XI. Technologies for DBMSs

Buffer Management
Reliability Control
Physical Access Structures
Query Optimization
Physical Database Design

Technologies for D˚BMSs

Ideally, we'd like to have a DBMS support the following:

- ***Query optimizer*** - selects a data access strategy;
- ***Access Methods Manager*** - executes the strategy using an
  - ✓ Relational Storage System, or
  - ✓ Object Manager.
- ***Buffer Manager*** - manages page accesses;
- ***Reliability Manager*** - manages failures;
- ***Concurrency Control Manager*** - manages interferences due to multi-user data access.

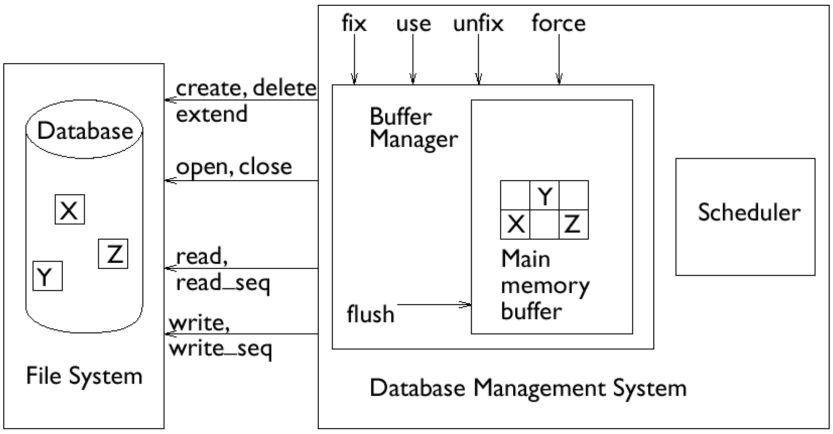
Buffer Management

- A ***buffer*** is a section of main memory pre-allocated to a DBMS for disk I/O.
- The buffer is organized in pages, of a size equal or multiple of the disk input/output blocks used by the operating system;
- The size of a page can range from a few Kbytes to about a hundred Kbytes.
- Access times for main memory are generally six orders of magnitude faster than access times for secondary memory
- Sometimes a buffer can store the entire database (***main memory database***)

Buffer Manager Principles

- The buffer manager deals with I/O of pages between main mamory and disk storage.
- Manager supports primitive operations such as *fix*, *use*, *unfix*, *flush* and *force*.
- The principles and policies of buffer management are similar to those of main memory management for operating systems, for example:
  - ✓ ***Data locality***: currently referenced data has a greater probability of being referenced in the immediate future;
  - ✓ ***80-20 Rule***: On average, 80% of the applications access 20% of available data.

Buffer Manager Organization



Primitive Buffer Operations

- ***fix --*** loads a page into the buffer, requires a read from disk only when the page is not already in the buffer; after the operation, the page is loaded and ***valid***, i.e., allocated to an active transaction; a pointer to the page is returned to the transaction.
- ***use*** -- used by a transaction to gain access to a page previously loaded in memory, confirming its presence in the buffer and its status as a valid page
- ***unfix*** -- signals the end of use of a page by a transaction;
- ***force*** -- synchronously transfers a page from to disk.
- ***flush*** -- transfers invalid (unused) pages to secondary memory asynchronously and independently of active transactions

`fix`

- Searches the buffer for requested page, if found and unused, its address is returned; else the page is read from disk and its address is returned.
- If the requested page is in the disk, a buffer page is selected for loading; if the buffer page is free, just load; else buffer page is sent to disk (***victim***) by invoking `flush`, then reads reads the requested page from disk and returns the page address.

Buffer Management Policies

- The ***steal policy*** allows the buffer manager to select an active page allocated to another transaction as a victim. ***No-steal policy*** disallows this possibility.
- The ***force policy*** requires that all active pages of a transaction are written to disk before committing. The ***no-force policy*** delegates the writing of pages of a transaction to the buffer manager.
- The ***no-steal/no-force*** policies are generally 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.

