DBMS Architecture

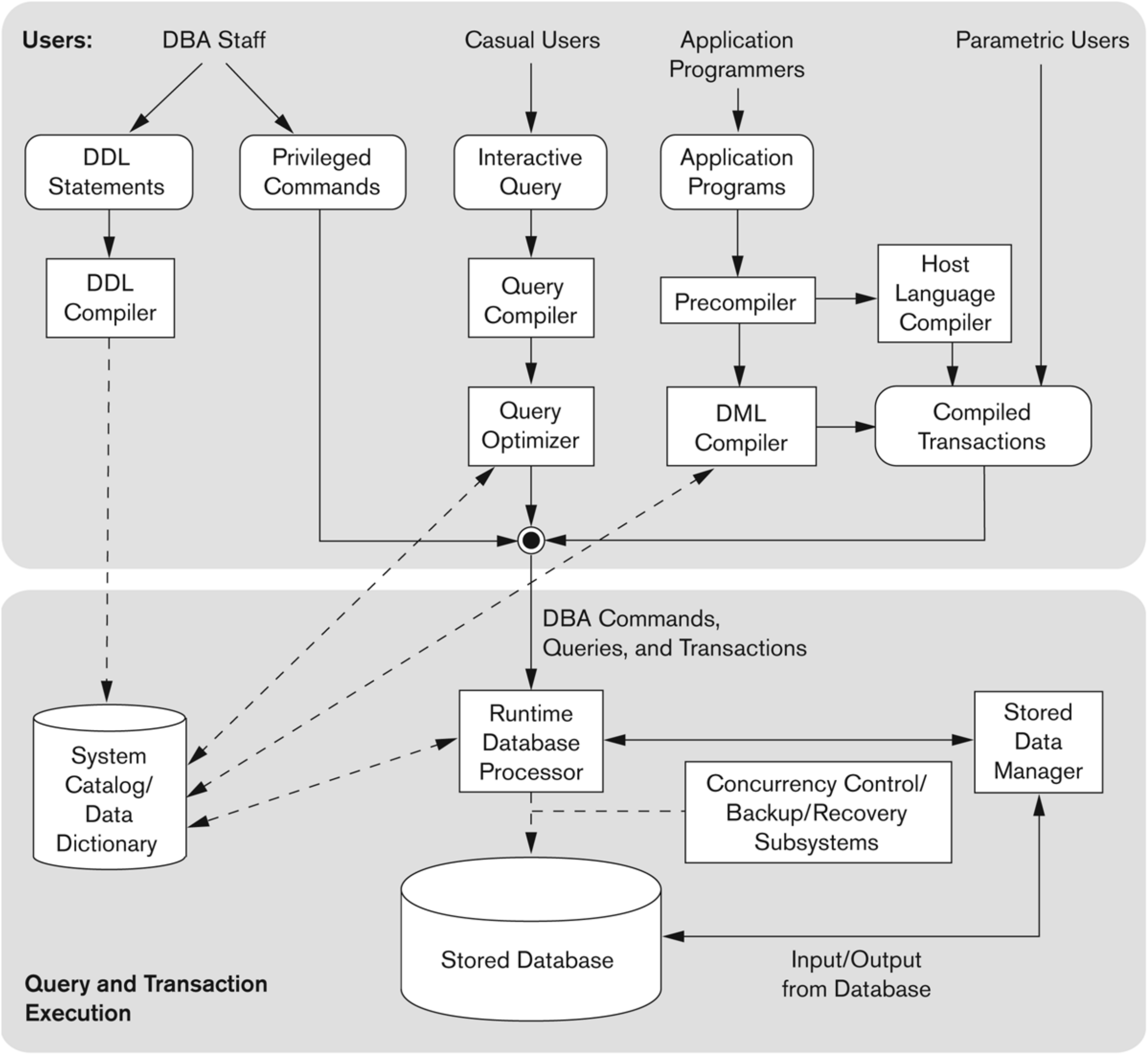
Prof. Daniel S. Kaster

Departamento de Computação
Universidade Estadual de Londrina
dskaster@uel.br

Outline

- Typical DBMS modules
- Query compilation and optimization
- Storage and indexing
- Query operator algorithms
  - Select
- Transaction processing and concurrency control
- Failure recovery

Typical DBMS Component Modules



Compilador

SQL sentence

↓

Lexical Analyzer

↓

Syntactic Analyzer

↓

Semantic Analyzer

↓

Canonical query tree

Query parsing (tokens)

Language syntax analysis

Verification of objects and identifiers (tables, attributes, etc.), type check, parameters, etc.

↓

Accesses the DBMS catalog

Query Tree

- A query tree is a representation of an SQL sentence as a tree of operators, in which:
  - Leaves are the input relations;
  - Internal nodes are the operators;
  - The execution occurs bottom-up, where each operation generates an intermediate result (table), which serves as input to the operation directly above in the query tree.
- The canonical query tree directly corresponds to how the query was written by the user

Example

- Consider the following query over the company example:

```
SELECT e.name
FROM employee e, works_on w, project p
WHERE p.pname = ‘aquarius’ AND p.pnum = w.pnum
  AND e.ssn = w.ssn AND e.bdate >= ‘1957-12-31’;
```

- Notice: pnum, pname and ssn are unique

Query Optimization

- The “optimization” step aims to generate a query tree equivalent to the initial one, but with more efficient execution
- Didactically, it consists of two parts:
  - Logical optimization: aims to explore the algebraic properties of relational queries
  - Physical optimization: aims to identify which algorithms can perform each operation most efficiently
    - Although the optimization process performs logical and physical optimization together

The Query Optimizator

Canonical query tree

Query Plan Enumerator

Performs query rewriting

Cost Evaluator

Estimates the cost using
the operators’ cost functions

Accesses the DBMS catalog
(Statistics)

Optimized query tree

Query Plan Enumerator

- The optimization step has to generate an execution tree equivalent to the initial one
  - To ensure producing the same result
- Therefore, this processing is limited to the transformations possible in a relational query
- According to the meaning of each operation, a set of transformation rules was defined
  - Following the strong mathematical foundation of the relational model

Equivalence Rules

- R1) Cascading σ

  $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

- R2) Commutativity of σ

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

- R3) Cascading π

  $\pi_{list1}(\pi_{list2}(\ldots(\pi_{listn}(R))\ldots)) \equiv \pi_{list1}(R)$

- R4) Commutativity between σ and π

  $\pi_{A1, A2, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \ldots, An}(R))$

- R5) Commutativity of $\times$ and $\bowtie$

  $R \times S \equiv S \times R$ e $R \bowtie_c S \equiv S \bowtie_c R$

Equivalence Rules(2)

- R6) Commutativity of σ with $\bowtie$ (or $\times$, $\cup$, $\cap$)

  $\sigma_c(R \bowtie S) \equiv \sigma_c(R)$ S ou $\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$
- R7) Commutativity of π with $\bowtie$ (or $\times$)

  $\pi_L(R \bowtie S) \equiv \pi_L((\pi_{A1,\ldots,An,A1',\ldots,An'}(R)) \bowtie (\pi_{B1,\ldots,Bn,B1',\ldots,Bn'}(S))$
- R8) Commutativity of set operators

  $\cup$ and $\cap$ are commutative, but – is not
- R9) Associativity of $\cup$, $\cap$, $\times$ and $\bowtie$

  Let θ be the operator: $(R \theta S) \theta T \equiv R \theta (S \theta T)$
- R10) Conversion of $(\sigma_c(R \times S))$ into $(R \bowtie_c S)$

Logical Optimization

- Logical optimization performs what is called query rewriting, aiming to obtain a more efficient query tree
- Fundamentals of logical optimization algorithms:
  - Apply SELECT and PROJECT operations whenever possible before binary operations, as binary operations are generally multiplicative functions on input sizes;
  - Replace sets of equivalent operations with less costly operations;
  - Define a suitable order for executing binary operations, as the order can significantly change the cost.

