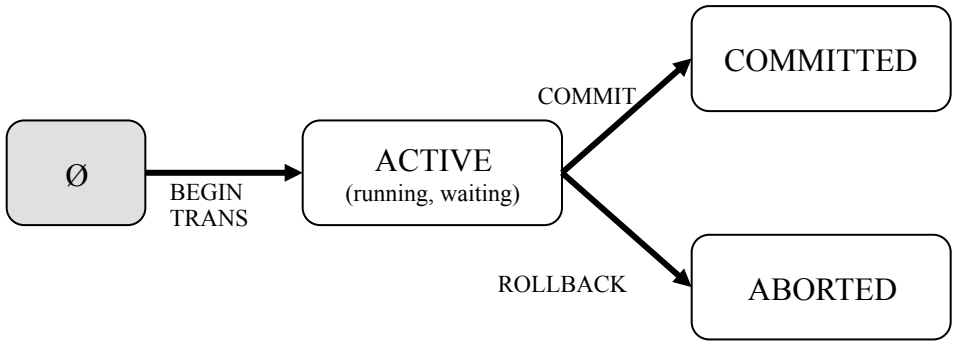
Outline

- Characteristics of the DBMS writes
  - Transactional context
    - Write ahead logging and Lazy writers
  - External algorithms
- Tablespaces
  - DBMS data containers
- Tuning the writes
  - Objectives
  - Tablespace tuning
  - Log tuning

Context#1

- Modification of the database state
  - Transactions contain **update**, **delete**, **insert** statements
    - The challenge for the DBA is to
      - minimize performance performance overhead
      - while the DBMS guarantees Atomicity and Durability
    - We ignore
      - Isolation issues – see lock tuning
      - Consistency issues – see query tuning
      - Database loading – see tuning the application interface

Atomicity and Durability



- Every transaction either commits or aborts. It cannot change its mind
- Even in the face of (some) failures:
  - Effects of committed transactions should be permanent;
  - Effects of aborted transactions should leave no trace.

Transactions aborted by:

- User (*e.g.,* cancel button)
- Transaction manager (*e.g.,* deferred constraint check)
- DBMS (*e.g.,* deadlock, lack of resources)

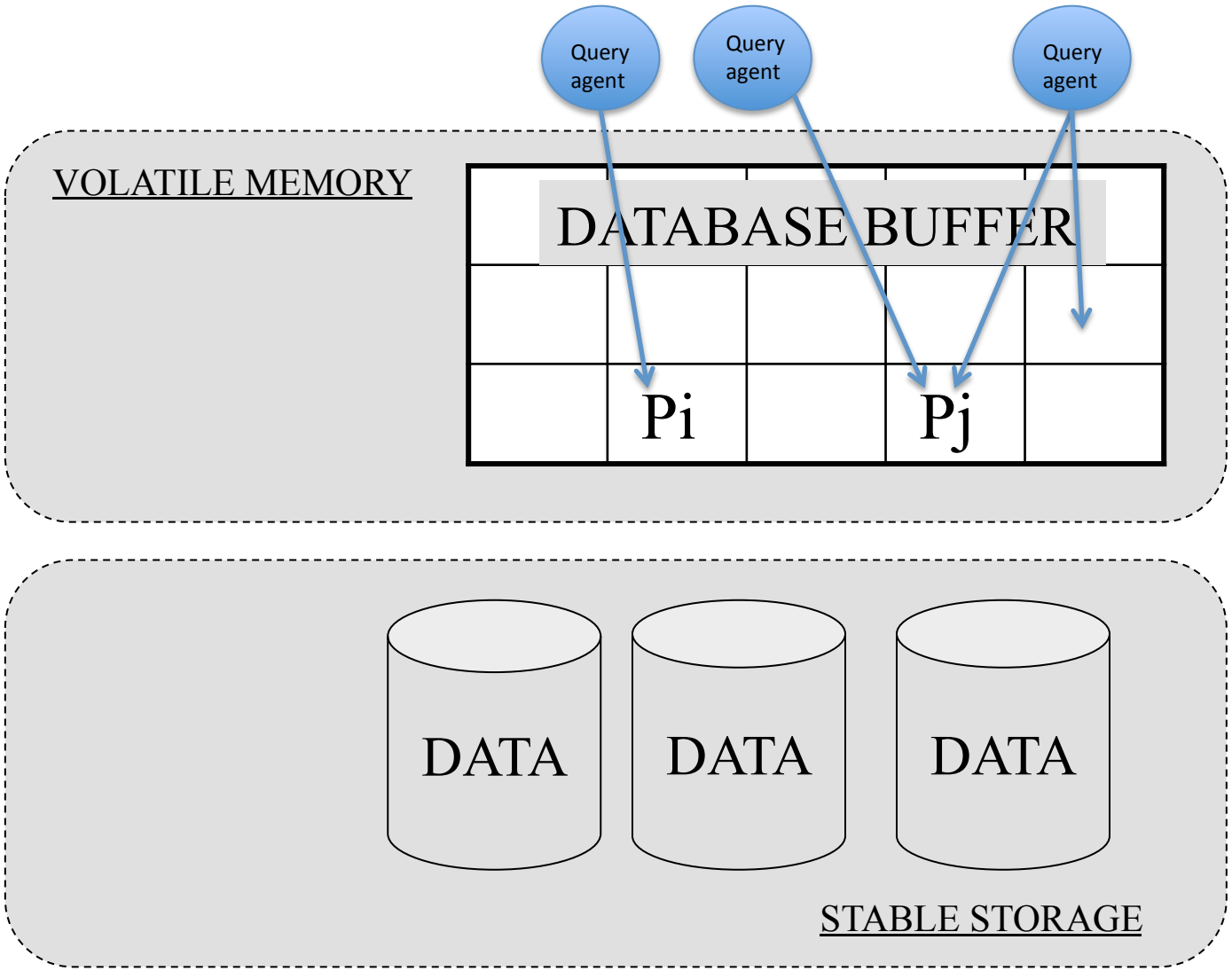
(Some) Failures

- Processor failure, system crash, power outage, software bug
  - Program behaves unpredictably, possibly erasing or corrupting RAM contents (*volatile memory*)
  - Contents of stable storage generally unaffected
  - Active transactions interrupted, database left in inconsistent state
- Media failures
  - Contents of stable storage is corrupted

Other Failures

- The failures indicated in the previous slides do not include
  - Multiple hardware failures (e.g., CPU and controller failure)
  - Disaster (e.g., building on fire)
- Fault-tolerance ≠ durability
  - Fault-tolerance requires solutions that are beyond the scope of atomicity and durability
  - Replication, redundant hardware, Geo-plexing

Update/insert/delete Processing



How to handle writes from memory to secondary storage so that the DBMS guarantees atomicity and durability?