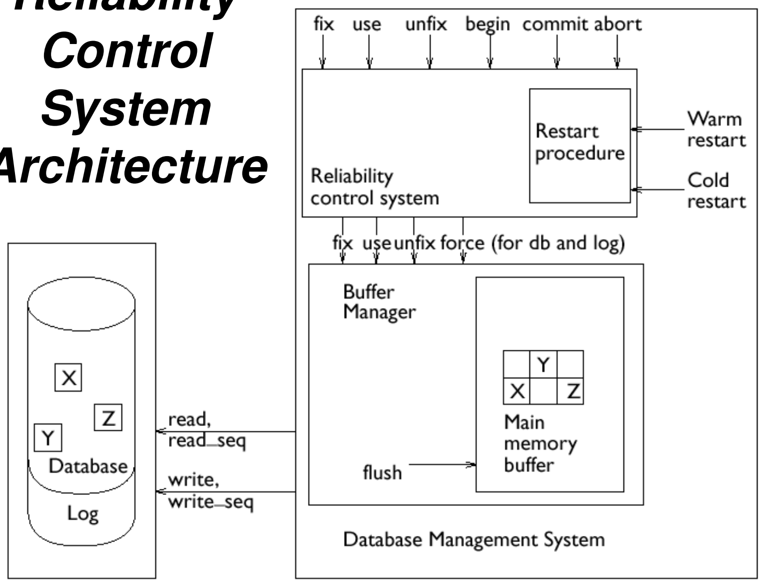
Buffer Managers and File Systems

- The file system is responsible for managing the data structures stored on disk, also the current state of disk use. It must identify which disk blocks are free and which are allocated to files
- A DBMS uses a 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 copied 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 System

- Responsible for executing the transactional commands **`begin transaction`** (**`B`**), **`commit work`** (**`C`**), **`rollback work`** (**`A`**, for abort), also the primitives for recovery after malfunctions **`warm restart`** and **`cold restart`**.
- Ensures atomicity and durability.
- Uses as main data structure the ***log***; this is a sequential file written in a stable memory. which registers the operations carried out by the DBMS.

Reliability Control System Architecture



DB Technology -- 11

Stable Memory

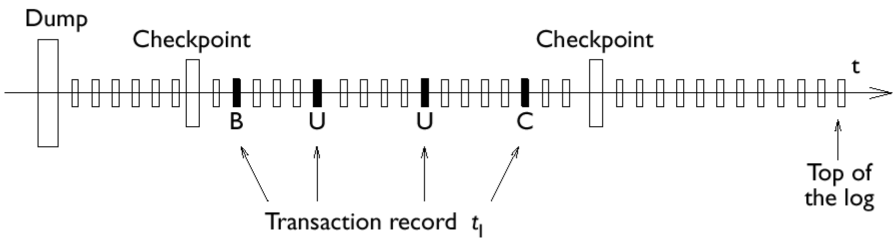
- ***Stable memory*** is a memory system that is ***failure-resistant***.
- There is no such thing in practice; however, replication and robust writing protocols can bring the probability close to zero.
- A failure of stable memory is assumed as ***catastrophic*** and impossible, at least in this context.
- Stable memories usually come in different forms:
  - ✓ a tape unit -- very reliable, but slow;
  - ✓ a pair of devices of different kind (e.g., a tape and a disk) -- better alternative;
  - ✓ two mirrored disk units.

DB Technology -- 12

Log Organization

- 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 records:
  - ✓ ***Transaction records*** -- keep track of the activities of the activities of a transaction by recording all operations **`B(T), I(T,O,AS), D(T,O,BS), U(T,O,BS,AS), C(T), A(T);`**
  - ✓ ***System records*** -- keep track of system-wide operations `dump` (rare) and `checkpoint` (more frequent).

Log Content



Undo and Redo

- **`Undo`** of an action on an object *O:*
  - ✓ **`update, delete`***:* copy **`BS`** into the object *O;*
  - ✓ **`insert:`** delete the object *O;*
- **`Redo`** of an action on an object *O:*
  - ✓ **`insert, update:`** copy **`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 $\mathbf{CK(T_1,T_2,..T_n)}$, which contains the identifiers of all active transactions;
  - ✓ In this way, the effects of all committed transactions 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

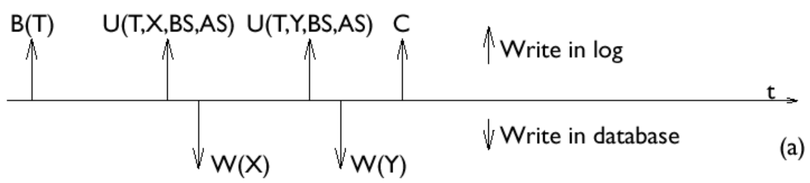
The reliability control system must follow two rules:

- ***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

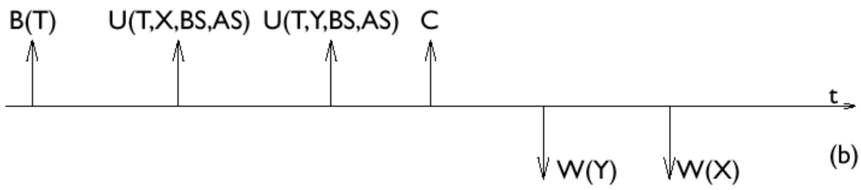
- The atomic outcome of a transaction is established at the time when it writes the commit record in the log synchronously, using the *force* primitive
  - ✓ Before this event, a failure is followed by the *undo* of the actions, thereby reconstructing the original state of the database;
  - ✓ After this 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.