A Generic Optimization Algorithm

1) Break conjunctive selections into a cascade of selections, to allow moving the selections (R1)

2) Push down selections whenever useful (R2, R4, R6)

3) Convert $\sigma_c(R \times S)$ into $(R \bowtie_c S)$ (R10)

4) Rearrange leaf nodes to optimize binary operations, positioning relations with more restrictive selection operations as leaves (R5, R8, R9)

5) Push down projections in the query tree (R3, R4, R7)

Physical Optmization

- The physical optimization step aims to identify the most efficient algorithms that can be used to execute each operator in the query tree to generate a physical execution plan with minimum cost
- Each algorithm has requirements to be selected
  - For example, the existence of indexes, the organization of the data file, the amount of memory available for the algorithm, etc.
- The positioning of an operator in the query tree during optimization impacts the availability of algorithms for the operator, tying together logical and physical optimizations
  - For example, you can only use indices of operands that are leaves of the execution tree (i.e., original relations)

Algorithms for Query Operators

- Many query operators have several alternative algorithms to execute the operator
- Physical optimization involves selecting the cheapest algorithm for each case
  - The cost of the algorithms is usually stated based on the number of disk accesses (or number of blocks accessed)

Algorithms: Select

- A trivial strategy to implement the select operator performs a sequential scan on the input relation and adds the tuples satisfying the condition to the output relation

```
seqScanSelect(Relation R, Condition condition) {
  for each tuple r in R do {
    if (condition(r)){
      result.add(r);
    }
  }
  return result;
}
```

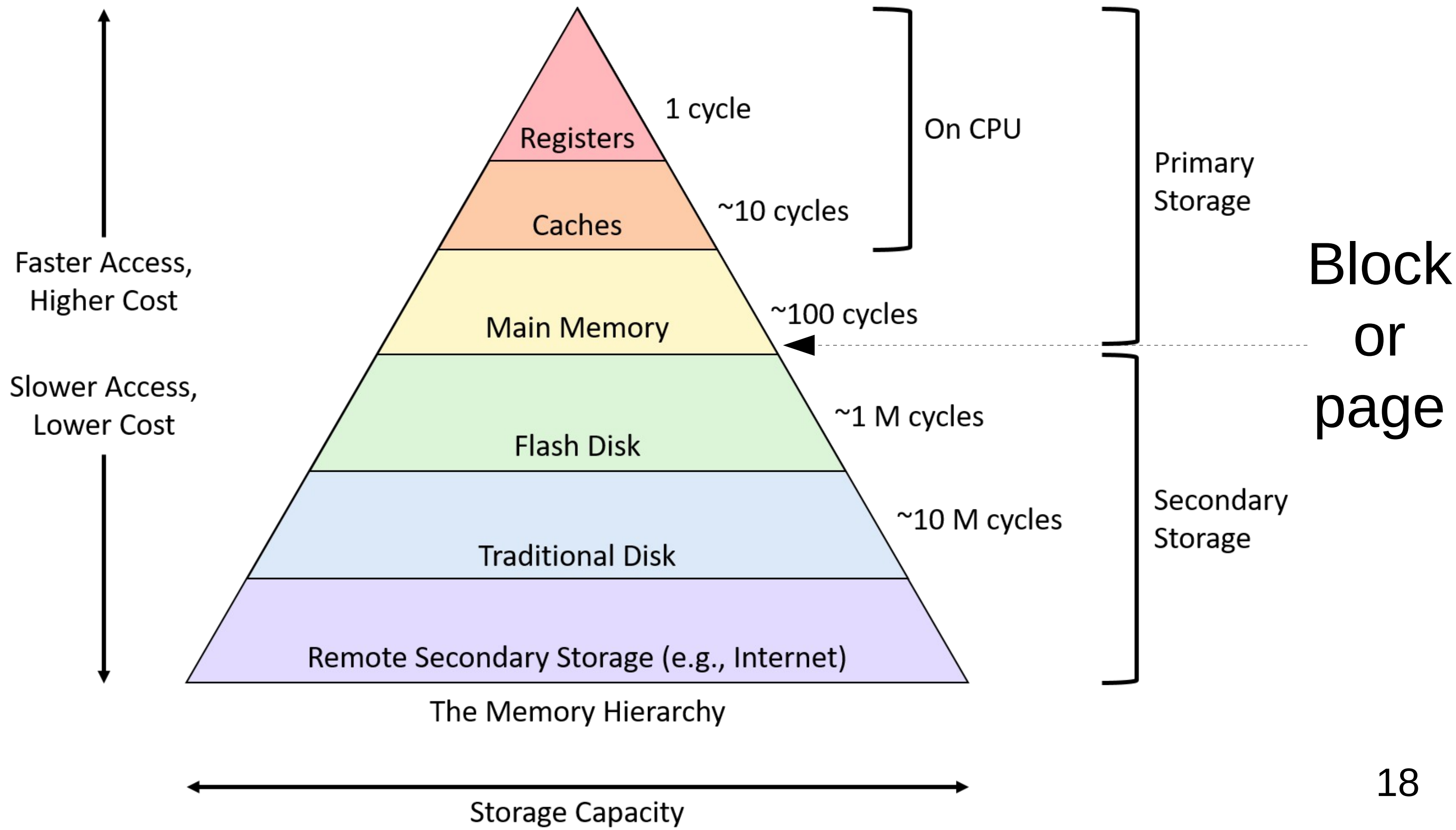
- However, this strategy is not suitable for accessing the secondary memory

DBMS Storage

- A database is stored as a collection of files
  - Each file is a sequence of records
  - A record is a sequence of values for fields/attributes
- This material considers the following simplification
  - Each persistent database object (table, index, etc.) is stored in a different file
  - Each file has records of the same data type

Data Transfer in the Memory Hierarchy

- Memory hierarchy relies on transfer units between levels to ease block replacement