Buffer Management

- When are dirty pages written to disk?
  - Modified pages are called dirty pages
  - Depends on buffer management policy
- Four types of buffer management policy
  - Steal vs. No Steal
    - **Steal**: Dirty pages modified by non-committed transactions might be written to disk
    - **No Steal**: Dirty pages modified by non-committed transactions might NOT be written to disk
  - Force vs. No Force
    - **Force**: Dirty pages modified by transaction T are forced to disk when T commits
    - **No Force**: Dirty pages modified by transaction T are NOT forced to disk when T commits

Buffer Management

	No Steal	Steal
Force		Undo
No Force	Redo	Undo Redo

- When a dirty page is stolen, the effect of a non-committed transaction is reflected on stable storage
  - Must be **undone** in case the transaction aborts, or the system crashes.
- When no-force is used, the effect of a committed transaction is NOT reflected on stable storage
  - Must be redone in case the transaction aborts, or the system crashes

Logging

- REDO and UNDO information stored in a *log.*
  - Sequence of log records
  - Log records as compact as possible to minimize the amount of data written to disk (typically multiple records per log page)
- Update records contains:

  LSN, XID, LastLSN, <pageID, offset, length, old data, new data>
- To each log record is associated
  - LSN: a log sequence number, i.e., a logical time stamp for the log record
  - XID: the transaction ID that owns the record
  - LastLSN: the log sequence number of the previous log record for the given XID
- Log file implemented as circular buffer
  - Unless media recovery is activated

Current database state = current state of data on disks + log

Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

1. Must force the log record for an update to disk *before* the corresponding data page is stolen.

   Guarantees Atomicity (undo)

2. Must write to disk all log records for a transaction *before it commits*.

   Guarantees Durability (undo & redo)

The log is used to (a) abort transactions and (b) perform crash recovery in case of crash or failure.

The assumption is that locking is in effect when log records are created – see lock tuning.

In terms of performance, the goal is to minimize

- The time spent writing to the log
- The time spent aborting a transaction
- The time spent performing crash recovery

Note that there are limits on the size of individual log files and on the number of log files. These limits actually form the boundaries of how much load can be accepted by a DBMS instance.

LOOK UP: ARIES

Transaction Abort

- How can a log be used to abort transaction T_i?
  - The effects of the stolen pages must be rolled back, using the before image [undo] in the update records associated to T_i
    - The log is scanned backwards
    - Each update is undone in last-in first-out order
  - Problem#1: When to stop scanning backwards?
    - Solution: Append a begin record for each transaction prior to its first update record

Transaction Abort

– Problem#2: Should the undo operation be logged?
  - YES. If three operations have to be rolled back and the system crashes, then it must be possible to find out from the log which updates have been rolled back
  - Solution:
    1. Introduce an ***Abort record*** that marks the start of the abort procedure
    2. Distinguish rollbacks from updates. Introduce new log record, called ***Compensation log record (CLR)*** that contains the before image that has been restored and a pointer (LSN) to the next record to be undone in this transaction
    3. Introduce an ***End record*** when the abort procedure completes

Example log

LSN	XID	Last LSN	TYPE	RECORD
0	1	null	Update	<pid: P3, old: 111, new: 222>
1	2	null	Update	<pid: P4, old:777, new: 111>
2	3	null	Update	<pid: P6, old:999, new: 222>
3	1	0	Update	<pid: P2, old: 123, new: 321>
4	2	1	Update	<pid:P0, old:000, new:111>
5	2	4	Abort	
6	2	5	CLR	<pid: P0, old:000, undoNextLSN: 1>
7	2	6	CLR	<pid: P4, old:777, undoNextLSN: null>
8	3	2	Update	<pid: P5, old:444, new:098>
9	2	7	End	

Crash Recovery

- Using the log, it must be efficient to answer the following questions:
  1. What were the transactions active at the time of the crash?
     - Their effects must be undone (if necessary)
  2. What were the committed/aborted transactions at the time of the crash?
     - Their effects must be redone (if necessary)

Log records

- Log records should allow to distinguish between active, committed and aborted transactions
  - ***Commit record*** appended to the log when transaction commit procedure starts
  - End record appended to the log when transaction has committed
- The recovery procedure should not have to scan the entire log whenever there is a crash
  - ***Checkpoint Record (CK)*** contains XID of the active transactions at a given point in time (LSN)

Checkpoint

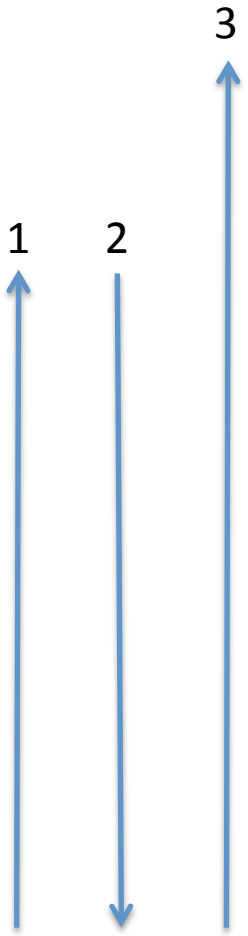
- Sharp checkpoint
  - Procedure:
    - System stops at time *t*
    - Dirty pages for all transactions active at time *t* are forced to disk
    - CK record is written to the log
    - System restarts
  - This way:
    - All of the effects of the transactions active at checkpoint time are reflected on disk
    - No need to look up the log records prior to the checkpoint to roll database forward when performing recovery

Recovery with Sharp Checkpoint

1. Scan log backward until CK record to maintain a list of all transactions that were active at the time of the crash
   - The first record encountered for a given transaction (XID) is not an End record
   - XID is contained in the checkpoint record
2. Log is scanned forward until end of the log
   - The after images (new) in all update records are used to roll the database forward
3. Log is scanned backward until lastLSN is null for all active transactions at the time of the crash. The before images are used to roll the database backward
   - Each active transaction at the time of the crash is in effect aborted, abort/CLR/end records are appended to the log as the rollback proceeds