Protocol for Writing Log (I)



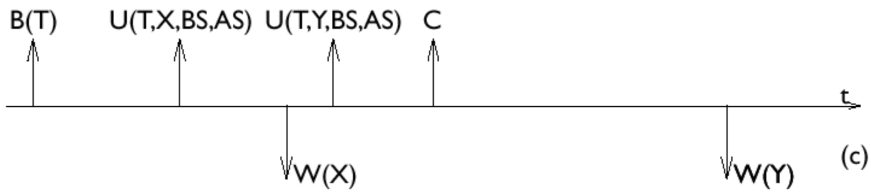
- The transaction first records **`B(T)`,** then carries out its update actions by writing first the log records and then the pages of the database, changing **`BS`** values to **`AS`** values.
- This way, at commit all the pages of the database modified by the transaction are already written on disk.
- This scheme does not require redo operations.

Protocols for Writing Log (II)



- The writing of log records precedes that of the actions on the database.
- This scheme does not requires undo operations either.

Protocols for Writing Log (III)



- Commonly used, the writing in the database once protected by the appropriate writing on the log, can happen at any time with regard to the writing of the commit record in the log.
- This scheme allows the buffer manager to optimize the execution of flush operations.
- However, it requires both undo and redo.

Failures in Database Systems

- A ***system failure*** is caused by a software bug, (e.g., of the operating system), or an interruption of the functioning of hardware (e.g., due to loss of power.)

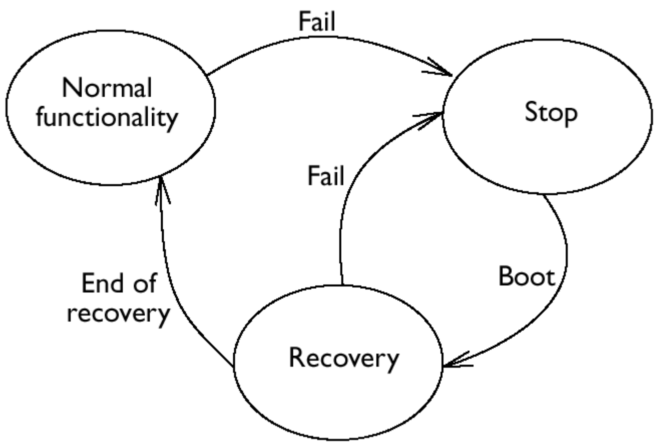
☹ Causes loss of the contents of main memory (and thus of all buffers); but no loss of the contents of secondary memory (☺).

- *A* ***device failure*** is a failure of a secondary memory device (e.g., a disk head crash).

☹ Causes loss of disk contents, but no loss of stable storage (i.e., the log)

- There are two types of restart protocols:
  - ✓ ***Warm restart*** used after system crashes;
  - ✓ ***Cold restart*** used after device crashes.

Fail-Stop Failure Model



Restart Process

- When the system restarts after a failure, it has to decide what to do with the transactions that were executing at the time of the failure.
- Transactions are classified as:
  - ✓ ***Completed*** -- when their actions were recorded in stable storage before the failure;
  - ✓ ***Committed but possibly not completed*** -- whose actions must be redone, e.g., because of data lost in the buffer;
  - ✓ ***Not committed*** -- whose actions have to be undone because the transaction did not commit.

Warm Restart

Consists of four sequential steps:

- ***Trace back*** the log until the most recent checkpoint record.
- ***Construct the UNDO set*** (i.e.,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.

Warm Restart Protocol

The protocol guarantees:

- ***Atomicity*** in the sense that all the transactions in progress at the time of failure leave the database either in the initial state or in the final one
- ***Durability*** in the sense that all pages of transactions in progress are written to secondary memory.

