SAP® MaxDB™
Logging
Version 7.7

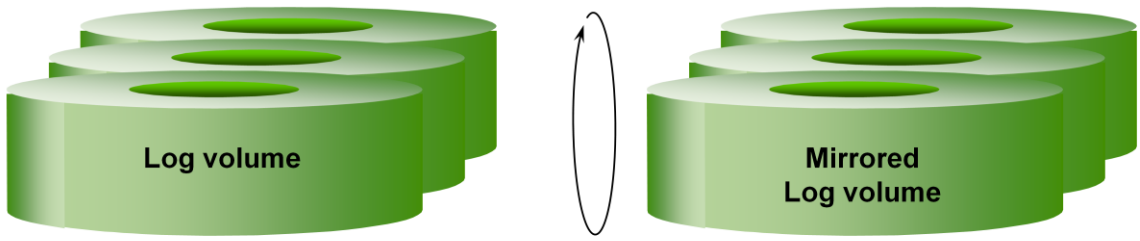
Heike Gursch
Werner Thesing

THE BEST-RUN BUSINESSES RUN SAP™

Overview Logging

- Log Configuration
- Log Backups / Overwrite Mode for the Log Area
- Log Volumes
- Log Full
- Redo and Undo Log Entries
- Log Queue Management
- DDL Statements
- DML Statements
- Savepoint
- Transaction Handling
- Parallel Restore Log / Restart
- DBIdent
- Homogeneous System Copy / Shadow Instance
- Hot Standby

Log Configuration



Configuration parameter:	Value:
UseMirroredLog	YES/NO
MaxLogVolumes	1-32

DBM commands to configure a mirrored log :

```
param_put UseMirroredLog YES
db_admin
db_execute RESTORE LOG VOLUME '<log volume>'
```


In the basic configuration, MaxDB writes all data changes to the log area. The log area consists of 1 to 32 log volumes.

The log area is overwritten in cycles. Before it can be overwritten, an area must be backed up. This does not apply to open transactions as the undo log entries for changes are recorded in the data area. At least one savepoint has to have taken place before the log entries can be overwritten.

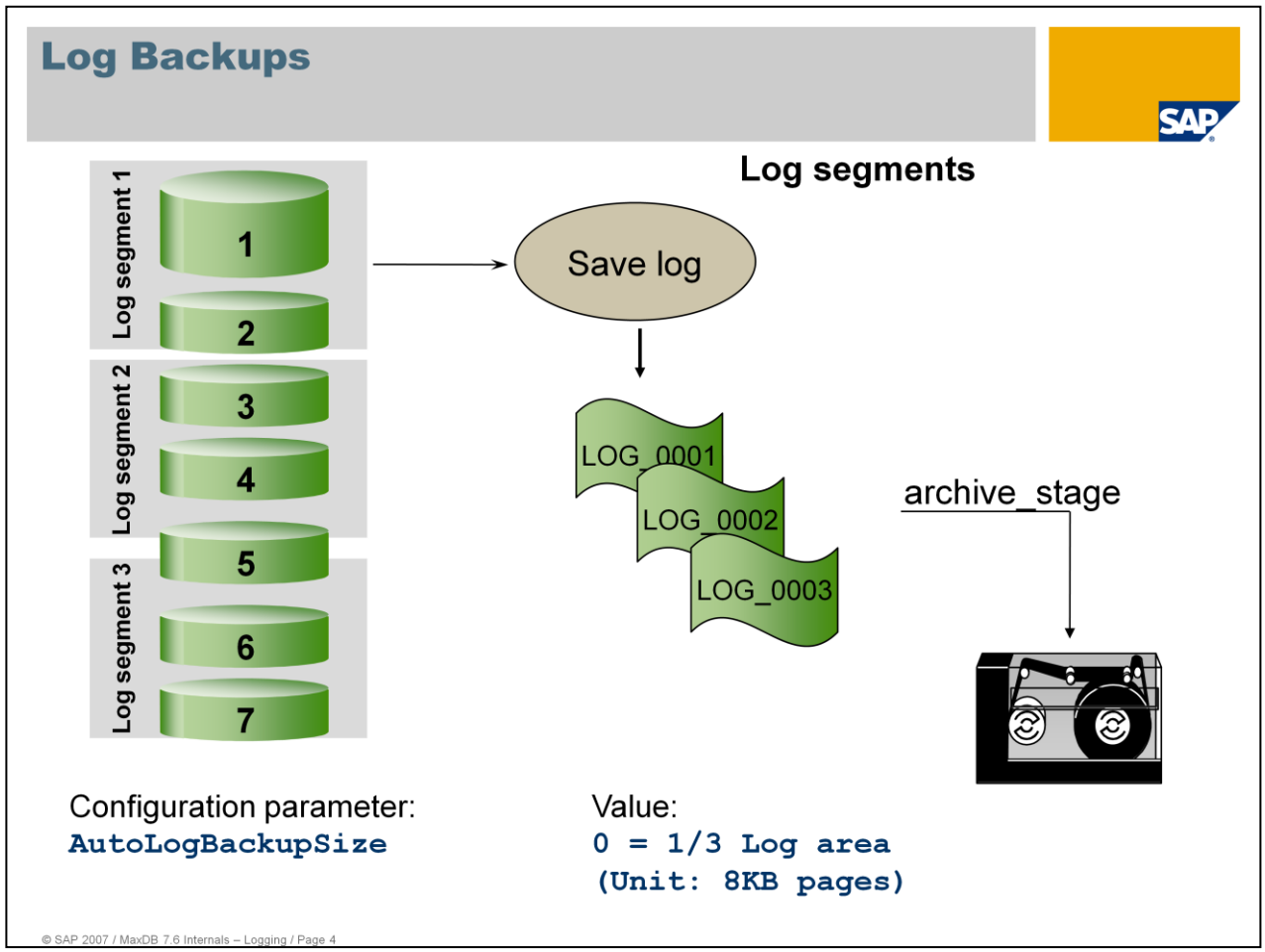
In systems in which the database has to ensure that data is persistent, the log volumes must be mirrored. This mirroring can be done by MaxDB. If the parameter UseMirroredLog (LOG_MIRRORED) is set to NO, the log must be completely mirrored with operating system functions or by the hardware (RAID 1).

- **If there is no copy of the log, disk error results in loss of data.**

For security and performance reasons, RAID 5 is not recommended. Striping the disks for the log volumes is only advisable if the disk system works with a large cache and the log volumes have been mirrored. The disk system cache should be large enough that the writing of log pages does not have to wait for physical disk I/Os.

The parameter MaxLogVolumes (MAXARCHIVELOGS) indicates the maximum number of log volumes. The log areas can be expanded up to this number in online mode.

If the log area is mirrored by the database, the failure of a log volume causes an emergency shutdown. A new disk is then needed. The new log volume can be integrated in the ADMIN state.
From version 7.4, the log volume can no longer be integrated in the ONLINE state.



The log is divided into fixed-length segments.

This segmentation does not represent a physical division of the log.

Log segments are backup units. They specify the size AutoLogBackupSize of the files in which the log pages are backed up. The database parameter AutoLogBackupSize (LOG_SEGMENT_SIZE) indicates the size of a log segment in log pages.

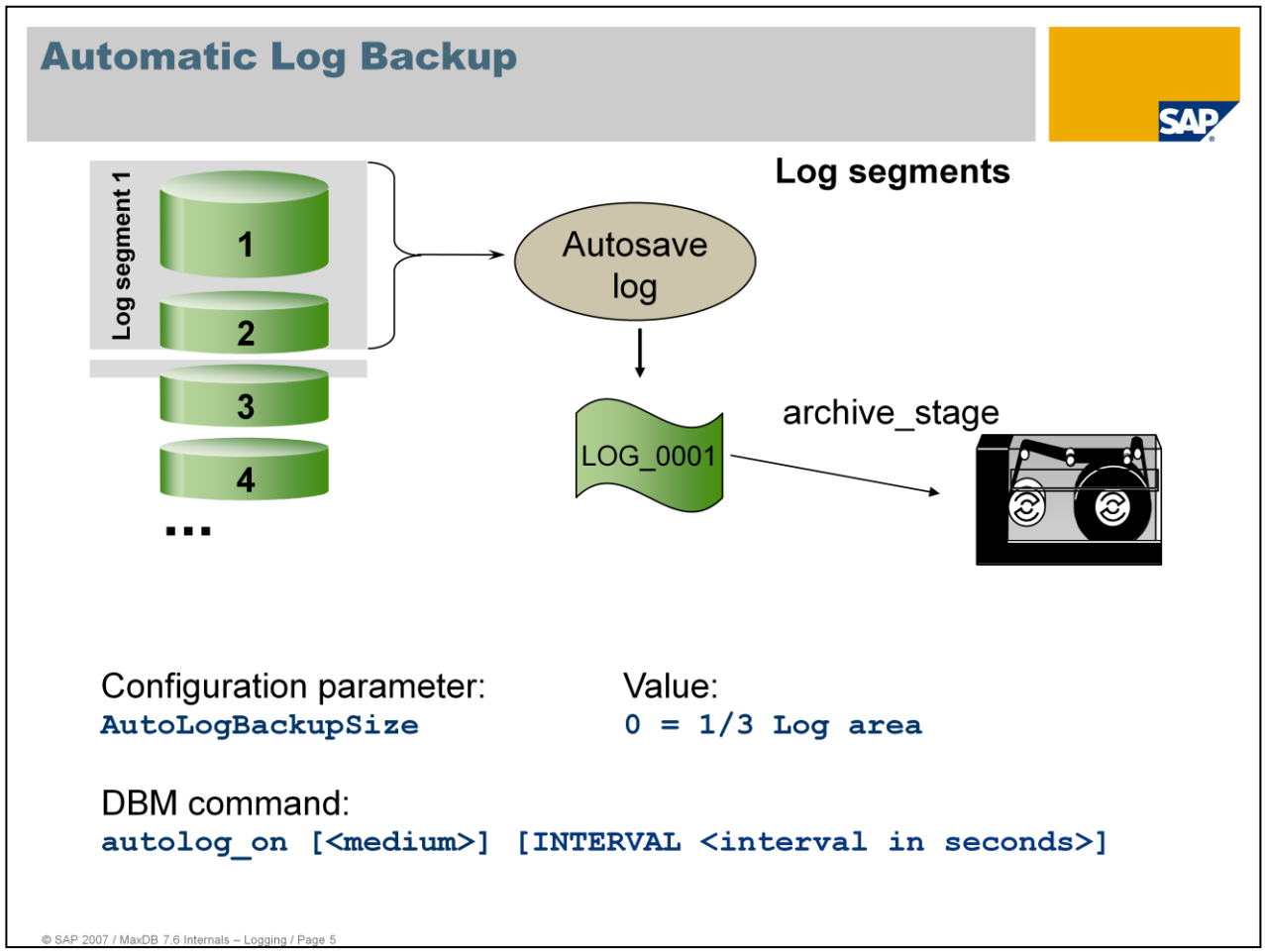
If the size 0 is entered for the kernel parameter , the database manager automatically calculates 1/3 of the of the total log area. If the log is expanded online, this does not change the size of the log segments.

Log segments are NOT closed by a savepoint.

Interactive log backup backs up all completed segments (segment-by-segment) and the log pages of the segment that is not set complete. So there can be backup files that are smaller than AutoLogBackupSize.

The segment size is no more than half of the log area.

SAP recommends saving log backups in backup files in a file system. These backup files can then be transferred to a backup tape with the Database Manager command archive_stage. External backup tools are supported.



The database kernel can create the log backup automatically.

You activate automatic log backup with the dbmcli command autolog_on.

The log backup is automatically created asynchronously upon completion of a segment.

A segment is completed when the following inequality is fulfilled:

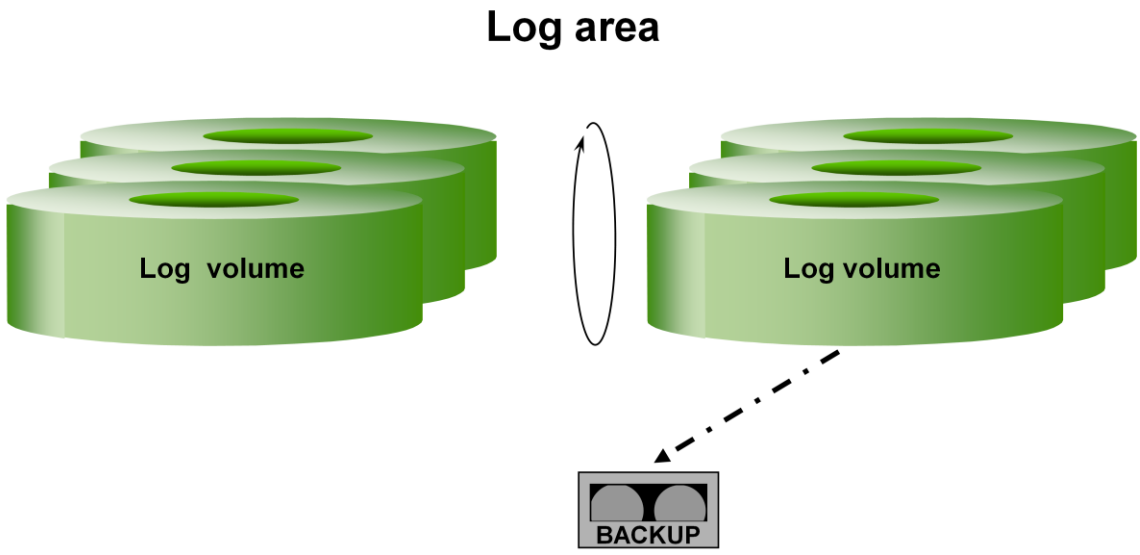
- Logpno of write pos – Last Logpno of last Segment > Segmentsize

The log backup is created using two server tasks: One server task reads the log pages from the log area and the other writes the log pages to the backup files.

As of MaxDB version 7.6.02 also a time interval may be set to launch the automatic log backup along this interval.

In versions smaller than 7.7.03 the parameter AutoLogBackupSize had the name LOG_SEGMENT_SIZE.

Overwrite Mode for the Log Area (without Backup)



DBM command (in state ONLINE or ADMIN):

```
db_execute SET LOG AUTO OVERWRITE ON/OFF
```


The log can be overwritten automatically without log backups. Use the DBM command „db_execute SET LOG AUTO OVERWRITE ON“ to set this status.

The behavior of the database corresponds to the log mode DEMO in older versions. From version 7.4.3, this behavior can be configured online.

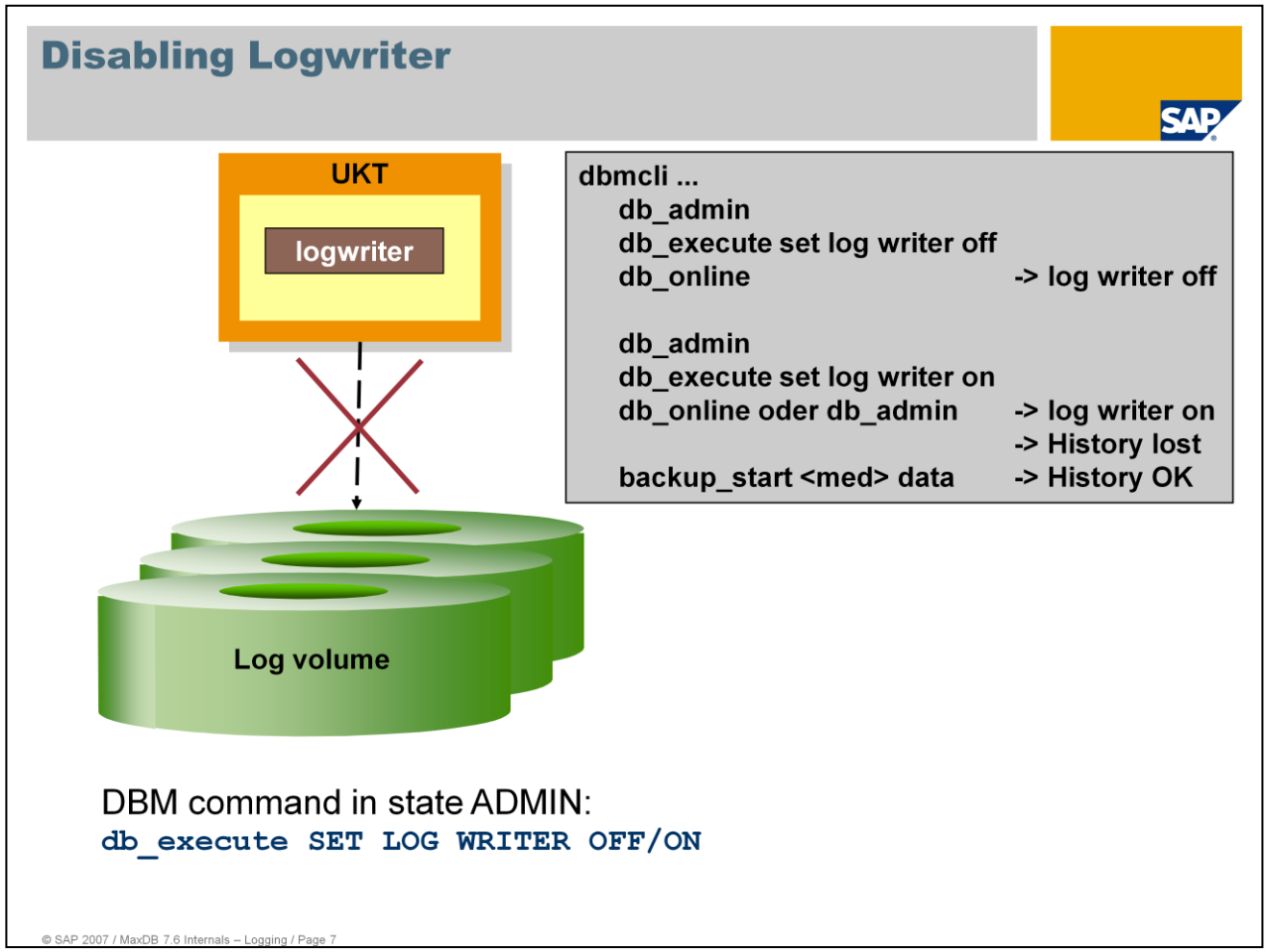
Log backups are not possible after activating automatic overwrite. The backup history is then interrupted; this is indicated in the backup history by the identifier HISTLOST (file dbm.knl).

The backup history starts again when you deactivate automatic overwrite without log backup using the command „db_execute SET LOG AUTO OVERWRITE OFF“ and create a complete log backup in the ADMIN or ONLINE state. Remember to reactivate automatic log backup when this is desired.

Automatic overwrite of the log area without log backups is NOT suitable for production operation. As there is no backup history for the subsequent changes in the database, it may not be possible to redo transactions in case of a recovery.

Advantages compared to the log mode DEMO in version 7.3:

- Automatic overwrite of the log area without log backups can be switched on and off in online mode.
- A complete data backup in online mode restarts the backup history.
- Automatic overwrite of the log area without log backups can be activated despite mirrored log volumes. Thus with the command „db_execute SET LOG AUTO OVERWRITE OFF“, the mirrored log is not reconfigured.