The Memory Hierarchy

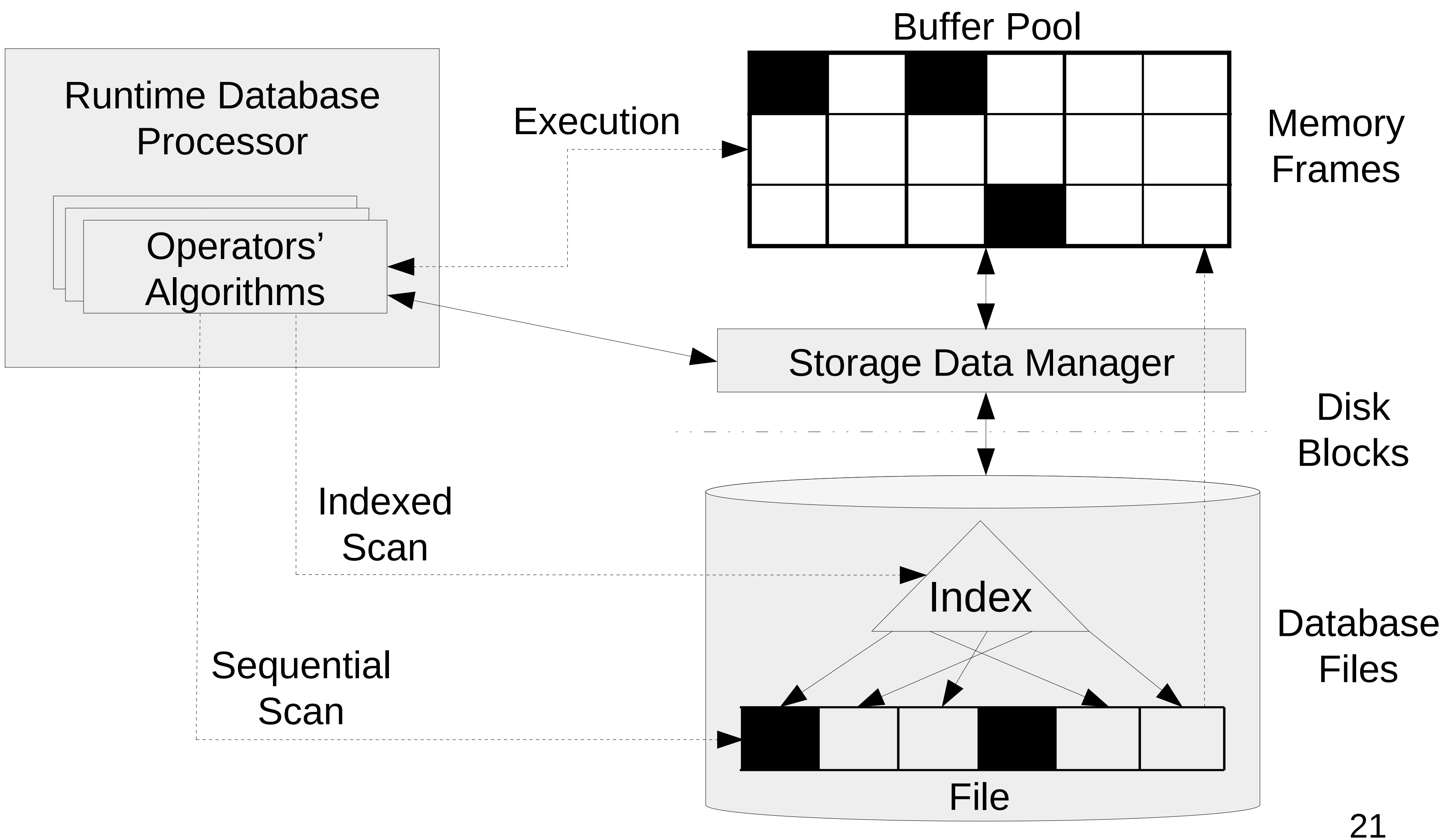
Paging of Files

- We cannot access a single byte on a disk
  - To access a byte, we have to access a block
- Files are organized in blocks/pages to benefit from this
  - The database files and the data structures that handle them are designed to reduce block accesses

Placing Records in Files

- Record: collection of related data values or items
  - Values correspond to record field
  - Example: a row in a table
- Large objects
  - Unstructured objects (BLOBs) or semiestructured objects (CLOBs)
  - Examples: an image file or a JSON file
- Allocated to disk blocks in both cases
  - Blocking factor
    - Average number of records per block for the file

The Storage Data Manager



Buffer Pool

- Memory buffers are also organized in fixed-size blocks
- DBMSs use different memory buffers to store different data
  - Data block buffers, log buffers, etc.
- Different buffers may use distinct block sizes and/or block substitution policies

Buffer Management and Replacement Strategies

- Buffer management information
  - Pin count
    - To pin a frame in the buffer pool
  - Dirty bit
    - To indicate a modified block
- Buffer replacement strategies
  - To free frames in the buffer pool
  - Strategies
    - Least recently used (LRU)
    - First-in-first-out (FIFO)

Select – Sequential Scan

- Operator algorithms usually rely on paged accesses

```
seqScanSelect(Relation R, Condition condition) {
  for each block b in R do
    for each tuple r in b do {
      if (condition(r)){
        result.add(r);
      }
    }
  }
  return result;
}
```

- The Storage Data Manager apply techniques to speed up the proccess, including *caching*, *prefetching*, *bulk access*, and others

Select – Indexed Scan

- The indexed scan is employed when there is an index on one or more attributes in the select condition
  - There are different algorithms according to the index types available

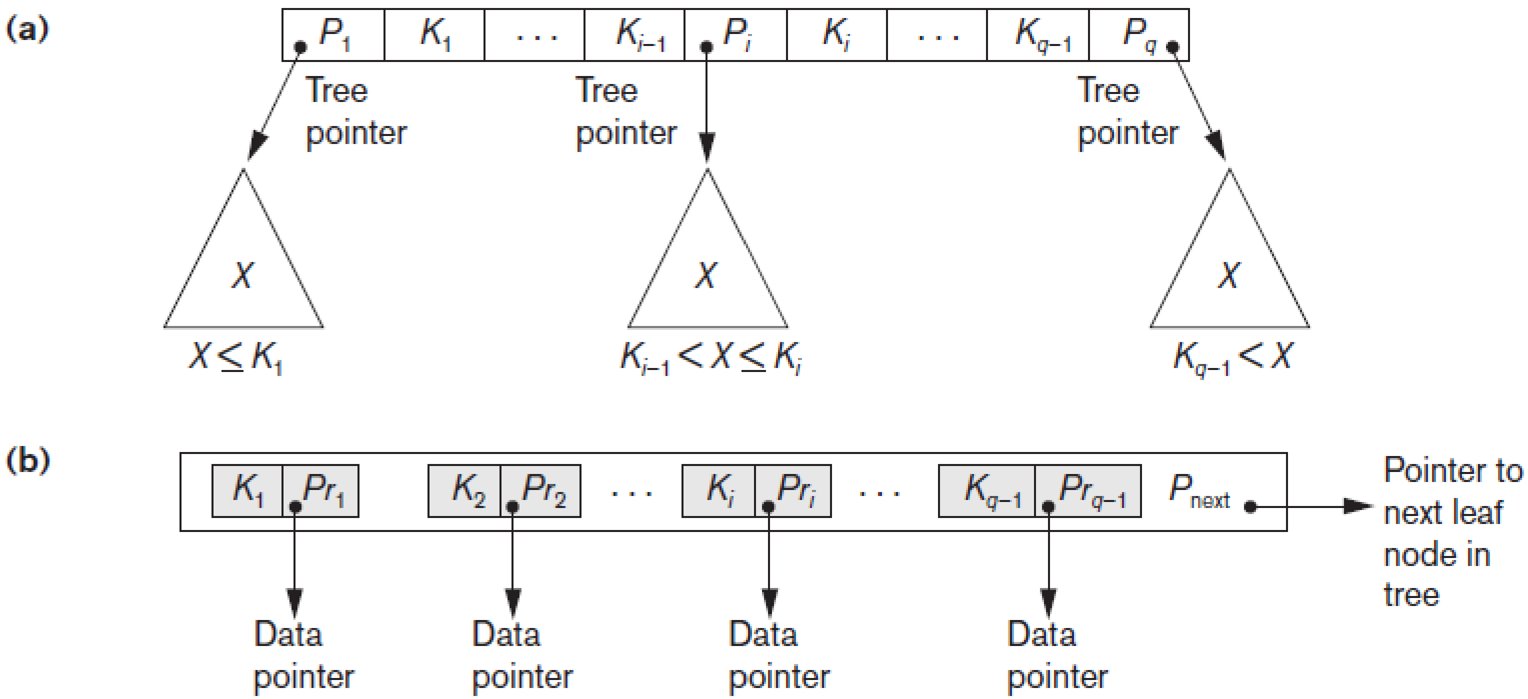
Secondary and Clustering Indexes

- Secondary index
  - The most common type of indexes
  - A table can have several secondary indexes
    - Syntax: CREATE [UNIQUE] INDEX my_ix ON table(attribute [<, attribute2, ...>]);
- Clustering index
  - Sorts physically the data file according to the indexing attribute and creates the index
    - To benefit from the block factor in range queries
    - The drawback is the overhead to maintain the file sorted
  - One clustering index per table
    - Syntax: CREATE [UNIQUE] CLUSTERING INDEX my_ix ON table(attribute [<, attribute2, ...>]);

