- Count number of rows in a table

Count 'custs'

Using HBase Java APIs



Connecting to HBase Table

```
Configuration conf = HBaseConfiguration.create();  
conf.set("hbase.master", "192.168.217.135:60010");  
conf.set("hbase.zookeeper.quorum", "192.168.217.135");  
  
connection = HConnectionManager.createConnection(conf);  
hbaseTable = connection.getTable("custs");
```

Storing or Updating Rows and Column values

// Put will create a new record if it does not exist

```
Put put = new Put( "4000010".getBytes() );  
put.add( "personalinfo".getBytes(), "firstname".getBytes(), "john".getBytes() );  
put.add( "personalinfo".getBytes(), "age".getBytes(), "30".getBytes() );  
put.add( "personalinfo".getBytes(), "profession".getBytes(), "painter".getBytes() );
```

```
hbaseTable.put( put );
```

// Put is also used for updating data

```
Put put = new Put( "4000010".getBytes() );  
put.add( "personalinfo".getBytes(), "age".getBytes(), "30".getBytes() );
```

```
hbaseTable.put( put );
```

// checkAndPut() for updating if existing value matches expected value

```
checkAndPut( "4000010".getBytes(), "personalinfo".getBytes(), "age".getBytes(),  
"30".getBytes(), put );
```

Deleting Rows

// Delete the row

```
Delete delete = new Delete( "400010".getBytes() );  
hbaseTable.delete( delete );
```

// Delete specific column of the row

```
Delete delete = new Delete( "4000010".getBytes() );  
delete.deleteColumn( "personalinfo".getBytes(), "age".getBytes() );  
hbaseTable.delete( delete );
```

// Delete all columns of a column-family

```
Delete delete = new Delete( id.getBytes() );  
delete.deleteFamily( "personalinfo".getBytes() );  
hbaseTable.delete( delete );
```

// checkAndPut() for updating if existing value matches expected value

```
checkAndDelete( "40000010".getBytes(), "personalinfo".getBytes(), "age".getBytes(),  
"30".getBytes(), delete );
```

Retrieving rows and their columns

// Delete the row

```
Get g = new Get(Bytes.toBytes( id ));  
Result r = hbaseTable.get(g);  
byte[] val = r.getColumnLatest( "personalinfo".getBytes(), "firstname".getBytes()  
).getValue();
```

// Get Family Map

```
g.getFamilyMap();
```

// Get a specific column

```
g.addColumn( "personalinfo".getBytes(), "firstname".getBytes() );
```

// Get all columns for a specific column family only

```
g.addFamily( "personalinfo".getBytes() );
```

// Get all versions of the column

```
List<KeyValue> vals = r.getColumn( "personalinfo".getBytes(), "firstname".getBytes() );
```

// Set maximum number of version to be retrieved

```
g.setMaxVersions( 3 );
```

Retrieving all rows or range of rows

// Retrieve all rows

```
Scan s = new Scan();  
ResultScanner rs = hbaseTable.getScanner(s);  
for ( Result r: rs )  
{  
    // retrieve the details of each row  
}
```

// Retrive a range of records

```
Scan s = new Scan( startid.getBytes(), endid.getBytes() );  
ResultScanner rs = hbaseTable.getScanner(s);  
for ( Result r: rs )  
{  
    // retrieve the details of each row  
}
```

Comparison Operators and Comparators

- Comparison Operators
 - LESS
 - LESS_OR_EQUAL
 - EQUAL
 - NOT_EQUAL
 - GREATER_OR_EQUAL
 - GREATER
- Comparators
 - BinaryComparator - Compare the current with the provided value.
 - NullComparator – Verifies if a given one is null, or not null.
 - RegexStringComparator - Does a pattern match on the regular expression given
 - SubstringComparator - Performs a contains() check

Filters

// Returns all rows that has age value more than 25

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("age"),  
    CompareOp.GREATER_OR_EQUAL,  
    new BinaryComparator(Bytes.toBytes( "25" ) ) );
```

