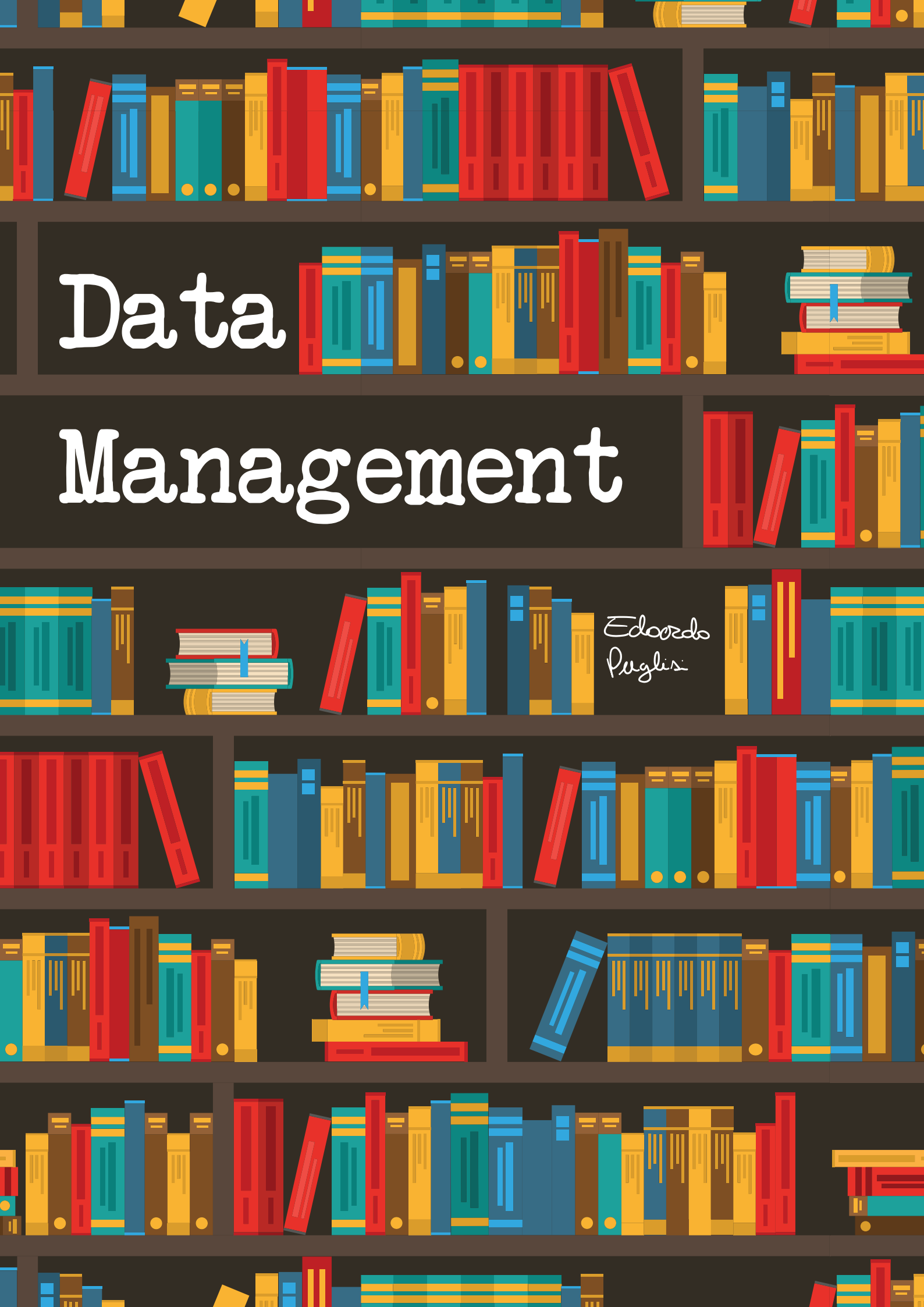

Data
Management
Edoardo
Puglisi

DATA BASE MANAGEMENT SYSTEM (DBMS)

NOSQL = not only SQL → relational model is still important!
SQL is a mix between Relational Algebra (sequence of operations to relations) and Relational Calculus (first-order logic)
Relational Algebra is composed by all expressions obtained from Union, Difference, Cartesian Product, Projection and Selection.

Union: $R \cup S$ = tuples in R or in S
Difference: $R - S$ = tuples in R but not in S
} R and S with some attributes
Cartesian Product: $|R \times S| = |R| \times |S|$ = combine everything from R with everything in S
Projection: removes all attributes not "projected" + eliminate duplicates. Can be used to remove attributes.
Selection: select tuples satisfying a given condition (boolean expression)

Intersection: $R - (R - S)$
Join: select tuples from $R \times S$ satisfying a given condition. Equijoin when condition is an equality. Natural Join is the equijoin on common attributes.

i.e.

R

A	B
.	.
.	.
.	.

S

A	C	D
.	.	.
.	.	.
.	.	.

→ Natural Join = $PROJ_{R.A,B,C,D}(SELECT_{R.A=S.A}(R \times S))$
[$R \bowtie S$]
↳ to avoid duplication of A

Improvement 1: pick R1 and join with S, then R2 ... and so on. No need to store whole $R \times S$
Improvement 2: sort on common attributes and stop earlier if needed.

Key: attribute for which there aren't two tuples with same value on it.
Foreign Key: attribute which value in every tuple must be contained in an other table (some attribute or not)

i.e.

User

ID	Name
1	Bob

Order

TOT	User ID	Prd.
5	1	Vodka

eg Graduated (grade, mark, school)
School (scode, city)

- Find cities where there is at least one school with a student graduated with 100

$PROJ_{city}(SELECT_{MARK=100}(Graduated \ JOIN_{school=scode} \ SCHOOL))$

$PROJ_{city}(SELECT_{MARK=100}(Graduated) \ JOIN_{school=scode} \ SCHOOL)$

} → equivalent, select first filters graduated tuples reducing JOIN size. Have they are equivalent

- Find schools where no students have mark = 100

$PROJ_{scode}(School) - PROJ_{school}(SELECT_{MARK=100}(Graduated))$

- Find the cities where all schools have a student graduated with 100

$PROJ_{city}(School) - PROJ_{city}(Select_{mark=100}(Graduated) \ JOIN_{scode=school} \ (School))$

• For all school, find students who graduated with min mark in the school.

$\text{PROJ}_{gcode,school}(Graduated - \text{PROJ}_{gcode,mark,school}(Graduated \text{ JOIN}_{mark>m \text{ and } school=s} (\text{PROJ}_{m \leftarrow mark, s \leftarrow school, g \leftarrow gcode}(Graduated))))$

↳ self Join — rename

SQL

```
Select A1 ... An
from   R1, ... RN
where  <condition>
```

$\Rightarrow \text{PROJ}_{A1 \dots An}(\underbrace{\text{Sel}_{cond}(R_1 \times R_2 \dots \times R_N)}_{\text{Join}})$

eg query 1

```
Select cities                          Select cities
from School, Graduated        ⇔       from School JOIN Graduated ON school=scode
where mark=100 AND school=scode        where mark=100
```

eg query 2

```
SEL scode
FROM School
WHERE scode NOT IN (SEL scode
                    FROM Graduated
                    WHERE mark = 100)
```

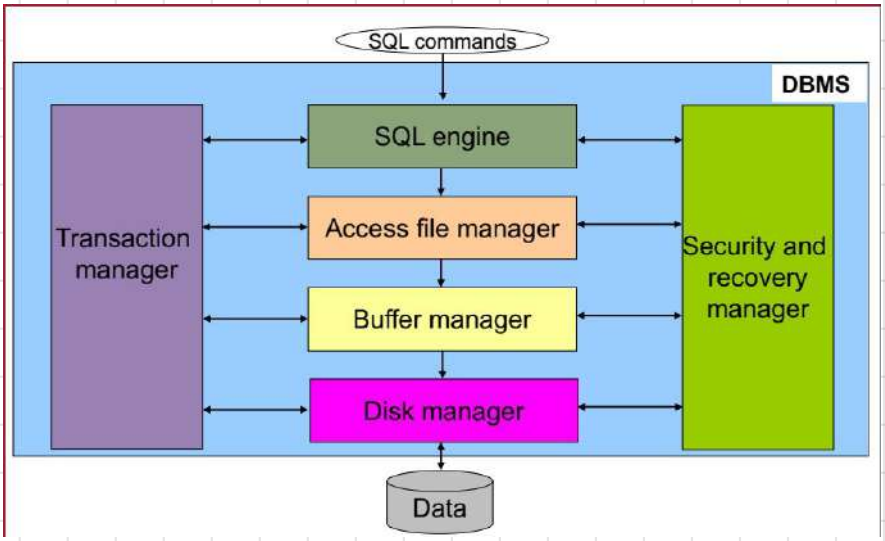
eg query 4

```
SEL gcode, school
FROM Graduated
WHERE gcode NOT IN (SEL a.gcode
                    FROM Graduated a, Graduated b
                    WHERE a.mark > b.mark AND a.school = b.school)
```

Select distinct ... = remove duplicates

Select A_1, SUM(A_2) From T_1 Group by A_1 = all A_1 duplicates are merged together and A_2 are added together. Group by usually is used together with functions, in this example "SUM".

