Create Indexes on Data Fields

After detailed discussion, our team decide to set indexes (including hashing index) on attributes of four frequently used tables. Due to the fact that users usually search some conditions using different combination of attributes, we decide to set indexes on each attribute separately. All these attributes are listed in the following:

1. player:
   1. player\_id: hashing index. The number of player\_id is quite huge and player\_id distributes quite sparsely which makes them unsuitable for bit-map. Also the user usually will query the player\_id in using the exact search where hashing index will perform much better than B-tree.
   2. season: bit-map index. There are totally 7 seasons in this database which is relatively small size. Therefore a bit-map comparing with hashing and b-tree will be much better since it not only searches data fast and occupy little storage.
   3. player\_name and player\_short\_name: B-tree index. First, players’ name distributed sparsely which makes it unsuitable for bit-map since it will use a lot of bits to represent different values. Usually, user will not only search the player’s name by exact search but also by fuzzy search like searching with prefix. In this scenario, B-tree will perform much better than other types of indexes because of its tree structure allows efficient search for prefix search and exact search.
   4. player\_short\_name: B-tree index (same reason as player.player\_name).
   5. club\_id: hashing index. There are tons of club resulting lots of different club ids, which will make bit-map structure not sufficient in storage. Due to the fact that club\_id is usually looked up by exact search, where hashing will perform much better than B-tree theoretically.
2. club:
   1. club\_id: hashing index (same reason as player.club\_id).
   2. season: bit-map index (same reason as player.season)
   3. club\_name: B-tree index (same reason as player.player\_name)
   4. league\_id: bit-map index. The total number of league\_id is relatively small which makes it possible to use bit-map with sufficient storage. Also, most of the search of league\_id is using exact search, where bit-map performs quite well.
3. league:
   1. league\_id: bit-map index (same reason as club.league\_id).
   2. season: bit-map index (same reason as player.season)
   3. league\_name: B-tree index. User might use fuzzy search like using prefix search where B-tree will perform much better than other two types.
   4. country\_region\_name: B-tree index (same as league.league\_name).
4. player\_positional\_rating:
   1. player\_id: hashing index (same reason as player.player\_id).
   2. season: bit-map (same reason as player.season).
   3. positional\_name: bit-map (same reason as player.season).
5. player\_best\_position:
   1. player\_id: hashing index (same reason as player.player\_id).
   2. season: bit-map (same reason as player.season).
   3. position\_name: bit-map (same reason as player.season).

However, the case is a lit bit different in the implementation. After checking the MySQL documents, we find the MySQL does not support bit-map indexing, which means we need to transform the type of bit-map index to hashing or B-tree.



Figure 1: line 19 indicating the index type[[1]](#footnote-1)

To support foreign key mechanism in our database, we need to use the InnoDB engine in our database construction since it is the only engine in MySQL providing this mechanism (which you can checked by typing “show engine” in MySQL).

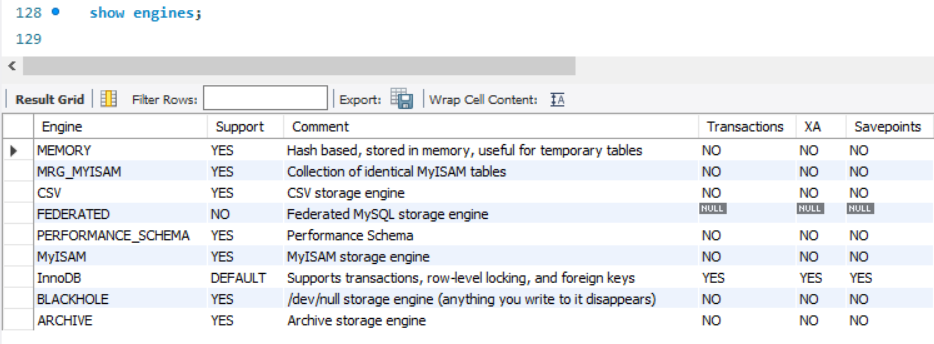


Figure 2: MySQL support engines

Whereas InnoDB does not support creating hashing index manually, instead it has a mechanism called Adaptive Hash Index, which will set the index to be hash like basing on its observation. It can be set up by turning on global variable innodb\_adaptive\_hash\_index.

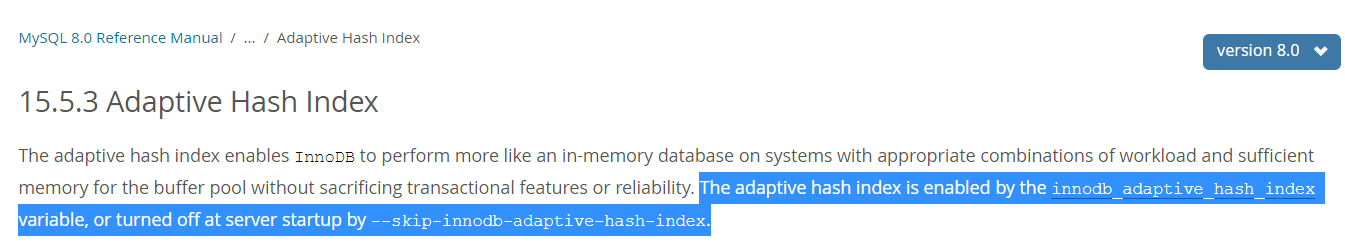


Figure 3: innodb adaptive hash index[[2]](#footnote-2)

Therefore, we need to turn set global variable innodb\_adaptive\_hash\_index = ON.