```
Scan s = new Scan();  
s.setFilter( filter );  
ResultScanner rs = hbaseTable.getScanner(s);
```

```
for ( Result r: rs )  
{  
    // retrieve row details  
}
```

Row Filter

// Returns all rows with less or equal to key value 4000100

```
Filter filter1 = new RowFilter(CompareOp.LESS_OR_EQUAL,  
new BinaryComparator(Bytes.toBytes("4000100")));
```

// Returns all rows with key values that ends with 10

```
Filter filter2 = new RowFilter(CompareOp.EQUAL,  
new RegexStringComparator("*10"));
```

// Randomly selects rows – useful for sampling

RandomRowFilter(float chance)

Chance is a value between 0 and 1.

SingleColumnValueFilter

// Returns all rows that has age value more than 25

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("age"),  
    CompareOp.GREATER_OR_EQUAL,  
    new BinaryComparator(Bytes.toBytes( "25" ) ) );
```

// Returns all rows that has first name "micheal"

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("age"),  
    CompareOp.EQUAL,  
    new SubstringComparator( "micheal" ) );
```

// Returns all rows DOES NOT match this filter

SingleColumnValueExcludeFilter

Family and Qualifier Filter

// Returns all rows that this column family

```
Filter filter1 = new FamilyFilter(CompareFilter.CompareOp.LESS,  
new BinaryComparator(Bytes.toBytes("courses")));
```

// Returns all rows that has this column

```
Filter filter = new  
QualifierFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,  
new BinaryComparator(Bytes.toBytes("math")));
```

Timestamps and Page Filter

// Returns all rows that has this timestamp

```
List<Long> ts = new ArrayList<Long>();  
ts.add(new Long( 92323090 ));  
ts.add(new Long( 12129019 ));  
ts.add(new Long( 12902129 ));  
Filter filter = new TimestampsFilter(ts);
```

// Returns only specified number of rows

```
Filter filter = new PageFilter(15);
```

// In this case the last row id need to be tracked and passed on to the next scan as start row as follows

```
Scan scan = new Scan();  
scan.setFilter(filter);  
scan.setStartRow(startRow);
```

Skip and While Match Filter

- Skips rows which qualifies the criteria
 - Define any filter and then add it to the skip filter

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("age"),  
    CompareOp.GREATER_OR_EQUAL,  
    new BinaryComparator(Bytes.toBytes( age )));
```

```
Filter filter2 = new SkipFilter( filter );  
scan.setFilter( filter2 );
```

- Retrieves all rows until this criteria is statisfied

```
Filter filter2 = new WhileMatchFilter(filter1);
```

Filter List

// Use multiple filters for query

```
List<Filter> filters = new ArrayList<Filter>(2);
```

```
SingleColumnValueFilter filter1 = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("age"),  
    CompareOp.GREATER_OR_EQUAL,  
    new BinaryComparator(Bytes.toBytes( age )));
```

```
SingleColumnValueFilter filter2 = new SingleColumnValueFilter(  
    Bytes.toBytes("personalinfo"),  
    Bytes.toBytes("profession"),  
    CompareOp.EQUAL,  
    new RegexStringComparator( profession ));
```

```
filters.add( filter1 );  
filters.add( filter2 );
```

```
FilterList filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL, filters);
```

```
Scan s = new Scan();  
s.setFilter( filterList );
```

Counters

- For collecting statistics.
- If a typical table column is used then reading, incrementing, writing it back, and in the process locking and unlocking the row for other writers to be able to access it can cause a lot of contention
- HBase provides a counter which can be modified with single API call

```
create 'counters', 'daily'
```

```
0 row(s) in 1.1930 seconds
```

```
incr 'counters', '20130818', 'daily:hits', 2
```

```
COUNTER VALUE = 2
```

```
get_counter 'counters', '20130818', 'daily:hits'
```

```
COUNTER VALUE = 2
```

```
HTable table = new HTable(conf, "counters");  
long cnt1 = table.incrementColumnValue( Bytes.toBytes("20110101"),  
Bytes.toBytes("daily"), Bytes.toBytes("hits"), 1 );
```