BUFFER MANAGEMENT



Database files are set of pages stored in blocks. These pages must be brought to main memory from secondary storage (HD/SSD) to be used in size of blocks.
Buffer is the portion of main memory used to compute these pages.
Buffer manager does the transfer job.
Buffer is organized in frames (main memory pages) of 2Kb - 64Kb size.
Buffer manager associates each page ID (PID) to the corresponding frame number.

FIX: load a specific page into the buffer. For each frame the buffer manager mantains:
- infos about which frame it contains (if any)
- pin-count: how many transactions use the page in the frame.
- dirty: value that indicates if the content of frame has been modified or not (true/false).

eg. Module M issues FIX for page P
① Buffer Manager (BM) checks if P already in some frame
② Yes → increments that frame pin-counter
③ No:
  3a) choose frame F according to certain replacement policy
  3b) if F is dirty (and a steal strategy is used) the page contained in it is written back to secondary storage
  3c) P is loaded into F, pin-count initialized to 0 and dirty to false.
④ Address of F is sent to M.

UNFIX: transaction notify that it doesnt need the page anymore and pin-counter is decremented
USE: transaction modifies the content of frame and dirty set to true
FORCE: Synchronous transfer to secondary storage of page contained in frame
FLUSH: Asyndhonous transfer to secondary storage of pages used by a transaction.

REPLACEMENT POLICY

The frame is choose among frames with pin-count = 0 if possible. If not, request is placed in a queue or the transaction is aborted. Otherwise:

① LRU: in a list of frames with pin-count = 0 the "least recently used" is picked
② Clock Replacement:
- Variable "current" points to frame in circle
- Each frame has a bit "referenced" that tell us if page contained in it has been used recently, but pin-count = 0
- When we need a frame we analyze the one pointed by "current"
- If pin-count > 0 → go next
- If pin-count = 0 and referenced = true → referenced = false and go next.
- If pin-count = 0 and referenced = false → choose the frame.

TRANSACTION MANAGEMENT AND CONCURRENCY

Transaction: set of reads and writes that act as a single logical unit
Every transaction contains "begin", "end", "commit" and "rollback" instructions.
Throughput: accepted transactions per second by a system (tps) → 100 - 1000 tps in DBMS
This means concurrency: if a transaction needs 0.1 sec and we want 100 tps → 10 concurrent transactions at least.
To solve problems of concurrency (work on same value for example) DBMS ensure the **isolation** property: any transaction is executed as it is the only transaction in the system.

ACID:

- Atomicity: for every transaction, all-or-none actions are executed
- Consistency: transactions don't compromise integrity constraints
- Isolation
- Durability: after commit, effects are registered and database permanently.

eg. SELECT is a transaction.

Schedule: sequence of actions of a set of transactions respecting the order within transactions.
- Partial if only of some transaction of the set. - Serial if there is no interleaving (no concurrency)
- **Serializable**: if equivalent to serial schedule (same output, but different actions order, for every input)

Serializability CAN'T be applied in practice because checking if two "programs" are equivalent is an indecible problem!
⇒ Relax the notion of equivalence (and also transaction definition)

ANOMALIES

- Reading temporary data, WR (write read) anomaly

T_1	T_2
begin	begin
READ(A,x)	
x := x-1	
WRITE(A,x)	
	READ(A,x)
	x := x*2
	WRITE(A,x)
	READ(B,x)
	x := x*2
	WRITE(B,x)
	commit
READ(B,x)	
x:=x+1	
WRITE(B,x)	
commit	

Serial result (T1 then T2) ⇒ A=250 B=250
In this case instead A=250 B=150
Its called write-read anomaly cause a transaction writes an element and a second one read it

- Update loss, RW anomaly

Serial result (with A=2) ⇒ A=4
In this case we have A=3 because the first update is lost: T2 read A=2 and writes A=3

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
	READ(A,x)
	x := x+1
WRITE(A,x)	
commit	
	WRITE(A,x)
	commit

- Unrepeatable read, RW anomaly

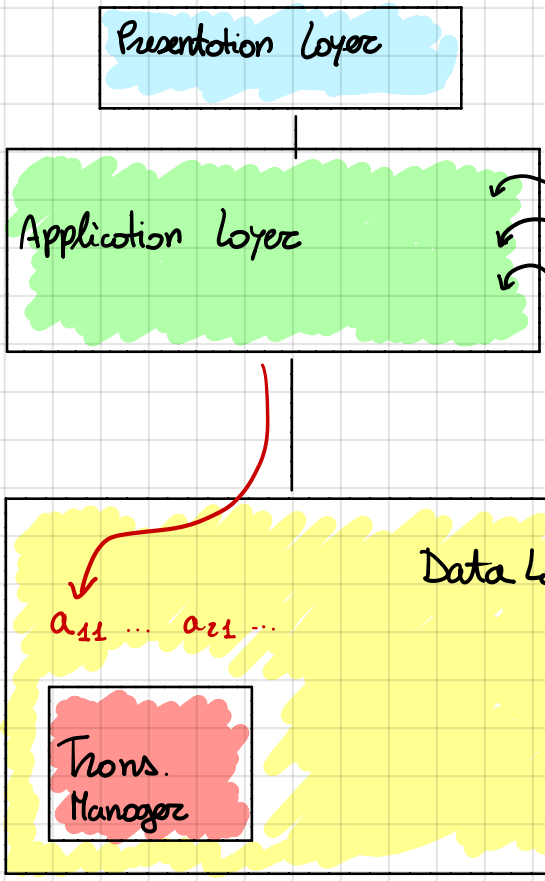
T_1	T_2
begin	begin
READ(A,x)	
	x := 100
	WRITE(A,x)
	commit
READ(A,x)	
commit	

T_1 read two different values.

- Ghost update, WW anomaly

T_1	T_2
begin	begin
WRITE(A,1)	
	WRITE(B,2)
WRITE(B,1)	
commit	
	WRITE(A,2)
	commit

Assume the integrity constraint A = B
T1 will see update from T2 as surprise, same for T2 from T1. Both violating the constraint.



T_1 created (user 1 wants to buy ticket)
T_2 created (user 2 wants to buy ticket)
T_3 created (user 1 wants to give score to buying service)

Constraints on transaction order are managed by application layer, not transaction manager
eg T1 before T_3

SCHEDULER:
① manages identifiers of transaction
② Ignores operations in the local store
③ Decides if a schedule is acceptable or not
eg commands
C1 = commit T1
R1(A) = T_1 reads A
W_1(A) = T_1 writ A

Assumptions:
① No transaction read/write same element twice and no read on element that it has written
② No rollback (for now)

We have two notions of serializability:

① VIEW-SERIALIZABILITY

- In a schedule S, we say that "$r_i(x)$ READS FROM $w_j(x)$" if $w_j(x)$ precedes $r_i(x)$ in S and there is no writes $w_k(x)$ between $w_j(x)$ and $r_i(x)$
- In a schedule S, we say that $w_i(x)$ is a FINAL-WRITE if $w_i(x)$ is the last write action on x in S

VIEW-EQUIVALENCE: let S_1 and S_2 be two (complete) schedules on some transactions. Then S_1 is view-equivalent to S_2 if S_1 and S_2 have same READ-FROM relation and same FINAL-WRITE set.

⇒ a (complete) schedule S on $\{T_1 \dots T_n\}$ is view-serializable if there exists a serial schedule S' on $\{T_1 \dots T_n\}$ that is view-equivalent to S

eg. read1(A,t) read2(A,s) s:=s+100 write2(A,s) t:=t+100 write1(A,t)

Serializable? Yes because $T1 \to T2$ and $T2 \to T1$ give same output ($A=100$)

View-Serializable?