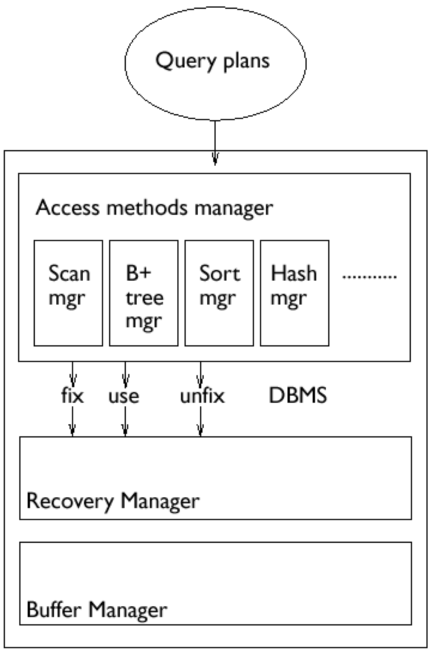
Cold Restart

Now parts of the database may have been damaged. Consists of three steps:

- During the first phase, the `dump` operation is executed and 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.
- Finally, a warm restart is conducted.

Physical Access Structures

- Used for efficient storage and manipulation of data.
- For each access structure there are ***access methods***, i.e. software modules providing data access and manipulation primitives for each physical access structure.
- Each DBMS has a limited number of access methods available.
- Sometimes, access methods can be used directly from an application without going through a DBMS.
- We will consider ***sequential***, ***hash-based***, and ***index-based*** access structures and methods.

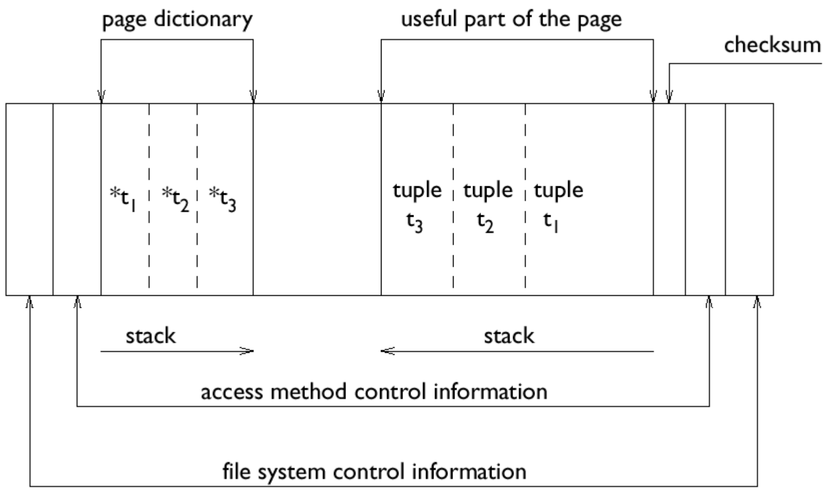


Architecture of the Access Manager

Tuples within Pages

- Each access method has its own page organization. Each page includes:
  - ✓ An ***initial part*** and a ***final part*** with control information for the file system and the access method;
  - ✓ A ***page dictionary***, containing pointers to elementary data contained in the page;
  - ✓ A ***useful part***, which contains data; page dictionary and useful data grow as opposing stacks within the page;
  - ✓ A ***checksum***, to verify that page data are valid.
- Tree structures have a different page organization.

Tuples within Pages



Page Manager Primitives

- ***Insert or update a tuple*** -- may require a reorganization of the page if there is sufficient space to manage the extra bytes introduced).
- ***Delete a tuple*** -- often carried out by marking the tuple as 'invalid'.
- ***Access a tuple field*** -- first identify tuple by means of its key or its offset, then locate field by its offset and length.
- Note that some page managers do not allow the separation of a tuple in different pages.
- Also, when all the tuples have the same size, the page dictionary is simplified.

Sequential Access 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 by a field in each tuple that controls the ordering, known as a ***key field****.*

Entry-Sequenced Sequential Structure

- This is optimal for carrying out sequential reading and writing operations.
- Uses all the blocks available in a file and all spaces within each block.
- Primitives:
  - ✓ Access with sequential scan operation;
  - ✓ Data loading and insertion happen at the end of the file and in sequence;
  - ✓ Deletes are normally implemented by leaving space unused;
  - ✓ Updates can cause problems because they increase the file size.

Array Sequential Structure

- This is useful only when the tuples are of fixed length.
- An array sequential structure consists of *n* adjacent blocks, each block with *m* available tuple slots.
- Each tuple is assigned 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).
  - ✓ Insertions always happen at the end of the file (indices are obtained simply by increasing a counter);
  - ✓ Deletions create free slots;
  - ✓ Updates are done in place.

Ordered Sequential Structure

- Each tuple has a position based on its key value.
- 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 files were periodically ***merged*** (this practice no longer applies.)
- Key problem: insertions or updates that increase physical space require reordering the whole file.
- Options to avoid global reorderings:
  - ✓ Leave some free slots at the time of first loading. This is followed by 'local reordering' operations;
  - ✓ Use an ***overflow file***, where new tuples are inserted into blocks linked to form an ***overflow chain***.