Observers

- Like triggers that is called before and after an event
- Regions Observers
 - prePut, postPut, preDelete, postDelete, preGet, postGet, preScan, postScan
 - preFlush, postFlush – flushing the memstore to hfiles
- WALObserver
 - Before and after writing events to WAL
- MasterObserver
 - Pre and post events like create, modify, delete tables

Writing an Observer

```
public class SampleObserver extends BaseRegionObserver {  
  
    @Override  
    public void start(CoprocessorEnvironment env) throws IOException {  
  
    }  
  
    @Override  
    public void postPut(ObserverContext<RegionCoprocessorEnvironment> e,  
        Put put,  
        WALEdit edit,  
        boolean writeToWAL)  
        throws IOException  
    {  
    }  
  
    @Override  
    public void stop(CoprocessorEnvironment env) throws IOException {  
  
    }  
}
```

- Add observer to the table
 - disable 'custsnew'
 - alter 'custsnew', METHOD => 'table_att', 'coprocessor' => 'file:///hbaseco.jar|com.enablecloud.hbase.custs.SampleObserver|1001|'
 - enable 'custsnew'
 - describe 'custsnew' // verify if coprocessor is added
 - alter 'custsnew', METHOD => 'table_att_unset', NAME => 'coprocessor\$1' // remove

Other Client Interfaces

- Thrift & AVRO
 - For writing non-java clients
 - Generate from available service definition files like hbase.thrift
- REST
 - Restful services – supports XML and Json messaging formats

Start and Stop the REST Gateway

```
./hbase-daemon.sh start|stop rest -p 7070
```

Using rest protocols

```
curl -H "Accept: application/json"
```

```
http://hadooplab.bigdataleap.com:7070/version
```

```
curl -H "Accept: application/json"
```

```
http://hadooplab.bigdataleap.com:7070/custs/4000010/personalinfo
```

Writing Java REST Client

```
// Connecting to REST Service gateway
Cluster cluster = new Cluster();
cluster.add("hadooplab.bigdataleap.com", 7070);

Client client = new Client(cluster);
RemoteHTable table = new RemoteHTable(client, "custs");

// Getting a specific record
Get get = new Get(Bytes.toBytes("4000010"));

Result r = table.get(get);
System.out.println( " Firstname : " + getColumnValue( r, "personalinfo", "firstname" ) );
System.out.println( " Lastname : " + getColumnValue( r, "personalinfo", "lastname" ) );
System.out.println( " Age : " + getColumnValue( r, "personalinfo", "age" ) );
System.out.println( " Profession : " + getColumnValue( r, "personalinfo", "profession" ) );

// Scanning a table for getting a range of records
Scan scan = new Scan();
scan.setStartRow(Bytes.toBytes("4000100"));
scan.setStopRow(Bytes.toBytes("4000120"));

ResultScanner scanner = table.getScanner(scan);
for (Result rscan : scanner) {
    System.out.println( " Firstname : " + getColumnValue( rscan, "personalinfo", "firstname" ) );
    System.out.println( " Lastname : " + getColumnValue( rscan, "personalinfo", "lastname" ) );
    System.out.println( " Age : " + getColumnValue( rscan, "personalinfo", "age" ) );
    System.out.println( " Profession : " + getColumnValue( rscan, "personalinfo", "profession" ) );
}

table.close();
```

Using Hive for Complex Queries

- Defining a Hive Table pointing to Hbase Table

```
CREATE EXTERNAL TABLE custshbase(custid int, firstname string,  
lastname string, age int, profession string)
```

```
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
```

```
WITH SERDEPROPERTIES ("hbase.columns.mapping" =  
":key,personalinfo:firstname,personalinfo:lastname,personalinfo:age,  
personalinfo:profession")
```

```
TBLPROPERTIES("hbase.table.name" = "custs");
```

- Once table is defined, it is as simple as writing an SQL Query
select profession, count(*) from custshbase group by profession;

