3. Files and Segments

Theo Härder
www.haerder.de

Main reference:
Theo Härder, Erhard Rahm: Datenbanksysteme – Konzepte und Techniken der Implementierung, Springer, 2001, Chapter 3.

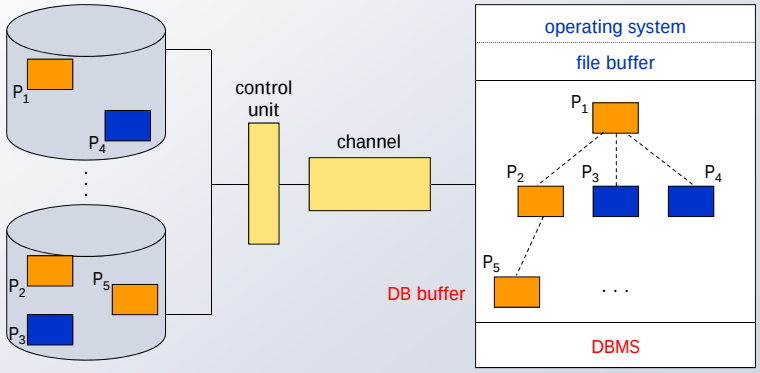
Jim Gray, Andreas Reuter: Transaction Processing – Concepts and Techniques, 5th printing, Morgan Kaufmann Publ., 1993, Chapter 2-3.

External Storage Management

- **Mapping of files and blocks**
  - 2-level storage hierarchy
  - General tasks of storage management
- **File system**
  - Operations, addressing, and mapping
  - Methods for block allocation
    - Static
    - Dynamic extent allocation
    - Dynamic block allocation
- **Measures to enhance fault tolerance**
  - Reads and writes of blocks
  - Use of storage redundancy
- **Mapping of segments and pages**
  - Segment concept
  - Deferred propagation strategies
    - Shadow page mechanism
    - Differential file
  - Evaluation of page allocation methods

Access Structures in a Two-Level Storage Hierarchy

- **Simplified I/O architecture:**
  - Model for the connection of external storage
  - Before processing, the pages have to be copied into the DB buffer
  - Modified pages have to be propagated to non-volatile storage



Mapping of Files and Blocks – File Services

- **Example: call of an operation at the upper interface**

```
        PAM   UPAMDAT, RD, FECB=ESTBLOCK, HP=1
                .
                .
                .
UPAMDAT FCB   FCBTYPE=PAM, LINK=EIN, IOAREA1=EINBER
```

- **Mapping function**

  block number ——— cyl-#, track-#, rec-#

- **I/O execution:**

  Arm positioning: EXCP → CCW
  Track selection: EXCP → CCW
  Record transmission: EXCP → CCW
  } I/O processor

- **Properties of the upper interface**
  - Set of numbered blocks within files
  - Read- and write access via block numbers
  - Block accesses can result in read errors, blocks can contain old or invalid data

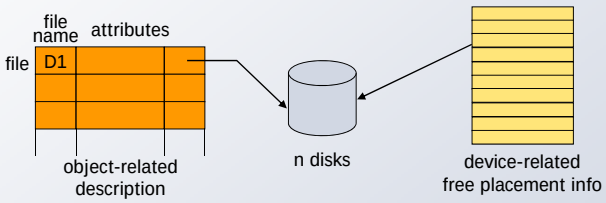
Mapping of Files and Blocks

- **Requirements**
  - Management of external storage media
  - Hiding of device properties
  - Mapping of physical blocks
  - Control of data transport from/to main memory
  - Realizierung of a multi-level storage hierarchy, if necessary
  - Fault tolerance measures
    (stable storage, mirrored disks, etc.)
- **Reasons for a file concept**
  - Selective activation of files: on/offline problem
  - Dynamic definition
  - Temporary files
  - Use of storage media with varying performance characteristics
  - Shorter addresses

DB storage $\triangleq$ set of files

Realization of a File System

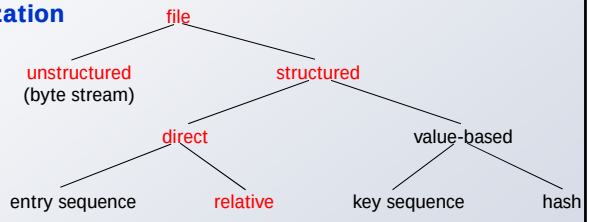
- **File catalog for all files (primordial file)**
  - Fixed position
  - Efficient implementation/processing
- Example:
  file catalog



- Object-related decription by file descriptor
  - File name, OwnerID, access control list, file size, storage allocation, . . .
  - Information about creation, most recent access, archiving, . . .
- Free placement administration for external storage
  - Formatted bit lists, hierarchical structure
- Creation/reservation of storage areas
  - Primary assignment, extension
- Unit of physical access: block
  - Blocks of variable size? → **fixed block size** per file
  - Chained I/O important for high I/O performance

