Transactions

This lecture assumes that you have watched videos (50 min) and answered questions from DB10 Indexes and Transactions of Jennifer Widom's (see link below)

Introduction to Transactions (13:43)
Transactions Properties (2:50 + 5:45 + 4:24)
Isolation Levels (7:47 + 1:44 + 4:55 + 3:15)

and done the Transactions Quiz

and/or read Chapter 7 of Ullman and Widom Intro to DB textbook

https://class.stanford.edu/courses/DB/Indexes/SelfPaced/courseware/ch-indexes/

Overview of Transaction Management

A unit of work called a *transaction* is a package of operations, which *from the standpoint of the user* are:

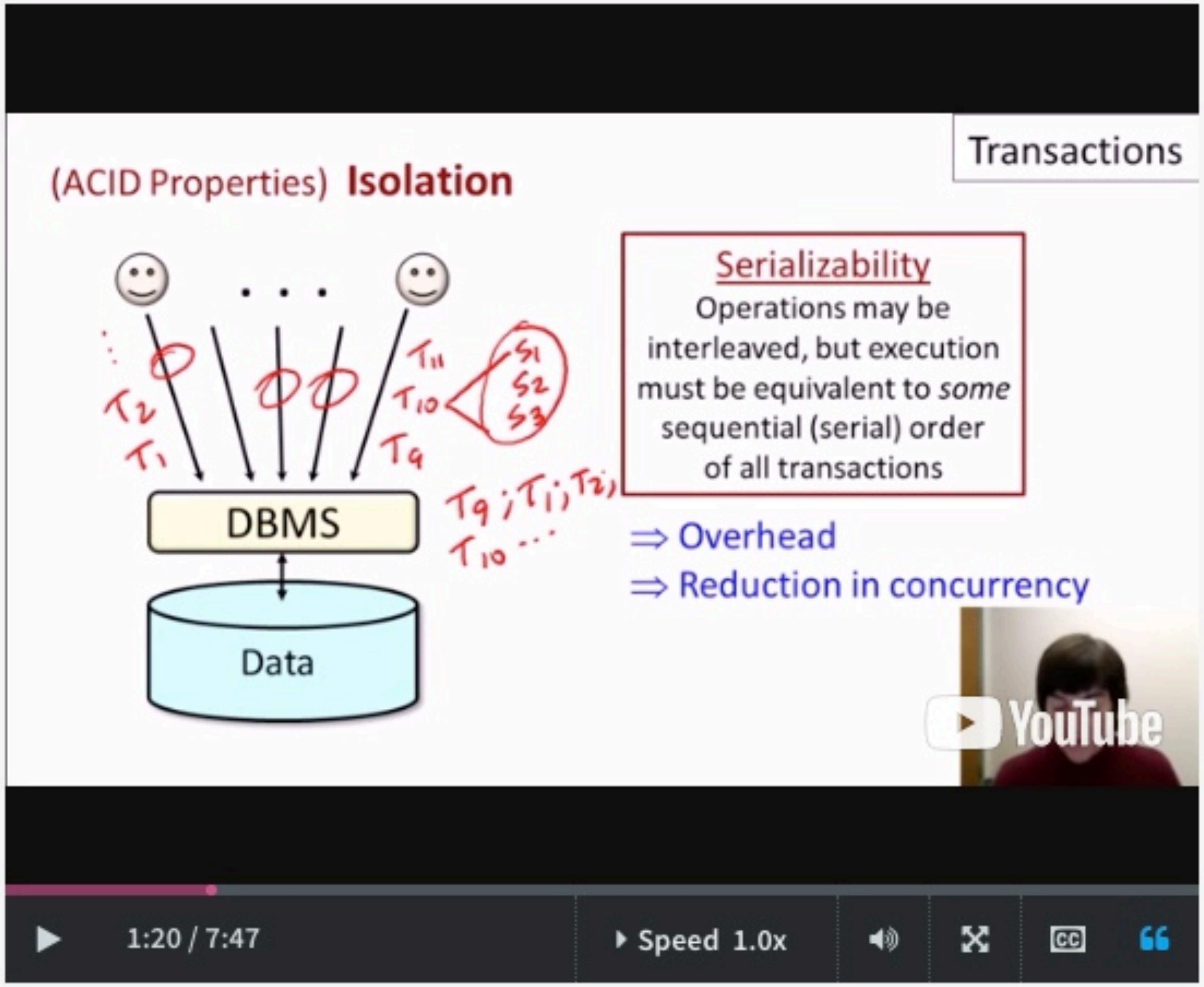
Atomic – all or no operations of a transaction are carried out (definitional), and this may necessitate "undoing" intermediate steps in the event of a crash or an abort

Consistent – when run on a consistent DB, a transaction should leave the DB without constraint violations (responsibility of user) and integrity constraints

Isolated – a transaction can be understood and executed independent of any other transaction (definitional)

Durable – when a transaction is reported as complete, its effects should persist (even in the event of a crash)

Video Part 1



Transactions

(ACID Properties) Isolation

Serializability

Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

⇒ Overhead

⇒ Reduction in concurrency

1:20 / 7:47 Speed 1.0x

first.

Then maybe T1, T2, and then T10,

and so on.

Serializability give us understandable behavior

and consistency but it does

have some overhead involved in

the locking protocols that are used

and it does reduce concurrency.

As a result of the overhead and

reduced concurrency, systems do

offer weaker isolation levels.

In the SQL standard, there are

three levels: read uncommitted, read

committed, and repeatable read. And

these isolation levels have lower

overhead and allow higher concurrency

https://lagunita.stanford.edu/courses/DB/Indexes/SelfPaced/courseware/ch-indexes/seq-vid-isolation_levels/

I/O and CPU activities can be and are overlapped to minimize (disk and processor) idle time and to maximize throughput (units of "work" per time unit). This motivates concurrent, interleaved execution of transactions.

