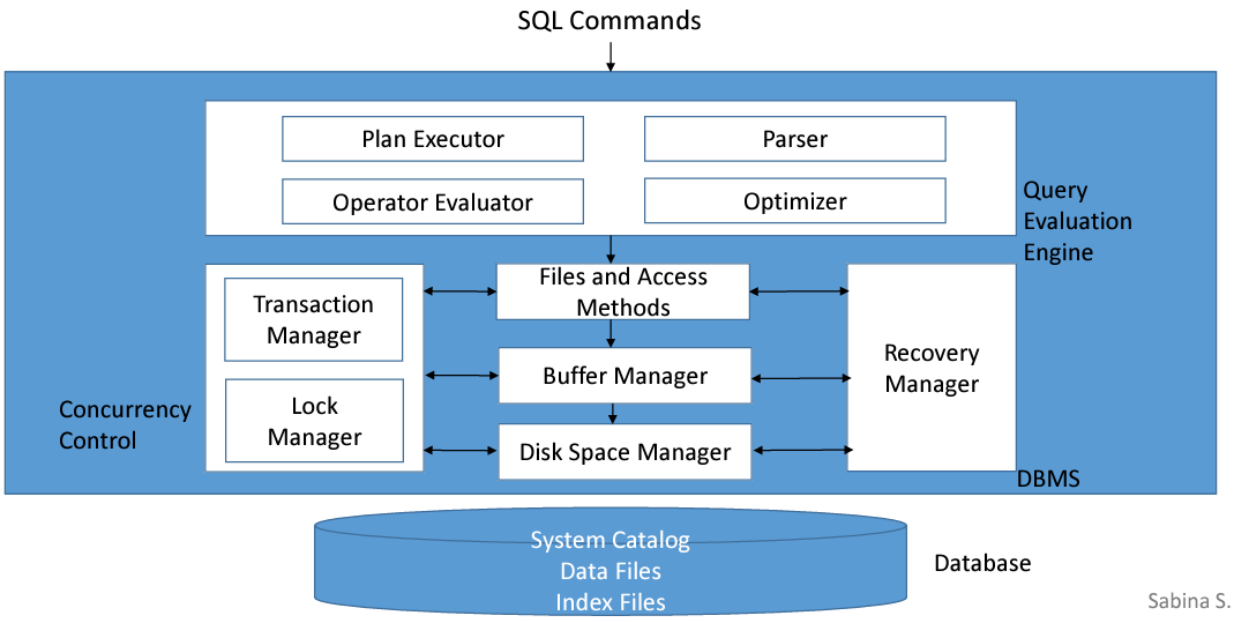
Database Management Systems

- Locking Protocols
  - 2PL = Two-Phase Locking
  - Strict 2PL
- Deadlock
  - Prevention
  - Detection
- ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Conflicts
  - Write - Read conflict = Read Uncommited / dirty read
  - Read - Write conflict = Unrepeatable Reads
  - Write - Write conflict = overwriting uncommited data = Blind Write
  - Phantom Read
- ISOLATION LEVEL
  - Read Uncommited
  - Read Commited
  - Repeatable Reads
  - Serializable
- Fragmentation
  - Horizontal
  - Vertical
- Access path
  - Indexing
- Relation Algebra
  - Projection
  - Selection
- Data Replication
  - Synchronous
  - Asynchronous
- Reduction Factor
- Writing Objects
- ARIES
- Transaction Table
- Dirty Page Table
- Reduction Factor

Lecture 1

DBMS Arhitecture



Concurrency in a DBMS

- scenarios

1. Agent TA1 processes customer C1 request for Flight 10
   ⇒ system gets DB tuple: <flight: 10, available seats: 1> → TA1
   ⇒ C1 takes time to decide
   TA2 does the same for C2 on Flight10
   ⇒ system gets DB tuple <flight:10, availables seats:1> → TA2
   ⇒ C2 books immediately ⇒ system updates: <flight:10, available seats:0>
   THEN
   → C1 decides to book the flight
   → TA1 is working with that same Tuple
   → Flight 10 last seat was sold TWICE

⇒
PROBLEM ⇒ Need to forbidd access in some cases

- What happens if access is forbidden?
  TA2 can only access the tuple after TA1 finished working with it
  ⇒ if C1 decide to buy, TA2 will retrieve the tuple <flight: 10, available seats: 0>
- NEW PROBLEM: Unacceptable delays WHEN Concurrent Execution is Forbidden→ If DB access is forbidden, if TA3 tries to book a ticket for C3 on Flight 11, he can't do that (even though TA1's operation won't affect Flight 11)

⇒ Concurrent Execution must be allowed , but one needs to keep in mind the effects of interleaving user programs

⇒ TRANSACTIONS

Transactions

- Transaction = any one execution of a program in a DBMS
- sequence of one or more operations : read / write / commit / abort
- Final operation: commit / abort
- Operations:
  - read(T, I)
  - write(T, I)
  - commit(T,I) - successful completion: Transaction T's changes are permanent
  - abort(T) - T is rollbacked = changes undone

Transaction Properties - ACID

- properties a DBMS must ensure for a transaction
- ATOMICITY
  - a transaction is atomic → either all operations are executed or none (ALL-OR-NOTHING)
- CONSISTENCY

- a transactions must preserve the consistency of the database after execution
  → a transaction is a correct program (idk)

- ISOLATION
  - a transaction is protected from the effects of concurrent other transactions
- DURABILITY
  - the effects of a successfully completed transaction must persist even if a system crash occurs before all the changes have been written to disk (MPP pica netu da tot se fac modificarile type shi)

ATOMICITY

- a transaction can commit / abort:
  - commit - after finalizing all operations
  - abort - after executing some of it's operations
- Either ALL operations are executed or NONE
- the DBMS logs all transactions' writes, so it can UNDO them if necessary
- Crash Recovery → guarantees atomicity if the system crashes
- EXAMPLE: AccountA transfers 100 lei to AccountB:
  AccountA ← AccountA - 100
  AccountB ← AccountB + 100
  → if the transaction fails in the middle, 100 lei must be restored to AccountA (rollback all operations)

CONSISTENCY

- = in the absence of other transactions, a transaction that executes on a CONSISTENT database state must leave the database in a consistent state
- = transactions do not violate the integrity constraints (ICs) specified on the database
  - EXAMPLEs of ICS: Restriction declared via CREATE TABLE, ADD CONSTRAINT ... etc.

- → transactions are correct programs
- EXAMPLE:
  AccountA ← AccountA - 100
  AccountB ← AccountB + 100
  → consistency constraint: the transaction preserves the total amount of money in the 2 accounts
- self-explanatory ???

ISOLATION

- transactions are PROTECTED from the effects of concurrent OTHER Transactions
- Transactions are independent from other users' transactions = a transaction is seen as if it were in a single-user mode

DURABILITY

- the system must guarantee the effects of a successful transactions won't be lost, even if failures occur
- → the Write-Ahead Log property = changes are written to the log (on the disk) before being reflected in the database
- → the Log ensures atomicty & durability

Interleaved Exectuions

T1: BEGIN	A ← A-100	B ← B+100	END
T2: BEGIN	A ← 1.05*A	B ← 1.05*B	END

- T1 and T2 are submitted together to the DBMS
- serial execution: T1 can execute before or after T2 on a database DB, resulting in one of the following database states:
  - State((T1T2), DB) or
  - State((T2T1), DB)
- interleaved execution: the net effect of executing T1 and T2 *must* be identical to State((T1 T2), DB) or State((T2 T1), DB)

THE RESULTS WILL BE DIFFERENT DEPENDING ON THE ORDER OF THE OPERATIONS:

T1	T2
A ← A - 100	
	A ← 1.05 * A
B ← B + 100	
	B ← 1.05 * B

T1	T2
A ← A - 100	
	A ← 1.05 * A
	B ← 1.05 * B
B ← B + 100	

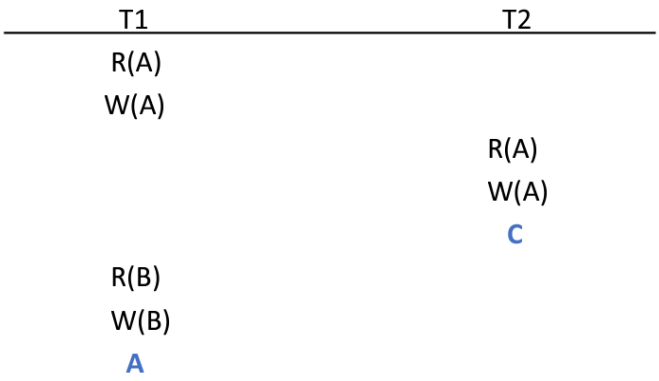
ANOMALIES

- 2 transactions read data ⇒ no conflict
- 2 transactions read and / or write completely separated data ⇒ no conflict

- 2 transactions operate on SAME DATA and one/both WRITE(S) ⇒ order of execution MATTERS
  - WriteRead (WR) conflict = T2 reads data written by T1
  - ReadWrite (RW) conflict = T2 is writting data previously read by T1
  - WriteWrite (WW) conflict = T2 is writting data previously written by T1

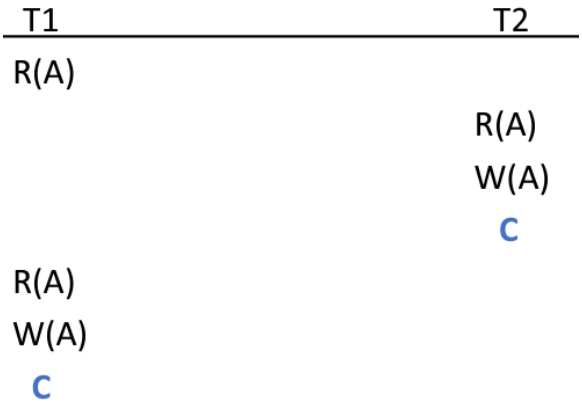
Reading uncommited data (WR conflict)

- = a DIRTY READ



- T2 reads data object A, but transaction T1 is not COMMITED and therefore it can be rollbacked later → Anomaly

Unrepeatable reads (RW conflict)



Overwriting uncommited data (WW conflict)

Scheduling Transactions

- schedule = a list of operations (read / write / commit / abort) of a set of transactions with the property that the order of the operations in each individual transactions is preserved

Non-Serial Schedule

- a schedule in which the actions of different transactions are interleaved

T1	T2
read(V)	
read(sum)	
	read(V)
	V := V + 50
	write(V)
	commit
read(V)	
sum := sum + V	
write(sum)	
commit	

Schedule
read1(V)
read1(sum)
read2(V)
write2(V)
commit2
read1(V)
write1(sum)
commit1

Serial Schedule

- a schedule in which the actions of different transactions are not interleaved

T1	T2
	read(V)
	V := V + 50
	write(V)
	commit
read(V)	
read(sum)	
read(V)	
sum := sum + V	
write(sum)	
commit	

Serializability

Serializability

- *C* – set of transactions
- *Sch(C)* – the set of schedules for *C*
- *serializable schedule*
  - a schedule $S \in Sch(C)$ is serializable if the effect of *S* on a consistent database instance is identical to the effect of some serial schedule $S_0 \in Sch(C)$

Lecture 2 - Transactions. Concurrency Control

Conflict Serializability

- *C* – set of transactions
- *Sch(C)* – the set of schedules for *C*
- *Op(C)* – set of operations of the transactions in *C*
- consider schedule $S \in Sch(C)$
- the *conflict relation* of *S* is defined as:
  - *conflict(S)* = $\{(op_1, op_2) \mid op_1, op_2 \in Op(C)$, op_1 occurs before op_2 in *S*, op_1 and op_2 are in conflict$\}$

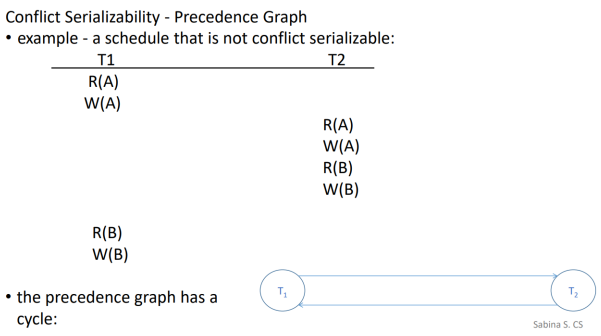
Conflict Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- two schedules S_1 and $S_2 \in Sch(C)$ are *conflict equivalent*, written $S_1 \equiv_c S_2$, if $conflict(S_1) = conflict(S_2)$, i.e.:
  - S_1 and S_2 contain the same operations of the same transactions and
  - every pair of conflicting operations is ordered in the same manner in S_1 and S_2

- schedule S is *conflict serializable* if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_c S_0$, i.e., S is conflict equivalent to some serial schedule

- Precedence Graph (Serializability Graph) of S has:
  - one NODE for every commited Transaction in S
  - an arc from Ti to Tj if an ACTION IN Ti precedes an ACTION IN Tj and conflicts it
- THEOREM:
  - A schedule S in Sch(C) (all schedules for C- set of transactions) is Conflict Serializable
    IF → it's Precedence Graph is
    Acyclic ⇔ it can be written as a serial schedule S0 such that S0 and S are conflict equivalent (have the same conflicts)

- In the following case,
  (Read(T1,A), Write(T2,A)) is a conflict where T1 operation Read(A) precedes T2 operation Write(A) ⇒
  T1→T2
  AND
  Read(T2,B) precedes Write(T1,B) → conflict ⇒
  T2 → T1
  so the followings' Sequence doesnt have an Acyclic



Precedence Graph, therefor it is
not conflict serializable

ALGORITHM FOR TESTING Conflict Serializability:

- create a node for each commited transaction
- create an arc (Ti,Tj) for each transaction where:
  - Tj executes a Read(A) after Ti executes a Write(A)
  - Tj executes a Write(A) after Ti executes a Read/Write(A)
- if the obtained graph is acyclic ⇒ Conflict Serializable

* A Schedule is Serializable if it can be written as a Serial Schedule and still have the same effect on a database

View Serializability

View Serializability
- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- let T_i, $T_j \in C$, S_1, $S_2 \in Sch(C)$; S_1 and S_2 are *view equivalent*, written $S_1 \equiv_v S_2$, if the following conditions are met:
  - if T_i reads the initial value of V in S_1, then T_i also reads the initial value of V in S_2;
  - if T_i reads the value of V written by T_j in S_1, then T_i also reads the value of V written by T_j in S_2;
  - if T_i writes the final value of V in S_1, then T_i also writes the final value of V in S_2.
- i.e.:

  each transaction performs the same computation in S_1 and S_2

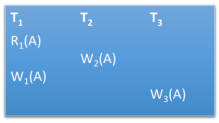
  and

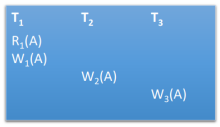
  S_1 and S_2 produce the same final database state.

Sabina S. CS

- basically S1 and S2 have the same effect on the database
- For example, operations on A are done in the same order, any A data object in the database, for any transaction Ti in C, where S in Sch(C)

• a schedule $S \in Sch(C)$ is <u>*view serializable*</u> if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_v S_0$, i.e., S is view equivalent to some serial schedule





Recoverable Schedules

= a schedule in which a transaction T commits only after all transactions whose changes T read commit

- let's say we have 2 transactions, T1 and T2
- and T1 modifies X and then T2 operates on it then commits
- T2 operates on a value of X that shouldn't be there and T1 aborts at the end
- but since T2 commited before the abort, that means that Schedule S is UNRECOVERABLE

You need to AVOID Cascading Aborts (Example: t1 aborts then t2 aborts)

Lock-Based Concurrency Control

- it is a tehnique used to guarantee serializability, recoverable schedules

Lock

- a tool used by the transaction manager to control concurrent access to data
- prevents a transaction from accessing a data object while another transaction is accessing the object
- SLOCK = shared or read lock =
  - if a transaction has an SLock on an object, then it can read it, but it cannot modify it

    - if T1 holds SLock on object A, other transactions can only acquire SLocks on the object, not XLocks
- XLock = exclusive or write lock =
    - if a transaction holds an XLock then it can both read and write the object
    - if a transaction holds an XLock on an object, no other transaction can get any type of lock on that object
- LOCK UPGRADE = an SLock can be upgraded to an XLock

Lock Table

- structure used by the lock manager to keep track of granted locks / lock requests
- entry in Lock Table:
    - number of transactions holding a lock on the object
    - lock type: SLock / XLock
    - pointer to a queue of lock requests

Transaction protocol

- a set of rules enforced by the transaction manager and obeyed by all transactions
    - Example – simple protocol: before a transaction can read / write an object, it must acquire an appropriate lock on the object

Transaction Table

- structure maintained by DBMS
- one entry / transaction - keeps a pointer to a list of locks held by the transaction

Lecture 3 - Transactions. Concurrency Control

Locking Protocols

Strict Two-Phase Locking (Strict 2PL)

- before a transaction can read/write it must acquire an SLock/XLock on the object
- all locks held by the transactions are released when it completes execution
- the Strict 2PL protocol allows only serializable schedules (only schedules with acyclic precedence graphs are allowed by this protocol)
- so basically only

Two-Phase Locking (2PL)

- before a transaction can read/write it must first acquire a lock on the object S/X
- once a transaction RELEASES a lock, it CANNOT requests others ⇒
  ⇒ 2 phases:
  - Phase 1 = growing phase: transaction acquires locks
  - Phase 2= shrinking phase: transaction releases locks
- if all transactions in a Schedule S obey 2PL ⇒ S is Serializable

Strict Schedules

- if transaction Ti has written A, then Tj can read/write object A only after Ti's completion (commit/abort)

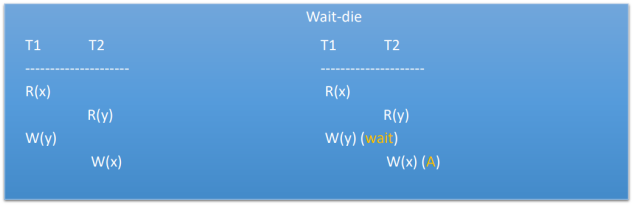
Deadlocks

- Lock-Based concurrency can lead to DEADLOCKS
- Deadlock = cycle of transactions waiting for another to release a locked resource; normal execution can no longer continue < = > deadlocked transactions cannot proceed until the deadlock is resolved

Prevention (Wait-die, Wound-wait)

- assign transactions a timestamp = when they begin

- they get higher priority the older they are
- Wait-Die:
  - T1 wants to access object locked by T2 with a conflicting lock
  - if T1 has priority > T2 priority ⇒ wait, otherwise aborted (die)