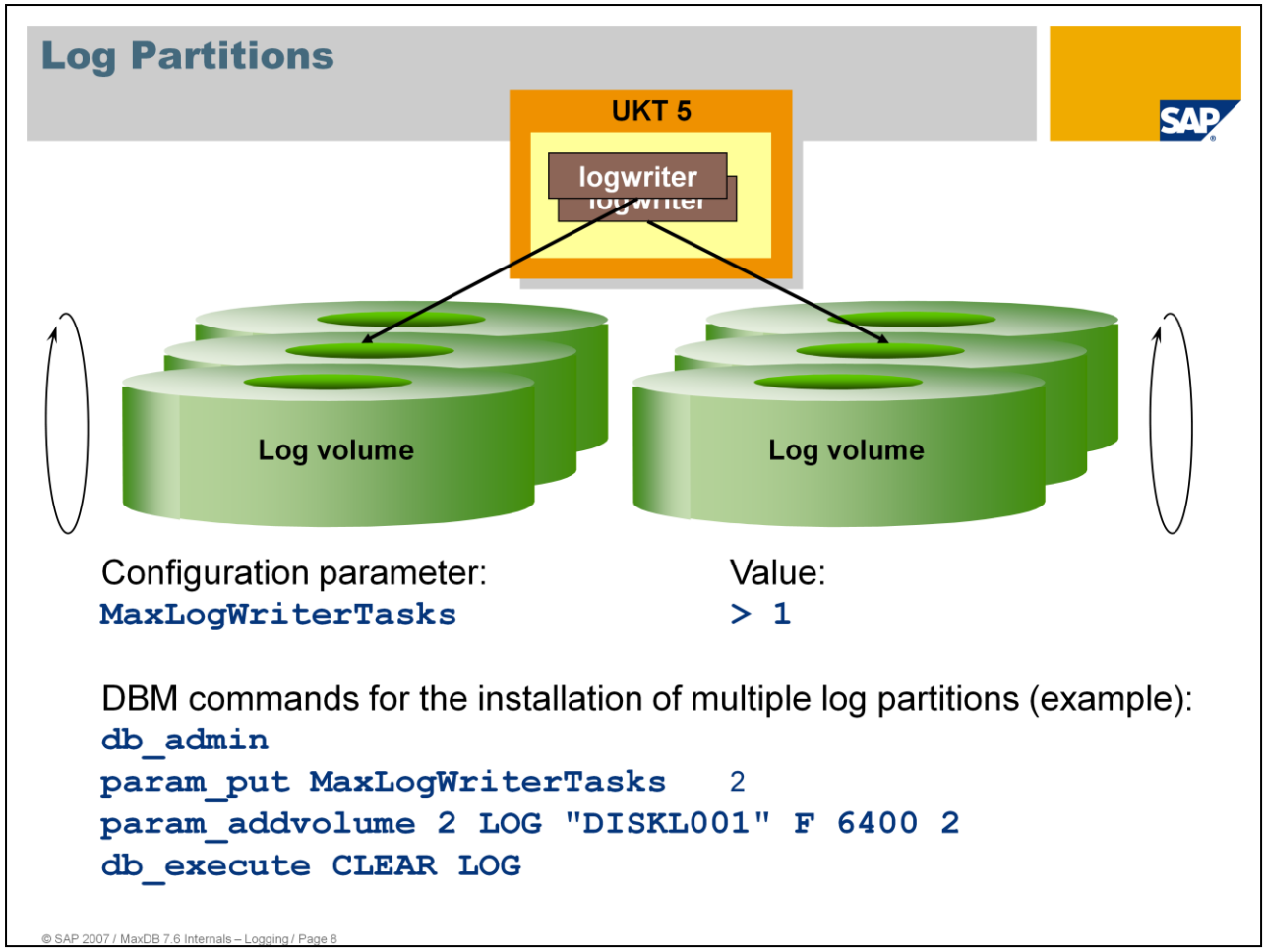
From version 7.4, it is possible to switch off the log writer. This is now allowed because the database is made consistent with each savepoint. A restart is possible without the log because all open transactions can be redone on the basis of the last savepoint.

You switch the log writer off with the DBM command „db_execute SET LOG WRITER OFF“. To do so, start the database in the ADMIN state.

Log backups are not possible after the log writer has been switched off. The backup history is interrupted, as indicated by the identifier HISTLOST.

The backup history restarts when you reactivate the log writer with the DBM command “db_execute SET LOG WRITER ON“ in the state ADMIN and create a complete log backup in the ADMIN or ONLINE state. Remember to reactivate automatic log backup when this is desired.

The log writer may not be switched off for production operation. The function serves to accelerate administration tasks such as upgrades and big load jobs.

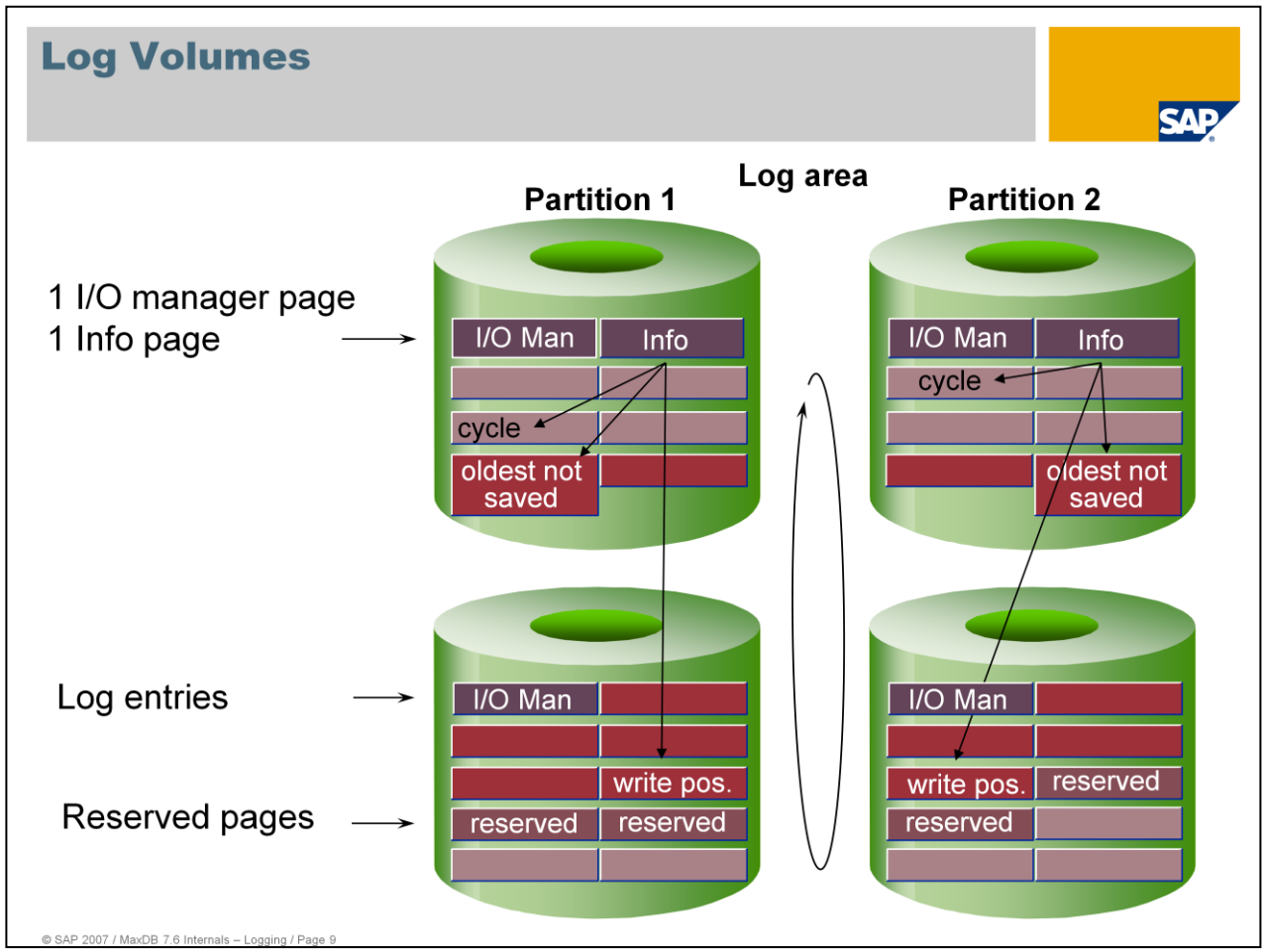


As of version 7.7 MaxDB allows the use of multiple log partitiions. With parallel writing to the log volumes the database prevents bottlenecks during the access to the log queue and additionally wait situations for writes into the log volumes.

Partitions and also volumes of the partitions may have different sizes.

Normally user tasks of a UKT are assigned to a special log writer and therewith to a partition. This implies that for some tasks the state „log full“ might occur even if there is still some free space for a user task of another UKT in the corresponding log partition. Perform a backup of the log area if the state „log full“ is shown.

With the use of the CLEAR LOG command the backup history is interrupted. Make sure that a complete data backup is performed. If desired the automatic log backup can be switched on again.



Log pages are always 8KB. They are written from the memory to the volume when they are full or when a log entry ends a transaction (COMMIT, ROLLBACK).

The log area is overwritten in cycles. Log pages can only be overwritten when they have been backed up. (Exception: SET LOG AUTO OVERWRITE ON).

In the header of the log there are two pages that are not cyclically overwritten.

The first page contains information from the I/O manager, for example the volume number and the numbers of the predecessor and successor volumes, if available.

Positions are administered on the info page. These include:

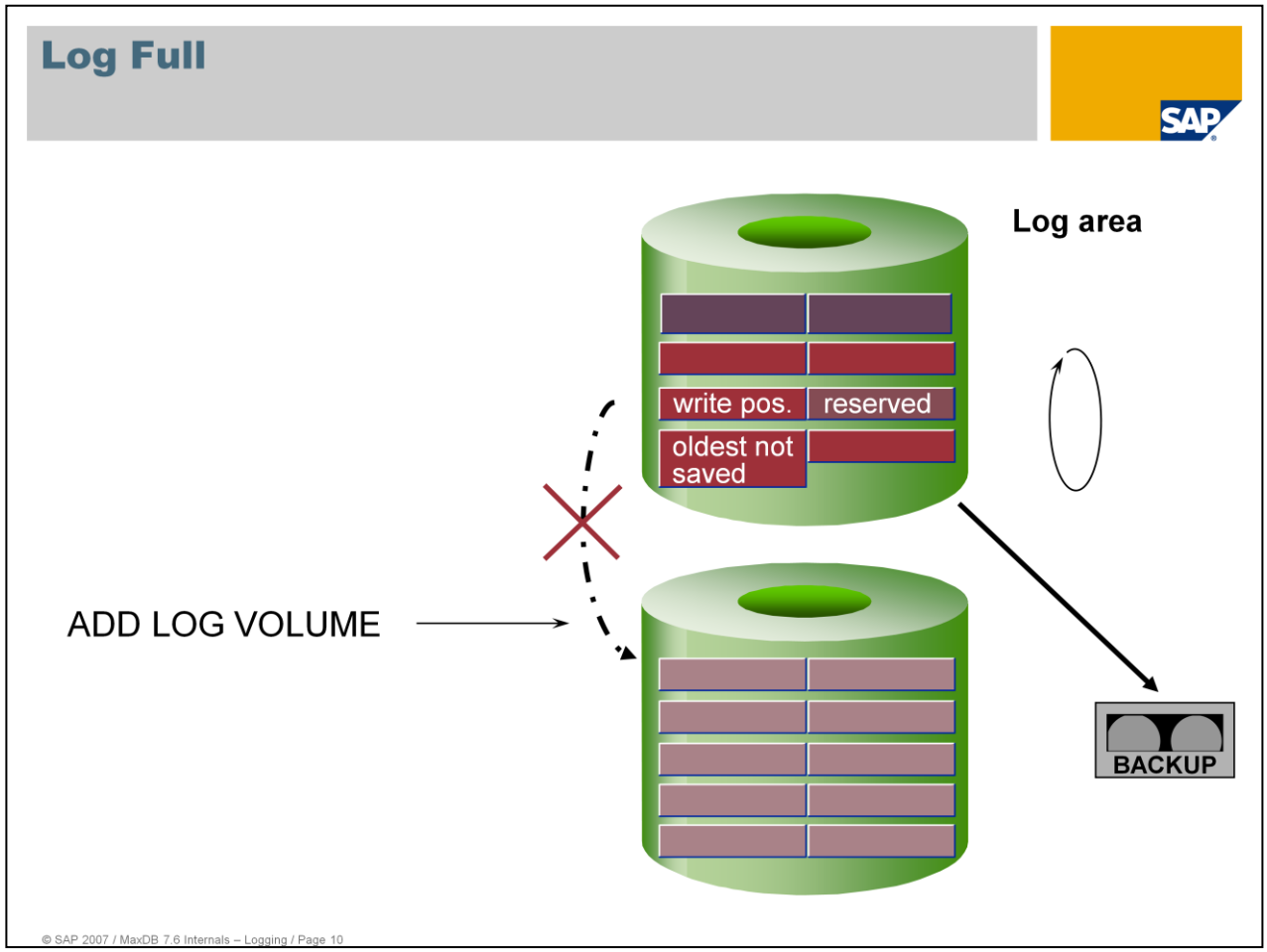
- the current write position,
- the position to which overwriting can proceed,
- the position that has not yet been backed up,
- the number of the last log backup.

The info page is written to the volume with each savepoint and, as a precaution, every 500 log pages.

If the current write position reaches the page up to which may be overwritten, the log is full. In this case, the database allows no further write transactions. Clients can still log on, however, as, from version 7.4, a connect is not written in the log.

The reserve pages cannot be filled with normal log entries. Following a log full status, at the next successful restart the first unused page is filled with a savepoint.

Each partition keeps its management information separately.



When the log has reached its capacity, creating a log backup is the only way to continue working in online mode. A log backup can be created in the ADMIN or ONLINE state. A log backup can also be created by activating automatic log backup.

If the database has the Log Full status, this is not reset by an expansion of the log area. The current write position is merely moved sequentially. It is not moved from the middle of a log volume to the beginning of the subsequent log volume.

If the log area is expanded through the addition of a new log volume, the new volume is only written when the current write position comes out of the last page of the predecessor volume.

When the Log Full status occurs, the database kernel writes a savepoint. Log entries can only be overwritten if they have been backed up and the position of the last savepoint in still in the log area (Exception: SET LOG AUTO OVERWRITE ON).

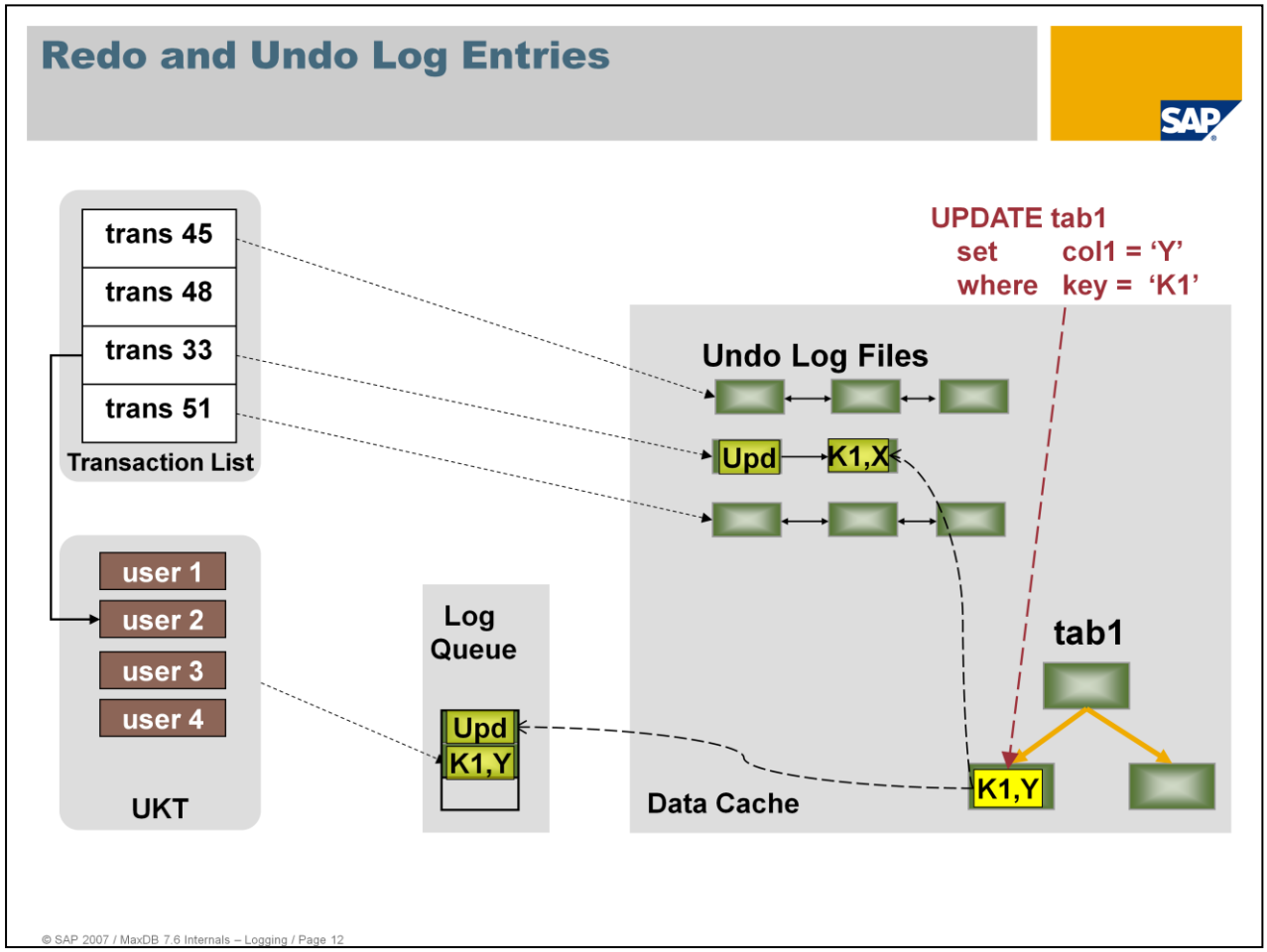
Log full state may occur for each partition. Some user tasks may be in a waiting state because of a filled log volume of their partition while other user tasks working with another partition can still perform changing operations.

The DBM extends the log area by one volume at a time. If you want to use multiple partitions specify the new volumes individually with their corresponding partition:

```
db_addvolume LOG DISKL001_2 F 18750 PARTITION 1
db_addvolume LOG DISKL002_2 F 18750 PARTITION 2
```

Transaction Management

CONNECT:	➢ Start of a MaxDB session ➢ Implicit start of a transaction ➢ No entry in log volume
COMMIT:	➢ Successful end of a transaction ➢ Implicit start of a new transaction ➢ Entry in log volume (only after changes)
ROLLBACK:	➢ Rollback of a transaction ➢ Implicit start of a new transaction ➢ Entry in log volume (only after changes)
COMMIT RELEASE:	➢ Successful end of a transaction ➢ End of the MaxDB session ➢ Entry in log volume (only after changes)
ROLLBACK RELEASE:	➢ Rollback of a transaction ➢ End of the MaxDB session ➢ Entry in log volume (only after changes)



MaxDB administers redo and undo logs separately.