Consider the following two transactions, T1 and T2:

T1: Read(A), Op_{11}(A), Write(A), Read(B), Op_{12}(B), Write(B), Commit

T2: Read(A), Op_{21}(A), Write(A), Read(B), Op_{22}(B), Write(B), Commit

Three interleaved *schedules* are (just showing disk reads and writes):

S1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

S2

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

S3

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

S1

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

S2

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

S3

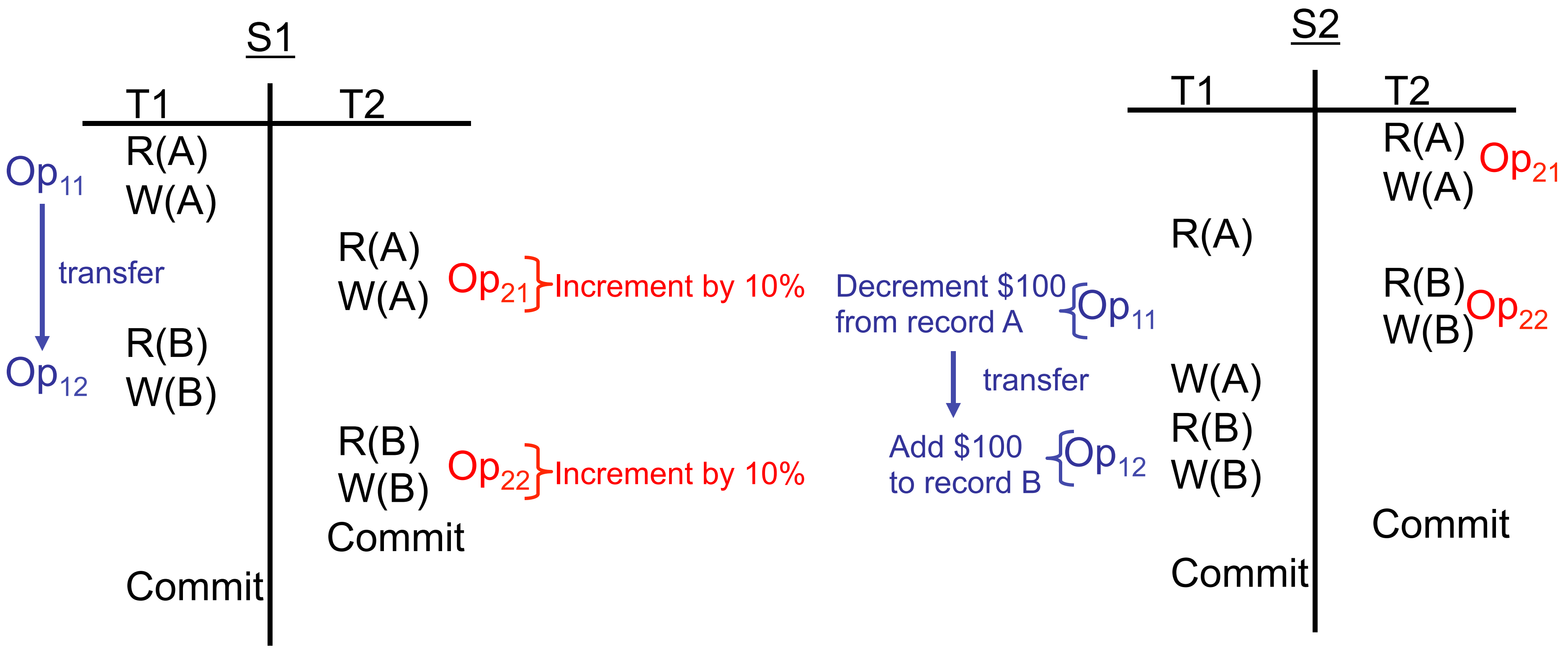
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

S1 is '*serializable*': it yields the same result as T1 run to completion, followed by T2 run to completion, or T1 ➔ T2 (under assumption of no failures/rollbacks)

S2 is 'serializable': T2 ➔ T1 (under assumption of no failures/rollbacks)

S3 is *not* serializable: it *may* yield different results than either T1➔T2 or T2➔ T1

Each of these schedules has examples of a *dirty read,* which can sometimes lead to anomolies (and which is why I write 'serializable' in quotes).



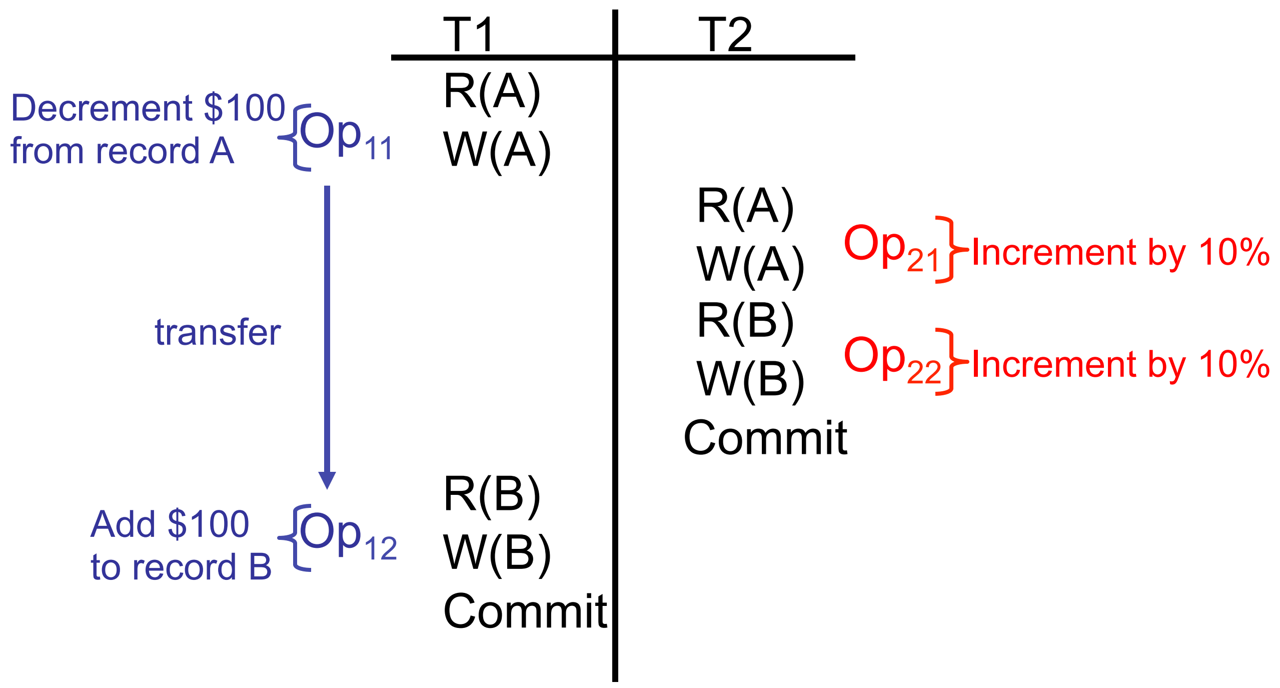
S1 is 'serializable':
T1 ➔ T2

S2 is 'serializable':
T2 ➔ T1

Different serializations, T1➔T2 and T2➔T1, need not lead to the same DB instances.