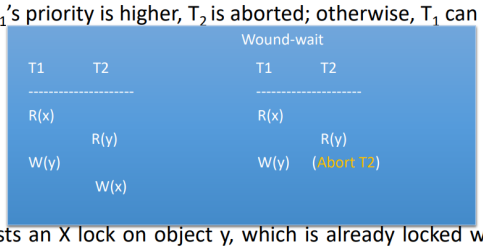


- T1 requests an X lock on object y, which is already locked with a conflicting lock by T2
- since T1 has a higher priority, it is allowed to wait
- T2 asks for an X lock on object x, already locked with a conflicting lock by T1
- since T2 has a lower priority, it is aborted
- T1 now obtains the requested lock on object y and proceeds with the write operation

Sabina S. CS

- Wound-Wait:
  - T1 wants access to object locked by T2 with a conflicting lock
  - if T1 priority > T2 priority ⇒ T2 aborted; else T1 waits

- assume T_1 wants to access an object locked by T_2 (with a conflicting lock)
  - Wound-wait
    - if T_1's priority is higher, T_2 is aborted; otherwise, T_1 can wait



- T1 requests an X lock on object y, which is already locked with a conflicting lock by T2
- since T1 has a higher priority, T2 is aborted
- T1 obtains the requested lock on object y and continues execution

Sabina S. CS

- under either of these policies, deadlocks cannot occur
- if an aborted transaction is restarted, it is given it's original timestamp

Detection

A. Wait-For Graph

- lock manager → to detect deadlock cycles
  - nodes
  - arc from Ti to Tj if Ti is waiting for Tj to release it's lock
- cycle in the graph ⇒ deadlock (Ti waits for Tj waits for Tk waits for Ti → cycle < = > deadlock)

    - if a cycle exists, one transaction from the cycle is aborted
- Choosing the Deadlock Victim criteria:
    - number of objects modified by the transaction
    - number of objects that are to be modified by the transaction
    - numbers of locks held
    - if a transaction is repeatedly chosen as a victim, it should be allowed to proceed at some point

B. Timeout Mechanism

- if a tranasction T waits too long for a lock, it's assumed that there is a deadlock so T is terminated

Phantom Problem

- in the presence of insert operations, i.e., if new objects can be added to the database, conflict serializability does not guarantee serializability
- Example: if T1 gets the lock on A and B, then releases, then T2 gets on B and C, and inserts smth in C and removes from D, then releases and T1 gets the lock on C and reads, that means that altough it is conflict serializable, it is not serializable:
    - there are no conflicts, because T2 commits before T1 accesses, BUT S1 = (T1, T2) will have T1 reading B and C without the insert/remove while S2 = (T2, T1) will have T1 reading C with the delte and B with the insert so both aren't equivalent to S, which has T1 reading B without insert, but C with the delete

Isolation Levels

- determines the degree to which a transaction is exposed to the operations of other concurrently running transactions
- greater concurrency → concurrency anomalies
- LEVEL
    - Read Uncommited

    - Read Commited
    - Repeatable Read
    - Serializable
- isolation levels can be set with the following command:
  • SET TRANSACTION ISOLATION LEVEL isolevel
  • e.g., SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- dirty writes are not allowed under any isolation level (dirty write - a transaction
  T1 modifies an object previously written by an ongoing transaction
  T2)

READ UNCOMMITED

- lowest degree of isolation
- a transaction T can read data modified by an Ongoing transaction
- no SLock when reading the data
- Problems that can occur:
    - dirty reads
    - unrepeatable reads
    - phantoms

READ COMMITED

- a transaction T can only read commited data
- HOWEVER, an objeact read by T can be modified while T is still in progress
- a transaction must acquire an exclusive lock (XLock) prior to writing an object
  → exclusive locks are released at the end of the transaction
- a transaction must acquire an SLock prior to reading an object
  → SLocks are released
  immediately

- PROBLEMS
  - NO dirty reads
  - unrepeatable reads
  - phantoms

Repeatable Reads

- a transaction T can only read commited data
- no object read by T can be modified while T is in progress (if T reads O twice, no transaction can modify O between the 2 reads)
- a transaction must acquire an SLock prior to reading / XLock prior to writing
  → exclusive locks are released at the end of the transaction
  → SLocks are released at
  the end of the transaction
- PROBLEMS:
  - NO dirty reads / NO Unrepeatable Reads
  - Phantoms

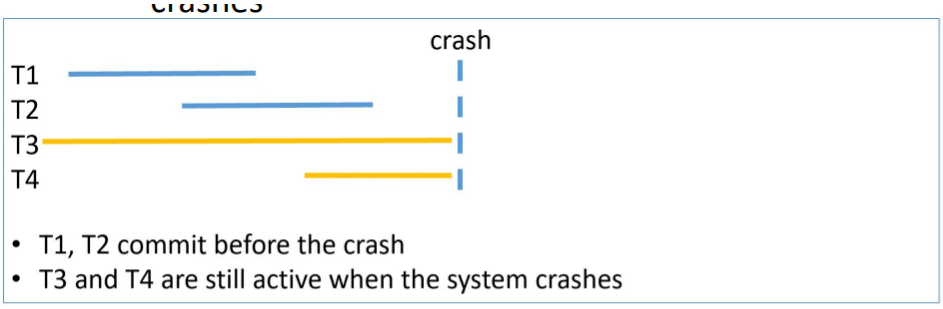
Serializable

- a transaction T can only read commited data
- no object read by T can be modified while T is in progress
- if T reads a set of objects based on a search predicate, this set cannot be changed by other transactions while T in progress
  Ex.: SELECT * FROM Students WHERE GPA >= 8 must return the same thing; no other transaction can add/delete a Student with GPA >= 8
- a transaction must acquire locks before reading/writing
  → which are released at the end of the transaction
- NO PROBLEMS CAN OCCUR = HIGHEST DEGREE OF ISOLATION

Lecture 4 - Crash Recovery

Recovery Manager

- this ensures 2 important properties of transactions →
  - Atomicity = effects of uncompleted transactions are undone
  - Durability = effects of commited transactions survive system crashes



- the system comes back up:
  - the effects of T1 & T2 must persist
  - T3 & T4 are undone (their effects are not persisted in the DB) Sabina S. CS

Transaction Failure - Causes

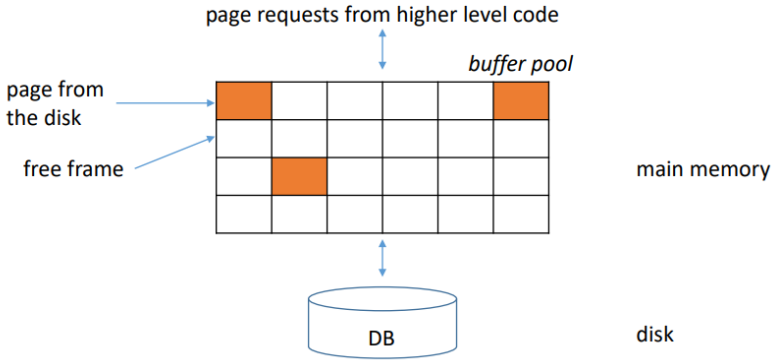
- SYSTEM FAILURE → hardware failures, bugs in the OS / database system etc.
  - all running transactions terminate
  - contents of internal memory - affected (lost)
  - contents of external memory - not affected
- APPLICATION ERROR → bugs, division by “0”, infinite loops etc.
  - a transaction fails → executed again only after error is corrected
- Action by the Transaction Manager (TM)
  - Example: deadlock resolution scheme → a transaction is chosen as a deadlock victim and terminated
    - the transaction might be successfull after executing again
- SELF-ABORT →
  - based on some actions, a transaction can choose to terminate itself
  - there are SPECIAL STATEMENTS for this: Abort, Rollback

- Special case of action by the TM

Normal Execution

- Read(O) → bring O from disk into frame in the Buffer Pool (BP) → copy O's value into a program variable
- Write(O) → modify an in-memory copy of O (in the BP) → write the in-memory copy to the disk

Buffer Manager*



Writing Objects

- Options
  - steal / no-steal
  - force / no-force
- Let's say transaction T changes object O (in a frame F in the BP)
  - STEAL approach:
    → T's changed can be written to disk before commit
    T2 needs a page in the BP; the BM chooses F as a replacement frame (while T in progress) ⇒ T2 steals from T
    - Steal approach is used by most systems
  - NO-STEAL approach:
    → T's changes cant be written to disk before commit

        - Advantage → changes of aborted transactions do not have to be undone
        - Drawback → you don't know if all pages modified by active transactions can fit in the Buffer Pool
    - FORCE approach:
      → T's changes are
      immediately forced to disk when it comits
        - Advantage → actions of commited transactions dont have to be re-done
          → by contrast, in no-force approach, if T commits at time t0 and at time t1, before changes could have been saved to the disk, the system crashes, then actions of T have to be re-done
        - Drawback → can result in excessive I/O
    - NO-FORCE approach:
      → T's changes are not forced to disk when it comits
        - No-Force approach is used by most systems

Storage Media

Volatile Storage

- information doesn't survive system crashes (main memory)

Non-Volatile Storage

- information survives system crashes (magnetik disks, flash storage)

Stable Storage

- information is NEVER lost
- techniques that approximate Stable Storage:
  → store on multiple disks and in several locations

Aries

LSN	Log
1	update: T1 writes P1
2	update: T2 writes P2
3	T2 commit
4	T2 end
5	update: T3 writes P3
6	update: T3 writes P2
crash, restart	

analysis
- active transactions at crash time: T1, T3 (to be undone)
- committed transactions: T2 (its effects must persist)
- potentially dirty pages: P1, P2, P3

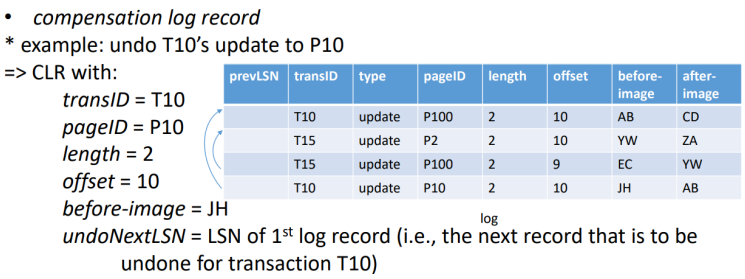
redo
- reapply all changes in order (1, 2, …)

undo
- undo changes of T1 and T3 in reverse order (6, 5, 1)

The Log

- this is the history of actions executed by the DBMS → stored in Stable Storage
- new records (actions executed by the DBMS) are added to the end of the log
- Log Tail → most recent fragment of the log, kept in main memory and forced periodically into Stable Storage
- Log Sequence Number (LSN) → unique id for every log
- pageLSN → every page P cotains the pageLSN = the LSN of the most recent record in THE LOG that describes a change to page P
- Log Record → fields:
  prevLSN - link to a transactions log records
  transID - id of that transaction
  type - type of the log record
- Log records are written for the following actions:
  - update page
  - commit
  - abort
  - end
  - undo an update

- Compensation Log Record = describes the action taken to undo the changed described by a log record



Transaction Table & Dirty Page Table

- Transaction table - contains information for recovery process
  - 1 entry for each ACTIVE TRANSACTION
    - transID, status → in progress, commited, aborted, lastLSN → most recent log record for that transID
- Dirty Page Table
  - 1 entry for each dirty page in the Buffer Pool
    - pageID & recLSN that points to the 1st log record that dirtied the Page

Checkpoiting

- used to reduce amount of work for the system to backup a crash

1. Write a beginCheckpoint record → new checkpoint
2. EndCheckpoint → includes TransactionTable & DirtyPageTable
3. Write a Master record to know a place on the Stable Storage that includes the LSNbck - lsn of the beginCheckpoint

- crash → restart → system looks for most recent checkpoint

Recovery

1. Analysis
   a. look for the most recent checkpoint → find active transactions and possible dirty pages
2. Redo

a. repeat history → all updates are reapplied / all changes to dirty pages are reapplied

3. Undo
   a. the effects of the transactions active at the time of the crash are undone in the opposite order

Lecture 5 - Security

- Database protection = security & integrity
- Security → Protecting the data against Unauthorized users
- Integrity → the operations that users are trying to execute are correct, the consistency of the database must be maintained at all times
- THE SYSTEM enforces CONSTRAINTS that the users must obey
- Constraints:
  - specified in a declerative language
  - saved in the system catalogue
  - Example: primary key constraint
  - The system monitors users' actions to enforce the specified constraints on their actions (ex.:inserts, updates cant violate Primary Key constraint)

Authorization Subsystem

- it checks any access request = requested object + requested operation + requesting user
  to be safe
- Discretionary control → users have different access right
  → operations are explicitely specified: User 1 can select from tables ...
  User 3 can insert/delete/update/select from table Students
  etc.
  - Example:
    AUTHORITY CA1 → name

GRANT RETRIEVE / INSERT / DELETE / UPDATE → privilege
ON Customers → table(s)
TO Alice, Bob → user(s)

- User U creates object O ⇒ User U has all privileges on object O (and he can grant those to other users:
  GRANT privilege_list
  ON object
  TO user_list
  [WITH GRANT OPTION]

  or revoke them:

  REVOKE SELECT, DELETE, UPDATE {FirstName, LastName}
  ON Customers
  FROM Jane RESTRICT

- Mandatory control →
  → Object = classification level (top secret, secret etc.)
  → User = clearance level (same as object)
  User U with clearance lvl x can access object O with classification lvl y
  ⇔ x>=y | U can update O ⇔ x==y

Audit Trail

- lists of operations and the users that performed them

SQL Injection

- Application → execution of an SQL statement (fixed, generated using input data from the user)
  - statement can be changed while being generated due to data supplied by the user; when the user wants additional access rights;
  - the user enters code into input variables; the code is concatenated with SQL statement and executed
- Basically, users can modify / access things that they wouldnt normally be able to, by giving certain specific inputs

Prevention

- Data validation:
  - use regular expressions to validate data
  - allow the users only a specific set of characters
- Modify problematic characters
  - double single quotes and double quotes
  - Example:
    SELECT ... WHERE user =
    "a" AND password =
    " " " OR 1 = 1 -- "
    , which
    retrieves rows with user a and password " OR 1 = 1 --
    instead of the whole users
- Use parameterized statements
  - SELECT ... WHERE user = ? AND password = ?, where “?” is a parameter

Data Encryption

- protecting data when the normal security mechanism in the DBMS are insufficient (Ex.: when someone gets physical access to the data)
- some DBMSs store data in an encrypted form; if the files containing the database can be copied, using the data in these files is impossible (or very difficult, as decryption cost is HUGE)

Codes and ciphers

CODE

- replace one word / phrase with another word/number/symbol
  Example: replace “Nacho are mere” with “Rares”

CIPHER

- replace each letter with another number/symbol
  Example: replace each letter with the preceding one (A=Z, b=a, c=b, D=C

etc.)

Stenography

- hide the EXISTANCE of the message (ex.: invisible ink)

Cryptography

- hide the MEANING of the message = encryption

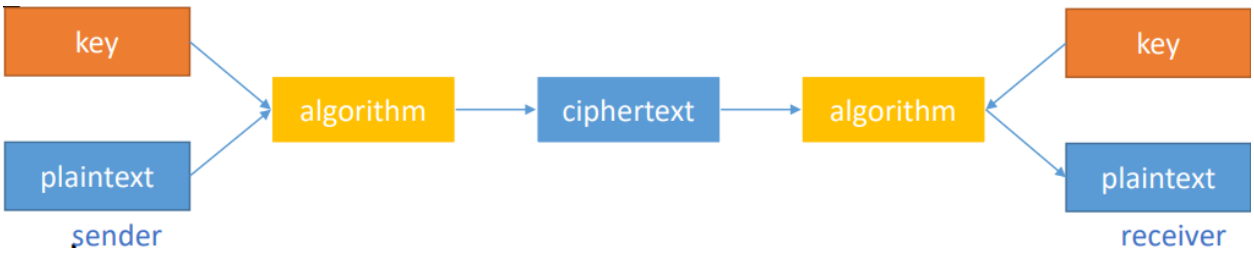
Transposition

- rearrange letters (create anagrams) - every letter is the same but on a different position

Substitution

- pair each letter in the alphabet with another and make the substitution
- every letter changes it's identity but its on the same position
- PlainText = message before encryption | PlainAlphabet
- CipherText = message after encryption | CipherAlphabet
- Example: shift by n positions, change capital letters to small ones and small to capital

Algorithms & Keys



Algorithms

- not secret
- it's a general encryption method (Ex.: replace each letter in the Plain alphabet with it's correspondent in the cipher alphabet)

Key

- MUST be kept SECRET!
- it's a detail for a particular encryption, like the chosen cipher alphabet

A problem arose: how to exchange messages? → key distribution problem

- if 2 parties want to exchange a private message, they need to also exchange the key, which has to be kept a secret for the message to work
- SOLUTION: one-way functions in modular arithmetic

one-way function: $Y^x \pmod P$ e.g., $7^x \pmod{11}$ 7, 11 – numbers chosen by Alice & Bob; not secret	Alice	Bob
1	Alice chooses a number A = 3 A - secret	Bob chooses a number B = 6 B - secret
2	Alice computes: $\alpha = 7^A \pmod{11} = 2$	Bob computes: $\beta = 7^B \pmod{11} = 4$
3	Alice sends α to Bob	Bob sends β to Alice
exchange	An eavesdropper can intercept α and β. This poses no problems, since these numbers are not the key!	
4	Alice computes: $\beta^A \pmod{11} = 4^3 \pmod{11} = 9$	Bob computes: $\alpha^B \pmod{11} = 2^6 \pmod{11} = 9$
the key	Alice and Bob obtained the same number: 9 - the key.	