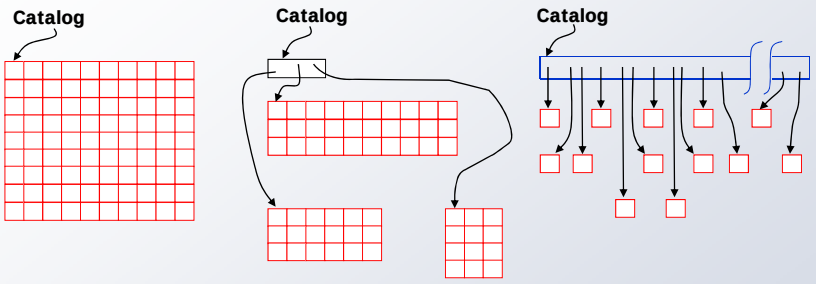
File Organizations

- **File system**
  - Manages the files created and executes the read-/write accesses
  - Keeps a descriptor for every file
- **Forms of organization**



  - Entry-sequenced file: insertion at end of file, record address is RBA (relative byte address), sequential access or direct access via RBA
  - Relative file: form of organisation is ARRAY OF RECORDS
  - Value-based files permit access via record keys
    - Key-sequenced file: organization of records in key sequence, direct access via key and sorted-sequential access
    - Hash file: direct access via key transformation
- **Terms** in existing systems (logical access methods)
  - SAM, ISAM/VSAM, . . ., BDAM/PAM, . . .

Block Allocation



- **Static file allocation**
  - Direct addressing
  - Minimal access costs
  - No flexibility
- **Dynamic extent allocation**
  - Addressing via a small table
  - Low access costs
  - Moderate flexibility
- **Dynamic block allocation**
  - Addressing via a large table
  - High access costs
  - Maximal flexibility

Block Allocation by Extent Tables

Directory

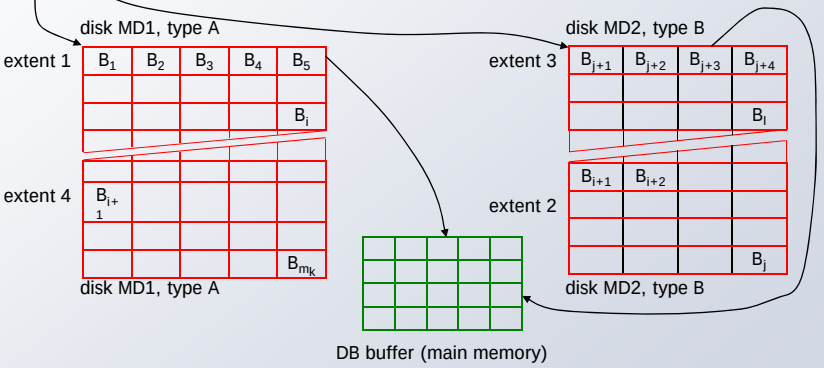
Extent table for file D_k

name	type	begin of extent CYL, Track	number of blocks
MD1	A	Z201, S5	i
MD2	B	Z350, S1	j-i
MD2	B	Z105, S1	l-j
MD1	A	Z499, S7	m_{k-l}

Presented actions for accesses:

- Fetch block B_5
- Fetch block b_{j+3}

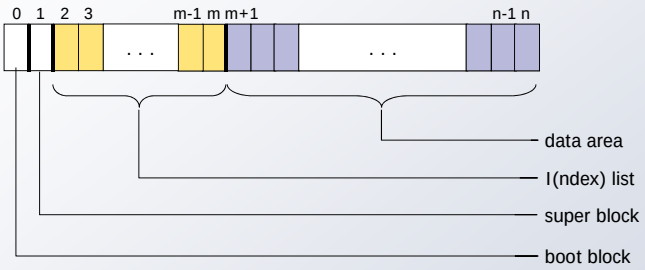
if a block is fetched for modification, it is propagated back after release



Typical implementations of this method create a *primary extent* which can be extended with up to 15 *secondary extents* (of equal size each)

