These slides are for use with

Database Systems

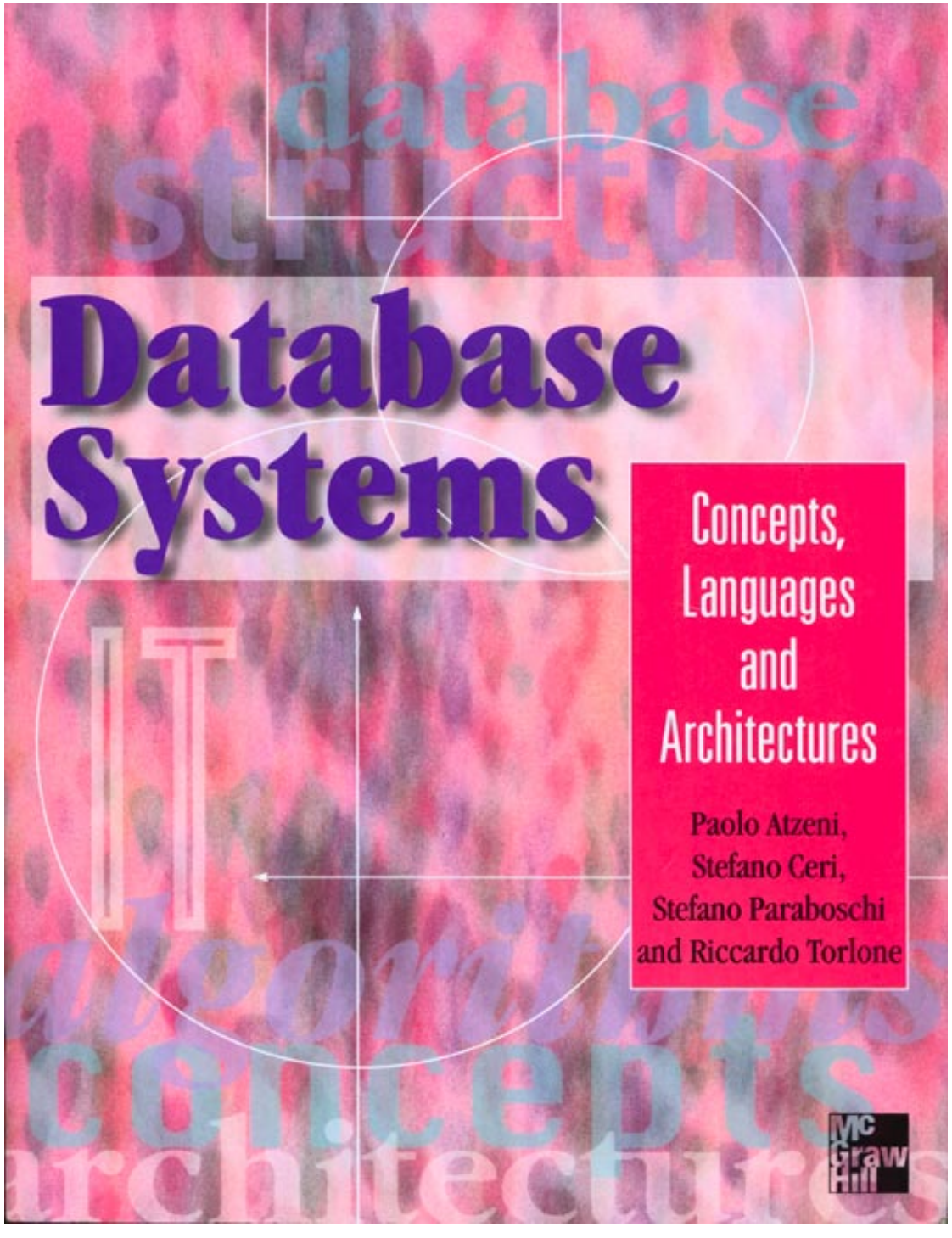
Concepts, Languages and Architectures

Paolo Atzeni • Stefano Ceri • Stefano Paraboschi • Riccardo Torlone

To view these slides on-screen or with a projector use the arrow keys to move to the next or previous slide. The return or enter key will also take you to the next slide. Note you can press the 'escape' key to reveal the menu bar and then use the standard Acrobat controls — including the magnifying glass to zoom in on details.

To print these slides on acetates for projection use the escape key to reveal the menu and choose 'print' from the 'file' menu. If the slides are too large for your printer then select 'shrink to fit' in the print dialogue box.

Press the 'return' or 'enter' key to continue . . .



Chapter 9

Technology of a database server

Click here for help

Technology of database servers

- Components:
  - Optimizer - selects the data access strategy
  - Access Methods Manager - executes the strategy
    - RSS (Relational Storage System)
    - OM (Object Manager)
  - Buffer Manager - manages page accesses
  - Reliability Manager - manages faults
  - Concurrency Control Manager - manages the interference due to multi-user data access

Approach

- Top-Down
- From theory to systems
- Centered on the notion of transaction
- Valid for both relational and object-oriented systems

Definition of a Transaction

- An elementary unit of work performed by an application, with specific features for what concerns correctness, robustness and isolation
- Each transaction is encapsulated within two commands
  - begin transaction (`bot`)
  - end transaction (`eot`)
- Within a transaction, one of the commands below is executed (exactly once)
  - `commit work` (commit)
  - `rollback work` (abort)
- Transactional System: a system capable of providing the definition & execution of transactions on behalf of multiple, concurrent applications

(Abstract) Example of Transaction

```
begin transaction
x := x - 10
y := y + 10
commit work
end transaction
```

Well-formed Transaction

- A transaction starting with `begin transaction`, ending with `end transaction`, in whose execution just one of the two commands `commit work` or `rollback work` is executed, and without data management operations processed after the execution of the `commit work` or `rollback work`

ACID Properties of Transactions

- ACID is an acronym for:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- A transaction is an atomic unit of work
- It cannot leave the database in an intermediate state:
  - a fault or error prior to commit causes the UNDO of the work made earlier
  - A fault or error after the commit may require the REDO of the work made earlier, if its effect on the database state is not guaranteed
- Possible behaviors
  - Commit = normality (99.9%)
  - Rollback requested by the application = suicide
  - Rollback requested by the system = murder

Consistency

- Consistency amounts to requiring that the transaction does not violate any integrity constraint
- Integrity constraint verification can be:
  - Immediate: during the transaction (the operation causing the violation is rejected)
  - Deferred: at the end of the transaction (if some integrity constraint is violated, the entire transaction is rejected)

Isolation

- Isolation requires that any transaction executes independently from the execution of all other concurrent transactions
  - [isolation requires that the concurrent execution of a collection of transaction yields to the same result as an arbitrary sequential execution of the same transactions]

Durability (Persistence)

- Durability requires that the effect of a transaction that has successfully committed be not lost (the effect will “last forever”)

Transactions and system modules

- Atomicity and durability are guaranteed by the Reliability Control System
- Isolation is guaranteed by the Concurrency Control System
- Consistency is managed during the normal query execution by the DBMS System (verification activities are generated by the DDL Compilers and executed during query processing)

Concurrency control

- Concurrency: highly desired
- Measured in *tps* (transactions per second) - typical values are tens to hundreds to thousands tps
- Typical examples: banks, airline reservation systems

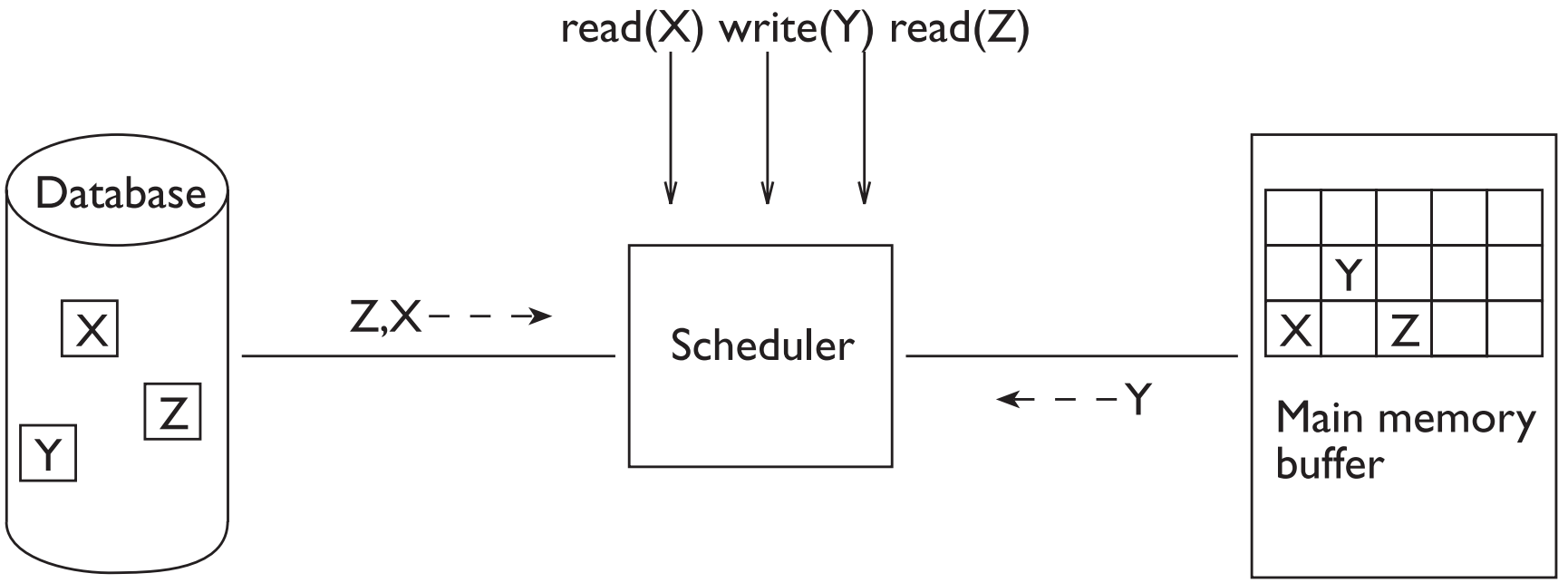
Architecture

- Input-output operations on abstract objects *x*, *y*, *z*
- Each input-output operation reads secondary memory blocks into buffer pages or writes buffer pages into secondary memory blocks
- For simplicity: one-to-one mapping from blocks to pages

Problem

- Anomalies due to concurrent execution

Architecture of the concurrency control system



Anomaly 1: Update loss

- Consider two identical transactions:
  - *t* 1 : *r*(x), *x* = *x* + 1, *w*(x)
  - *t* 2 : *r*(x), *x* = *x* + 1, *w*(x)
- Assume initially *x*=2; after serial execution *x*=4
- Consider concurrent execution:
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
$x = x + 1$	
	`bot`
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
`commit`	
	$w_2(x)$
	`commit`

- One update is lost, final value *x*=3

Anomaly 2: Dirty read

- Consider the same two transactions, and the following execution (note that the first transaction fails):
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	`bot`
	$r_2(x)$
	$x = x + 1$
`abort`	
	$w_2(x)$
	`commit`

- Critical aspect: t_2 reads an intermediate state (dirty read)

Anomaly 3: Inconsistent read

- t_1 repeats two reads:
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$r_1(x)$	
`commit`	

- t_1 reads different values for x

Anomaly 4: Ghost update

- Assume the integrity constraint $x + y + z = 1000$;
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = x + y + z$	
`commit`	

- In the end, $s = 110$: the integrity constraint is not satisfied
- t_1 sees a ghost update

Concurrency control theory

- Transaction: sequence of read or write actions
- Each transaction has a unique, system-assigned transaction identifier
- Each transactions is initiated by the `begin transaction` command and terminated by `end transaction` (omitted)
- Example: $t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$

Notion of Schedule

- Represents the sequence of input/output operations presented by concurrent transactions
- Example: $S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$
- Simplifying assumption: we consider a *commit-projection* and ignore the transactions that produce an abort, removing all their actions from the schedule
- This assumption is not acceptable in practice and will be removed

Foundations of concurrency control

- *Objective*: refuse the schedules that cause the anomalies
- *Scheduler*: a system that accepts or rejects the operations requested by transactions
- *Serial schedule:* one in which the actions of all the transactions appear in sequence

  $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$
- *Serializable schedule:* one that produces the same result as some serial schedule S_j of the same transactions
  - Requires a notions of equivalence between schedules
  - Progressive notions: *view-equivalence*, *conflict-equivalence*, *two-phase locking*, *timestamp-based*
- Observation: a scheduler allows the identification of a more or less wide-ranging class of acceptable schedules at the cost of testing for equivalence

View-Serializability

- Preliminary definitions:
  - $r_i(x)$ *reads-from* $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
  - $w_i(x)$ in a schedule S is a *final write* if it is the last write of the object x to appear in S
- V*iew-equivalent schedules* ($S_i \approx_V S_j$): if they possess the same *reads-from* relation and the same final writes
- A schedule is called *view-serializable* if it is view-equivalent to some serial schedule
- The set of view-serializable schedules is called VSR

View-Serializability

- Complexity of view-serializability:
  - Deciding on the view-equivalence of two given schedules: done by a polynomial algorithm
  - Deciding on the view serializability of a generic schedule: NP-complete problem

Examples of view serializability

- $S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  $S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  $S_5 : w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  $S_6 : w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$
  - S_3 is view-equivalent to the serial schedule S_4 (thus, it is view-serializable)
  - S_5 is not view-equivalent to S_4, but it is view-equivalent to the serial schedule S_6, and thus this also is view-serializable
- $S_7 : r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$
  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
  $S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$
  - S_7 corresponds to update loss, S_8 to inconsistent reads and S_9 to ghost updates; they are not view-serializable