Example above: incrementing accounts by 10% after transfer (S1) versus before transfer (S2)

S3



S3 is *not* serializable: it *may* yield different results than either T1➔T2 or T2➔ T1

10% increments are made on both tables at lowest value. In general, this can be a problem (sometimes) with dirty reads (when one transaction reads data that has been changed by another transaction prior to that other transaction commiting).

Strict two-phase locking (2PL)

1. If a transaction wants to read an "object", it first obtains a *shared* lock on the "object"

2. If a transaction wants to modify/write an "object", it first obtains an exclusive lock on the object

3. All locks held by a transaction are released when the transaction is complete (upon Commit)

A shared lock on an object can be obtained in the absence of an exclusive lock on the object *by another transaction*.

An exclusive lock can be obtained in the absence of any lock by another transaction

Basically, locking is concerned with ensuring atomic and isolation properties of individual transactions, while exploiting parallelism/interleaving.

What “objects” can be locked?

Entire tables

Individual records within a table

A set of records that satisfy a condition (e.g., TransNumber = abc)

An entire indexing structure on an attribute for a table

Individual nodes (index pages) within the indexing structure

Individual data pages

In general, we want exclusive locks on the smallest “objects” possible?

Can individual attribute fields of an individual record be locked?
Check it out….

Which of these are legal schedules (that interleave transactions T1 and T2) under 2PL?

These schedules only show the locks (X: exclusive lock; S: shared lock), reads (R) and writes (W). Between a read and write of an object (resource), there would also be an operation on the object (e.g., update a tuple, an index, etc), which I don't show.

Remember, in 2PL, locks are only released upon commit, but can be changed
(Shared -> Exclusive) during transaction

S4

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

S5

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
X(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

S6

T1	T2
X(A)	
R(A)	
W(A)	
	X(A)
	R(A)
	W(A)
X(B)	
R(B)	
W(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Commit

S4

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

Lock released so doesn't block

Lock released so doesn't block

OK

S5

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
~~X(A)~~	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

deadlock

Illegal under Strict 2PL

NO

S6

T1	T2
X(A)	
R(A)	
W(A)	
	~~X(A)~~
	R(A)
	W(A)
X(B)	
R(B)	
W(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Commit

NO

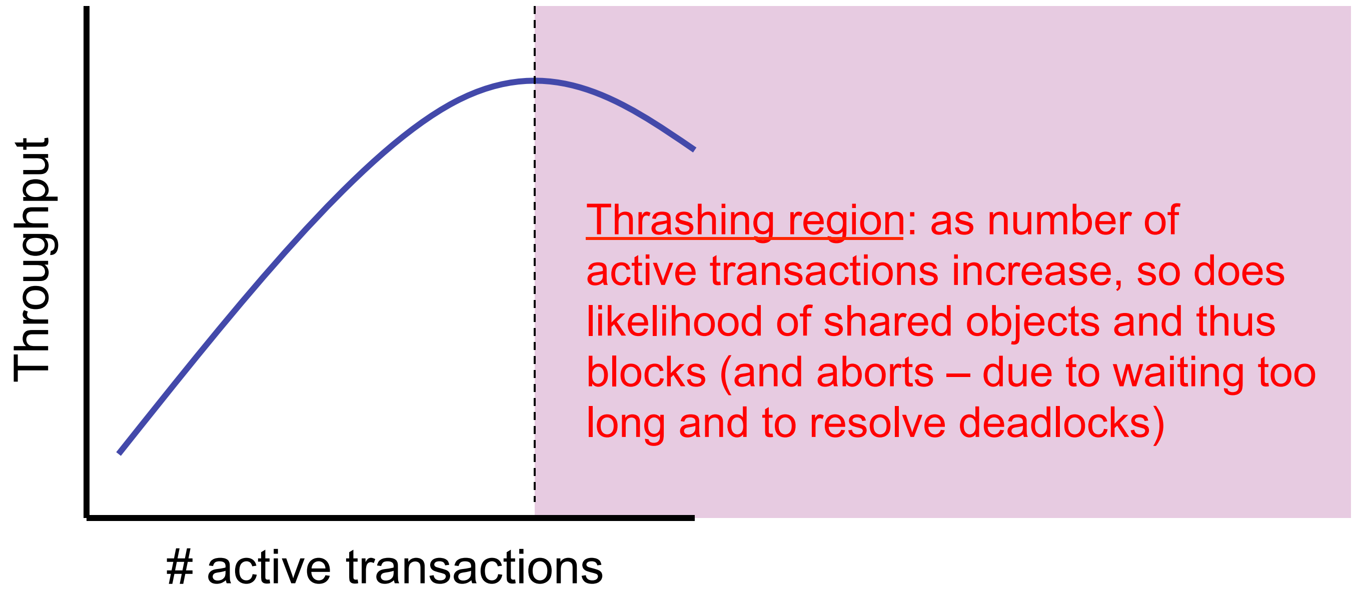
Can you do any interleaving of T1 and T2 under strict 2PL at all?

In cases where transactions involve the same objects, Strict 2PL can radically limit opportunities for parallelism/interleaving