File System in UNIX

- **Base structure (for the organization of disk storage)**

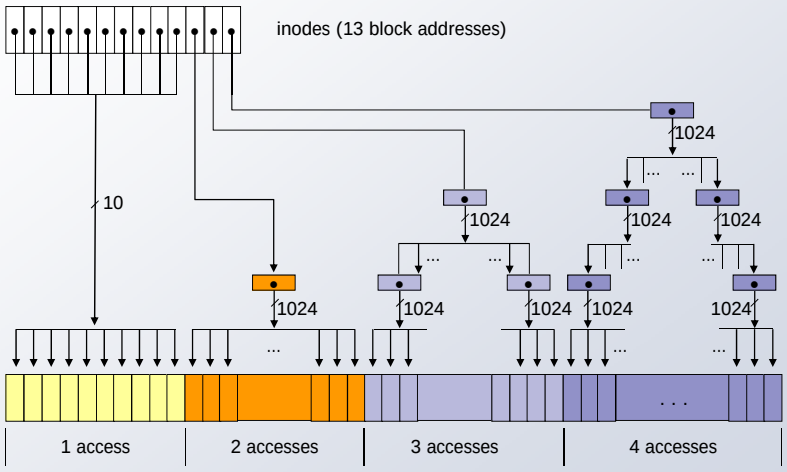


- **Important description information**

Directory entries relate to the file name an index I ($1 \le I \le m-1$).
Each node of the I-list contains the following information:

- Identification of the file owner, protection bits
- Addresses of 13 physical blocks, file size in bytes
- Time of creation, most recent reference and modification
- Number of references to the file, kind of file

Storage Allocation in the UNIX File System



The file blocks at the lowest level are only logically contiguous

Block Allocation – Optimization

- Log-structured files

  Proposal rather appropriate for applications having a multitude of small files, mostly with a short life time
  - Avoidance of random reads using large main-memory buffers
  - **Write optimization** of modified blocks embodies the **key idea**.
    All modified blocks are sequentially written at the end of file processing (transaction), in a single batch, to the current end of the file which is organized like a log file and is overwritten in a cyclic way. Aged blocks are then released

    ➥ Complex management of blocks and their metadata; concept cannot be easily transferred to DB files

Catalog Catalog

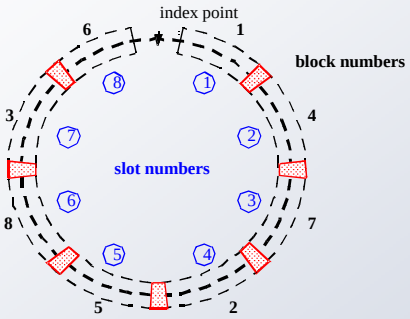
Files: . . . A B C D E| . . . |F G H| . . . | . . . A B C D E| . . . |F G H| . . . |

Disadvantages (http://en.wikipedia.org/wiki/Log-structured_file_system):
The design rationale for log-structured file systems assumes that most reads will be optimized away by ever-enlarging memory caches. This assumption does not always hold:
- On magnetic media—where seeks are relatively expensive—the log structure may actually make reads much slower, since it fragments files that conventional file systems normally keep contiguous with in-place writes.
- On flash memory—where seek times are usually negligible—the log structure may not confer a worthwhile performance gain, because write fragmentation has much less of an impact on write throughput. However many flash based devices can only write a complete block at a time because they must first perform a (slow) erase cycle before being able to write.

Block Allocation – Optimization (2)

- **Displacement method**

m=8
I=3



- First block in $SN_1 = 1$
- Computation of slot number SNi ($2 \le i \le m$)
  - $SN_i = (SN_i\text{-}1 + I - 1) \bmod (m) + 1$
  - I as measure of displacement ($2 \le I \le m\text{-}1$) serves for the adjustment to the computer speed

➥ Sequential disk accesses are accelerated without hindering random accesses

Measures to Enhance Fault Tolerance

- **MTTF of different disk errors***

Type of Error	MTTF	Recovery	Consequences
Soft data read error	1 hour	Retry or ECC (error correcting code)	None
Recoverable seek error	6 hours	Retry	None
Maskable hard data read error	3 days	ECC	Remap to new sector and rewrite good data
Unrecoverable data read error	1 year	None	Remap to new sector, old data lost
Device needs repair	5 years	Repair	Data unavailable
Miscorrected data read error	10^7 years	None	Read wrong data

* Gray, J., Reuter, A.: Transaction Processing – Concepts and Techniques, Morgan Kaufmann, 1993.

Measures to Enhance Fault Tolerance (2)

Simple read

- Read can be interrupted by transient errors
- Additional measure: unsuccessful read is repeated n times (to protect against transient errors)

Simple write

- Block write as an atomic action (either the entire block is correctly transferred to the specified slot or the slot remains unchanged) cannot be guaranteed
- Write actions can lead to wrong results because of transient and permanent errors

➥ Write error in the catalog?