Designing HBase Schema



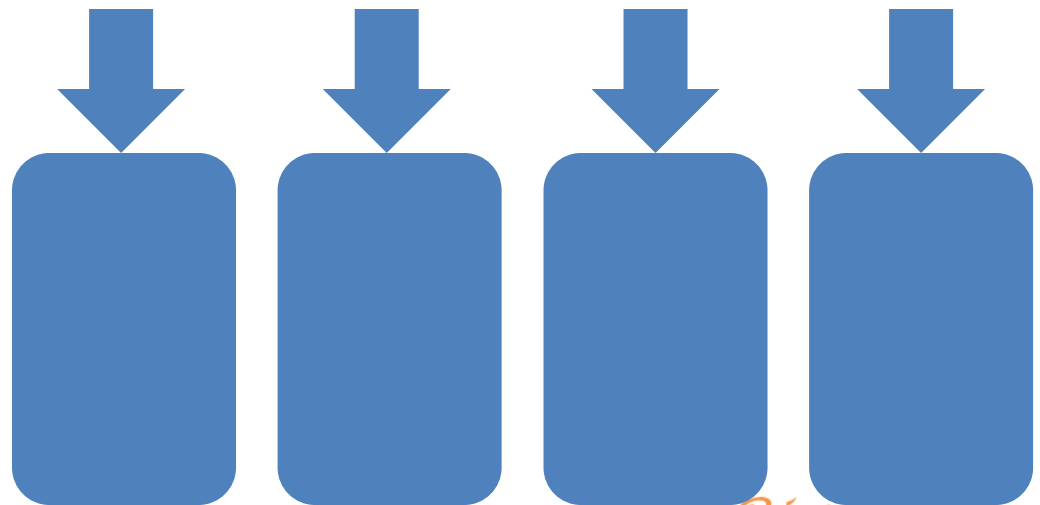
Schema Design

- HBase allows flexibility of adding column families later and column qualifiers dynamically
- Nevertheless designing the schema upfront is always a good practice
- Key points
 - Even distribution of load
 - Designing of row-keys
 - Application profiling
 - Read-heavy or write-heavy
 - Random Access or Sequential Access
 - Denormalization and Nested Entities
 - Overcoming Limitations of HBase Design

