Homework 7 Answers

14.15 The table shown in Figure 14.19 lists dentist/patient appointment data. A patient is
given an appointment at a specific time and date with a dentist located at a particular
surgery. On each day of patient appointments, a dentist is allocated to a specific
surgery for that day.

(a) The table shown in Figure 14.19 is susceptible to update anomalies. Provide
examples of insertion, deletion and update anomalies.

Example insertion anomaly: The details about dentists (staffNo/dentistName)
and patients (patientNo/patientName) are repeated in the relation, so
each insertion must guarantee that these details are consistent.
Example insertion anomaly with null values: If we just want to insert a doctor
into the practice, then we have to insert null values for patient
information.
Example update anomaly: If we want to modify the name of a dentist or a
patient, we must ensure that we find and modify all occurrences of
those names in the relation.
Example deletion anomaly: If we delete all of a patient’s appointments, we lose
all information about the patient. If we delete all of a doctor’s
appointments, we lose all information about the doctor.
(b) Describe and illustrate the process of normalizing the table shown in Figure 14.19 to 3NF. State any assumptions you make about the data shown in this table.

The student should state any assumptions made about the data shown in the table. For example, we may assume that a patient is registered at only one surgery. Also, a patient may have more than one appointment on a given day.

1.

1NF

<table>
<thead>
<tr>
<th>staffNo</th>
<th>aDate</th>
<th>aTime</th>
<th>dentistName</th>
<th>patNo</th>
<th>patName</th>
<th>surgeryNo</th>
</tr>
</thead>
</table>

fd1 and fd4 violates 2NF

2NF

<table>
<thead>
<tr>
<th>staffNo</th>
<th>aDate</th>
<th>aTime</th>
<th>patNo</th>
<th>patName</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>staffNo</th>
<th>aDate</th>
<th>surgeryNo</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>staffNo</th>
<th>dentistName</th>
</tr>
</thead>
</table>

Fd3' violates 3NF

3NF

FK

<table>
<thead>
<tr>
<th>staffNo</th>
<th>aDate</th>
<th>aTime</th>
<th>patNo</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>staffNo</th>
<th>aDate</th>
<th>surgeryNo</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>patNo</th>
<th>patName</th>
</tr>
</thead>
</table>

FD3' violates 3NF
2. Look at the hotel database shown in the exercises at the end of Chapter 4 and think about the types of queries that might be made on such a database. Feel free to examine the queries over that database shown in the exercises at the end of Chapter 6. However, I don’t think many of the queries in those exercises would be frequently executed, and hence it would be hard to justify an index based on many of those queries. Conversely, I think there are many potential high-frequency queries that have been left out of those exercises (e.g., show me the rooms checking out today at the hotel I’m managing). Based on the queries you choose, and what was discussed in class about physical database design, describe what indices you would choose to create for each of the four relations and justify each index by indicating the queries that might make use of these indices.

Hotel

(hotelNo, hotelName, city)

hotelNo: It’s the primary key and you should typically index the primary key since it’s going to be involved in a huge number of joins.
hotelName: You can expect to get a large number of daily queries from potential guests about the availability of rooms at hotel X where X is the name of the hotel
city: Some potential guests might want to know which hotels in your chain are located in a certain city and management might wish to get daily reports about hotel bookings, grouped by city. Overall I would not expect there to be a huge number of queries requiring city as a key, and hence would be disinclined to use it as an index.

Room

(roomNo, hotelNo, type, price)

(roomNo, hotelNo): They’re the primary key and you should always index the primary key because of the join queries you expect to perform. Additionally, you can certainly expect a fair number of booking queries that are going to require that you retrieve the cost of the room, and that will require an access via the primary key.
type: By itself type is not so helpful, because while you can expect a great many queries that want to know about a type of a room, it’s usually going to be as part of a hotel query. Hence a potential index is (hotelNo, type) and so is (hotelNo, type, price), since customers might want to know about all family rooms at the Holiday Inn that are less than $40. However, there are not likely to be an unduly large number of rooms of any one type, so it’s probably ok to leave the index at (hotelNo, type) and just sort through the rooms that get returned.
price: By itself price is not so helpful, because a guest is normally wanting to find out about room prices at a specific hotel. Hence a potential index is (hotelNo, price).

Booking

(hotelNo, guestNo, dateFrom, dateTo, roomNo):

(hotelNo, guestNo, dateFrom): It’s the primary key, so there will be many joins on these attributes. Additionally, almost every check-in will involve
a query on these three attributes, so an index on these three attributes seems mandatory.

hotelNo: I do not foresee many queries that need to return just bookings by hotelNos, but there are going to be a lot of queries involving hotelNos (e.g., show me all available rooms at H003 for Monday night), so probably there should be an index for hotelNo. Most queries will have related attributes in the query, such as “give me all bookings at H003 that are checking in today” or “check the availability of room 425 at the Holiday Inn”, so it might be good to have hotelNo included in multi-attribute indices. See below discussion for the most appropriate ones.

guestNo: It is likely that you will get a fairly heavy stream of guests wanting information about their bookings, and hence you will have many queries that retrieve booking information for a guest. An index for guestNo seems to be in order.

dateFrom: Every hotel is going to want to prepare for check-ins each day so you can expect many queries on the dateFrom/hotelNo fields. A good index is probably (dateFrom, hotelNo). You might have to experiment whether dateFrom or hotelNo is the primary key in this index. There is also going to be a heavy number of queries that simply ask about rooms that are currently occupied, and will be of the form (currentDate >= dateFrom) so a single index on dateFrom seems important.

dateTo: Same consideration is dateFrom—every hotel wants to know about its daily checkouts, so (dateTo, hotelNo) is a good index and a single index on dateTo seems important, given that occupancy check queries will have the predicate (currentDate <= dateTo).

roomNo: Many daily queries of the form “check the availability of doubles at the Holiday Inn on Oct. 14” are likely to occur and these queries will translate “doubles” to a (hotelNo, roomNo) pair, so probably an index on (hotelNo, roomNo) is in order. It’s unlikely that a roomNo will be presented in isolation, since it does not have much meaning without an attached hotelNo, so I doubt there would be an index on roomNo alone.

Guest (guestNo, guestName, guestAddress)

guestNo: guestNo is the primary key so you would expect to have a great many joins involving guestNo. Thus there should be an index on guestNo.

guestName: There needs to be an index on guestName because many of the queries on the guest database are likely to be via guestName, as most guests are not going to remember their guest numbers.

guestAddress: It’s hard to see many daily queries that need to retrieve a guest’s address. For marketing reasons you might want to occasionally run queries that group guests by city or state so that you can get information about where your guests live, but you probably won’t run these queries often enough to warrant maintaining an index.