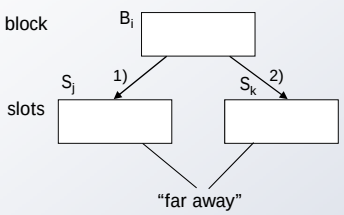
Read-after-write

- After a simple write, the block is immediately read and compared with the original block
- Operation sequence is repeated until the block is successfully written
- Write is protected against transient errors. Permanent errors can, however, lead to wrong results

Measures to Enhance Fault Tolerance (3)

Duplexed write (stable write)

- Every block has a version number which is incremented upon each invocation of the operation
- The block is written in specified sequence in two different slots S_j and S_k
- **Principle:** stable storage



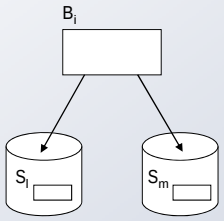
- Simple write: 1) then 2) (synchronous)
- Atomicity for a single block
- Various disks, controllers, I/O paths

- **Assumption:** block is not written to a wrong slot; otherwise, read-after-write is required
- Read of B_i takes place at first from slot S_j. If successful, it is assumed that the block is the youngest valid version of B_i
- If read of slot S_j fails, slot S_k is read
- Because a system crash can only interrupt a single write, there is always a version of the block available at restart

➥ Hence, **writes are protected against permanent errors.**
It is **unexpected** that both versions are not readable

Further Principles of Storage Redundancy

- **Unit of propagation: 1 block**
  - ➥ Fault tolerance measure for a single block against write errors, system crash, block (track, device) destruction
- **Logged write**
  - After read, block B_i is written with its **old content** to a **secure place** L
  - After update, B_i is propagated to its former place (update-in-place) using simple write (*read-after-write*, if necessary).
  - If write was successful, the copy of B_i on L is not needed anymore
- **Principle: mirrored disks**



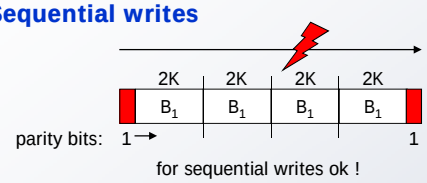
  - On disk or file basis
  - Synchronous or asynchronous
  - At a time or delayed (shadow process)
  - No atomicity automatically through the algorithm

Further Principles of Storage Redundancy (2)

- **Detection of erroneous blocks**
  - Protection against certain kinds of corruption:
    - Parity bits, check sums
  - Disk hardware can automatically figure out using parity bits, whether or not a sector was completely written
  - ➥ Sufficient, if a block is fully stored in a sector (here identical to slot)
- **Slot = sector**
  - Use of a single bit both in the first and last byte of a block
  - Every time, **identical values** are assigned to both consistency bits
- **Multi-sector slots**
  - Erroneous block is only detected, if the block sequence (starting at block begin) is observed when the sectors are written
  - SCSI drives autonomously carry out a reordering of the sector writes to improve write performance
  - Check sums across entire blocks substantially reduce probability to miss a partial write; however, this error cannot fully be excluded (very expensive)
  - Corresponding to the **n Sectors, a block is (logically) divided**; from each of these fractions, a bit is taken and used as check bit. These n bits are filled with identical values and inverted upon each write
    - ➥ What has to be done that these n bits do not falsify user data in the block?

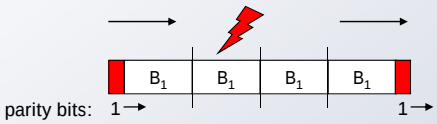
Detection of Incompletely Written Blocks

1. Sequential writes

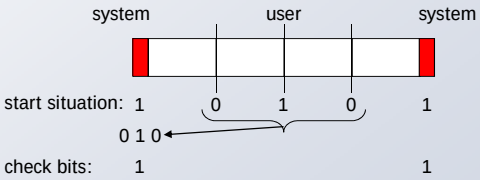


for sequential writes ok !

2. Problem: writes “out of sequence”



3. Solution: write of check bits in each sector



Mapping of Segments and Pages

- **Segment concept**
  - Enables deferred propagation
  - Allows to selectively introduce additional properties (attributes), e.g., to increase fault tolerance
  - Offers segments as units of locking, recovery, and access control
  - Preserves the advantages of the file concept when appropriately mapped to files
- **DB buffer management** (see chapter 4)
  - Fix and Unfix of pages in the DB buffer
  - Preparation of I/O requests to the file services
  - Optimization of replacement strategies
  - Support of segments of different types
  - New tasks: management of pages of variable size and of large objects

➥ **Division of logical DB address space in segments having visible page boundaries**

Mapping of Segments and Pages – Page Allocation Structures

- **Example: Operations at the upper interface**

- **Mapping function:**

page number ⟷ block number
main memory ⟷ external storage

- **Tasks:**

Replace an old block → PAM UPAMDAT, WR, . . .
Read block with page P_i → PAM UPAMDAT, RD, . . .