Even distribution of load - Avoid Hot-spotting



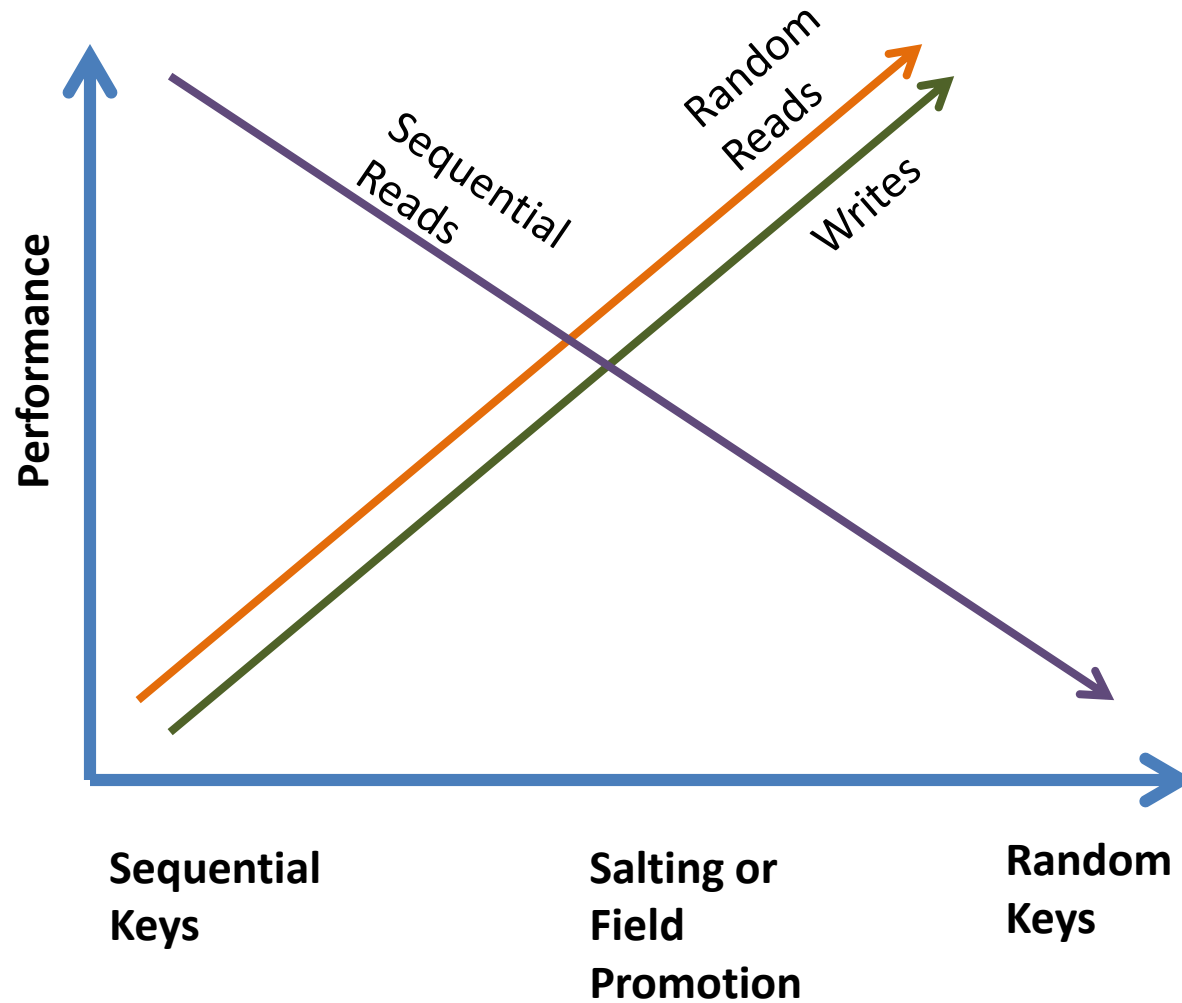
The key should be identified in such a way that the reads and writes are spread across regions almost evenly.



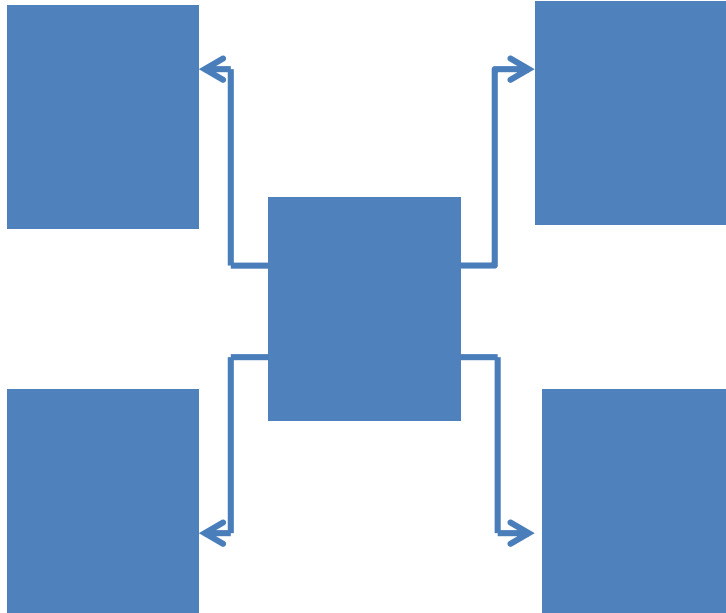
Designing Keys

- Event distribution of Writes
 - Salting – Adding a random number to the key
 - Prefix the natural keys with a random number
 - 0-key1, 1-key2, 2-key3,0-key4, 1-key5,2-key6
 - Distribute the keys to all regions in round robin fashion
 - Key Field promotion
 - Add one of the column qualifiers to the key itself. For example, add the system id to the timestamp
- Randomization
 - Using hashing algorithm – this would completely randomize the distributions