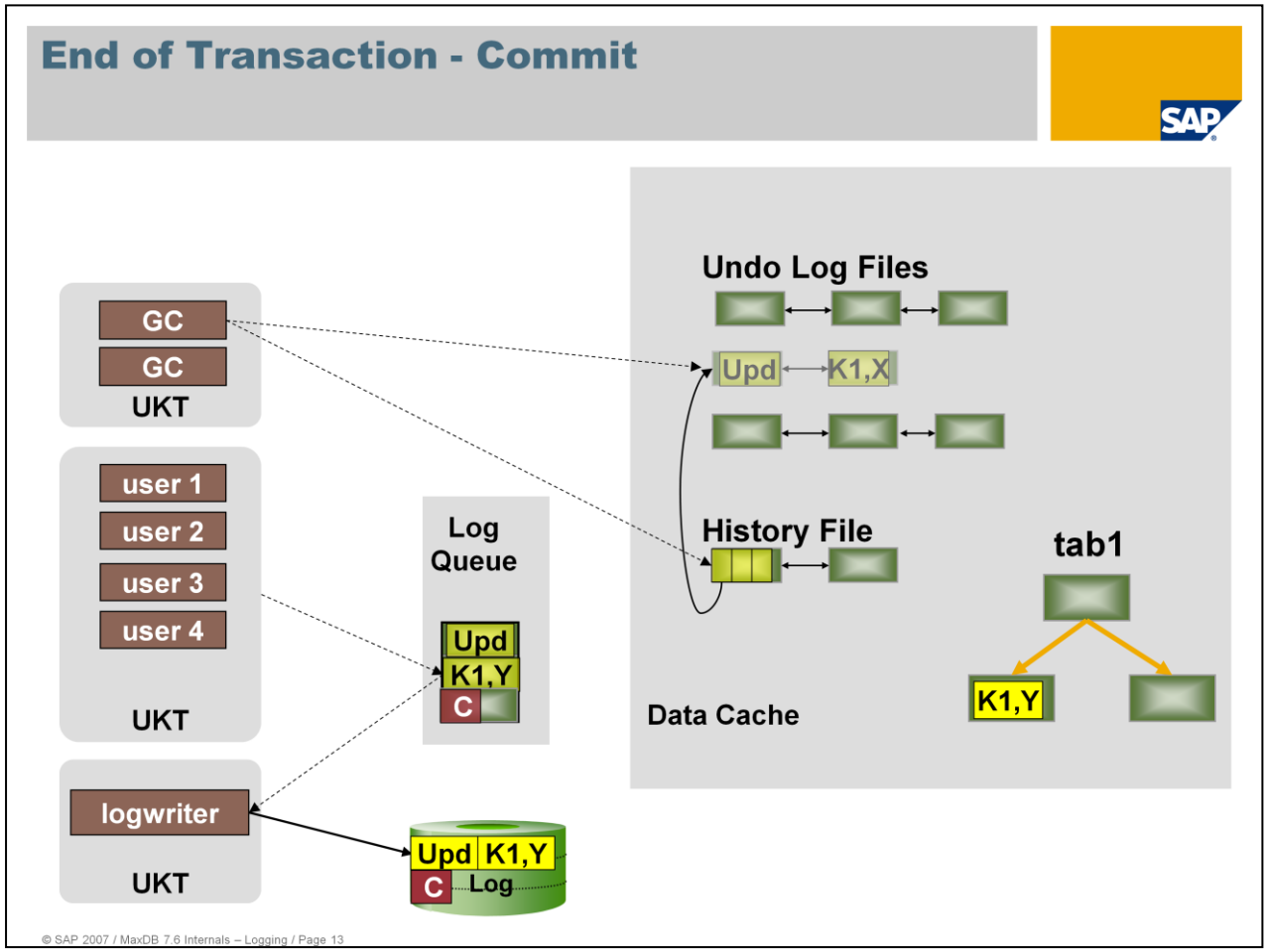
In the present example, user session 2 is working in transaction 33. It is updating a record in the table "tab1".

Transaction 33 has a link to an undo log file. The user enters the field values that were valid before the update into the undo log file with the action update and the primary key of the record. Each transaction uses its own undo log file. This prevents collisions from taking place while accessing undo log files.

The user copies the new field values together with the action and the primary key of the record into log queue.

Undo log entries enable you to roll back a transaction.

The redo log entries enable the complete recovery of data following a crash or disk errors.



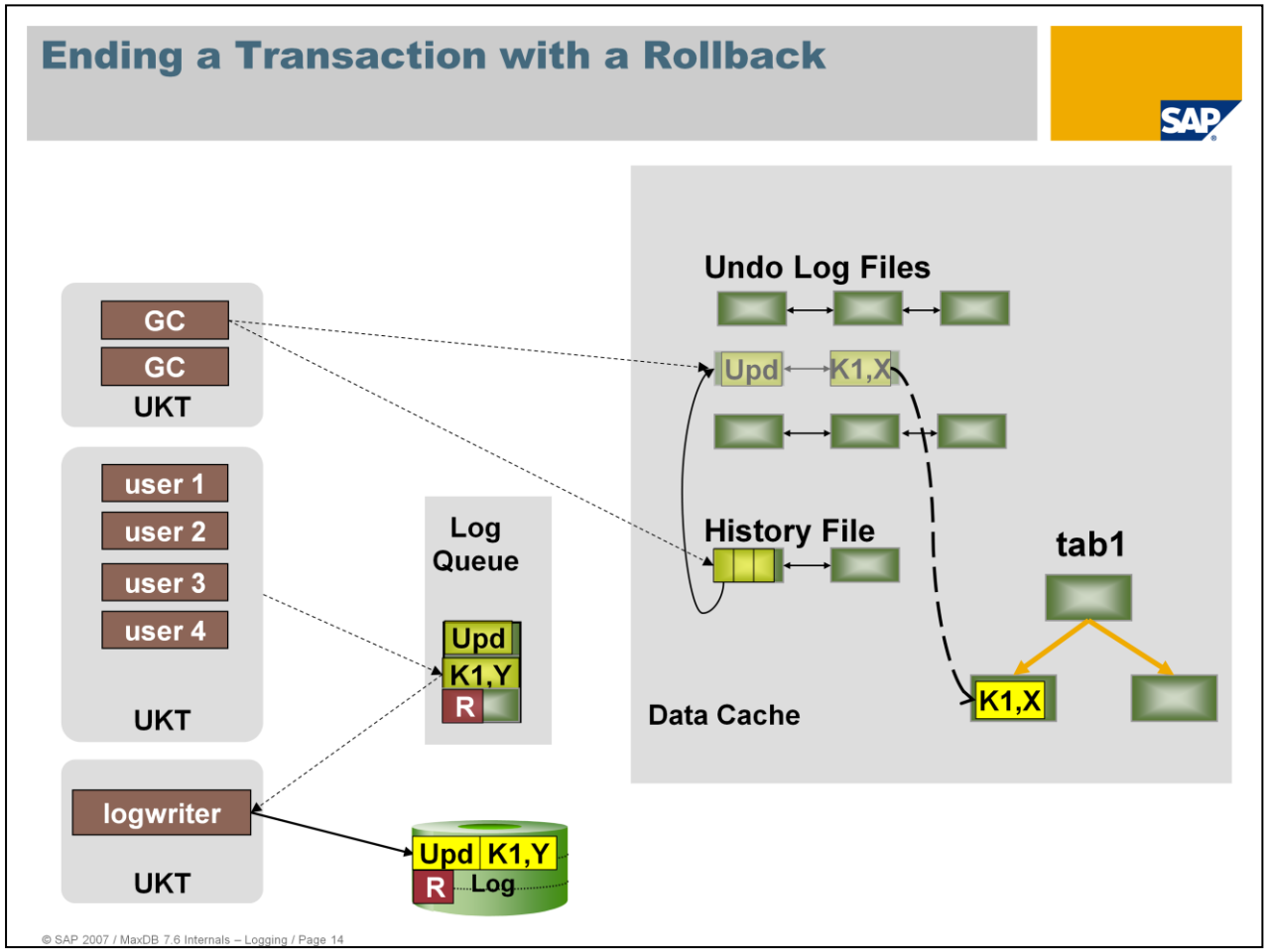
In the present case, the transaction is closed with a commit.

The user enters the commit in the log queue and activates the log writer.

The log writer reads all pages that contain entries of the transaction and were not yet written from the log queue and writes them to the log volume.

The user deletes the undo log file when the log writer has confirmed the successful write operation for the commit.

As of version 7.6, Garbage Collectors regularly check the history file every few seconds. They delete the undo log files found there. Thus the user task is disburdened by the delete operation of the undo log entries. The database can then confirm the Commit to the application more quickly.



In the present case, the transaction is closed with a commit.

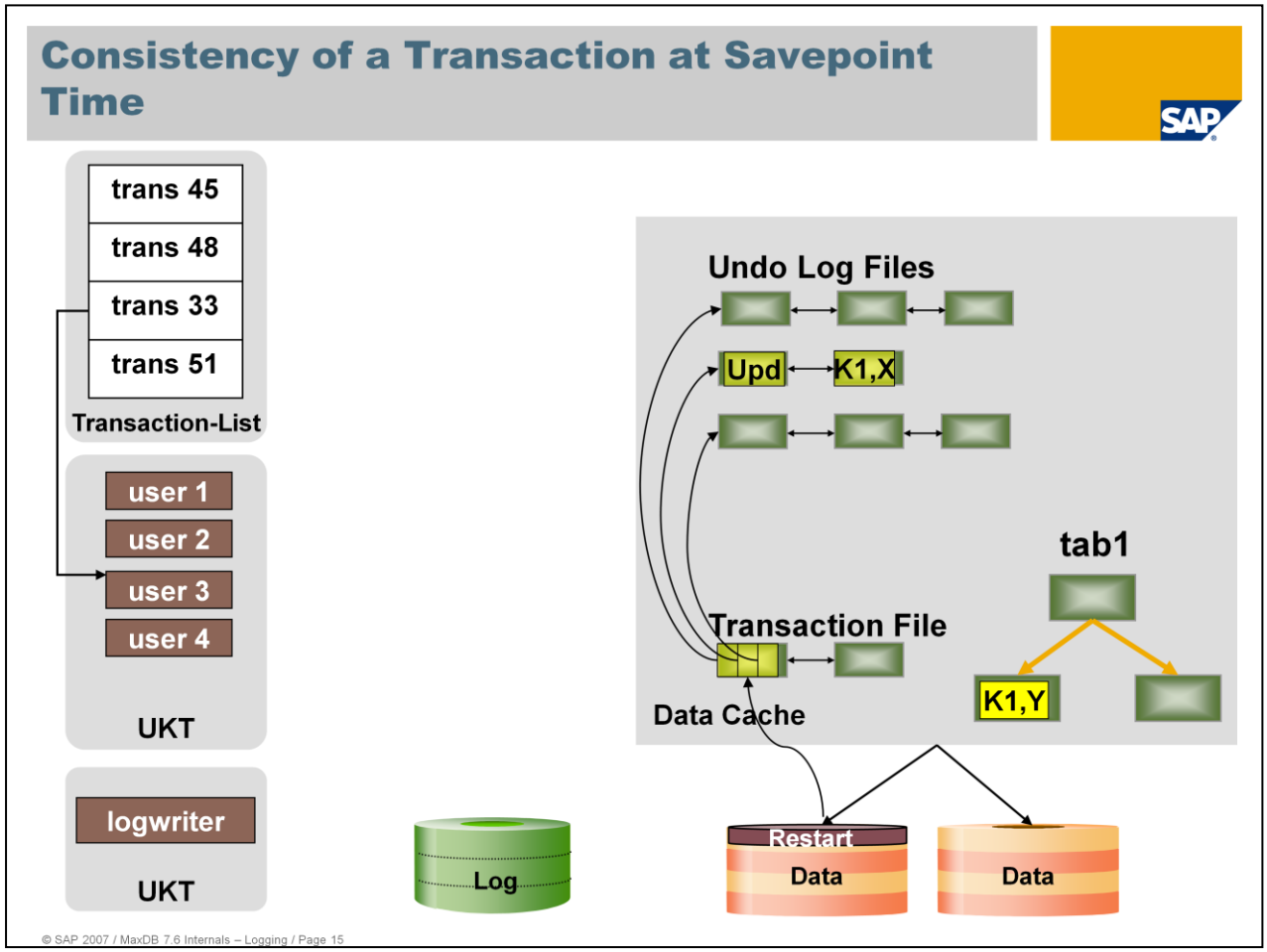
In a rollback, the user first retracts all the changes to the table data that are listed in the undo log file.

The user enters the commit in the log queue and activates the log writer.

The log writer reads all pages that contain entries of the transaction and were not yet written from the log queue and writes them to the log volume.

The user deletes the undo log file when the log writer has confirmed the write operation to the log volume.

As of version 7.6, Garbage Collectors regularly check the history file every few seconds. They delete the undo log files found there. Thus the user task is disburdened by the delete operation of the undo log entries. The database can then confirm the Rollback to the application more quickly.



Undo log files consist of one or more permanent data pages. A savepoint handles them just like data pages with table and index information; in other words, they are written to the data volumes.

History administration notes the first page of each undo log file in the history file. The first page of the history file is recorded in the restart page at the end of the savepoint. So a restart can start at the last savepoint and find the actions of all open transactions.

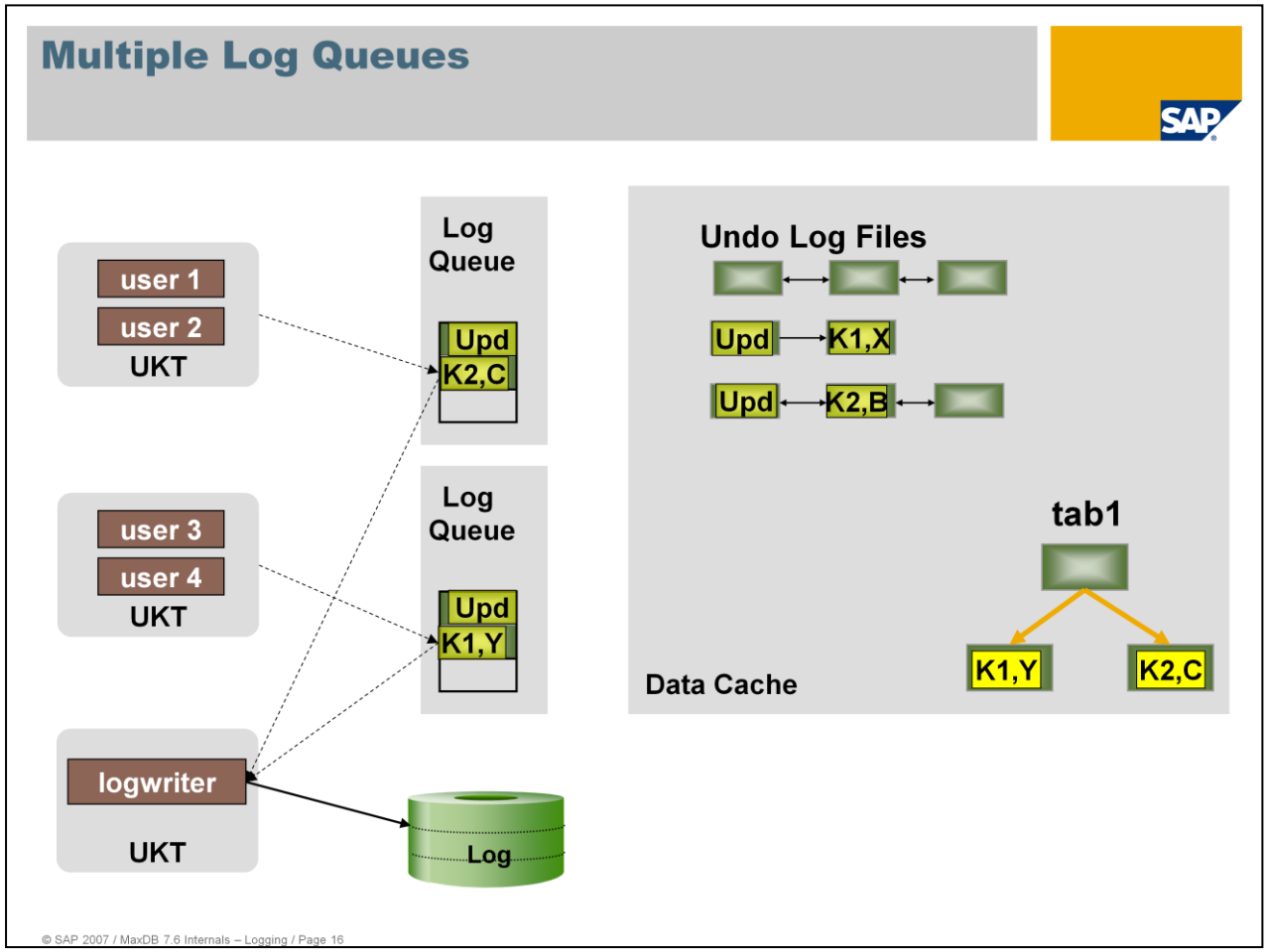
With a savepoint, the database writes a status in the volumes that can be reassumed without use of the log.

If the savepoint is created for a data backup, this data backup can be imported into another instance as a system copy. The data backup also contains the undo log files for open transactions.

A restart of the system copy works even though no corresponding logs have been provided. In this case, the restart undoes all transactions that were open at the time of the savepoint. The requisite information is available in the form of the undo log files.

Before restarting the system copy, after restoring the data backup you can also import log backups.

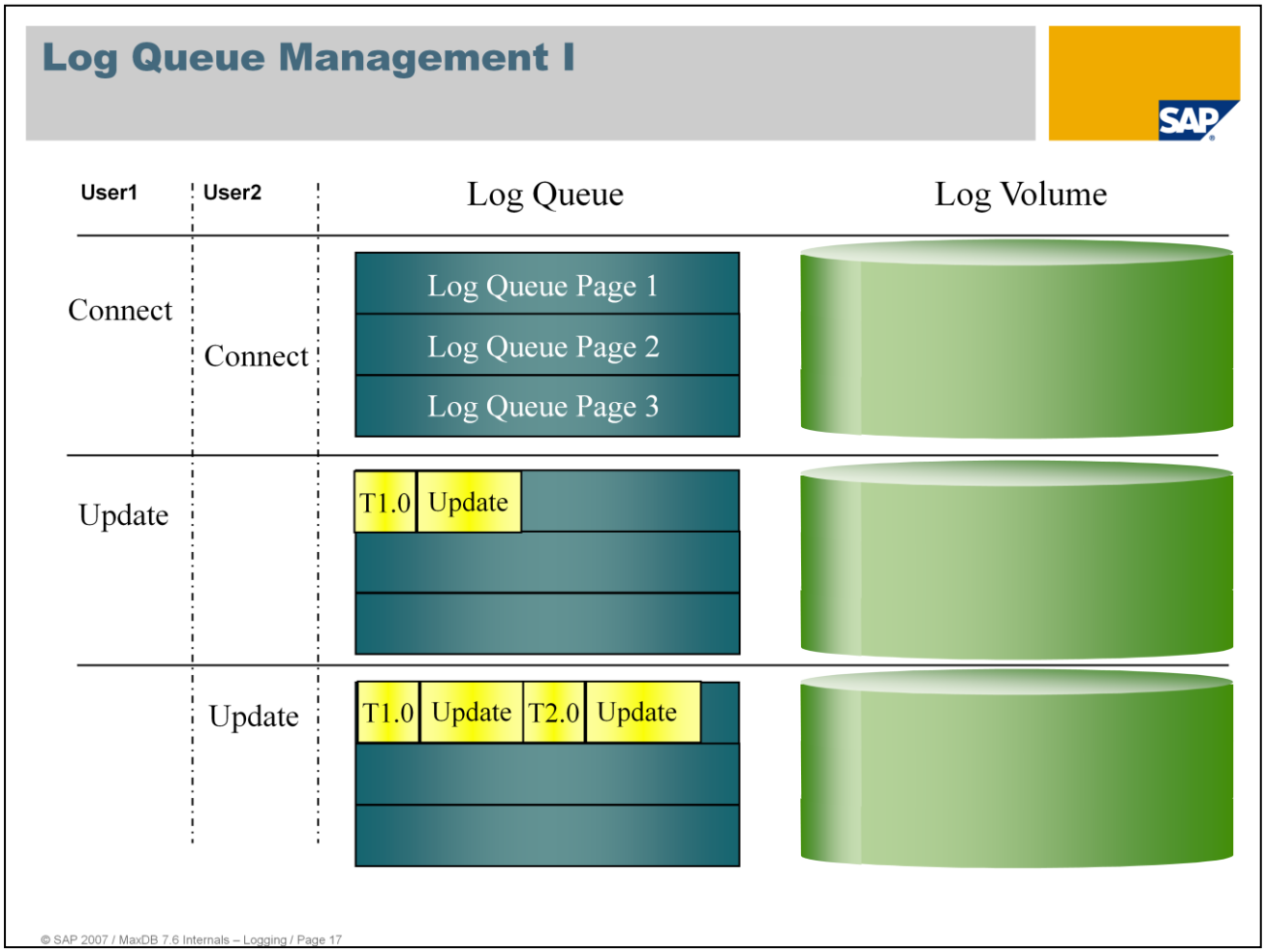
As you can restore the instance to a transaction-consistent status with the savepoint data, version 7.4 and up no longer feature the **checkpoint** familiar to users of earlier versions.