$r_1(A)\ r_2(A)\ w_2(A)\ w_1(A)$ dont consider local store op. ($t:=t+100$ etc)

$\alpha: T_1 \to T_2$ ⇒ violates FINAL-WRITE rule

$\beta: T_2 \to T_1$ ⇒ violates READ-FROM rule

} → no view-equivalence ⇒ no view-serializable

CHECK view equivalence of $S1$ and $S2$ from $\{T_1 \dots T_n\}$

① compute READ-FROM on $S1$ and $S2$ and compare

② compute FINAL-WRITE set on $S1$ and $S2$ and compare

Equivalence is not decidable → view-equivalence yes (Ptime)

CHECK if $S1$ in $\{T_1 \dots T_n\}$ is view-serializable

① try all possible serial schedule in $\{T_1 \dots T_n\}$ and see if at least one is view-equivalent to $S1$

eg. S: $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$ is it view-serializable?

S': $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$ same final-write state ($w_2(z)$) and every reads still read from same write

⇒ serializable

eg S: $r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$

- $T_1 \to T_2$ changes final-write ×
- $T_2 \to T_1$ changes read-from because $r_1(x)$ will read from $w_2(x)$ ×

View-serializability is hard to compute though ⇒ CONFLICT-serializability

② CONFLICT-SERIALIZABILITY

In the same way, we say that a schedule is conflict-serializable if exists a serial schedule CONFLICT-EQUIVALENT

What's a conflict: two actions are conflicting in a schedule if they belong to different transaction, they operate on same element and AT LEAST one of them is a write.

Two schedules $S1$ and $S2$ on some transactions are conflict-equivalent if $S1$ can be transformed to $S2$ through a sequence of swaps of consecutive non-conflicting actions.

Th. $S1$ and $S2$ are conflict-equivalent iff there are no actions a_i of T_i and b_j of T_j such that:

- a_i and b_j are conflicting, and
- mutual position of them in $S1$ is different from the mutual position of them in $S2$.

How to check conflic-serializability in PTIME?

Build precedence-graph P(S) of schedule S where:
- nodes are $\{T_1 \dots T_n\}$ of S
- edges $T_i \to T_j$ iff there exists $P_i(A)$ and $Q_j(A)$ such that:
  - $P_i(A) <_S Q_j(A)$ $\Rightarrow P_i(A)$ comes before $Q_j(A)$ in S
  - at least of them is a write.

$O(n^2)$

Every edge shows a constraint of a serial schedule to be conflict-equivalent to S.
$T_i \to T_j \Rightarrow T_i$ must be before T_j
Presence of cycles means the impossibility to build a serial schedule conflict-equivalent, iff it is acyclic then S is conflict-serializable

PRELIMINARY LEMMA: if S1 and S2 are conflict equivalent then P(S1) = P(S2)
Prove: $P(S1) \neq P(S2) \Rightarrow$ at least one constraint on S1 is not satisfied on S2
Inverse is not valid
eg. $S_1: w_1(A)\, r_2(A)\, w_2(B)\, r_1(B)$ } $P(S_1) = P(S_2)$ but S_1 is not conflict-equivalent to S_2
$S_2: r_2(A)\, w_1(A)\, r_1(B)\, w_2(B)$

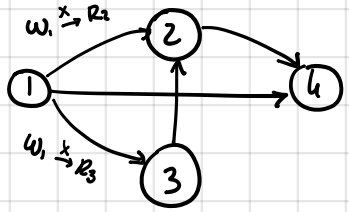
TOPOLOGICAL ORDER

Sequence of nodes such that if $T_i \to T_j$ is in the graph then T_i appears before T_j in the sequence.
- if the graph is acyclic then there exists at least one topological order → Alg 1
- if S is a topological order of G and there is a path from n1 to n2 then n1 is before n2

Alg 1: pick node without incoming edge, output it (add to sequence), delete it, repeat...

CHECK conflict serializability:
- build P(S) from S
- check if P(S) is acyclic
- return true if acyclic, false otherwise

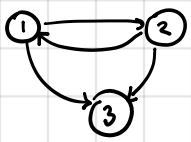
eg. $w_1(x)\, r_2(x)\, w_1(z)\, r_2(z)\, r_3(x)\, r_4(z)\, w_4(z)\, w_2(x)$



$\Rightarrow$ Top. Order : 1324 $\to T_1\ T_3\ T_2\ T_4$
yes!

Th. S1 and S2 on same transaction. If S1 and S2 are conflict-equivalent then they are view-equivalent
Th. If S is conflict-serializable then it is view-serializable too. Inverse doesn't hold.
↳ Prove: !view eq $\Rightarrow$!conflict eq → given $S: w_1(y)\, w_2(y)\, w_2(x)\, w_1(x)\, w_3(x)$ we find a not view equivalent serial schedule $S': T_3\, T_2\, T_1$. $P(S') != P(S) \Rightarrow$!conflict equivalent

eg. $w_1(y)\, w_2(y)\, w_2(x)\, w_1(x)\, w_3(x)$
P(S) cyclic → not conflict serializable
but view-serializable to 1 2 3



ORDER PRESERVING CONFLICT SERIALIZABILITY

CSR = class of conflict serializable schedules

Order Preservation: a schedule is OPCS if it is conflict equivalent to a serial schedule S' and $\forall t, t' \in trans(S)$: if t completely precedes t' in S then the same holds in S'. OCSR defines class with this property.

$OCSR \subset CSR$

eg $w_1(x) r_2(x) c_2 w_3(y) c_3 w_1(y) c_1 \quad \in CSR \quad \notin OCSR$

COMMIT – ORDER PRESERVING CONFLICT SERIALIZABILITY

A schedule S is commit order preserving conflict serializable if $\forall t_i, t_j \in trans(S)$: if there are conflicting actions $p \in t_i, q \in t_j$ in S such that p precedes q in S then c_i precedes c_j in S. COCSR denotes this class.

$COCSR \subset CSR$

A schedule S is in COCSR iff there is a serial schedule S' conflict equivalent to S such that $\forall t_i, t_j \in trans(S)$: t_i precedes t_j in S' iff c_i precedes c_j in S.

$COCSR \subset OCSR$

eg. $w_3(y) c_3 w_3(x) r_2(x) c_2 w_1(y) c_1 \quad \in OCSR$

$\notin COCSR$

A scheduler based on conflict-serializability maintains the precedence graph → very costly for commercial systems where there are huge graphs.

How to trade off in practice → LOCKS

CONCURRENCY WITH LOCKS

IDEA: transactions can ask permission to access resource. Lock ($l_i(A)$) and UNLOCK ($u_i(A)$)

Rule 1: Every transaction is well-formed ⇒ every action $p_i(A)$ is in a critical-section i.e. in a sequence of actions delimited by a pair of lock-unlock on A eg T_i: .... $l_i(A)$ .... $p_i(A)$ ... $u_i(A)$...

Rule 2: The schedule is legal ⇒ no transactions in it locks A when a different one has already locked it and still not unlocked eg S: .... $l_i(A)$ .... $u_i(A)$ ...

(no $l_j(A)$)

eg S: $l_1(A)\, l_1(B)\, r_1(A)\, w_1(B)\, l_2(B)\, u_1(A)\, w_1(B)\, r_2(B)\, w_2(B)\, u_2(B)\, l_3(B)\, r_3(B)\, u_3(B)$

Well-formed because every $p_i(x)$ is between $l_i(x)$ and $u_i(x)$

Not legal because critical sections are overlapped (red and green)

A scheduler based on locks does:

- when an action request is issued by a transaction, checks if this makes the transaction ill-formed (not well-formed): if yes abort
- in case of possible "not legality" it blocks the request until resources is unlocked
- keeps tracks of locks in the lock table.

eg

T1	T2
$l_1(A)$	
$r_1(A)$, A++	$l_2(A)$ note → blocked
$w_1(A)$	
$u_1(A)$	$l_2(A)$ ok
	⋮

Anyway this is not sufficient to grant serializability.
To achieve it we add a protocol called TWO PHASE LOCKING (2PL): a schedule S with exclusive locks follows the 2PL protocol if in each transaction T_i appearing in S, all locks precede all unlocks

$S: \dots l_i(A) \dots u_i(A) \dots$
no unlock for T_i — no lock for T_i

The scheduler manages to issue all the unlocks after all locks and check if the schedule is still legal.
Note: may happen that to ensure that the schedule is legal, the scheduler blocks both transaction causing DEADLOCK