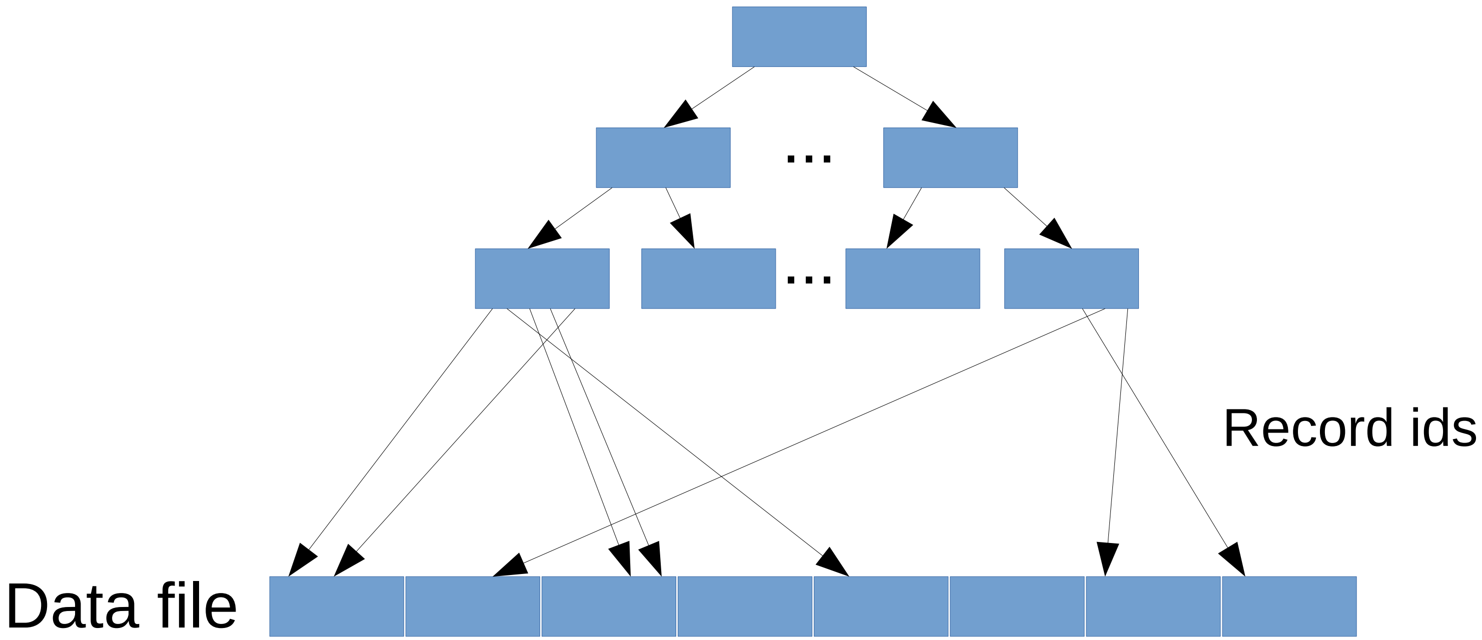
The B+ -Tree Dynamic Multilevel Index

- Disk-based search tree
  - Nodes have the size of a block
  - Given two search keys $K_{i-1} < K_i$, the elements stored in the corresponding subtree are $K_{i-1} < X \leq K_i$
  - Tree pointers are pointers to blocks: data file + block
  - Record pointers (rids) have the structure: data file + block + record
- Balanced tree
  - Reorganized at each insert or delete using split and unsplit of nodes
  - Nodes are also at least half full, except the root node
  - Bottom-up construction

B+ -Tree: Node Types

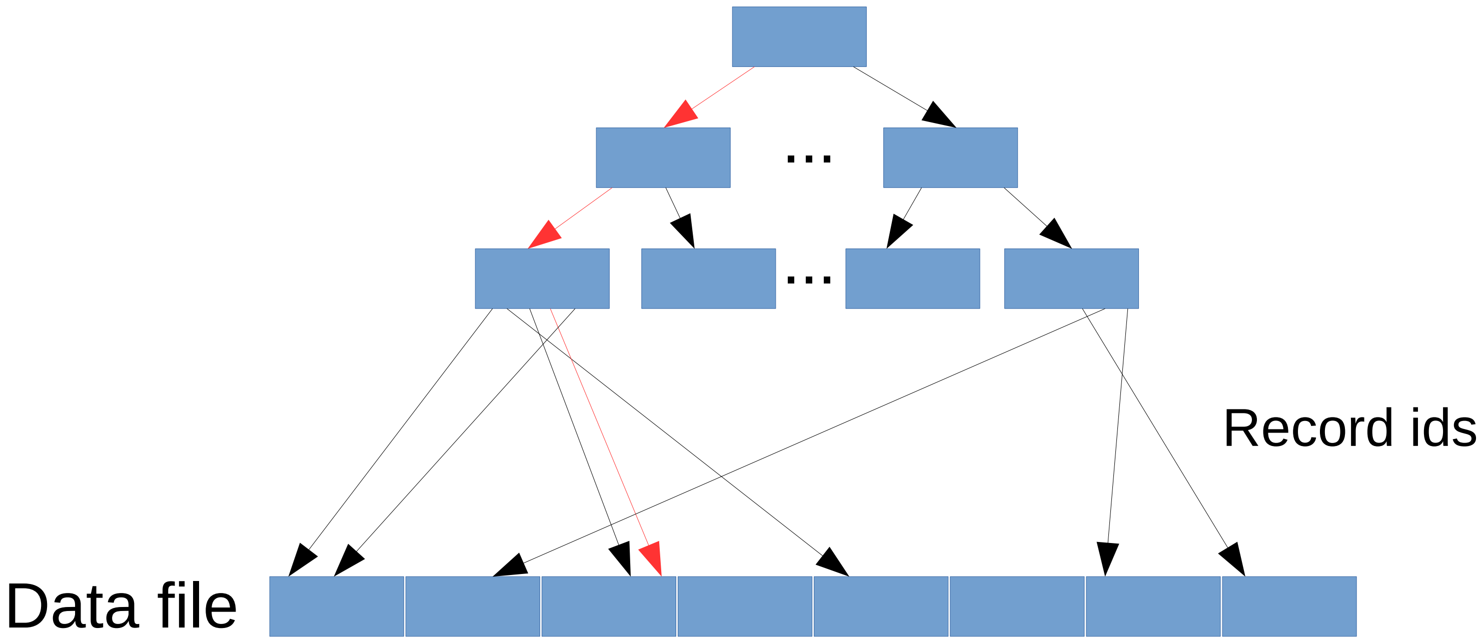


Secondary Indexes



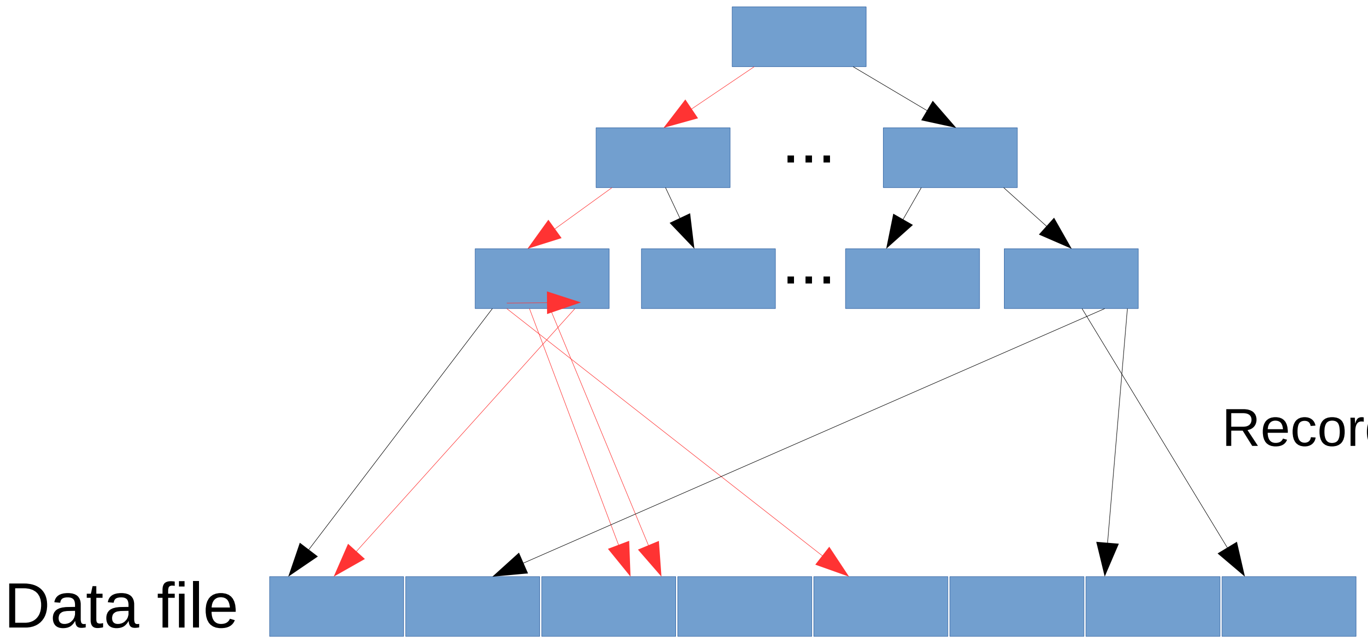
Indexed Scan using Secondary Indexes

- Unique scan: WHERE attribute = value (attribute is key)