- Alice and Bob are able to establish a key without meeting beforehand
- Mark (eavesdropper, trying to intercept exchanged messages) only knows the $7^x \pmod{11}$ function, $\alpha = 2, \beta = 4$
- Mark doesn't know the key and cannot obtain it, since:
  - he doesn't know A and B (these numbers are kept secret);
  - it's extremely difficult for him to obtain A from α (or B from β), since $7^x \pmod{11}$ is a one-way function; this is especially true if the chosen numbers are very large

Lecture 6 - Evaluating Relational Operators. Query Optimization

* queries – composed of relational operators:

- selection (σ)
  - selects a subset of records from a relation
- projection (π)
  - eliminates certain columns from a relation
- join ($\otimes$)
  - combines data from two relations
- cross-product (R1 × R2)
  - returns every record in R1 concatenated with every record in R2
- set-difference (R1 – R2)
  - returns records that belong to R1 and don't belong to R2
- union (R1 ∪ R2)
  - returns all records in relations R1 and R2
- intersection (R1 ∩ R2)
  - returns records that belong to both R1 and R2

- Optimizer : input SQL Query Q and it outputs an efficient execution plan for Q
- Algorithms for operators → based on 3 techniques

1. Iteration
   - examine iteratively all tuples in inpute relations / all data entries in indexes (data entry < data record)
2. Indexing
   - used when there's a selection condition or a join condition
   - examine only the tuples that satisfy the condition
3. Partitioning
   - partition the tuples, decompose the operation into cheaper operations on partitions
     Example: sorting, hashing

Access Paths

- way of retrieving tuples from a relation
  - file scan
  - index ‘i’ + a matching condition C

* access paths - example:
- relation *Students[SID, Name, City]*
- *I* - tree index on *Students* with search key *<Name>*
- query Q:
  ```
  SELECT *
  FROM Students
  WHERE Name = 'Ionescu'
  ```
- condition *C*: *Name = 'Ionescu'*
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- the following condition also matches the index: *Name > 'Ionescu'*

- condition C: Attribute operation Value , where operation in {<,<=, =, <>, >=, >}
- C matches index i if: search key of index is Attribute and:
  - i is a tree index
  - i is a hash index and operation is “=”
- if the hash index has a search key with multiple attributes, then the condition C must have an equal (=) operation for each of the search key elements
- if the tree index has a search key <a, b, c> (multiple attributes), then C must have one term for each prefix of the search key (a = x and b = 4 works | a =x works | b=y doesnt work, as it doesn’t have the prefix a)

Selectivity of an access path

- selectivity = number of pages retrieved, both data pages and index pages are counted
- most selective access path = retrieves the minimum number (if file scan -10 pages and index - 3 paged, then it will retrieve via index)

General selection condition

- in general, a selection conditions has one or more terms of the form:
  - attribute operation constant
  - attribute operation attribute2
  - combined with V or ∧

```
SELECT *
FROM Exams
WHERE SID = 7 AND EDate = '04-01-2021'
```

$$\sigma_{SID=7 \wedge EDate='04-01-2021'}(Exams)$$

- select = sigma, V = or, ∧= and

- to express a cond C in CNF (conjuctive normal form)
  - collection of conjucts connected with ∧ (and)
  - a conjuct has one/more terms connected with V (or)

condition *(EDate < '4-1-2021' ∧ Grade = 10) ∨ CID = 5 ∨ SID = 3*
is rewritten in CNF:
(EDate < '4-1-2021' ∨ CID = 5 ∨ SID = 3) ∧ (Grade = 10 ∨ CID = 5 ∨ SID = 3)

- When there are more conditions than the index has in a search key, the tuples are retrieved to match the index condition and then the other conditions are applied
  Example: index I with search key <a, b, c> and cond a = 20 AND b = 10 AND c = 5 AND d = 11
  use index I to retrieve tuples that satisfy a=20, b=10, c=5 and then apply d=11

Example

Students (SID: integer, SName: string, Age: integer)
Courses (CID: integer, CName: string, Description: string)
Exams (SID: integer, CID: integer, EDate: date, Grade: integer, FacultyMember: string)

Students
- every record has 50 bytes
- there are 80 records / page
- 500 pages of Students tuples

Courses
- every record has 50 bytes
- there are 80 records / page
- 100 pages of Courses tuples

- Exams
  - every record has 40 bytes
  - there are 100 records / page
  - 1000 pages of Exams tuples

- all are 400bytes / page
- size of E x S is large, so computing it by selection is inneficient

* joins

```
SELECT *
FROM Exams E, Students S
WHERE E.SID = S.SID
```

- algebra: $E \otimes_{E.SID=S.SID} S$

joins – implementation techniques
- iteration
  - Simple/Page-Oriented Nested Loops Join
  - Block Nested Loops Join
- indexing
  - Index Nested Loops Join
- partitioning
  - Sort-Merge Join
  - Hash Join

Simple Nested Loops Join

- for each record in E, take each record in S and check E.SID = S.SID
  - costs are 1000(pages E) * 100(E records /page) * 500(pages in S) + 1000(pages in E)
  - because E is scanned once and then S is scanned 100 ∗ 1000 times (each page)

Page-Oriented Nested Loops

- refined Simple Nested Loop
- cost: 1000 + 1000*500

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

Block Nested Loops Join

- you store the smaller relation R1 in main memory
- then keep 2 extra buffer pages B1 and B2
- use B1 to read the larger relation R2 one page at a time, and use B2 as output buffer
- for each tuple in R2, read in B1, search R1 for matching tuples
- this only costs: 500 + 1000 in our case, as R1 and R2 are only scanned once
- REFINEMENT: build an in-memory hash-map for R1

- if not enough main memory to store the smaller R1:
  - use one buffer page to scan the inner table S and one page for the result
  - use all remaining pages to read a block from the outer table (block = set of pages from E that can fit in the main memory currently)
  - S is scanned for each block in outer relation E, but E is only scanned once
  - cost: 1000(pages in E) + 500(pages in S) * numberOfBlocksInE
  - number of blocks in E = number of pages in E / size of block in main memory

Indexing - Index Nested Loops Join

```
foreach tuple e in E do
    foreach tuple s in S where e_i == s_j
        add <e, s> to the result
```

- if there is an index on the join column of S, S can be considered as inner table and the index can be used
- cost
  - M + ((M*p_E) * cost of finding corresponding records in S)

* E - M pages, p_E records / page * * 1000 pages * * 100 records / page*
* S - N pages, p_S records / page * * 500 pages * * 80 records / page *

Sorting

- Types
  - Explicitly required: ORDER BY
  - To Eliminate Duplicates: SELECT DISTINC
  - Used by certain Operators: JOIN | UNION | INTERSECTION | SET-DIFFERENCE | GROUPING
- Internal Sorting = Data to be stored fits in main memory
- External Sorting = Data to be stored doesnt fit in main memory
  - breaks data collection into subcollections, sorts the subcollections = run, writes run to disk then merges runs

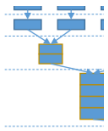
Sorting - Simple Two-Way Merge Sort

- uses 3 buffer pages, 2 inputs and one output

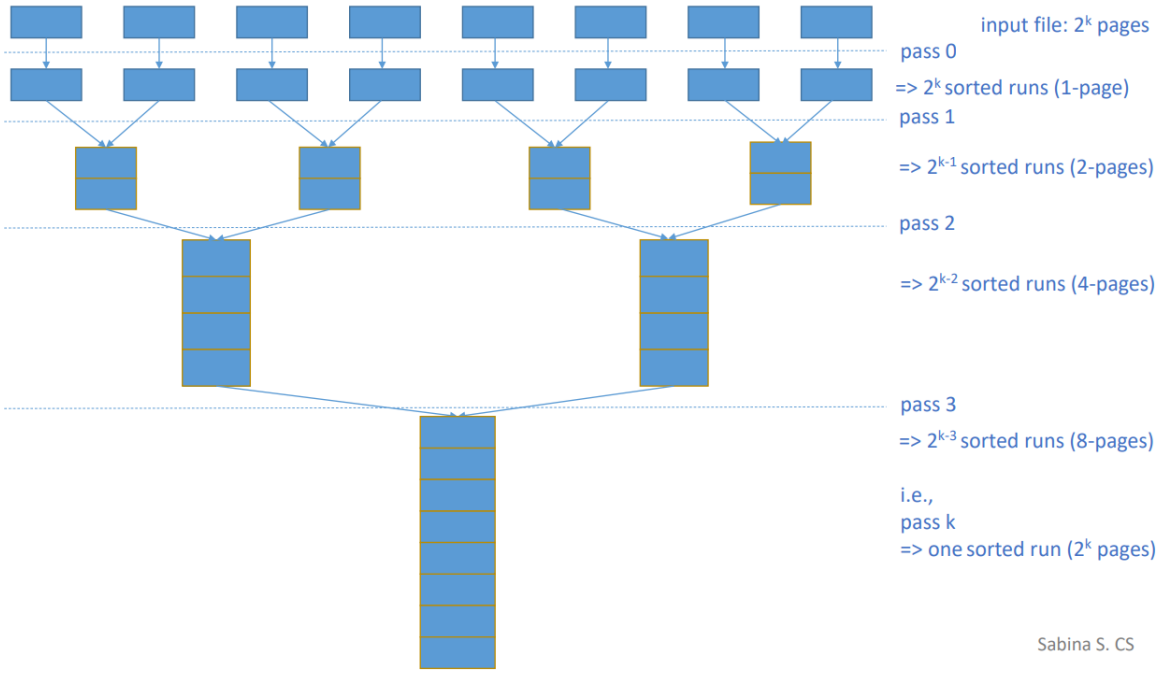
- STEP 0 - for each page P in data collection: sort it and save it to disk
  ⇒
  ⇒ 1-page “runs” saved on disk
- STEPS 1,2,3... - read and merge pairs of runs from the previous path ⇒
  ⇒ 2-page runs, then 4 page runs and so on

- number of passes:
  - $\lceil log_2 N \rceil + 1$

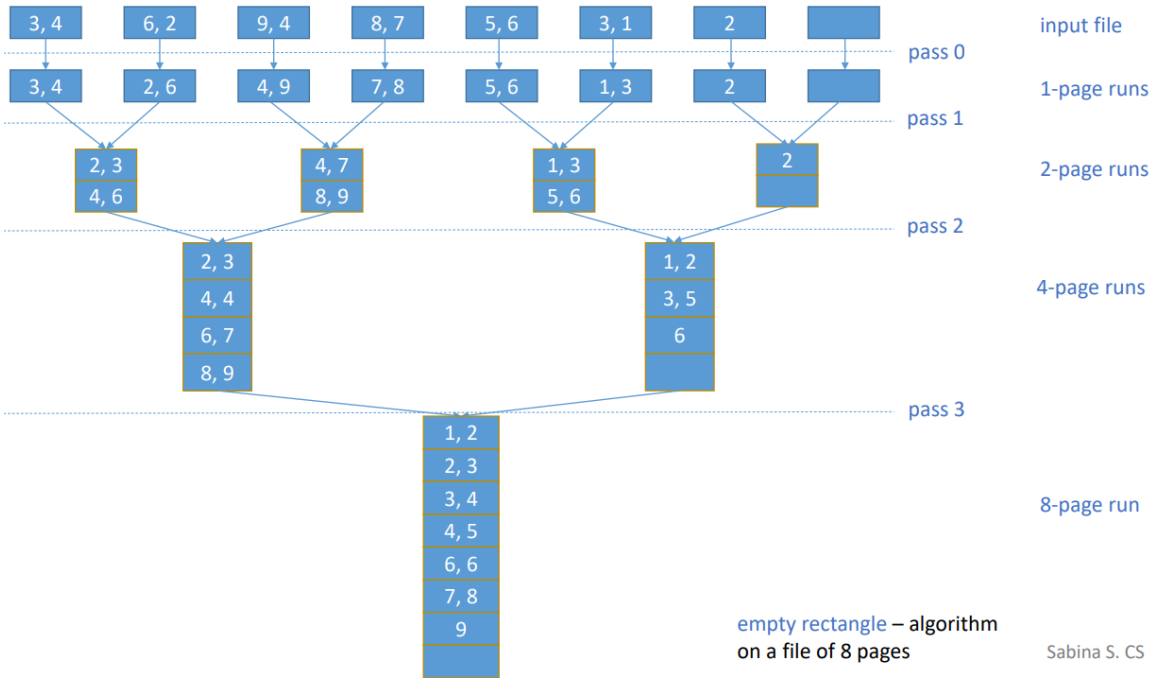
  where *N* is the number of pages in the file to be sorted



- total cost:
  - 2 * number of pages * number of passes
  - $2 * N * (\lceil log_2 N \rceil + 1)$ I/Os



Sabina S. CS



Lecture 7 - Evaluating Relational Operators. Query Optimization

Sorting - External Merge Sort

- In the Simple Two-Way Merge Sort, buffer pages arent used effectively (if there are 200 available buffer pages, still only 2 are used) → generalize the 2WayMSort algo to minimize no passes and effectively use main-memory
- N = number of pages to be sorted / input | B - buffer pages available
- STEP 0: Use all B buffer pages to read & sort (in-memory) a page from the input
  ⇒ N / B runs
- STEPS 1,2,3...: Use B-1 pages for input and 1 page for output

- cost
  - N – number of pages in the input file, B – number of available pages in the buffer
  - in each pass: read / process / write each page
  - number of passes: $\lceil log_{B-1}\lceil N/B \rceil \rceil + 1$
  - total cost: $2 * N * \left(\left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1 \right)$ I/Os

Sorting - Sort Merge Join

- this is an equality join E $\otimes_{i=j}$ S (E.ith column = S.jth column)
- sort E and S on the join column : by using smth like External Merge Sort
- merge E and S: look for tuples e in E, s in S such that ei = sj

- cost of merging:
  every partition in S is scanned as many times as there are matching tuples in E
- ⇒ Worst Case Scenario when all records in E/S are = on join column
  → M * N where:
  M - no pages E, N- no pages S
- ⇒ Best Case: M + N

cost:
- sorting E
  - cost: O(MlogM)
- sorting S
  - cost: O(NlogN)

Hash Join

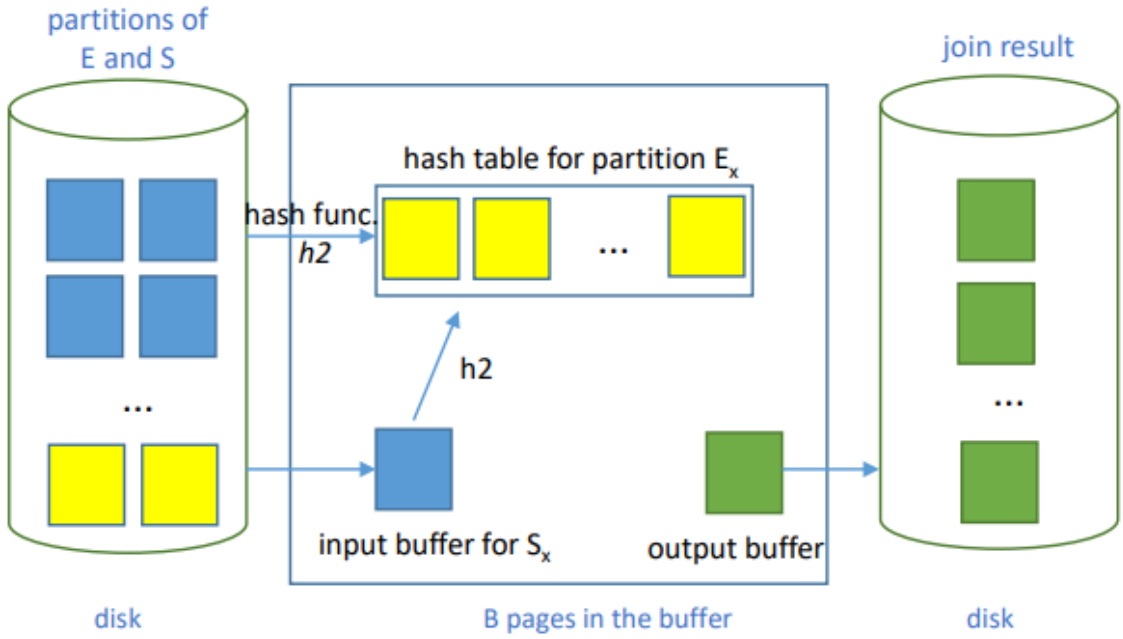
- equality join, one join column: E $\otimes_{i=j}$ S
- HAS 2 PHASES: partitioning (building phase) & probing (matching phase)

Partitioning Phase

- u use one buffer page for INPUT, and the rest for output
- u hash E and S on the join column with the same hash function h, and when one output buffer page fills up, flush the page to the disk

- when reading a partition Ek, you check only the tuples in Sk (meaning that the tuples in Ek have the same hash on ith column as the ones in Sk on the jth column)

Probing Phase



cost:

- partitioning: both E and S are read and written once => cost: 2*(M+N) I/Os
- probing: scan each partition once => cost: M+N I/Os

=> total cost: 3*(M+N) I/Os

- assumption: each partition fits into memory during probing

General Join Conditions

- Equality over several attributes:
  E.SID = S.SID AND E.attrE = S.attrS
  - index nested loops join
    - Exams – inner relation:
    - build index on Exams with search key <SID, attrE>
    - can also use index on SID or index on attrE
    - Students – inner relation (similar)
    - sort-merge join → sort Exams on <SID, attrE>, sort Students on <SID,

attrS>
• hash join → partition Exams on <SID, attrE>, partition Students on <SID, attrS>
• other join algorithms → essentially unaffected

- Inequality comparison

  • E.attrE < S.attrS
  • index nested loops join: B+ tree index required
  • sort-merge join: not applicable
  • hash join: not applicable
  • other join algorithms: essentially unaffected

Selection

- When having a select, for example
  SELECT * FROM E WHERE E.FacultyMember = 'Ionescu'
  where E has a B+ tree index on FacultyMember
  - it's expensive to scan E whole(1000 I/Os) so u use the index instead
- if there's no index on the attribute and not sorted → scan whole data (1000I/Os)
- no index, sorted → binary search → log2(1000) approx 10 I/Os
- B+ tree index on attr → search tree index to find 1st index entry that satisfies the qualifying condition → scan leaf pages to retrieve all OK entries → for each entry, retrieve the tuple

---

- SELECT * FROM E WHERE E.FacultyMember < 'C%', B+-tree index on FacMemb
- 10 000 tuples = 100pages
- clustered B+ tree index ⇒ 100I/Os
- non-clustered → cost up to 1 I/O per tuple (WCS 10 000 I/Os)

- General selections: selections without disjunctions
- C-CNF condition without disjunctions

- evaluating options:
  - use the most selective access path
    - index I → apply conjucts in C that match I
    - apply rest of conjucts
    - Example: c<100 and a=3 and b=5 → use B+ tree index for c<100 → for the resulted tuples check a=3 and b=5 (can use a hash index on a and b and check c<100 for each retrieved tuple)
  - use several indexes → several conjucts match indexes using a2/a3
    - compute set of rids of candidate using index → intersect them → apply other conjucts
    - Example: c<100 and a=3 and b=5 → B+ tree index for c< 100 → hash index for a=3 →
      → compute Result1 intersected Result2 → Rint → check b=5 for each record in Rint

