

Database Tuning

Recovery Tuning

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg



<http://dbresearch.uni-salzburg.at>

Sommersemester 2019

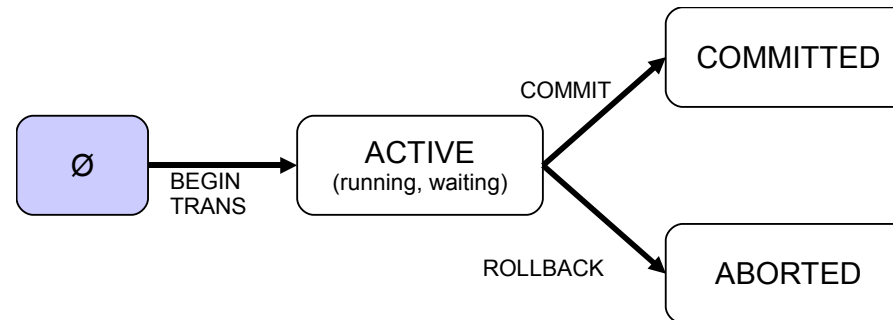
Version August 8, 2019

Adapted from “Database Tuning” by Dennis Shasha and Philippe Bonnet.

Outline

- 1 Recovery Tuning
 - Logging and Recovery
 - Tuning the Recovery Subsystem

Atomicity and Durability in Case of Failure



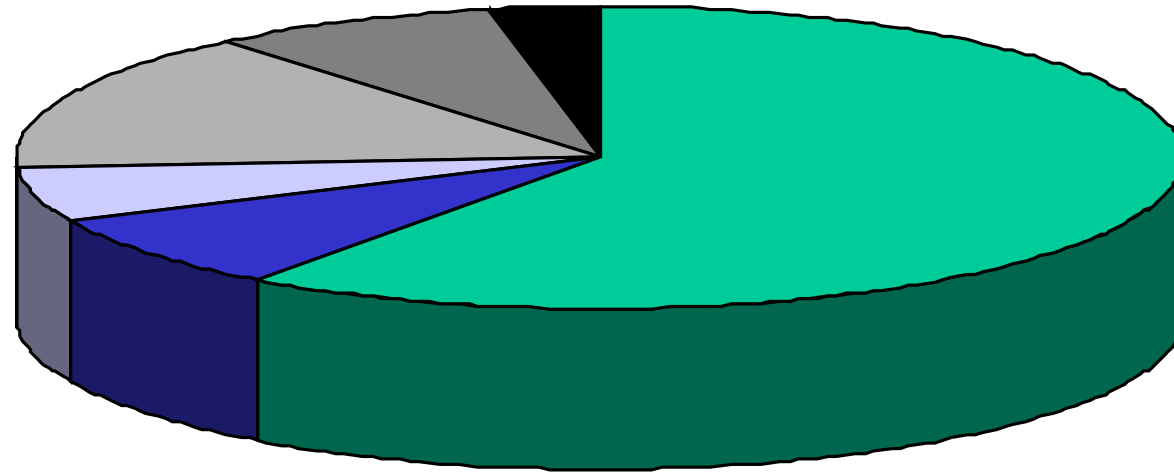
States of a Transaction

- **Durability:** After a transactions commits, changes to the database persist even in the case of system failure.
- **Atomicity:** after failure, reconstruct database such that
 - changes of all committed transactions are reflected
 - effects of non-committed and aborted transactions are eliminated
- **Recovery subsystem:** Guarantee atomicity & durability in failure case.

Failure Types

- **Software:**
 - 99% are Heisenbugs (non-reproducible, due to timing or overload)
 - Heisenbugs do not appear if system is restarted
 - example: error due to isolation level that was chosen too low
- **Hardware:** failure in physical device
 - CPU, RAM, disk, network
 - fail-stop: device stops when failure occurs, e.g., CPU
- **Maintenance:** problem during system repair or maintenance
 - examples: recover from failure, backup
- **Operations:** regular operations
 - regular system administration and configuration
 - user operations
- **Environment:** factors outside the computer system
 - examples: fire in the machine room (Credit Lyonnais, 1996), 9/11

Failure Probability



- **Software**
- **Hardware**
- **Maintenance**
- **Operations**
- **Environment**
- **Unknown**

From J.Gray and A.Reuters
Transaction Processing: Concepts
and Techniques

Which Failures Can Database Systems Tolerate?