- **Properties of the upper interface**
  - **Linear address space** divided into pages of fixed size
  - Within a segment, the modules of the layer above refer to objects to be processed by page addresses (Fix operation) and offsets; a page fixed in the DB buffer can be directly manipulated (like in a virtual address space)
  - A DB segment is **non-volatile** (if not explicitly specified)

Realization of DBS

Mapping of files & blocks
File system
Mapping of blocks
Enhancing fault tolerance
Mapping of segments & pages
Shadow page mechanism
Differential files

Segment Types in a DBMS

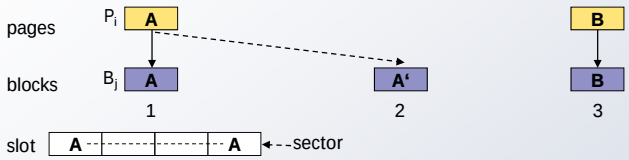
- **Classification of segment types**

properties \ segment-types	type 1	type 2	type 3	type 4	type 5
usage	public	private	private	private	private
life duration	permanent	permanent	permanent	permanent	temporary in a transaction
opening and closing	automatically by the system		explicitly by the user		
recovery in case of a failure	automatically by the system		explicitly by the user	no recovery mechanism	

- **Examples characterizing the use of segment types**
  - Type 1: catalog, schema information, log, all shared DB parts
  - Type 2: parts of the DB reserved for distinct users or user groups
  - Type 3: local copies of parts of the DB (views) for specific users (*snapshots*)
  - Type 4: auxiliary files for user programs
  - Type 5: temporary storage, e.g., for sort programs

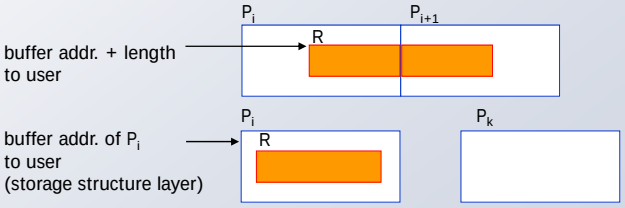
Why Visible Pages and Segments?

- **Explicit mapping on blocks and files enhances fault tolerance**



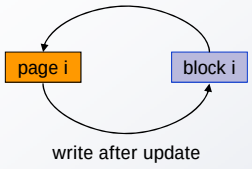
- **DB buffer having a record interface?**
  - **"Linear" memory mapping:** problems of invisible page boundaries
  - "Spanned record facility"
  - Neighbored pages must be adjacently allocated in the DB buffer

  ➥ Substantial mapping complexities and replacement problems



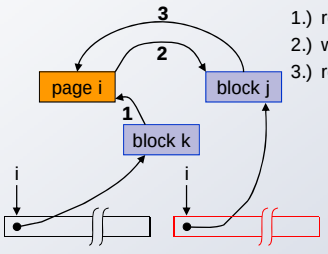
Methods for Page Allocation

- **Direct page allocation (*update-in-place*)**



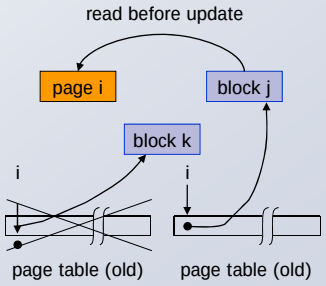
write after update

- **Indirect page allocation**



1.) read before update
2.) write after update
3.) read after update

page table (old) page table (new)



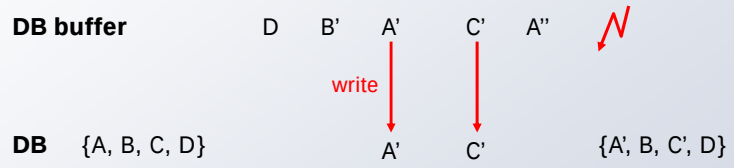
page table (old) page table (old)

➥ State after deferred propagation (set oriented)

Propagation of pages – DB Consistency after Crash

- **Direct propagation:** $P_i \Rightarrow B_j$, $P_{i'} \Rightarrow B_{j'}$

  ➥ Write = propagate



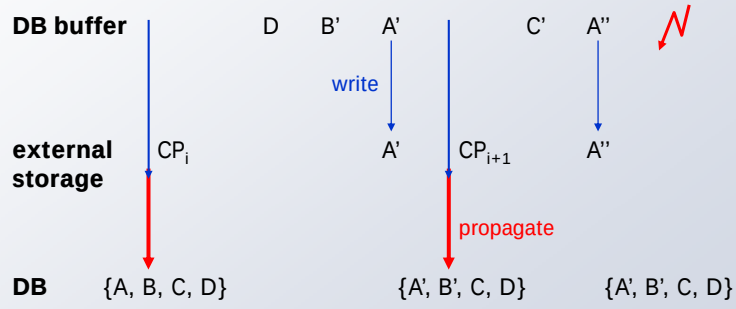
➥ DB after crash: **chaos consistent** !