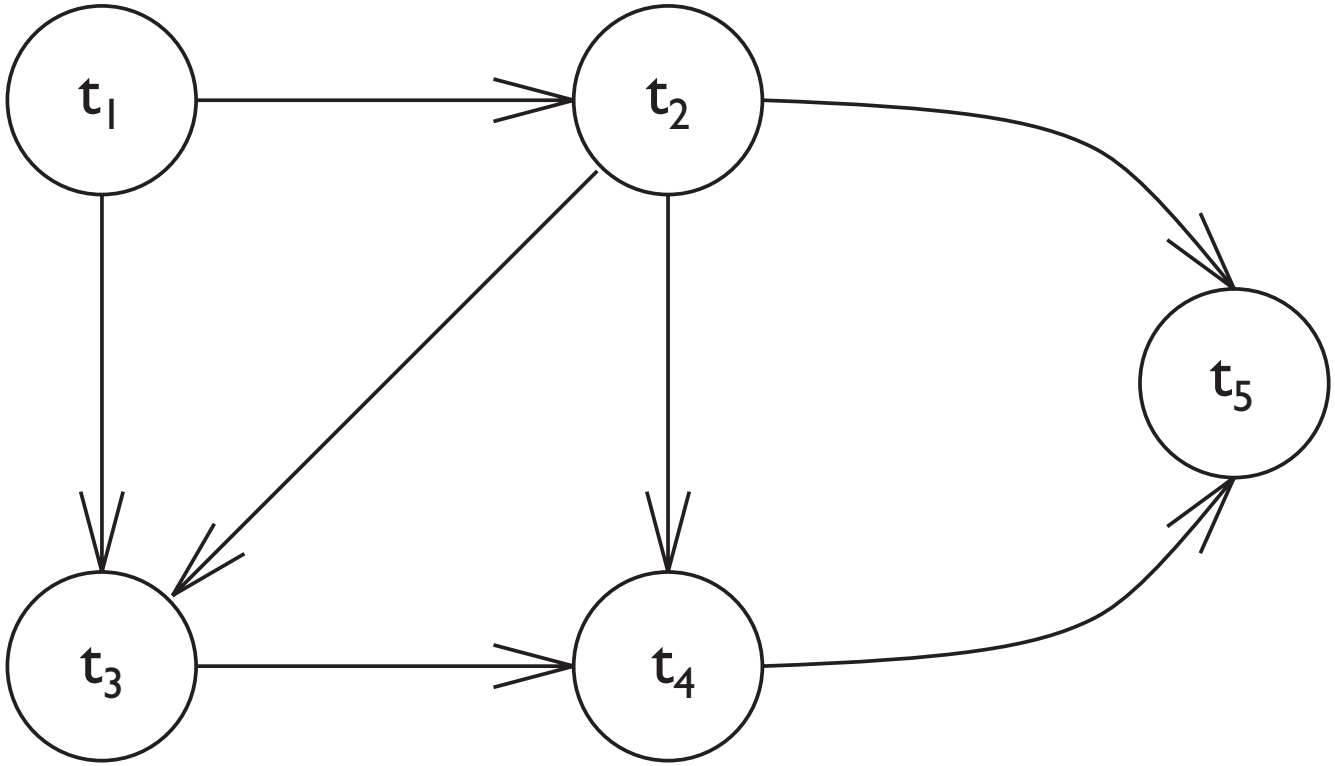
Conflict-serializability

- Preliminary definition:
  - Action a_i is in *conflict* with a_j (i ¹ j), if both operate on the same object and at least one of them is a write. Two cases:
    - *read-write* conflicts (*rw* or *wr*)
    - *write-write* conflicts (*ww*).
- *Conflict-equivalent schedules* ($S_i \approx_C S_j$): if they present the same operations and each pair of operations in conflict is in the same order in both the schedules
- A schedule is therefore *conflict-serializable* if there is a serial schedule that is conflict-equivalent to it. The set of conflict-serializable schedules is called CSR
- CSR is properly included into VSR

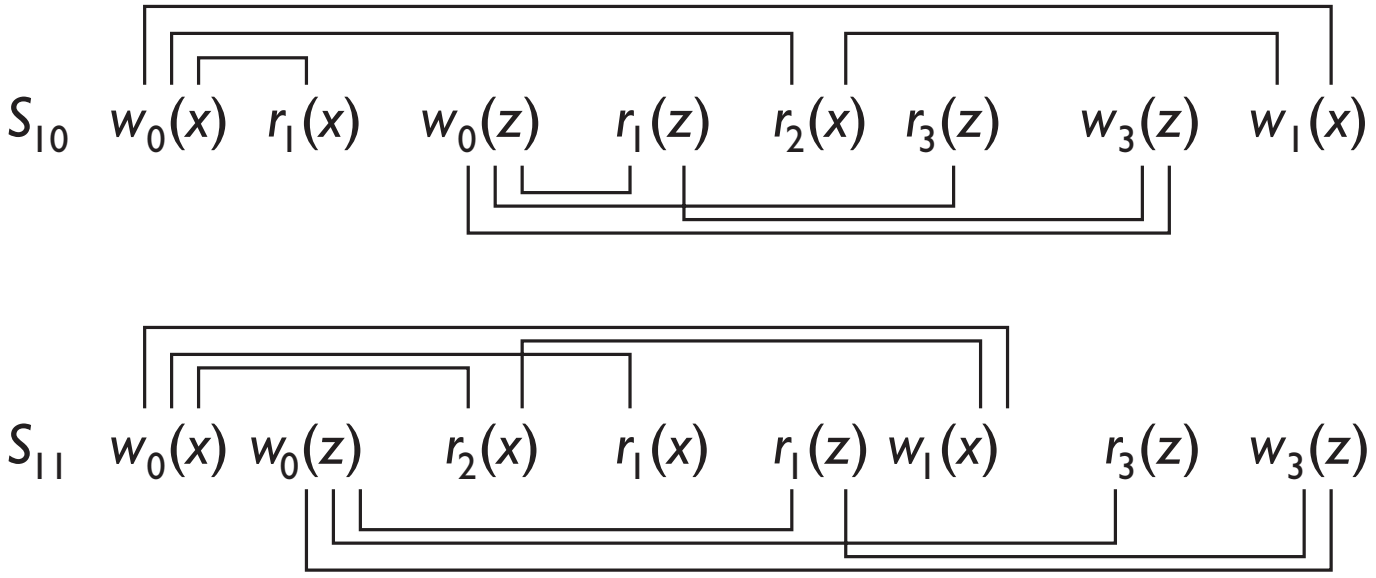
Testing conflict-serializability

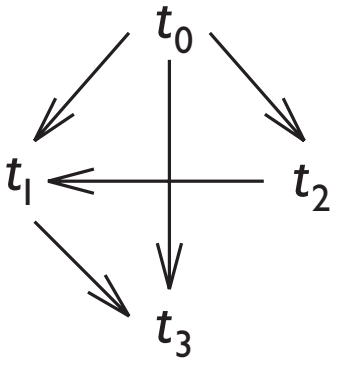
- By means of the *conflict graph*, with;
  - a node for each transaction t_i
  - an edge from t_i to t_j if there is at least one conflict between an action a_i and an action a_j such that a_i precedes a_j
- A schedule is in CSR if and only if the graph is acyclic
- Testing for cyclicity of a graph has a linear complexity with respect to the size of the graph itself
- Conflict serializability is still too laborious in practice, especially with data distribution

Conflict graph for a schedule



A schedule S_{10} conflict-equivalent to a serial schedule S_{11}





Two-phase locking

- Used by almost all commercial DBMSs
- Principle:
  - All read operations preceded by *r_lock* (shared lock) and followed *unlock*
  - All write operations preceded by *w_lock* (exclusive lock) and followed *unlock*

  A transaction following these rules is *well formed wrt locking*
- When a transaction first reads and then writes an object it can:
  - Use a write lock
  - Move from shared lock to exclusive lock (*lock escalation*)
- The *lock manager* receives these primitives from transactions and grants resources according to conflict table
  - When the lock is granted, the resource is acquired
  - At the unlock, the resource is released

Behavior of the lock manager

- Based on conflict table:

Request	**Resource state**		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

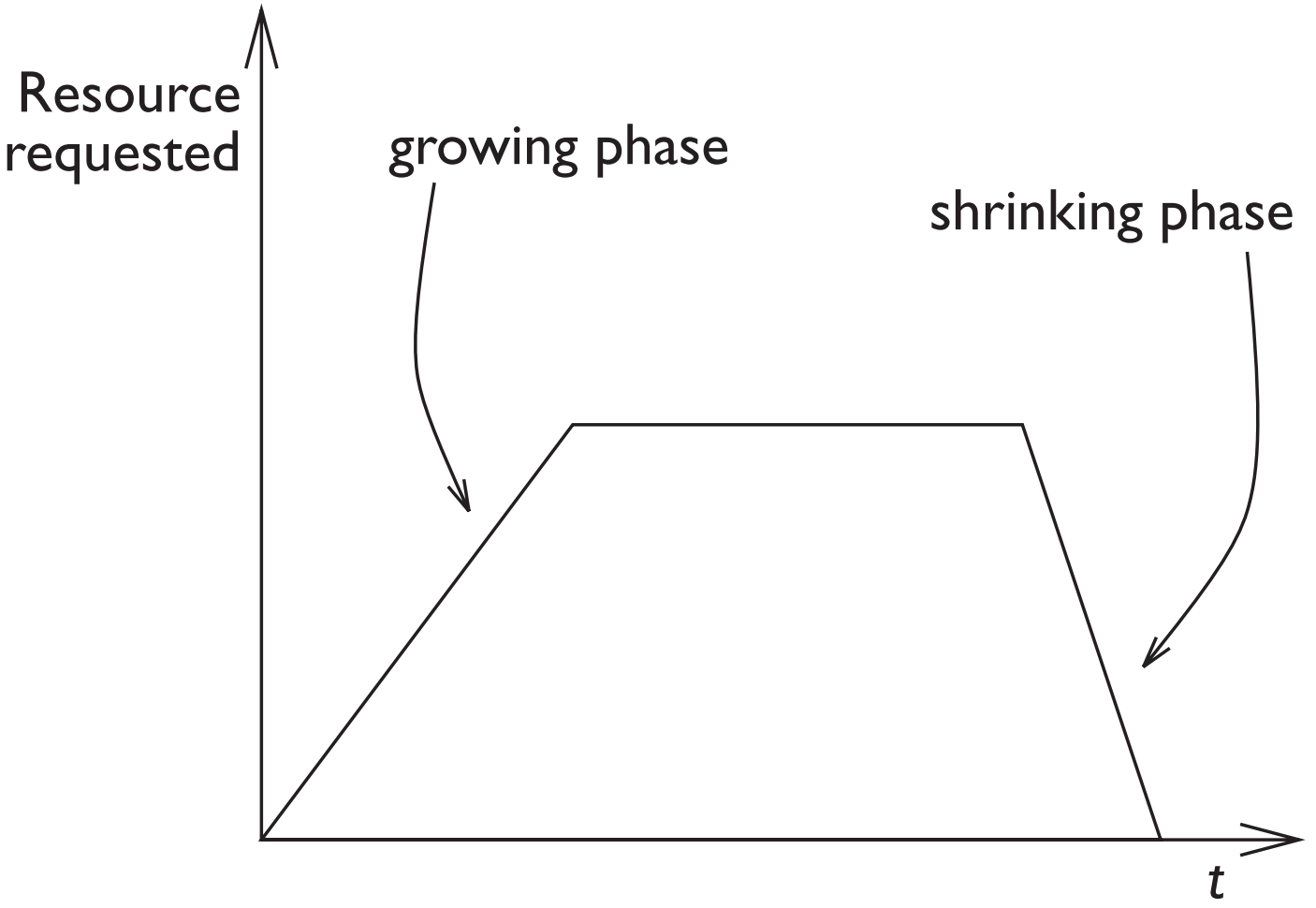
  - A counter keeps track of the number of readers; the resource is released when the counter is set to zero.
- If a lock request is not granted, the requesting transaction is put in a waiting state
- The waiting ends when the resource is unlocked and becomes available
- The locks already granted are stored in a lock table, managed by the lock manager

Two-phase locking

- Serializability requires in addition that locking is in two phases: *A transaction, after having released a lock, cannot acquire other locks*
- Two phases: growing and shrinking
- If a scheduler use well-formed transaction, conflict-based lock granting, and two-phases, then it produces the class of 2PL schedules
- 2PL schedules are serializable and strictly included into CSR
- Example of a schedule that is in CSR and not in 2PL:

  $S_{12} : r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$

Representation of allocated resources for 2PL



Strict two-phase-locking

- We still need to remove the hypothesis of using a commit-projection
- To do so, turn to strict 2PL (by adding a constraint):
  The locks on a transaction can be released only after having carried out the commit/abort operations
- This version is used by commercial DBMSs; it eliminates the dirty read anomaly

Concurrency control based on timestamps

- New mechanism based on *timestamp*:
  - an identifier that defines a total ordering of temporal events within a system
- Every transaction is assigned a timestamp *ts* that represents the time at which the transaction begins
- A schedule is accepted only if it reflects the serial ordering of the transactions based on the value of the timestamp of each transaction

Basic timestamp mechanism

- Each scheduler has a counter RTM(*x*) and WTM(*x*) for each object
- Each scheduler receives timestamped read and write requests upon objects:
  - *read*(*x*,*ts*): if *ts* < WTM(x) then the request is rejected and the transaction is killed, otherwise the request is accepted and RTM(*x*) is set equal to the greater of RTM(*x*) and *ts*
  - *write*(*x*,*ts*): if *ts* < WTM(*x*) or *ts* < RTM(*x*) then the request is rejected and the transaction is killed, otherwise the request is accepted and WTM(*x*) is set equal to *ts*

  Under assumption of commit-projection
- The method causes the forced abort of a large number of transactions
- To remove the commit-projection assumption, must buffer writes until commit, and this introduces waiting of transactions