As of version 7.6, MaxDB supports the use of multiple log queues. The database parameter LogQueues (LOG_QUEUE_COUNT) determines the number of log queues.

In the standard, the value for LogQueues is equivalent to the value for MaxCPUs (MAXCPU). Each UKT with user tasks writes to its own log queue. This prevents collisions at the log queues.

The database still works with a logwriter, which imports the log pages from the log queues and writes them to the log area.



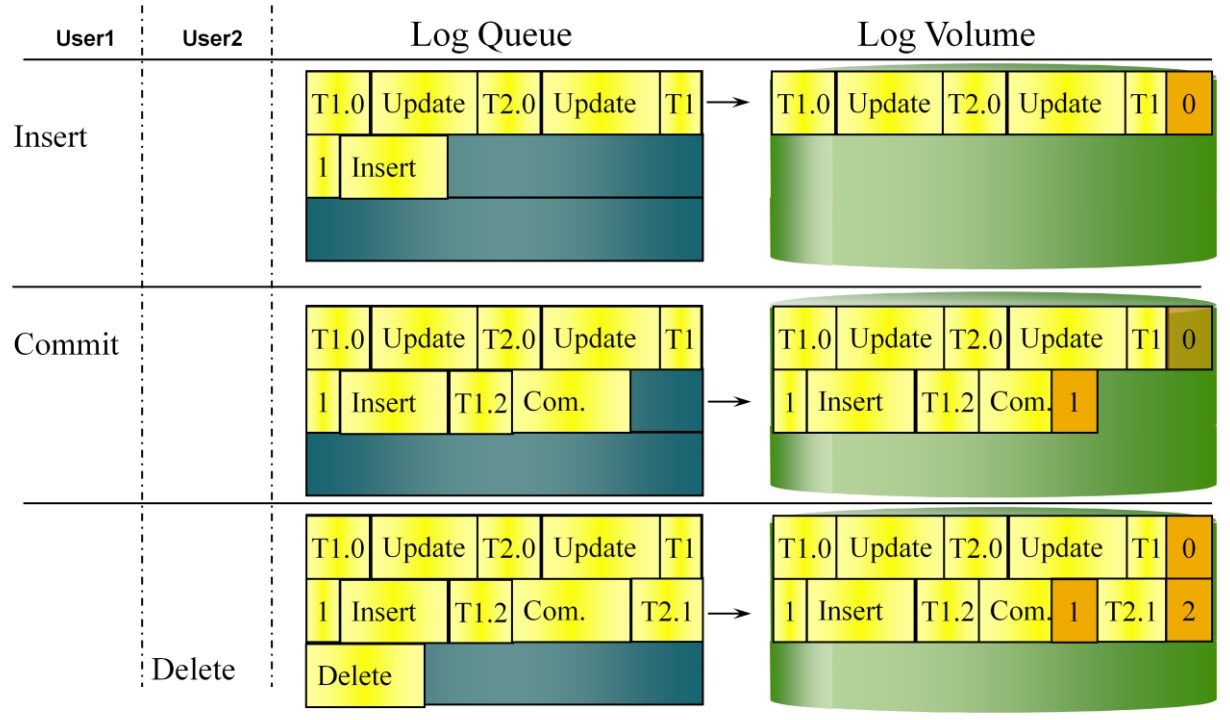
In this example the log queue has a size of 3 pages. Users write the redo log entries in the log queue, but not directly to the log volume.

Users 1 and 2 log on to the database. Only redo log entries are written in the log queue, so the connect does not need to be written in the log.
In version 7.3, the connect was also written in the log. Thus it was not possible to log on to the database in the Log Full status.
As of version 7.4, logon if status is Log Full or Database Full is possible.

User 1 executes an UPDATE statement in the database. An update changes data in the database, so a redo log entry is made for this statement in the log.

User 2 then executes an UPDATE statement. The entry in the log queue gets the ID T2.0.

Log Queue Management II



The INSERT executed by user 1 fills the log queue page. The first part of the redo log entry is written in the first page of the log queue page. The second part is written in the next log queue page. The user now instructs the log writer to copy the log page to the log volume.

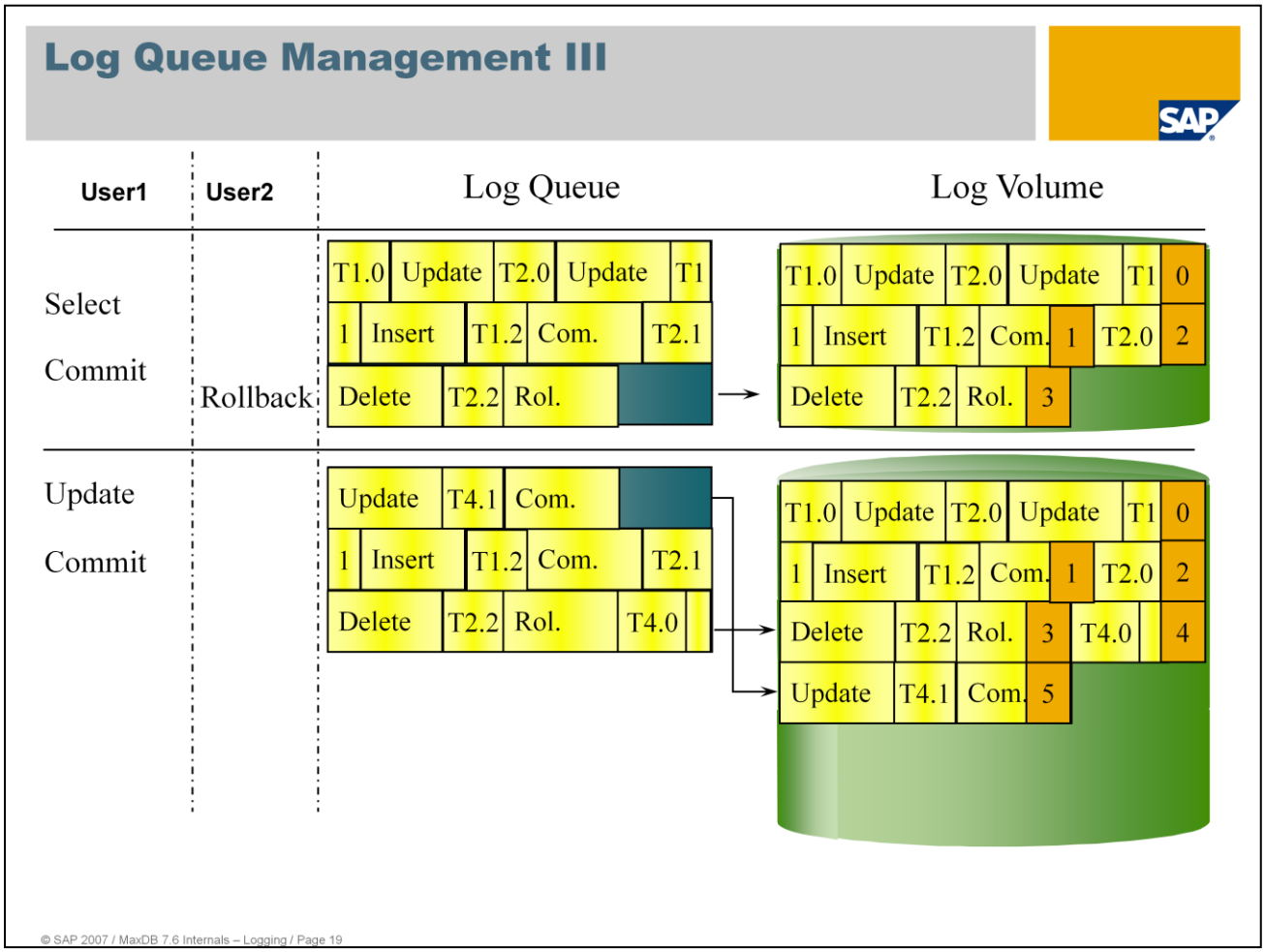
An I/O sequence is assigned for each write I/O in the log volume. This I/O sequence serves to synchronize the parallel restore log or restart. When the first log page is written, it is assigned the I/O sequence 0.

User 1 ends the transaction with a commit. The transaction can only be confirmed as completed and ready for application when the commit is in the log volume. The log writer writes the incompletely-filled log page to the log volume and assigns a new I/O sequence.

User 2 fills the second log page with the redo log entry for a delete. The log writer writes the page to the log volume. User 2 can continue to work and does not have to wait until the page is in the volume.

If multiple log queues are used, the logwriter will not overwrite blocks in the log. New entries always will be put to the next block. This leads to a certain higher log consumption.

If multiple log partitions are used the I/O sequence numbers may be distributed to different log volumes. Redo Log recomposes the order of sequences in a consecutive manner when reading from the log partitions.

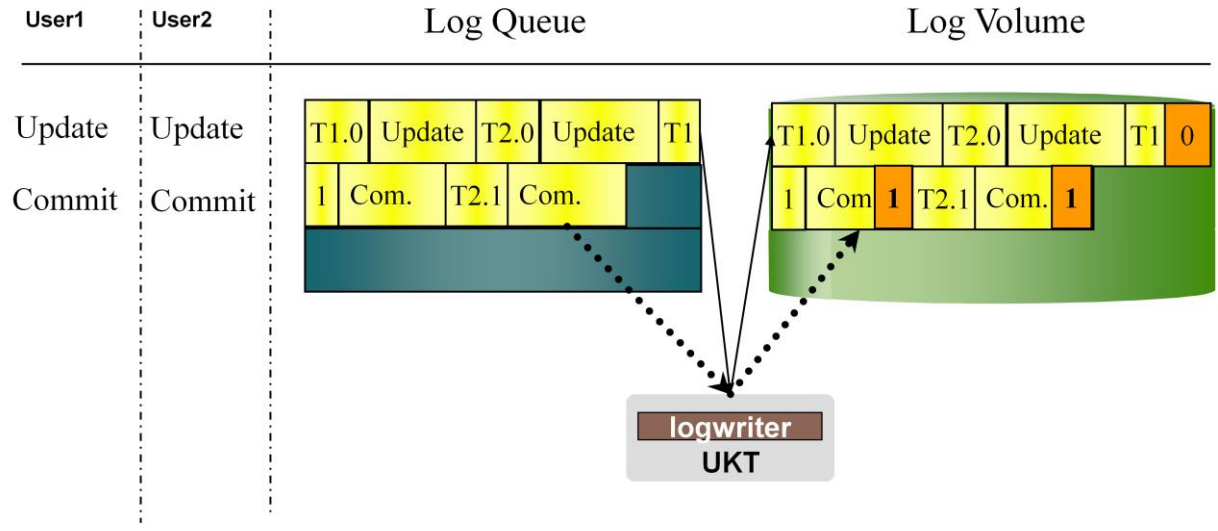


User 1 selects only data for the duration of the transaction. The transaction 3 makes no changes. No log entries are written for the selects and the subsequent commit.

User 2 ends the transaction with a commit. When the transaction is rolled back, the undo log entries are read from the undo log files in the data area. When all undo log entries have been rolled back, the redo log entry for the rollback is written in the log queue. The user waits until this entry is in the log volume.

User 1 makes changes and completely fills the log queue. Further entries are now written in the first page of the log queue. With the commit, this page is also copied to the log volume. As the log area is generally larger than the log queue, writing can continue there.

Group Commit



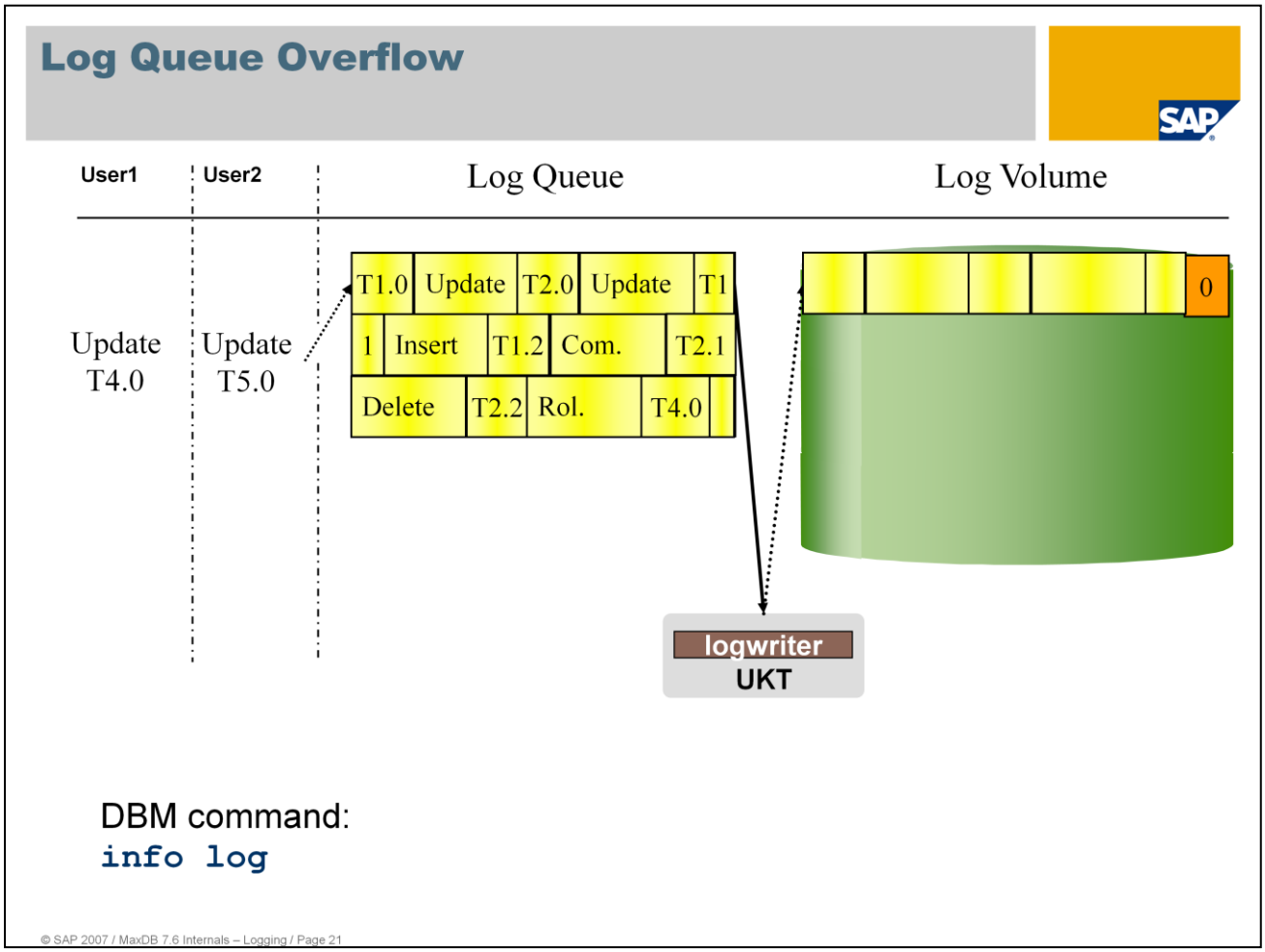
Configuration parameter:
`MaxLogWriterDelay`

Default value:
`0 µs`

When the log writer is busy copying log pages from the log queue to the log volume, multiple commit and rollback entries can accumulate in the current log page. These can then be copied to the log volume with an I/O. This effect is known as a group commit.

Group commits can be aided by setting the parameters MaxLogwriterDelay (_DELAY_LOGWRITER). In the benchmark environment, this led to minor improvements in throughput. In production operation, the standard parameter settings should be used.

If the log writer is very busy, log pages that have not yet been copied to the log volume by the log writer can gather in the queue. In this case, the log writer combines several pages into a larger block and copies them to the log volume with an I/O.



If the log writer cannot copy the log pages in the log queue fast enough, the log queue can fill up with pages that have not yet been written. This effect is known as log queue overflow.

If this happens, all users who want to perform changes have to wait. Thus this situation can be very critical for performance.

In most cases, log queue overflows occur due to slow write I/Os for the log volumes. Mostly you will get less overflows if the I/O is accelerated. In high workload situations the configuration of additional log partitions may reduce the number of overflows. Sometimes it be necessary to enlarge the log queue.

The DBM command "info log" displays the number of log queue overflows since the last restart of the instance.

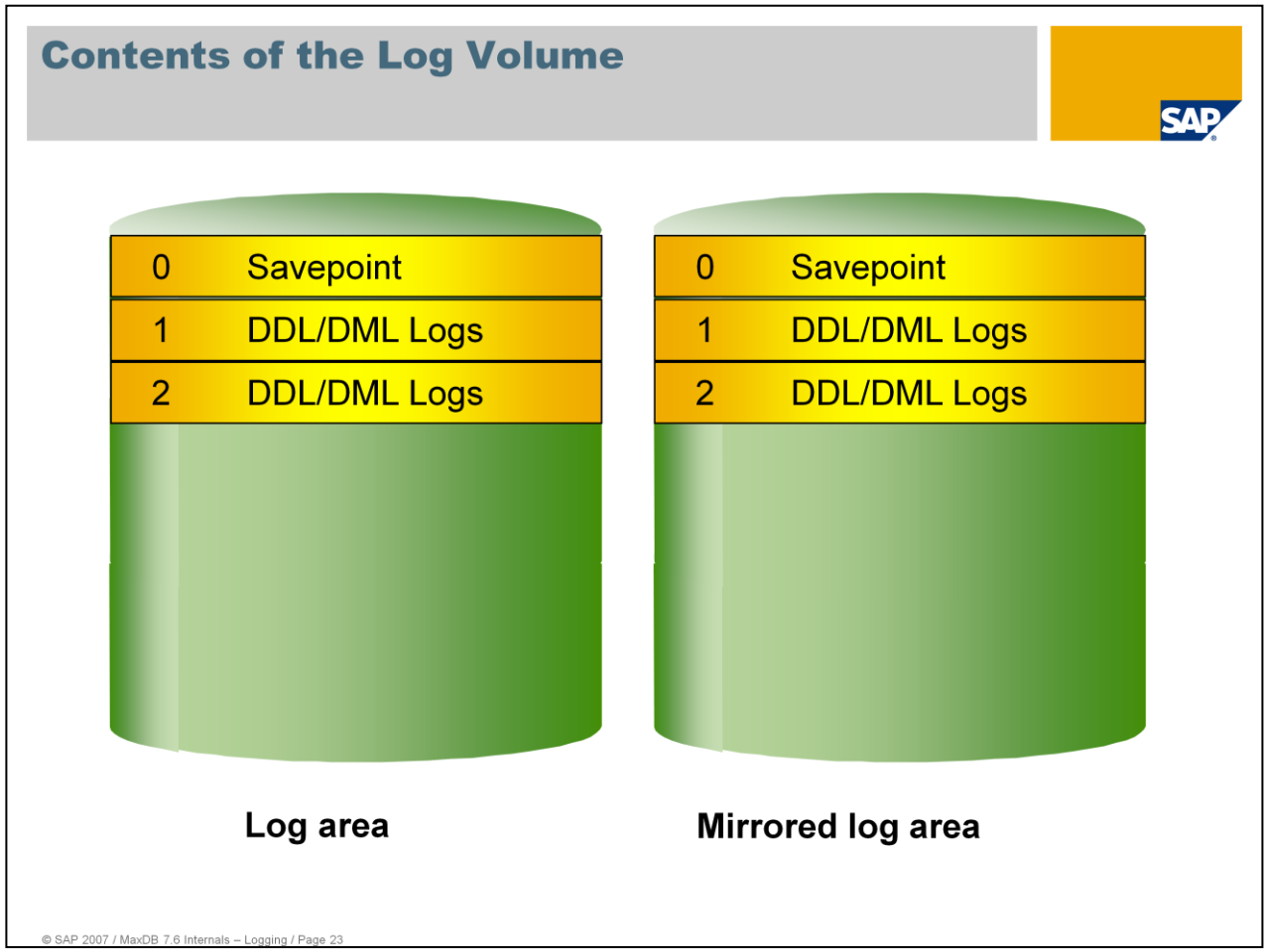
Log Queue: Who has to wait?