---

- General selections: selections with disjunctions
- C - CNF condition WITH disjunctions : $(a < 100 \vee b = 5) \wedge c = 7$
- Ex. 1:
  - hash index on b and on c ⇒ use index on c, then apply a < 100 V b=5 for resulted tuples
- Ex. 2:
  - a<100 V b=5 , B+ tree index on both a and b → use index on a to retrieve tuples where a<100 & use index on b to retrieve tuples where b=5 → compute Union (V = or)
    (if all matching indexes use a2 / a3 ⇒ take union of rids, retrieve corresponding tuples)

Lecture 8 - Evaluating Relational Operators. Query Optimization

Projection

- $\Pi_{SID,\ CID}$(Exams)

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- to implemenet projection you → eliminate Unwanted Columns & Duplicates
- Projection algorithms = partitioning technique → sorting & hashing

Projection based on sorting

1. Step 1 = scan E → get set E′ that contains only desired columns
    a. cost → scan = num.pages in E I/Os +
    write temporary = T I/Os (T is O(M) and depends on no columns and their sizez
2. Step 2 = sort tuples in E′, sort key = all columns
    a. cost = O(TlogT) (also O(MlogM)
3. Step 3 = scan sorted E′ → compare adjenct tuples & eliminate duplicates

EXAMPLE

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 20
    - E' can be sorted in 2 passes
    - sorting cost: 2 * 2 * 250 = 1000 I/Os
- final scan of E' - cost: 250 I/Os

=> total cost: 1000 + 250 + 1000 + 250 = 2500 I/Os

* example

```
SELECT DISTINCT E.SID, E.CID
FROM Exams E
```

- scan Exams: 1000 I/Os
- size of tuple in E': 10 bytes

=> cost of writing temporary relation E': 250 I/Os

- available buffer pages: 257
  - E' can be sorted in 1 pass
  - sorting cost: 2 * 1 * 250 = 500 I/Os
- final scan of E' - cost: 250 I/Os

=> total cost: 1000 + 250 + 500 + 250 = 2000 I/Os

* E – record size = 40 bytes * * 1000 pages * * 100 records / page*

IMPROVEMENT

- adapt sorting algorithm to do projection with duplicate elimination:
  - modify pass 0 of External Merge Sort (eliminate unwanted columns)
  - modify merging passes to eliminate dups

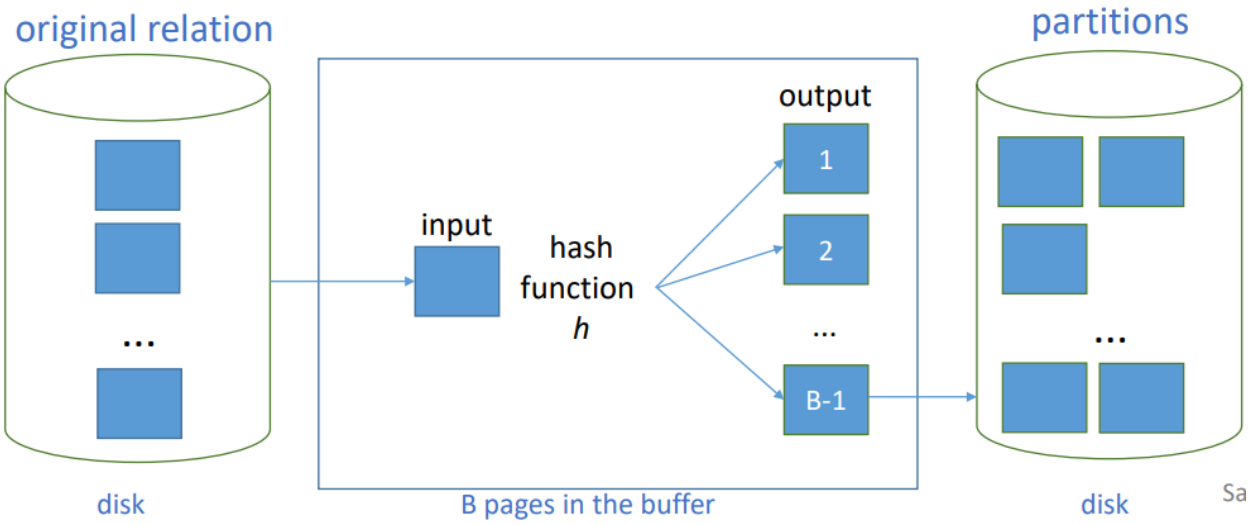
Projection Based on Hashing

- Phases = partitioning & duplicate elimination

PARTITIONING

- 1 input Buffer Page (read one page at a time)→ for each tuples t remove unwated field ⇒ t′ → apply hash function h to t′ → write t′ to the ouput buffer page that is hashed to by h ⇒
- B-1 partitions on B-1 Buffer Pages for output (one output page / partition) = = collection of tuples with
  no unwanted fields and common hash value

  (tuples on different partition are 100% distinct)



DUPLICATE ELIMINATIONS

- for each partition → read partition P → build in-mem hash table with hash fu h2 ≠ h on all fields
- if new tuple hashes to the same value as an existing tuple → check if they are distinct & eleminate duplicates
  - write the duplicate free hash table to the result file (+ clear in-mem hash table)
- In case of partition overflow→ apply hash-based projection technique recursively(sub partitions)
- COST = partitioning → read E= M I/Os | write E= T I/Os | eliminate dups= read partitions: T I/Os

Set Operations

- intersection, cross-products = special cases of join:
  - join condition for intersection → equality on all fields | no join condition for cross-product
- union, set-difference → similar

Union

- R U S (R reunit cu S) → hashing → partition R and S with same function h → for each S-partition: → build in-mem hash table using h2 for S-partition → scan corresponding R-partition and add tuples to hash table, discard duplicates → write out hash table → clear hash table

Aggregate Operations

Without Grouping

- scan relation → maintaing running information about scanned tuples
  - like: COUNT, SUM, AVG, MIN, MAX

With grouping

- sort relation on grouping attributes → scan to compute aggregate operations for each group → to improve, combine sort & aggregation computation | alternative approach based on hashing

Using existing indexes

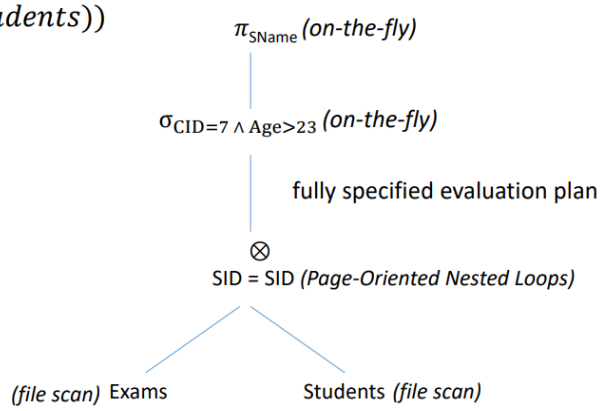
- index w a search key that includes all the required attributes → work w data entries in the index, not data records → attribute list in the group by is a prefix of the index search key(tree index) →
  → get data entries(and records if needed) in the required order (avoid sorting)

Query Optimisation

- Optimizer → has the objective = given a query Q - find a good evaluation plan for Q
  - generates alternatives for Q + estimates their cost + chooses the one with the smallest cost
  - uses info from system catalogs
- Query Q = SELECT S.SName FROM Exams E, Students S
  WHERE
  E.SID = S.SID AND E.CID = 7 AND S.Age > 23

$$\pi_{SName}(\sigma_{CID=7 \wedge Age>23}(Exams \otimes_{SID=SID} Students))$$

- query evaluation plan
  - extended relational algebra tree
  - node – annotations
    - relation
      - access method
    - relational operator
      - implementation method



Sabina S. CS

- Page-Oriented Nested Loops Join→ Exams outer relation
- selection and projection are applied on the fly to each tuple in the JOIN RESULT (not stored)

Pipelined evaluation

-

```
SELECT *
FROM Exams
WHERE EDate > '1-1-2020' AND Grade > 8
```

T1 *T2*

$$\sigma_{Grade>8}(\sigma_{EDate>'1-1-2020'}(Exams))$$

- index *I* matches *T1*
- v1 - *materialization*
  - evaluate *T1*
  - write out result tuples to temporary relation *R*, i.e., tuples are *materialized*
  - apply the 2nd selection to *R*
  - cost: read and write *R*

Sabina S. CS

v2 – *pipelined evaluation*

- apply the 2nd selection to each tuple in the result of the 1st selection as it is produced
- i.e., 2nd selection operator is applied *on-the-fly*
- saves the cost of writing out / reading in the temporary relation *R*

Query Blocks - Units of Optimisation

- parse Q ⇒ collection of Query Blocks → passed to Optimizer → optimize 1block at a time
- Query Block - SQL query
  - without nesting
  - exactly one SELECT clause, one FROM clause
  - at most one WHERE, one GROUP BY, one HAVING (where condition - CNF = Conjuctive Normal Form = (A1 OR B1 OR C1 OR...) AND (A2 OR B2 OR ...) AND ... AND (An OR Bn or...)

  - query Q:

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = (SELECT MAX(S2.Age)
                           FROM Students S2)
GROUP BY S.SID                                    nested block
HAVING COUNT(*) > 2
```

  - decompose query into a collection of blocks without nesting

```
SELECT S.SID, MIN(E.EDate)
FROM Students S, Exams E, Courses C
WHERE S.SID = E.SID AND E.CID = C.CID AND C.Description = 'Elective' AND
                  S.Age = Reference to nested block
GROUP BY S.SID
HAVING COUNT(*) > 2
```

- then express query block as a relational algebra expression
- GROUP BY, HAVING – operators in the extended algebra used for plans
- query treated as a σ π × algebra expression

$$\pi_{S.SID,\ MIN(E.EDate)}(\ HAVING_{COUNT(*)>2}(\ GROUP\ BY_{S.SID}(\ \sigma_{S.SID = E.SID \wedge E.CID = C.CID \wedge C.Description = 'Elective' \wedge S.Age = value\_from\_nested\_block}(\ Students \times Exams \times Courses\ ))))$$

- Block Optimization = find best plan P for the σ π × expression → evaluate P (result set RS) →
  → sort/hash RS ⇛ groups → apply HAVING to eliminate some groups → compute aggregate expresions in SELECT for each remaining group

---

EXAMPLE

- For the QUERY: SELECT S.SName FROM Exams E, Students S WHERE E.SID = S.SID AND E.CID = 7 AND S.Age > 23 → COSTS ARE HIGH: 1000 +1000*500 = 501 000 I/Os E-1000pages, S-500
  - you first scan E(1000) then join them (for each record in E you parse all in S)
- OPTIMIZATION
  1. to reduce sizes of the relations used for join, we push SELECTIONS and PROJECTIONS first
     basically we apply S.Age > 23 on Students and E.CID=7 on Exams→ then we JOIN
  2. (2nd option) = Investigate use of indexes:

     clustered static hash index on Exams(CID) && hash index on Students(CID)
     a. join is done with Index Nested Loops with pipelining (the result of the selection is not materialized)

b. we first do the E.CID=7 (we have clustered index) then we do the join since we have an index on Students CID and then we do the las selection (S.Age > 23) (we could probably put it ahead of the join if it doesnt mess up the indexes)

Lecture 9 - Evaluating Relational Operators. Query Optimization

- IBM's System R Optimizer → tremendous influences
  - use statistics to estimate the costs of query evaluation plans
  - consider only plans with binary joins in which inner relation is a base relation
  - focus optimisation on SQL queries without nesting
  - dont eliminate dups when projections are used unless DISTINCT is used

Estimating the Cost of a Plan

- estimating the cost for an eval plan for a query block
- for each node N in the three → estimate the cost of correspoding operation + estimate the size of N's result and whether it is sorted → N result = input of N's parent node
- these estimates affect the estimation cost, size and sort order of N's parent
- estimating costs - use data about the input relations (such statistics are stored in the DBMS's system catalogue) like
  number of pages, existing indexes etc.
- optimizer doesnt necessarily find the best plan, but it's purpose is to avoid worst plans and find a good plan

Statistics maintained by DBMS System Catalogue

- are updated periodically, not every time data is changed

- for relation R →
  - cardinality = number of tuples in R
  - size = number of pages in R
- index I →
  - cardinality = number of key values for I
  - size = number of pages for I → B+tree index ⇒ number of leaf pages
  - height = for tree indexes, number of non-leaf levels in I
  - range = minimum / maximum key value in I

Lecture 10 - Distributed Databases

Centralized Database systems

- all data at a single site
- each transaction is processed sequentially
- centralized lock management
- processor fails ⇒ entire system fails

Distributed Database systems

- data stored at several sites
- each site is managed by a DBMS - can run independently (these autonomous components can also be heterogenous ⇒ impact on query processing, query optimization, concurrency control, recovery)

Properties of Distributed DataBase Systems

- Distributed Data Independence =
  - users write queries without knowing the location of the data
  - cost-based query optimization → takes into consideration communication costs & differences in computation costs across sites

- Distributed Transaction Atomicity =
  - users can write transactions accessing multiple sites as they would write local transactions
  - transactions are still atomic = if a transaction commits ⇒ all changes persist
    if it aborts ⇒ none of it's changes persist
- Example: company has 3 offices in Skopje, Caracas, Baia Mare
  - Baia Mare employee data is managed from the Baia Mare office and so on
  - The Company has to access all employees data (ex to compute total payroll expenses)
    ⇒ where should we store employee data?
  - first, you have to store, on each site, data from that site's employee (BM-BM, Sk-Sk, Ca-Ca)
    ⇒ if one of the sites becomes unavailable the others can still work
  - If Caracas(headquarters) has to compute TOTAL Payroll but one of the other sites crashes ⇒ problem → SOLUTION: replicate data such that each combination of 2 sites gives you all the data you need (Caracas = Caracas + Skopje | Skopje = Skopje + BM | BM = BM+ Caracas)

Types of Distributed Databases

- homogenous = same DBMS software at every site
- heterogenous (multidatabase system) = different DBMSs at different sites
  - ⇒ gateaway protocols = mask the differences between different DB servers

Challanges - Distributed Databases

- Distributed Database Design
  - where to store data?
  - depends on data access patterns of most important applications
  - sub-problems: fragmentation and allocation(where to store what data)

- Distributed Query Processing
  - centralized query plan
    - objective - to minimize the number of disk I/Os and execute queries as fast as possible
  - distributed context - some additional factors to consider
    - communcations costs
    - opportunity for paralelism (space of possible query plans are much larger)
- Distributed Concurrency Control:
  - serializability
  - distributed deadlock management (if 2 transactions that depend on each other are executed at different sites)
- Reliability of distributed databases:
  - transaction failures - network might fail OR one/more processors may fail
  - data must be synchronized accross sites

Storing Data in a Distributed DBMS

- accessing relations at remote sites ⇒ communication costs
- → reduce costs → fragmentation / replication

Fragmentation

- fragmentation - split data into more fragments that are stored where they are most used
  - 3 types of fragmentation = horizontal / vertical / hybrid
- Horizontal: subset of rows with one/more similar column(s)
- Vertical: subset of columns - one fragment keeps a few columns, one keeps others, and so on
- Hybrid: subset of rows that also keep only a few columns (vertical + hybrid)

Replication

- replication - store multiple copies of a relation / relation fragment - can be replicated at one or more several sites
  - example: Relation R - Fragments F1,F2,F3 at headq B, but F1 is held at A too (if its used there frequently) ⇒ faster query eval(u can use at A the copy of A) + increased availability of data F1
- types - Synchronous vs Asynchronous

Updating Distributed Data

Synchronous

- Transaction T modifies relation R ⇒ before T commits, it synchronizez all of R's copies
- 2 basic techniques
  - voting = when modifying O, T must write a majority of copies
    when reading O, T2 must write enough copies to make sure it read an updated one
  - read-any write-all = explicit → most common approach, as reads are expensive
- before an update transaction T working with F1 (in the case of fragments presented above) can commit, it needs to lock all the copies of the relation/fragment (with read-any write-all)
  → if one
  site is down, T cannot commit until it is resolved
  → if
  even there are no failures, the process is expensive (commiting many messages) ⇒ async is MORE USED

Asynchronous

- T modifies R ⇒ R's copies are synched periodically | a transaction T2 reading 2different copies of R may see different data, as some copies will be out of sync for a brief period of time

- 2 approaches (difference is between number of updatable copies)
  - Peer-To-Peer site replication → several copies of O can be master copies(updatable) = changes to a master copy is propagated to all others copies of O ⇒ need to resolve conflict when 2 master copies of O are changed in conflicting manner ⇒ adhoc approaches
    - best utilize when conflicts cannot arise = each master site owns a fragment of O (usually horizontal = a few rows) and any 2 fragments that could be updated at different master sites are disjoin
  - Primary-site replication → exactly one copy of O is chosen as the PRIMARY = MASTER copy
    ⇒ site who holds it is
    Primary-Site → other sites can copy O/fragments of O(secondary-site) and subscribe to the primary-site ⇒ all changes to the primary copy of O are propagated to the secondary copies of O
    - 2 steps: capture the changes made by commited transaction + apply to secondary copy
    - log-based capture + continuos apply ⇒ minimizes delay in propagating changes
    - procedural capture + application-driven-apply ⇒ most flexible way to process changes
  - Capture → log-based capture OR procedural-capture
  - log-based capture → the log is used to generate the Change Data Table(CDT) structure:
    write log-tail to stable storage→ write all log records affecting replicated relations to CDT
    - changes of aborted transactions MUST be removed from CDT (so in the end it contains only logs of commited update transactions)
    - smaller overhead and smaller delay, but dependancy on proprietary log details
  - procedural capture → capture is performed with an automatically invoked procedure which takes a snapshot(copy) of the primary relation

    - apply → applies changes collected in the Capture Step (from CDT/Snapshot) to 2nd copies
        - primary-site can continously send the CDT or 2ndary sites can periodically request the CDT or Snapshots ⇒ each 2ndary site runs a copy of the Apply process

Distributed Query Processing

Lecture 11 - Distributed Databases

Baze De Date

https://www.youtube.com/playlist?list=PLB0r2LbiP_PyPQ6De6DXn6e4bOwCa_jSe

- a few notes taken from the lectures from Dan Mircea Sucius lectures

Databases

- large collections of thata stored over a long period of time for analysis purposes & record-keeping
- there are a few data models for databases

Model ierarhic

- tree structure, nodes = entity, relations (1-1, 1-n, 0-1)
- problem: data replication is a must
- Ex: directories and philes within in a computer

Model retea

- graph structure
- no need for duplication of data

Relational Model

- table, primary key, foreign key, constraints etc.

Object oriented model

- classes, attributes, functions, associations, inheritance
- it’s natural, but very bad performance wise

Data storing

- scheme + data ⇒ instance of data
- scheme = structure, defined by user → what data u can store

DBMS - database management system

- tools for creating a database, retrieving data(information), secure data, concurrent access of data, data consistency etc.
- Examples: Oracle, PostregreSQL, MySQL, Teradata, SQLServer etc.
- NoSQL → Redis, Cassandra, MongoDB, CouchDB, Scalaris etc.
- Multimedia DBMS → for videos, images etc.
- GIS - Geographical Information Systems (more layers for the same geographical point)
- Data Warehouse = a ‘database’ of databases → used for reports and such
- Data Stream Managemet System → you constantly get data and when u dont have enough space → compress the most irrelevant ones

Relation Data Model

- multiple relations (tabele)
  - constraints (name:string; sid - primarykey etc.)
- primary key - unique, defining each individual record (like CNP)
- foreign key - refers to a primary key of another relation

SQL Queries

- created by IBM(sustem R) in 1970
- query to get specific data from the DBMS
- Data Definition Language(DDL)
  - define relation scheme → create/delete/alter table and view
  - define constraints
- Data Manipulation Language(DML)
  - operations on instances of a database → insert / remove / update records in a table + selects

Select

- Select * From Students S WHERE S.age = 21 → students with age=21
  Select S.name, S.group ... → retrieves only 2 attributes of students w...
- DISTINCT → remove duplicates
- S in the case above is an alias (alias can be used for columns too)

Strings

- string expressions:
  - WHERE S.Name LIKE 'B_%B' → _ = 1 character, % = 0 or more characters

INNER JOIN

- when you try to get attributes from more than 1 table you use the inner join between 2/more tables on a certain matching condition
  Example: what students are enrolled in what courses
  Student - sid, name, email
  Enrolled - sid, cid, grade
  Course - cid, cname, credits
  SELECT S.name,
  C.name FROM Students S(alias)
  INNER JOIN Enrolled E ON E.sid = S.sid
  INNER JOIN (what you got from above) Course C ON E.cid = C.cid

LEFT OUTER JOIN

- given the example above, if you would want to also get the students that arent enrolled in a course you would use LEFT OUTER JOIN →
  → this gets all records from the Left-side table

RIGHT OUTER JOIN

- similar to left outer join - to also get grades assigned to non-existing student ids (inconsistency in DB)

FULL OUTER JOIN

- combination of both left and right joins

NULL value

- non-existing value of a certain attribute
- when WHERE finds a null value on that attribute will always return False
- IS NULL / IS NOT NULL → special operator

Aggregation Operators

- COUNT(*) → count all from a table
- COUNT([DINSTINCT] A) → count all from a table where attr A ≠ NULL
- SUM([DISTINCT] A) → sum of all values on attr A
- AVG([DINSTINCT] A) → avg of all values on attr A
- MAX(A) → max value of attr A accross all records
- MIN(A) → min value of attr A accross all records

GROUP BY /HAVING

- when selecting smth w a group by, you can only select columns specified in the target-columns of thje group by
- GROUP BY target-columns HAVING qualifications

SORT

- ORDER BY attribute-of-the-select | [DESC] - decreasing

Functional Dependencies

- RELATION STRUCTURE + CONSTRAINTS
- this is why the need of a 3rd relation arises
  example:
  MOVIE DIRECTOR CINEMA PHONE TIME - table should be split into:
  Movies: TITLE DIRECTOR
  Cinemas: CINEMA PHONE
  Screens: CINEMA MOVIE TIME
- Functional dependencies: when a field entirely depends on another
  example: title → director (a movie will always have the same director)
  title is functionally dependent on director
  cinema, time → movie (at a given cinema, at a given time we will certainly have the same movie

Normal Forms

- 1NF, 2NF, 3NF, BCNF etc.
- a db having a normal form means that it is accordingly to certains standards → we dont have to worry about certain problems and so on

Relational Algebra

Projection

- SELECT DISTINCT E.cid, E.grade
  FROM Enrolled E

$$\pi_{\text{cid, grade}}(\text{Enrolled})$$

SELECTION

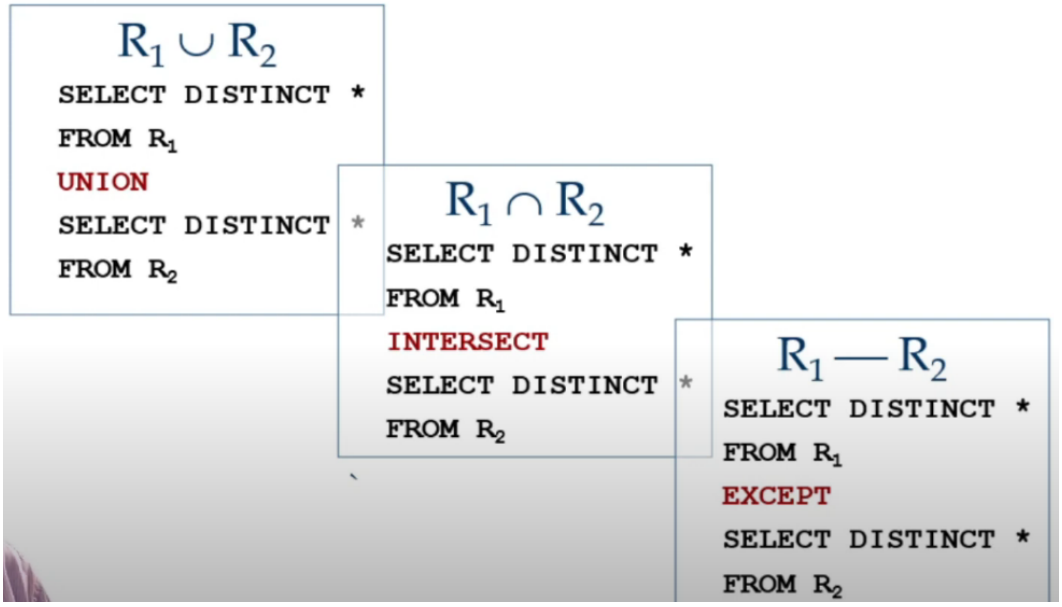
- SELECT DISTINCT *
  FROM Enrolled E
  WHERE E.grade > 8

$$\sigma_{grade > 8}(Enrolled)$$

- Logic operators: AND OR NEGATION

$$(C1 \wedge C2), (C1 \vee C2), (\neg C1)$$

Intersection, Union, Set Difference



Cartesian Product

- R1 X R2 ⇔ SELECT DISTINCT * FROM R1, R2

θ-Join (teta-join)

$$Students \otimes_{Students.sid=Enrolled.sid} Enrolled$$