Note: so far we said that transactions issue lock/unlock commands but we can configure the scheduler to do it automatically. For this reason when talking about a sequence we can "omit" locks/unlocks eg $l_1(A)\, r_1(A)\, l_1(B)\, u_1(A)\, l_2(A)\, w_2(A) \dots = r_1(A)\, w_2(A) \dots$

How exclusive locks and 2PL scheduler works:
- T_i aborted if its request shows that T_i is not well-formed
- T_i aborted if its lock request doesn't follow 2PL
- T_i paused if requests a lock on already locked resource. Deadlock management if needed.
- Builds lock table

We can compose 2PL with conflict-serializability noting that every schedule with lock/unlock operation can be seen as a "traditional" schedule

Th. Every legal schedule constituted by well-formed transactions following the 2PL protocol (with exclusive locks) is conflict-serializable.
Proof: S composed by N well formed transactions following 2PL
- Base step: $N=1 \rightarrow$ S is serial and trivially conflict-serializable.
- Inductive step: $N>1$. Suppose T_i is the first transaction that executes unlock $u_i(x)$. We now show that we can move all operations of T_i in front of S without swapping any pair of conflicting actions. Consider $w_i(y)$, it can't be preceded by any conflicting actions. Indeed suppose there is a conflicting action $w_j(y)$ that precedes it: $\dots w_j(y) \dots u_j(y) \dots l_i(y) \dots w_i(y)$

But we said that T_i is the first to unlock so:
- $u_i(x) \dots w_j(y) \dots u_j(y) \dots l_i(y) \dots w_i(y)$
- $\dots w_j(y) \dots u_i(x) \dots u_j(y) \dots l_i(y) \dots w_i(y)$

→ Both not follow 2PL

$\Rightarrow S'' =$ [actions of T_i][S'] = [actions of T_i][the remaining actions] = conflict equivalent to S
$\Rightarrow S' =$ N-1 transactions well-formed and follow 2PL = conflict-serializable
$S''' =$ serial schedule on N-1 transactions conflict equivalent to S'
$\Rightarrow$ [T_i][S'''] = conflict equivalent to S $\rightarrow$ S conflict-serializable

ex. $S1: r_1(x)\ w_2(x)\ w_3(y)\ w_1(z)\ w_3(z)$ is this a 2PL schedule?

$l_1(x)\ r_1(x)\ l_1(z)\ u_1(x)\ l_2(x)\ w_2(x)\ u_2(x)\ l_3(y)\ w_3(y)\ w_1(z)\ u_1(z)\ l_3(z)\ w_3(z)\ u_3(y)\ u_3(z)$

Yes it is!

ex $S2: r_1(x)\ w_2(x)\ w_3(y)\ w_3(z)\ w_1(z)$

No because T_1 need to lock z before unlock x and this block T_3 on z

$l_1(x)\ r_1(x)\ l_3(y)\ w_3(y)\ l_3(z)\ w_3(z)\ u_3(z)\ l_1(z)\ w_1(z)\ u_1(z)\ u_1(x)\ l_2(x)\ w_2(x)\ u_2(x)\ u_3(y)$

(T_2 waits; T_2 continues)

↳ $S2': r_1(x)\ w_3(y)\ w_3(z)\ w_1(z)\ w_1(z)\ w_2(x) != S2$

Note: no order needed (if fine with protocol anyway)

Th. there exists a conflict-serializable schedule not following 2PL (with exclusive locks)

SHARED LOCK

Particular lock that grants lock from multiple transaction only for reading

eg $sl_i(A)\ r_i(A)\ sl_j(A)\ r_j(A)\ldots$

- Well-formed:
  - every $r_i(A)$ is preceded by $sl_i(A)$ or $xl_i(A)$ with no $u_i(A)$ between
  - every $w_i(A)$ is preceded by $xl_i(A)$ without $u_i(A)$ between
  - every lock (x or s) on A by T_i is followed by $u_i(A)$

Note: $sl_i(A)$ followed by $xl_i(A)$ is called lock upgrade (from shared to exclusive lock)

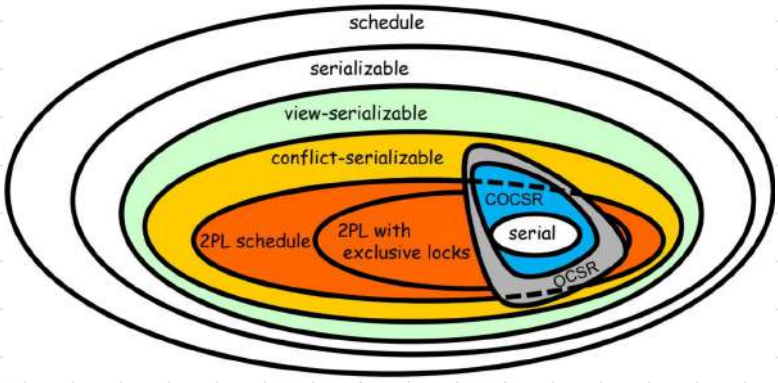
- Schedule is legal
  - a $xl_i(A)$ isn't followed by any $xl_j(A)$ or any $sl_j(A)$ without $u_i(A)$ between
  - a $sl_i(A)$ isn't followed by any $xl_j(A)$ without $u_i(A)$ between
- 2PL: in every transaction of S all locks are before unlocks

Lock already granted to Ti on A \ New lock requested by Tj ≠ Ti on A	S	X
S	yes	no
X	no	no

Now scheduler job is more complex → more waiting transactions. Most common execution is "FIFO" to avoid starvation.

Th. Every legal schedule with well-formed transactions following 2PL is conflict-serializable

Th. Exists a conflict-serializable schedule not following 2PL.



DEADLOCK MANAGEMENT

The probability of deadlock grows linearly with the number of transactions and quadratically with the number of lock requests in each of them.
To manage it there are multiple way:

① Timeout

System gives a timeout t to the transactions, after that it kill the transactions still waiting for the lock. Easy but problems if t too high or too low. Also risk of individual block (some transaction killed several times)

② Deadlock Recognition

A "wait-for" graph is monitored: edge $T_i \to T_j$ means T_i is waiting T_j. If there is a cycle, the deadlock is solved killing a transaction in it (maybe the one with fewer operations)

③ Deadlock Prevention (wait-die)

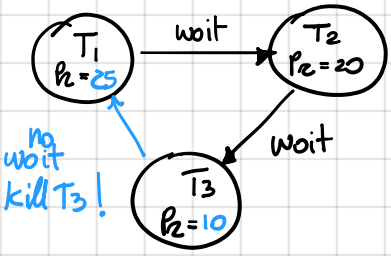
Every transaction has a priority. In case of conflict on lock T_i is allowed to wait T_j only if $pr(T_i) > pr(T_j)$, otherwise T_i is killed

In "wait-for" graph, $T_i \to T_j$:

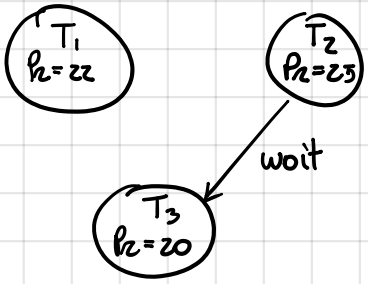
- $Pr(T_i) > pr(T_j) \Rightarrow$ OK
- $Pr(T_i) \leq Pr(T_j) \Rightarrow$ kill T_i

eg $xl_1(y)\ xl_3(x)\ xl_2(x)\ xl_1(x)\ xl_3(y)$ ⟶

T_1 uses y
T_3 uses x
T_2 waits T_3
T_1 waits T_2
T_3 killed



eg $xl_3(x)\ xl_2(x)\ xl_1(x)$



$Pr(T_1) > Pr(T_3)$ but $Pr(T_1) < Pr(T_2)$
So T_1 waits T_3 → Two options:
① T_1 before T_2 ⇒ risk of starvation of T_2
② T_2 before T_1 ⇒ violates the rule of priority
⇒ kill T_1

Note: often on exams the wait-for graph doesn't use priority but only takes into account the edges.