- Some software failures:
 - crashing client
 - crashing operating system
 - some server errors
- Hardware failure:
 - CPU fail-stop and erasure of main memory
 - single disk fail-stop (if enough redundant disks are available)
- Environment: Power outage
- Backups still important:
 - recovery system does not substitute backups
 - backups required for failures not covered by recovery system
 - example: accidental deletions, natural disaster

Durability

- **Durability** in databases:
 - goal: make changes permanent before sending commit to client
 - implementation: store transaction data on stable storage
- **Stable storage**: immune to failure (only approximated in practice)
 - durable media, e.g., disks, tapes, battery-backed RAM
 - replication on several units (redundant disks to survive disk failure)
- **Problems**:
 - non-durable buffers in some system layer
 - partial disk writes

How To Deal with Non-Durable Buffers?

- **Non-durable buffer** in some system layer:
 - database tells system to write a disk page
 - but disk page remains in some non-durable buffer
- **Operating system buffer**:
 - write operations are buffered
 - `fsync` flushes all pages of a given file – OK
- **Disk controller cache**:
 - common in RAID controllers
 - battery-backed cache – OK
 - other caches may lead to inconsistencies in case of failure
- **Disk cache**: switch off for log disk (critical!)
 - `hdparm -I /dev/sda` shows meta data of disk `/dev/sda`
 - `hdparm -W 0 /dev/sda` switches disk buffer off

How To Deal with Partial Disk Writes?

- **Partial disk writes:**
 - database writes disk page which consists of several sectors
e.g., 8kB page consists of 16 sectors (512B each)
 - power failure during write: page may be only partially written
 - leads to inconsistent database state
- **Disk controller:** battery backed cache
 - data in cache is written at restart after power outage
 - consistent state is restored
- **Operating system:** file system
 - file system that prevents partial writes, e.g., Raiser 4
- **Database:** e.g., `full_page_writes` in PostgreSQL
 - before-image of page is stored before updating it
 - recovery: partially written page is restored and update is repeated

Guaranteeing Atomicity

1. **Before images:** state at transaction start
 - used to undo the effects of a uncommitted transaction
 - before image must remain on stable storage until commit
2. **After images:** state at transaction end
 - used to install effects of transaction after commit
 - after image must be written to stable storage before commit