For the description of the shadow page mechanism see:
Raymond A. Lorie: Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems (TODS) 2:1, 91-104 (1977)

Propagation of pages – DB Consistency after Crash (2)

- **Deferred propagation:** $P_i \Rightarrow B_j$, $j \neq k$, $P_{i'} \Rightarrow B_k$

  ➥ Write ≠ propagate

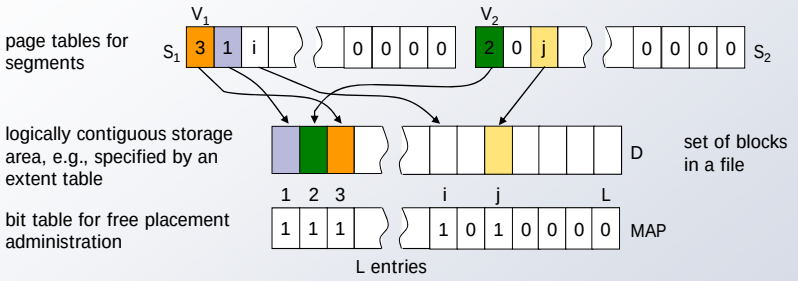


Suitable choice of checkpoints CP_i: operation boundaries to be considered!

➥ DB after crash: **action consistent** (API-, TA consistent) !

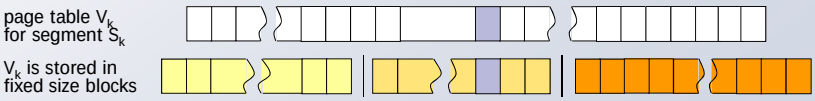
Storage Management for the Shadow Page Mechanism

- **Principle of indirect page allocation**



➥ Mapping of page no. → block no. using a page table

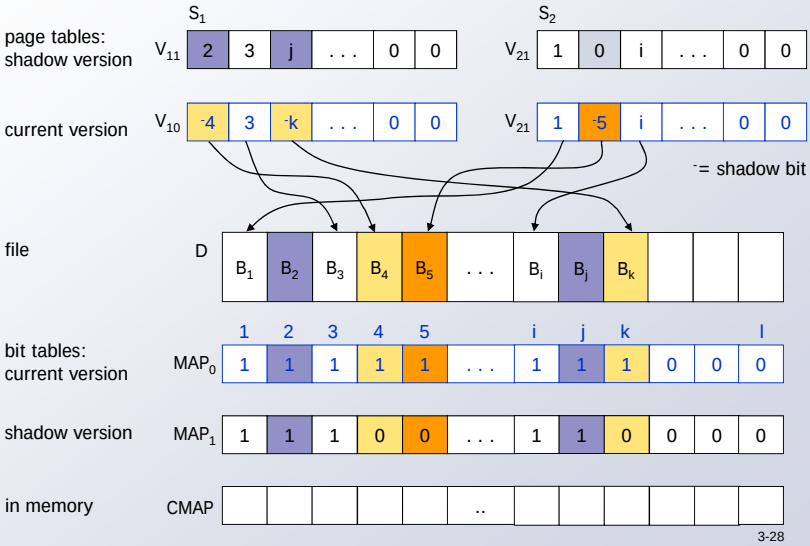
- **Decomposition of page table V_k into single blocks**



➥ Page table must be stored in linear address space and mapped onto blocks, too!

