**Multithreaded Concurrent Batch Ingestion**

Files are read line by line and QueuePopulator Threads add these lines to Queue.

In parallel, there are Queue Reader threads which poll this queue and pull these lines from this Queue and add this to native WriteBatch Object.

WriteBatch Objects are local to Queue Reader threads, this enables Concurrent Write Batch Objects creation and concurrent multi batch uploads leading to better concurrency.

Queue Populator Thread1 ---> <------- Queue Reader Thread1

Queue Populator Thread2 ---> **Central Blocking Queue** <------- Queue Reader Thread2

Queue Populator Thread3 ---> <------- Queue Reader Thread3

---N1 Threads ---- N2 threads (N2 is optimised on the

basis of number of cores)

QueueReader Threads each contain local copy of WriteBatch objects.

These threads poll Blocking queue and fill native WriteBatch object.

When Batch limit of BatchWrite Objects is reached, batch is written to levelDB.

As each QueueReader contains local copy of WriteBatch, this enable concurrent batches population and being written to levelDB.

When Native Write Batch of Thread is filled and written, new WriteBatch object is initialised for that thread.

Thread pools continue to work, till Queue is empty.

There is a notification signal placed in while loop which is true, as Queue can be empty in intermediate stage as well at time of Queue Population.

We wan't threads to keep running at this stage, as Queue will get more elements in future.

When Queue populators are done and we know, Queue can't have more new elements, this signal is changed to false.

If a Thread Encounters Empty Queue with false signal, its cycle is complete and it should exit.

Consequent UUIds/Email Map after batch ingestion of user data are written to a global Concurrent Array List shared between Queue Reader Threads which is returned by Batch Upload API.

For Lifecycle management of Threads, This Global Array List is returned after executor pool is terminated i.e all its tasks are complete.