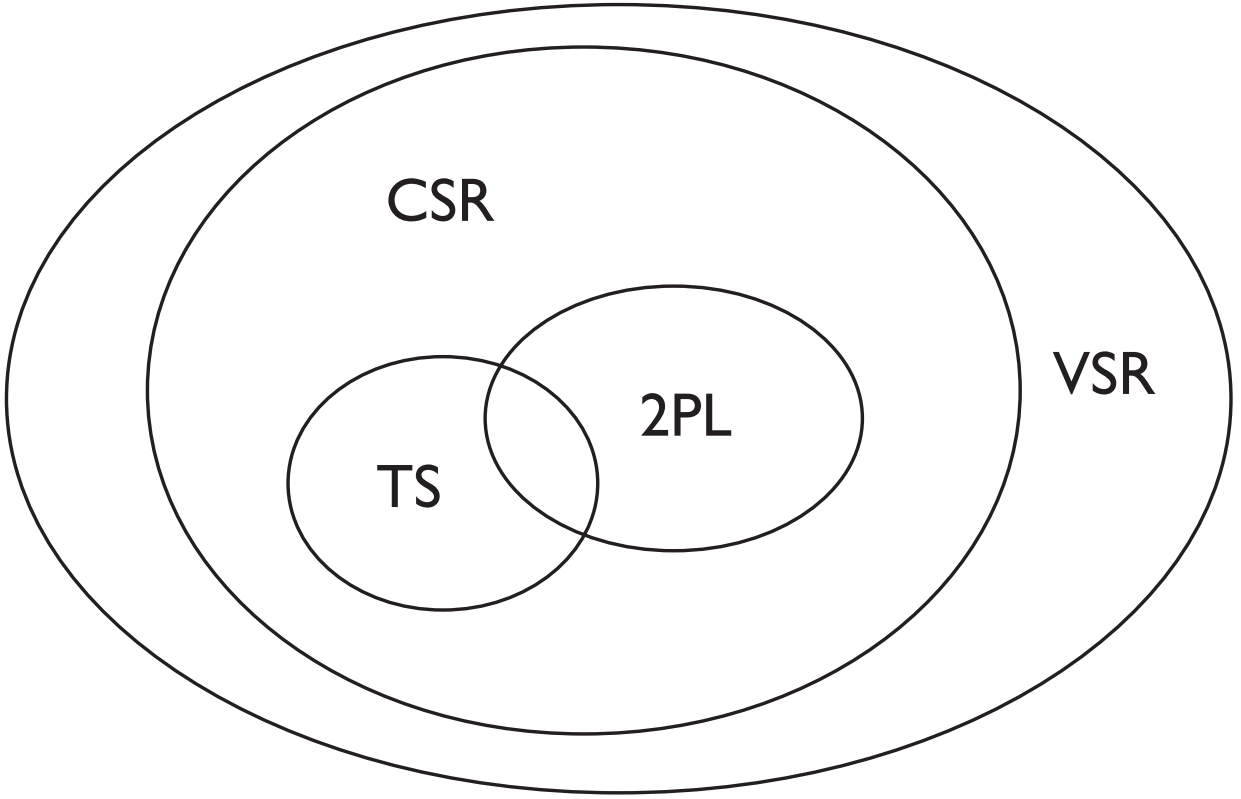
Example of timestamp-based concurrency control

Request	Response	New value
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no	t_8 killed
write(x,11)	ok	WTM(x) = 11
read(x,10)	no	t_{10} killed

Multiversion concurrency control

- Main idea: writes generate new copies, reads make access to the correct copy
- Writes generate new copies each with a WTM. At any time, $N > 1$ copies of each object x are active, with $WTM_N(x)$. There is only one global $RTM(x)$
- Mechanism:
  - *read*(x,*ts*): is always accepted. x_k selected for reading such that: if $ts > WTM_N(x)$, then $k = N$, otherwise k is taken such that $WTM_k(x) < ts < WTM_{k+1}(x)$
  - *write*(x,*ts*): if $ts < RTM(x)$ the request is refused, otherwise a new version of the item of data is added (*N* increased by one) with $WTM_N(x) = ts$
- Old copies are discarded when there are no read transactions interested in their values

Taxonomy of the classes of schedule accepted by the methods VSR, CSR, 2PL and TS



Comparison 2PL vs TS

- In 2PL the transactions are put in waiting. In TS they are killed and then restarted
- The serialization order in 2PL is imposed by conflicts, while in TS it is imposed by the timestamps
- The necessity of waiting for the commit of the transaction causes strict 2PL and buffering of writes in TS
- 2PL can give rise to deadlocks (discussed next)
- Restarting costs more than waiting: 2PL wins!

Lock management

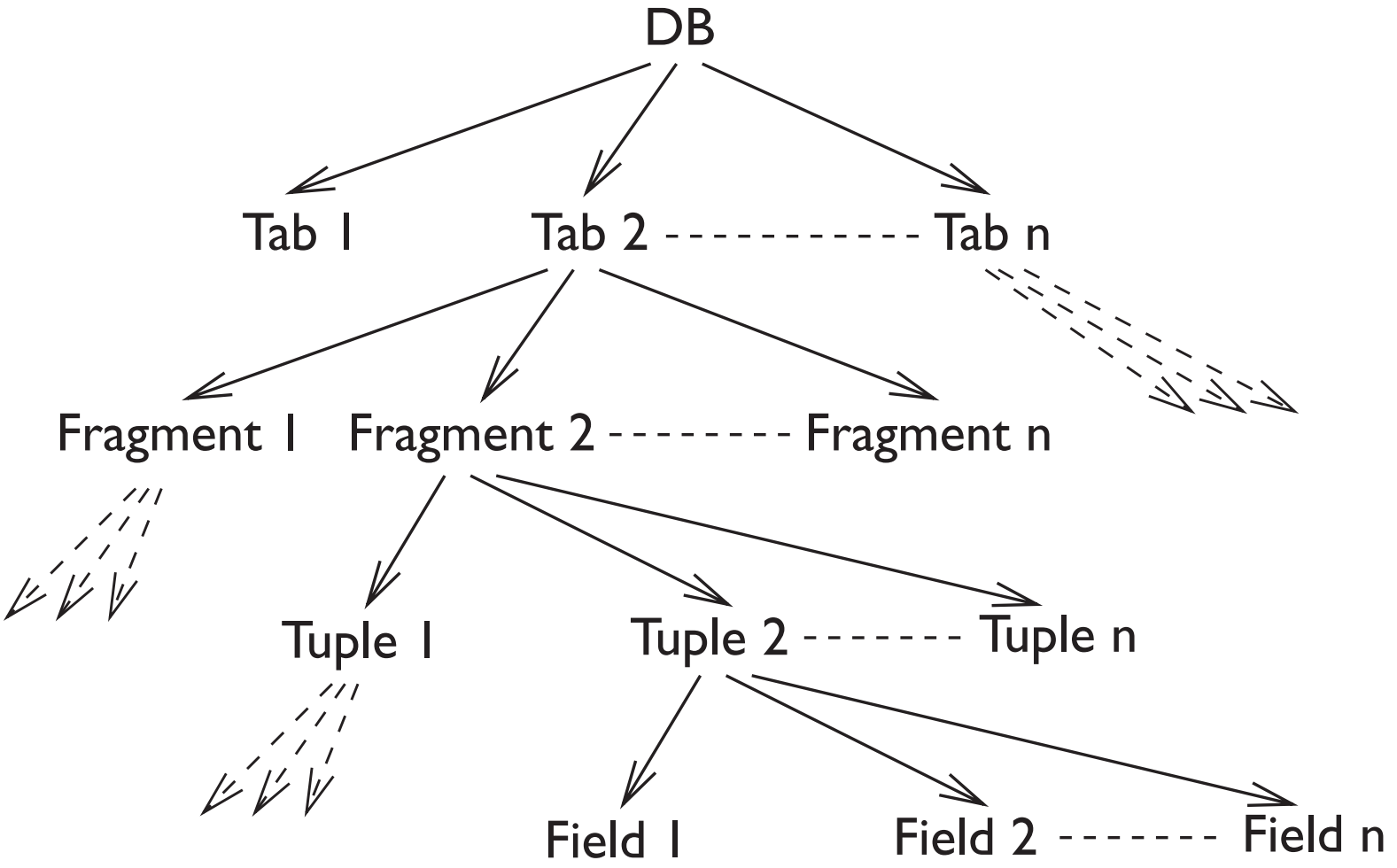
- Interface:
  - `r_lock(T, x, errcode, timeout)`
  - `w_lock(T, x, errcode, timeout)`
  - `unlock(T, x)`

    `T`: transaction identifier
    `X`: data element
    `timeout`: max wait in queue
- If timeout expires, `errcode` signals an error, typically the transaction rolls back and restarts

Hierarchical locking

- In many real systems locks can be specified at different granularity, e.g. tables, fragments, tuples, fields. These are organized in a hierarchy (possibly a directed acyclic graph)
- 5 locking modes:
  - 2 are shared and exclusive, renamed as:
    - XL: exclusive lock
    - SL: shared lock
  - 3 are new:
    - ISL: intention shared lock
    - IXL: intention exclusive lock
    - SIXL: shared intention-exclusive lock
- The choice of lock granularity is left to application designers;
  - too coarse: many resources are blocked
  - too fine: many locks are requested

The hierarchy of resources



Hierarchical locking protocol

- Locks are requested from the root to descendents in a hierarchy
- Locks are released starting at the node locked and moving up the tree
- In order to request an SL or ISL on a node, a transaction must already hold an ISL or IXL lock on the parent node
- In order to request an IXL, XL, or SIXL on a node, a transaction must already hold an SIXL or IXL lock on the parent node
- The new conflict table is shown in the next slide

Conflicts in hierarchical locking

Request \ Resource state	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Lock options offered by SQL2

- Some transactions are defined as `read-only` (they can't request exclusive locks)
- The level of isolation can be set for each transactions
- `Serializable` guarantees max isolation: keeps predicate locks so as not to change the content even of aggregate functions evaluated on data sets
- `Repeatable read` is equal to strict 2PL (note: repeated reading of values are the same, but repeated readings of aggregates over data are not)
- `Committed read` excludes the reading of intermediate states (uncommitted data)
- `Uncommitted read` does no concurrency control at all on read

Deadlocks

- Created by concurrent transactions, each of which holds and waits for resources held by others
- Example:
  - t_1: *read*(x), *write*(y)
  - t_2: *read*(y), *write*(x)
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $read_1(x)$, $read_2(y)$ $w\_lock_1(y)$, $w\_lock_2(x)$
  - This is deadlock!
- Deadlock probability grows linearly with number of transactions and quadratically with the number of lock requests of each transaction (under suitable uniformity assumptions)

Deadlock resolution techniques

- A deadlock is a cycle in the *wait-for* graph which indicates wait conditions between transactions
  - (node=transaction, arc=wait condition).
- Three techniques:
  1. Timeout (problem: choice of timeout with trade-offs)
  2. Deadlock detection
  3. Deadlock prevention
- Deadlock detection: performs the search for cycles in a wait-for graph
- Deadlock prevention: kills the transactions that could cause a cycle (thus: it overkills)
  - Options for choosing the transaction to kill:
    - via *pre-emptive* policies or *non-pre-emptive* policies

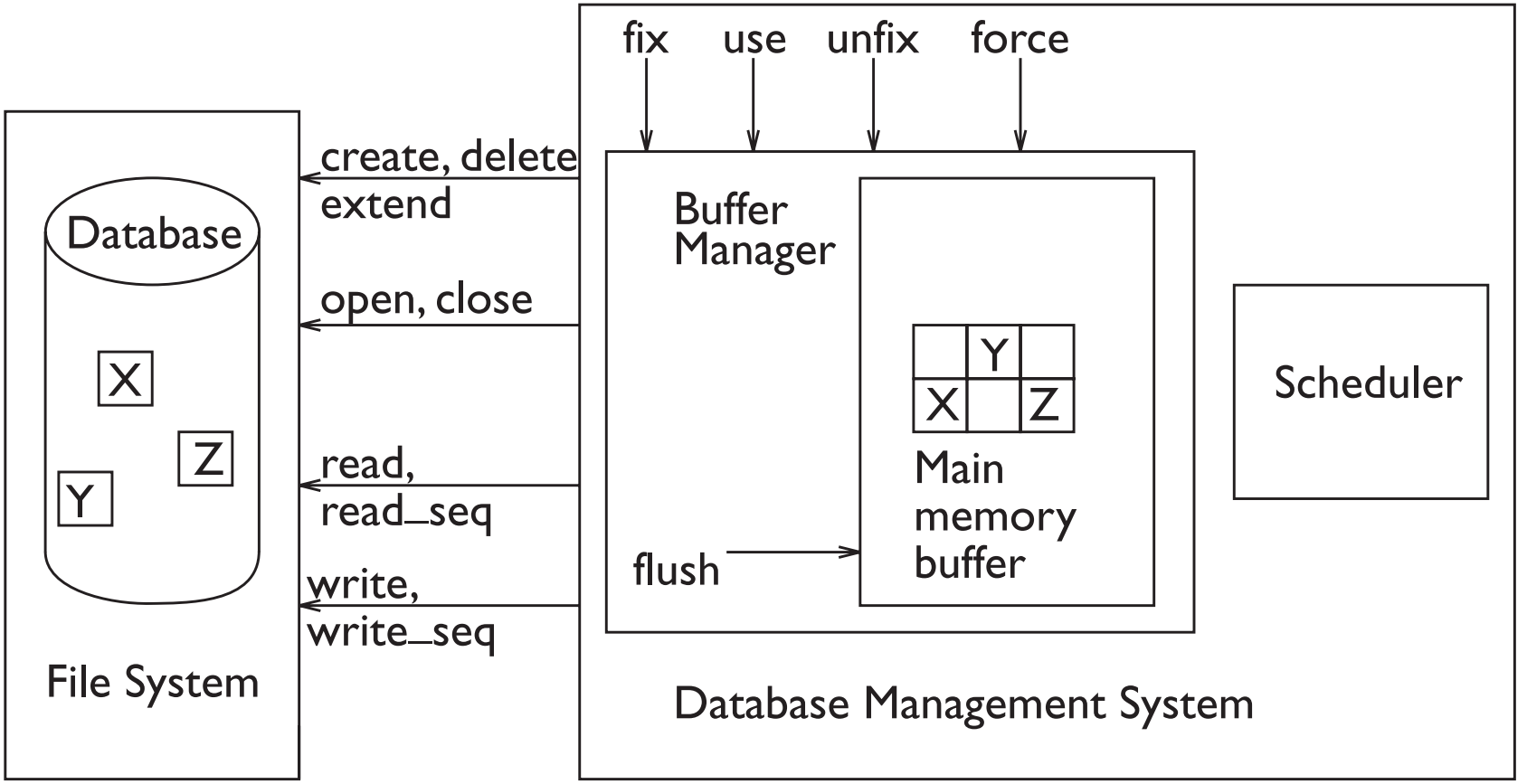
Buffer management

- Buffer: a large area of the main memory pre-allocated to the DBMS and shared among the various transactions
- The buffer is organized in pages, of a size equal or multiple of the input/output blocks used by the operating system;
  - Size of pages: from a few Kbytes to about a hundred Kbytes
  - Access times to main memory: six orders of magnitude faster than access times to secondary memory
- It can possibly store the entire database (we call it a *main-memory resident database*)

Buffer manager organization

- Provides primitives *fix*, *use*, *unfix*, *flush* and *force*. Manages input/output operations in response to these primitives
- The policies of buffer management are similar to those of main memory management by the operating systems, subject to:
  - Principle of *data locality*: currently referenced data has a greater probability of being referenced in the future
  - Empirical law: only 20% of data is typically accessed by 80% of applications