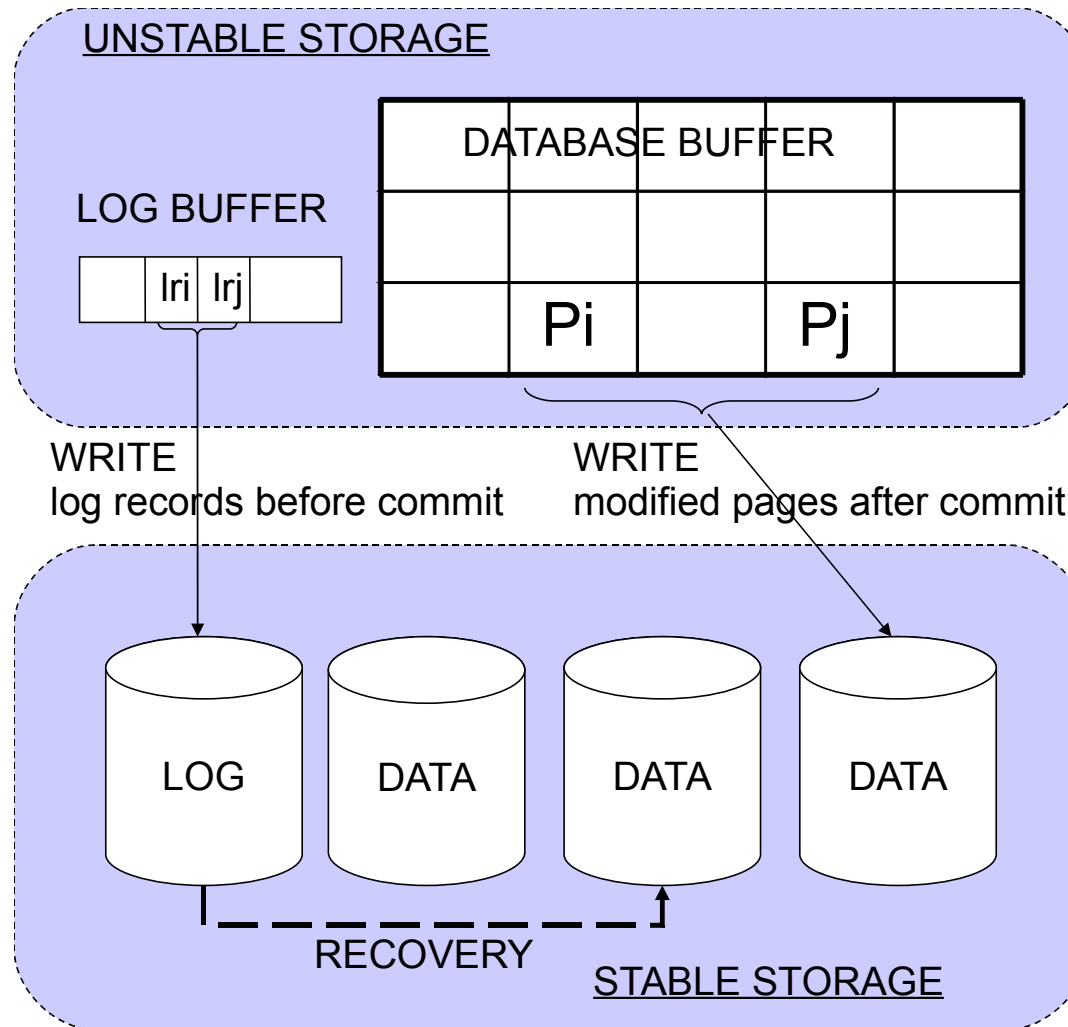
Concepts

- **Data files:** tables, indexes
- **Log file:** stores before and after images
- **Database buffer:** contains pages that transactions modify
- **Dirty page:** buffer page with uncommitted changes

Write-Ahead Logging

- WAL commit:
 - write after images to **log file** before transaction commits
 - **data files** can be updated later (after commit)
- WAL abort:
 - variant 1: explicitly store before image in log
 - variant 2: use data file as a before image
 - only in variant 1 it is safe to write dirty pages to the data file
 - dirty pages are typically written when the database buffer is full
- **Example:** WAL for a transaction T that modifies pages P_i and P_j
 - pages P_i and P_j are loaded to the database buffer
 - transaction T modifies the pages P_i and P_j
 - database generates log records lr_i and lr_j for the modifications
 - database writes log records to stable storage before committing
 - modified pages are written to data file after transaction T commits

Write-Ahead Logging



Logging Variants

- **Logging granularity:** what does a log record store?
 - page-level logging
 - byte-level logging (log partial pages)
 - record-level logging
- **Logical logging:** log operation and argument that caused update
 - e.g., operation: `insert into employee,`
argument: `(103-4403-33,Brown)`
 - saves disk space
 - implemented in DB2

Logging Guarantee

- Guarantee by logging algorithms:

current database state = current state of data files + log

- Current database state:
 - reflects all committed transactions
- Current state of data file:
 - reflects only committed transactions physically in data file
 - some transactions may be committed and stored in the log, but not yet written to the database

Checkpoint and Dump

- **Checkpoint:** force data files to reflect current database state
 - write all committed changes to data file
 - committed changes may be in database buffer or log
- **When do checkpoints happen?**
 - at regular intervals (tuning parameter)
 - log is full (Oracle)
 - explicit SQL command
- **Dump:** transaction-consistent database state
 - entire database including changes of all committed transactions
 - recovery guarantee:

current database state = database dump + log (after dump)

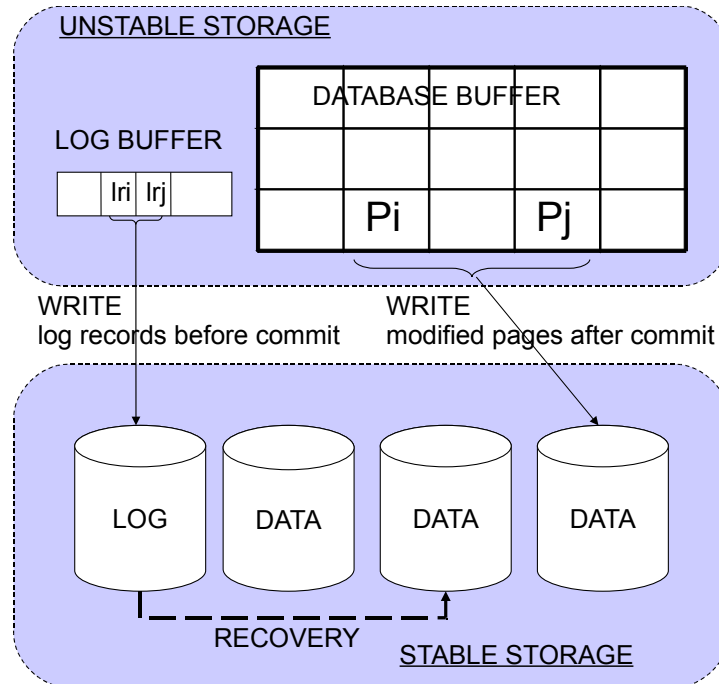
Recovering after Main Memory and Disk Failure