Example log

LSN	XID	Last LSN	TYPE	RECORD
0	1	null	Update	<pid: P3, old: 111, new: 222>
1	2	null	Update	<pid: P4, old:777, new: 111>
2			CK	<XIDs: [<XID:1, LSN:0>,<XID:2,LSN:1>]>
3	3	null	Update	<pid: P6, old:999, new: 222>
4	1	0	Update	<pid: P2, old: 123, new: 321>
5	2	1	Update	<pid:P0, old:000, new:111>
6	2	5	Abort	
7	2	6	CLR	<pid: P0, old:000, undoNextLSN: 1>
8	2	7	CLR	<pid: P4, old:777, undoNextLSN: null>
9	3	3	Update	<pid: P5, old:444, new:098>
10	2	8	End	



Fuzzy Checkpoint

- Problem with sharp checkpoint:
  - The DBMS stops while all dirty pages are flushed
- Solution: Fuzzy checkpoint
  - The DBMS does not stop at checkpoint time
  - The system records the list of dirty pages at the time of the checkpoint together with the list active transactions
  - All recorded dirty pages have to be written to disk before the next fuzzy checkpoint
  - CK record replaced by ***Begin Checkpoint*** and ***End Checkpoint*** log records

Recovery with Fuzzy Checkpoint

1. Scan log backward until second End checkpoint record is encountered to maintain a list of all transactions that were active at the time of the crash
   - The first record encountered for a given transaction (XID) is not an End record
   - XID is contained in the second End checkpoint record
2. Log is scanned forward until end of the log
   - The after images (new) in all update records are used to roll the database forward
3. Log is scanned backward until lastLSN is null for all active transactions at the time of the crash. The before images are used to roll the database backward
   - Each active transaction at the time of the crash is in effect aborted, abort/CLR/end records are appended to the log as the rollback proceeds

Physical Logging

- Update records contain before and after images
  - Roll-back: install before image
  - Roll-forward: install after image
- Pros:
  - Idempotent. If a crash occurs during recovery, then recovery simply restart as phase 2 (rollforward) and 3 (rollback) rely on operations that are indempotent.
- Cons:
  - A single SQL statement might touch many pages and thus generate many update records
  - The before and after images are large

Logical Logging

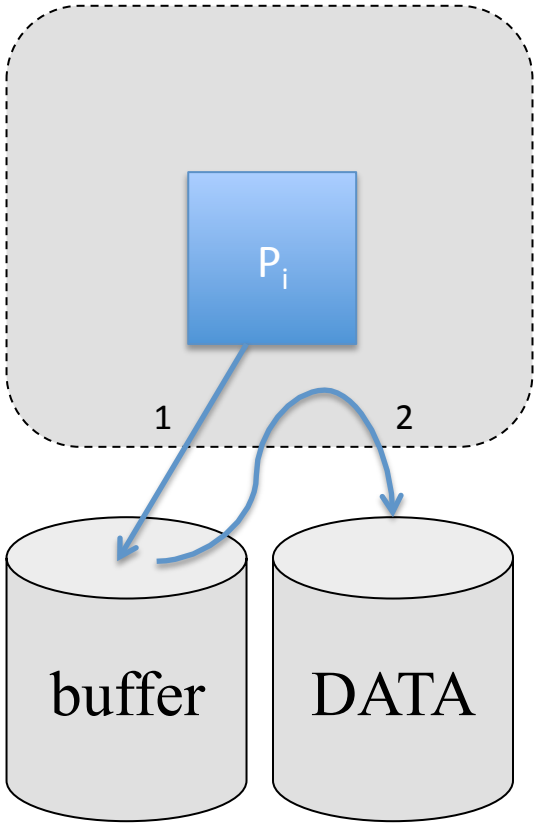
- Update records contain logical operations and its inverse instead of before and after image
  - E.g., <op: insert t in T, inv: delete t from T>
- Pro: compact
- Cons:
  - Not idempotent.
    - Solution: Include a LastLSN in each database page. During phase 2 of recovery, an operation is rolled forward iff its LSN is higher than the LastLSN of the page.
  - Not atomic
    - What if a logical operation actually involves several pages, e.g., a data and index page? And possibly several index pages?
    - Solution: Physiological logging

Physiological Logging

- Physiological logging
  - Physical across pages
  - Logical within a page
- Combines the benefits of logical logging and avoids the problem of atomicity, as logical mini-operations are bound to a single page
  - Logical operations split into mini-operations on each page
  - A log record is created for each mini-operation
  - Mini-operations are not idempotent, thus page LSN have to be used in phase 2 of recovery

How to deal with Media Failures?

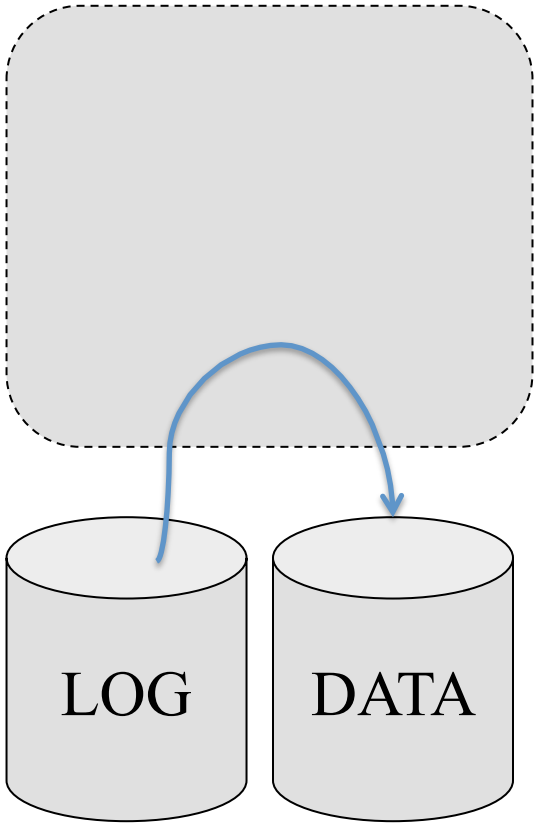
- Double-write buffer
  - If problem when writing to the separate buffer, dataspace is not corrupted. Write can be repeated on buffer.
  - If problem when copying the page from buffer to tablespace, then dataspace might get corrupted. But a valid copy of the page exists in the buffer and copy can be reexecuted.



How to deal with Media Failure?