Architecture of the buffer manager



Primitives for buffer management

- *fix:* used to load a page into the buffer, requires read operations from the secondary memory only when the chosen page is not already resident in the buffer
  - After the operation, the page is loaded and *vali*d, that is, allocated to an active transaction; a pointer to the page is returned to the transaction
- *use*: used by the transaction to gain access to the page previously loaded in the memory, confirming its allocation in the buffer and its status as a valid page
- *unfix:* indicates that the transaction has terminated the use of the page, which is no longer valid
- *force*: synchronously transfers a page from the buffer manager to the secondary memory
- *flush*: Transfers invalid pages to secondary memory asynchronously and independently of the active transactions

Management of the *fix* primitive

- Search in the buffer for requested page, if found the address is returned
- Selection of a free page in the buffer, read of page from secondary memory, the page address is returned
- Selection of a non free page in the buffer, called *victi*m:
  - only among non-valid pages (*fix* can fail when the search fails)
  - also among valid pages allocated to other transactions
- Victim is rewritten to secondary memory invoking the *flush* operation, then the page is read from secondary memory and the page address is returned

Buffer management policies

- The *steal* policy allows the buffer manager to select an active page allocated to another transaction as a victim
- The *no-steal* policy excludes this possibility
- The *force* policy requires that all the active pages of a transaction are written to secondary memory before committing
- The *no-force* policy entrusts the writing of the pages of a transaction to the buffer manager
- The *no-steal/no-force* pair of policies is preferred by the DBMSs
- There is also the possibility of 'anticipating' the loading and unloading times of the pages, by means of *pre-fetching* and *pre-flushing* policies

Relationship between buffer manager and file system

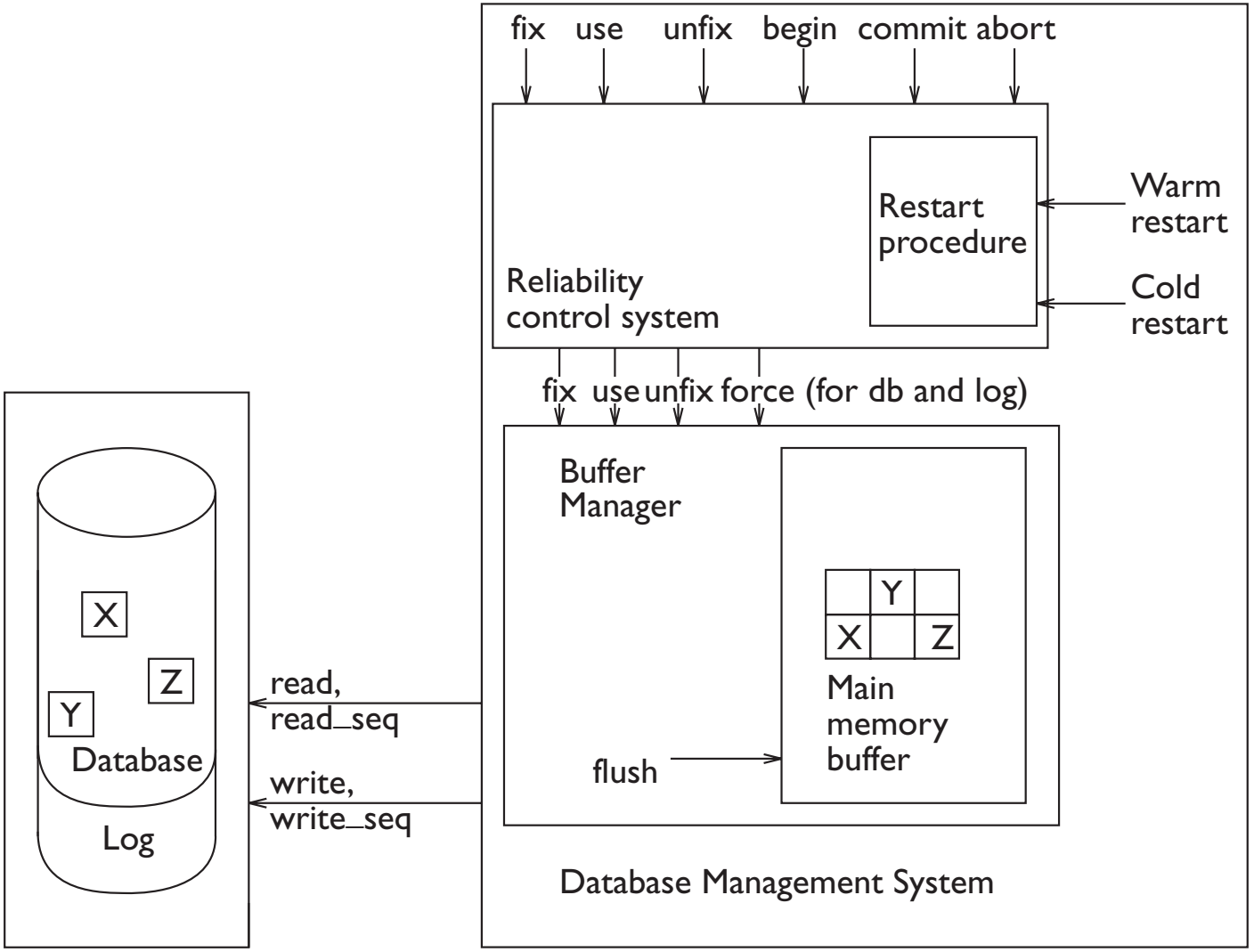
- The file system is responsible for knowing the structure of the secondary memory in directories and the current situation of secondary memory use. It must identify which blocks are free and which are allocated to files
- DBMSs use the file system for the following functions:
  - The creation (`create`) and removal (`delete`) of a file
  - The opening (`open`) and closing (`close`) of a file.
  - `read(fileid,block,buffer)` for the direct access to a block of a file which is transcribed to the buffer page.
  - `read_seq(fileid,f-block,count,f-buffer)` for sequential access to a fixed number (`count`) of blocks of a file
  - The dual primitives `write` and `write_seq`

Reliability control

- Responsible for executing the transactional commands:
  - `begin transaction` (B)
  - `commit work` (C)
  - `rollback work` (A, for abort)

  and the primitives for recovery after malfunctions:
  - *warm restart* and *cold restart*
- Ensures atomicity and durability
- Uses as main data structure the *log*:
  - A permanent archive which registers the various actions carried out by the DBMS
  - Two metaphors: Arianna's thread, Hansel and Gretel's crumbs of bread

Architecture of the reliability control system



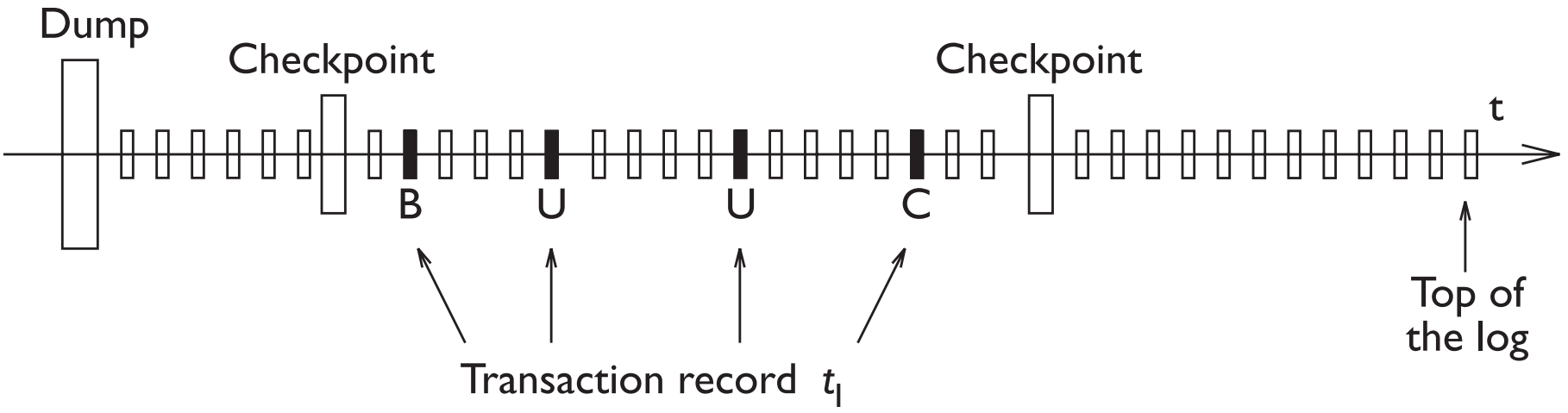
Stable memory

- A memory that is failure-resistant
- It is an abstraction, in that no memory can have zero probability of failure (but replication and robust writing protocols can bring such a probability close to zero)
- A failure of stable memory is assumed as *catastrophic* and impossible, at least in this context
- Organized in different ways depending on the criticality of the application:
  - a tape unit
  - a pair of devices of different kind (e.g.: a tape and a disk)
  - two mirrored disk units

Log organization

- The log is a sequential file managed by the reliability control system, written in the stable memory
- The actions carried out by the various transactions are recorded in the log in chronological order (written sequentially to the top block)
- There are two types of log record
  - *Transaction records*
    - `begin`, B(T)
    - `insert`, I(T,O,AS)
    - `delete`, D(T,O,BS)
    - `update`, U(T,O,BS,AS)
    - `commit`, C(T), or `abort`, A(T)
  - *System records*
    - `dump` (rare)
    - `checkpoint` (more frequent)

Description of a log



Undo and redo

- Undo of an action on an object *O:*
  - `update`, `delete`*:* copy the value *BS* into the object *O*
  - `insert`: delete the object *O*
- Redo of an action on an object *O:*
  - `insert`, `update`: copy the value *AS* into the object *O*
  - `delete`: re-insert the object *O*
- *Idempotence* of *undo* and *redo:* an arbitrary number of undos and redos of the same action is equivalent to the carrying out of such actions only once:
  - *undo*(*undo*(*A*)) = *undo*(*A*)
  - *redo*(*redo*(*A*)) = *redo*(*A*)

Checkpoint

- A *checkpoint* is carried out periodically, recording active transactions and updating secondary memory relative to all completed transactions
  - After having initiated a checkpoint, no commit operations are accepted by the active transactions
  - The checkpoint ends by synchronously writing (*forcing*) a `checkpoint` record CK(T_1,T_2,..T_n), which contains the identifiers of the active transactions
  - In this way effects of the transactions that have carried out a commit are permanently recorded in the database

Dump

- A *dump* is a complete copy of the database, which is normally created when the system is not operative
  - The copy is stored in the stable memory, typically on tape, and is called *backup*
  - A dump record *DUMP* in the *log* signals the presence of a backup made at a given time and identifies the file or device where the dump took place

Transactional rules

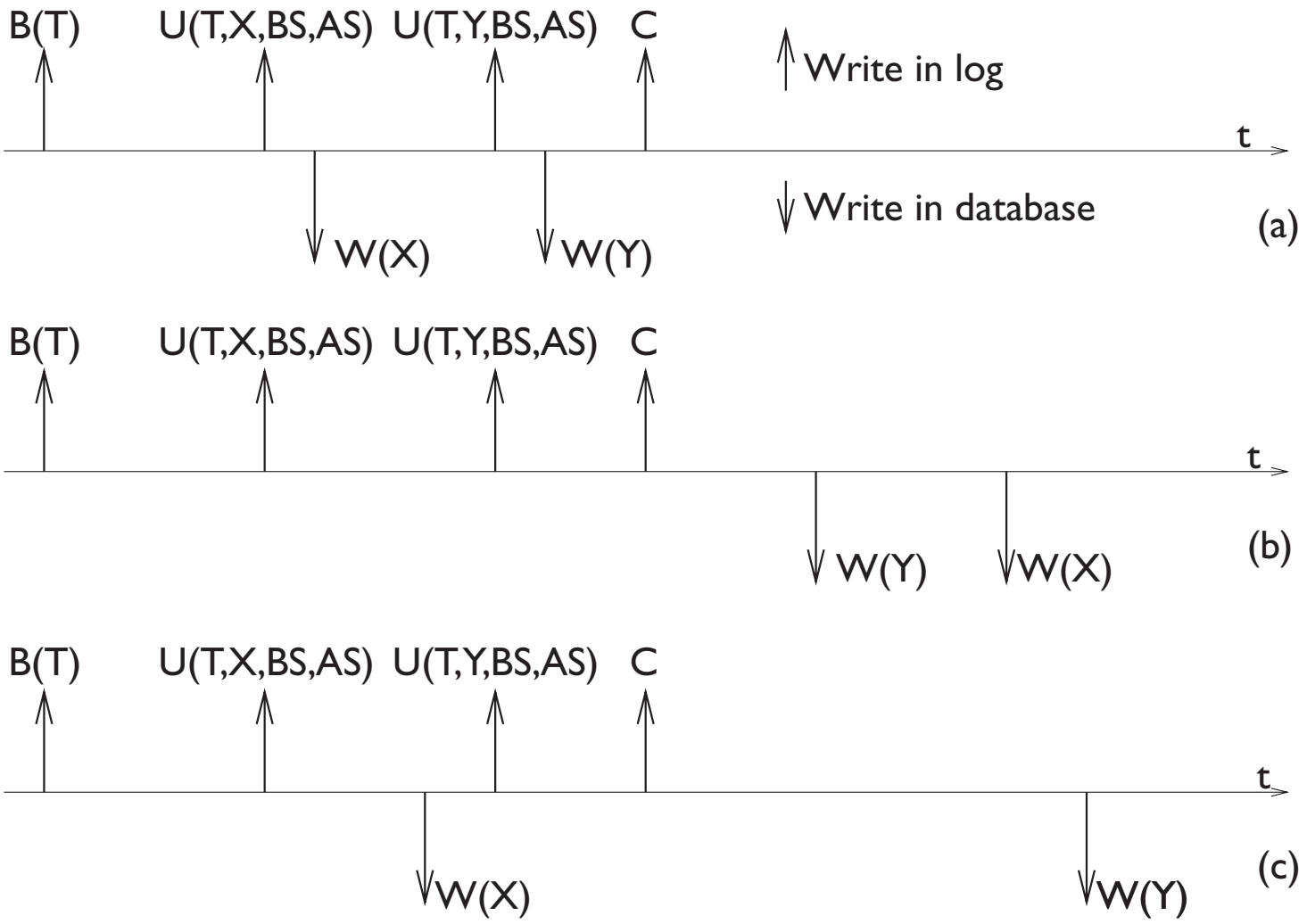
The reliability control system must follow two rules:

- *WAL* rule (write-ahead log): before-state parts of the log records must be written in the log before carrying out the corresponding operation on the database
- *Commit-Precedence* rule: after-state parts of the log records must be written in the log before carrying out the commit