- **Main memory failure:** database buffer is lost
 - log needs to be considered only starting after last checkpoint
 - all committed changes before checkpoint are already in data file
- **Data disk failure:** (disk with log is still OK)
 - database dump required
 - log after database dump needs to be considered
 - checkpoints irrelevant
- **Log disk failure:** disaster!
 - committed transactions after last checkpoint get lost
 - database may be inconsistent - last consistent state is last dump
 - to prevent disaster, replicate disk with log
 - make sure to avoid risk of non-durable buffers and partial writes

Outline

- 1 Recovery Tuning
 - Logging and Recovery
 - Tuning the Recovery Subsystem

Tuning Activities

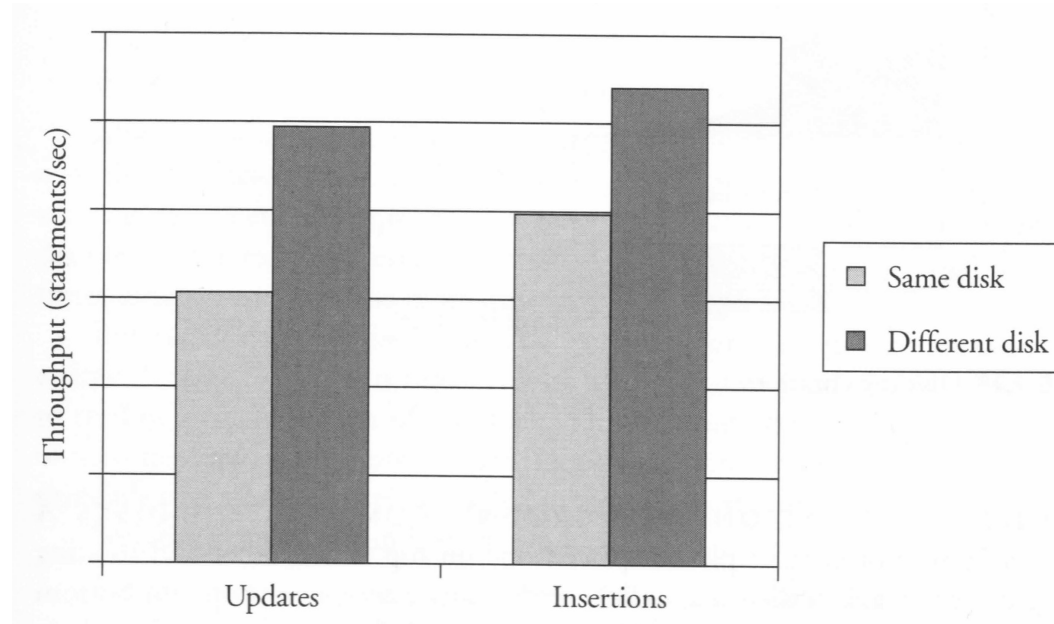


1. Log on separate disk
2. Log buffer tuning: group commit
3. Log buffer tuning: trading in durability
4. Tuning data writes (checkpoints)

1. Log on Separate Disk

- Update transaction must write to the log, i.e., to the disk
- If log and data files share disk, disk seeks are required.
- Separate disk for log:
 - sequential writes instead of seeks (10 to 100 times faster)
 - log independent from data files in case of disk failure
 - disk setting can be tailored to log (e.g., switch off buffer)
- PostgreSQL: How to move log to an other disk?
 - log directory: `pg_xlog`
location: `show data_directory;` (needs admin permission)
 - move log directory to log disk and create symbolic link

