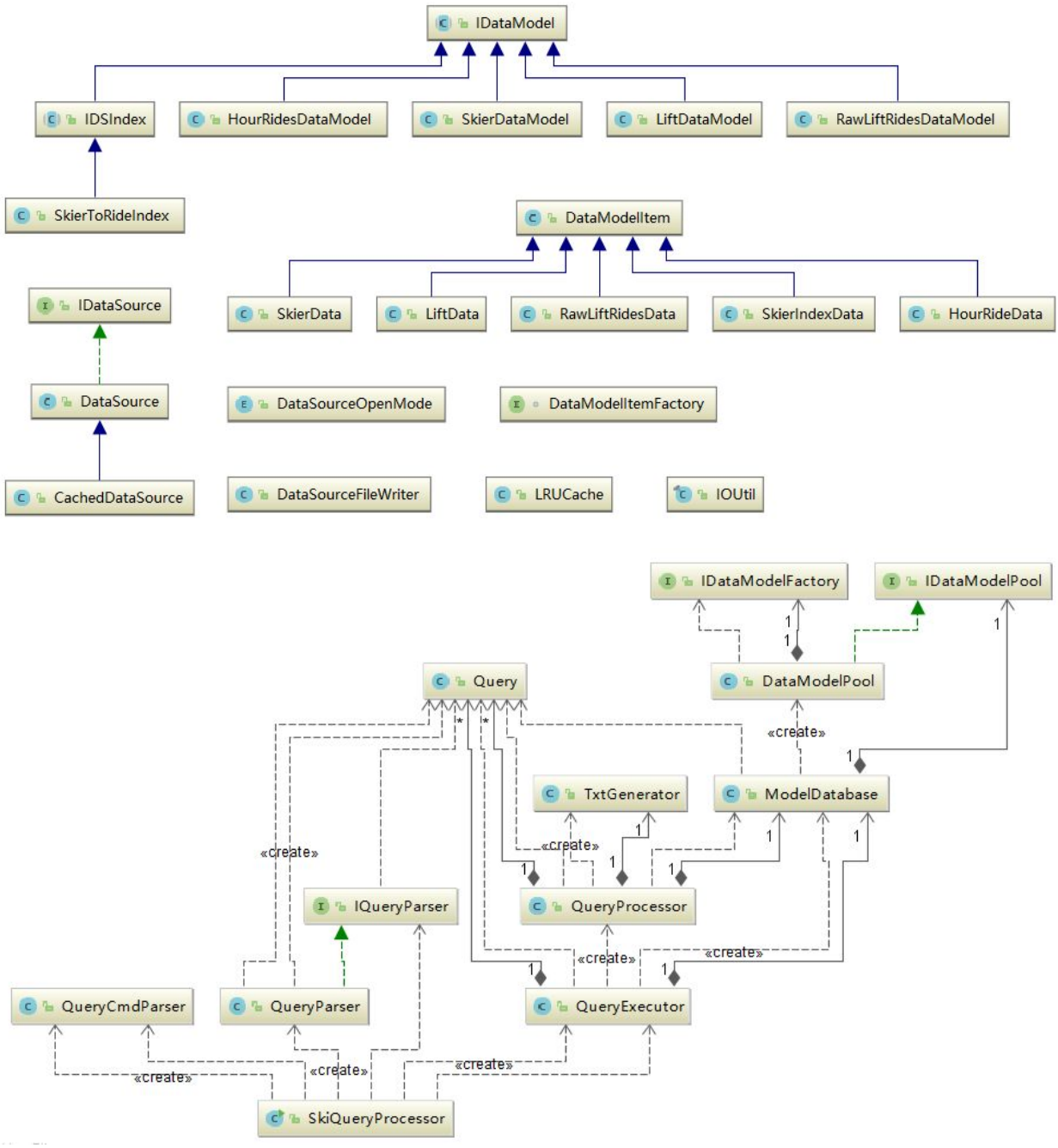
Creating Data Models :

We encoded our model as fixed width item row where row is just an abstraction to logical represent data but underlying data is just stream of bytes. All data model files are sorted by their key which is linear as that worked for all models in our assignment. Given a key we can seek to that particular location as our logical row width is same and then read bytes for that fixed width and convert it back to our logical row representation.