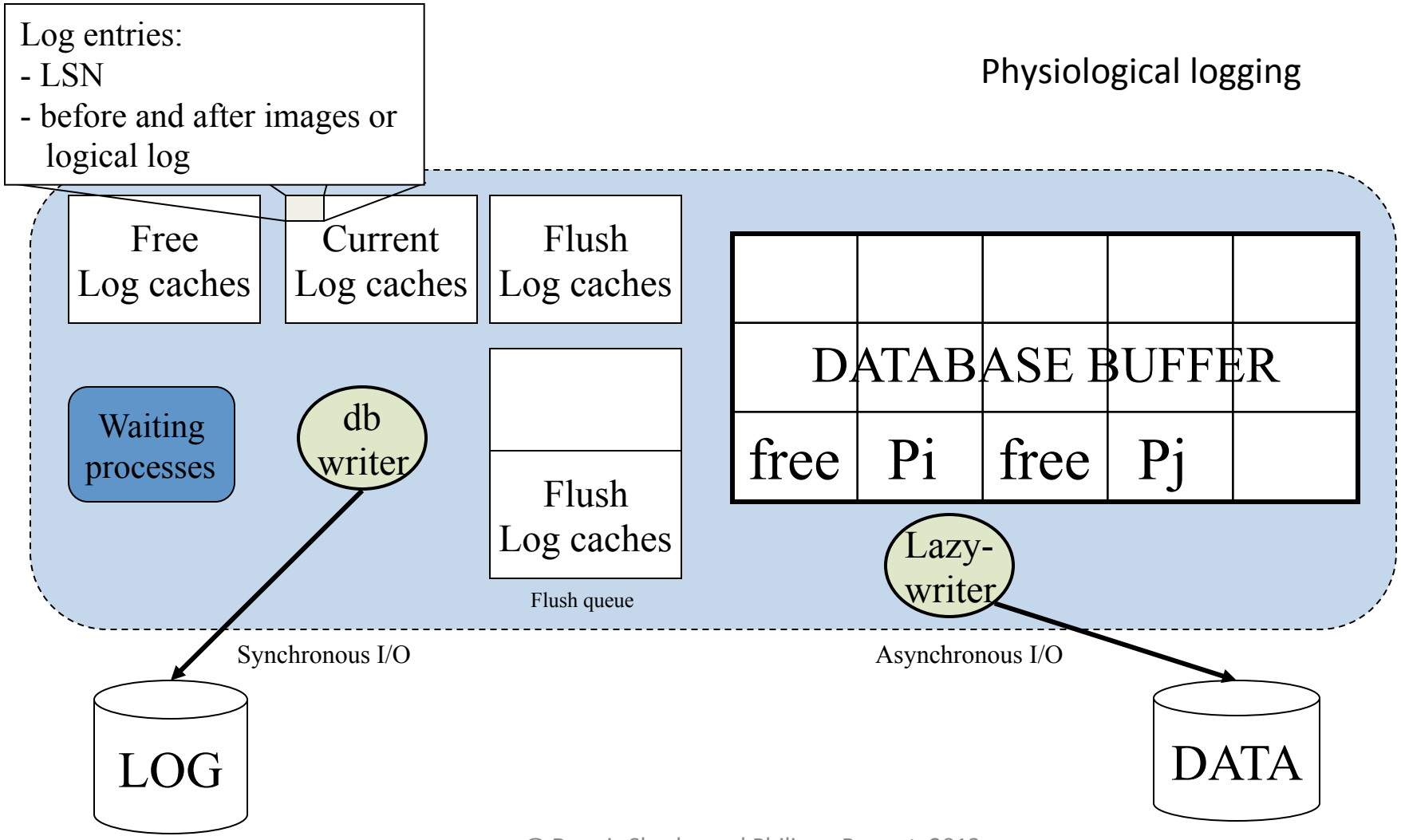
- Regular backup + roll forward recovery
  - Backup database/tablespace
  - Archived logs
  - Apply records from archived and active log to bring current state of data on disk up to date

Note: On Oracle, switching active log files when a log file is full triggers a media recovery checkpoint (i.e., a roll forward from the full log)