Copying entries from the log queue to the log volume is initiated by:

Condition	Transactions	I/O Wait
Log Page full	After copy the user process can immediately write to the next log queue page.	No
Commit / Rollback	The ending transaction has to wait until all log entries belonging to it have been written to disk.	Yes
Savepoint	None of the transactions has to wait.	No

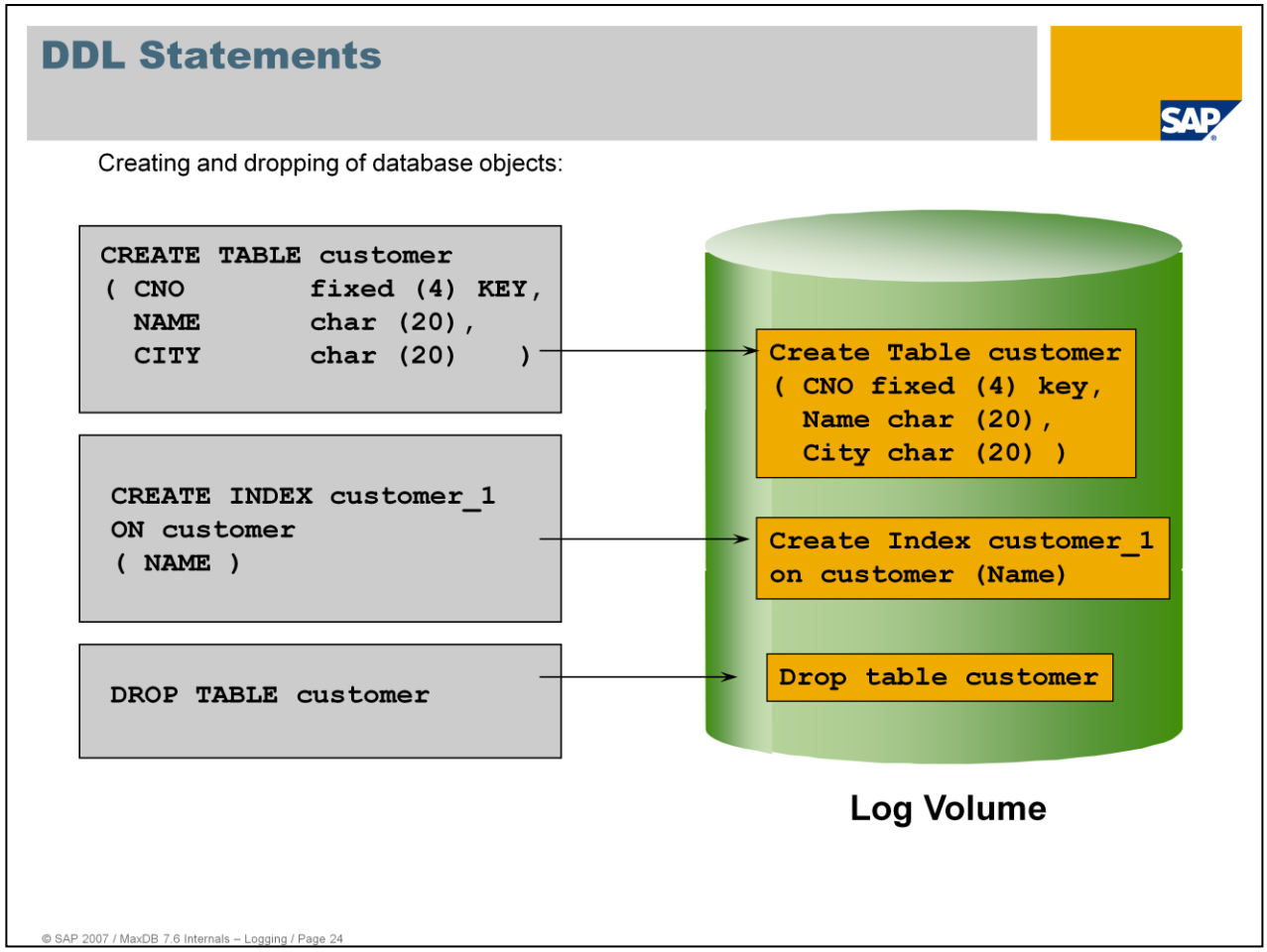
The copying of the log queue to the log volume is triggered by:

- A full log queue page
  The user transaction can begin to write the subsequent log page as soon as the entry is in the log queue. It does not have to wait until the log writer has written the entry in the log queue to the log volume.
- A commit or rollback
  The transaction has to wait until all corresponding entries have been written to the log volume.
- A savepoint
  User transactions can continue working. They do not have to wait until the savepoint entry has been written to the log volume.



In a new installation of the database, the log volumes are formatted. The DBM command db_activate without option RECOVER launches the first restart of the database. A savepoint is also written. Thereafter, the database is ready for use and all database components can connect to the database.

All changes by data definition and data manipulation commands are written in the log.



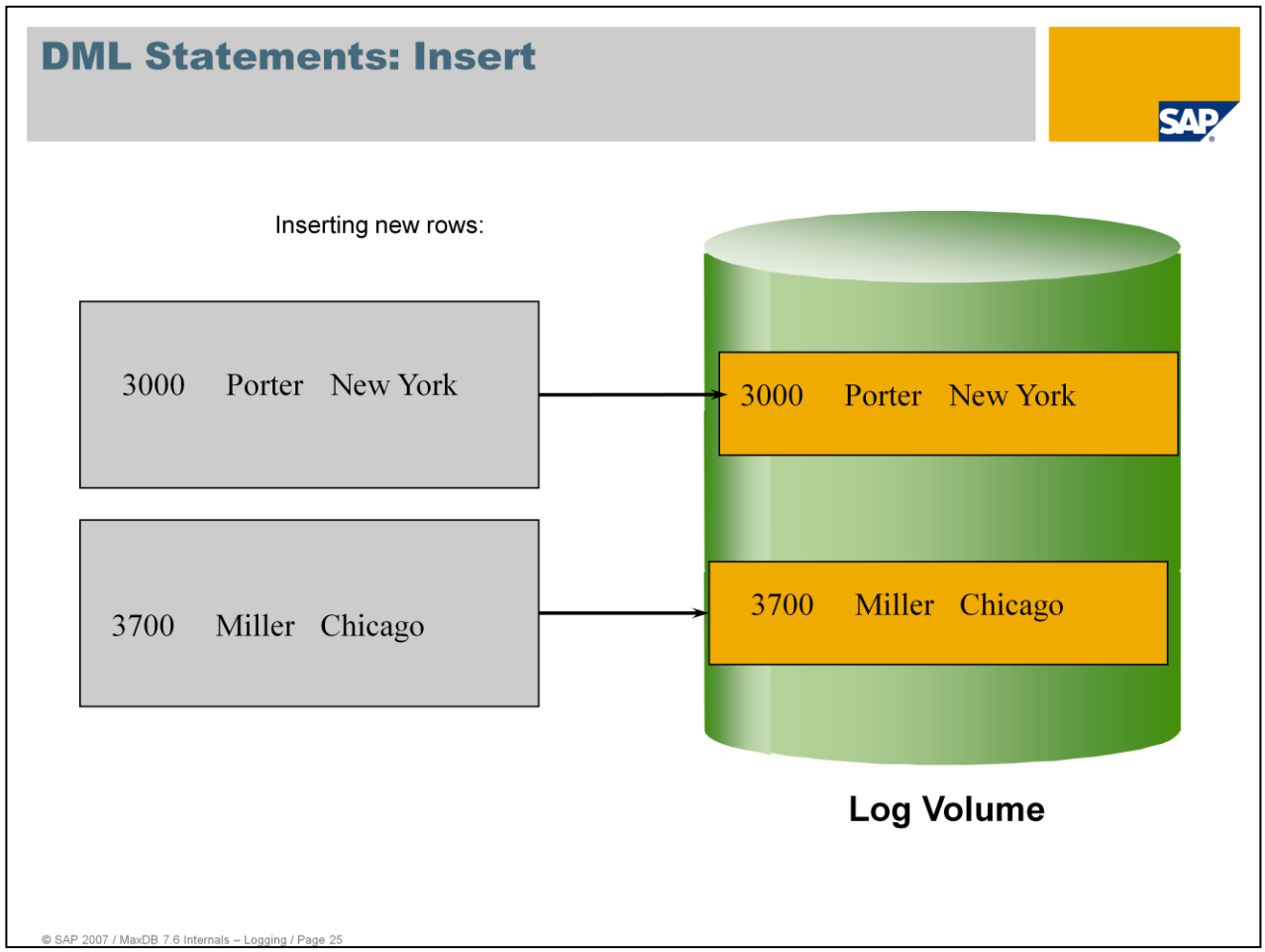
Creation and deletion of database objects:

If a new object is created in the database, the CREATE statement is written to the log volume as a redo log entry. Moreover, the kernel writes an undo log entry to the undo log file of the corresponding transaction.

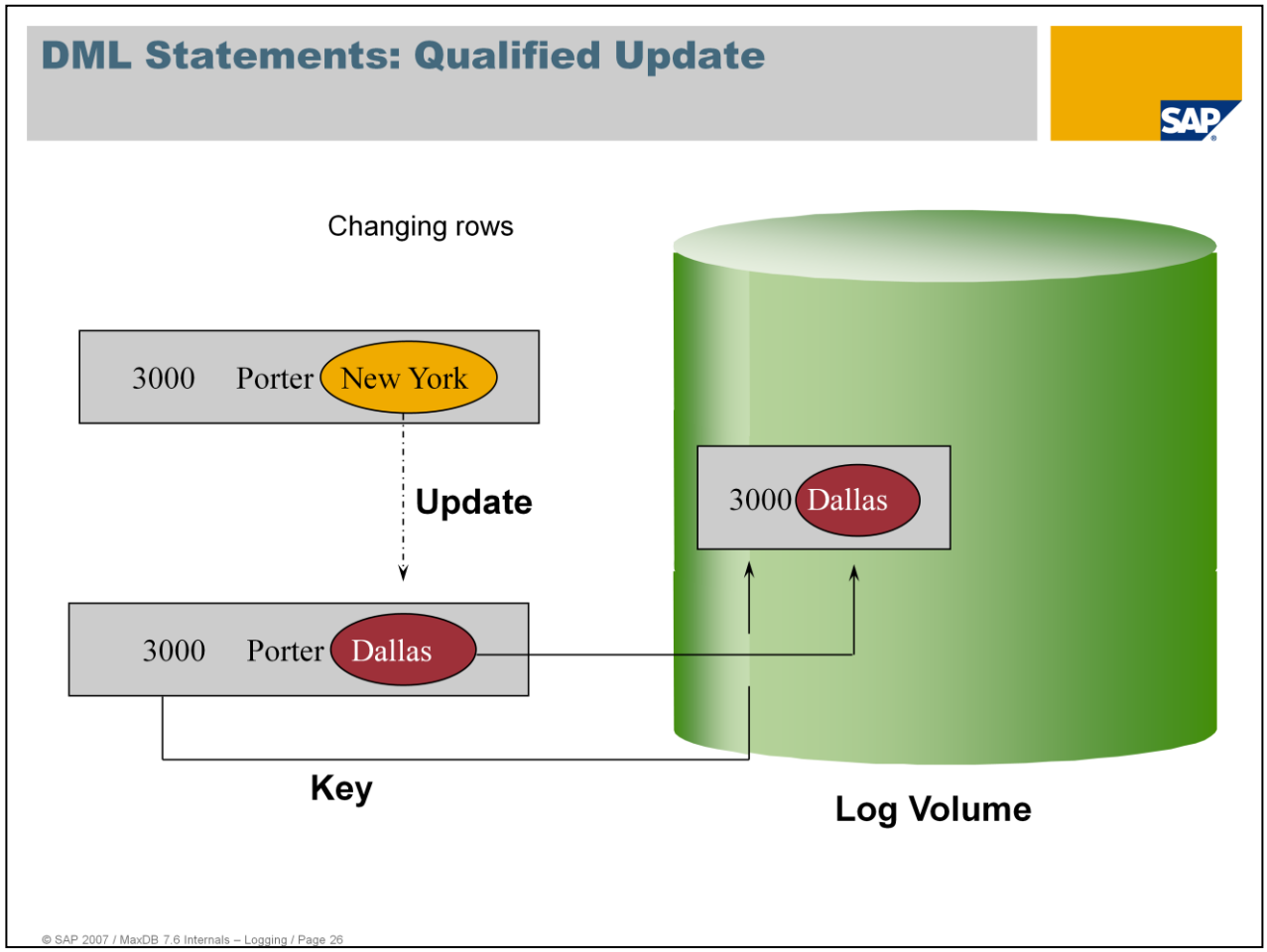
If an index is created in the database, this statement, too, is written to the log. The creation of an index also triggers a savepoint.

If an index is created in the database, this statement, too, is written to the log.

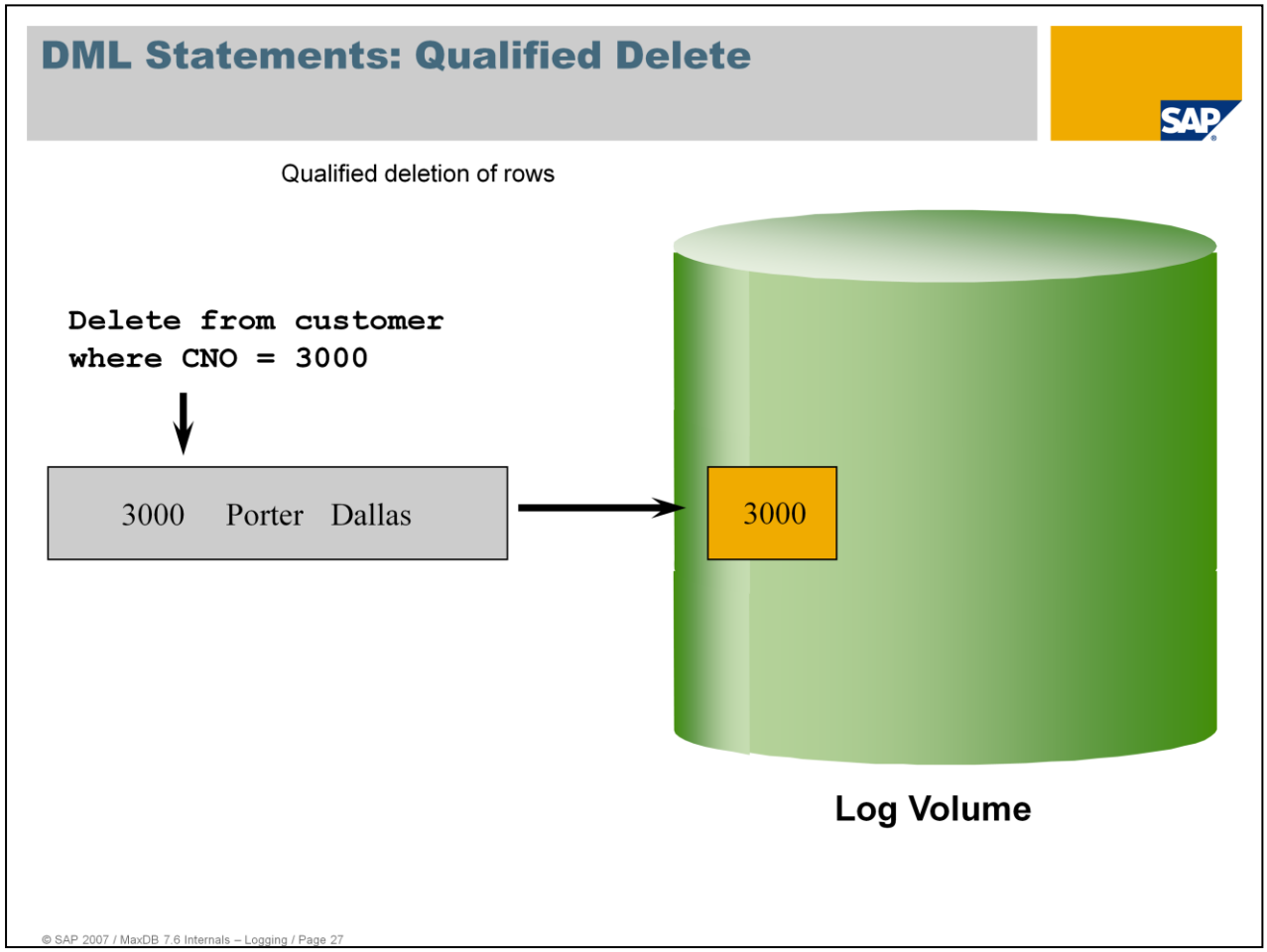
The statements are not written to the log in plain text, but rather in the form of stack code, which is generated by the SQL manager.