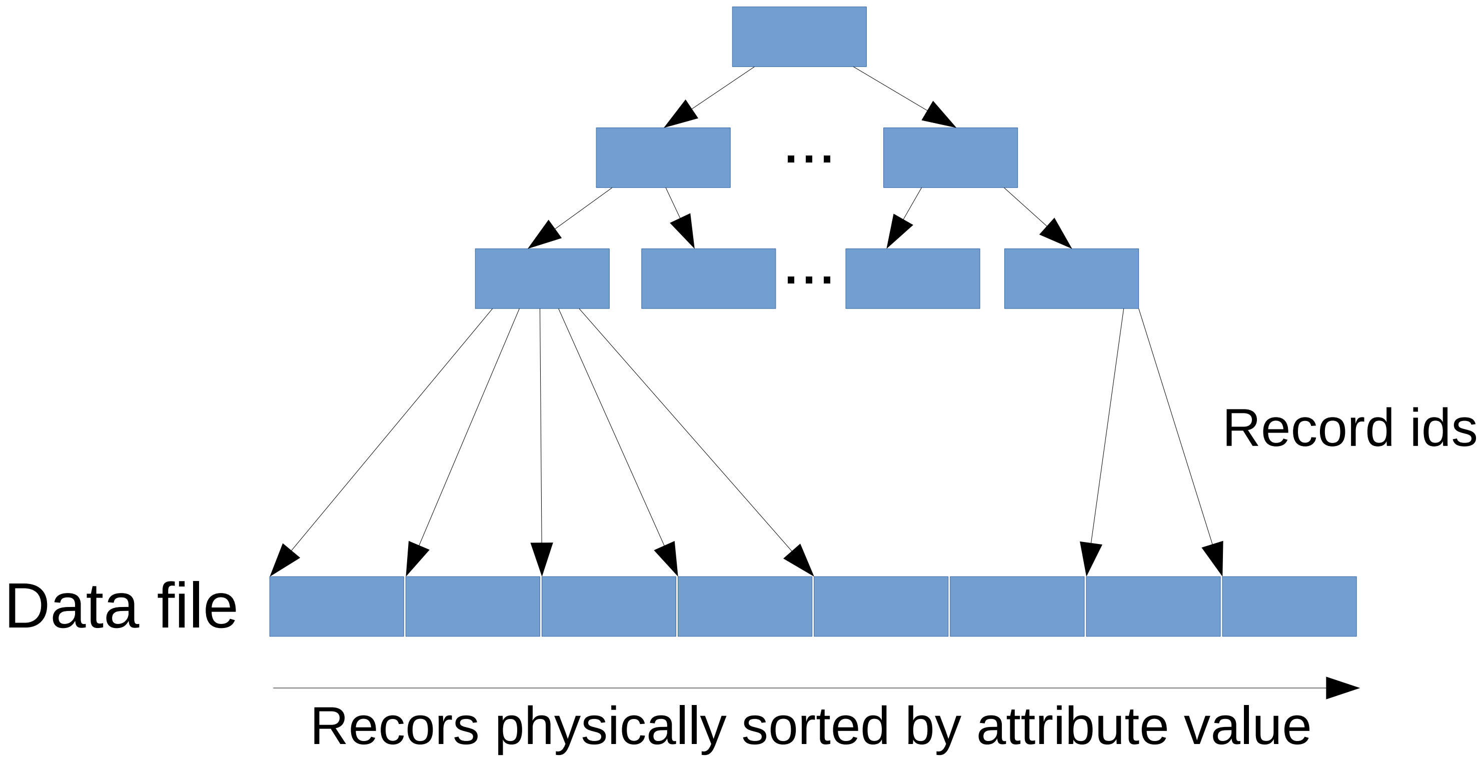


Indexed Scan using Secondary Indexes (2)

- Range scan: WHERE attribute = value (attribute is not key)
  WHERE attribute >= value, etc.