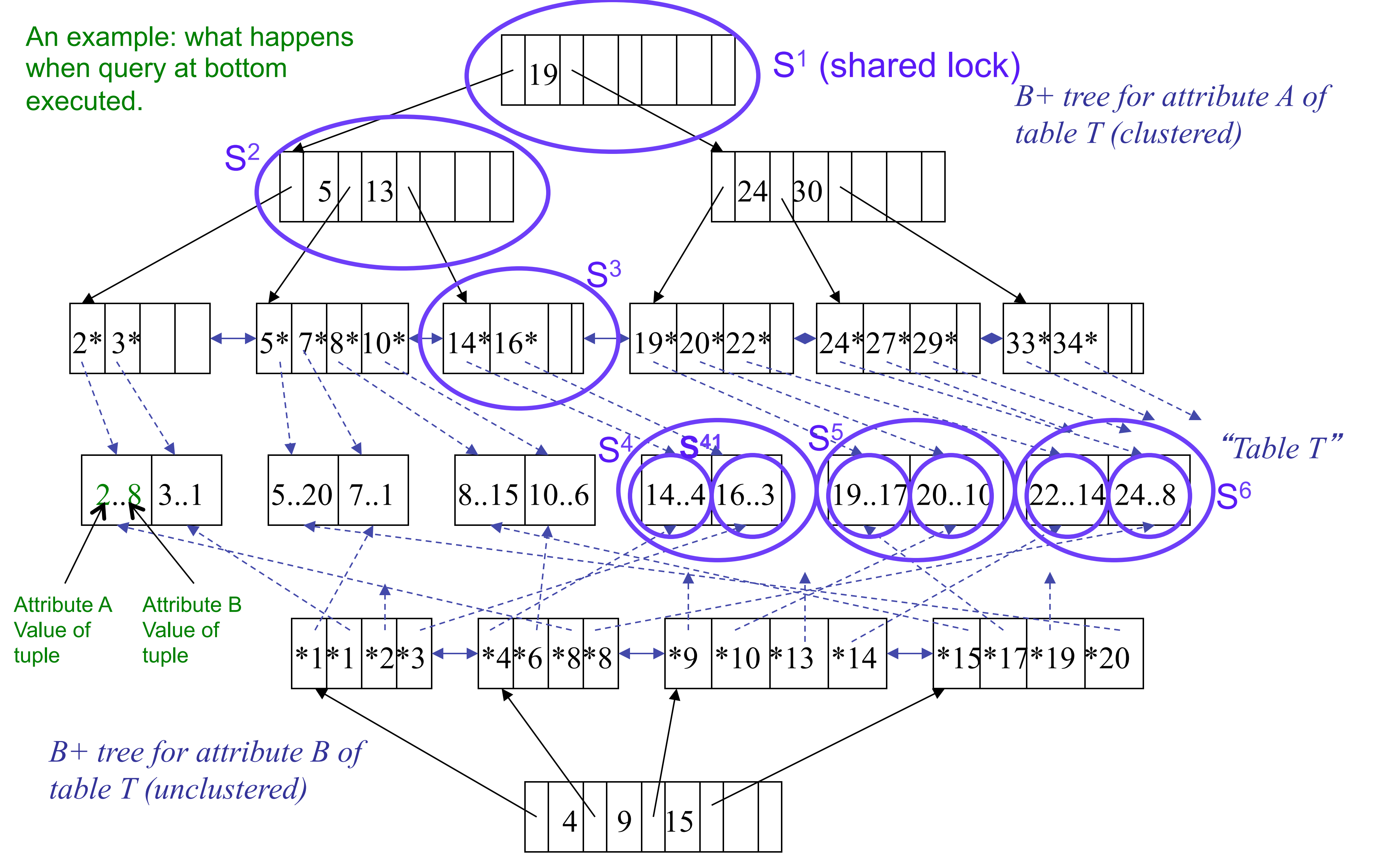
…. But Strict 2PL makes interleaving safe, and the good news is that

in practice, there are many transactions that do not involve the same objects and that can be interleaved to improve throughput

and even transactions that share objects (through reads) can be interleaved with strict 2PL (and shared locks)



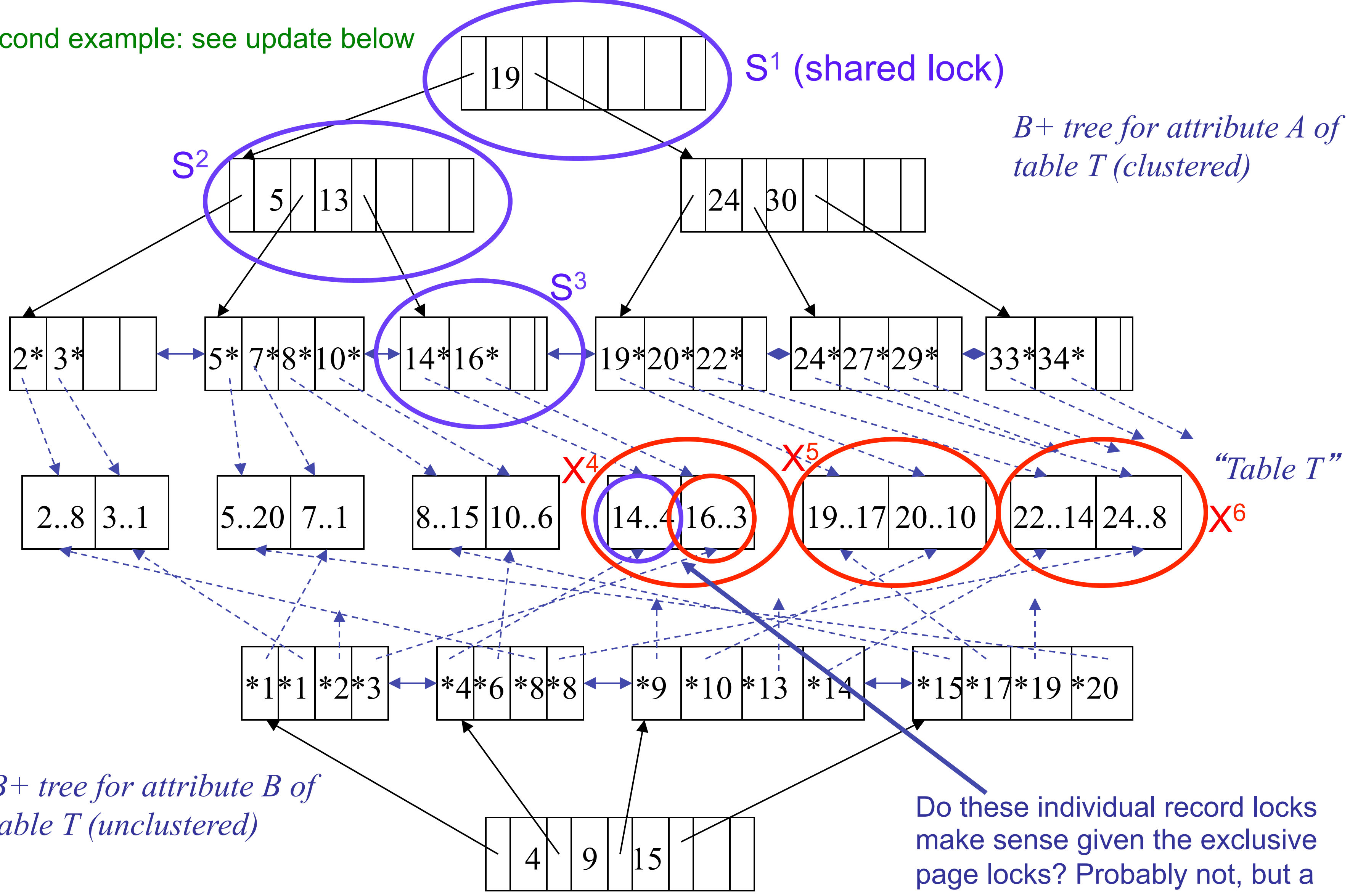
An example: what happens when query at bottom executed.