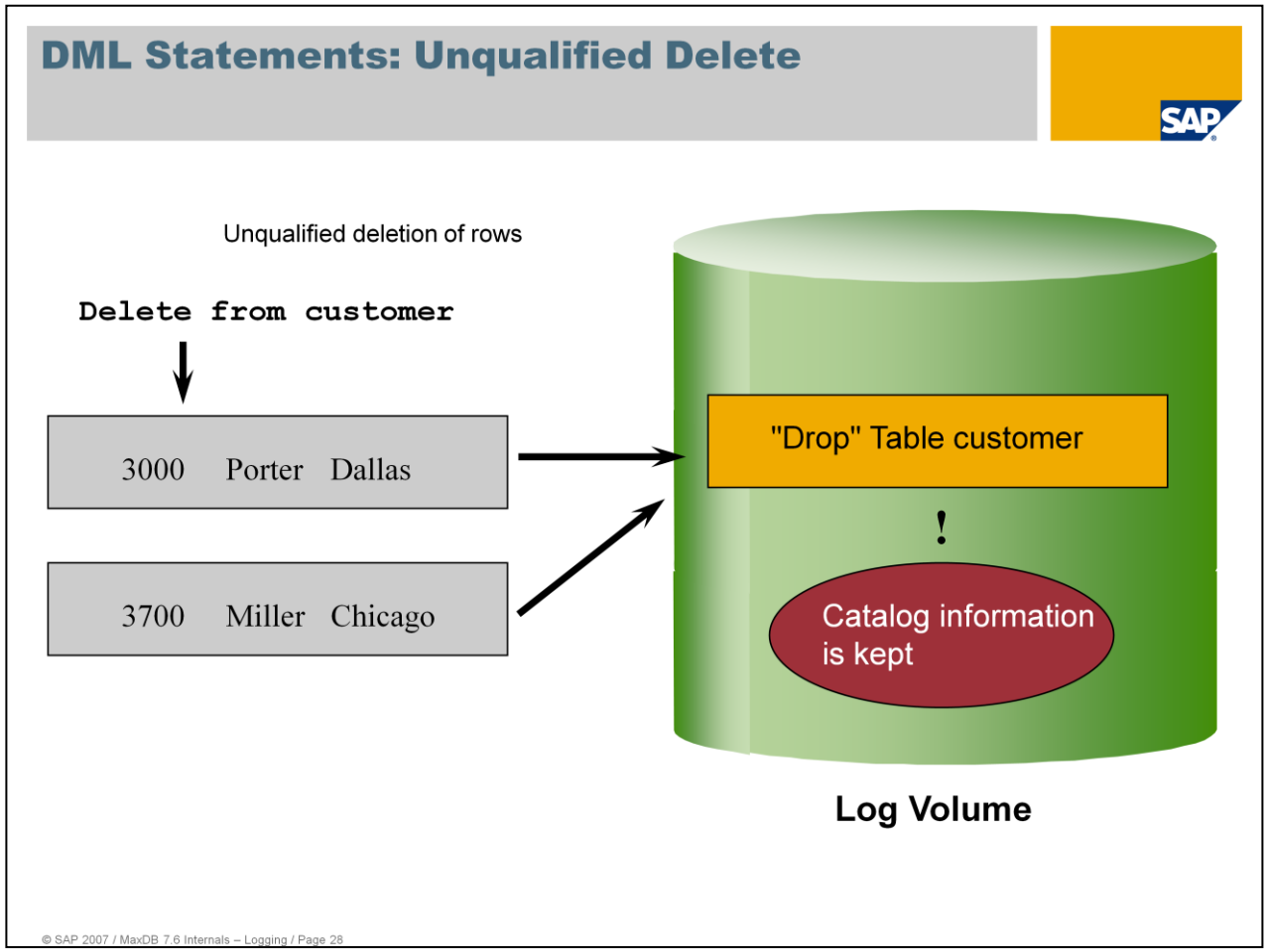
If new records are inserted into a table, the entire new record is written in the log.



If existing records are changed in the database, the key of the record to be changed is written in the log with the changed field values. For variable-length fields, the length byte of the field is also stored.



For qualified deletion of records, only the primary keys of and the table ID of the records to be deleted are written in the log. The other field values are not needed for a redo.



For unqualified deletion, the records to be deleted are not written in the log.

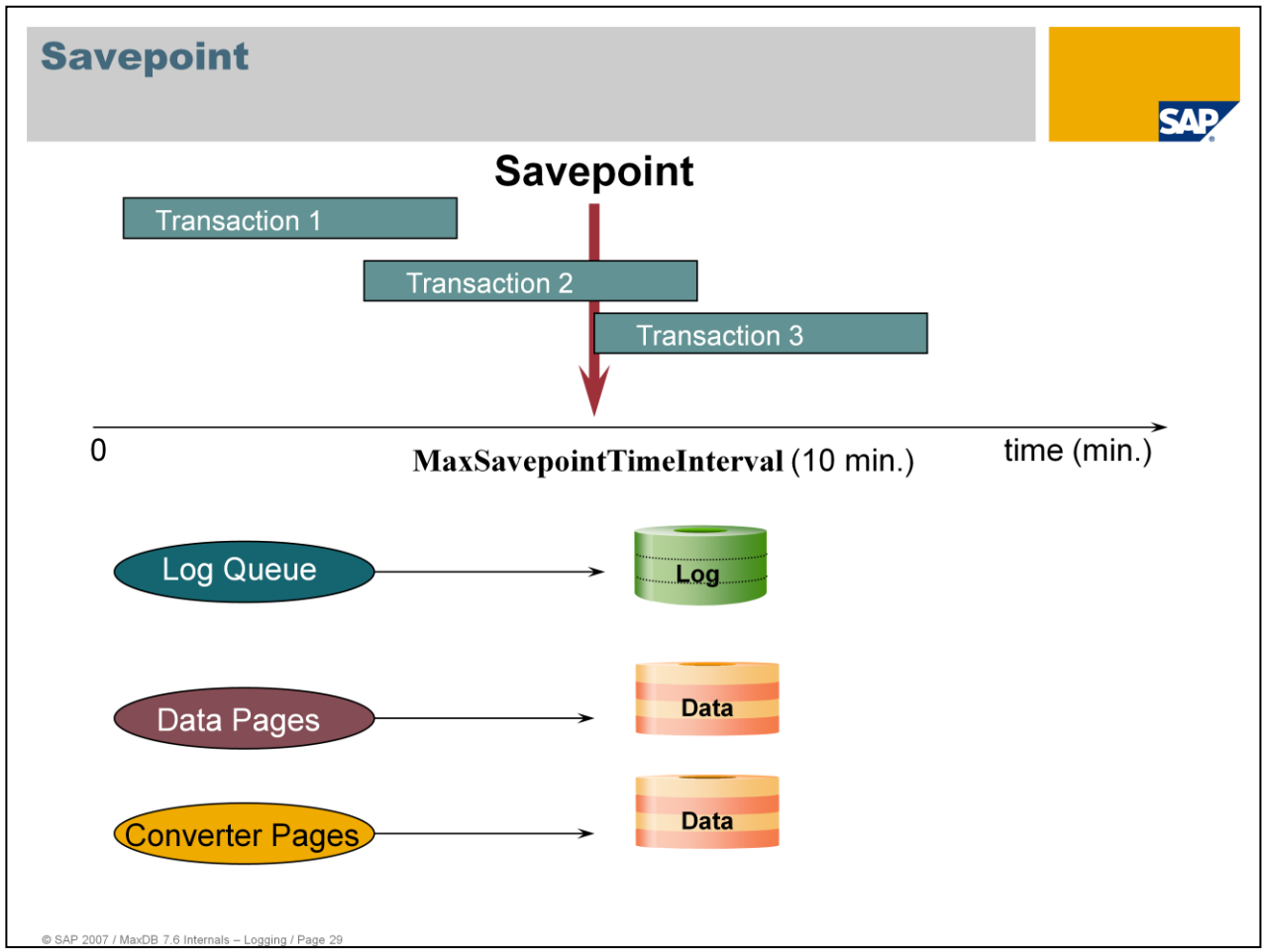
The deletion of all the records in a table is handled like a DROP Table, that is, a ‘DROP TABLE’ statement, which triggers the deletion of the records, is written in the log. But the deletion of the data records does not occur physically; rather, the B* tree of the table is recopied. That way the data can be restored quickly in case of a rollback.

After the commit, the tree is deleted recursively. This action executes a server task in the background. The user does not have to wait until the B* tree is deleted.

The catalog information of the table is retained in a deletion.

Even for unqualified deletion of the table data, the JDBC interface expects the number of deleted records in the return. So MaxDB version 7.5 counts the number of records. If TRUNCATE is used, the database kernel does not determine the number of records to be deleted.

As of Version 7.6, unqualified delete determines the number of records directly from the file directory and therefore does not need to count.



A savepoint speeds up database restarts. Savepoints are written asynchronously.

Savepoints are executed at regular time intervals, but they are also triggered by certain actions (for example CREATE INDEX) in the database. In the standard, a savepoint is started every 10 minutes. You can configure the time interval between savepoints with the parameter MaxSavepointTimeInterval. With this parameter, you can specify the minimum number of seconds that must elapse after a savepoint before a new savepoint is started. The parameter only affects time-controlled savepoints.

CAUTION: if you increase the value in MaxSavepointTimeInterval, you do reduce the number of savepoints and thus the workload, but this can also slow down the restart after a crash.

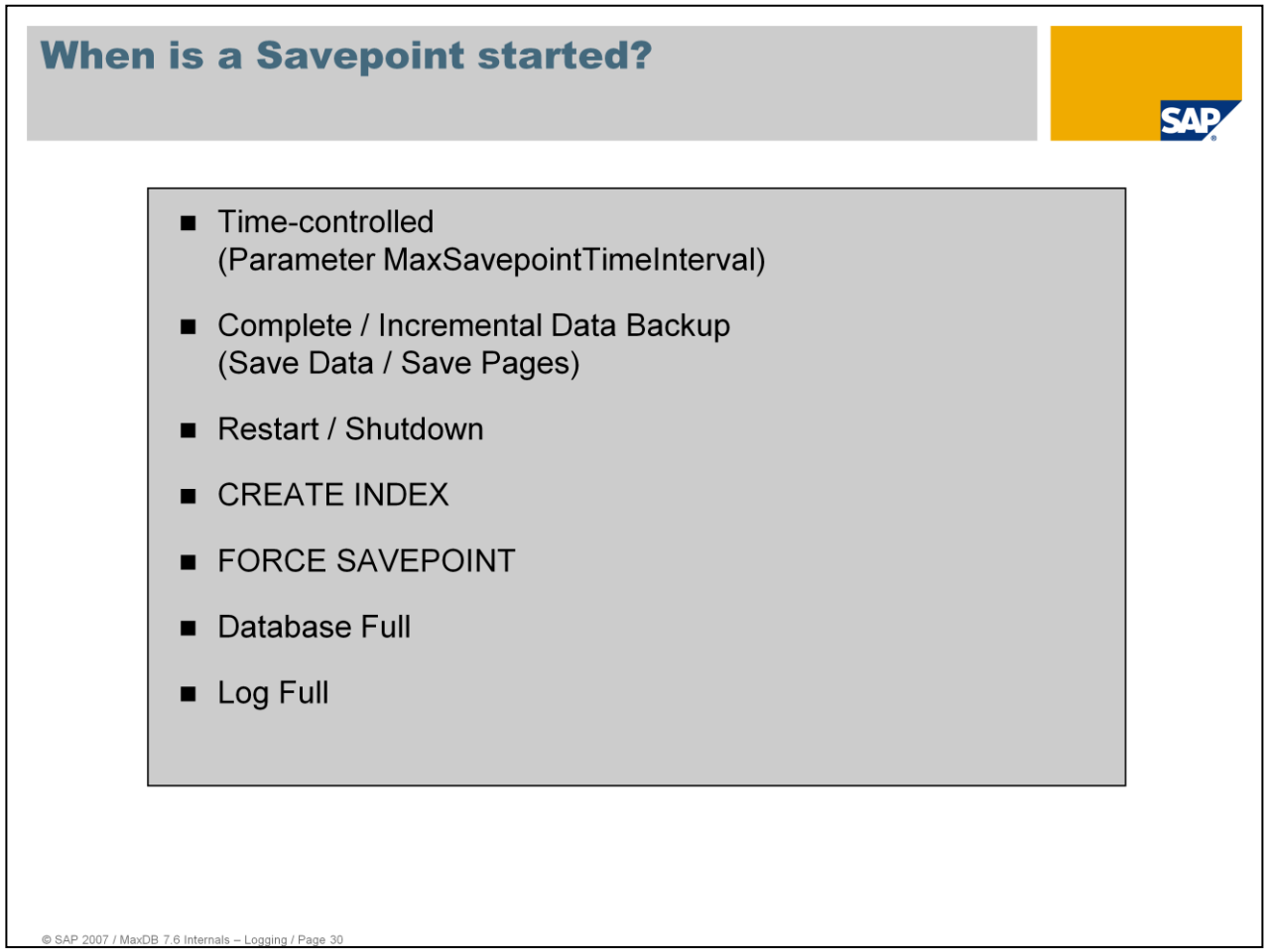
Too-frequent writing of savepoints causes the contents of the caches to be written too often with too little data, which hurts performance (I/O wait).

Time-controlled savepoints are not executed if no changes are made in the database.

A savepoint is also written when a "CREATE INDEX" statement is sent by a transaction.

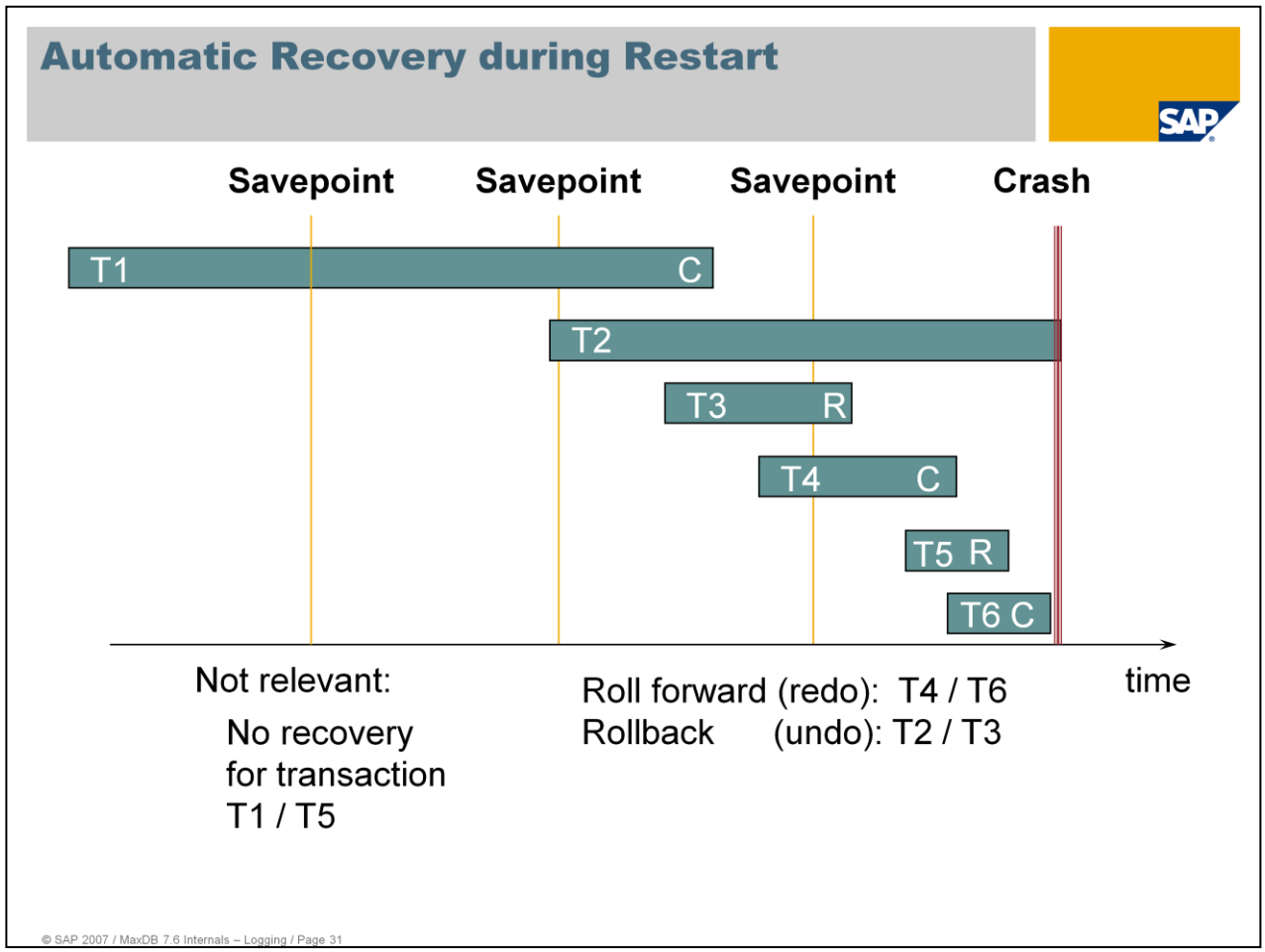
The savepoint triggers the writing of the log queue, data cache and converter cache to the volumes.

In versions smaller than 7.7.03 the parameter MaxSavepointTimeInterval was called _RESTART_TIME.



When is a savepoint requested?

- Savepoints are time-controlled. With the standard setting, a savepoint is started every 10 minutes.
- Data backups first start a savepoint and then back up all data covered by the savepoint. Changes to data during the backup are not included in the backup.
- When the database is shut down, the database kernel waits until a savepoint is carried out and all changed pages in the cache have been written to the volumes.
- Restart is ended with a savepoint.
- A savepoint is started at the end of a CREATE INDEX.
- You can start a savepoint manually with the SQL statement FORCE SAVEPOINT.
- When the kernel sees that the data area or the log is filling up, it starts a savepoint. This minimizes the restart time if the database is stopped with a full data or log area.



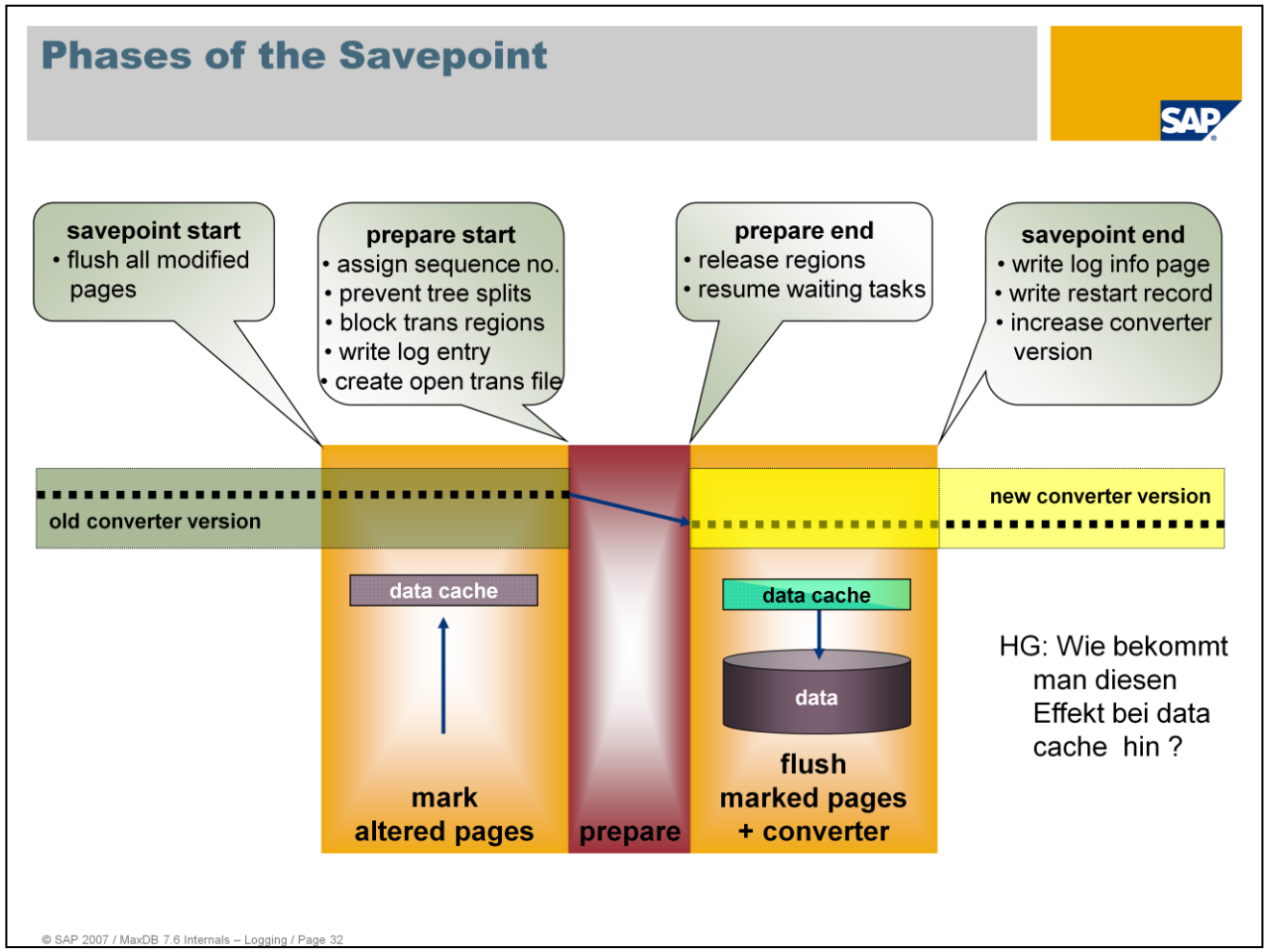
Recovery at restart

The restart implicitly redoes or undoes the transactions; to do this, it goes back to the last savepoint.