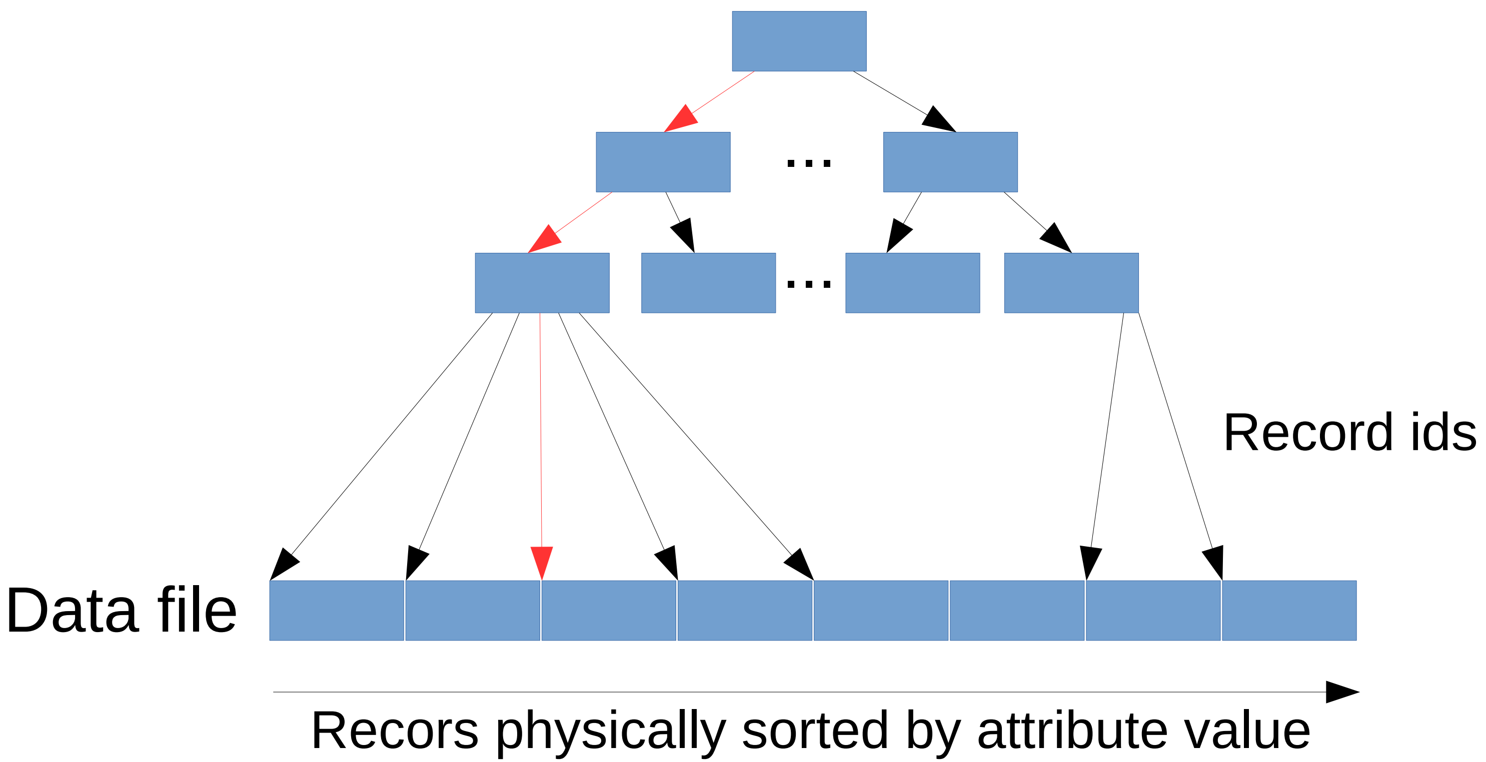


Clustered Indexes



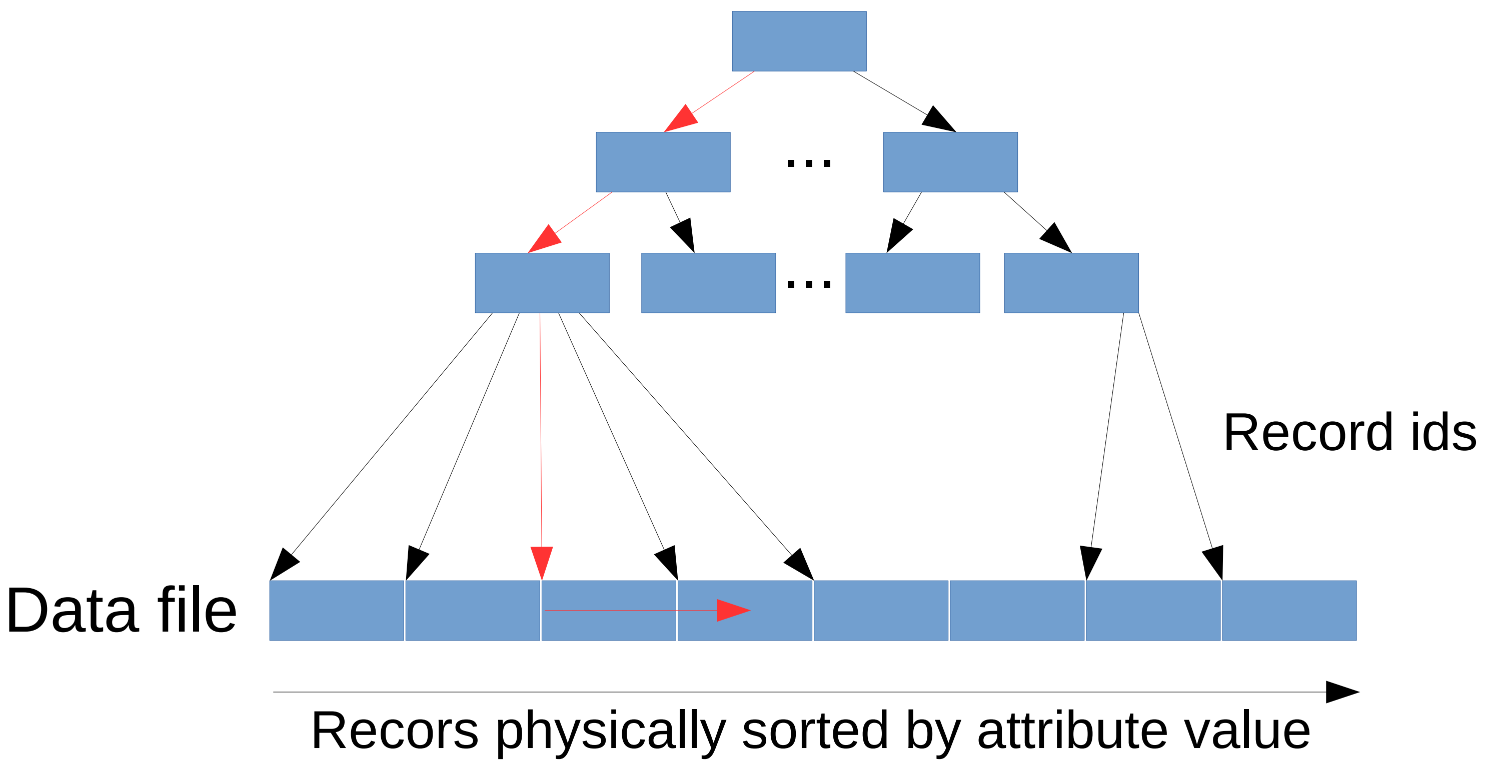
Indexed Scan using Clustered Indexes

- Unique scan: WHERE attribute = value (attribute is key)



Indexed Scan using Clustered Indexes (2)

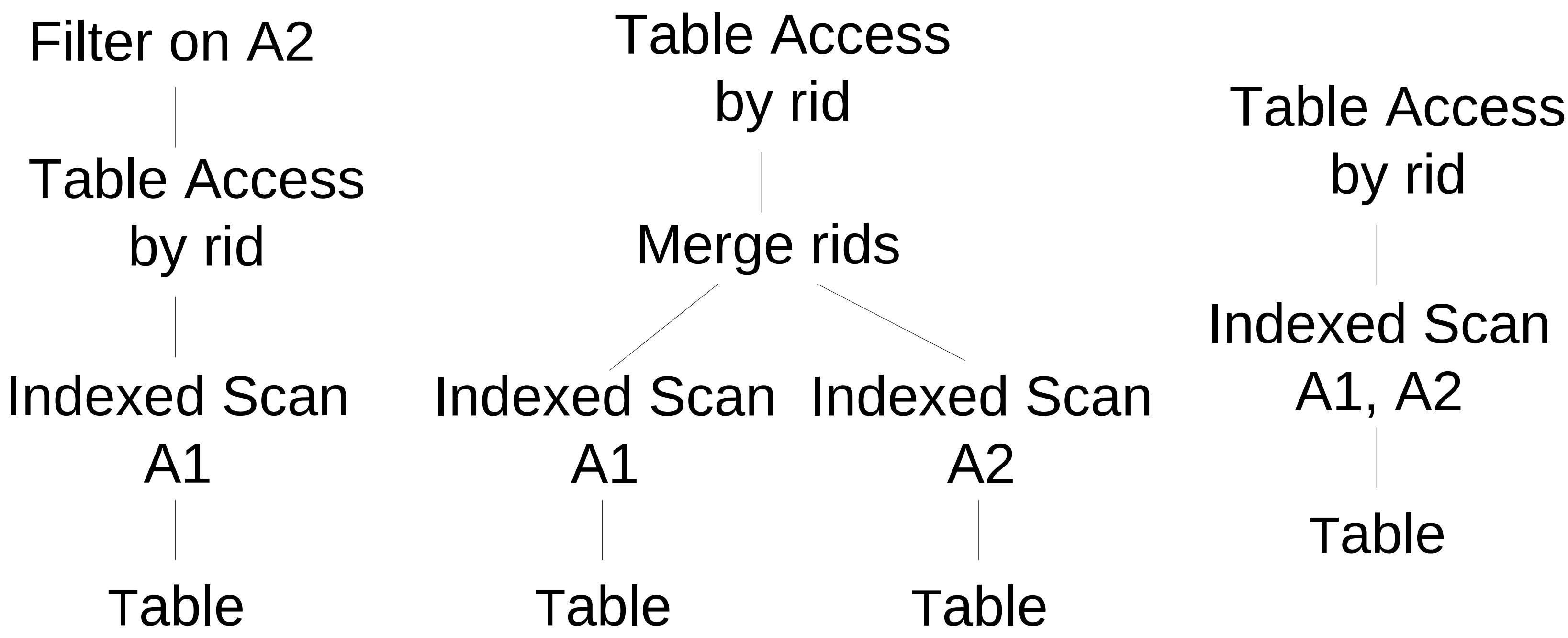
- Range scan: WHERE attribute = value (attribute is not key)
  WHERE attribute >= value, etc.



Select with Composite Conditions

- The select operator's condition may involve more than one attribute from the same table

  WHERE t.A1 = v1 AND t.A2 = v2

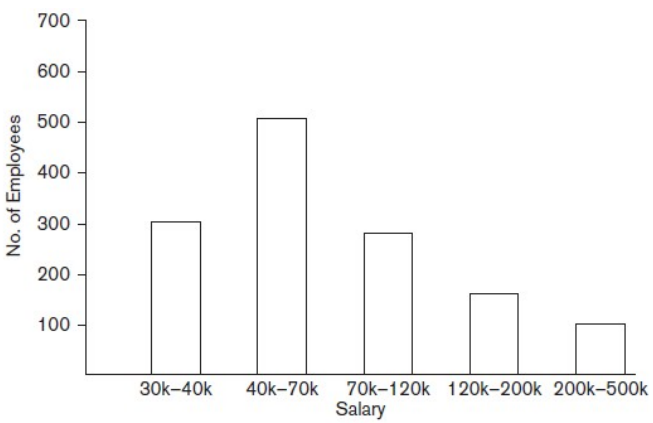


How to Decide the Algorithm to Choose?