Transaction outcome

- The atomic outcome of a transaction is established at the time when it writes the commit record in the log synchronously, using the *force* primitive
  - Before this event, a failure is followed by the *undo* of the actions, so reconstructing the original state of the database
  - After this event, a failure is followed by the *redo* of the actions carried out to reconstruct the final state of the transaction
- `abort` records can be simply written asynchronously into the top block of the log

Protocols for the joint writing of log and database



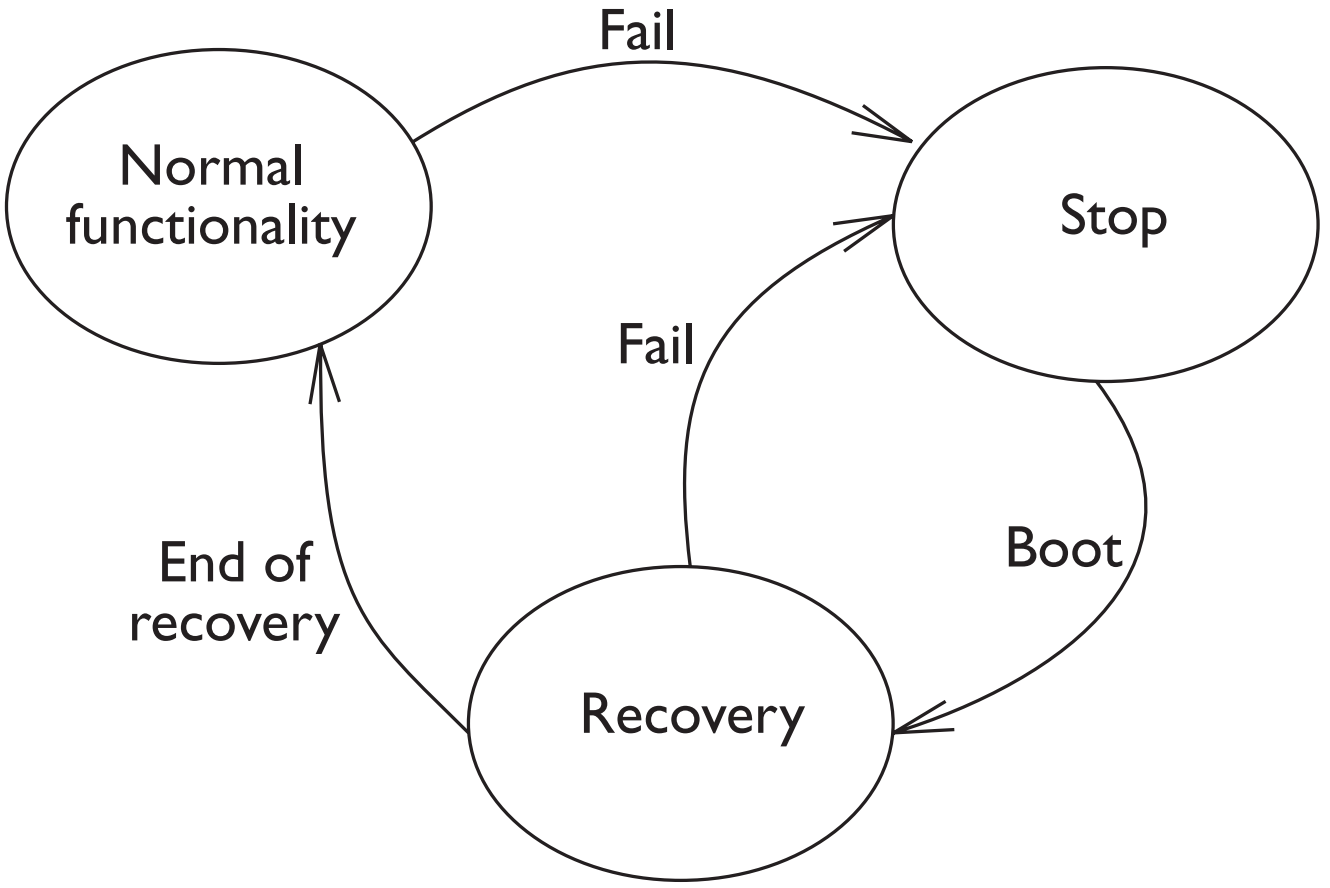
Optimizations

- Several log records are normally written into the same log page
- Several log records of the same transactions can be forced at commit time
- Several transactions can commit together by forcing their `commit` record with the same operation (*group commit*)
- Systems can exploit parallelism in writing logs

Failures in data management

- *System failures:* software bugs, for example of the operating system, or interruptions of the functioning of the devices (due, for example, to loss of power)
  - With loss of the contents of the main memory (and thus all the buffers)
  - With no loss of the contents of secondary memory
- *Device failures:* failures of secondary memory devices (for example, disk head crashes)
  - with loss of secondary memory contents
  - with no loss of stable storage (i.e.: of the log)
- *Restart protocols*
  - *Warm restart* used with system crashes
  - *Cold restart* used with device crashes

Fail-stop model of the functioning of a DBMS



Restart process

- Objective: classify transactions as:
  - Completed (whose actions were recorded in stable storage)
  - Committed but possibly not completed (whose actions must be redone)
  - Not committed (whose actions have to be undone)
- Assumption: no end record in the log

Warm restart

Four successive phases:

- Trace back the log until the most recent checkpoint record.
- Construct the *UNDO* set (transactions to be undone) and the *REDO* set (transactions to be redone).
- Trace back the log until the first action of the 'oldest' transaction in the two sets, *UNDO* and *REDO*, is found, and undo all the actions of the transactions in the *UNDO* set
- Trace forward the log and redo all the actions of the transactions in the REDO set

The protocol guarantees:

- *atomicity:* all the transactions in progress at the time of failure leave the database either in the initial state or in the final one
- *durability:* all pages of transactions in progress are written to secondary memory

Cold restart

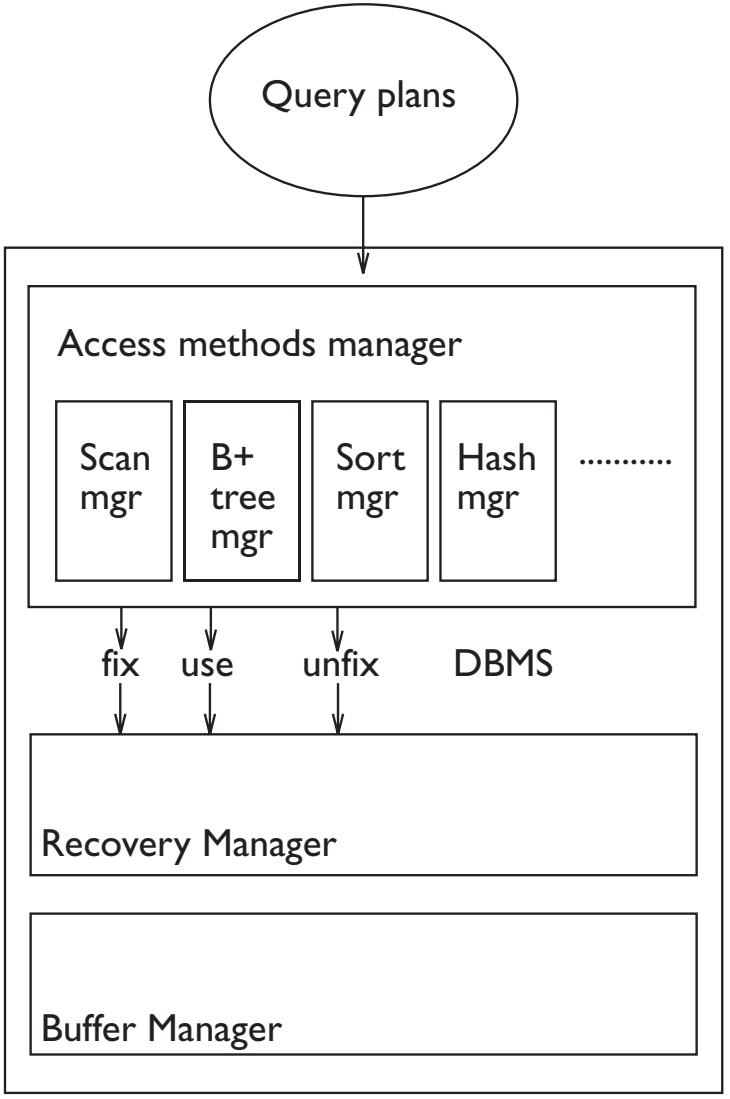
Divided into three phases.

- During the first phase, the *dump* is accessed and the damaged parts are selectively copied from the database. The most recent dump record in the log is then accessed
- The log is traced forward. The actions on the database and the commit or abort actions are applied as appropriate to the damaged parts of the database. The situation preceding the failure is thus restored
- A warm restart is carried out

Physical access structures

- Used for the efficient storage and manipulation of data within the DBMS
- Encoded as *access methods*, that is, software modules providing data access and manipulation primitives for each physical access structure
- Each DBMS has a limited number of types of access methods available
- Can be coded within applications that do not use a DBMS
- We will consider sequential, hash-based, and index-based data structures

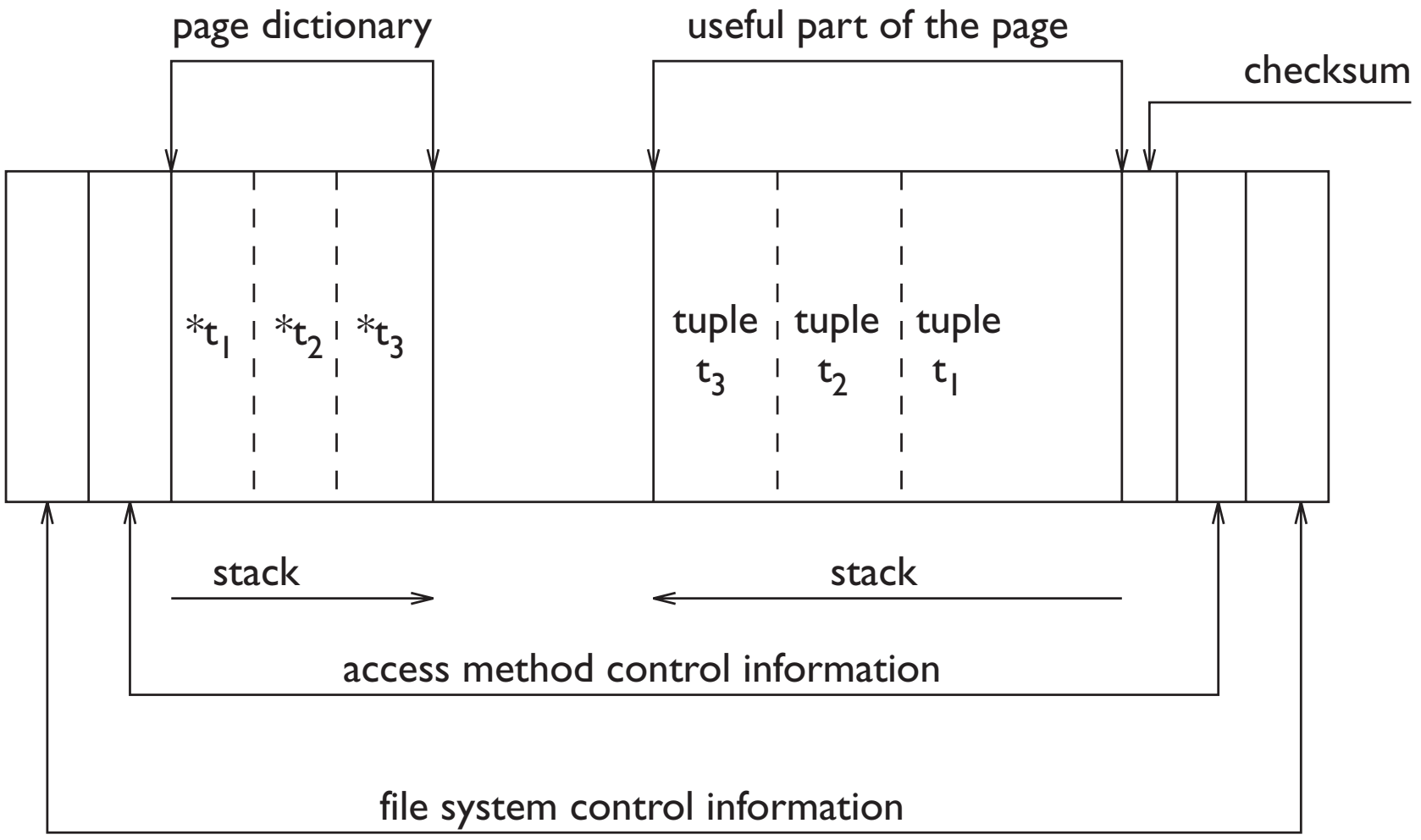
Architecture of the access manager



Organization of tuples within pages

- Each access method has its own page organization, we review pages of sequential and hash-based methods
- Each page has:
  - An initial part (*block header*) and a final part (*block trailer*) containing control information used by the file system
  - An initial part (*page header*) and a final part (*page trailer*) containing control information about the access method
  - A *page dictionary*, which contains pointers to each item of useful elementary data contained in the page
  - A *useful part*, which contains the data. In general, the page dictionary and the useful data grow as opposing stacks
  - A *checksum*, to verify that the information in it is valid
- Tree structures have a different page organization

Organization of tuples within pages



Page manager primitives

- *Insertion and update of a tuple* (may require a reorganization of the page if there is sufficient space to manage the extra bytes introduced)
- *Deletion of a tuple* (often carried out by marking the tuple as 'invalid')
- *Access to a field of a particular tuple*, identified according to the offset and to the length of the field itself, after identifying the tuple by means of its key or its offset

NOTES:

- Some page managers do not allow the separation of a tuple on more than one page
- When all the tuples have the same size, the page structure dictionary is simplified

Sequential structures

- Characterized by a sequential arrangement of tuples in the secondary memory
- In an *entry-sequenced* organization, the sequence of the tuples is dictated by their order of entry
- In an *array* organization, the tuples are arranged as in an array, and their positions depend on the values of an index (or indexes)
- In a *sequentially ordered* organization, the sequence of the tuples depends on the value assumed in each tuple by a field that controls the ordering, known as a *key field*

Entry-sequenced sequential structure

- Optimal for the carrying out of sequential reading and writing operations
- Uses all the blocks available for files and all the spaces within the blocks
- Primitives:
  - Accessed with sequential *scan*
  - Data loading and insertion happen at the end of the file and in sequence
  - Deletes are normally implemented by leaving space unused
  - More problems are caused by the updates that increase the file size