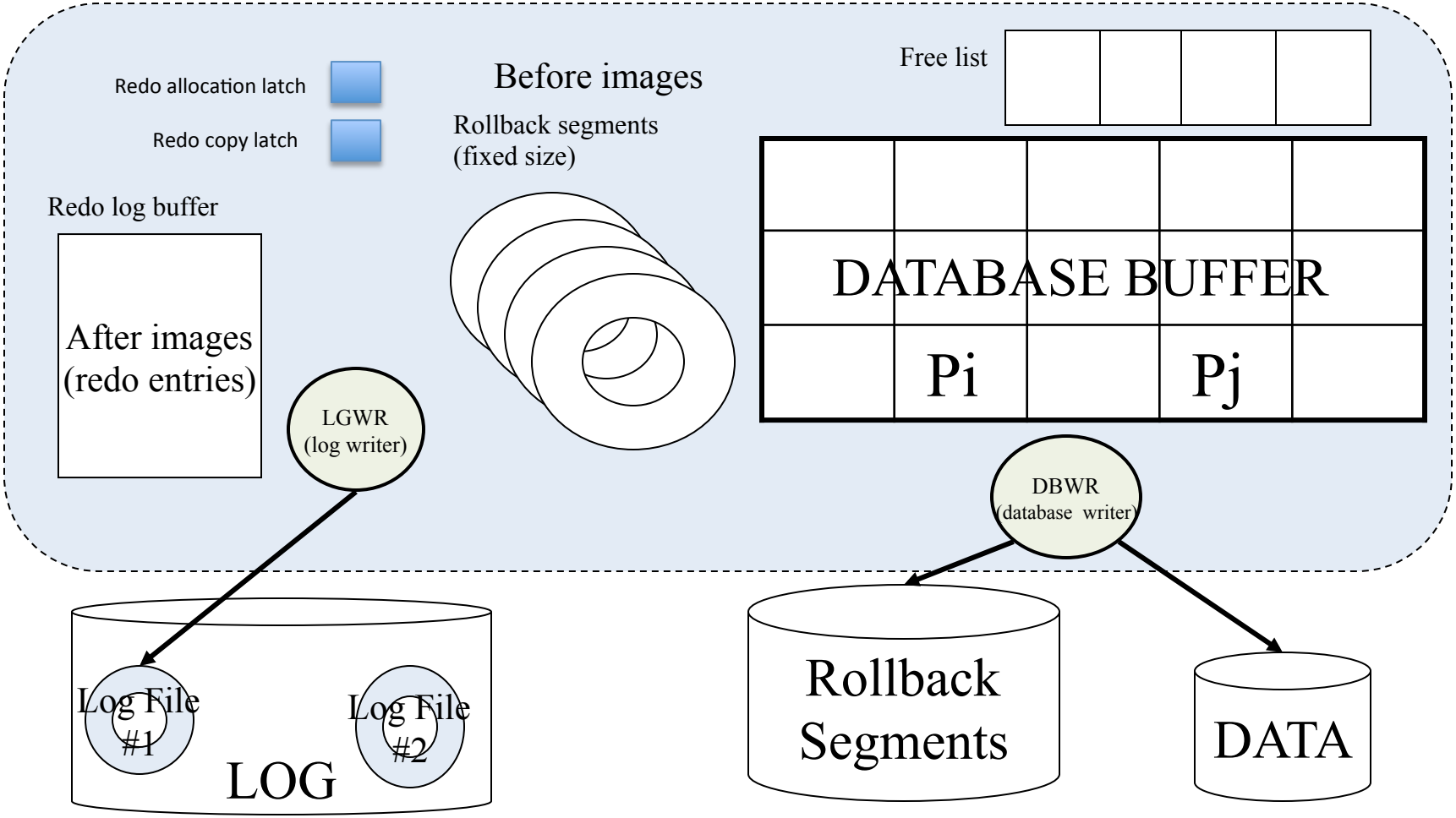
Source: Oracle Core by Jonathan Lewis

Logging in SQL Server, DB2



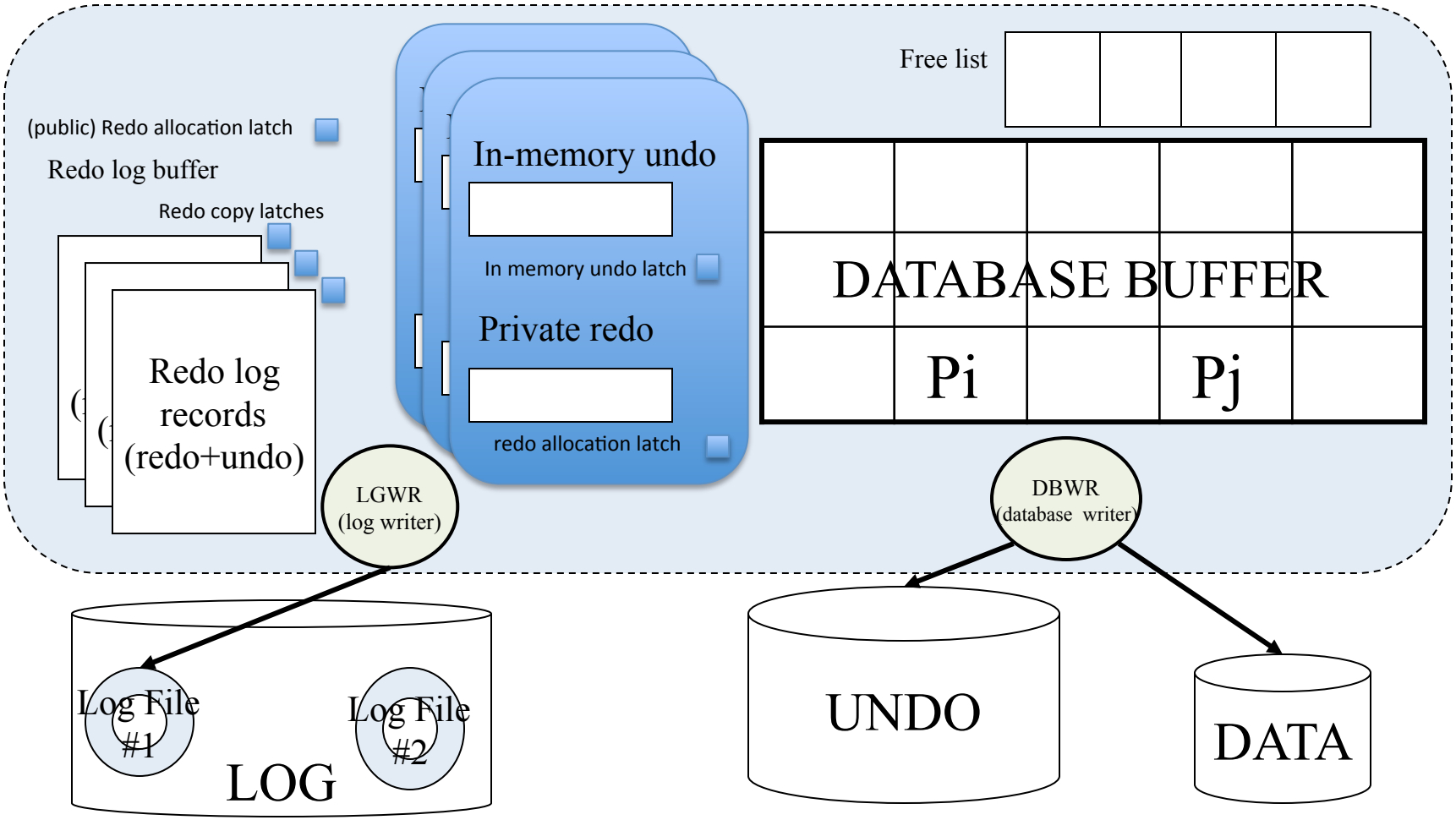
Logging in Oracle prior to 10g

Physiological logging



Logging in Oracle after 10g

Physiological logging



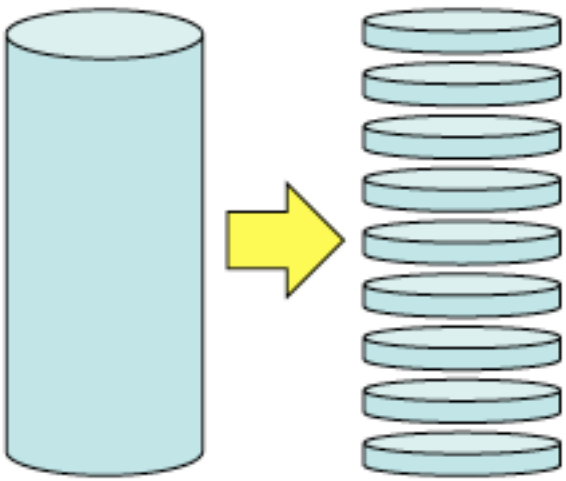
Context#1 Wrap-up

- The application submits transactions that insert/delete/update tables
- What write IOs are performed?
  - Case#1: The buffer is not full
    - Log records are written to the log at commit time
    - Dirty data/index pages modified by these transactions are written to the data tablespaces at some point after the transaction commits, and before the next checkpoint
  - Case#2: The buffer is full
    - Dirty pages (possibly modified by other transactions) are stolen to make room for new pages as they are required
    - Log records are written to the log at commit time, and when dirty pages are stolen
    - Dirty data/index pages modified by these transactions are written to the data tablespaces at some point after the transaction commits, and before the next checkpoint