All transactions that were not complete at the time of the last savepoint are subject to a redo or undo. Completed transactions that were started after the last savepoint are also subject to a redo.

Example (see slide):

- Transactions 1 and 5 are not relevant for the restart. T1 is completely in the data area. T5 is not in the data area and was reset by a rollback.
- Transactions 2, 3 and 4 were not complete at the time of the last savepoint.
- T2 and T3 are rolled back from the savepoint. The changes of the transactions are read from the undo log files ➲ UNDO .
- T4 is redone starting at the savepoint. The changes are read from the log area ➲ REDO.
- Transaction T6 has to be completely redone. The changes are found in the redo log entries in the log area ➲ REDO.



A savepoint is not a point in time, but covers a span of time. It is divided into three phases.

1. Phase:

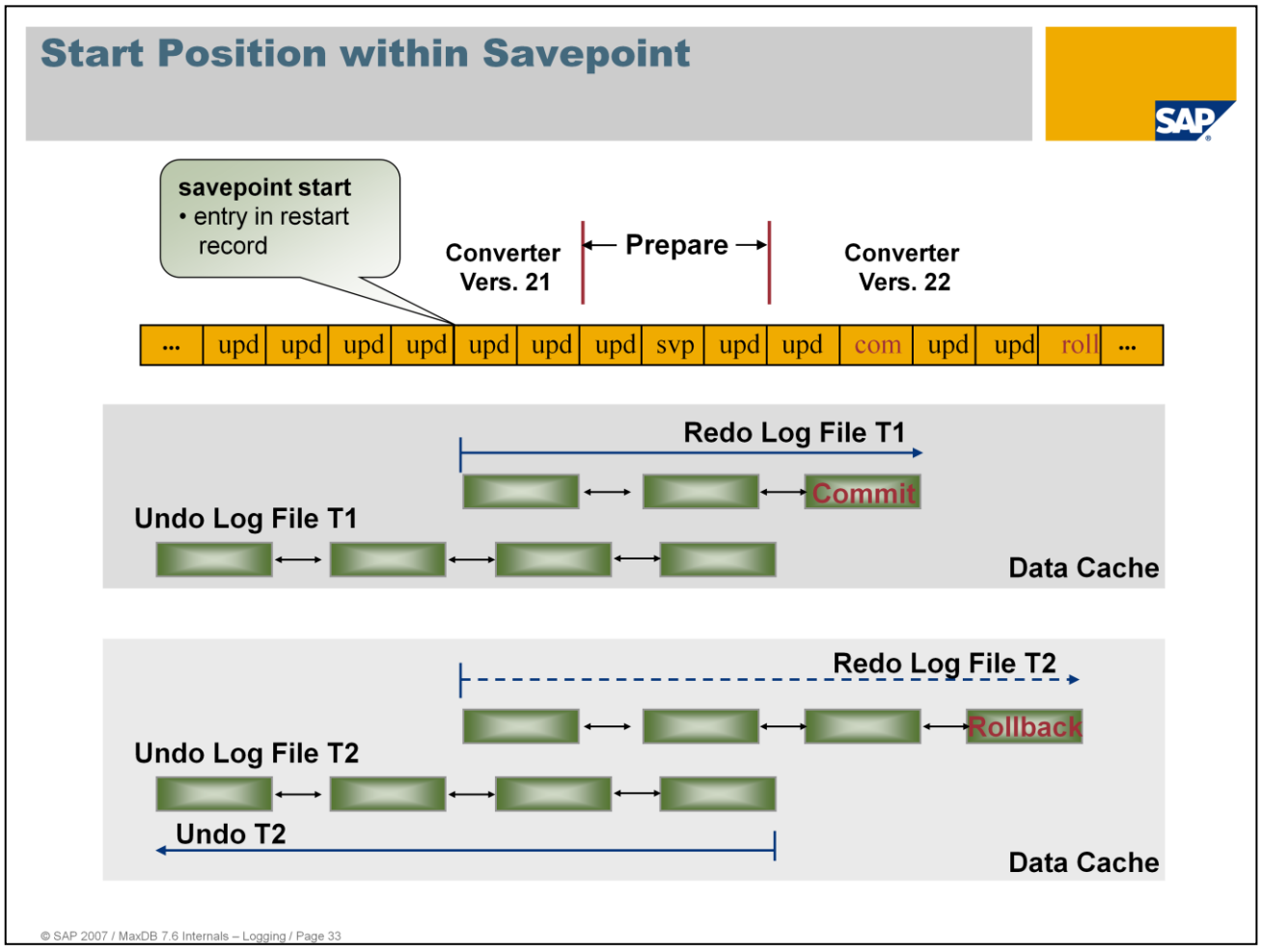
- In this phase, all data pages that have been marked as changed in the cache since the last savepoint and have not yet been written to the data volumes are now written to the data volumes.
- All data pages that are changed in this phase are marked.

2. Phase (Prepare):

- The second phase is known as the prepare phase. At the beginning of the prepare phase,
  - the sequence values that are relevant for the savepoint are determined,
  - B* tree operations are blocked,
  - the transaction regions are reserved,
  - an entry is written in the log,
  - and a page chain (open trans file) with references to the open transactions is generated.
- When the log entry for the savepoint has been written and the open trans file created,
  - all reserved regions are released and
  - all waiting tasks are woken up.
- The duration of the prepare phase is generally in the millisecond range.

3. Phase:

- In the third phase, all the data pages that changed in the first phase are written to the data volumes.
- After that, the converter is written in parallel to the data volumes.
- Then the log info page is written to the log and the restart record is written to the first position of the first data volume. The savepoint is not finished until the restart record has been successfully written.
- Following completion of the savepoint, the converter version (also savepoint version) is counted up by 1. Changes to data pages in the third phase can now be written to the data volumes using the new converter version.



The restart record contains the current log write position at the starting time of the corresponding savepoint. The restart reads the log entries starting at this point.

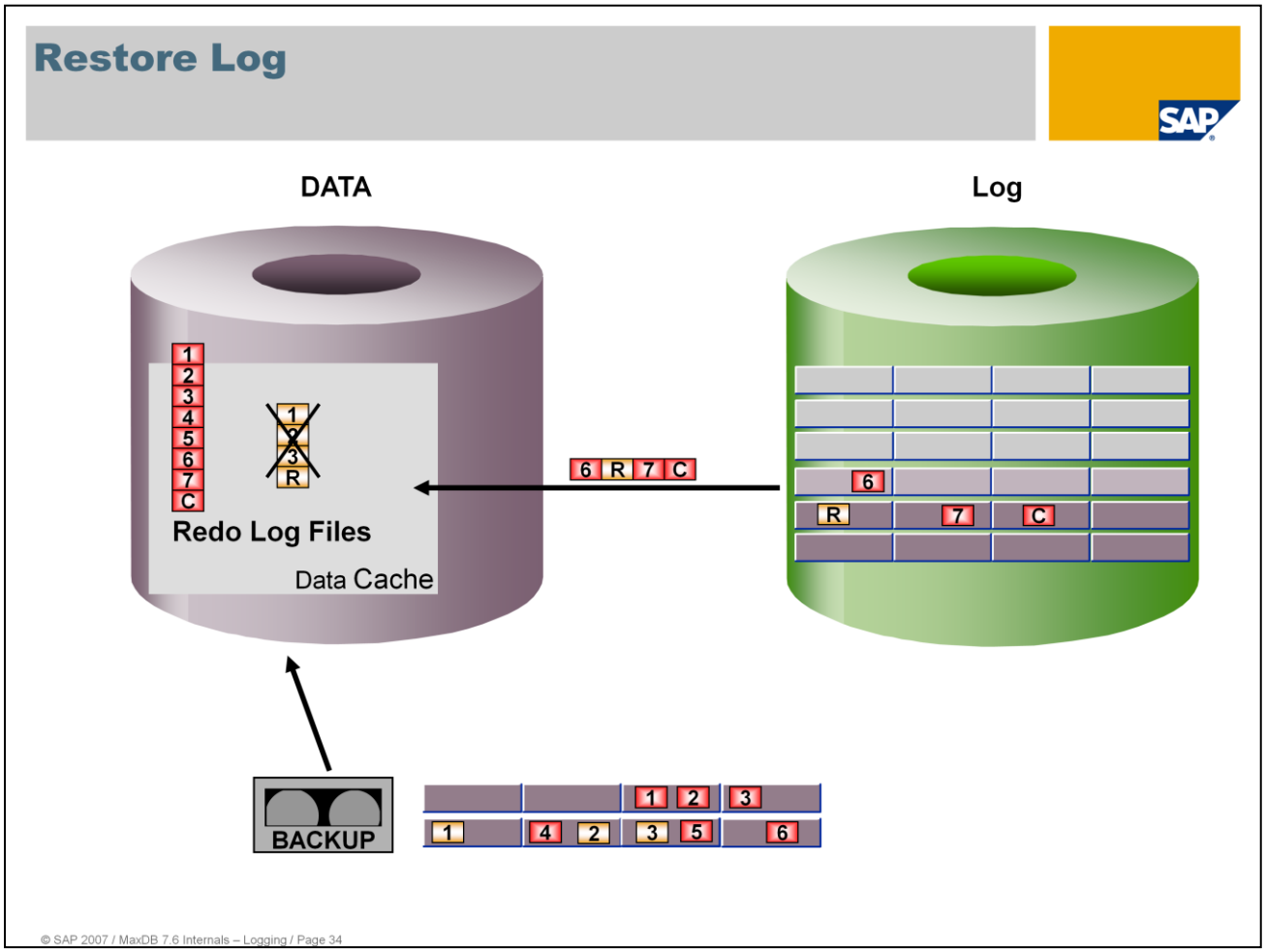
The log reader copies the redo log entries for each transaction to a redo log file in the data area.

If the log reader reads a commit for a transaction, a redo task processes the redo log files and carries out the corresponding changes in the data area. Changes may be carried out that were already written in the data area by the savepoint.

If the log reader reads a rollback for a transaction or reads no end for a transaction, the redo task processes the undo log file of the transaction. The undo log file contains all undo log entries of changes that were written by the savepoint. The redo task redoes the changes made in online mode.

The log reader and redo tasks are listed as server tasks in the task overview.

The savepoint entry in the log is not needed for the redo operation. It is used for security checks.



Restore Log and restart create a redo log file in the data area for each transaction.

Only those redo log entries from the log backups that are no longer in the log volumes are copied.

A transaction is not redone until the redo log file has been completely generated.

Restore Log **without until** specification does not write to the log volume. Thus it is important to back up the log area before the Restore.

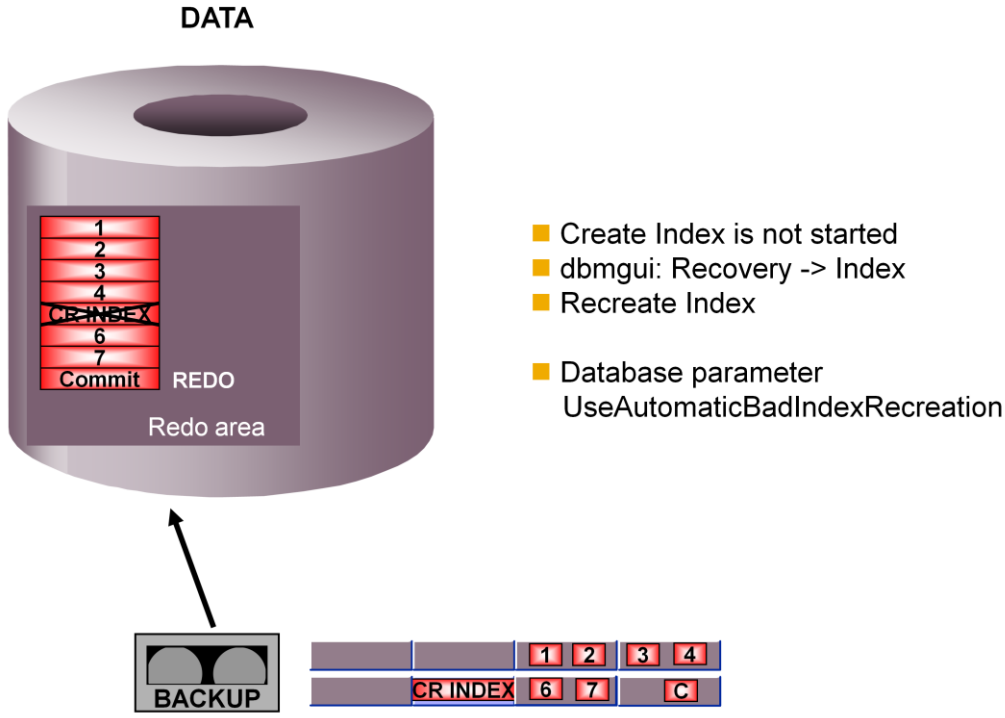
With Restore Log **with until** specification, after all redo log entries have been processed, a savepoint is written and the log is deleted from the until position onwards. This interrupts the log history. Therefore, in a production system,

- the log should be backed up before the Restore Log Until and
- a data backup should be generated once the Restore is completed.

Savepoints are written during the Restore Log. With these savepoints, the generated log files are also written to the log volumes. Following crashes, the last savepoint is the restart position. When you restart, start the Restore with the last log backup that was read.

The DBM command db_restartinfo in the ADMIN state displays the log page on which the Restore Log starts.

Restore Log: Create Index



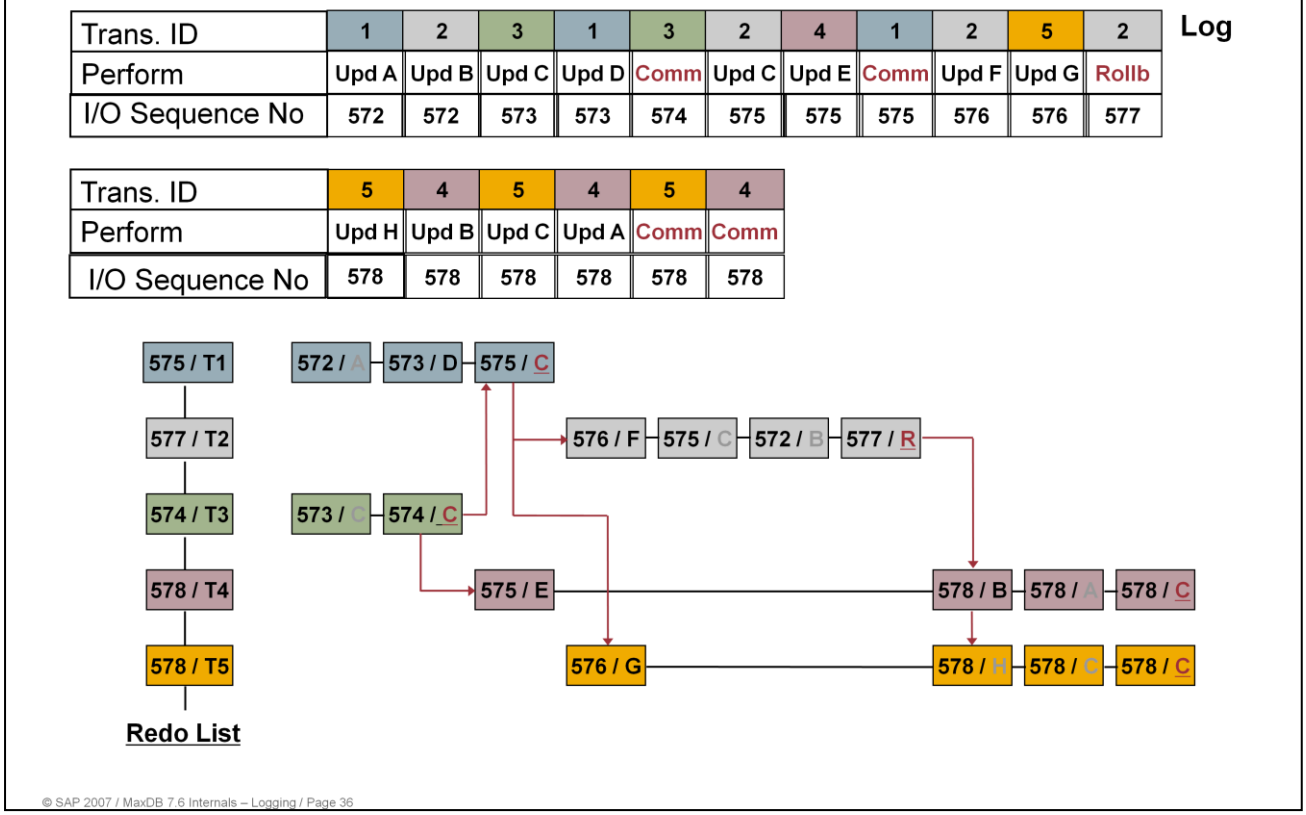
CREATE INDEX statements are not repeated in the case of a redo.

That can speed up RESTORE LOG considerably.

The indexes are recreated with RECREATE INDEX when the database is in the warm state.

When the parameter UseAutomaticBadIndexRecreation (AUTO_RECREATE_BAD_INDEXES) is set to YES, the corresponding indexes are automatically generated at the end of the restart/redo. Users can log on to the database when the indexes have been created.

Parallel Redo during Restart



Whenever possible, MaxDB redoes/undoes log entries in parallel. The lock mechanism cannot be used for synchronization as is possible in the online operational state. Transactions may not "overtake" each other; that is, they must be processed in the same order as in the online operational state.

The log I/O sequence of the transaction closures (commit/rollback) is used for synchronization when redoing or undoing transactions.

The log reader creates an entry in the redo list when a redo log file is fully generated. This entry contains the transaction ID and the I/O sequence of the transaction closure.

Multiple redo tasks then process the redo log files in parallel.

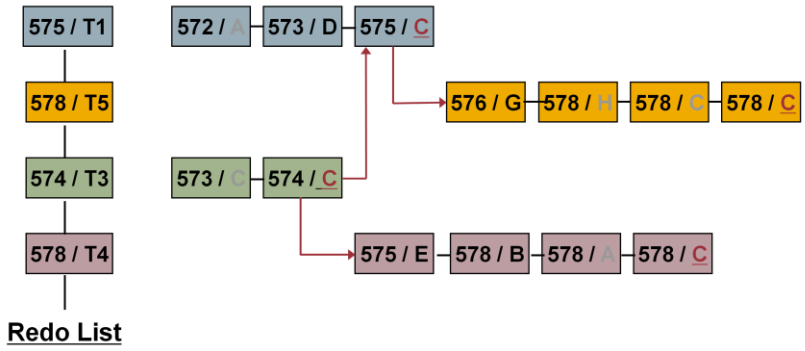
The redo task only redoes a log entry once all transaction closures with a smaller I/O sequence numbers have been processed.