Ex. $r_1(A)\, w_1(C)\, r_2(B)\, w_1(B)\, w_2(C)\, r_3(C)\, w_1(D)\, w_2(D) = S$

Describe behaviour of 2PL scheduler

$sl_1(A)\, xl_1(C)\, r_1(A)\, w_1(C)\, sl_2(B)\, r_2(B)$ $xl_2(C)$ → T_1 waits for B, T_2 waits for C ⇒ DEADLOCK

(↑ NOT ALLOWED)

Possible solution: KILL T_2 → ... $w_1(C)\, xl_1(B)\, w_1(B)\, xl_1(D)\, u_1(C)\, sl_3(C)\, r_3(C)\, w_1(D)$

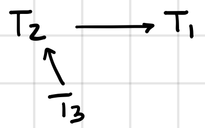
⇒ $S' : r_1(A)\, w_1(C)\, w_1(B)\, r_3(C)\, w_1(D) \;!= S \rightarrow S$ not in 2PL

① S suffers of deadlock ⇒ $S \notin 2PL$

② S doesn't suffer of deadlock $\overset{?}{\Rightarrow}$ $S \in 2PL$ NO!

Ex $S: r_1(A)\, r_2(A)\, r_3(B)\, w_1(A)\, r_2(C)\, r_2(B)\, w_2(B)\, w_1(C)$

- Is view-serializable? If yes provide a view-equivalent serial schedule
  yes $T_3\, T_2\, T_1$
- Is conflic-serializable? If yes provide all conflict-equivalent serial schedules (imply)



Yes $T_3\, T_2\, T_1$

(imply both up (not this time ofc))

- Is 2PL schedule (only exclusive lock)?
  No because to issue $r_2(A)$ T_1 has to unlock A but then T_1 need to lock again A for $w_1(A)$
- Is 2PL schedule (shared and exclusive)
  yes:
  $sl_1(A)\, sl_2(A)\, sl_3(B)\, r_1(A)\, r_2(A)\, r_3(B)\, u_3(B)\, sl_2(C)\, sl_2(B)\, xl_2(B)\, u_2(A)\, w_1(A)\, r_2(C)\, r_2(B)\, w_2(B)$
  $u_2(C)\, u_2(B)\, xl_1(C)\, w_1(C)\, u_1(A)\, u_1(C)$

RECOVERABILITY OF TRANSACTIONS

With the possibility of "rollbacking" transactions, a new anomaly comes up: dirty read.

Dirty read occurs when T_2 read from w_1 and rollbacks: T_2 now has read a value (from T_1) not valid

Note: commit = system should ensure that all actions are permanent

rollback = system should ensure actions have no effect

Rollback of T_i may cause rollback of T_J that may cause rollback of T_x and so on: this is called coscading rollback

For this reason we talk about recoverable (or not) schedules: S is recoverable if no transaction in it commits before all other it has "read from" commit ie. if no rollback cause problems.

Serializable ⇏ Recoverable and Recoverable ⇏ Serializable but serials are both.

Even if recoverable, a schedule may still suffer of coscading rollback (eg many transactions need to be killed). S avoid coscade rollback if every T_i read values written by already commited T_J $(i \neq J)$. [ACR]

Serial ⇒ ACR ⇒ Recoverable

T_i writes on T_J if both write on same element and w_i comes after w_J without any other write on that element between.

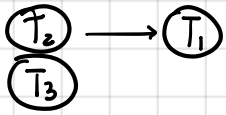
A schedule is strict if ACR and every T_i "writes only on" T_J already committed.

A schedule is rigorous if $\forall \langle a_i, b_J \rangle$ pair of conflicting actions the commit c_i is between a_i and b_J

Rigorous → Strict

Ex $S: r_2(A)\, r_3(B)\, w_1(A)\, r_2(C)\, r_2(D)\, w_1(D)$

- Is view-serializable? If yes provide a view-equivalent serial schedule
  Yes: $T_2\ T_3\ T_1$ and $T_3\ T_2\ T_1$
- What's the precedence graph of S? Is it conflict-serializable? If yes provide all conflict-equivalent serial schedule

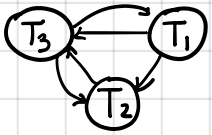


Yes it is!
$\Rightarrow T_3\ T_2\ T_1$ or $T_2\ T_3\ T_1$

- Is it 2PL (with exclusive locks)?
  $l_2(A)\, r_2(A)\, l_3(B)\, u_2(A)\, l_1(A)\, l_2(C)$ Not 2PL

Ex $S: r_3(B)\ w_1(A)\ w_3(B)\ r_1(B)\ r_2(A)\ w_3(A)\ w_2(A)$

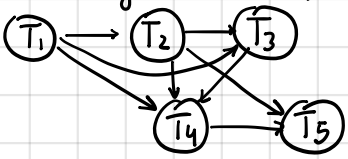
- Is view-serializable? If yes provide a view-equivalent serial schedule
  Yes $T_3\ T_1\ T_2$
- Whats the precedence graph? Is it conflict-serializable?



Not conflict serializable

Ex $S: r_1(x)\ w_2(x)\ r_3(x)\ r_1(y)\ r_4(z)\ w_2(y)\ r_1(v)\ w_3(v)\ r_4(v)\ w_4(y)\ w_5(y)\ w_5(z)$

- Is it view-serializable?
  $W_2(x) \rightarrow r_3(x)$
  $W_3(v) \rightarrow r_4(v)$
  $\rightarrow T_1\ T_2\ T_3\ T_4\ T_5$. Its view-serializable
  Note T_1 is set first cause it reads from default value (no writes before it)
- Is it conflict-serializable?



Conflict-serializable

- Is it 2PL with both lock types
  $sl_1(x)\, r_1(x)\ sl_1(v)\, u_1(x)\, xl_2(x)\, w_2(x)\, xl_2(y)\, u_2(x)\, sl_3(x)\, r_3(x)\ u_2(y)\ sl_1(y)\ u_1(y)\ xl_2(y)$!!!

Ex $S: r_1(x)\ r_2(z)\ r_1(z)\ r_3(x)\ r_3(y)\ w_1(x)\, w_3(y)\, r_2(y)\, w_4(z)\, w_2(y)$

- Is it 2PL with both lock types?
  $sl_1(x)\, sl_2(z)\, sl_1(z)\, sl_3(x)\, sl_3(y)\, xl_3(y)\ r_1(x)\, r_2(z)\, r_1(z)\, r_3(x)\, r_3(y)\ u_3(x)\ w_1(x)\ w_3(y)\, u_3(y)\, sl_2(y)\, u_1(z)$
  $xl_2(y)\, u_2(z)\, xl_4(z)\, w_4(z)\, u_4(z)\, w_2(y)\ u_2(y)\, u_1(x)$
  Yes it 2PL

S: a_i c_i b_j c_j [rigorous]

S': serial schedule coherent with the order of commits i.e if first commit is c_1 then first transaction is T_i and so on

Prove S' is conflict equivalent to S.

Suppose S' has a_i b_j in reverse order : b_j a_i $\Rightarrow$ $T_j \dots T_i$ $\Rightarrow$ impossible

$\Rightarrow$ S' is conflict-equivalent to S $\Rightarrow$ S conflict-serializable

What about 2PL with these new notions?

A schedule is strict 2PL if 2PL and strict (you dont say?): follow 2PL and all exclusive locks of every transaction T are kept by T until T commits or rollback

Strict 2PL $\Rightarrow$ serializable

Strong Strict 2PL : same but on all types of lock. $\Rightarrow$ Rigorous.

Ex. Given a schedule $S: \{T_1 \dots T_N\}$, the "strong graph" is a graph that has T_i as nodes and egdes from T_i to T_j for each pair of actions $\langle a_i, a_j \rangle$ on some element X and a_i appears before a_j

Prove:

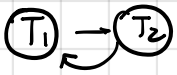
- If the strong graph associated to S is acyclic then S is conflict-serializable?

Yes because precedence graph is a subgraph of strong graph

Strong graph acyclic $\Rightarrow$ Precedence graph acyclic $\Rightarrow$ conflict-serializable

- If S is conflict serializable then the strong graph is acyclic?

No, counter example

$r_1(A)$ $r_2(A)$ $w_2(B)$ $r_1(B)$ 

CONCURRENCY WITH TIMESTAMP

Every transaction has an unique timestamp : $ts(T_i) < ts(T_j)$ if T_i arrives first to the scheduler
Every schedule respecting timestamp order is conflict serializable because conflict equivalent to serial schedule corresponding to timestamp order.

$rts(x)$: the highest timestamp of transaction that have read x
$wts(x)$: timestamp of last transaction who wrote x
$wts\text{-}c(x)$: timestamp of last committed transaction that wrote x
$cb(x)$: true if last transaction who wrote x committed, false otherwise.