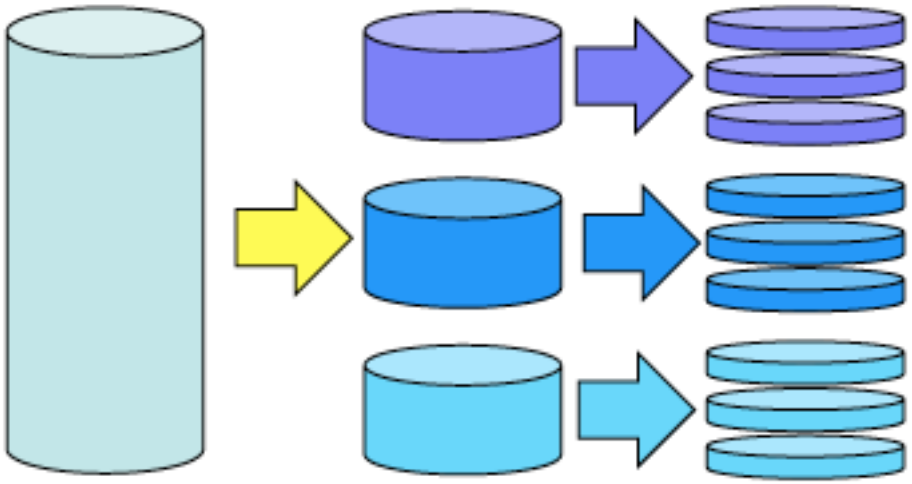
Context #2

- External algorithms for sorting/hashing manipulate a working set which is larger than RAM (or larger than the buffer space allocated for sorting/hashing)
  - Sort or hash is performed in multiple passes
  - In each pass data is read from disk, hashed/ sorted/merged in memory and then written to secondary
  - Pass N+1 can only start when Pass N is done writing data back to secondary storage

RAM occupation vs. Page size



One pass partitioning



Multi-pass partitioning (2 passes)

- For hashing: RAM occupation = 2 * page size * Nb partitions
- How to set page size?
  - Large page size (up to 1MB) is good for IO throughput , …
  - … but it might cause multiple passes

External Storage

- The buckets written/read during an external algorithm contain temporary data
  - Buckets are not relations
  - The data should not persist
- Most systems allow to define tablespaces for temporary data
- In MySQL external algorithms do not rely on the storage manager, they directly access the file system

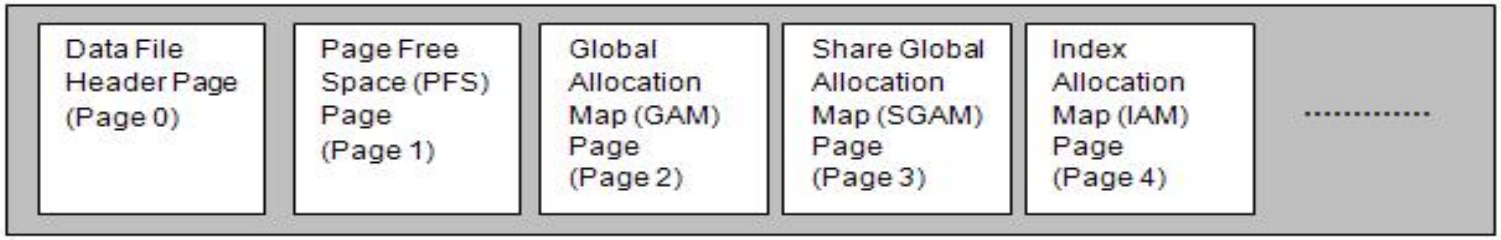
Tablespaces

- Layer of storage abstraction on top of the file system
  - Database objects (tables, indexes, large objects, temporary data)
    - The log is not stored on a tablespace, but on dedicated files
    - Oracle and DB2 distinguish between
      - System (catalog) vs. user tablespaces
      - Manually vs. automatically managed
      - Whether logging is turned on for a given tablespace
    - DB2 associates buffer pools to tablespaces
  - A tablespace is mapped onto one or more files

LOOK UP: Oracle tablespace, DB2 tablespace, SQL Server file groups, InnoDB tablespaces