Parallel Redo during Restore Log

Trans. ID	1	2	3	1	3	2	4	1	2	5	2
Perform	Upd A	Upd B	Upd C	Upd D	Comm	Upd C	Upd E	Comm	Upd F	Upd G	Rollb
I/O Sequence No	572	572	573	573	574	575	575	575	576	576	577

Log

Trans. ID	5	4	5	4	5	4
Perform	Upd H	Upd B	Upd C	Upd A	Comm	Comm
I/O Sequence No	578	578	578	578	578	578



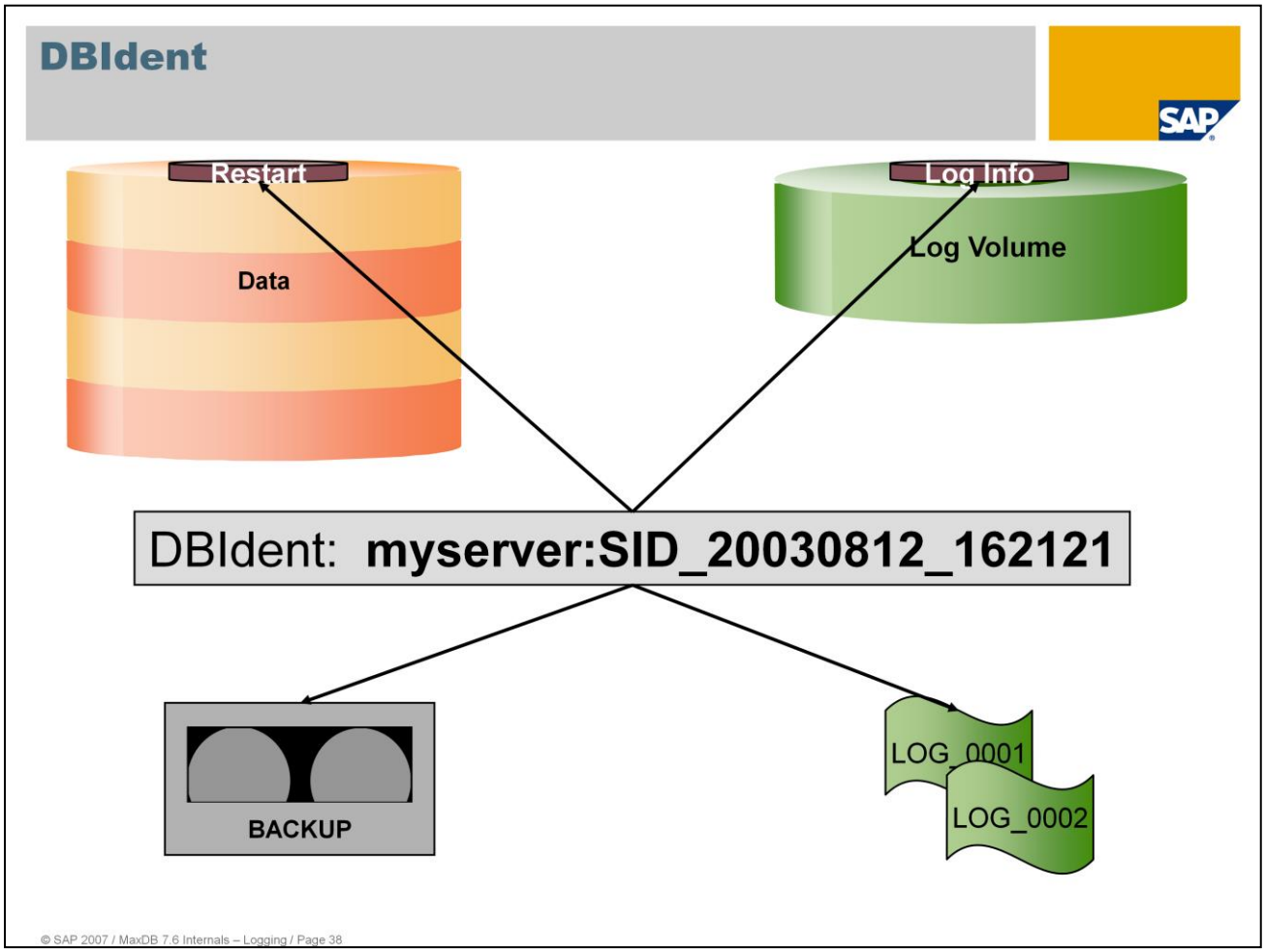
In this example, the rollback transaction does not have to be rolled back as it was not written to the data area by a savepoint in the online operational state.

The redo log file is deleted immediately afterwards. The entry does not have to be added to the redo list.

Waiting transactions can continue to work. A further form of parallel processing now takes place.

The locking mechanism used here is different than ordinary locking. Locks are administered via the redo list. The locking mechanism in the online operational state cannot, in this case, guarantee that the transaction will not overtake each other.

There is no restart at the end of the Restore as the transactions are already redone during the process. The database can be switched to the online operational state directly.



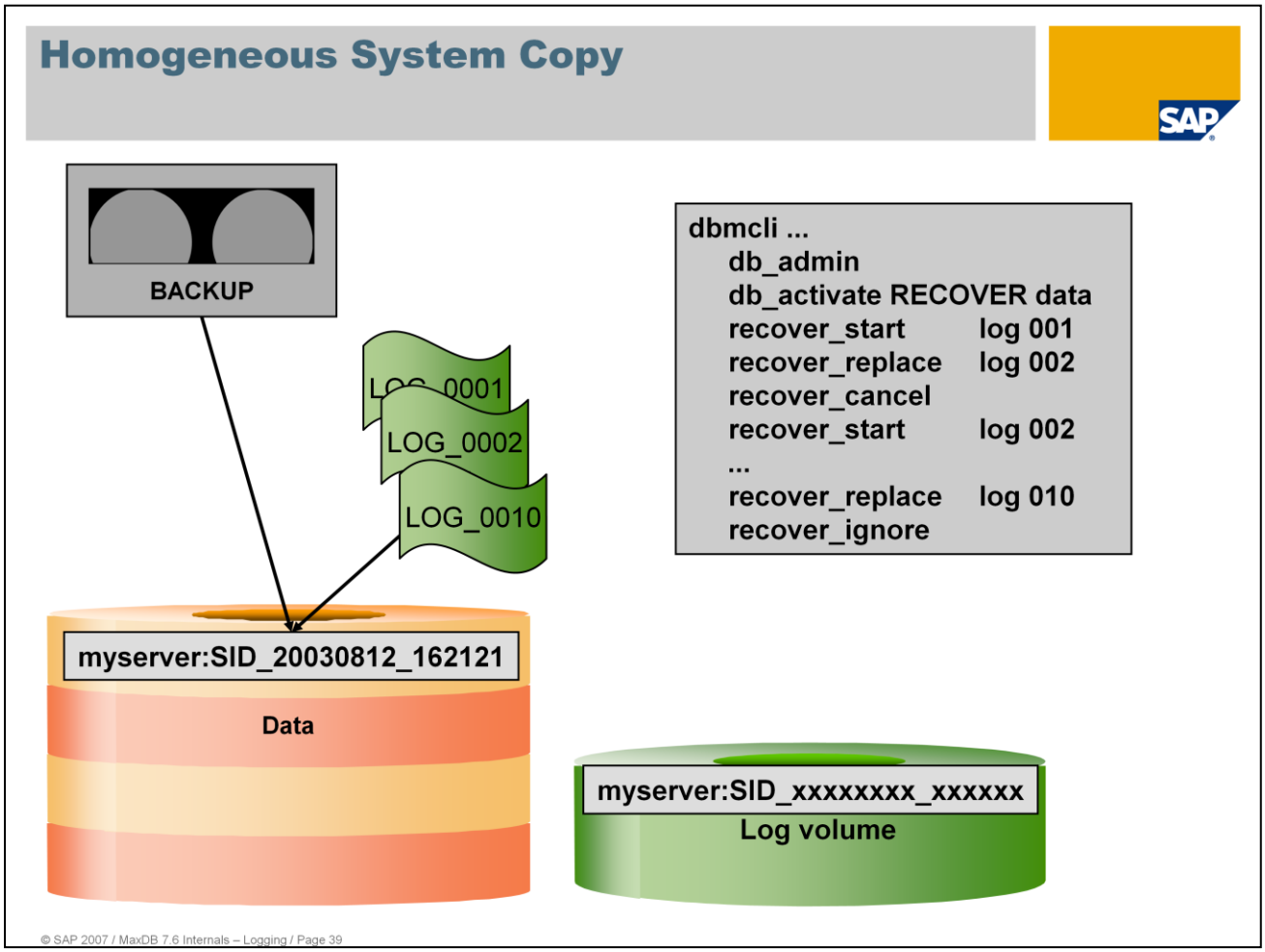
The log and data of an instance must always match. The data of one instance may not, for example, be mixed with the log of another instance during log recovery.

To avoid the occurrence of incompatibility between log and data, the database kernel writes the DBIdent in

- the restart record, which is located in the second block of the first data volume;
- the log info page, which is located in the second block of the first log volume;
- every data backup (Save Data / Save Pages); and
- every log backup.

The DBIdent the names of the database server and the database instance as welll as a time stamp indicating when the backup history began. This time stamp is set with until specification, for example at the first restart and at the end of a Restore Log.

The names of the database server and the instance do not necessarily correspond to the current server and instance names. Database instances can be renamed without the DBIdent being changed (see OSS note 604680). Instances can change database servers in the HA cluster.



Homogeneous system copies are allowed if the processor types of the source and target servers are the same. For more information, see OSS note 129352.

Data backups contain the undo log entries of all open transactions. They can import a data backup into another instance per Restore and execute a restart. The restart ensures the consistency of the database by undoing the open transactions.

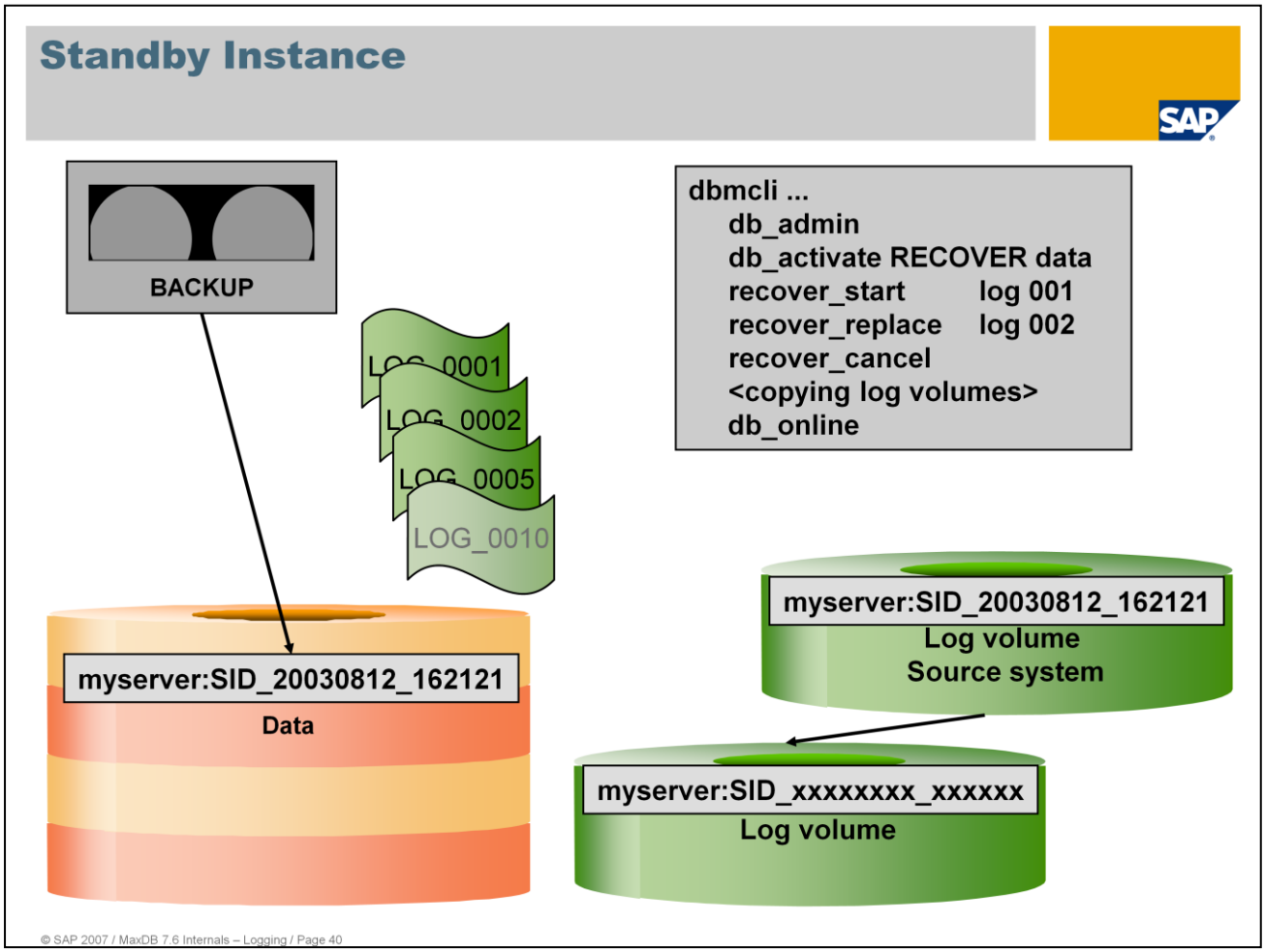
Note that in this case the restart only works if the DBIdent in the log matches the DBIdent in the data area or if the log was initialized with "db_activate RECOVER“. The "db_activate RECOVER“ statement does not fill the DBIdent. The restart is what sets the DBIdent in the data and log areas.

Restore Data enters the original DBIdent in the data area. This makes it possible to import log entries and log backups from the source system. The Restore Log process can be interrupted and resumed. In this case, the dbmcli command recover_ignore executes the restart and sets the new DBIdent.

If log entries from the source instance are redone in the target instance, the database version of both instances must be the same at build level.

As of version 7.5, a log backup that was created online also contains the last incompletely-filled log page, that is, all redo log entries that were generated since the last log backup.

Caution: in older versions, this log page was only backed up once it was completely full. In older versions, if you require a particular starting point, set the source instance to the ADMIN operational state to create the last log backup for the system copy. This can be avoided by performing some "dummy" updates on a table. The dummy updates would fill the last log page.

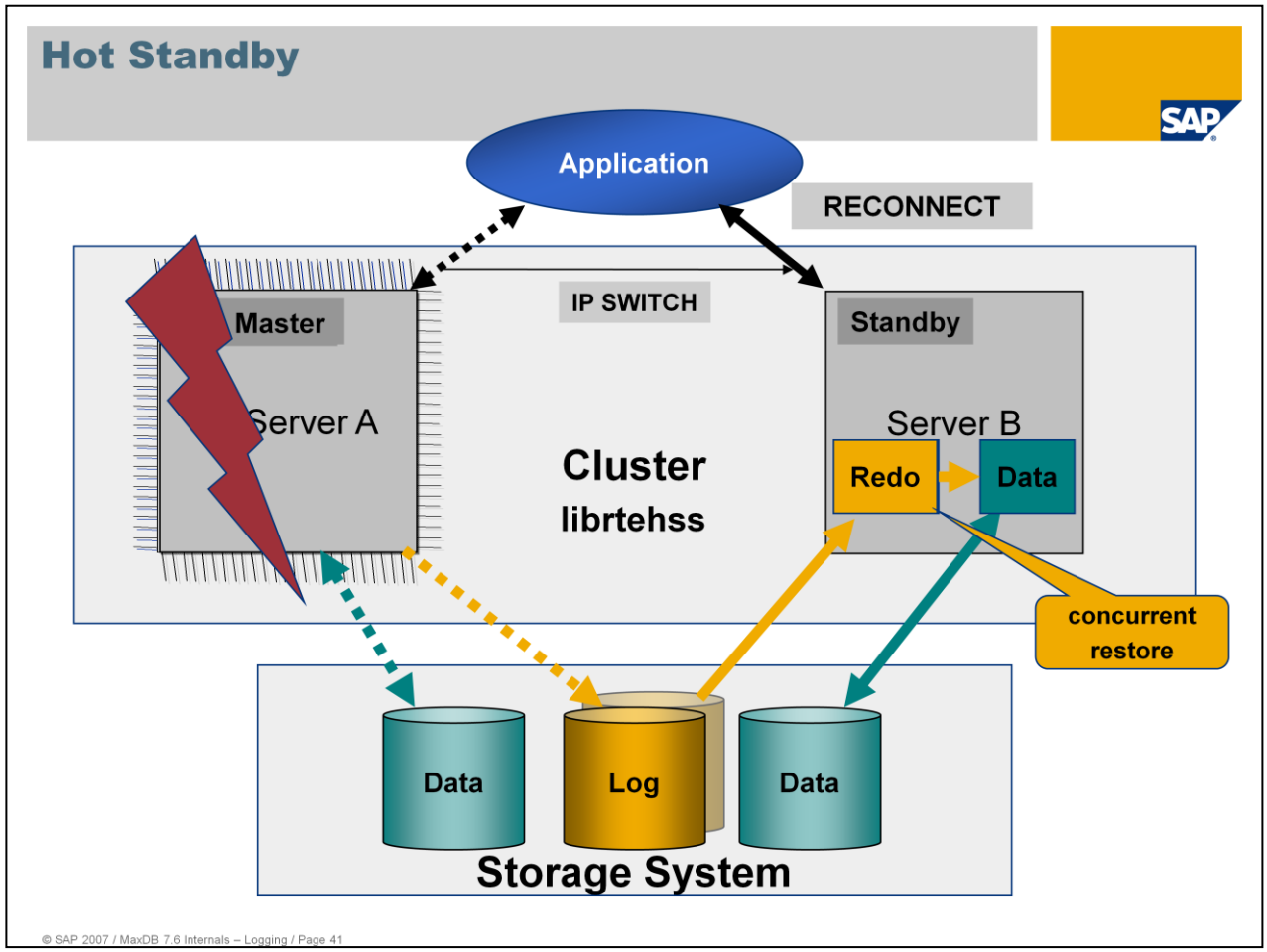


MaxDB also offers the possibility of operating so-called standby databases. A standby instance is supplied with a data backup from an original instance. Subsequently the log backups from the original instance are imported into the standby instance.

If the log volumes of the original instance are still available before starting the standby instance, you can copy these log volumes to the log volumes of the standby instance. Then continue with recover_start. Once the redo log entries can be read from the log volumes, the log reader no longer reads from the log backups. The database is automatically set to the ONLINE operational state.

The DBIdent is retained. The log backup history is not interrupted.

Caution: The "db_activate RECOVER“ should not be used for a normal recovery. "db_activate RECOVER“ is used in the creation of a new instance. For a normal recovery, "db_activate RECOVERY" would unnecessarily prolong the recovery time. The current log in the log area is lost.



As of version 7.5, MaxDB supports hot standby environments.

A production instance (master instance) works with its data area and writes changes to the data to the log area.

When the standby instance is initialized, the data area of the standby instance is supplied with a snapshot of the data of the current master instance. In standby mode, the standby instance reads from the common log area and redoes any log entries. The standby instance only has read authorization for the log area. The master instance informs the standby instance at short intervals how far the standby instance can read from the log area.

If the master instance fails, the cluster agent executes a takeover by the standby instance. It sets the standby instance to the online operational state. The library "librtehss“ enables the standby instance to write in the log from now on. The takeover by the standby instance takes only a few seconds; only a few redo log entries need to be redone. All necessary resources (for example caches) are already in operation and do not need to be requested when the takeover occurs. The data pages that were changed during the recovery are already in the cache.

Prerequisites for hot standby:

- Use of a storage system that supports I/O consistent snapshots or Split Mirror. SAP OSS note 371247 described this requirement.
- The storage system allows one instance to write in the log and other instances to read from it. "librtehss, which is supplied by the manufacturer of the storage system, can switch the read/write mode.
- The library "librtehss“ supplies the standby instance with the data from the master instance when the former is initialized.
- A cluster agent monitors the production database instance and starts a takeover if it fails.

MaxDB supports multiple instances.

Previously, Hot Standby was implemented by

- IBM for IBM ESS, DS8000 and SVC. For further information see http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100442.
- EMC implemented for Symmetrix with Timefinder Clone early 2006).

Thank you!