Design of the Secondary Index

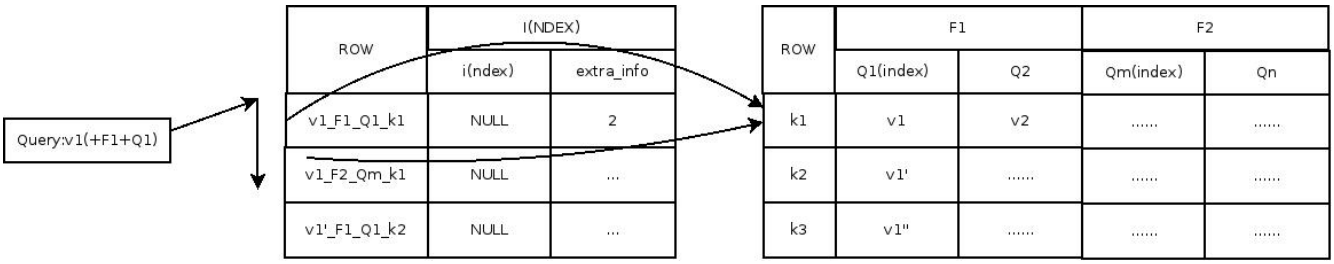
Background

Now HBase just support two may one read type : Scan(include Get). As I know, the goal of Hbase is "not a drop-in replacement for your RDBMS", but there are still some need for the secondary index. For example, I want to get out the search engine logs' record that the value is/contains this query (may be "iphone") or that query(may be "hadoop"), and so on. Now I could write a mapreduce job for "iphone", and another mapreduce job for "hadoop". Also I can write the coprocessor to get the values. Both the two methods should "scan" the whole "table". The SecondaryIndex can help us to find out the records immediately. According the Hypertable's secondary index's implementation, I think we also can implement one.

Design

1. We use the coprocessor to build up the index table. There is one things we just should do: prePut(Put), put the index to the index table. And then Put into the primary table.
2. We provide basic interface to manipulate the index table.(Java API, hbase shell)

