Question 2

2a) A simple interface that could cause violations is a form where the user types id, zipcode, and state every time with no checks. If the user enters the same id + zipcode as before but in a different state, the FD (id, zipcode) means the state is broken.

2b) With timestamps added, id is no longer unique. So, a simple new primary key is: (id, timestamp). This way, each student can now have multiple versions over time without conflicts.

Question 3

a) SELECT * FROM item WHERE id = 1234;
This can use the hash index on id because it makes finding one specific item very fast.

b) SELECT * FROM item WHERE no_purchases > 0;
This cannot really use an index. Almost the whole table matches this condition, so an index doesn't save time.

c) SELECT * FROM item WHERE rank BETWEEN 1 AND 10;
This can use the clustered index on rank. Since the table is ordered by rank, those rows are already stored next to each other.

d) SELECT * FROM item WHERE rating = 4 AND price = 7;
This can use the (rating, price) index because it jumps straight to the few items that match that exact combination.

e) SELECT * FROM item WHERE rating = 4 AND price BETWEEN 7 AND 8;
This can also use the (rating, price) index. It allows the database to go right to that small price range under rating = 4.

f) SELECT * FROM item WHERE rating BETWEEN 4 AND 5 AND price BETWEEN 7 AND 8;
This can use the (rating, price) index too, because the result is still a tiny part of the table.

g) SELECT * FROM item WHERE rating = 3;
This can somewhat use the (rating, price) index. It helps locate the rating = 3 region, but since many items have rating 3, the benefit is smaller.

h) SELECT * FROM item WHERE price = 10;
This can use the (rating, price) index by checking each rating group for price = 10. It's still fast because very few items have that exact price.

i) SELECT * FROM item WHERE rank BETWEEN 1 AND 20 AND rating = 3;
This can use the clustered index on rank. The rank range is tiny, and filtering for rating = 3 afterward is quick.

Question 4

4b) It can. All bid prices are already stored in the bid_price index, so you can just scan the index to get the average.

4c) It can. The (make, model) index has one entry per car. Grouping by the keys in the index gives you the counts you need without touching the table.

4d) It can't. This query needs VIN, customer_no, and year. No index contains all of these together, so you have to read the actual tables.

4e) It can't. To answer this, you need both make and year. But one index has make, and the other has year, and neither has both. So you can't do it using indexes alone.