SELECT T.C FROM T WHERE **T.A > 14** AND T.B <= 10

What locks and in what order?

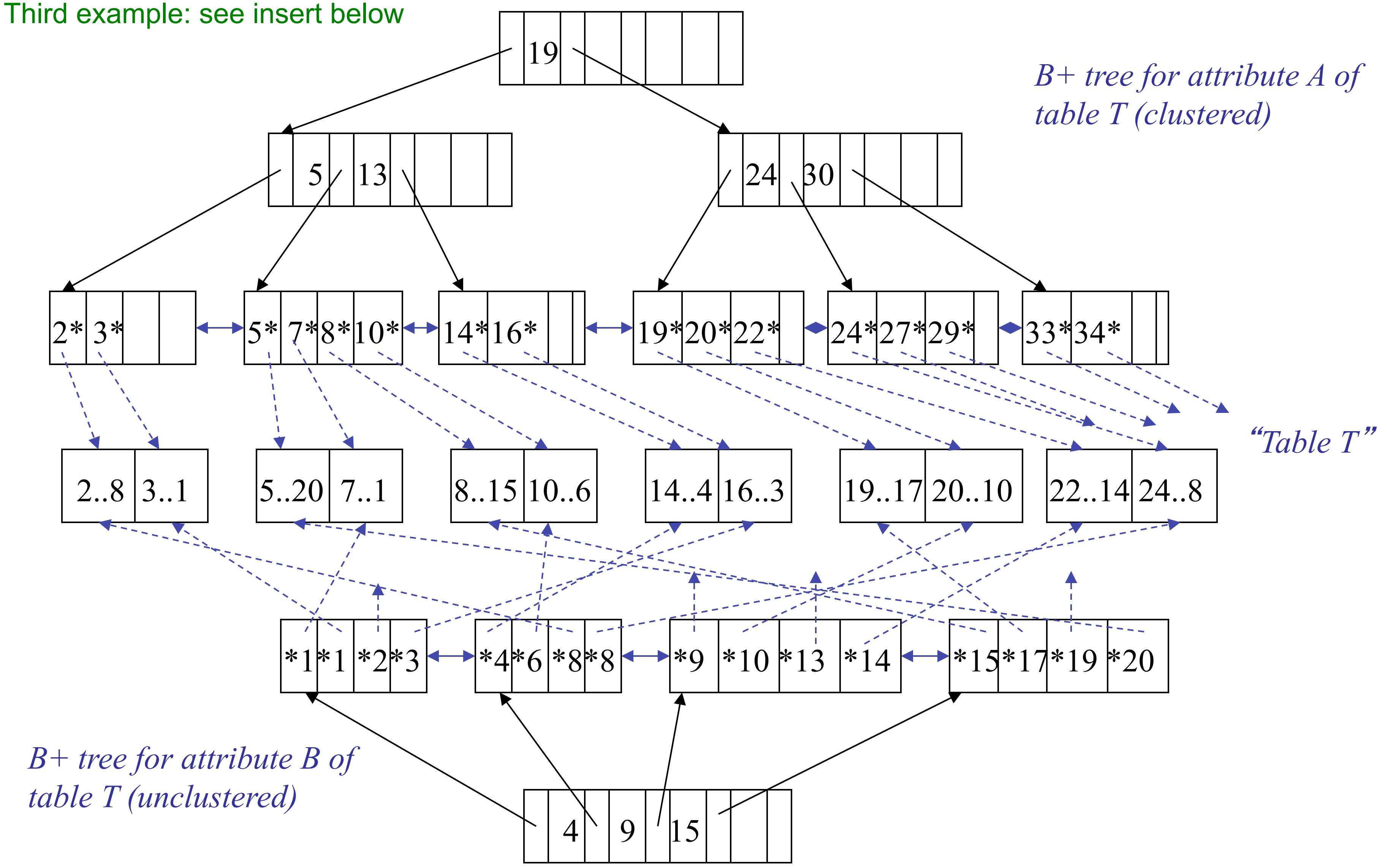
Second example: see update below



Do these individual record locks make sense given the exclusive page locks? Probably not, but a shared lock of the node, and an exclusive lock of a tuple would make sense. The granularity of objects that can be locked will vary with SQL platform