Hash-Based Structures

- Ensure 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
- This is the most efficient technique for queries with equality predicates, but inefficient for queries with interval predicates. (...***why***?)

Details on Hash-Based Structures

- Primitive interface: `hash(fileId,key):BlockId`
- The implementation of `hash` consists of two parts
  - ✓ ***folding***, transforms the key values into positive integers, uniformly distributed over a large range;
  - ✓ ***hashing,*** transforms the positive integer into a number between 0 and B - 1.
- This technique works best if the file is made larger than necessary. Let:
  - ✓ T the number of tuples expected for the file;
  - ✓ F the average number of tuples in each page;

  then a good choice for B is $T/(0.8 \times F)$, using only 80% of the available space in the file.

Collisions

- Occur when the same block number is returned by the algorithm to several different keys.
- Collisions are addressed by adding an overflow chain to each page. There are extra costs in scanning the chain.
- The following table computes the length of the overflow chain:

$T/(F \times B)$ \ (F)	1	2	3	5	10
.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

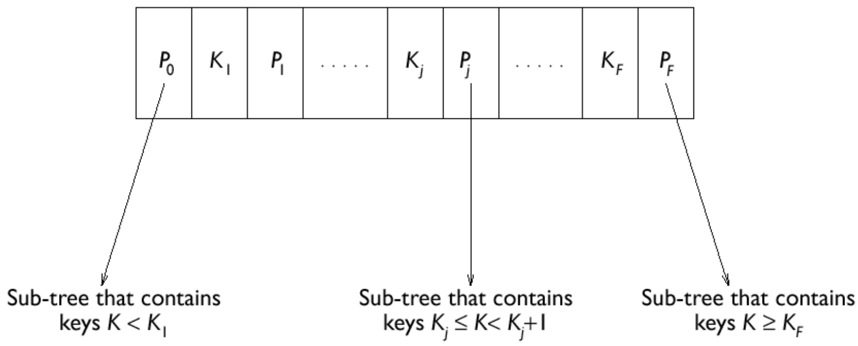
Tree Structures (B Trees)

- These are the most frequently used physical storage structures for relational DBMSs.
- Tree structures give 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 that the primary key of a relation and the key of a tree structure storing this relation need not be the same.

Tree Structure Organization

- Each tree has a single root node, a number of intermediate nodes, and a number of leaf nodes.
- The links between nodes are defined by pointers.
- Each node coincides with a page or block of 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 (almost) equal; this way, the access times to the information contained in the tree are almost constant (and optimal.)

Tree Structure Organization



Node Contents

- Each intermediate node contains F keys (in lexicographic order) and $F + 1$ pointers.
- Each key K_j , $1 \le j \le F$, 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 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 $K_j \le K < K_{j+1}$.
- The value $F + 1$ is called the ***fan-out*** of the tree.

Search Technique

- Assume that we are looking for the tuple with key *V*.
- At each intermediate node, if $V < K_1$ follow the pointer P_0, else if $V \geq K_F$ follow the pointer P_F, otherwise, follow the pointer P_j such that K_j £ $V < K_{j+1}$
- The leaf nodes of the tree can be organized in two ways:
  - ✓ In ***key-sequence*****d** trees the tuples are contained in the leaves of the tree;
  - ✓ In ***indirect trees*** leaf nodes contain pointers to the tuples, that can be allocated by means of any other 'primary' mechanism (e.g.,, entry-sequenced or hash.)
- Sometimes the index structure is sparse; you may then locate a key value close to the value being sought, then carry out a sequential search.

Tuple Insertion Operations

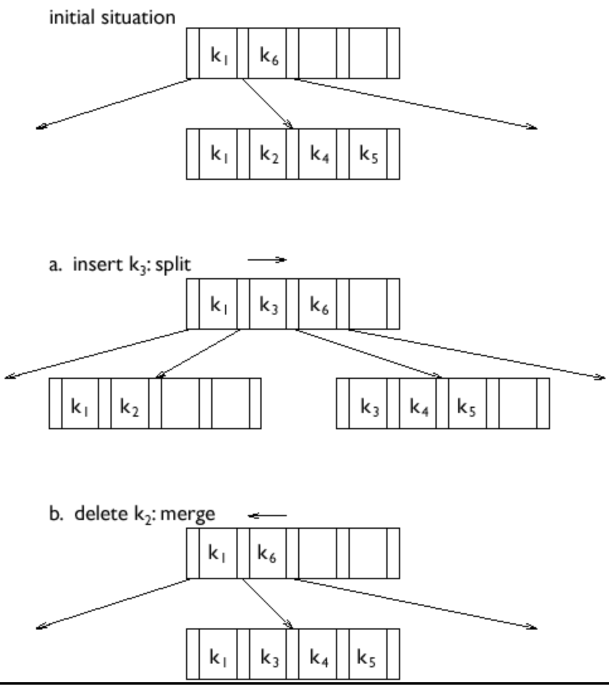
- An insertion operation searches up to a leaf page.
- When there is no free space on a page during an insertion, a ***split operation*** is necessary, allocating two leaf nodes in place of one.
- A split causes an increment in the number of pointers on the next (higher) level in the tree; this may cause further split operations.