Array sequential structure

- Possible only when the tuples are of fixed length
- Made of *n* of adjacent blocks, each block with *m* of available slots for tuples
- Each tuple has a numeric index *i* and is placed in the *i*-th position of the array
- Primitives:
  - Accessed via `read-ind` (at a given index value).
  - Data loading happen at the end of the file (indices are obtained simply by increasing a counter)
  - Deletions create free slots
  - Updates are done on place

Ordered sequential structure

- Each tuple has a position based on the value of the key field
- Historically, ordered sequential structures were used on sequential devices (tapes) by batch processes. Data were located into the *main file,* modifications were collected in *differential files,* and the files were *periodically merged.* This has fallen out of use
- The main problems: insertions or updates which increase the physical space - they require reordering of the tuples already present
- Options to avoid global reorderings:
  - Leaving a certain number of slots free at the time of first loading. This is followed by ‘local reordering’ operations
  - Integrating the sequentially ordered files with an *overflow file*, where new tuples are inserted into blocks linked to form an *overflow chain*

Hash-based structures

- Ensure an efficient *associative* access to data, based on the value of a *key* field, composed of an arbitrary number of attributes of a given table
- A hash-based structure has *B* blocks (often adjacent)
- The access method makes use of a hash algorithm, which, once applied to the key, returns a value between zero and B-1
- This value is interpreted as the position of the block in the file, and used both for reading and writing tuples to the file
- The most efficient technique for queries with equality predicates, but inefficient for queries with interval predicates

Features of hash-based structures

- Primitive interface: `hash(fileid,Key):Blockid`
- The implementation consists of two parts
  - *folding*, transforms the key values so that they become positive integer values, uniformly distributed over a large range
  - *hashing,* transforms the positive binary number into a number between 0 and *B* - 1
- This technique works better if the file is made larger than necessary. Let:
  - *T* the number of tuples expected for the file
  - *F* the average number of tuples stored in each page,

  then a good choice for *B* is *T*/(0.8 x *F*), using only 80% of the available space

Collisions

- Occur when the same block number is returned by the algorithm starting from two different keys. If each page can contain a maximum of *F* tuples, a collision is critical when the value of *F* is exceeded
- Solved by adding an overflow chain starting from that page. They give the additional cost of scanning the chain
- The average length of the overflow chain is tabled as a function of the ratio T/(*F* x B) and of the average number *F* of tuples per page:

	1	2	3	5	10	(*F*)
.5	0.5	0.177	0.087	0.031	0.005	
.6	0.75	0.293	0.158	0.066	0.015	
.7	1.167	0.494	0.286	0.136	0.042	
.8	2.0	0.903	0.554	0.289	0.110	
.9	4.495	2.146	1.377	0.777	0.345	

T/(*F* x*B*)

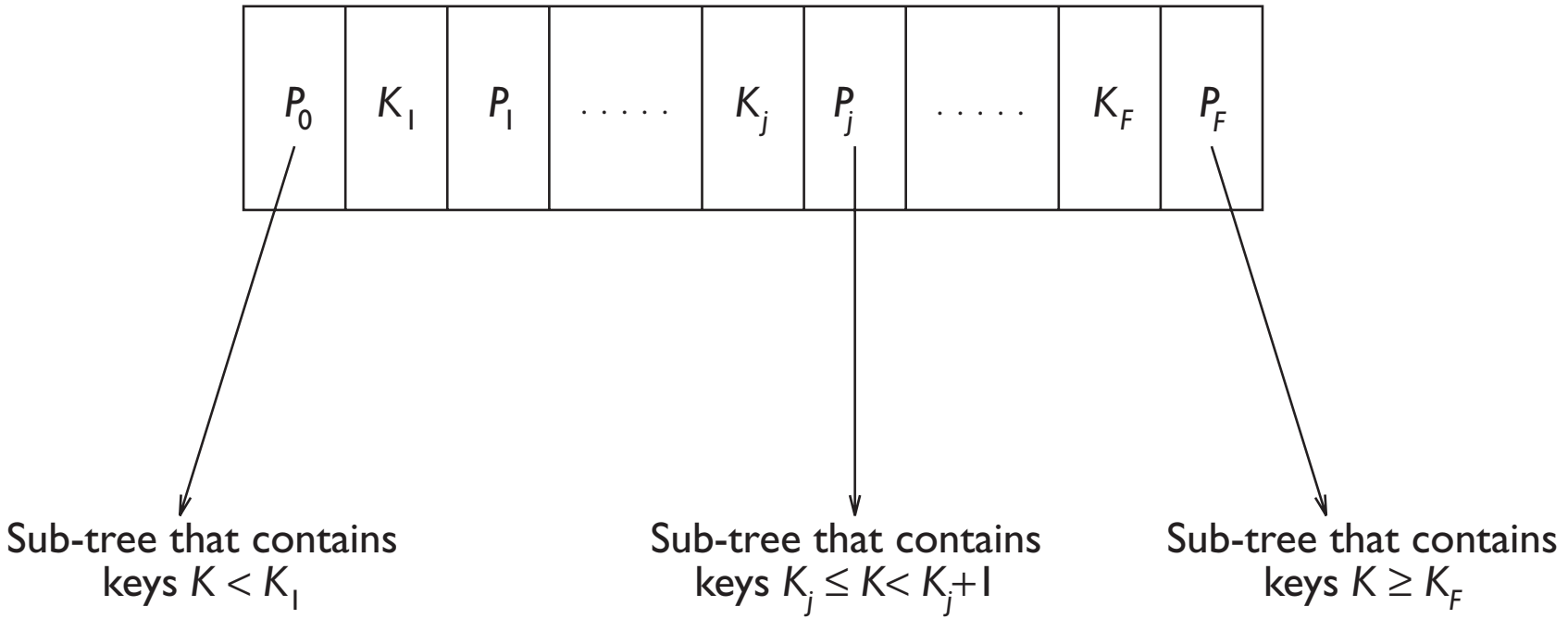
Tree structures

- The most frequently used in relational DBMSs
- Gives associative access (based on a value of a *key*, consisting of one or more attributes) without constraints on the physical location of the tuples
- Note: the primary key of the relational model and the key of tree structures are different concepts

Tree structure organization

- Each tree has:
  - a root node
  - a number of intermediate nodes
  - a number of leaf nodes
- The links between the nodes are established by pointers
- Each node coincides with a page or block at the file system and buffer manager levels. In general, each node has a large number of descendants (fan out), and therefore the majority of pages are leaf nodes
- In a *balanced tree*, the lengths of the paths from the root node to the leaf nodes are all equal. In this case, the access times to the information contained in the tree are almost constant (and optimal)

Information contained in a node (page) of a B+ tree



Node contents

- Each intermediate node contains F keys (in lexicographic order) and $F + 1$ pointers
- Each key K_j , $1 \leq j \leq F$, is followed by a pointer P_j; K_1 is preceded by a pointer P_0
- Each pointer addresses a sub-tree:
  - P_0 addresses the sub-tree with the keys less than K_1
  - P_F addresses the sub-tree with keys greater than or equal to K_F
  - P_j, $0 < j < F$, addresses the sub-tree with keys included in the interval $K_j \leq K < K_{j+1}$
- The value $F + 1$ is called the *fan-out* of the tree

Search technique

- At each intermediate node:
  - if $V < K_1$ follow the pointer P_0
  - if $V \geq K_F$ follow the pointer P_F
  - otherwise, follow the pointer P_j such that $K_j \leq V < K_{j+1}$
- The leaf nodes of the tree can be organized in two ways:
  - In *key-sequence*d trees the tuple is contained in the leaves
  - In *indirect trees* leaf nodes contain pointers to the tuples, that can be allocated by means of any other 'primary' mechanism (for example, entry-sequenced, hash, or key-sequenced)
- In some cases, the index structure is sparse (not complete). They locate a key value close to the value being sought, then a sequential search is carried out

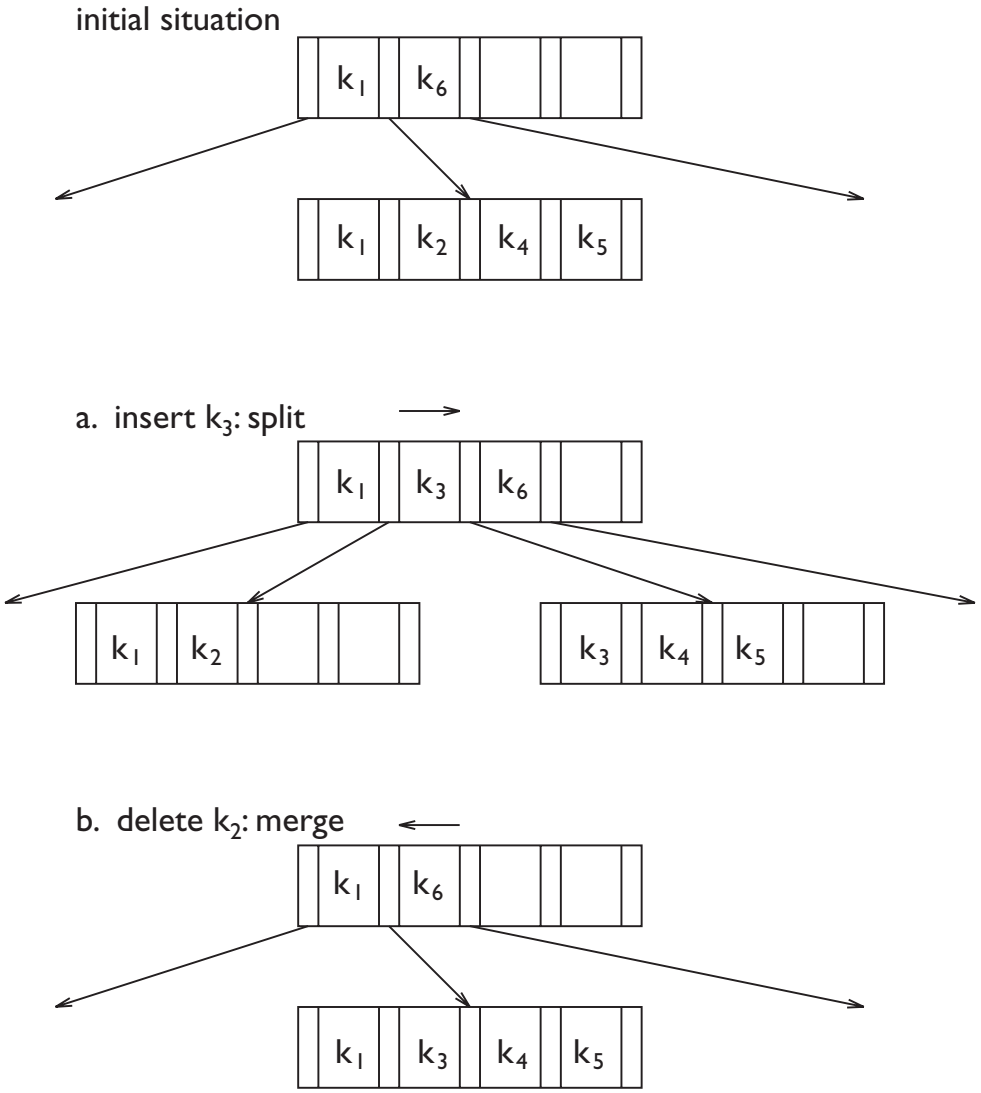
Split and merge operations

- Insertions and deletions done by using the search technique up to a leaf page
- Insertions are easy when there are free slots in the page
  - When the page has no available space, a split operation is necessary, allocating two leaf nodes in place of one. A split causes an increment in the number of pointers on the next (higher) level in the tree and may cause further split
- A deletion can always be carried out in situ
  - When a key present in other nodes of the tree is deleted, it is better to recover the successive key and substitute for it
  - When the deletion leaves two adjacent pages underused, they are concentrated into a single page by a merge operation. A merge causes a decrement in the number of pointers on the next (higher) level in the tree and may cause further merge

Split and merge operations

- The modification of the value of a key field is treated as the deletion of its initial value followed by the insertion of a new value
- The careful use of the split and merge operations makes it possible to maintain the tree balanced, with an average occupancy of each node higher than 50%

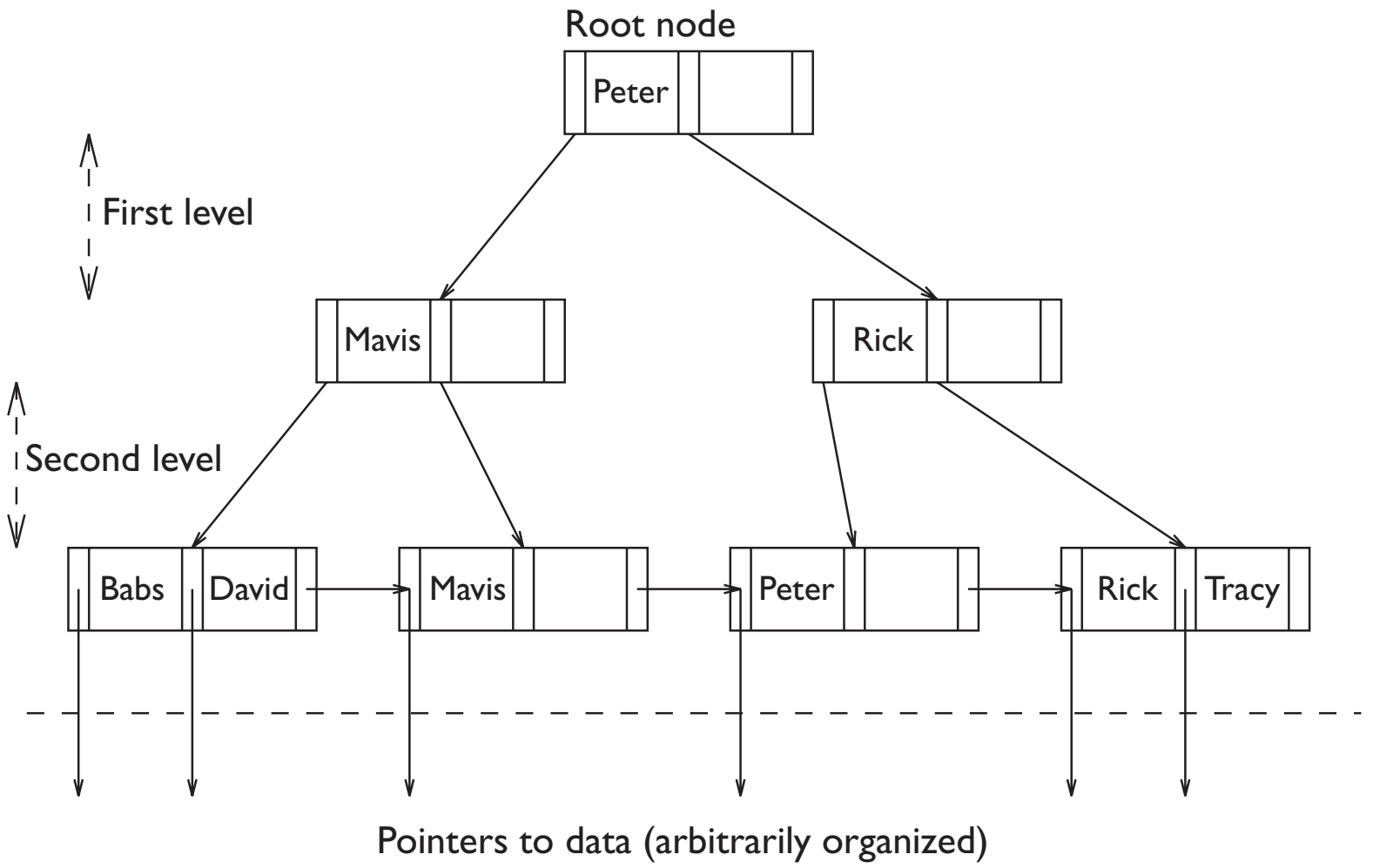
Split and merge operations on a B+ tree structure