Choosing Keys



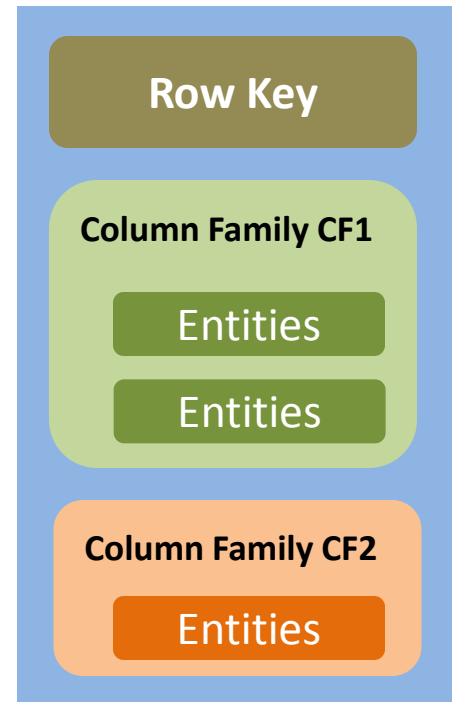
De-normalization & Nested Entities



Denormalize the data and keep under one row, that need to be retrieved simultaneously in a single access. This would avoid reading multiple rows or tables and joining the columns.

The column values can be entities if it can be stored as bytes.

Only one level of nesting is possible.



Key Considerations

- Tall Table or Fat Table
 - Depends on access patterns
- Group column qualifiers into column families depending on the access patterns
 - PersonalInfo – Name, Age etc.
 - Courses - Subjects
- Column qualifiers can be treated as data.
 - Name column qualifiers as data items like username, transaction id etc.
- Transactional context is at row level.
 - Avoid designing row-keys which will involve require complex transaction logic in client code

Key Considerations (Continued...)

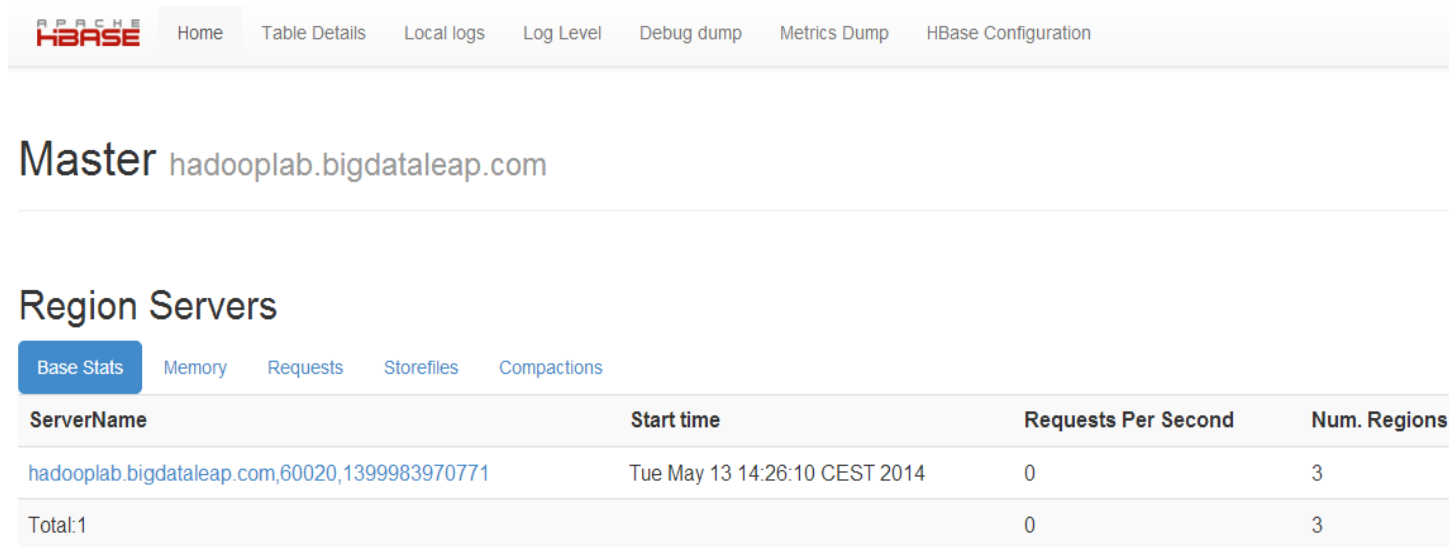
- The column family and qualifier names are stored in all cells along with the value.
 - Be concise while choosing names.
- Reverse Timestamp
 - If read required only last few rows based on timestamp then use `LONG.MAX_VALUE - timestamp`
- Secondary Indexes
 - There are no secondary indexes built for any other column. The only index available is for row-key
 - Secondary indexes can be built explicitly using another table – storing the value and row-key
 - Can be built in Batch Mode – But this will not reflect real time data
 - Can be built in real time using coprocessors – But will make writes costly