Tuple Deletion Operations

- A deletion operation can always be carried out.
- When the key of a to-be-deleted tuple is in an intermediate node, it is best to recover the next key and substitute for it.
- When a deletion operation leaves two adjacent pages underused, they are conbined into a single page through 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 merges higher up in the tree.

Split and Merge Operations

- The update 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 split and merge operations makes it possible to maintain the tree balanced, with an average occupancy of each node higher than 50%.

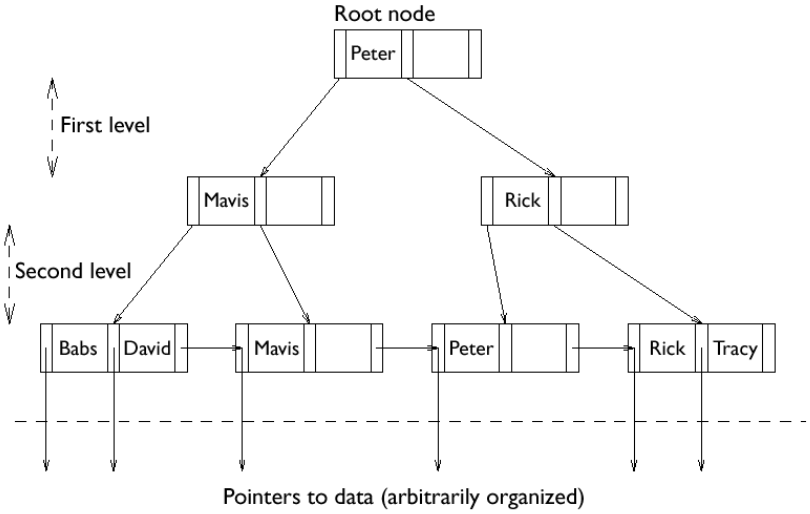


Split and Merge Operations

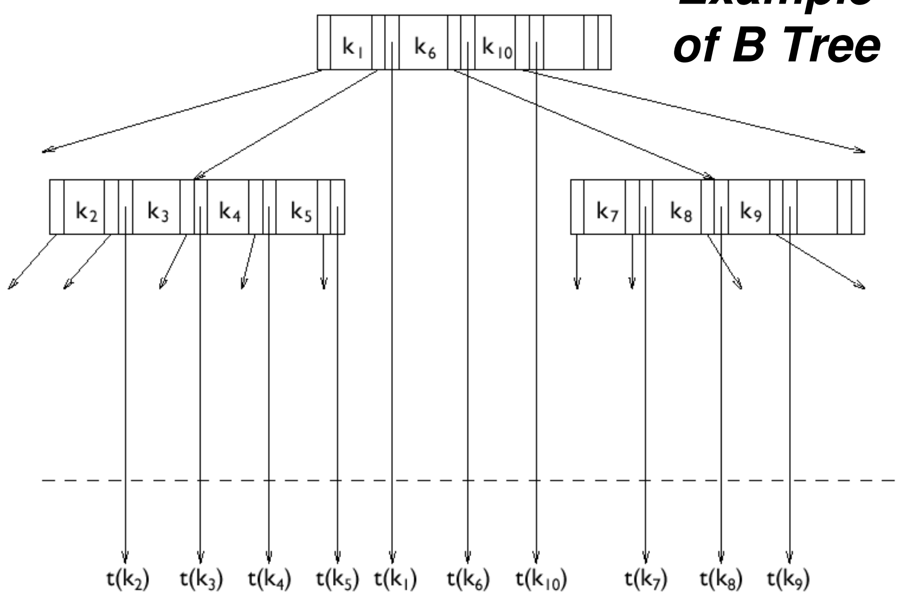
B and B+ Trees

- B+ trees are B trees, but their leaf nodes are linked into a chain, which connects them in the order imposed by the key.
- B+-trees support interval queries efficiently, and are widely used to implement relational DBMSs.
- On the other hand, B trees don't support sequential access method of leaf nodes.
- For B-trees, 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 subtree with keys greater than K_i and less than K_{i+1}.