```
SELECT DISTINCT *
FROM Students,Enrolled
WHERE Students.sid =
rolled.sid
```

```
SELECT DISTINCT *
FROM Students
INNER JOIN Enrolled ON
Students.sid=Enrolled.sid
```

Equi-Join

- joins on the condition c

$$R_1 \otimes_{E(c)} R_2$$

Natural Join

- joins on common name fields

$$R_1 \otimes R_2$$

Rename

- rename R5 to R4 and column a is renamed to b

$$\rho(R_4(a \to b), R_5)$$

Indexes

- used for speeding up the querying process
- they have search key = 1/more fields=attributes → used for SEARCHING
  - not necessarily primary key
- it is used for efficient search based on that search key

- when u modify the relation, you have to modify the data in the indexes too ⇒ ⇒ have only necessary indexes
- indexes are used in main-memory ⇒ it should be as small as possible
  - you can create an index on an index
- 3 types of indexing
  - store all attributes of the relation you made the index on
  - <k=search-key, rid> , rid = memory address where the record that has the search-key = k is
  - <k, list of rids>

Types of indexes

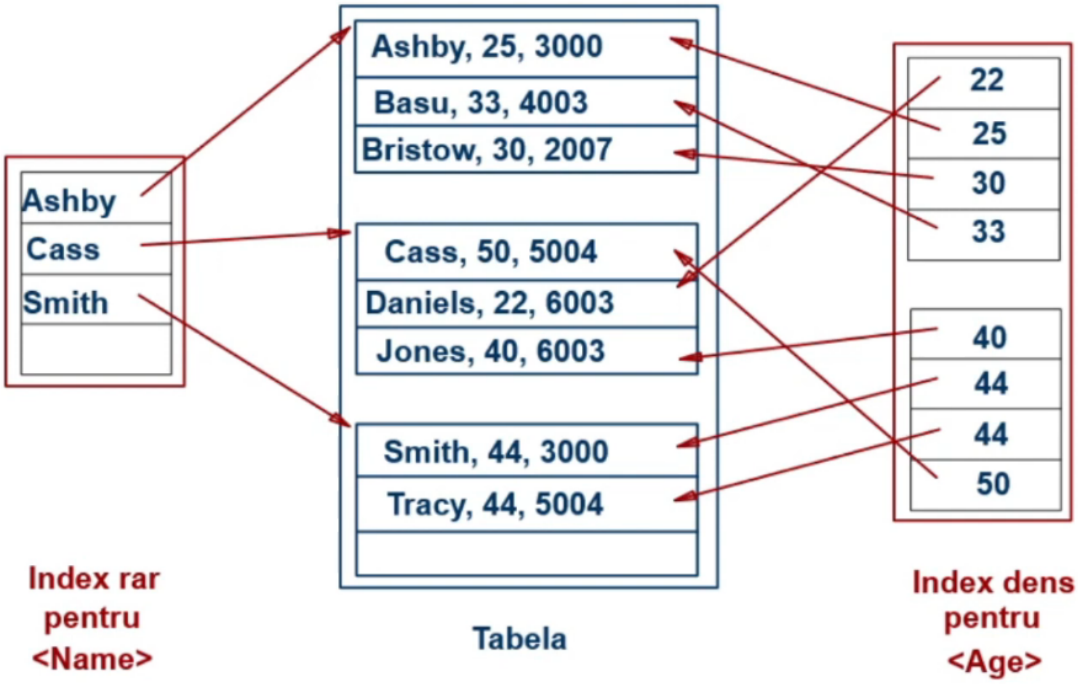
Primary vs Secondary

- primary - includes primary key
- secondary - contain dups

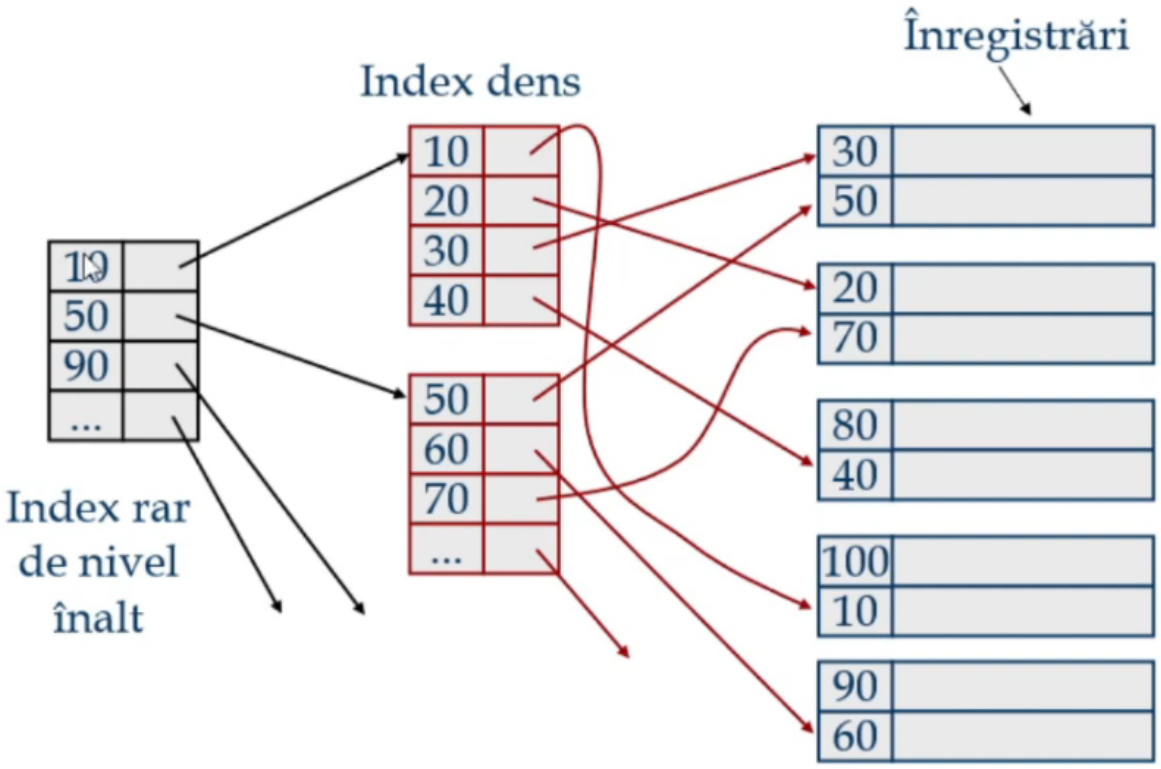
Clustered vs Non-clustered

- clustered - the order of records in the index is approximatively the same as the one in the relation
  - can be clustered for exactly one search key

Rare vs dense (?)



- MULTILEVEL INDEXES



Tree as indexes

- better for inserting / deleting data
- sorted binary trees used
- to optimize → use binary trees such that for each node, his furthest leaf on the left and on the right have approximatively the same height (LeftH - RightH = 0/1/-1)
- ⇒ there are some cases when trees become unbalanced ⇒ need for balancing

B Tree

- most popular method for organising indexes
- a node in a B Tree can contain multiple values
- each node keeps keys and pointer to their sub-trees
- each leaf is equally distanced from the root

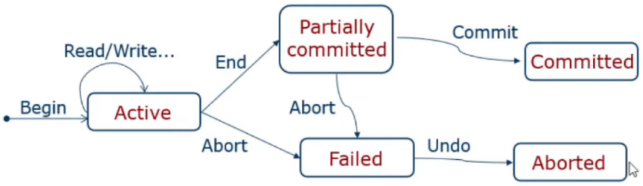
- 

Transactions

- a sequence of more operations compose a transaction
- transactions must leave the DB in a consistent state
- we use transactions for concurrency
- for the DBMS, a transactions is a series of reads / writes → inserts, updates, deletes matter

Steps

- BEGIN → READ → WRITE → END
  - → COMMIT-transaction (if everything was right)
  - → ABORT-transaction (if smth went wrong ⇒ rollback)
  - Undo / Redo



ACID Properties

ATOMICITY

- either all operations in the transactions are commited, or none

- IF a transaction fails - aborts → the changes must be undone / rollbacked

Consistency

- no constraints can be violated

Isolation

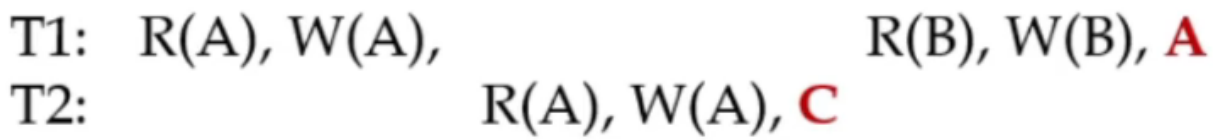
- concurrency must always be possible → serializability
- if more transactions are concurrently executed, the effect on the databse must be equal as if they were executed in a series:
  just like if we executed T1 first and only after it commited we executed T2 or the other way

Durability

- when a transaction commits successfully we must see the results in the DB

Dirty Reads

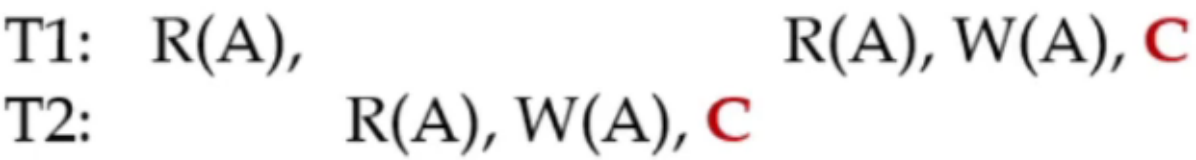
- reading uncommited data
- in the case below, when T1 aborts → A is rollbacked to the value it had before executing T1, so all the changes done by T2 are also not stored;
- T2 did a dirty read, as it read an uncommited value of A



Unrepeatable Reads

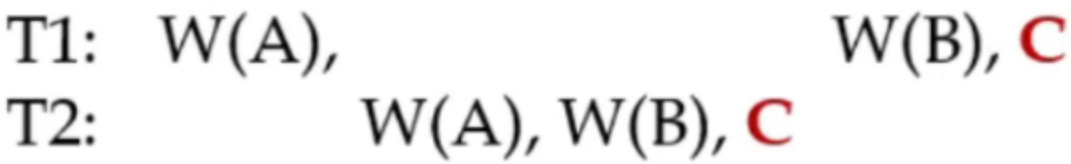
- when a transactions reads smth twice, and it is different the 2nd time
-

- *Unrepeatable Reads* (conflict RW):



Blind Writes

- overwriting uncommited data
- B will have the value written by T1 and A will have the value written by T2



Phantom Reads

- when an insert is made between a transaction selects for the 1st time and then for the 2nd time

Transaction Planning

SERIAL SCHEDULE

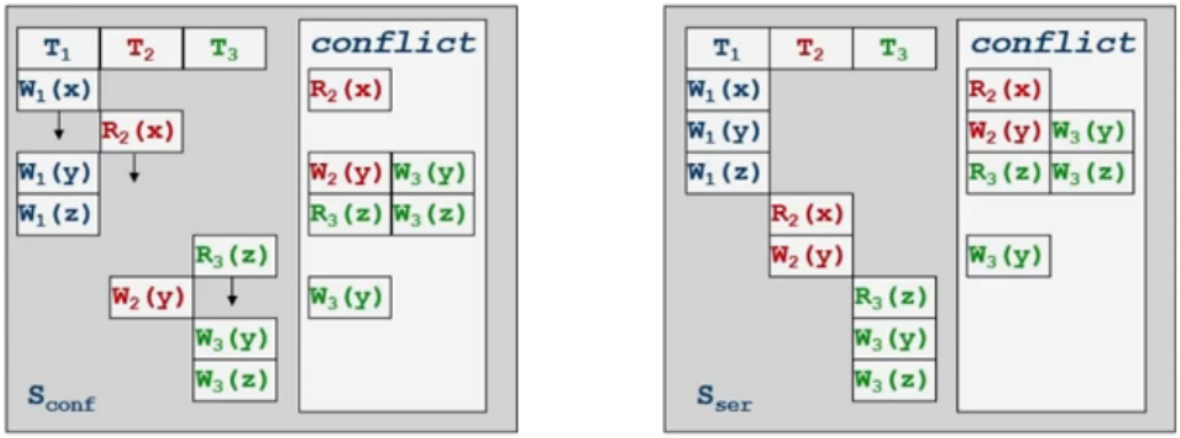
- a transaction starts only after the other finishes
- they are not concurrent

NON-SERIAL SCHEDULE

- non-serial, transactions intersect
- the result must be the same as in a serial schedule

CONFLICT SERIALIZABLE

- a schedule must be equivalent to a serial schedule in order to conform ACID principles
- conflict equivalent ⇒ two schedules that have the same conflicts



- the 2 above have the same conflicts ⇒ the 1st schedule is OK
- CONFLICT SERIALIZABLE = has the same conflicts as a serial schedule

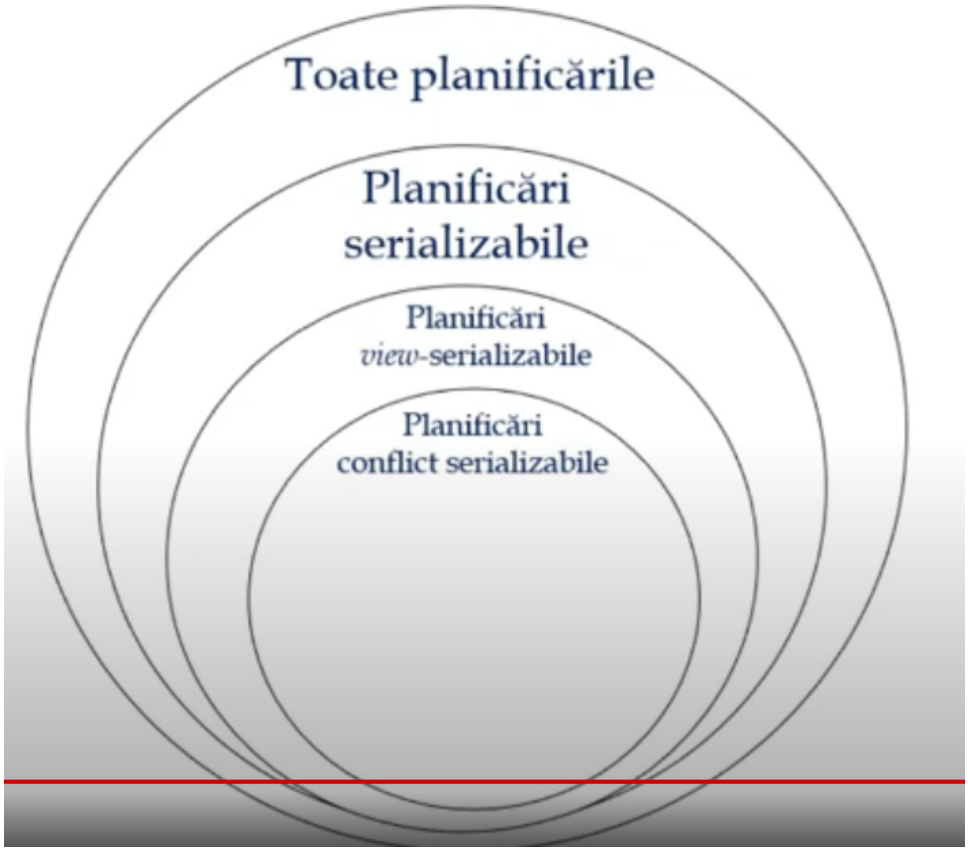
PRECEDENCE GRAPH - test for conflict serializability

- oriented graph, 1 node for each transaction
- T1 → T2 if a read/write on object O is executed by T2 after one is performed by T1
  except if there are only reads one after the other and no writes in between
- no cycles ⇒ conflict-serializable

VIEW SERIALIZABLE - test for conflict serializability

- two schedules S1 and S2 are view-equivalent if:
  - if Ti is the first transaction to read A in S1, it must be the same in S2
  - if Ti reads A modified by Tj in S1, it must be the same in S2
  - if Ti is the last to modify A in S1, it must be the last to modify A in Tj as well

Serializability Overview



CONCURRENCY CONTROL

Lock Based Concurrency

- when a transaction uses an object, it blocks it, so no other transaction can use it

Shared Lock

- if a transaction has a shared lock on an object, other transactions can read it

XLock

- if a transaction has an XLock on an object, no other transaction can read/write it

2PL = Two Phase Locking protocol

- not the best
- a transaction cant read/write without aquiring an S/Xlock on an object

First Phase

- transaction gets all needed locks
- until all Locks are aquired, no locks are released

2nd Phase

- after transactions got all the locks needed, they will unlock them in REVERSE ORDER

Strict 2PL

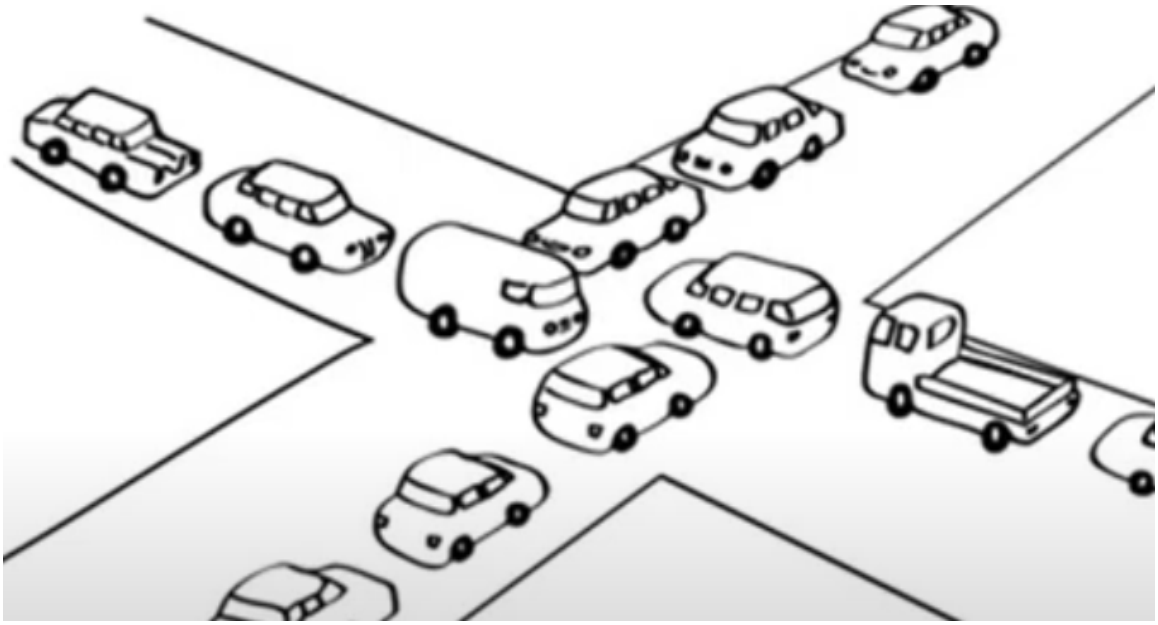
- similar to 2PL, but it will release it’s locks only after commiting, not after finishing working with a data object
- allows only serializable schedules

Lock table

- a table that stores which table has which type of lock on which objects
- also stores a list of transactions that are waiting for a lock

DEADLOCKS

- deadlocks occur when a transaction waits for another that waits for another that waits for the 1st transaction



- you can either DETECT or PREVENT a deadlock

Prevent Deadlocks

- the older the transaction, the higher priority (the faster it gets a lock on an data object)
- if Ti wants access to an object Locked by Tj:
  - WAIT-DIE: if Ti has higher priority, waits for Tj; otherwise Ti dies
  - WOUND-WAIT: if Ti ahs higher priority, Tj dies; otherwise, Ti waits
- if an aborted transaction is restarted, it is given it's original timestamp

Detect Deadlocks

A. Wait-For Graph

- arc from Ti to Tj if Ti is waiting for Tj to release it's lock
- cycle in the graph ⇒ deadlock
  - if a cycle exists, one transaction from the cycle is aborted
- Choosing the Deadlock Victim criteria:
  - number of objects modified by the transaction
  - number of objects that are to be modified by the transaction

    - numbers of locks held
    - if a transaction is repeatedly chosen as a victim, it should be allowed to proceed at some point

B. Timeout Mechanism

- if a tranasction T waits too long for a lock, it's assumed that there is a deadlock so T is terminated

Distributed Databases

Centralized Database

- same location, same server
- if system fails, db can't be accessed anymore

Distributed Database

- fragmented database on multiple location, different servers, hard disk, cpus etc.
- Database = all those fragments
- PROBLEMS → Atomicity, Consistency
    - it is more complex, harder to manage
- PROs:
    - faster access
    - disponibility
    - modularity
    - local autonomy
- CONs:
    - fragmenting & allocation of data → where to store it and how to fragment it
    - Concurrent Queries → communication costs, opportunities for parallel processing

- Concurrency Control → serializability, deadlocks; propagating modifications→ all modifications have to be the same for each site
- Consistency → synchronizing data

DBMS in a Distributed Database

- one single DBMS for all
- a DBMS for each site ⇛ a master server = gateway is needed → this one is what makes communication possible between different sites
  - synchronous
  - asynchronous

Fragmenting Data

Horizontal fragmenting

- data records (rows in a table) are stored in different sites

Vertical

- all data records, but only certain attributes

Properties of storing fragmented data

- Completeness → stores all records
- Disjunction → doesn't store the same information twice
  (primary key in case of vertical fragmenting)
- Reconstruction → joining all fragments, we get exactly the whole relation

Replication

- storing certain data in different sites at the same time
- ADVANTAGES:
  - in case one site fails, that data can still be access
  - faster query evaluation

- PROBLEMS:
    - one site modifies a fragment that is present in other sites ⇒
      ⇒ synchronous / asynchronous propagation of modifications

Synchronous

- a modification has to be immediately propagated to all the other fragments that contain the modified data
- Technique A (Read-any Write-all)
    - transaction T1 modifies O: T1 must write all copies of O
    - transaction T2 reads O: T2 can read any copy of O
    - more used, as reads occur more often than writes
- VOTING TECHNIQUE
    - to modify object O, a transaction T1 must write a majority of its copies
    - when reading O, a transaction T2 must read enough copies to make sure it's seeing at least one current copy
    - each copy has a version number

Asynchronous

- relations are updated every once in a while
- Peer-To-Peer technique
    - there are certain “master copies” → all master copies must be modified instantly, and in time the master copy/copies are transmited to secondary copies
    - PROBLEM: when 2 master copies are modified at the same time
    - Example: used when master copies store completely disjunct fragments of data