Basic idea :
- actions of T in S must be considered as being logically executed in one spot
- logical time of actions in T is the timestamp of T $ts(T)$
- commit bit used to avoid dirty-read anomaly

System manages physical and logical time : $rts(x)$ and $wts(x)$ are timestamps of transaction who read and wrote last x according to logical time. T executed at physical time t is accepted if its ordering in physical time is compatible with logical time $ts(T)$ (done by scheduler)

Read ok : $B(T_1)$ t_1 $B(T_2)$ t_2 $B(T_3)$ t_3 $w_1(x)$ t_4 $r_3(x)$ t_5 $\boxed{r_2(x)\ t_6}$

- Physical time of $r_2(x)$ is t_6 > physical time of w_1 (t_4)
- Logical time of $r_2(x)$ is $ts(T_2)$ > logical time of $w_1(x)$ = $wts(x) = ts(T_1)$

→ No incompatibility

Then :
if $cb(x)$ = true
- after read of x from T $rts(x) = \max\{rts(x), ts(T)\}$
- $r_2(x)$ executed and schedule procedes

if cb = false (like in the example)
- T_2 is put in waiting for commit or rollback of transaction T_1 that was the last to write

Read too late : $B(T_1)$ t_1 $B(T_2)$ t_2 $w_2(x)$ t_3 $\boxed{r_1(x)\ t_4}$

- Physical time of $r_1(x) = t_4$ > physical time of $w_2(x) = t_3$
- Logical time of $r_1(x) = ts(T_1)$ < logical time of $w_2(x) = wts(x) = ts(T_2)$

→ $r_1(x)$ and $w_2(x)$ are incompatible

⇒ T_1 rollbacks and a new execution of it starts with new timestamp

Write ok : $B(T_1)$ t_1 $B(T_2)$ t_2 $B(T_3)$ t_3 $r_1(x)$ t_4 $w_2(x)$ t_5 $\boxed{w_3(x)\ t_6}$

- Physical time of $w_3(x) = t_6$ > physical time of $r_1(x) = t_4$ and $w_2(x)$ (last write) $= t_5$
- Logical time of $w_3(x) = ts(T_3)$ > logical time of $r_1(x) = ts(T_1)$ and $w_2(x) = ts(T_2)$

→ no incompatibility

Then :
if $cb(x)$ = true or no active transaction wrote on x
- $wts(x) = ts(T_3)$
- $w_3(x)$ of T_3 executed and schedule procedes.
- $cb(x)$ = false

if cb(x) = false
T_3 is put in waiting for commit or rollback of last writer of x (T_2)

Thomas rule: $B(T_1)_{t_1}$ $B(T_2)_{t_2}$ $B(T_3)_{t_3}$ $r_1(x)_{t_4}$ $w_2(x)_{t_5}$ ... $w_1(x)_{t_6}$

- Physical time of $w_1(x) = t_6$ > physical time of $r_1(x) = t_4$ and some logical time because of some transaction
- $w_2(x)$ comes after $w_1(x)$ in logical time ⇒ execution of $w_1(x)$ = update loss

Then:
if cb(x) = true
Just ignore $w_1(x)$ and overwrite it with $w_2(x)$
if cb(x) = false
T_1 is put in waiting for commit or rollback of last writer of x

Write too late: $B(T_1)_{t_1}$ $B(T_2)_{t_2}$ $r_2(x)_{t_3}$ $w_1(x)_{t_4}$

- Physical time of $w_1(x) = t_4$ > physical time of $r_2(x) = t_3$] incompatible
- logical time of $w_1(x) = t_3(T_1)$ < logical time of $r_2(x) = t_3(T_2)$]

T_1 aborted and re-executed with new timestamp

Action ri(X):

```
if      ts(Ti) >= wts(X)
then    if cb(X)=true or ts(Ti) = wts(X)                          // (case 1.a)
        then set rts(X) = max(ts(Ti), rts(X)) and execute ri(X)   // (case 1.a.1)
        else put Ti in "waiting" for the commit or the
             rollback of the last transaction that wrote X        // (case 1.a.2)
else    rollback(Ti)                                              // (case 1.b)
```

Action wi(X):

```
if      ts(Ti) >= rts(X) and ts(Ti) >= wts(X)
then    if cb(X) = true
        then set wts(X) = ts(Ti), cb(X) = false, and execute wi(X)  // (case 2.a.1)
        else put Ti in "waiting" for the commit or the
             rollback of the last transaction that wrote X          // (case 2.a.2)
else    if ts(Ti) >= rts(X) and ts(Ti) < wts(X)                     // (case 2.b)
        then if cb(X)=true
             then ignore wi(X)                                      // (case 2.b.1)
             else put Ti in "waiting" for the commit or the
                  rollback of the last transaction that wrote X     // (case 2.b.2)
        else rollback(Ti)                                           // (case 2.c)
```

When Ti executes ci:

```
for each  element X written by Ti,
          set cb(X) = true
          for each  transaction Tj waiting for cb(X)=true or for the
                    rollback of the transaction that was the last to
                    write X, allow Tj to proceed
choose the transaction that proceeds
```

When Ti executes the rollback bi:

```
for each  element X written by Ti, set wts(X) to be wts-c(X), i.e., the
          timestamp of the transaction Tj that wrote X before Ti, and set
          cb(X) to true (indeed, Tj has surely committed)
          for each  transaction Tj waiting for cb(X)=true or for the
                    rollback of the transaction that was the last to
                    write X allow Tj to proceed
choose the transaction that proceeds
```

Timestamp reduce risk of deadlock but dont avoid it
Deadlock is related to commit-bit
eg. $w_1(B)\ w_2(A)\ w_1(A)\ r_2(B)$
On $w_1(A)$ T_1 waits for rollback of T_2 and viceversa T_2 waits T_1 on $r_2(B)$

Some conflict-serializable schedules are not accepted by timestamp method eg $r_1(y)\ r_2(x)\ w_1(x)$:
if we dont use the Thomas rule then the schedule obtained removing all actions of rollbacked transactions is conflict-serializable
Otherwise it may be not eg $r_1(A)\ w_2(A)\ c_2\ w_1(A)\ c_1$. However if the schedule S is processed by timestamp method with Thomas rule then the schedule obtained from S by removing all actions ignored by the Thomas rule and all action of rollbacked transactions is conflict-serializable.

In the same way schedules that are accepted by timestamp scheduler may be 2PL or not.