UPDATE T SET T.C = T.C+1 WHERE **T.A > 14** AND T.B <= 10

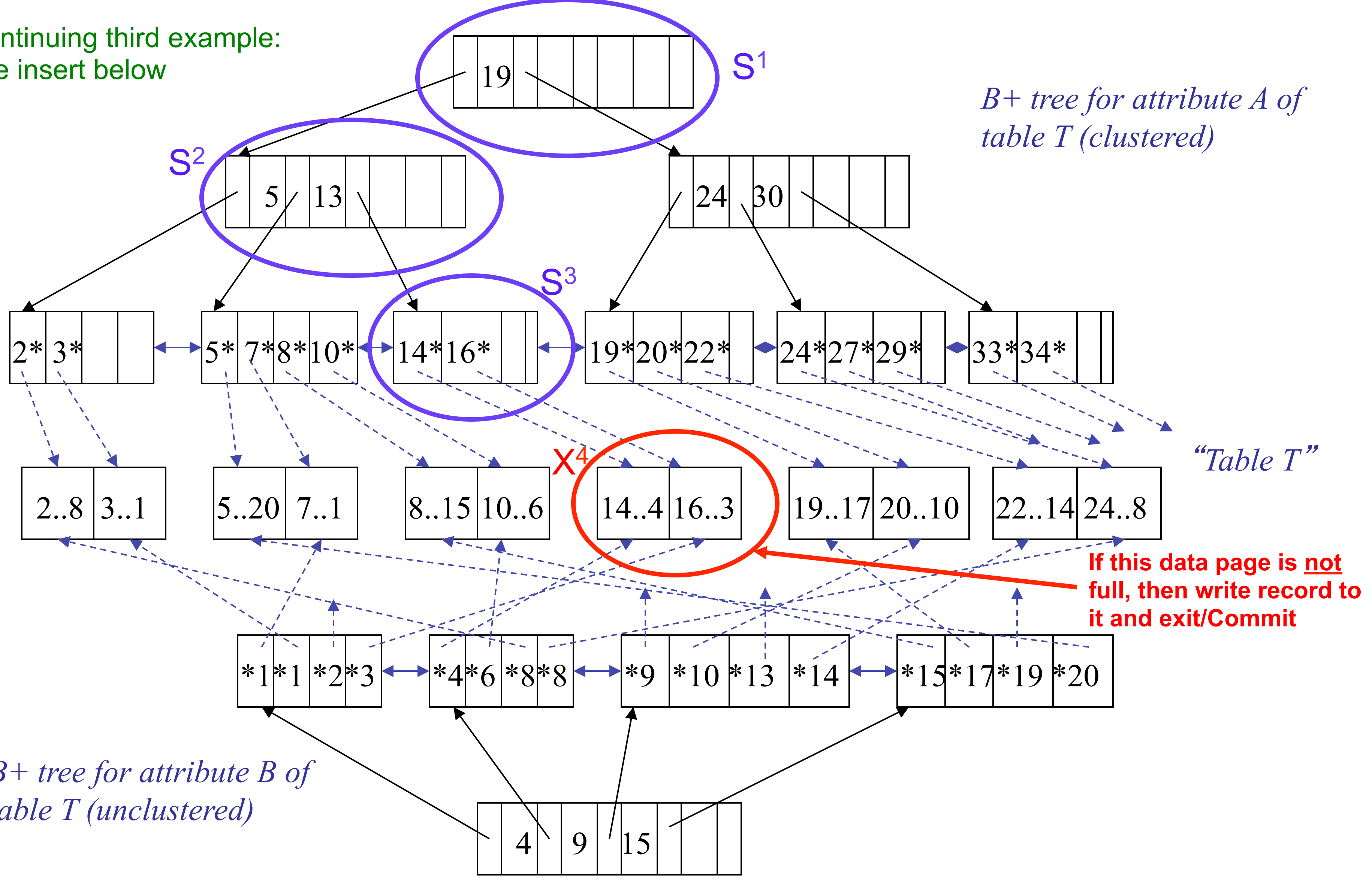
Third example: see insert below



INSERT INTO T (A, B, C) VALUES (**18**, 12, …)

What locks and in what order?

Continuing third example:
see insert below

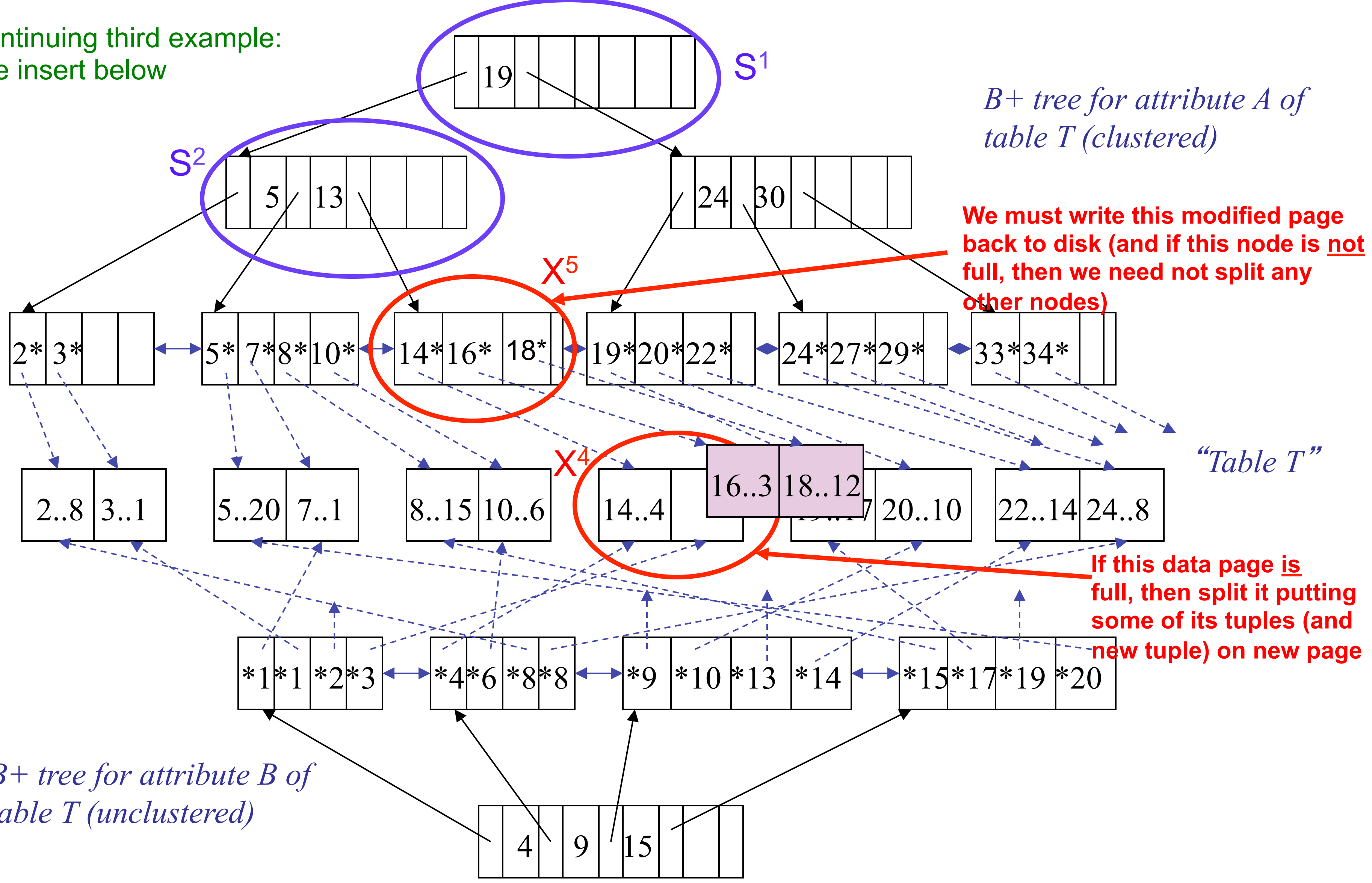


If this data page is not full, then write record to it and exit/Commit

INSERT INTO T (A, B, C) VALUES (**18**, 12, …)