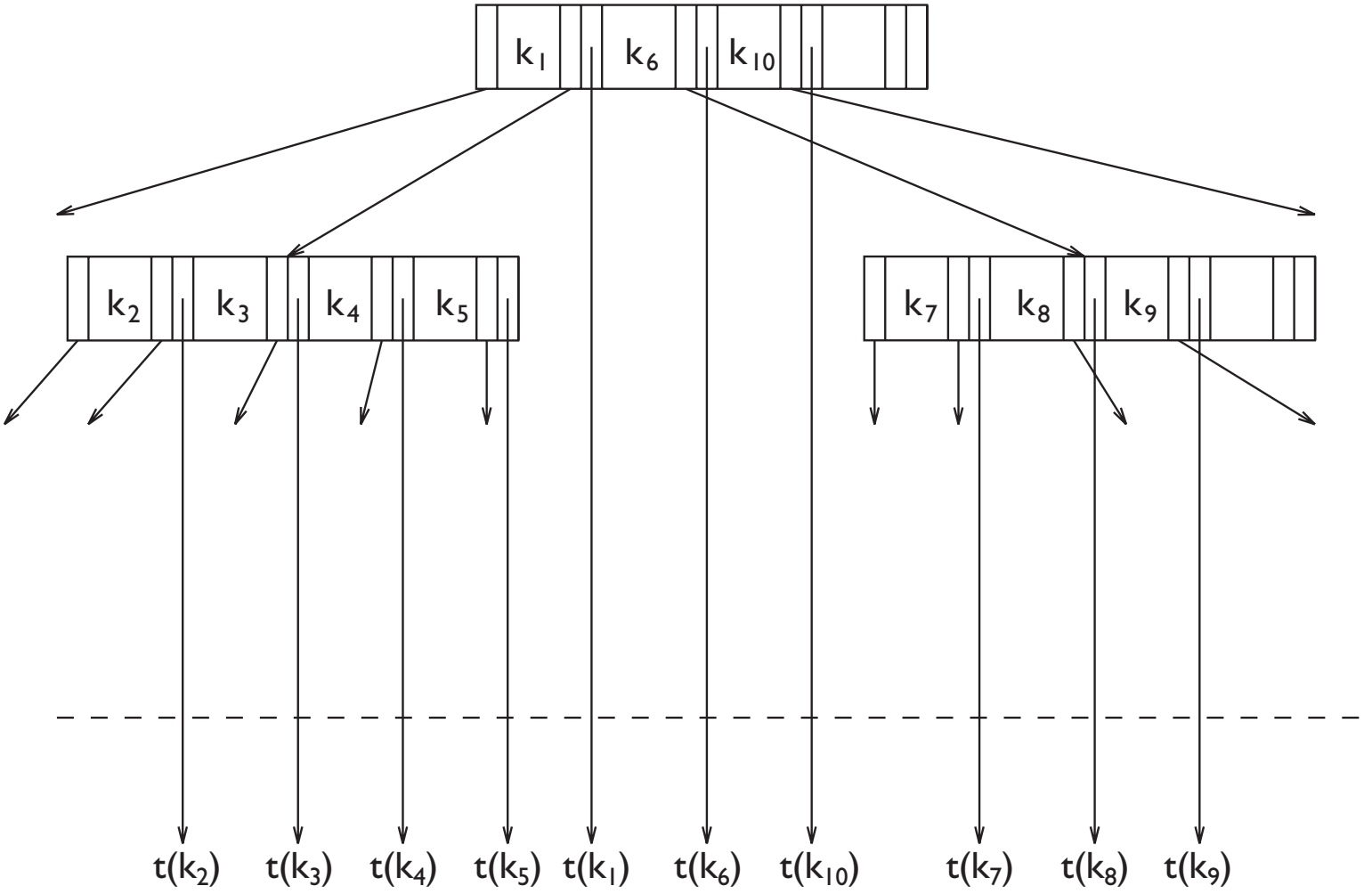
Difference between B and B+ trees

- B+ trees:
  - The leaf nodes are linked by a chain, which connects them in the order imposed by the key
  - Support interval queries efficiently
  - Mostly used by relational DBMSs
- B trees:
  - There is no sequential connection of leaf nodes
  - Intermediate nodes use two pointers for each key value K_i
    - one points directly to the block that contains the tuple corresponding to K_i
    - the other points to a sub-tree with keys greater than K_i and less than K_{i+1}

Example of B+ tree



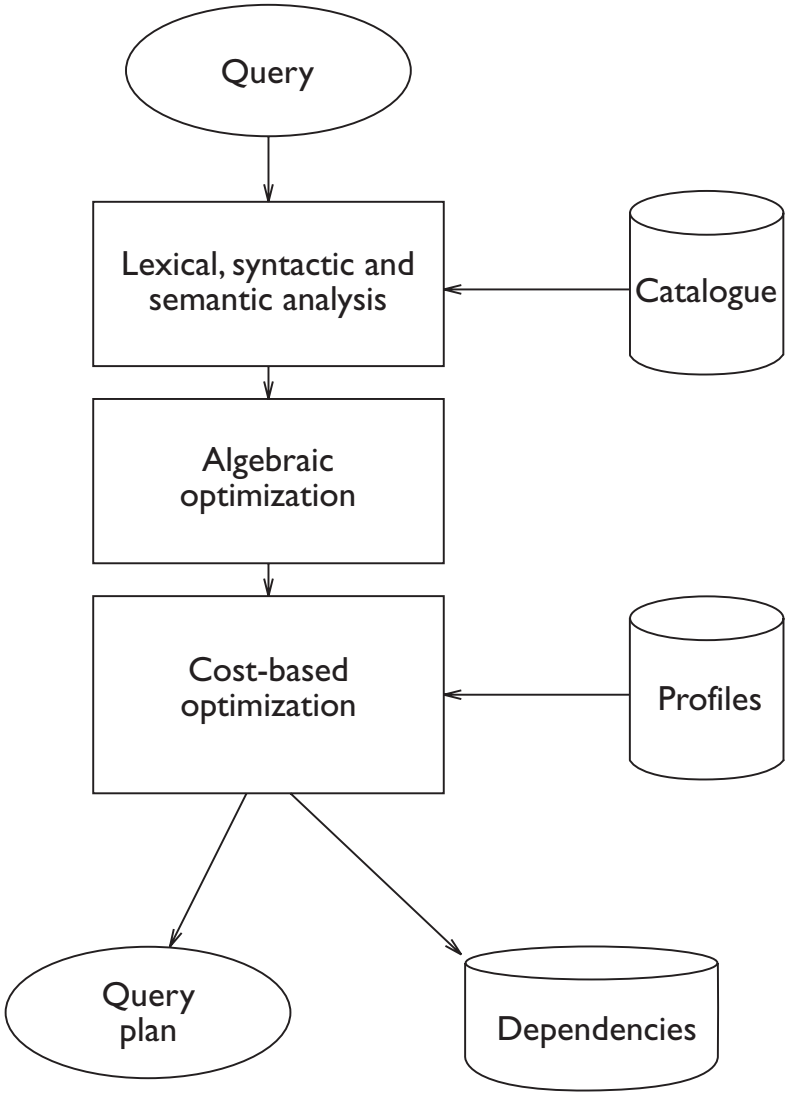
Example of a B tree



Index design

- *Primary index*: one with key-sequenced structure, supporting a sequential-type search
  - Normally unique (i.e., on the table's primary key)
- Secondary indexes: many with indirect trees, which can be either *unique* or *multiple*
  - Normally on attributes used by query conditions
- *Efficiency* is normally satisfactory, because the pages that store the first levels of the tree often remain in the buffer due to other transactions
- *Optimization* of the occupied space occurs by means of the compression of key values, by using:
  - prefixes in the high levels of the tree
  - suffixes in the low levels of the tree

Compilation of a query



Query optimization

- *Optimizer*: an important and classic module in the architecture of a databas
- It receives a query written in SQL, produces an access program is obtained in ‘object’ or ‘internal’ format, which uses the data structures provided by the system. Steps:
  - Lexical, syntactic and semantic analysis, using the data dictionary
  - Translation into an internal, algebraic form
  - Algebraic optimization (execution of all the algebraic transformation that are always convenient, such as the ‘push’ of selections)
  - Cost-based optimization
  - Code generation using the physical data access methods provided by the DBMS

Approaches to query compilation

- *Compile and store*: the query is compiled once and carried out many times
  - The internal code is stored in the database, together with an indication of the dependencies of the code on the particular versions of tables and indexes of the database
  - On changes, the compilation of the query is invalidated and repeated
- *Compile and go*: immediate execution, no storage

Relation profiles

- Profiles contain quantitative information about the tables and are stored in the data dictionary:
  - the cardinality (number of tuples) of each table *T*
  - the dimension in bytes of each tuple of *T*
  - the dimension in bytes of each attribute A_j in *T*
  - the number of distinct values of each attribute A_j in *T*
  - the minimum and maximum values of each attribute A_j in *T*
- Periodically calculated by activating appropriate system primitives (for example, the `update statistics` command)
- Used in cost-based optimization for estimating the size of the intermediate results produced by the query execution plan

Internal representation of queries

- Internal representation as trees, whose:
  - leaves correspond to the physical data structures (tables, indexes, files)
  - intermediate nodes represent data access operations that are supported on the physical structures
- Include sequential scans, orderings, indexed accesses and various types of join

Scan operation

- Performs a sequential access to all the tuples of a table, at the same time executing various operations of an algebraic or extra-algebraic nature:
  - Projection of a set of attributes
  - Selection on a simple predicate (of type: $A_i = v$)
  - Sort (ordering)
  - Insertions, deletions, and modifications of the tuples currently accessed during the scan
- Primitives:

  `open`, `next`, `read`, `modify`, `insert`, `delete`, `close`

Sort operation

- Various methods for ordering the data contained in the main memory, typically represented by means of a record array
- DBMSs typically cannot load all data in the buffer; thus, they separately order and then merge data sets, using the available buffer space

Indexed access

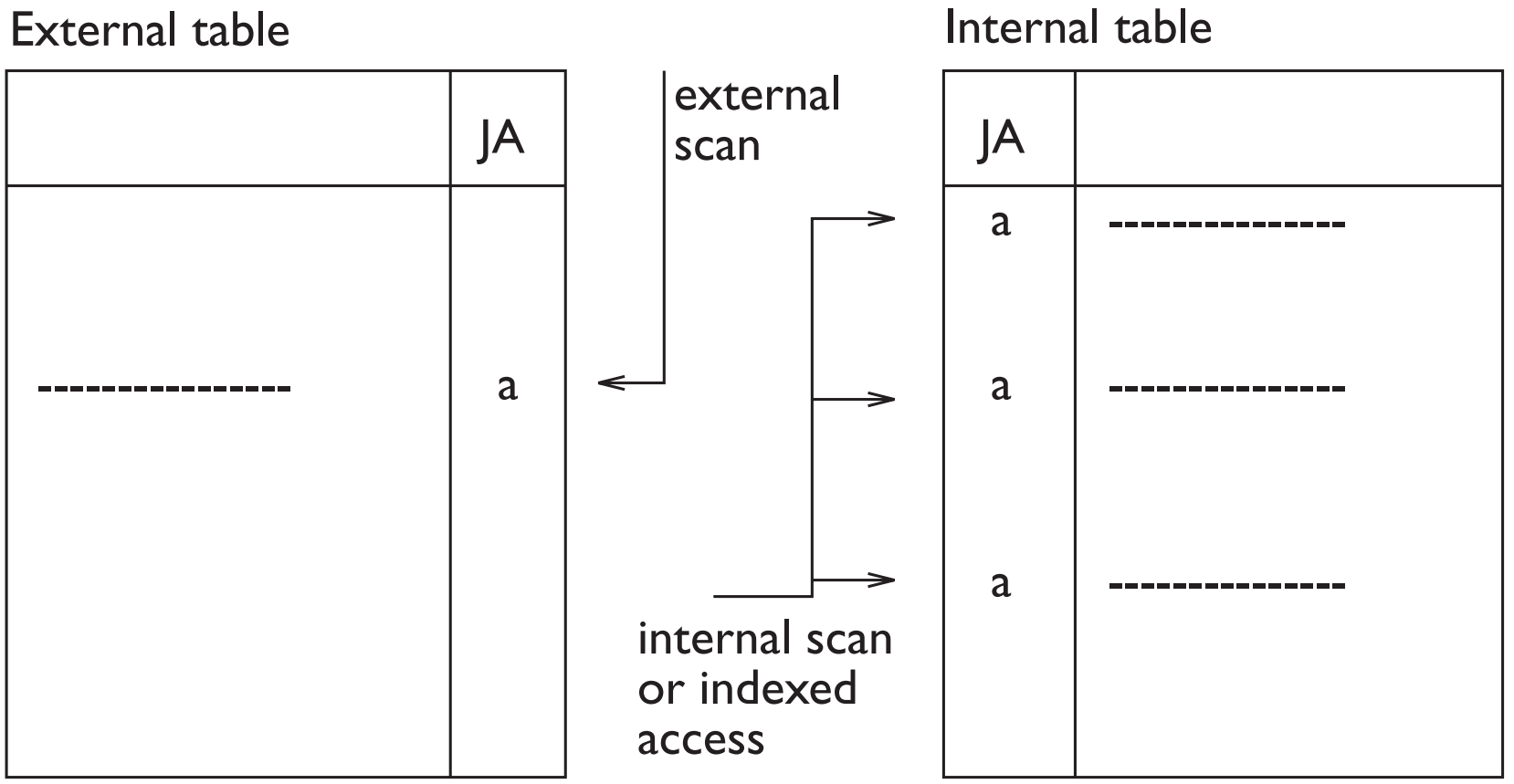
- Indexes are created by the database administrator to favor queries when they include:
  - simple predicates (of the type $A_i = v$)
  - interval predicates (of the type $v_1 \leq A_i \leq v_2$)