Example of B+ Tree



Example of B Tree



Query Optimization

- A ***query optimizer*** is an important component of a DBMS architecture. An optimizer takes as input an SQL query and produces an ***access plan*** in 'object' or 'internal' format.
- Optimization steps:
  - ✓ Lexical, syntactic and semantic analysis, using the data dictionary;
  - ✓ Translation into an internal, algebraic form;
  - ✓ Algebraic optimization through transformations;
  - ✓ Cost-based optimization;
  - ✓ Code generation using the physical data access methods provided by the DBMS.

Approaches to Query Compilation

- Queries are often used multiple times.
- ***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 (e.g., new index), the compilation of the query is invalidated and needs to be redone.
- ***Compile and go*** -- involves compilation for immediate execution, with no storage of the compiled query.

Relation Profiles

- A ***profile*** contains quantitative information about a table and is stored in the data dictionary:
  - ✓ the cardinality (number of tuples) of each table *T;*
  - ✓ the size in bytes of each tuple of *T;*
  - ✓ the size 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;*
- A profile is periodically recomputed through appropriate system primitives (e.g., **`update statistics`**).
- Profiles are used in cost-based optimization to estimate the size of intermediate results produced by the query execution plan.

Internal Representation of Queries

- Queries are internally represented as trees whose:
  - ✓ Leaves correspond to physical data structures (tables, indexes, files)
  - ✓ Intermediate nodes represent data access operations that are supported by the physical structures
- These operations include sequential scans, orderings, indexed accesses and various types of join.

Scan Operation

- Performs a sequential access of all the tuples in a table, and performs on each tuple various operations of an algebraic or other nature:
  - ✓ Projection of a set of attributes;
  - ✓ Selection on a predicate (e.g., $A_i = v$);
  - ✓ Sort (ordering);
  - ✓ Insertions, deletions, and updates of the tuples that are being 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 available buffer space.

Indexed Access

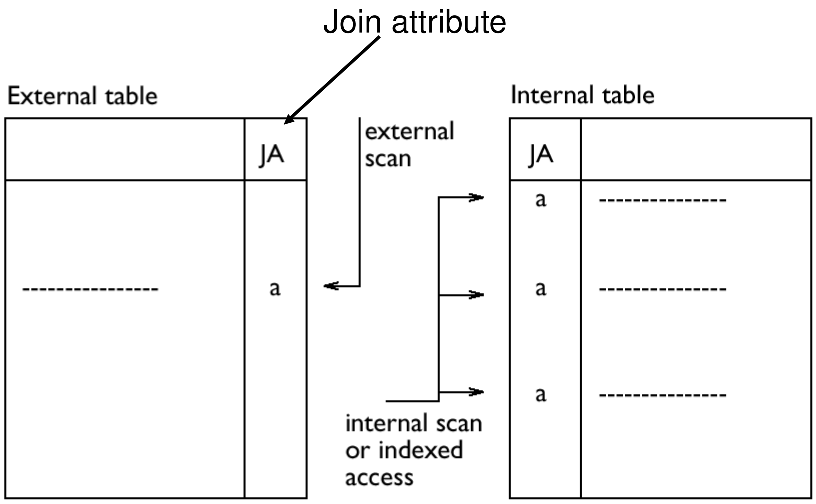
- Indexes are created by the database administrator to favor queries when they include:
  - ✓ simple predicates (of the type $A_i = v$)
  - ✓ interval predicates (of the type $v_1 \leq A_i \leq v_2$)

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

Join Methods

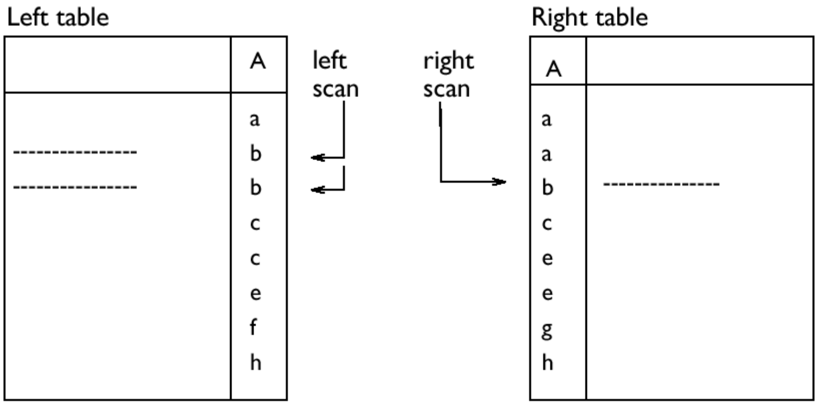
- Joins are the most costly operation for a DBMS.
- There are various methods for join evaluation, among them we note ***nested-loop***, ***merge-scan*** and ***hashed*** join operations.

