19.2) 1. CDE, ACD, BCD

2. R is in 3NF because B, E, and A are all parts of keys.

3. R is not in BCNF because none of A, BC, and ED contain a key.

19.3) 1. Functional dependencies over R: Z🡪Y , X🡪Y, XZ🡪Y

2. Same as Part 1. The functional dependency set is unchanged.

19.5) 1a) Strongest Normal Form: 1NF

1b) BCNF Decomposition: AB, CD, ACE

2a) Strongest Normal Form: 1NF

2b) BCNF Decomposition: AB, BF

3a) Strongest Normal Form: BCNF

4a) Strongest Normal Form: BCNF

5a) Strongest Normal Form: BCNF

19.6) 1) BC 🡪 A does not hold over S

2) No, given just an instance of S, we cannot identify any dependencies. We can only state that certain dependencies are not violated by this instance, but cannot make an assertion that it holds with respect to S.

19.7) 1a) B

1b) R is 2NF but not 3NF

1c) C🡪D and C🡪A are both violations of BCNF. Can decompose R into AC, BC, and CD.

2a) BD

2b) R is in 1NF but not 2NF

2c) Both B🡪C and D🡪A cause violations to BCNF. Decompose into AD, BC, and BD.

3a) ABC and BCD

3b) R is in 3NF but not BCNF

3c) D🡪 A and A is not a key so ABCD is not BCNF. There is no BCNF decomposition because decomposing R into AD, BCD does not preserve the dependency ABC🡪D.

4a) A

4b) R is in 2NF but not 3NF

4c) BC🡪D violates BCNF since BC does not contain a key. Decompose R into BCD, ABC.

5a) AB, BC, CD, AD

5b) R is in 3NF but not BCNF

5c) C🡪A and D🡪B cause violations. There is no BCNF decomposition.

19.10 1a) BD

1b) Unsatisfactory because the join of BC and AD is the Cartesian product of which could be much bigger than ABCD.

2a) AB, BC

2b) The decomposition is lossless, but the dependency AB🡪C is not preserved so ABC needs to be added.

3a) A, C

3b) It is already in BCNF because A and C are candidate keys so there is no point in decomposing R.

4a) A

4b) This is a lossless join decomposition but it is not dependency preserving because B🡪C is not preserved.

5a) A

5b) This is not the best decomposition. It is a lossless join, but isn’t dependency preserving.

16.2 1) If transaction T2 performed W(Y) before T1 performed R(Y) and then T2 aborted, then the value read by transaction T1 would be invalid and it would abort upon cascade.

2) Strict 2PL would require T2 to obtain an exclusive lock on Y before writing to it. This lock would have to be held until T2 committed or aborted; this would block T1 from reading Y until T2 was finished, but there would be no interference.

3) One reason is that it ensures only ‘safe’ interleaving of transactions so that transactions are recoverable, avoid cascading aborts, etc. Another reason is that strict 2PL is very simple and easy to implement. The lock manager only needs to provide a lookup for exclusive locks and an atomic locking mechanism (such as with a semaphore).

16.3) 1) T2:R(X), T2:R(Y), T2:W(X), T1:R(X) where T1:R(X) is a dirty read

2) T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X) where T2 will get an unrepeatable read on X.

3) T2:R(X), T2:R(Y), T1:R(X), T1:R(Y), T1:W(X), T2:W(X) where T2 has overwritten uncommitted data.

4) Strict 2PL resolves these conflicts as follows:

(1) In S2PL, T1 could not get a shared lock on X because T2 would be holding an exclusive lock on X. Thus, T1 would have to wait until T2 was finished.

(2) Here T1 could not get an exclusive lock on X because T2 would already be holding a shared or exclusive lock on X.

(3) Same as above.