Continued on next page

Continuing third example:
see insert below



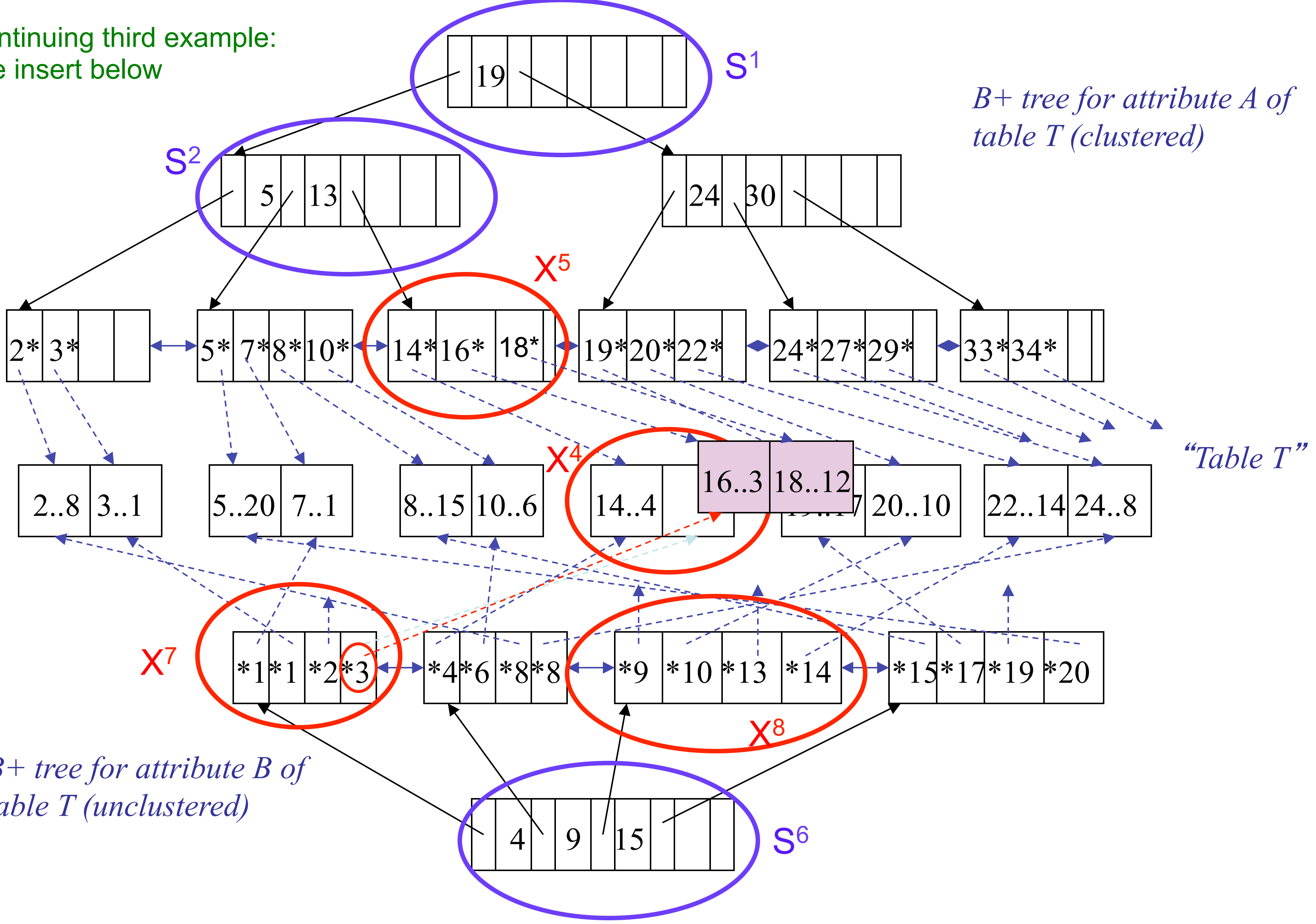
We must write this modified page back to disk (and if this node is <u>not</u> full, then we need not split any other nodes)

If this data page <u>is</u> full, then split it putting some of its tuples (and new tuple) on new page

INSERT INTO T (A, B, C) VALUES (**18**, 12, …)