Nested-Loop Join



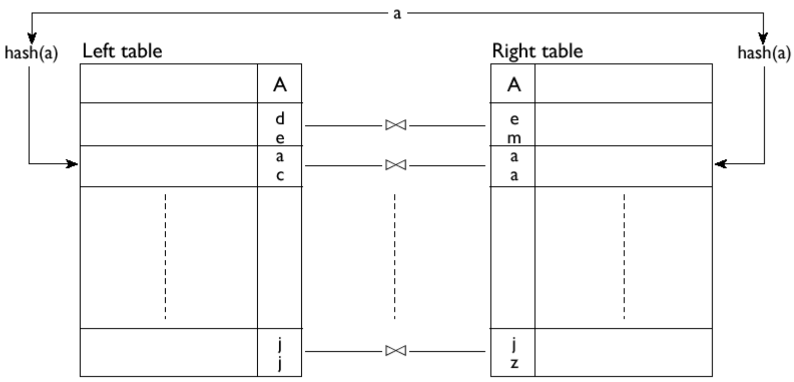
Merge-Scan Join

Both table must be ordered according to the join attribute



Hashed Join

- A hashing function *h* on the join attributes is used to store both tables



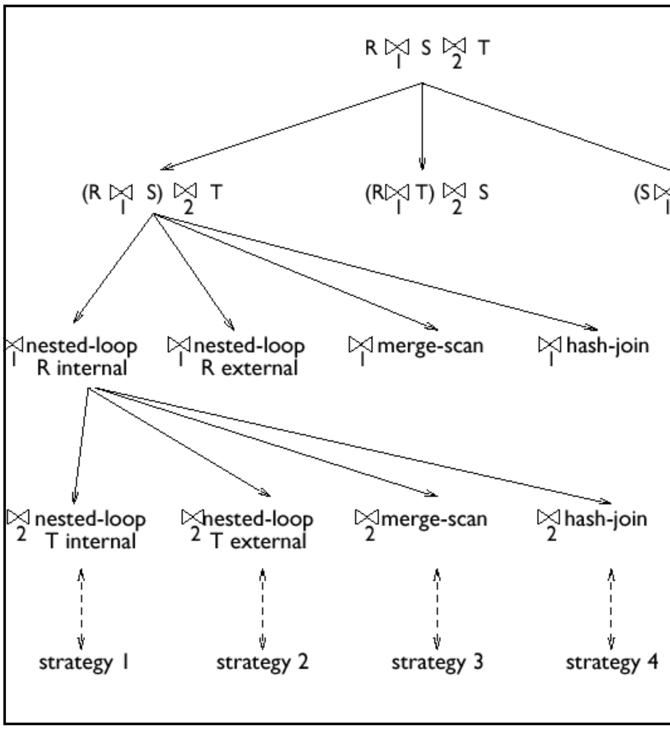
Cost-Based Optimization

- This is 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 order of join operations;
  - ✓ Options to be assigned to each operation, e.g.,, choosing a join method;
  - ✓ Use of parallelism and pipelining to improve performance.
- Further options are available in selecting a plan for distributed database query processing.

Approaches to Query Optimization

- The approach often used includes the following:
  - ✓ Make use of approximate cost formulas;
  - ✓ Construct a ***decision tree***, where each node corresponds to a 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, where cost is near that of an optimal solution.



Example of Decision Tree

Physical Database Design

- The final phase of the database design process.
- This phase takes as input the logical schema of a database along with predictions for the application load.
- The phase produces as output a physical schema, made up of the definitions of the relations and of the physical access structures used, along with related parameters.
- Depends on the features that are supported by the underlying DBMS.

Physical Design for Relational DBMSs

- Most relational DBMSs support index and tuple clustering.
- Physical design can be reduced to the activity of identifying indices for each relation.
- The key of a relation is usually involved in selection or join operations; for this reason, each relation normally supports a unique index on the primary key.
- Other indices are added to support frequent queries.
- If the performance is unsatisfactory, we can tune the system by adding or dropping indices.
- It is useful to check how indices are used by queries, by using the **`show plan`** command.

Indices in SQL

- Commands for creating or dropping an index 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*