SecondaryIndex structure

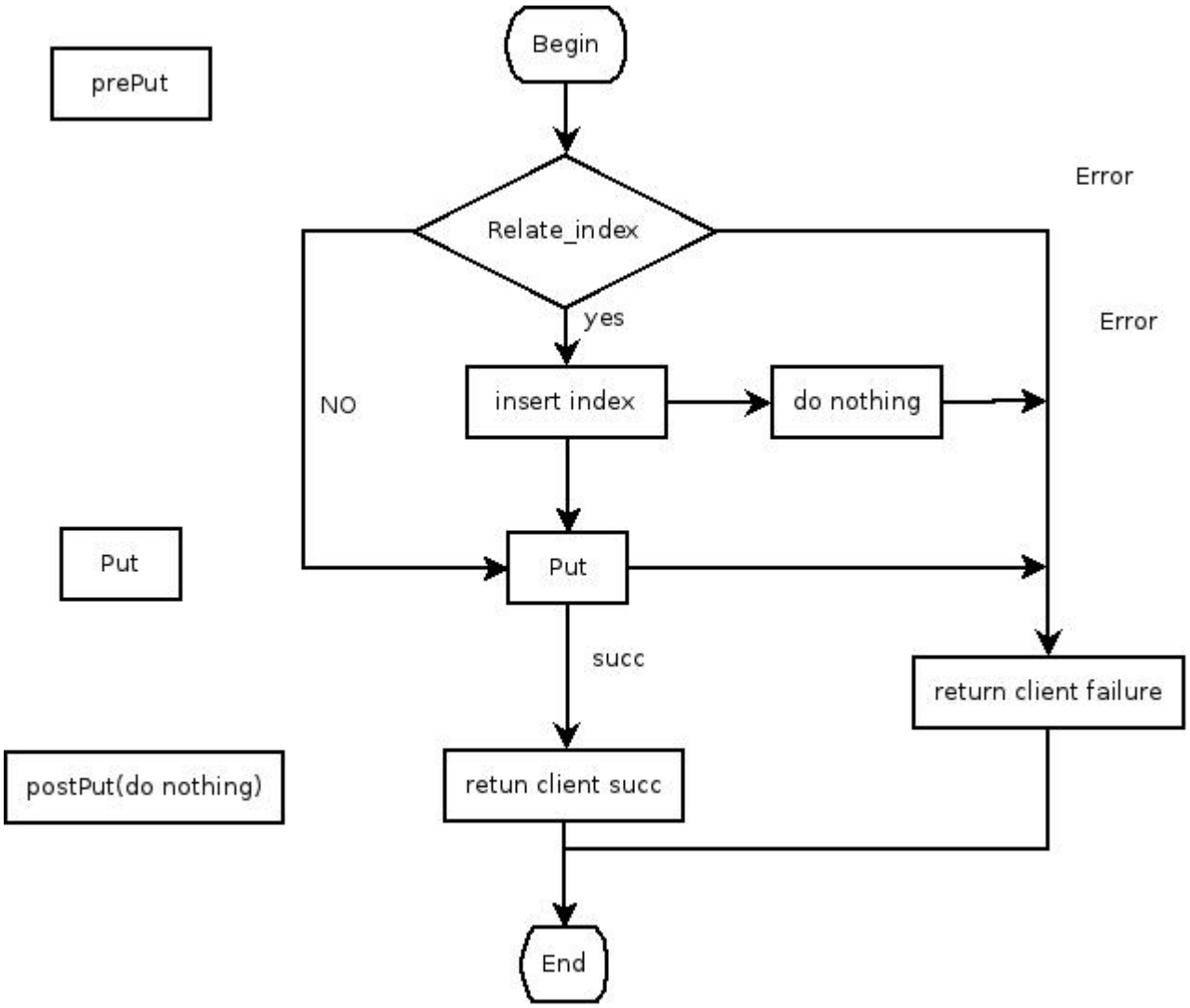


row : value_Family_Qualifier_Key(v_f_q_k)
Family : I(NDEX)
Qualifier : i(in deep, we can add more qualifier such as the static rank for the value to the key and more)
Value : empty
TimeStamp: same as the Put
Version : Integer.MAXVALUE(We save all version, for the sense that the prePut successfully, and primary table's put failed).

Sync or Async

According to the simple logic(simple for roll back), and the high write performance , we choose Sync for prePut, yes, just put('v_f_q_k').