Continued on next page: must update B index too

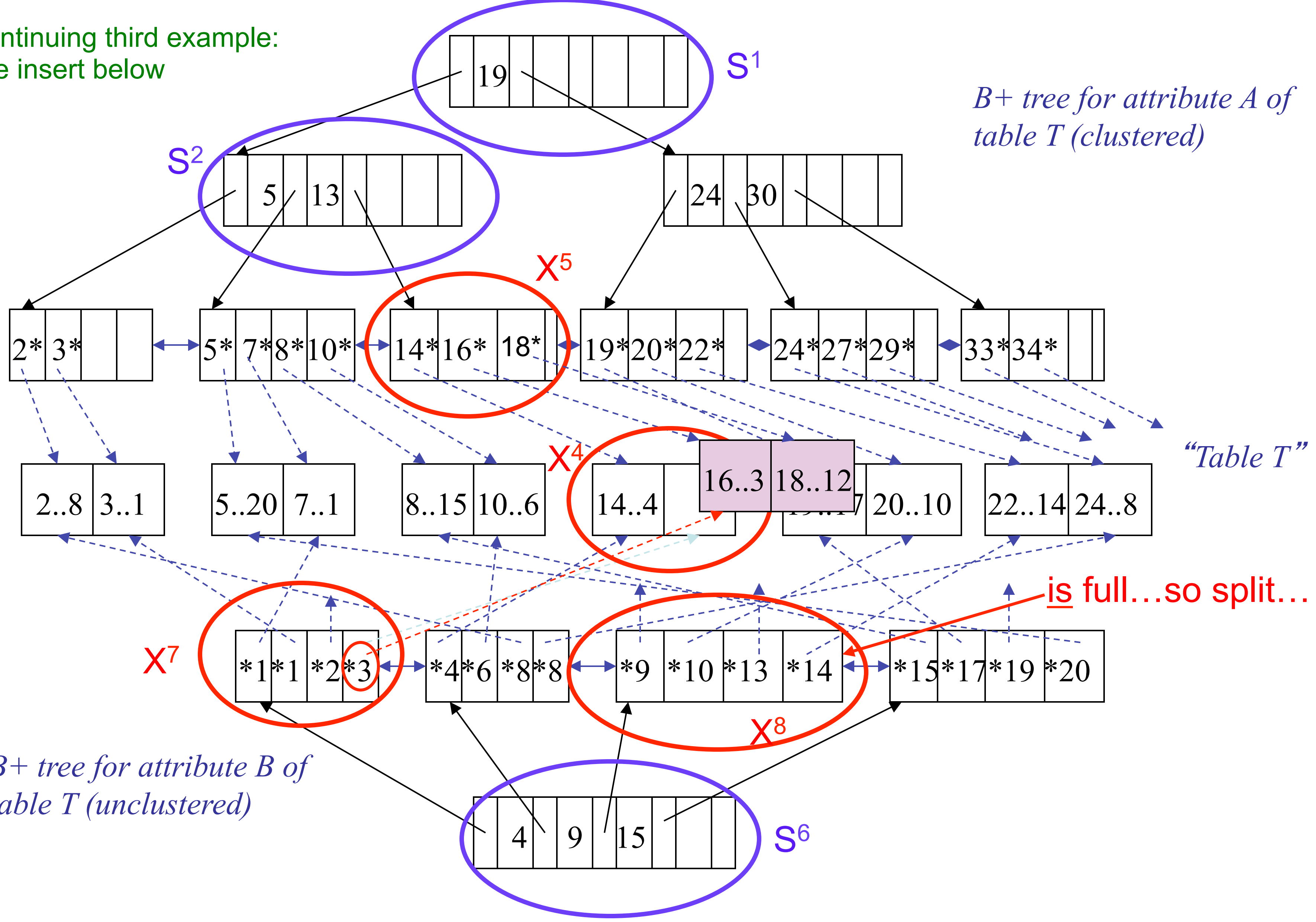
Continuing third example:
see insert below



INSERT INTO T (A, B, C) VALUES (**18**, 12, …)

Continued on next page:

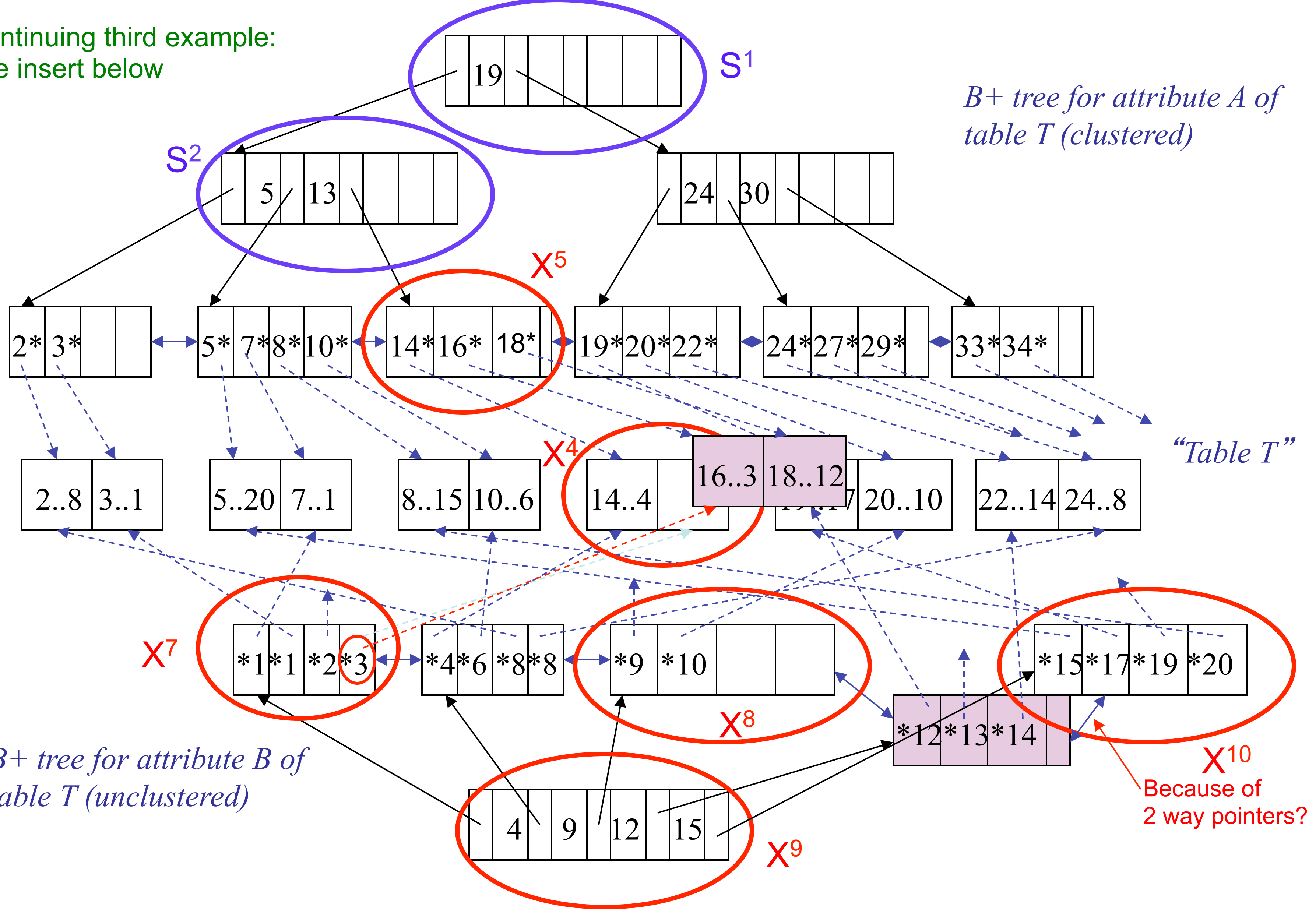
Continuing third example:
see insert below



INSERT INTO T (A, B, C) VALUES (**18**, 12, …)

Continued on next page

Continuing third example:
see insert below



INSERT INTO T (A, B, C) VALUES (**18**, 12, …)