We changed our previous parallel data processor consumers to write dat files in such a way that either can be accessed in O(1) for all queries. For lift model we just represented each row as 2 fields (2 * 4(int size) bytes) to have liftId(key) and numRides in each row. Similarly for hour model we wrote 11 fields in row, 1 for hour(key) and 10 fields for liftIds according to their busy time. For skier we wrote 4 fields 1 for skier(key) and then numRides, totalVertical and numViews. Raw Lift model was little tricky as we it was not sorted and we didn't want to hold that much data in memory and sort it by skier which will still require binary search to get all skiers since it can be in more than 1 row. To address that we created another field which is rideId(key) and wrote the raw rides with sorted rideId. Then we created an index file which has skierId as key and all rideIds from rawLift dat file so we can later find all rideIds from index and then search in rawLift file. We found that skier has at max 4 rides per hour (24 per day), so we took higher threshold and assumed at max skier can do 100 rides per day and desgined index to have 1 col for skier(key), 1 (number of rides), 100 for rideIds by skier (relevant only upto number of rides). This index allow us to still answer skier details query in O(1).

All DataModels are abstracted and use generic Row Type which essentially represents column which user wants from a row in that model but underlying its just fixed width byte stream on file or int[] of fixed fields in application.
DataModel also provide functionality to do CRUD operations on these models given a key.
DataModel also contain a datasource which just need to know the number of fields from datamodel so it can create/access model accordingly. It just operates on int fields provided to datamodel and converts that into fixed 4 size bytes so data width of row is consistent.
Each DataModel has 2 modes to open it, one is create mode and one is access mode. First mode is create which is used when initially setting up model by previous assignment parallel data consumers as we want to write these models fast using buffered writers.
Second mode is access mode which perform read and updates on single key.



Accessing Data Models

DataModel have to be opened in access mode so queries can access them. It uses random access file to seek to particular position in file given a key to perform read and updates.

Also doing diskIO just for few bytes every time is not very IO efficient as disk is slower than memory and for some data files we are going to touch same row many times. So, we created a cached data source which uses LRUCache to cache some rows given id so we can just return

query from cache, cache is write through and gets updated when some row is updated along with updating data in file. This turns out to improve performance a lot.

Data Model Pool

One simple way to use data model is to create it separately for each thread which create lot of file handles in linux and extra overhead of syscalls to read/write which linux has schedule and causes lot of random seeks. And this is not even useful as we are already bounded by disk not by cpu so opening too many file handles in each thread won't increase performance rather degrade it for above reasons.

So, instead of creating data model in each thread we created a Thread safe Generic Data Model Pool which maintains a fixed size pool of data models and it provide 2 apis one to get data model from pool and one to return it back to pool once it has used it. If all models are used by threads, new thread requesting it will be blocked until some other thread return the model and notify waiting thread. Normally work done by thread should be very fast and with caching it should be much faster so most of the time threads won't block or block for small amount of time.

Shared Model Database With Lock Striping

This is shared database that is shared among all threads of queryExecutor. It maintains pool of models of each type and provides an api to perform query on it. For each query type it first try to get its model from the pool, perform query and return model back to pool, perform some operation on query result and returns it back. Also, it maintains a pool of locks which are only taken by query1 (skier summary query). Since query1 reads the row and update it back to database this is a critical section for same skierId as this is a read modify write operation. As we don't want to maintain 40k locks for each skier but still allow different skier to write without taking single lock all the time the pool of locks can be useful. For each skier we first get its hash and find which lock from pool to use, take that lock, get model, perform operation, return model and release the lock. This way we protect queries against same skierId to corrupt information by paying small penalty of rare case of skierId's hash colliding to same bucket in all threads at same time.

QueryParser

Just reads the input file given from input and process until we have numberOfQueries provided from input, return that list of queries after parsing where query is comprised of enum queryType and keyId.

QueryExecutor

QueryExecutor gets list of queries from parser which it divides equally among threads as all the queries are usually O(1) access so its easier to just partition among threads by number of

queries rather can cost per query and then partitioning it per cost. It also creates shared model database and provide it to all queryProcess threads so they can perform their subset of queries against it. QueryProcessor threads uses cyclic barrier to start performing queries together and wait for it to finish as well.