Advanced Hbase Operations



Monitoring and Managing Services & Regions

- Start and Stop services individually
 - `hbase-daemon.sh start|stop master`
 - `hbase-daemon.sh start|stop regionserver`
 - `hbase-daemon.sh start|stop master-backup`
- HBase master UI running on `http://masternodeip:60010/`



The screenshot shows the HBase Master UI for the instance `hadooplab.bigdataleap.com`. The navigation bar includes links for Home, Table Details, Local logs, Log Level, Debug dump, Metrics Dump, and HBase Configuration. The "Region Servers" section is active, displaying a table with the following data:

ServerName	Start time	Requests Per Second	Num. Regions
hadooplab.bigdataleap.com,60020,1399983970771	Tue May 13 14:26:10 CEST 2014	0	3
Total:1		0	3

Monitoring and Managing Services & Regions

- `hbase> status 'summary' | 'simple' | 'detailed'`
- Trigger minor compaction
 - `hbase shell> compact 'tablename'`
- Trigger major compaction
 - `hbase shell> major_compact 'tablename'`
- `Linux> hbase hbck`
 - No of live and dead regionserver nodes
 - Whereabouts of hbase files and tables (hdfs location)
 - No of regions for each table and allocation of keys (start and end keys) for each region
 - Health status of each table
- `Linux> hbase hbck -detailed`
 - Provides detailed mapping of region with region servers
 - Provides details of region and table inconsistencies

Pre-splitting the regions

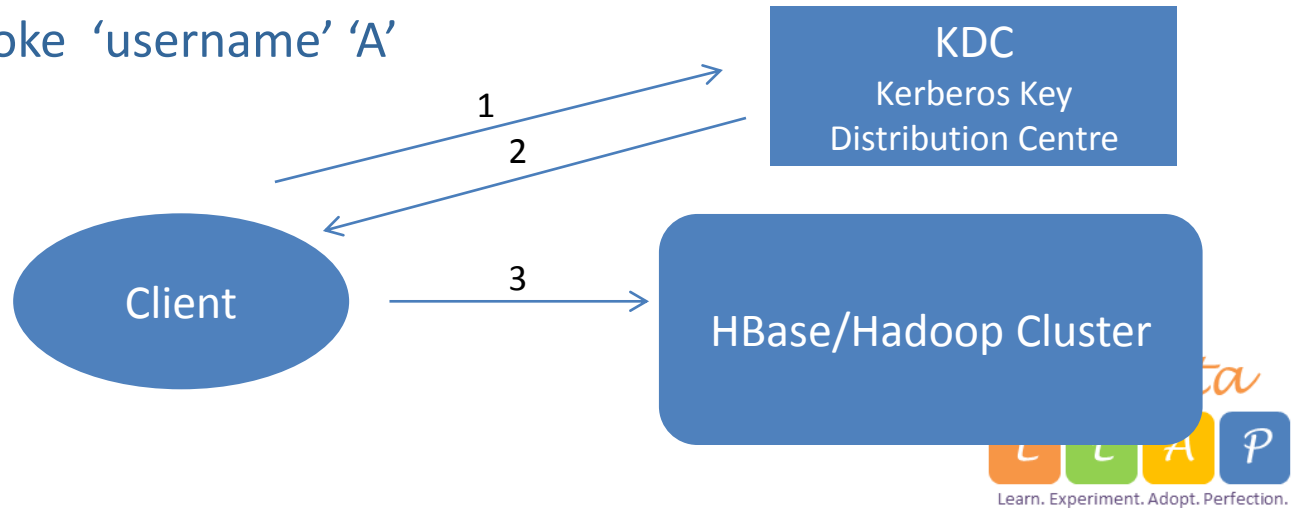
- Create a key distribution file
 - `cat keydist`
- Each line in the file specifies the start key for each region
- Create table schema using key distribution file
 - `create 'custs' , 'personalinfo', {SPLITS_FILE => '~/keydist'}`

```
400010
400050
400100
400150
400200
400250
```