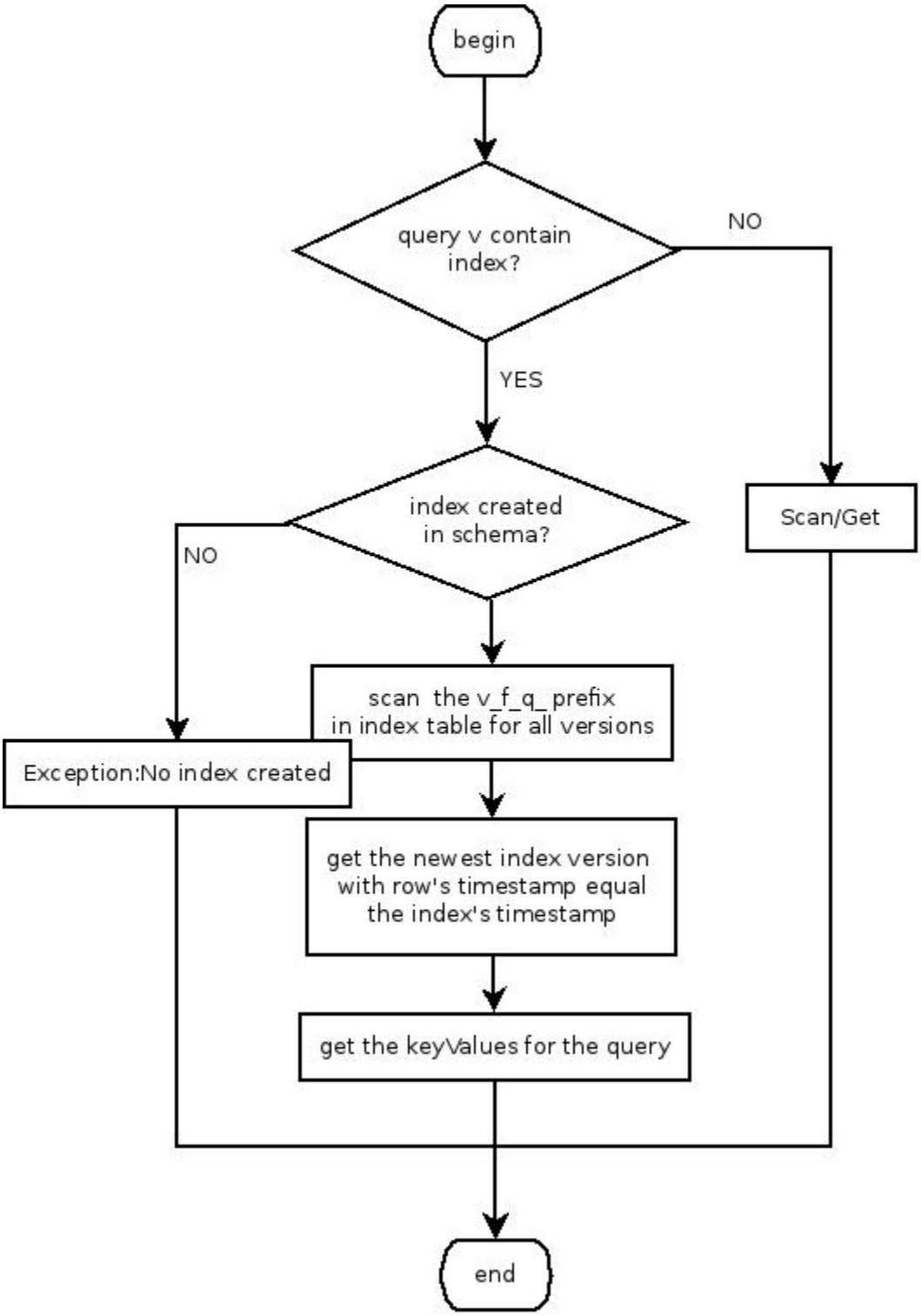
Put



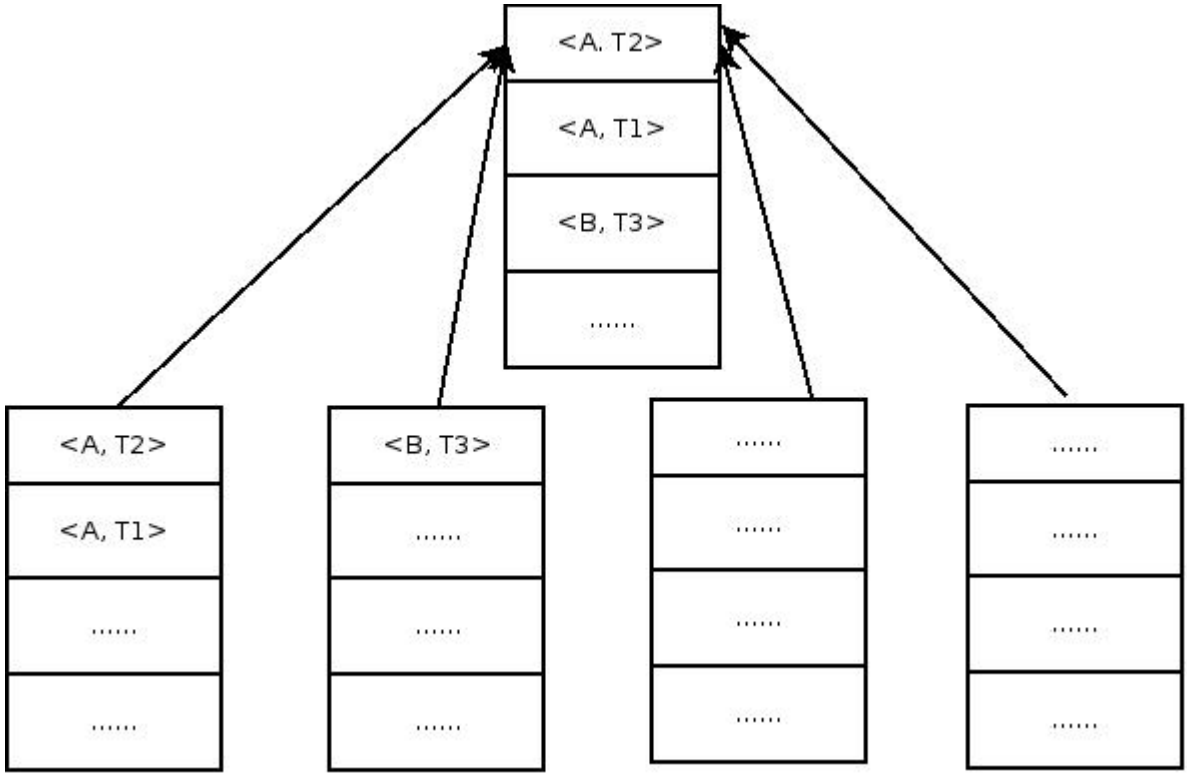
Delete

No index table op. The stale index data will be cleaned when the primary table does compaction.

Read(Scan/Get)



How is the Compaction works(like merge sort?)



Index table purge

1. We ignore those index that the prePut is successful and primary table's put failed, because the data is very small, the cost is just read two times for the index
2. During the primary table compaction, we also get out all the expired data, and do Delete for the expired data in the index table.

   There are two implementation for the delete,
   a) preCompact() for primary table, with compact() it will scan the Hfiles two times, may cause high I/O.
   b) Modified the current compactStore()'s code for index delete.
   Maybe the b) implementation is more effective.
3. The index table will also do compaction, and delete the Deletes.

Interface

1. Add index for hbase shell:
   hbase shell> alter 't1', METHOD => 'table_att', 'Coprocessor' =>

'hdfs://xxxx/xx.jar|org.apache.hadoop.hbase.SecondaryIndex| Priority| index.columns=" foo1:bar1,foo1:bar2,foo2:bar3"'
The underlying:
When all regions open, if it find that the _${TABLE_NAME}_INDEX_ does not exist, one region will create the table. May be this action should be called at client. Then an endpoint will also be called by client automatically. Before the endpoint run over, the read and write are opened to for client, because the index's timestamp is same as Put.

2. Disable/Enable, the index table will also disable and enable. Because after Disable, we may drop the primary table, then we will also drop the index table for resource.

3. JAVA Search API:
   HTable.Search(byte[] tablename, byte[] family, byte[] qualifier, byte[] value)
   In deep we can add word segment index, and the AND OR NOT op.

4. Add search rubby command
   hbase shell> search 't1', '{COLUMN=>'F1:Q1', VALUE=> 'v1' , LIMIT=>10}'

Difficulty and risk

1. During primary table compact, the StoreScanner just read family.getMaxVersions() versions' data, the old data will be skiped, but the old data's indices should be deleted, So we must read all version data for indices' delete, and family.getMaxVersions() version for store's compaction.

2. Maybe this is acceptable: some table has no compaction forever. These table is so small, so when the index table has expired data, the waste will also small.