Experiment – Separate Disk for Log



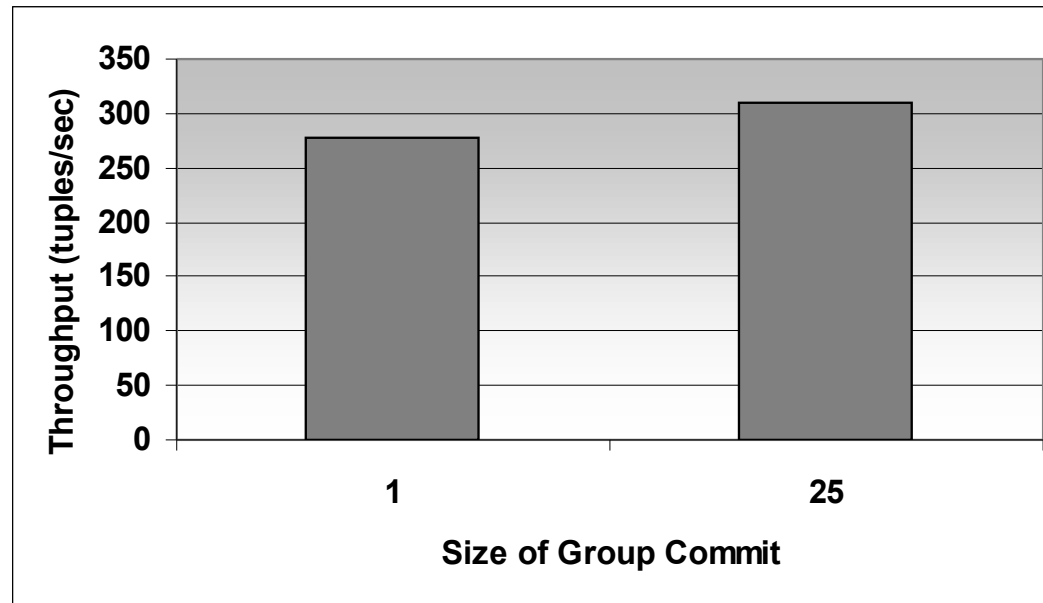
- 300k inserts or update statements.
- Each statement is a separate transaction and forces a write.
- Same disk: data files and log are on the same disk.
- Different disks: log has its own disk.

Oracle 9i on Linux server with internal hard drives (no RAID controller)

2. Group Commit

- Log buffer is flushed to disk before each commit.
- Group commit:
 - commit a group of transactions together
 - only one disk write (flush) for all transactions
- Advantage: higher throughput
- Disadvantages: some transactions must wait before committing
 - locks are held longer (until commit)
 - lower response time for waiting transactions

Group Commit – Experiment



- Increasing the group commit size increases the throughput.

DB2 UDB V7.1 on Windows 2000

WAL Buffer and Group Commit in PostgreSQL

- **WAL buffer**: Write ahead log buffer
 - RAM buffer, z.B. 768kB (`wal_buffers`)
 - all log records are written to this buffer
 - WAL page is flushed at commit or every 200ms (`wal_writer_delay`)
 - data is written to a file called WAL segment (16MB each)
- **`commit_delay`**: (default: 0)
 - time delay between a commit and flushing WAL buffer
 - during waiting period, hopefully other transactions commit
 - if other transaction commits, do group commit
 - if no other transaction commits, waiting time is lost
- **`commit_siblings`**: (default: 5)
 - minimum number of concurrent open transactions for group commit
 - if less transactions are open, `commit_delay` is disabled

3. WAL Tuning: Trading in Durability (PostgreSQL)

- `synchronous_commit`: (default: on)
 - call `fsync` to force operating system to flush disk buffer
 - commit only after `fsync` returns
 - switch off if you do not want to wait for `fsync`
 - parameter can be set for each transaction individually
- Switching off synchronous commit increases performance.
- Worst case: database consistency not in danger
 - system crash may cause loss of most recently committed transactions
 - lost transactions seem uncommitted to database and are cleanly aborted at startup, resulting in consistent database state
 - client thinks that transaction committed, but it was aborted
 - maximum delay between commit and flush (risk period):
 $3 \times \text{wal_writer_delay}$ ($= 3 \times 200\text{ms}$ by default)¹
- `fsync`: (default: on)
 - switching off `fsync` might result in unrecoverable data corruption
 - `synchronous_commit`: similar performance, less risk