Authentication Authorization

- Grant or Revoke Access
 - **Grant Read/Write Access**
 - `grant|revoke 'username' 'R' 'tablename', 'column-family'`
 - `Grant|revoke 'username' 'R' 'tablename', 'column-family', 'col-qualifier'`
 - **Create Table Access**
 - `grant|revoke 'username' 'C'`
 - **Manage Cluster Access – Admin Access**
 - `grant|revoke 'username' 'A'`

1. Client Authenticates
2. KDC grants a ticket
3. Client sends the ticket to Cluster



Performance Tuning



Performance Tuning

1. HBase Heap Size

- Configure larger heap size as today's machines come with large RAM
- export ***HBASE_HEAPSIZE=8000*** in ***\$HBASE_HOME/conf/hbase-env.sh*** file

2. Increase the number of handlers in region servers

- Default is 10, which is a low value. Increase to 40, if the underlying hardware configured is of higher capacity. Has more CPU and RAM available.
- Set ***hbase.regionserver.handler.count*** to ***40*** in hbase-site.xml

3. Increase BLOCK Size

- HFile BLOCKs are smallest unit of data the HBase reads from StoreFiles
- Increase the BLOCK Size to minimize the number of DISK IOs
 - ***Create / alter 'tablename', {NAME => 'n', BLOCKSIZE => '16384'}***

4. Enable Bloom Filter

- Bloom Filters are indexes stored along with data in Store Files
- ***Create / Alter 'tablename', {NAME => 'colfam1', BLOOMFILTER=> 'ROW/ROWCOL'}***

Performance Tuning

5. Splitting of Regions

- Regions will split if the size exceeds 1 GB and will stop all work on the regions and effect performance of the regions
- Increase the region size to large value

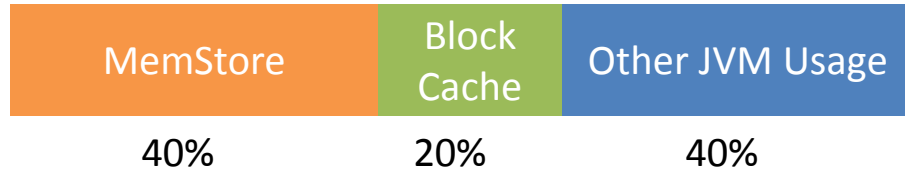
```
<property>  
<name>hbase.hregion.max.filesize</name>  
<value>107374182400</value>  
</property>
```
- Pre-splitting regions

6. VM Garbage Collection

- Avoid running Full GC, which can take a long time and start GCs early
- **`export HBASE_OPTS="$HBASE_OPTS -XX:CMSInitiatingOccupancyFraction=60"`** in **`$HBASE_HOME/conf/hbase-env.sh`** file
- Set **`-XX:ParallelGCThreads`** to higher number based on CPU power and JVM implementation

Performance Tuning

Default the memory allocation in region servers



7. Increase Memory Store size on Write Heavy Applications
 - ***hbase.regionserver.global.memstore.upperLimit*** to **.50**
 - ***hbase.regionserver.global.memstore.lowerLimit*** to **.40**
8. Increase BLOCK CACHE size on Read Heavy Applications
 - ***hfile.block.cache.size*** to **.30**