    - Example: only one master copy can get access to a data object a time
- Principle-Site replication
    - only one master/primary copy

- all other are subscribers → secondary
- when a user wants to use a secondary copy → it goes to the master copy and it is updated before the user can perform any operation on that secondary copy
- subscribers also interogate the primary copy from time to time
- CAN BE → Log-Based OR Procedural
- Log-Based → Change Data Table (CDT) = all commited modifications on a certain fragment → aborted transaction modifications must be removed from CDT ⇒ CDT only keeps commited transaction modifications
- Procedural → it periodically takes snapshot of the database ⇒ when a data is accessed by a user, that data first takes the most recent snapshot

Database Security

- access rights for each specific user (what tables he can read / modify)
- the one who creates a certain table / view , that person has complete rights over that table/view
- GRANT privileges ON object TO users [WITH GRANT OPTION]
  - privileges: SELECT, INSERT(nume-col)/UPDATE(nume-col), DELETE, REFERENCES - can define foreign keys to that specific relation
- REVOKE - to revoke privileges from different users , on different objects
  - if we do REVOKE to user X and user X gave privileges to other users ⇒ all those users that got privileges from X will have their privileges REVOKED
- Privileges based on ROLES
  - each role has certain rights on certain tables / views / columns etc.
  - you can give one/more roles to each individual users

Mandatory access

- each object in the DB has a security class

    - each user has a certain permission for a security class
    - not used that much, but it's more secure than role / privilege based

Bell-LaPadula model

- security class → Top Secret > Secret > Confidential > Unclassified
- User S can read object O only if: class(S) >= class(O) (

Retrieving Data

Atomicity

- you rollback transactions that failed (didn't commit) when a system crash occurs
- so if one transaction was still active at the time of the crash, all the modifications done by it have to be rollbacked

Durability

- you must guarantee that modifications made by commited transactions must remain on the DB

Checkpoint

- stops all transactions and forces all current modifications on disk
- executed any m minutes or t transactions

Updating data

- immediately → you update the disk right after a modification occurs
- suspended → all modifications go onto the disk after a certain number of transactions commit
- in-place → the original page on the disk is overwritten by the page on the buffer modified

- shadow → the buffer page is copied onto a temporary hard-disk copy of the original
  so if O is the orignal and Oc is a copy on disk, shadow will instantly update Oc; in case of a system crash, just ignore Oc

Write-Ahead Logging Protocol

- first, a log entry of the current modification has to be stored in the log before anything can be stored on the hard disk → for rollback purposes
- all log entries must be on the log before a transaction commits

Buffer Manager vs Recovery Manager

- Buffer Manager = optimising → as few I/Os operations (input or output onto disk)
- Recovery Manager = atomicity & durability
- conflict between the 2 - buffer wants to save to disk less than recovery manager → buffer wins

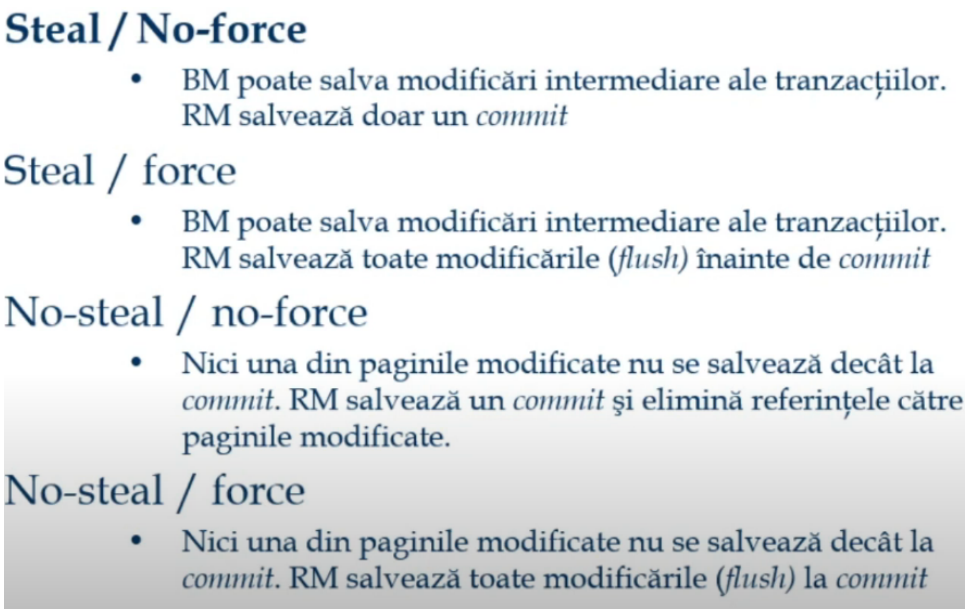
STEAL VS NO-STEAL

- can BufferManager store to disk before instructions from recovery manager?
- when main-memory is full and the bm wants to read a name page to the buffer page → one page must be written to disk
  - Buffer Manager can’t store pages that do not have pin count 0 (number of transactions that currently use this page)
  - Recovery Manager - you don’t store intermediate transaction updates, only final ones
- STEAL = buffer manager steals a page from the recovery manager and stores it to disk, even if the transaction is still ongoing
- No-Steal = RM still has a reference to all modified pages in the buffer

FORCE VS NO-FORCE

- When a transaction fully commits → RM wants to store it to disk, BM doesn’t

- FORCE → RM forces the BM to save the modifications made by a transaction to disk
- No-Force → BM wins



Recovery Distributed Databases

- if a transaction works with data from multiple sites and the system crashes, we must assure that we don't store data on any of the sites → more complicated protocol
- Each site has it's own log protocol

2PC - Two Phase Commit Protocol

- we have a main-site and secondary-sites
- the main-site sends a "prepare" message to secondary-sites
- secondary sites → send back yes/no message
- if ALL secondary sites send a "yes" message → main-site send another message, "commit",
  else → main-site send an "abort" message → secondary sites "end" message (in log) and "done" to coordinator
- Log Record → TransactionID, CoordinatorID = main-site

- abort / commit messages/commands from the coordinator have a coordinatorID and a transactionID
- How is data Recovery achieved in 2PC?
- Log → if there's no "end" → rollback

LOCKS

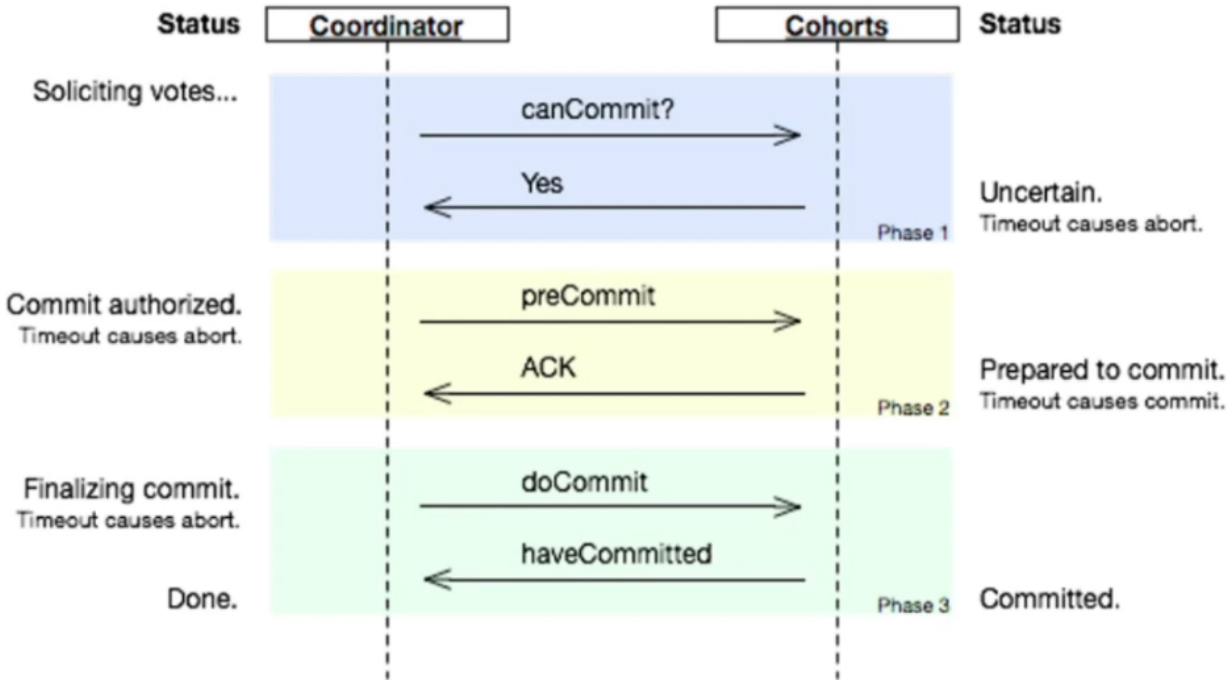
- if coordinator is locked, no subordinate can do anything until a command from it
- if a subordinate doesnt respond during protocol → if he didn't respond to "prepare" → abandon T
- if he responded to prepare but didnt respond to commit/abort → waits
- when "done" is sent, then transaction T knows that it's sub-transaction that sent that "done" has stored successfully to disk
- if the coordinator fails after prepare, but before commit/abort ⇒ transaction T fails
- if a sub-transaction doesnt modify BD, then coordinator can ignore it

2PC (with Presumed Abort ?)

- doesn't wait for "done" messages
- sub-transaction send "reader" to prepare, if they dont modify data

3PC - Three Phase Protocol

- no message received back = abort (except if "doCommit" is not sent back, in this case coordinator sends a special-message if it fails)



ARIES protocol of Data Recovery

- each log entry has a Log Sequence Number (LSN)
- each page has a pageLSN = LSN of the most recent log entry that modified the page
- system also has fluhsedLSN = max LSN up to which everything is saved to disk

Log Entry

- LSN
- prevLSN
- TransID
- pageID
- length
- offset
- before-image

- type - Update, Commit, Abort, Checkpoint, End= success of a commit/abort, CLR =for Undo

CLR

- when an operation is Undone, a CLR is added to signal that
- simply shows what operations were canceled (undo)
- prevents multiple undos

Transaction Table

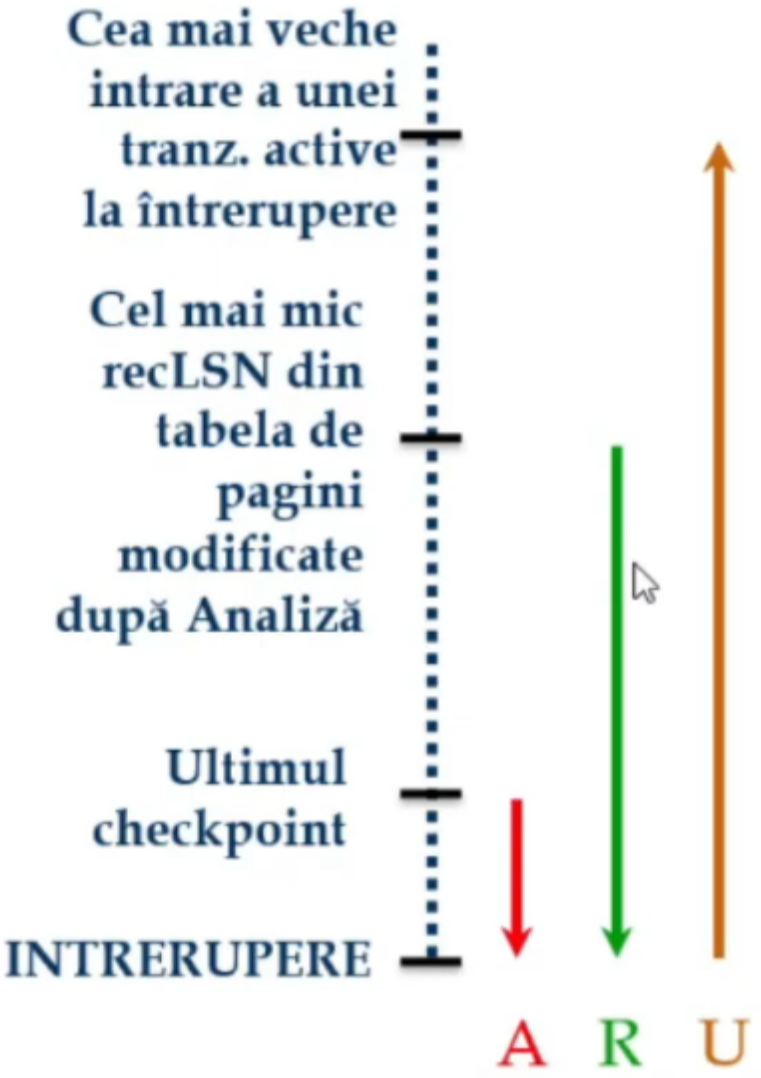
- a record for each active transaction
- Has → transactionId, state(running / commited / aborted) & lastLSN → last operation executed by the transaction

Dirty Page Table

- one record for each page modified in the buffer
- also has: recLSN = LSN of the 1st log entry that modified that page

ARIES Phases

1. Analysis = browse LogTable from the most recent checkpoint to the final to identify all active transactions and all pages modified in the buffer at the moment of the system failure/crash
    a. encounters end ⇒ remove
2. Redo = redo all operations that modified buffer pages and commited before the crash, to asure that the updates where stored to disk
3. Undo = active transaction's updates are all undone → from last to first, using prevLSN



NoSQL Databases

- MongoDB, Cassandra etc.
- less used than Relation Databases(>50 years old) - ACID properties
- What is wanted from a DB:
  - Availabiliy (”five nine” availability)
  - Scalability (vertical/horizontal)
  - Performance

What is NoSQL?

- Definition 1: all databases that do not use SQL
  - also no Object-Oriented-Databases
- Definition 2: Not Only SQL
  - Cassandra: java-like query language, CQL
- Definition 3:
  - DOES NOT use relation model for structuring data
  - DOES NOT allows data access with SQL standard

Main-Differences

- Relational Databases
  - join operations
  - indexes
  - ACID properties
  - row based
- CAP Theorem - an alternative
  - Consistency -
  - Availability -
  - Partition Tolerance -
  - CAP theorem says that no DB can have all 3 properties
- BASE Model (the alternative approach from ACID properties)
  - Basic Availability - available most of the times (it seems to be working most of the times)
  - Soft-state - writing consistency isnt needed (copies do not need to be consistent always)
  - Eventual consistency - DB will, at some point, be consistent

    - Read Repair / Delay Repair
        - Read Repair = make it consistent at each read
        - Delay Repair = not controller by user ops
- NoSQL db can be
    - column-based = you go column by column and take every-row for each of that columns
    - document-based = no actual db design - no structure, no constraints, just like MongoDB
        - Database - collection of data
        - Collection - container for documents
        - Document - component of the collection (like a table)
        - Field - similar to relational model
        - Embbeded Document - similar, but not exactly like JOIN
        - Primary key
        - Secondary key
        - INDEXES - B tree

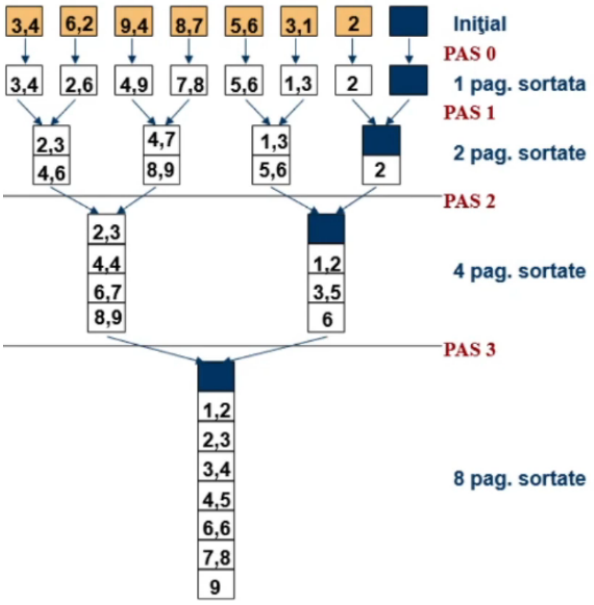
Evaluating Relational Operators. Query Optimisation

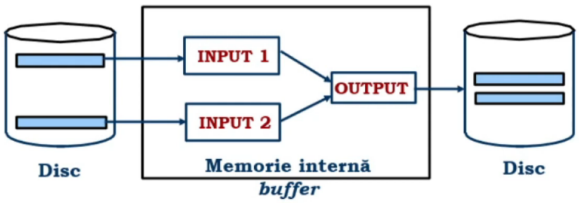
Sorting

- used for sorting, removing duplicates, for group by and for joins
- Quick Sort, Heap Sort etc. are efficient, but all the data that needs to be sorted might not fit inside the internal memory ⇒ External Sort - we sort as much as we can fit inside main-memory, we temporarily store that on disk, repeat, until all is sorted, than sort again → repeat
- Optimisation = reduce number of I/Os from the hard-disk

Merge Sorting

- need 3 free pages inside buffer
- Step 0: read each page, sort that page, save it temporarily on hard disk
- Steps 1,2,3...: 3 pages → 2 input pages and one output page, merge sorting them → save to disk, repeat

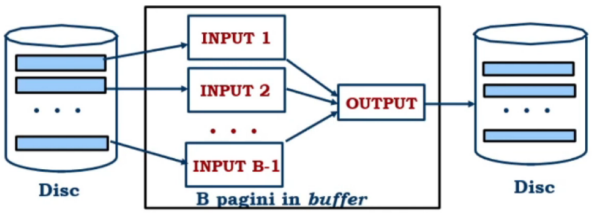