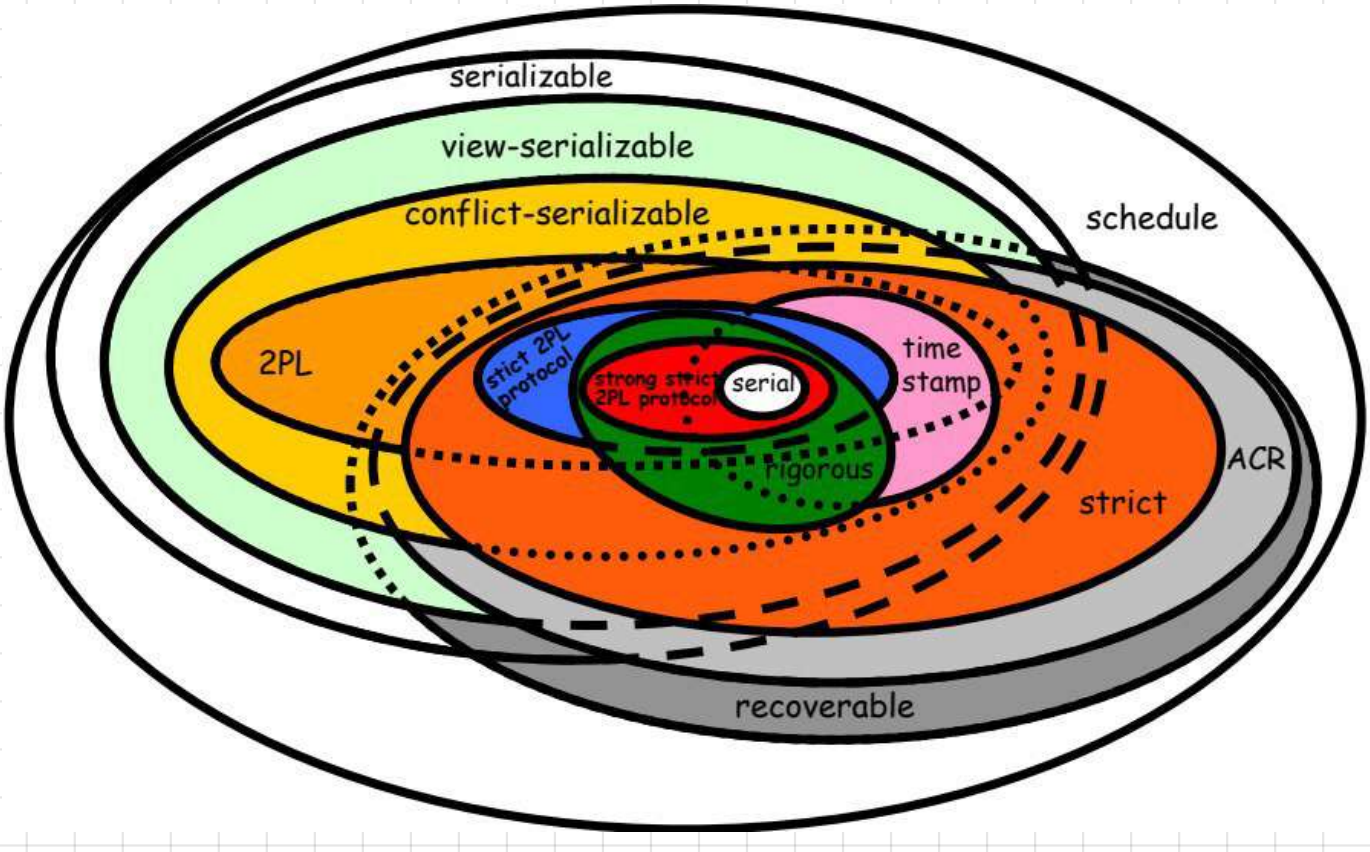
COMPARISON WITH 2PL

Waiting Stage:
- 2PL: just wait
- TS: transactions reading or writing too late are killed and re-started. Waiting only for commit or rollbacks

Serialization Order:
- 2PL: determined by conflicts
- TS: determined by timestamps

Need to wait for commit:
- 2PL: solved by strong strict 2PL
- TS: buffering of write actions (wait cb(x) = true)

Deadlock:
- 2PL: risk of deadlock
- TS: less probable



Prove/disprove that if S creates a deadlock when processed by a 2PL scheduler then s is not conflict-serializable.

→ Deadlock cause cycle in "wait-for" graph

→ Wait-for graph = reversed precedence graph

→ precedence graph has cycle too! → not conflic-serializable ⇒ Proven

Ex $rts(A)=rts(B)=wts(A)=wts(B)=wts\text{-}c(A)=wts\text{-}c(B)=0$ and $cb(A)=cb(B)=true$.

Suppose system clock is 0 and system uses clock for timestamping. Illustrate actions of timestamp scheduler of $r_1(A)\ r_2(B)\ r_3(A)\ r_2(A)\ w_1(A)\ w_3(A)$ and tell if the resulting sequence is strong 2PL (both locks)

- $r_1(A)$ read ok because $ts(T_1)=1 \geq wts(A)=0$ and $cb(A)=true$ → $rts(A)=1$
- $r_2(B)$ read ok because $ts(T_2)=2 \geq wts(B)=0$ and $cb(B)=true$ → $rts(B)=2$
- $r_3(A)$ read ok because $ts(T_3)=3 \geq wts(A)=0$ and $cb(A)=true$ → $rts(A)=3$
- $r_2(A)$ read ok because $ts(T_2)=2 \geq wts(A)=0$ and $cb(A)=true$ → $rts(A)=3$ no change (takes the max)
- $w_1(A)$ write too late because $ts(T_1)=1 < rts(A)=3$ → w_1 rollback
- $w_3(A)$ write ok because $ts(T_3)=3 \geq rts(A)=3$ and $ts(T_3) \geq wts(A)=0$ → $wts(A)=3$ $cb(A)=false$

⇒ $r_2(B)\ r_3(A)\ r_2(A)\ w_3(A)$ → Strong Strict 2PL

Ex Same as before: $r_1(B)\ w_1(A)\ w_2(B)\ w_1(B)\ r_2(A)$

Note $ts(T_1)=1$ while $ts(T_2)=3$ cause clock is equal to 3 when T_2 starts

- $r_1(B)$ read ok because $ts(T_1) \geq wts(B)$ and $cb(B)=true$ → $rts(B)=1$
- $w_1(A)$ write ok because $ts(T_1) \geq wts(A)$, $cb(A)=true$, $ts(T_1) \geq rts(A)$ → $cb(A)=false$ $wts(A)=1$
- $w_2(B)$ write ok because $ts(T_2) \geq wts(B)$, $cb(B)=true$ and $ts(T_2) \geq rts(B)=1$ → $wts(B)=3$ $cb(B)=false$
- $w_1(B)$ waiting for T_2 commit because of Thomas rule (case 2): $cb(B)=false$, $ts(T_1)=rts(B)$ and $ts(T_1)<wts(B)$
- $w_2(A)$ waiting for T_1 commit because $cb(A)=false$ and $ts(T_2)>wts(A)$

⇒ DEADLOCK

MULTI-VERSION CONCURRENCY CONTROL

$S: w_0(x)\ w_0(y)\ r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_0\ c_1\ c_2$ not conflict serializable

but we can fix this if $r_1(y)$ could read the old version y_0 of y to be coherent with $r_1(x)$ that read x_0
→ S equivalent to $S': T_0\ T_1\ T_2$

How?

- each write creates new version
- each read can choose on which version to read
- versions are transparent to application (garbage collection)

Multiversion Timestamp: don't block reads but do them on right version.

- Every "legal" write generates a new version of element : $w_i(x) \rightarrow x_i$
- $wts(x_n) = ts(T_n)$ = last write
- $rts(x_i) = ts(T_i)$ = last read

NEW RULES

- $w_i(x)$:
  - if $r_j(x_k)$ s.t. $wts(x_k) < ts(T_i) < ts(T_j)$ already occurred → $w_i(x)$ refused (write too late)
  - otherwise → write on new version and $wts(x_i) = ts(T_i)$
- $r_i(x)$:
  executed on version x_j with $wts(x_j)$ being the last write on x (of course $\leq ts(T_i)$). $rts(x_j)$ updated as usual
- Every version with no active transactions are deleted
- To ensure recoverability, commit of T_i is delayed until all writing transactions that read by T_i commit
- For each version, scheduler maintains $range(x_i) = [wts, rts]$ from $wts(x_i)$ to last read of x_i
  $Ranges(x) = \{ range(x_i) \mid x_i \text{ is a version of } x \}$

eg. current version of $A = A_0$ and $rts(A) = 0$

T1(ts=1)	T2(ts=2)	T3(ts=3)	T4(ts=4)	T5(ts=5)	
r1(A)					reads A0, and set rts(A0)=1
w1(A)					writes the new version A1
	r2(A)				reads A1, and set rts(A1)=2
	w2(A)				writes the new version A2
			r4(A)		reads A2, and set rts(A2)=4
				r5(A)	reads A2, and set rts(A2)=5
		w3(A)			rollback T3

TRANSACTIONS IN SQL

SQL works with sessions (simply the tab, multiple tabs - multiple sessions)
A transaction is defined as simple sql command (insert, select etc) or as a sequence of them if between BEGIN and COMMIT/END

Anomalies

- Dirty Read: transactio reads element written by non committed one
- NonRepeatable Read: transaction read twice some element
- Phantom Read: transaction T_1 execute "range" query (eg "where age < 40 and age > 20), then T_2 insert a new tuple satisfying this range and then T_1 re-execute some query with new results (can be avoided with locks)

Isolation Levels

Every transaction can decides its own level of isolation to avoid different anomalies

Isolation Level	*Dirty Read*	*Nonrepeatable Read*	*Phantom Read*
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

RECOVERY MANAGEMENT

How data manager acts for example when a transaction is aborted?
Recovery Manager deals with atomicity and persistency (durability) and is responsable for beginning, committing, rollbacting transactions and restoring correct state of DB after a fault.
It uses log files, records of actions in stable storage (failure resistant, abstraction)
Failures:
- System failures: system crash, system error or application exception (division by zero), local error conditions of transaction, concurrency controll
- Storage media failures: disk failures and catastrophic events

Fail-Stop Model: when a fail occurs the system is "stopped". Then it "boot" and restart. If we have a fail again then it is "stopped" again otherwise it goes to normal state

LOG FILE

Log file is a sequential file, in chronological order.
Inside of it we have two types of records: transaction record and system record
Note: actions of transactions are assumed executed when recorded in the log (even if not already applied on secondary storage (DB).