Tablespace Structure

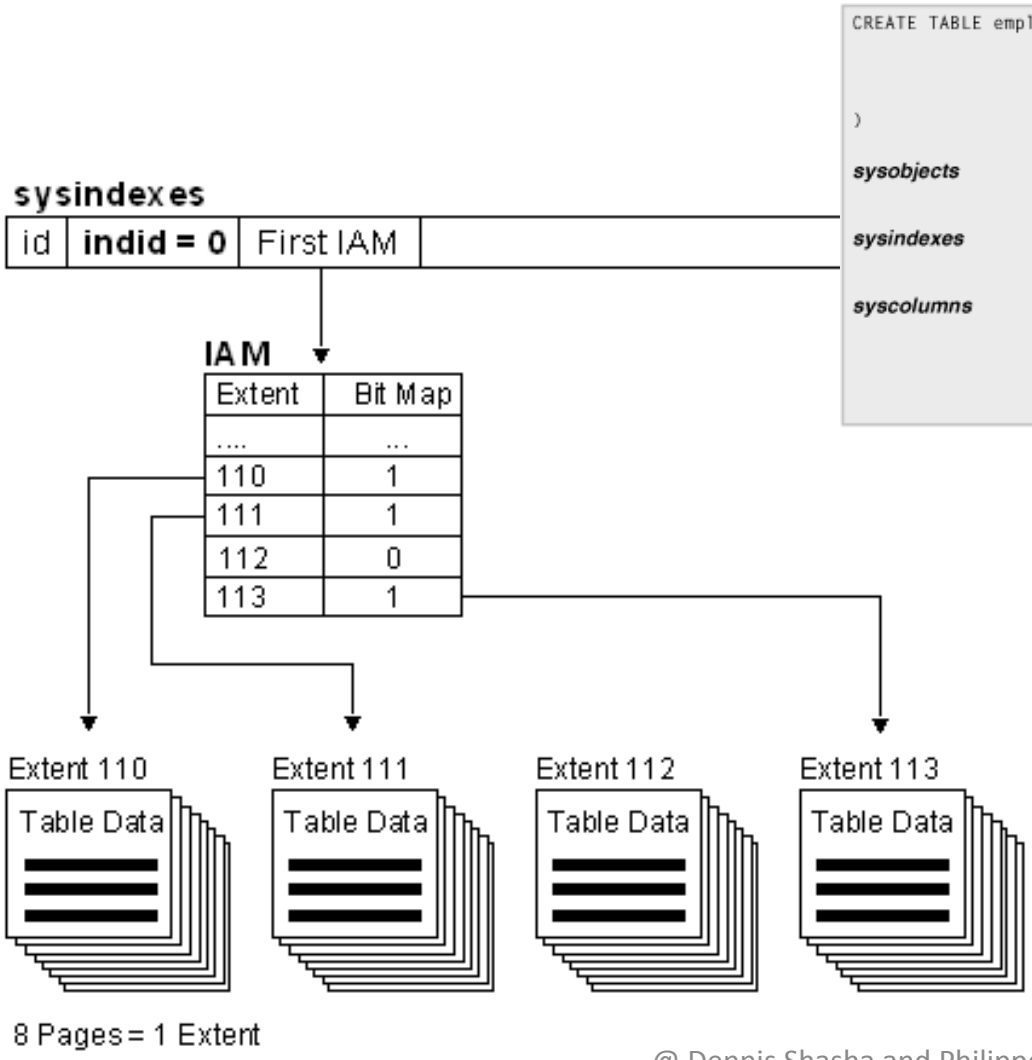
SQL Server



- Extent-based allocation
  - 1 Extent = 8 pages
  - Mixed/Uniform extents
- GAM bitmap over 64000 extents
  - Is extent allocated?
- SGAM bitmap over 64000 extents
  - Is extent mixed and has at least 1 unused page?
- PFS page over 8000 pages
  - 1B per page: How much is page used?

Source: Karen Delaney

Finding Data Pages




```
CREATE TABLE employee     (
                          emp_lname          varchar(15)       NOT NULL,
                          emp_fname          varchar(10)       NOT NULL,
                          address            varchar(30)       NOT NULL,
                          phone              char(12)          NULL,
                          job_level          smallint          NOT NULL
)
```

sysobjects

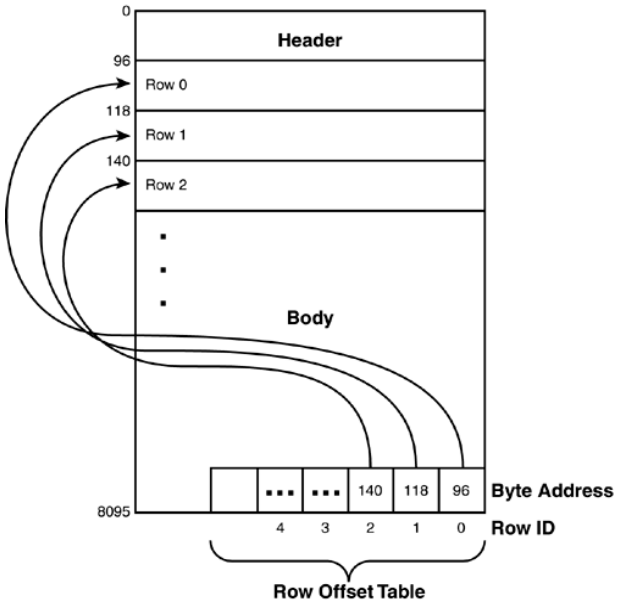
id	name	uid	xtype
165575628	employee	1	U