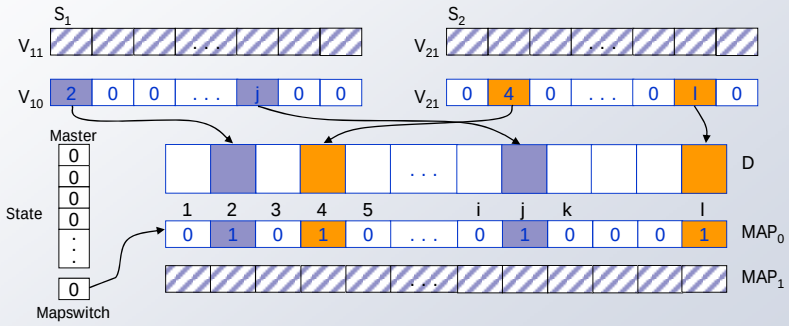
Shadow Page Mechanism – Principle

- Deferred propagation of updates



Shadow Page Mechanism

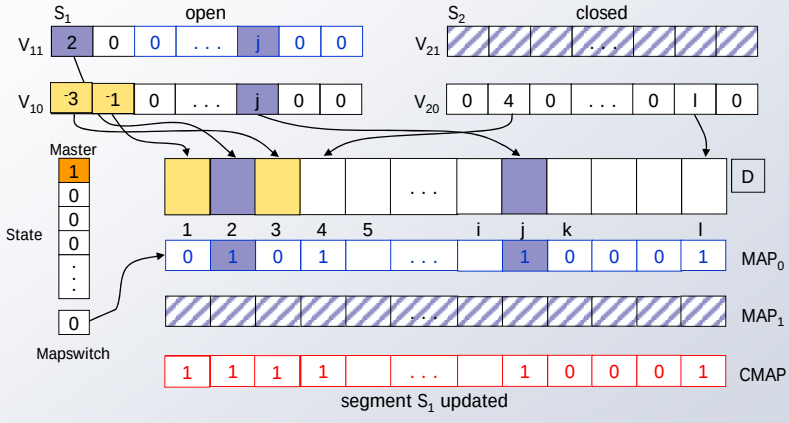
- **Application:** Implementation of fault tolerant systems
- Required storage areas and data structures
  the hatched structures are unused storage areas in the initial situation, which are only required after opening for update processing



- State (i) contains the opening state for Segment i (all segments closed).
- Mapswitch indicates which of the both tables (on an equal footing) Map0 and Map1 contains the current mapping of occupied blocks
- If a segment is closed, its content satisfies certain consistency conditions controlled by higher layers

Shadow Page Mechanism (2)

- **Processing of updates**



- On a set D of blocks, several segments can be opened at a time for update processing
- Map0, Map1 , and CMAP are related to the set D of blocks. CMAP contains for all segments the blocks occupied with shadow pages resp. current pages
- A page is only assigned to a new block after the first update after segment is opened

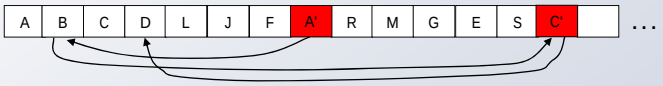
Shadow Page Mechanism – Functioning Principle

- If **segment k is to be opened for updates,** the following steps have to be executed:
  - Copy V_{k0} to V_{k1}
  - State(k) := 1
  - Write Master in an uninterruptible operation
  - Create a working copy CMAP of MAP_0 in main memory
- If **page P_i** is to be updated the first time since segment is opened, the following actions have to be executed:
  - Read page P_i from Block j = $V_{k0}(i)$
  - Find a free block j' in CMAP
  - $V_{k0}(i)$ = j'
  - Mark page P_i as updated in $V_{k0}(i)$
  - For further updates of P_i, block j' is used
- **Ending an update interval (creating a checkpoint):**
  - Create the bit list in MAP_1 reflecting the current storage occupancy (new blocks occupied, old ones released)
  - Write MAP_1 (no overwrite of MAP_0)
  - Write V_{k0}
  - Write all modified blocks
  - State(k) := 0, Mapswitch = 1 (MAP_1 is current)
  - Write Master in an uninterruptable operation
- Rollback of **opened segments** only requires to copy V_{k1} into V_{k0} and to set State(k) to 0

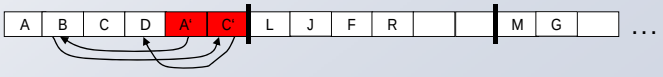
Shadow Page Mechanism (3)

- **Preservation of physical clustering**
  because of the dynamic reallocation of blocks, the shadow page mechanism cannot preserve in case of updates the physical contiguity of logically related pages
  Example: pages A, B, C, D be referenced in this sequence in all transactions:

Update of A → A' and C → C' leads to:



An improvement can be achieved by a division of the file into physical clusters of size p and the segments in logical clusters of size l. Then a logical cluster is mapped to and always contained in a physical cluster.
Example with p=6, l=4:



Using disks, a physical cluster is a cylinder, in general

Evaluation of the Shadow Page Mechanism

- **Advantages**
  - Rollback to most recent consistent state is very simple
  - More flexible propagation protocols for log files: buffering is possible until switching to the new state
  - Logical logging is possible, because an operation-consistent state is always available
  - In case of disastrous failures, a higher probability exists to reconstruct a useful DB state
- **Disadvantages**
  - Auxiliary structures (MAP and page table V_i) become so large that block decomposition is necessary
  - Page tables V_i occupy about 0.05-0.1% of the DB size, which, in case of large DBs (≥ n TB), may lead to a large share of page faults in the DB buffer provoked by accesses to V_i (argument may be weakened in future)
  - **Periodical checkpoints** enforce propagation of all modified pages in the DB buffer
  - **Physical clustering of** logically related pages is impaired resp. annihilated
  - Additional storage space for double occupancy: therefore, long batch (update) programs are not well supported