¹during busy periods the WAL writer favors writing whole pages and may wait up to $3 \times \text{wal_writer_delay}$

4. Tuning Data Writes

- At commit time
 - database buffer (in RAM) has committed information
 - log (on disk) has committed information
 - data file may not have committed information
- Why is **data not immediately written** to data file?
 - each page write requires a seek
 - resulting random I/O bad for performance
- Convenient writes:
 - wait and write larger chunks at once
 - write when cheap, e.g., disk heads are on the right cylinder

Database Writes – Tuning Options

- Fill ratio of the database buffer (RAM):
 - Oracle: `DB_BLOCK_MAX_DIRTY_TARGET` specifies maximum number of dirty pages in database buffer
 - SQL Server: pages in free lists falls below threshold (3% by default)
- Checkpoint frequency:
 - checkpoint forces all committed writes that are only in database buffer or log to the data file
 - less frequent checkpoints allow more convenient writes
 - less frequent checkpoints increase recovery time

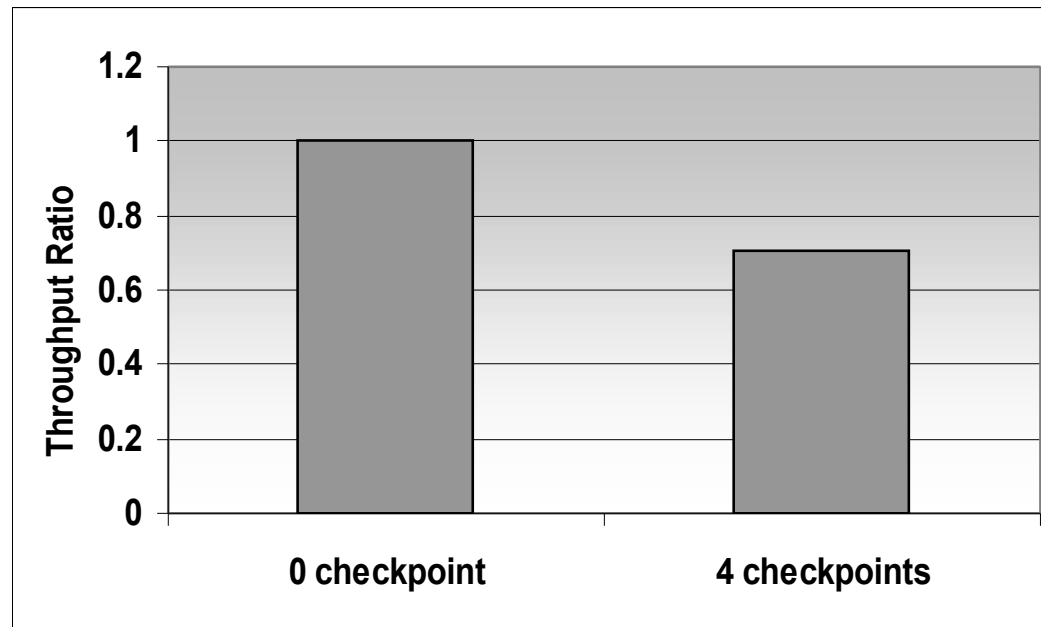
Checkpoint Tuning in PostgreSQL

- Checkpoints have a cost:
 - disk activity to transfer dirty pages to data file
 - if `full_page_writes` is on (avoid partial disk writes), a before image of each page in the buffer that is modified after the checkpoint must be stored in log
- Checkpoint is triggered if one of the following is reached:
 - `checkpoint_timeout` (5min): max interval between checkpoints
 - `max_wal_size` (1GB): max overall size of log segments (16MB each)

Checkpoint Tuning in PostgreSQL

- Spreading checkpoint traffic:
 - checkpoint traffic is distributed to reduce I/O load
 - `checkpoint_completion_target` (0.5): fraction of time before next checkpoint will happen
 - checkpoint should finish within this time period
- Monitoring checkpoints:
 - `checkpoint_warning` (30s): write warning to log if checkpoints happen more frequently
 - frequent appearance indicates that `max_wal_size` should be increased

Checkpoint Tuning – Experiment



- Long transaction with many updates.
- Checkpoints triggered while transaction still active (log file too small).
- Negative impact on performance: size of log files should be increased.

Oracle 8i EE on Windows 2000