sysindexes

id	indid	dpages	reserved	rows	first
165575628	0	1	2	0	0x560000000100

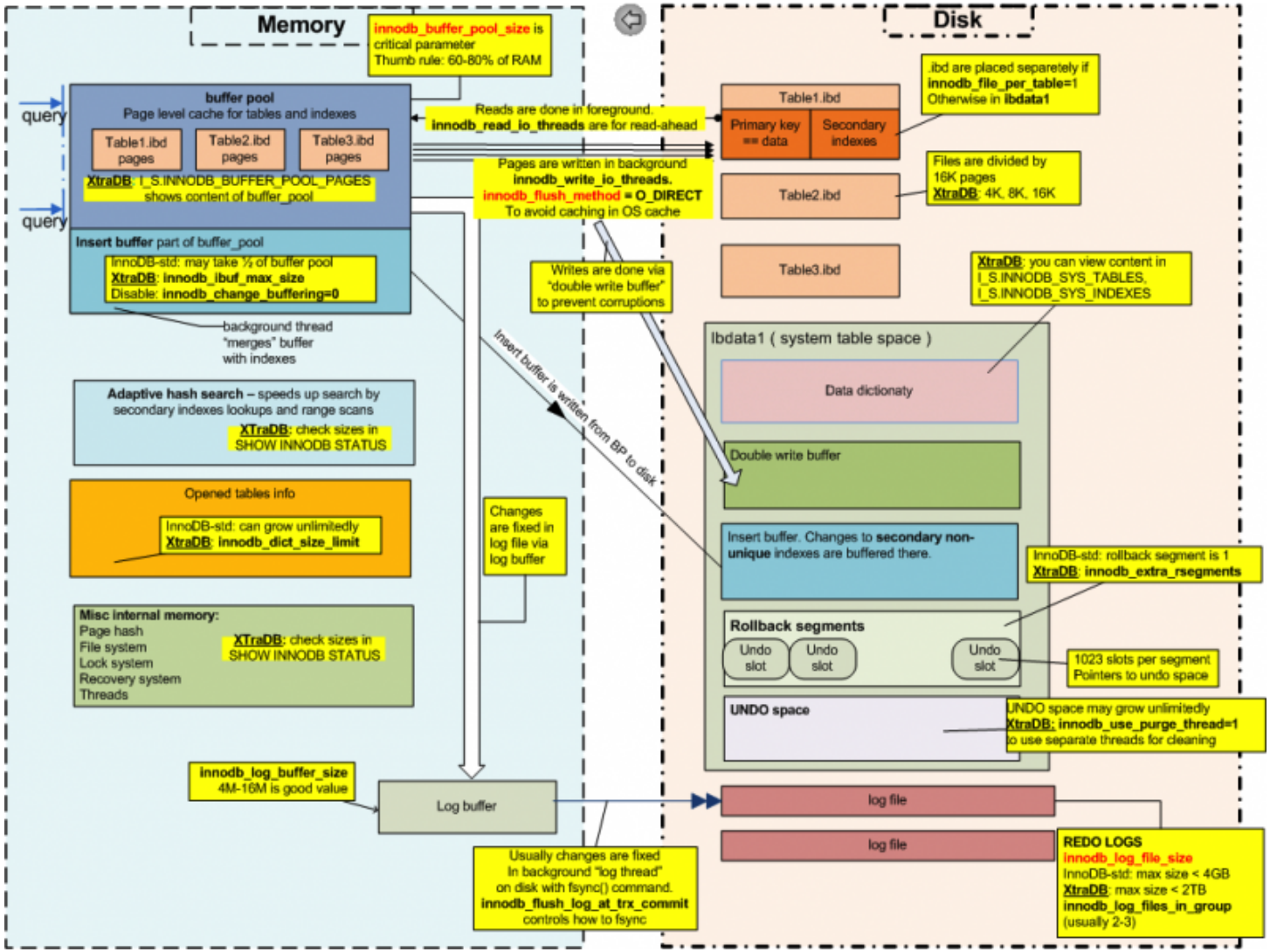
syscolumns

id	colid	name	xtype	length	xoffset
165575628	1	emp_lname	167	15	-1
165575628	2	emp_fname	167	10	-2
165575628	3	address	167	30	-3
165575628	4	phone	175	12	4
165575628	5	job_level	52	2	16



Source: Karen Delaney

InnoDB/XtraDB Storage Model



@ Dennis Shasha and Philippe Bonnet, 2013

Source: Percona

Tablespace Parameters

- Tablespace size
  - Costly resizing needed if tablespace too small
- Page size
  - Default value (4K), set when creating a tablespace
- Extent size
  - Number of pages per extent
  - Tradeoff:
    - Many pages favours contiguity in the logical space and thus favours sequential access
    - Few pages reduces fragmentation
- Prefetch size
  - Number of consecutive pages read by prefetcher

Tuning the Writes

- The overall objective when modifying the database state is:
  - Avoid full logs
    - Worste case it stops all transactions. It might also cause media recovery checkpoint.
  - Minimize performance overhead as writes are performed
    - When the DBMS bufer is not full
      - Only writes to the log are relevant
    - When the DBMS buffer is full
      - Both writes to the log and asynchronous writes to data tablespaces are relevant
    - Ideally, writes to the log are as fast as sequential writes on the fastest disk in the system
  - Minimize the time needed to abort a transaction or perform crash recovery
    - Ideally, aborting a transaction does not require IOs
    - Crash recovery based on three sequential passes on the log
- The overall objective for external algorithms is to
  - Avoid multiple passes
  - Minimize the time requires for sequentially writing/reading multiple buckets

Tuning Goals

- Choose and maintain appropriate size for log files
  - *Manage checkpoint frequency*
  - *Avoid very long updates*
- Bridge the gap between (i) sequential write performance on a file and writes to the log, and (ii) sequential read/write performance on a file and read/writes on temporary data
  - Avoid IO interferences to the log file(s)
    - *Log on a separate disk/SAN? Temp tablespaces on a separate disk?*
  - Favour IO contiguity when writing to the log or temp data
    - *Adjust page and extent size*
  - Avoid any waits when writing the the log buffer, when flushing the log buffer
    - *Group commits*

Mind The Gap

- Need to establish a reference performance for sequential read/writes on log disk and temporary data, and for random writes on data disks
  - Use flexible IO tester tool – fio
  - Simple job descriptions for sequential writes/reads and for random writes
    - Asynchronous, direct IOs
    - numjobs
      - 1 [1 log writer per instance]
      - max(nb partitions, #cores) [number of lazy writers]
    - iodepth: 1..32 [depends on rate of writes], 1 for reads
    - Page size: 4k,8k,16k,32k

Mind the Gap

- Tune DBMS (page size, extent size, log buffer size, number of lazy writers) to get write performance close to reference on file system
  - Should be a constant overhead in terms of throughput for a given workload
    - Measure disk throughput in GB/sec with OS tool (iostat) while transactions are executed

Rightsizing Log Files

- The active log is defined as the portion of the log which is required for crash recovery.
- The size of the active log is the maximum of:
  1. Checkpoint frequency
     - Determines the start of roll forward phase
  2. Transaction duration
     - Determines the end of roll back phase

Reduce the Size of Large Update Transactions

- Consider an update-intensive batch transaction (concurrent access is not an issue):

  It can be broken up in short transactions (mini-batch):

  + Does not overfill the log buffers
  + Does not overfill the log files

  Example: Transaction that updates, in sorted order, all accounts that had activity on them, in a given day.
  Break-up to mini-batches each of which access 10,000 accounts and then updates a global counter.

  Note: DB2 has a parameter limiting the portion of the log used by a single transaction (max_log)

Rightsizing Log Files

- Find out how much data is written to the log at peak load
- Pick checkpoint frequency so that active log is smaller than maximal log file size, i.e., **use a single log file per DBMS instance**!
  - High frequency of checkpoints will lead to writes to the data tablespaces, but at least those writes are performed at a steady rather than in batches whenever roll forward recovery is needed.
    - This is a trade-off between write performance overhead and recovery performance
    - Two log files only to tolerate media failure of a log file.
- As a result:
  - Length of roll forward recovery (needed for media recovery or crash recovery) fixed by checkpoint frequency

Put Log on a Separate Disk

- Improve log writer performance
  - HDD: sequential IOs not disturbed by random IOs
  - SSD: minimal garbage collection/wear leveling
- Isolate data and log failures

Group Commits

- For small transactions, log records might have to be flushed to disk before a log page is filled up
- When many small transactions are committed, many IOs are executed to flush half empty pages
- Group commits allow to delay transaction commit until a log page is filled up
  - Many transactions committed together
- Pros: Avoid too many round trips to disk, i.e., improves throughput
- Cons: Increase mean response time