- Information stored in DBMS catalog and used by optimizer to compute the cost of the algorithms
  - Tables: # of rows, # of disk blocks, average record size, organization (e.g., sorted or not sorted), etc.
  - Indices: # of levels, # of leaf blocks, uniqueness, etc.
  - Attributes: # of distinct values, minimum and maximum values, etc.

Attribute Selectivity

- Allows calculation of selection cardinality
  - Estimated number of records that satisfy a selection condition on that attribute
    - Examples: $\sigma_{salary\ =\ 120k}(Emp)$; $\sigma_{salary\ >=\ 120k}(Emp)$
  - Statistics:
    - Minimum, maximum and # of distinct values
    - Histogram



Transaction Processing

- A database is a shared resource accessed by many users and processes concurrently
- Not managing this concurrent access to a shared resource will cause problems
  - Problems due to concurrency
  - Problems due to failures
- Transaction processing comprehends techniques to manage a database in a concurrent multi-user environment

Transactions

- A transaction (sequence of executing operations) may be:
  - Stand-alone, specified in a high level language like SQL submitted interactively, or
  - More typically, embedded within application program
- Transaction boundaries: Begin_transaction and End_transaction
  - Application program may include specification of several transactions separated by Begin and End transaction boundaries
  - Transaction code can be executed several times (in a loop), spawning multiple transactions
- Transactions can end in two states:
  - Commit: transaction successfully completes and its results are committed (made permanent)
  - Abort: transaction does not complete and none of its actions are reflected in the database

What Can Go Wrong?

	T1 at ATM window #1		T2 at ATM window #2
1	read_item(savings);	a	read_item(chequing);
2	savings = savings - $100;	b	chequing = chequing - $20;
3	write_item(savings);	c	write_item(chequing);
4	read_item(chequing);	d	dispense $20 to customer;
5	chequing = chequing + $100;		
6	write_item(chequing);		

- System might crash after transaction begins and before it ends:
  - Money lost if between 3 and 6 or between c and d
  - Updates lost if write to disk not performed before crash
- Chequing account might have incorrect amount recorded:
  - $20 withdrawal might be lost if T2 executed between 4 and 6
  - $100 deposit might be lost if T1 executed between a and c
    - In fact, same problem if just 6 executed between a and c

ACID Properties

- Atomicity
  - A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all
- Consistency
  - A correct execution of the transaction must take the database from one consistent state to another
- Isolation
  - Even though transactions are executing concurrently, they should appear to be executed in isolation
  - That is, their final effect should be as if each transaction was executed in isolation from start to finish
- Durability
  - Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure

ACID Properties (2)

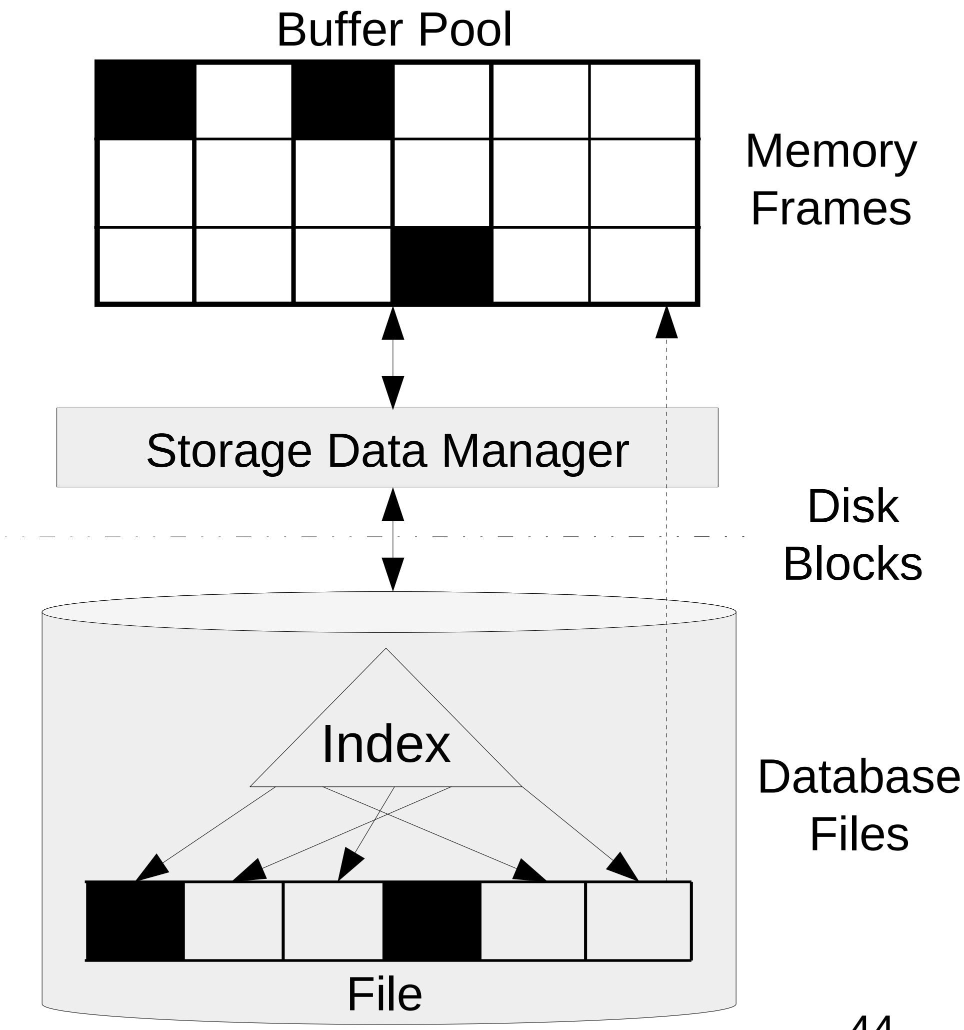
- Enforcement of ACID properties
  - Consistency (and application program correctness)
    - Responsibility of the database constraint system
  - Isolation
    - Responsibility of the concurrency control mechanism
  - Atomicity and Durability
    - Ensured by the recovery system

Transaction Processing Model

- Simple database model:
  - Database: collection of named data items
- Granularity (size) of each data item immaterial
  - A field (data item value), a record, or a disk block
  - TP concepts are independent of granularity
- Basic operations on an item X:
  - read_item(X): Reads a database item X into a program variable
    - For simplicity, assume that the program variable is also named X
  - write_item(X): Writes the value of program variable X into the database item named X
- Read and write operations take some amount of time to execute