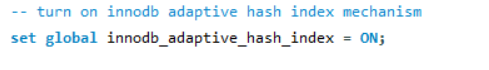


Figure 4: set innodb\_adaptive\_hash\_index = ON

Therefore we only need to set all the index to B-tree which is the default setting and InnoDB will change it to hash when necessary.

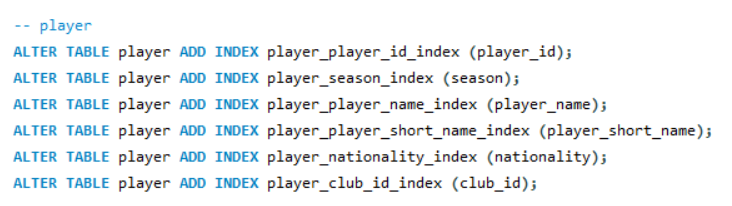


Figure 5: add indexes to table (partial)

After creating indexes, the performance of retrieving data using exact search and prefix fuzzy search will be improved a lot. The following is the test sample which use FLUSH operation and no-query-cache to avoid query cache affect the testing result. Notice that the query with index is deliberately be tested before the query after dropping index, which means if the memory still cache data, it will benefit the later query. With this setting, the later one without index still performs much worse than the previous one with index.

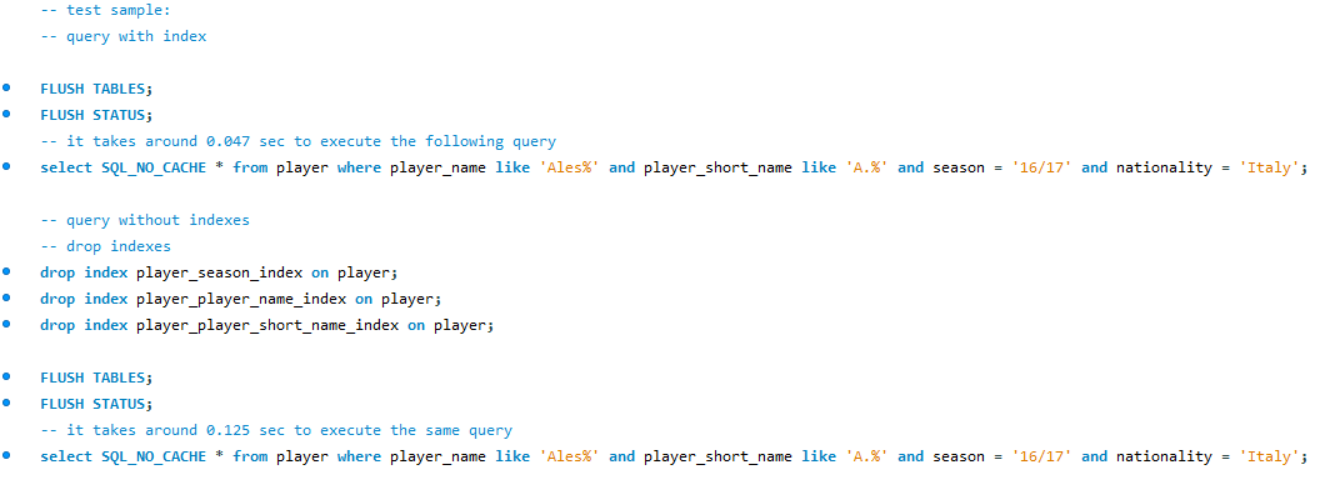


Figure 6: test sample for index

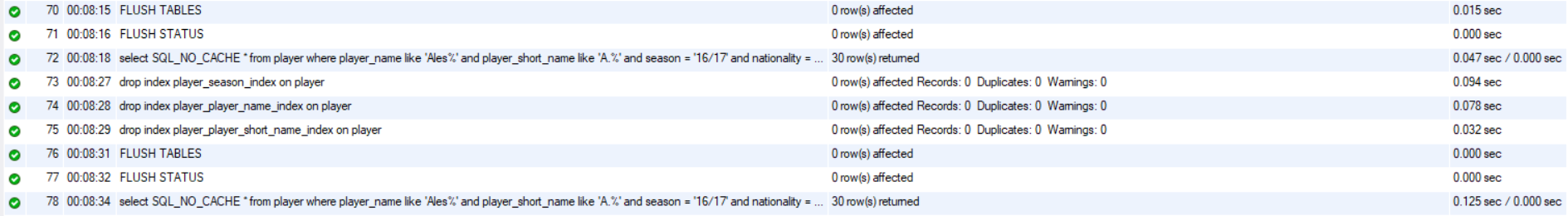


Figure 7: result of test sample

Notice in the above figure, query 73 (select with index) performs much better than query 78 (select without index). The sql codes in this section are written in /FIFADB/Codes/ step\_4\_create\_indexes.sql.

1. Retrieved from https://dev.mysql.com/doc/refman/8.0/en/create-index.html [↑](#footnote-ref-1)
2. https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html [↑](#footnote-ref-2)