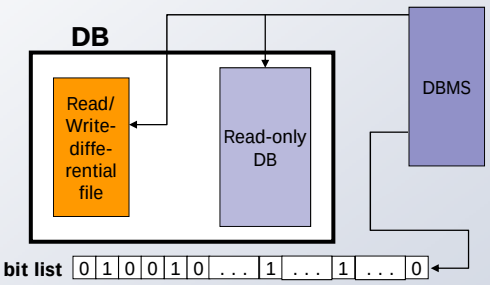
Differential File Method

- **Idea:**

  If a file is opened for update processing, an additional, temporary file is created, into which all modified blocks are written during the update interval. At the same time, the original file is not modified. At the end of the update interval, all modified blocks are copied from the temporary file to the original (permanent) file. Hence, the propagation of updates takes place in a deferred way.

  The essential problem is to decide during update processing for a given page no., whether the current version is in the temporary file or in the original file.

- **Principle:**



➥ Usage of a bit list indicating whether a page was **potentially** updated

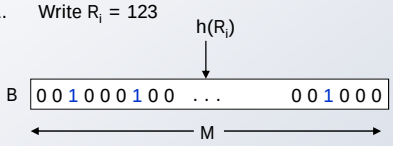
➥ Hash-based mapping achieves limitation of storage demand

Dennis G. Severance, Guy M. Lohman: Differential Files: Their Application to the Maintenance of Large Databases. ACM Trans. Database Syst. 1(3)): 256-267(1976)

Bloom Filter

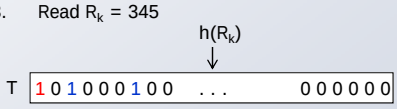
- **Mode of functioning**
  - Bit list B of size M in main memory
  - If a record is modified, its signature is mapped to B
  - Hash functions for record mapping address x bits which are set to`1´ in B
- **Location of record R_i**
  - Creation of the x characteristic bits in temporary bit list T
  - **AND** operation of T and B in Res
  - If all x bits in Res are set → PERHAPS
    otherwise → NO
- **Example**

1. Write R_i = 123



2. Write R_j = 456

3. Read R_k = 345



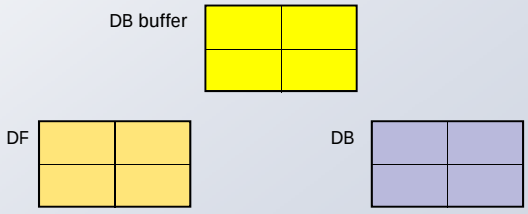
➥ If (B **AND** T) = T,
then answer **PERHAPS** !

Differential File Method* (3)

- **Example:**
  - Occupancy: DF = 5%, DB = 95% of actual records
  - Access cost C:

    1) C (DF+DB) =

    2) C (Bloom) =
- **Checkpoint:**



* Bloom Filter, originally developed in 1970 to support more efficient spelling and hyphenation of texts , were successfully applied to many computer science problems in recent year. They offer efficient and simple solutions to problems in query processing, data management, and routing in large networks (e.g., P2P networks, WWW, etc.) and, therefore, greatly demonstrate the advantages of randomized algorithms, that is to say simplicity and efficiency.

Summary

- **Storage allocation structures require efficient file concept**
  - Many files are of varying, not statically fixed size
  - Dynamic growth and shrinking is required
  - Permanent and temporary files
- **Recommended file properties:**
  - Direct and sequential block access
  - Block size should be defined on a file basis
  - Block allocation via dynamic extents

  ➥ Dynamic block allocation (in UNIX represented as a multi-level tree) not appropriate for large files
- **Segment concept**

  Enables the implementation of additional properties for DB processing (recovery, clustering for tables, etc.)
- **Two-level mapping**
  - Of segment/page onto file/block and these to slots on disk enables introduction of mapping redundancy through deferred propagation

Summary (2) and Literature

- **Deferred propagation strategies**
  - Are more expensive than direct ones, own, however, implicit fault tolerance
  - They burden normal processing in favor of recovery
- **Direct propagation strategy (*update-in-place*)**
  - Simple implementation
  - No additional costs for page allocation at runtime
  - Fault tolerance only through explicit logging&recovery functions

- Lorie, R.: Physical Integrity in a Large Segmented Database. ACM Transactions on Database Systems (TODS) 2(1): 91-104 (1977)
- Rahm, E.: Hochleistungs-Transaktionssysteme, Vieweg, 1993 , Ch. 5 and 6, http://lips.informatik.uni-leipzig.de/pub/1993-2
- Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System. ACM TODS 10(1): 26-52 (1992)