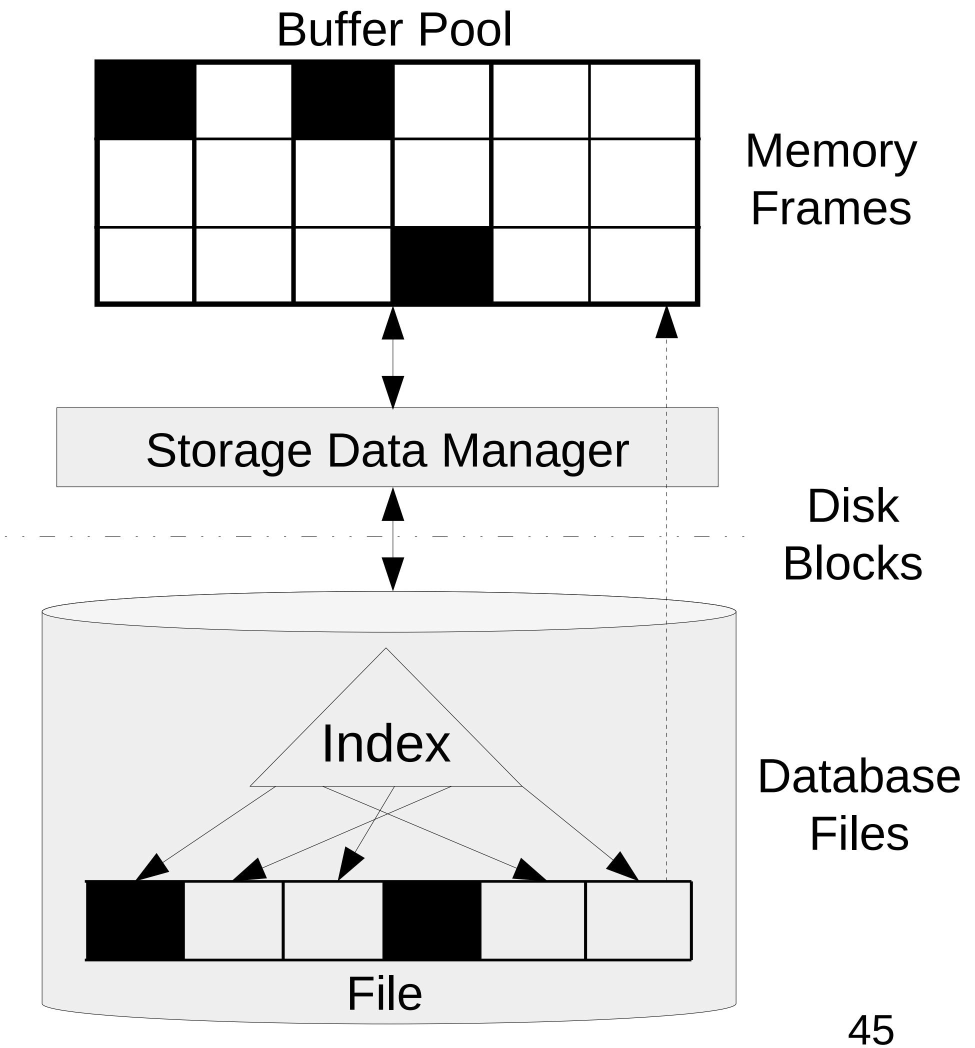
Read and Write Operations

- **read_item(X)** includes the following steps:
  - Find the address of the disk block that contains item X
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
  - Copy item X from the buffer to the program variable named X



Read and Write Operations (2)

- **write_item(X)** includes the following steps:
  - Find the address of the disk block that contains item X
  - Copy that disk block into a buffer in main memory (if it is not already in some main memory buffer)
  - Copy item X from the program variable named X into its correct location in the buffer
  - Store the updated block from the buffer back to disk
    - Either immediately or, more typically, at some later point in time



Concurrency Control Protocols

- Two-phase locking protocols
  - Lock data items to prevent concurrent access
- Timestamp ordering protocols
  - Assign a unique identifier for each transaction
  - Apply rules to control how transactions access items according to the timestamps
- Multiversion techniques
  - Kept several versions of an item
  - Accept some read operations that would be rejected in other techniques by reading an older version of the item while maintaining serializability
- Optimistic techniques
  - Perform no checking is done while the transaction is executing
  - Execute a validation phase to check whether any of transaction's updates violate serializability, and commit or abort transactions based on result

Failure Recovery

- Several approaches exist each using some subset of recovery concepts
  - Write-ahead logging
  - In-place versus shadow updates
  - Immediate versus deferred update

The UNDO-REDO Approach

- After a crash:
  1) UNDO transactions that had not committed
     - To ensure Atomicity
     - Partial results of uncommitted transactions should be discarded
  2) REDO committed transactions
     - To ensure Durability
     - Modified blocks in memory might have not be written back to disk
- Uses a system log with write-ahead logging policy and checkpointing
- Example: the AIRES recovery algorithm, used in many IBM relational database products

System Log

- Append-only file
  - Keep track of all operations of all transactions
  - In the order in which operations occurred
- Stored on disk
  - Persistent except for disk or catastrophic failure
  - Periodically backed up
  - Guard against disk and catastrophic failures
- Main memory buffer
  - Holds records being appended
  - Occasionally whole buffer appended to end of log on disk (flush)

System Log Entries

- [start_transaction, T]
  - Transaction T has started execution.
- [write_item, T, X, old_value, new_value]
  - T has changed the value of item X from old_value to new_value.
  - Before Image (old_value) needed to undo(X)
  - After Image (new_value) needed to redo(X)
- [commit, T]
  - T has completed successfully and committed
  - T’s effects (writes) must be durable
- [abort, T]
  - T has been aborted
  - T’s effects (writes) must be ignored and undone

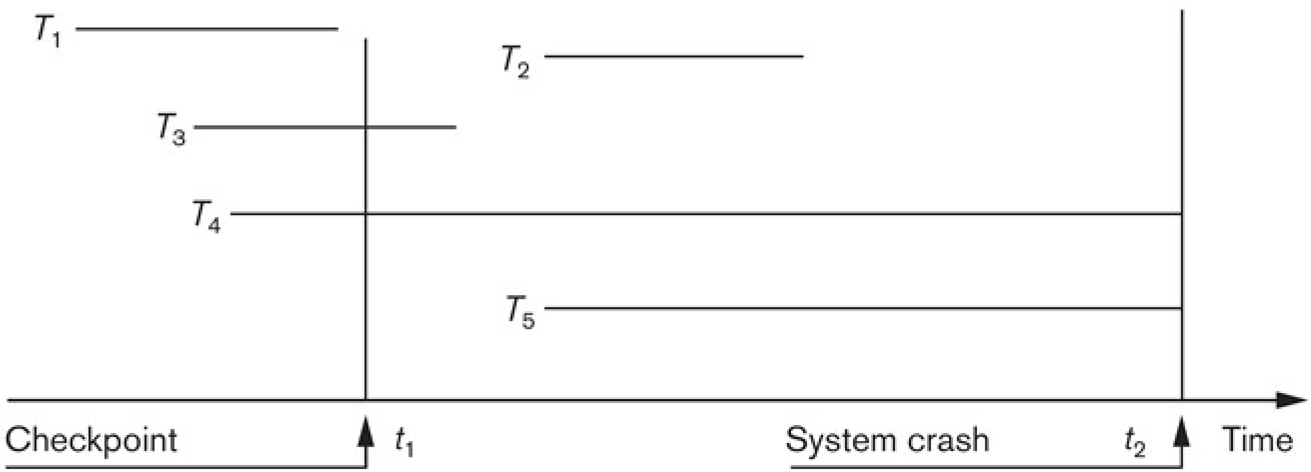
Write-Ahead Logging (WAL)

- Used to ensure that the log is consistent with the database, and to ensure that the log can be used to recover the database to a consistent state
- Two rules:
  - Log record for a page must be written before corresponding page is flushed to disk
    - For Atomicity, so that each operation is known and can be undone if necessary
  - All log records must be written before commit
    - For Durability, so that the effect of a committed transaction is known
- A transaction is said to be committed (reached the commit point) when:
  - All of its operations are executed, and
  - All its log records are flushed to disk

Checkpointing

- Employed to save redo effort
- Occasionally flush data buffers
  1) Suspend execution of transactions temporarily
  2) Force-write modified (dirty) buffer data to disk
  3) Append [checkpoint] record to log
  4) Flush log to disk
  5) Resume normal transaction execution
- During recovery, redo required only for log records appearing after [checkpoint] record

Transaction Status for Recovery



- UNDO/REDO
  - T1: do nothing
  - T2, T3: REDO
  - T4, T5: UNDO

The UNDO/REDO Recovery Process

1) ANALYSIS
   a) Scan log from the checkpoint
   b) Identify the transactions to be redone and undone
2) REDO (roll-forward)
   a) Scan the log from head to tail (forwards in time)
   b) Redo updates of committed transactions
   c) Use after image for new values
3) UNDO (roll-back)
   a) Scan log from tail to head (backward in time)
   b) Restore before image by undoing updates of active transactions, writing compensating log records for undone operations
      - e.g., the compensation action for an insert is a delete
   c) In case of crash during recovery, the algorithm redoes up to the last compensation operation and starts undoing from the previous one