- total-passes= 1+log2(N)
- total cost = 2*N*(1+log2(N))

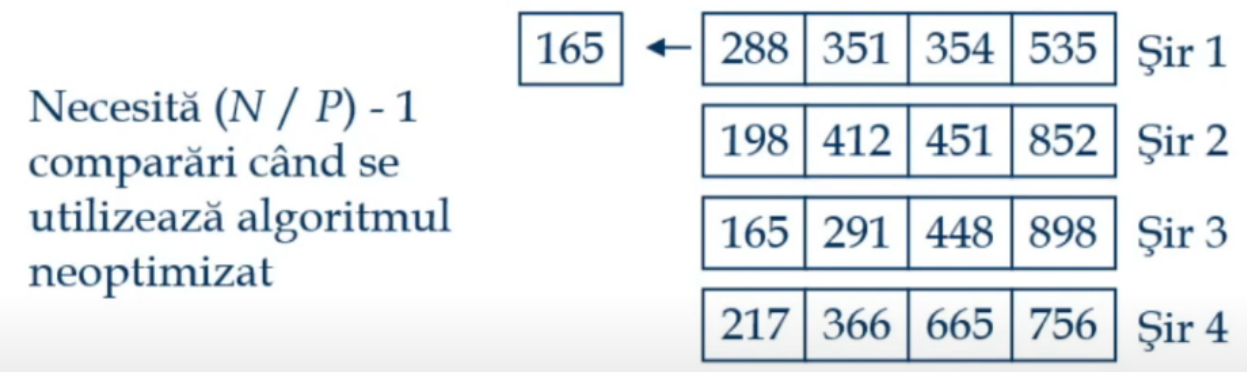
OPTIMISATION

- if you have more free pages, use all those pages for mergin
- STEP 0: use all B pages at a time to sort the N pages from the Disk → N/B steps
- STEPS 1,2,3... : merge B-1 pages at a time
- total-passes: 1 + logB-1(N/B)
- cost = 2*N* [total-passes]
- → more buffer pages ⇒ faster sort

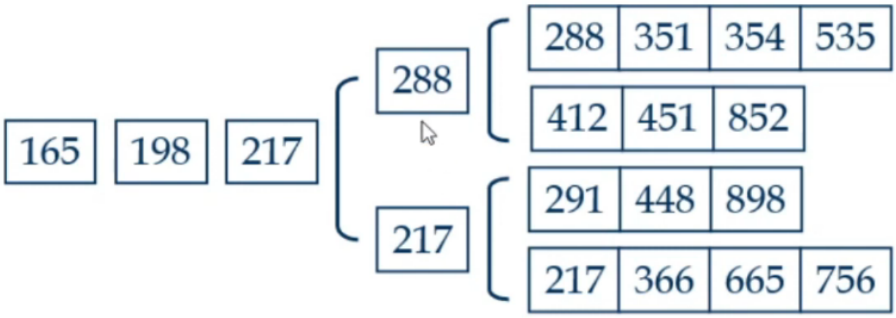


Selection Tree

- Problem: when trying to merge the pages it takes a lot of time to find the minimum



- Solution: decision tree



Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui
Elementele noi sunt "împinse" în față
Procesul se repetă până când tot arborele se goleşte

Evaluating Relation Operators

STUDENTS - 80 RECORDS / PAGE, 500 PAGES

COURSES - 80 RECORDS / PAGE, 100 PAGES

EVALUATIOSN - 100 RECORDS / PAGE, 1000 PAGES

Join

SELECT *
FROM Evaluations E
INNER JOIN Students S on E.sid = S.sid

- Cartesian product then selecting would be too inneficient
  M = pages in evaluation, Pe = records / page
  N = pages in S , Ps = records / page
- Metric = Number of used pages (I/Os - input / output operations)
- Iteration → Simple, Page Oriented, Block
- Index → Index Nested Loop Join

Simple Nested Loop Join = (M + Pr * M * N)

forEach tuple r in E do
    for each tuple s in S do
        if r.sid = e.sid then add <r,s> to result

- forEach record in the external table E, we scan each tuple from the internal relation S
- uses just 3 pages

STEPS

- we read each page in E
  - for each of those pages, for each tuple in this page, we read each page in S

COST

- M + Pe * M * N = 1000 + 100 * 1000 * 500 I/Os
- M (pages in E) + Pe*M*N (for each record in E → Pe * M, we read all N pages)
- better if we make S the external table and E the internal one ⇒
- ⇒ N + Ps * N * M = 500 + 80 * 500 * 1000 I/Os

Page Oriented Nested Loop Join = (M + M * N)

This time, once we read a page from E and one page from S, we will compare each tuple r in that page from E with each tuple s in that page from S; unlike SNLJ, where for each tuple in a page we would read each page and start again.

- uses just 3 pages

STEPS

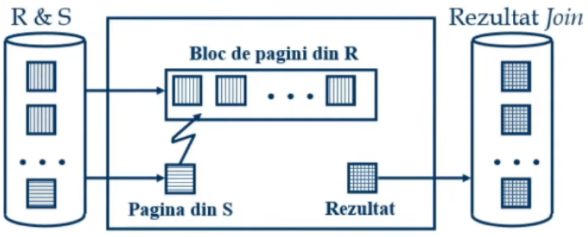
- we read each page in E
  - we read a page from S and we scan all compare all records in PageE with all records in PageS, instead of all pages from S with tuple1 in PageE, then reread all pages from S for tuple2 in PageE and so on

COST

- much better than before
- M + M * N = 1000 + 1000 ∗ 500 I/Os = 501 000 I/Os
- M (each page in E once) + M * N (for each page in E, we read each page in S) → better to use S as external as it has less page ⇒ 500 500 I/Os

Block Nested Loops Join - (M + [M/B] * N)

- generalisation of Page Oriented Nested Loop Join, but we use more pages
- let's say we have B available pages in Buffer ⇒ 1 output, for the external relation we read B (-2) pages and we read each page in S for all those B pages we read from E



COST

- let's say B = number of pages in buffer
- M + [M/B] * N = 1000 + [1000/100] * 500 = 6000 I/Os
- M (external relation) + [M/B] (how many times we can read from E, parte intreaga) * N (each page in the internal relation) = 1000 + [1000/100] * 500

- if we use S as the internal relation
  N + [N/B] * M = 500 + 5 ∗ 1000 = 5500 I/Os

Index Oriented Nested Loop Join - ( )

- when we have an index on the Join column of one of the relation ⇒ use a binary search on the index to see
- cost → M + M*Pe * costOfFindingSRecord
- costOfFindingSRecord = costIndexSearch + costReadRecord
  - costIndexSearch → 2-4 for B tree, 1.2 for direct-access index
  - costReadRecord → clustered index = 1 I/O
    → unclustered index = 1 I/O for each tuple in S (worst case scenario)

COST

EXAMPLE - direct-access index on S.sid

- M + Pe * M * (1.2+1) = 1000 + 100 ∗ 1000 ∗ 2.2 = 221,000 I/Os
- 1.2 → direct access index
- 1 → to read each tuple

EXAMPLE - direct-access index on E.sid

- N + Ps * N * 2.2 = 500 + 80 ∗ 500 * (1.2 + 1/2.5) = 88,500 / 148,500 I/Os
- 1.2 → index
- 1 → if the index is clustered, then we know all of those records with the same E.sid are grouped on the same page
- 2.5 → we can only assume that each student has an equal amout of Evaluations ⇒
  ⇒ 100 ∗ 1000 Evals / 500∗80 Students = 100,000 / 40,000 = 2.5

Sort Merge Join - M * log2M + N*log2N + (M+N)

- we sort both relations before using external merge sort with all pages →

- → we then store them in E′ / S′ →
- → we then use those E′/S′ relations:
  we take a page from E and one from S and we keep browsing through records to see if we have a match →
  so for this step we read exactly M + N pages
  - cost for this could be M*N in the following case:
    - each tuple in E and each tuple in S have exactly the same value e1.attribute1 = e2.attribute1 = ... = em.attribute1 = s1.attribute1 = ... sn.attribute1
- COST OF SORTING RELATION WITH N PAGES
  = N * log2N

COST

- M * log2M + N*log2N + (M+N) = 7500 I/Os

Sort Merge Join Optimised - (B > Rad(M) → 3*(M+N))

- when you sort pages from E / S and get monotonies, you keep thoso monotonies as they are and do the “interclasare” with them directly
- ⇒ you basically have to read E and S three times each
- THIS ONLY WORKS IF YOU HAVE ENOUGH PAGES TO READ ALL THE MONOTONIES (SORTED PAGES) AT ONE TIME IN THE BUFFER
  - B > Rad(L) , L = number of pages from the bigger relation

Hash-Join (B > Rad(M) → 3*(M+N))

- We hash each relation and we obtain B-1 partitions.
- We then apply a second hash function h2 with B-2 pages from the partitioned E, and for each page from the partitioned S we try to match them
- COST = 3*(M+N), but we need to be able to fit the partition of R in the main-memore
  - B > Rad(M)

Evaluating Select & Projections

Example 1

- SELECT * FROM STUDENTS S WHERE S.NAME < 'C%'
- if we have no index, not-sorted ⇒ scan all students → cost = N
- if we have no index, sorted ⇒ binary-search → cost = Log2N (number of pages)
- if we have index on Name
    - direct-access → 1.2
    - B tree → 2-4
    - Clustered → approx what % of pages would contain names that start with A/B/C →
      50 pages (3 letters/30 letters ⇒ 500 pages/10)
    - Unclustered → same approximation, but we might need to actually to read one for each of those record (500*80 records / 10 → 4000)

Example 2

- (day < 8/9/24 AND grade = 10) OR cid=5 OR sid=3
- Step0: convert to CNF ⇒ (day < OR cid = OR sid =) AND (grade = OR cid= OR sid=)
- Step1:

Example Projection

SELECT DISTINCT E.A E.B FROM EVALUATIONS E

COST = N + T + TLog2T + T → sort

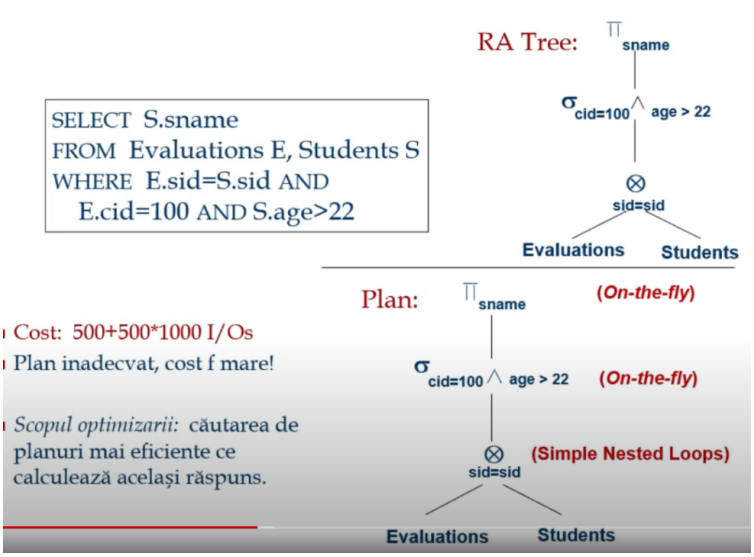
COST = N + 2T → HASH

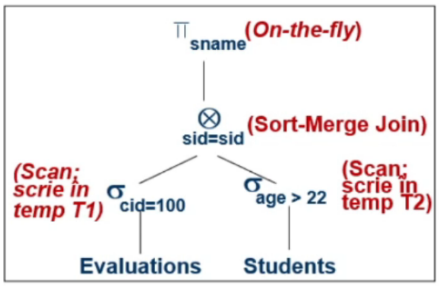
Query Optimisation

- Index scan - if we use a single attribute from a relation

- Table scan - if we use more and we dont have an index on all of them
- Sorted scan - 1st sort the relation, then access it
- Can’t find exactly the best plans, but avoid the worst ones
- System R optimizer → only left-deep join plans
  - we join 2 plans, then we join that result with another relation and so on;
  - tries to filter Cross products

EXAMPLE





- we do the projection first to minimise the cost of the X prd

---

RECAP

Transfer from site A to site B

- td = time to read a page → applied everywhere external is used
- ts = time to ship page → applied anywhere internal is used (from another site)

With caching

- i think this means that you only read each page once and it remains there (?)

Simple Nested Loop Join

Cost

- (M + Pr * M * N)
- M = number pages external rel
- N = number pages internal rel
- Pr = number records/page external rel

How it works

- for each tuple in ExternalRel
  for each tuple in InternalRel
  if condition = true ⇒ add <tuple1, tuple2>

Page Oriented Nested Loop

Cost

- M + M * N
- M = number of pages external relation
- N = number of pages internal relation

How it works

- we read each page from external relation R
  - for each of this pages, we read each page from S → for each tuple r in PageR, we go through each tuple in PageS

Block Nested Loop

Cost

- M + [M/(B-2)] * N
- B = number of buffer pages

How it works

- similar to Page Oriented, but we use B-2 pages for reading from R

Index Nested Loops Join

- if there is an index on the join column of S, internal table

Cost

- M + M * Pe * (cost of finding corresponding tuple in S)

Hash Join

Partitioning Phase

- hash both E and S on the required column with function “h”

Probing Phase

- take all pages from E, one from S and one for output
- use a hash function h2 != h to match records from S to ones in E

COST

- 3(M+N)

Encode a message with a key

Steps

1. Associate each character with a code
    a. Example: each letter with a number like
    a b c d e f g h i j ...
    1 2 3 4 5 6 7 8 9 ...
2. Write the key and it’s codes under, in a table and do the same for the message
    a. Split the message into length(key) fragments
    b. Under each fragment, write, in order, the codes of the key

c. Sum them up → you get the codes of the encoded message

Serializability

- a schedule has the same effect on a database as a serial-schedule

Precedence Graph

- one NODE for each transaction in schedule S
- one arc from Ti to Tj if an action in Ti precedes an action in Tj and conflict it
  - you draw Ti → Tj if Ti is the one that does smth first and there is conflict
- cycle in graph ⇒ non-serializable

View Serializability

- S1 and S2 are view equivalent if:
  - if Ti reads the initial value V in schedule S1, then Ti also reads the initial value V in S2
  - if Ti reads the value of V written by Tj in S1, then Ti also reads the value written by Tj in S2
  - if Ti writes the final value of V in S1, then Ti also writes the final value of V in S2

Locks

SLock = shared lock (read lock)

- a lock that allows T to read data
- other transactions can also get an SLock on that object

XLock = exclusive lock (write lock)

- a lock that allows T to read and modify
- no other transaction can get a lock on that object