  These predicates are *supported* by the index
- With a conjunction of predicates:
  - the DBMS chooses the most selective supported predicate for the primary access, and evaluates the other predicates in main memory
- With a disjunction of predicates:
  - if any of them is not supported a scan is needed
  - if all are supported, indexes can be used only with duplicate elimination

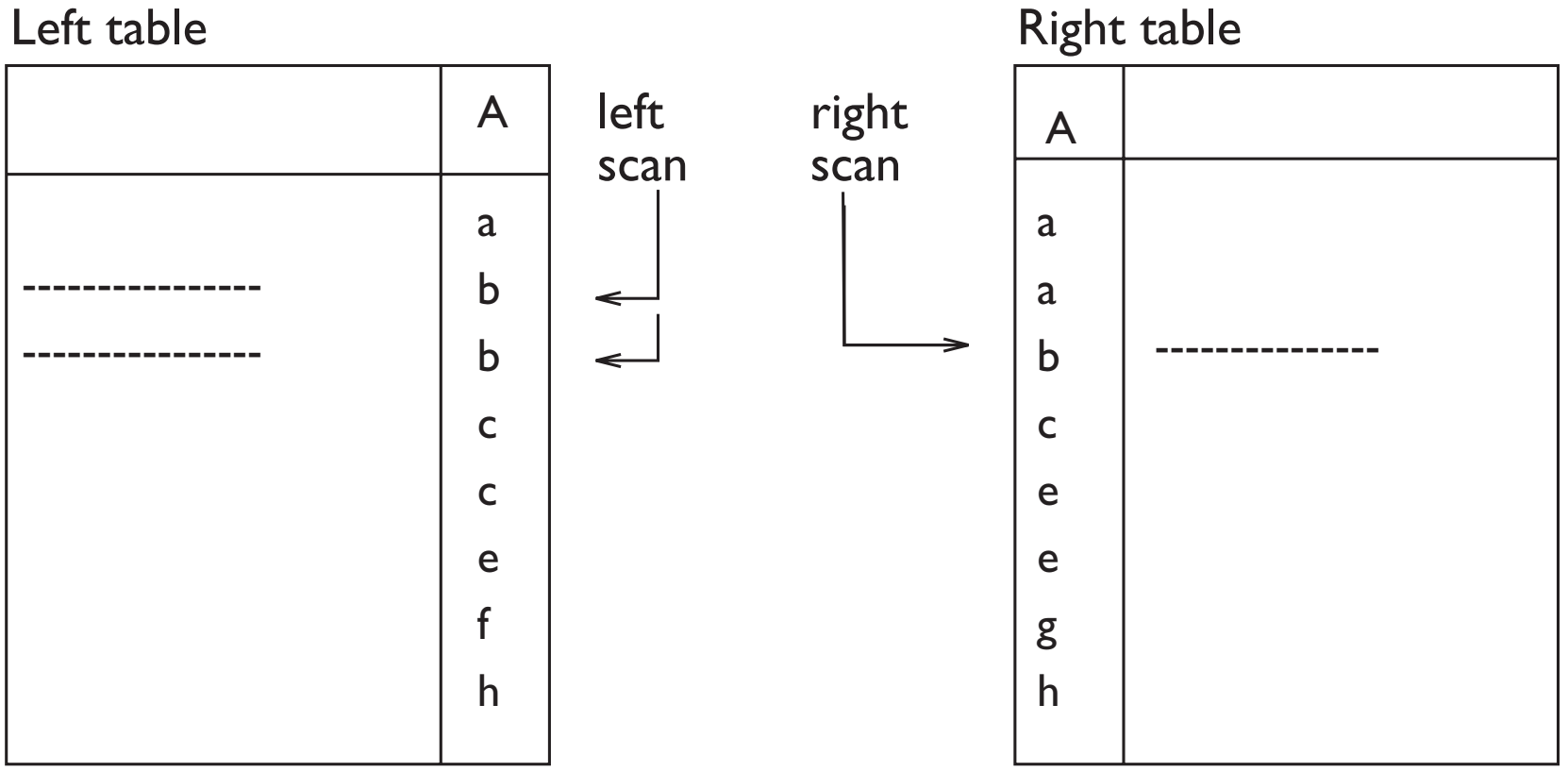
Join methods

- Joins are the most costly operation for a DBMS
- There are various methods for join evaluation, among them:
  - *nested-loo*p, *merge-scan* and *hashe*d
- The three techniques are based on the combined use of scanning, hashing, and ordering

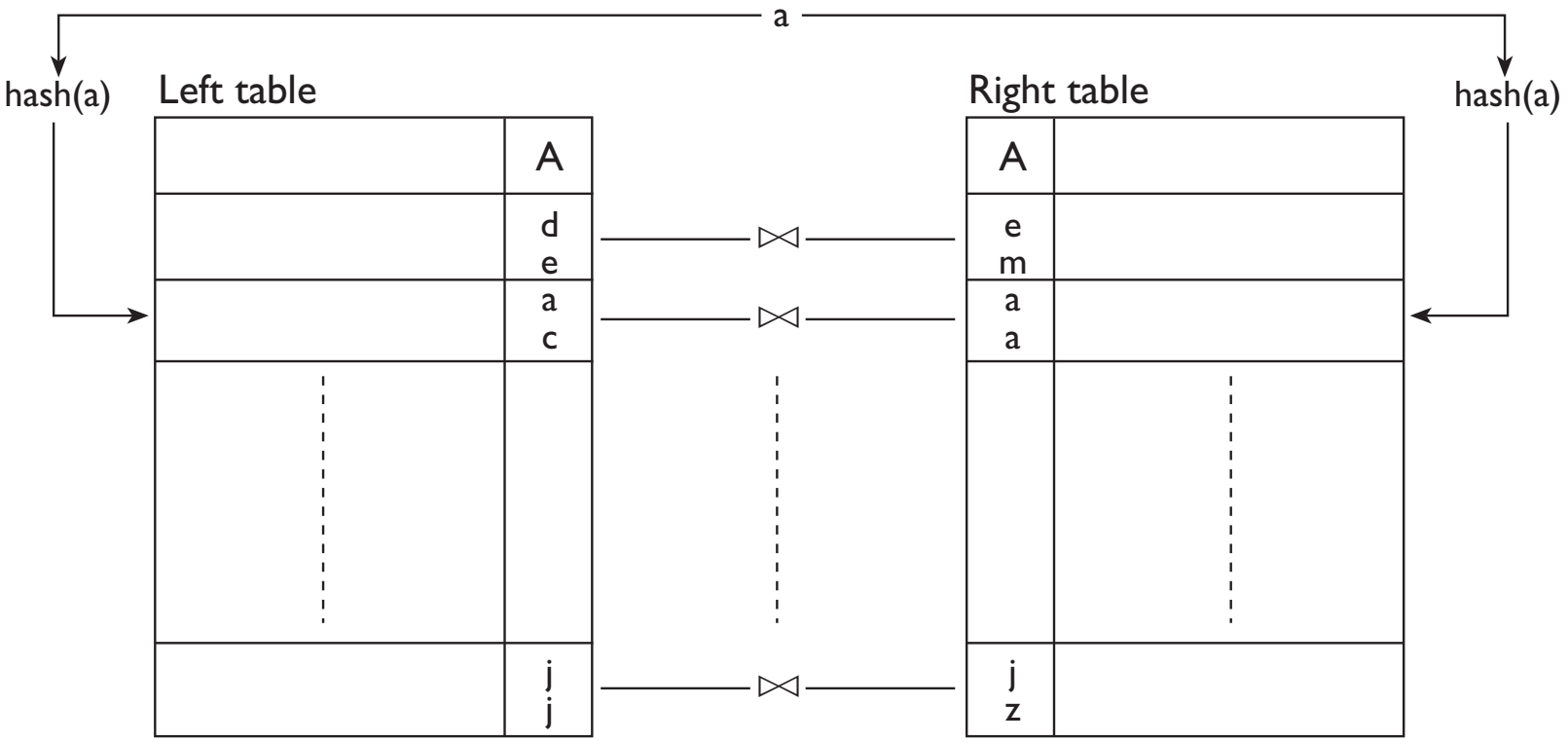
Join technique with nested-loop



Join technique with merge scan



Join technique with hashing



Cost-based optimization

- An optimization problem, whose decisions are:
  - The data access operations to execute (e.g., scan vs index access)
  - The order of operations (e.g., the join order)
  - The option to allocate to each operation (for example, choosing the join method)
  - Parallelism and pipelining can improve performances
- Further options appear in selecting a plan within a distributed context

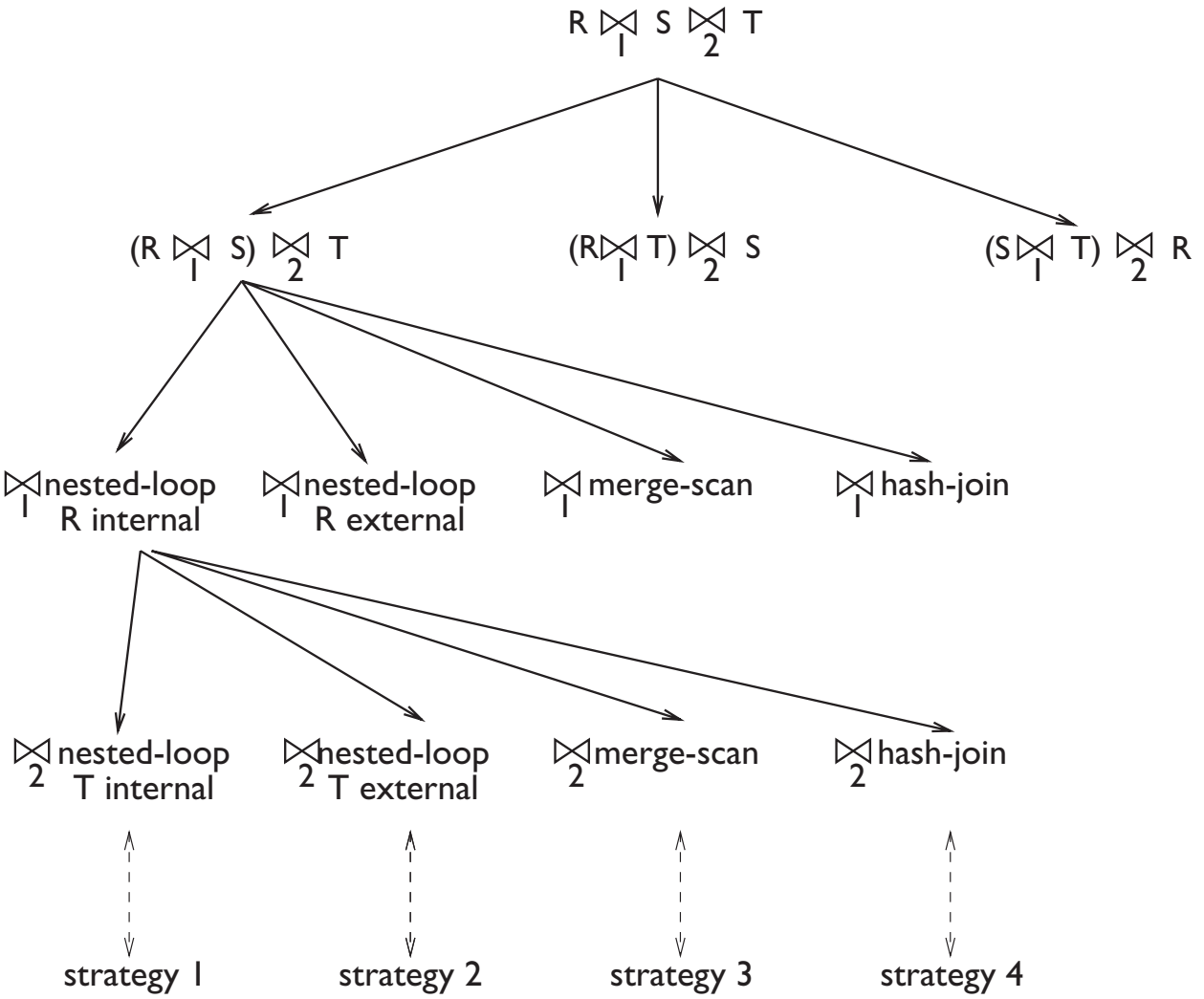
Approach to query optimization

- Optimization approach:
  - Make use of profiles and of approximate cost formulas
  - Construct a *decision tree*, in which each node corresponds to the choice; each leaf node corresponds to a specific *execution plan*
  - Assign to each plan a cost:

    $$C_{total} = C_{I/O} \times n_{I/O} + C_{cpu} \times n_{cpu}$$

  - Choose the one with the lowest cost, based on operations research (branch and bound)
- The optimizers should obtain ‘good’ solutions whose cost is near that of the optimal solution

Execution options in a conjunctive query



Physical database design

- The final phase in the process of database design
- Takes as input the logical schema of the database and the predictions for the application load
- Produces as output the physical schema of the database, made up of the definitions of the relations and of the physical access structures used, with the related parameters
- Depends on the characteristics of the chosen DBMS

Physical design for the relational model

- Most relational DBMS supports only indexes and tuple clustering
- Physical design can be reduced to the activity of identifying indexes to be defined on each relation
- The key of a relation is usually involved in selection or join operations (or both). For this reason, each relation normally supports a unique index on the primary key
- Other indexes are added so as to support the most common query predicates
- If the performance is unsatisfactory, we can tune the system by adding or dropping indexes
- It is useful to check how indexes are used by queries, by using the `show plan` command.

Definition of indexes in SQL

- Commands in relational systems for the creation and dropping of indexes are not part of standard SQL, but their syntax is rather similar in all DBMSs
- Syntax of the commands for the creation and dropping of an index:
  - `create` [`unique`] `index` *IndexName* `on` *TableName*(*AttributeList*)
  - `drop index` *IndexName*