Lock Table

- structure used by lock manager to keep track of granted lock & lock requests
  - one entry has the following
    - lock type → SLock / Xlock
    - number of transactions holding a lock on that object (1 for XLock, 1/more for SLock)
    - pointer to a queue of lock requests

Transaction Table

- one entry / transaction
- keeps a pointer to a list of locks held by transaction

Locking Protocols

2PL = Two-Phase Locking

- before a transaction can read/write ⇒ must get lock on object
- once a transaction releases ONE LOCK ⇒ can't acquire others
  - ⇒ 2 phases :
    - Growing Phase → transaction acquires locks 1,2, ..., n
    - Shrinking Phase → transaction releases locks n, n-1, ..., 1

Strict 2PL

- same as 2PL AND the following
- all locks are released when transaction T completes execution
- only serializable schedules

Deadlock

- if transactions that wait for one another to acquire a lock in their Growing Phase are in a cycle
- if an aborted transaction → restarted ⇒ initial timestamp

Prevention

- each transaction has a starting TIMESTAMP
- OLDER ⇒ HIGHER PRIORITY

1. Wait-Die
    a. T1 wants access of object O locked by T2 with a conflicting lock
        i. T1 higher priority > T2 ⇒ T1 can wait
        ii. T1 lower priority < T2 ⇒ T1 dies
    b. T2 wants lock on object X taken by T1
        i. If T2 has lower priority ⇒ it dies ⇒ now T1 gets lock on object O
2. Wound-Wait
    a. T1 higher priority > T2 ⇒ T2 DIES
    b. T1 lower priority < T2 ⇒ T1 waits

Detection

1. Wait-For-Graph
    a. make a graph with a node for each transaction
    b. arc from Ti to Tj if Ti waits for Tj
    c. cycle in graph ⇒ deadlock ⇒ one transaction from cycle is aborted
    d. Choosing a Victim depends on
        i. number of modified transaction → more = less likely to be victim
        ii. number of objects to be modified → more = more likely to be victim
        iii. numbers of lock held
        iv. is a transaction has been chosen to be victim many times ⇒ wont be victim again
2. Timeout mechanism
    a. if transaction T waits for too long → assume deadlock → terminate T

ACID

Atomicity

- either all operations are executed or NONE

Consistency

- a transaction must leave the db in a consistent state
    - transactions are correct and dont violate integrity constraints

Isolation

- a transaction is protected from the effects of others

Durability

- the effects of a commited transaction must remain

Conflicts

Write - Read conflict = Read Uncommited / dirty read

- reading data previously written by another transaction
- T2 reads data written by T1

Read - Write conflict = Unrepeatable Reads

- reading data twice, but seeing a different value without modifying it
- T2 writes data read by T1

Write - Write conflict = overwriting uncommited data = Blind Write

- overwriting uncommited data
- T2 writes data written by T1

Phantom Read

- when an insert is made between two reads of a transaction by another

ISOLATION LEVEL

- NO DIRTY WRITES → XLock when writing

Read Uncommited

- lowest degree of isolation
- T can read data prev modified by ongoing transaction T2
- no SLock needed
- problems → dirty read, unrepeatable read, phantom read

Read Commited

- SLock required to read data, released immediately
- XLock required to write data, released at end of transaction
- however T1 can write data previously written by ongoing T2
- problems → unrepeatable reads, phantom read

Repeatable Reads

- a transaction T can only read commited data
- must acquire SLock - read / XLock - write
  - both of which are released only after commiting
- problems → phantom reads

Serializable

- transaction T can only read commited data
- if T reads on object on a search predicate (table), this set cannot be changed while T in progress
- must acquire SLock / Xlock which are released at the end
- NO PROBLEMS

Fragmentation

Horizontal

- subset of rows
- are disjoint
- reconstruction → union of fragments

Vertical

- subset of rows
- are not disjoint, both keep at least primary key
- reconstruction → natural join

Access path

- selectivity → the number of retrieved pages when using the access path to obtain the
  desired tuples

Indexing

- hash index matches if operation is “=” for the search-key elements
- i is a b tree index w search key <a,b,c,> ⇒ a,b,c works | a,b works | a works

Relation Algebra

Projection

- pi symbol ⇒ selects only a few columns

Selection

- sigma symbol ⇒ has a where clause

Data Replication

Synchronous

- voting = when modifying O → modify majority of copies
  when reading → read enoug copies to make sure you read an updated one
- read-any-write-all = most common approach, you read any and write all

Asynchronous

- Peer-To-Peer = more master copies
  - utilised when the fragments are disjunct
  - problem when 2 master copies are modified at the same time
- Primary-site-replication = only one master copy
  - capture either log based = ChangeDataTable → log tail to stable storage
    or procedural with snapshots from time to time
  - apply → apply either changes from CDT or from the snapshot
    - primary sites can either continously send CDTs or
      2nd sites ask before reading for CDT / snapshot

Reduction Factor

7. Consider the query:
SELECT *
FROM R1, R2, R3
WHERE p1 AND p2 AND p3
The conditions tested by the predicates in the WHERE clause are statistically independent. The cardinality of a relation R is denoted by |R|. The reduction factor associated with predicate p is denoted by RF(p). The cardinality of the query's result set can be estimated by:

- response is
- B

a. $\frac{|R1|*|R2|*|R3|}{RF(p1)+RF(p2)+RF(P3)}$
b. $|R1| * |R2| * |R3| * RF(p1) * RF(p2) * RF(p3)$
c. $RF(p1) * RF(p2) * RF(p3) - (|R1| + |R2| + |R3|)$
d. $|R1| + |R2| + |R3| + RF(p1) + RF(p2) + RF(p3)$
e. none of the above answers is correct.

Writing Objects

steal approach
• T's changes can be written to disk before it commits
• transaction T2 needs a page; the BM chooses F as a replacement frame (while T is in progress); T2 steals a frame from T
• no-steal approach
• T's changes cannot be written to disk before it commits
• force approach
• T's changes are immediately forced to disk when it commits
• no-force approach
• T's changes are not forced to disk when it commits

no-steal approach
• advantage - changes of aborted transactions don't have to be undone (such changes are never written to disk!)
• drawback - assumption: all pages modified by active transactions can fit in the BP
• force approach
• advantage - actions of committed transactions don't have to be redone
• by contrast, when using no-force, the following scenario is possible: transaction T commits at time t0; its changes are not immediately forced to disk; the system crashes at time t1 ⇒ T's changes have to be redone!
• drawback - can result in excessive I/O

ARIES

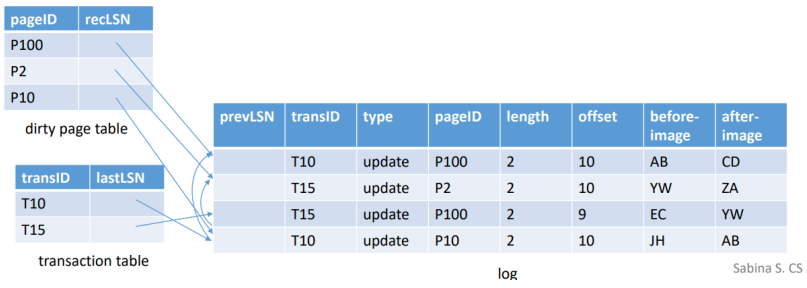
- ARIES
  • recovery algorithm; steal, no-force approach
  • system restart after a crash - three phases:
  • analysis - determine:
  • active transactions at the time of the crash
  • dirty pages, i.e., pages in BP whose changes have not been written to disk
  • redo: reapply all changes (starting from a certain record in the log), i.e., bring the DB to the state it was in when the crash occurred
  • undo: undo changes of uncommitted transactions
  • fundamental principle - Write-Ahead Logging
  • a change to an object O is first recorded in the log (in a log record LR)
  • LR must be written to stable storage before the change to O is written to disk

Transaction Table

- has 1 entry / active transaction
- fields: active transaction id and lastLSN = LSN of the most recent log record for that transaction
  - also has status = commited, aborted, in progress(not shown)

Dirty Page Table

- 1 entry / page modified
- fields: page id + recLSN = record that FIRST modified that page



Reduction Factor

- index ⇒ 1/NKEYS

•

column1 = column2
• indexes I1 on column1, I2 on column2
⇒ RF: 1/MAX(NKeys(I1), NKeys(I2))
• only one index I (on one of the 2 columns)
⇒ RF: 1/NKeys(I)
• no indexes
⇒ RF: 1/10
• column > value
• index I on column
⇒ RF: (IHigh(I) - value) / (IHigh(I) - ILow(I))
• no index on column or column not of an arithmetic type
⇒ a value less than 0.5 is arbitrarily chosen
• similar formulas can be obtained for other range selections
Sabina S. CS
Estimating Result Sizes
• column IN (list of values)
⇒ RF: (RF for column = value) * number of items in list (but at most 0.5)
• NOT condition
⇒ RF: 1 - RF for condition

Projection based on sorting

1. Step 1 = scan E → get set E′ that contains only desired columns
    a. cost → scan = num.pages in E I/Os +
    write temporary = T I/Os (T is O(M) and depends on no columns and their sizez
2. Step 2 = sort tuples in E′, sort key = all columns
    a. cost = O(TlogT) (also O(MlogM)
3. Step 3 = scan sorted E′ → compare adjenct tuples & eliminate duplicates

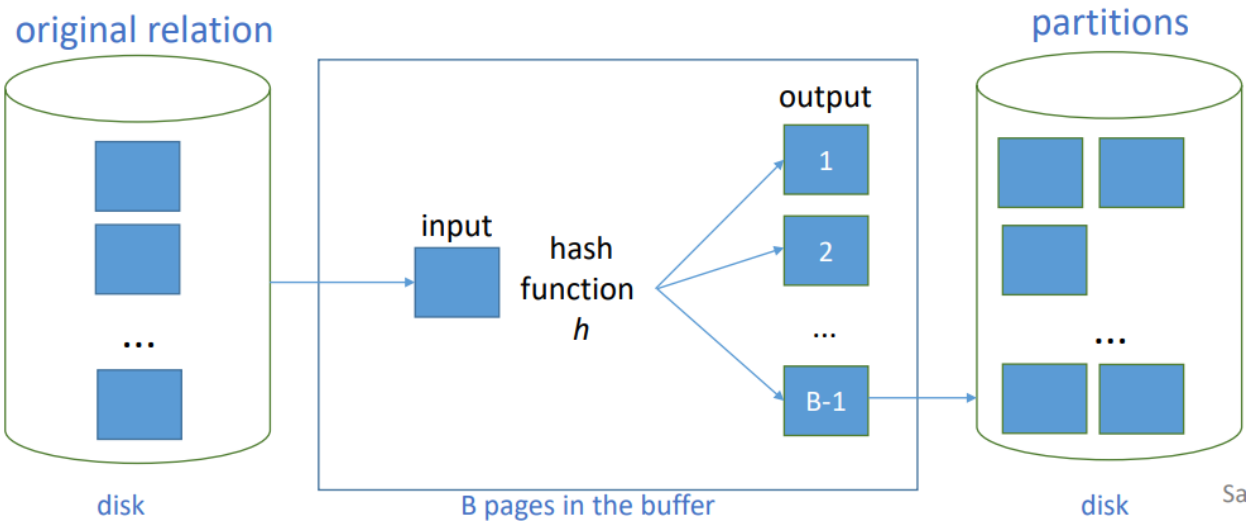
Projection Based on Hashing

- Phases = partitioning & duplicate elimination

PARTITIONING

- 1 input Buffer Page (read one page at a time)→ for each tuples t remove unwated field ⇒ t′ → apply hash function h to t′ → write t′ to the ouput buffer page that is hashed to by h ⇒
- B-1 partitions on B-1 Buffer Pages for output (one output page / partition) = = collection of tuples with
  no unwanted fields and common hash value

  (tuples on different partition are 100% distinct)



DUPLICATE ELIMINATIONS

- for each partition → read partition P → build in-mem hash table with hash fu h2 ≠ h on all fields
- if new tuple hashes to the same value as an existing tuple → check if they are distinct & eleminate duplicates
  - write the duplicate free hash table to the result file (+ clear in-mem hash table)
- In case of partition overflow→ apply hash-based projection technique recursively(sub partitions)
- COST = partitioning → read E= M I/Os | write E= T I/Os | eliminate dups= read partitions: T I/Os