Transaction Record:
O = element of DB
AS = after state, value of O after the operation
BS = before state
eg. Transaction T:
- begin: B(T)
- insert: I(T, O, AS)
- delete: D(T, O, BS)
- update: U(T, O, BS, AS)
- commit: C(T)
- abort A(T)

System Records
Checkpoint: register in log active transactions
- for every transaction committed after last checkpoint their buffer pages are copied in secondary storage (flush) → No "redo" needed if fail!
- A record CK(T_1...T_N) is written in log (force) with T_1... T_N active transactions uncommitted

Executed periodically
Dump: copy of entire DB state (backup in stable storage)
Done offline (transactions suspended) and writes a dump record in log (force)

- undo: restore state of element O at time preceding the execution of an action
  eg update/delete → assigns BS to O
  insert → delete O
- redo: "repeat" the operation (contrary of undo)
  eg insert/update → assigns AS to O
  delete → delete O

ATOMICITY

Outcome of transactions is established when either C(T) or A(T) is written in log.
C(T) is written synchronously from buffer to log, A(T) instead asynchronously.
When fail occurs:
- uncommitted transaction: to ensure atomicity, we undo actions.
- committed transaction: to ensure durability, we redo actions.

Recovery manager follows these rules

WAL (write-ahead log):

log records are written in log before data records are written in secondary storage. (log before DB).
This to allow to undo operation to always write old value to secondary storage from uncommitted transactions.

COMMIT-PRECEDENCE:

Log records are written before commit of transaction. This to allow the redo operation.

For each operation, the manager must decide a strategy for writing in DB. (U, I, D):

Immediate effect:

operations executed in S.S immediately after log record. Buffer manager writes effect of actions of T on S.S. before C(T) is written in log. → all pages of DB modified by a transaction are written in S.S. for sure → REDO not needed.

Delayed effect:

operations executed in S.S. after commit of transaction (only after C(T) in log) → UNDO not needed
Note: records are written in log before S.S. anyway.

Mixed effect:

for operation O, both previous methods are possible (manager decides)

In case of system failure ...

WARM RESTART

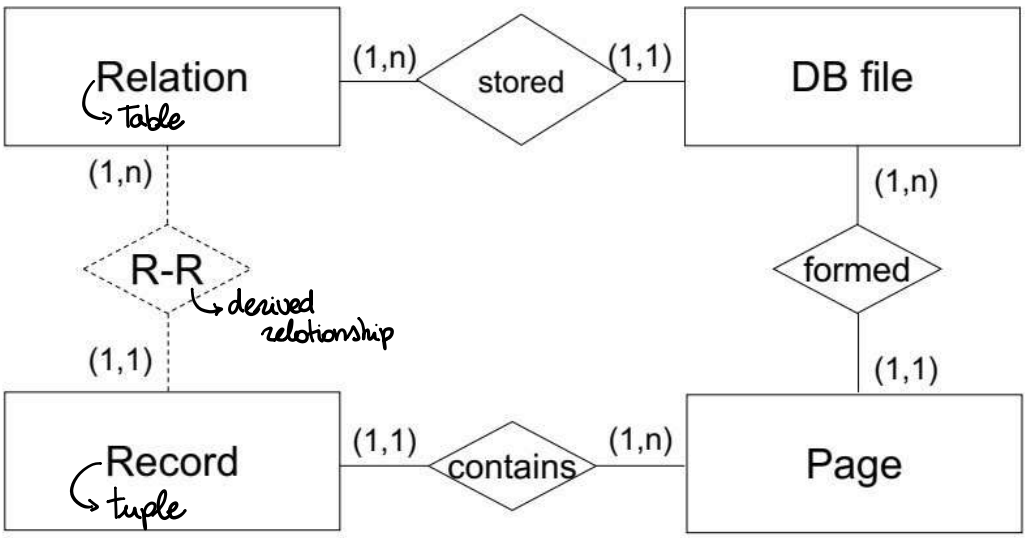
Assume mixed effect strategy, 5 steps.
1) Go backward in log until most recent checkpoint
2) Set S(UNDO) = {active transactions at checkpoint} and S{REDO} = {}
3) Scroll log adding to S(UNDO) the transactions with the corresponding begin record, and to S(REDO) those with commit record
4) Undo Phase: scroll backward and undo transactions in S(UNDO) until the oldest begin of them (even before the checkpoint)
5) Redo Phase: scroll forward redoing transactions in S(REDO)

In case of disk failure:

COLD RESTART

1) Search most recent dump in log and load it in S.S. (in some way we dont care how)
2) Forward recovery of dump state
   - re-apply all actions in log in same order of log
   - At m we have the DB state before crash
3) Execute warm restart

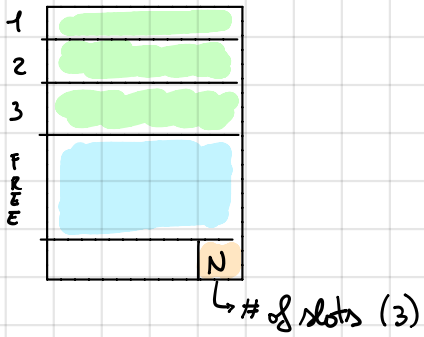
FILE ORGANIZATION

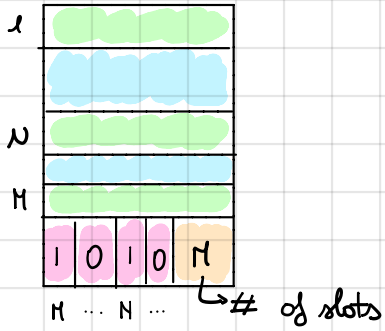


DB file ≠ file
= set of pages

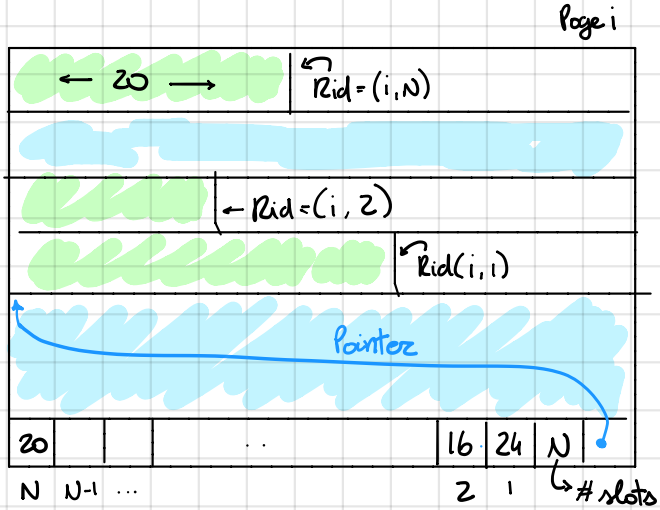
PAGES AND RECORDS

Page dimension is BLOCK and has an address (page id), constituted by slots each of them has an address. + header
Slot = memory space for a record with an identifier in the context of page
Each record has <page id, slot id> = record id. + header.
Pages with fixed length records can be packed or unpacked according to the fact that slots are occupied one by one without free space between them or not. In second case we have to keep track of free or empty slots.





Page with variable length records:



What if a record doesn't fit in a page? Split in fragments to let them fit. Splitted records are called spanned and need extra info: bit to know if it is a fragment and if yes first or second and pointer to other part.

SIMPLE FILE ORGANIZATION

File is a collection of pages each one containing collection of records. Usually a file is used for a single relation (table) but not always.

Heap File: no order or special criterion to store records. Pages allocated and de-allocated when needed. Need to keep track of pages, record and free space. Heap file can be represented through lists: list of full or with free space pages linked together (linked list). Two different list for full and with free space pages.
Other way is through directory: list of pages each containing pointer to real pages (like folders for pages). More efficient searching for free space.

Sorted File: records are sorted within each page and also pages are sorted. Pages are in a sort of array.

Hashed File: pages organized in groups called bucket. A bucket has a primary page and other overflow pages linked to it. Primary pages are sequentially ordered and works as a hash map: search $K \rightarrow h(K) \bmod N \rightarrow$ address of bucket containing K (primary page pointer)

Complexity: B = pages in file R = records in page D = time for write/read C = average time for processing a record (later we will be interested only in pages access)

Heap:

- SCAN: $B(D+RC)$ → for each B, load it (D) and for each record (R) process it (C)
- EQUALITY SELECTION: $B(D+RC)$ → same as scan but if element is unique then average cost will be $(D+RC)\frac{B}{2}$
- RANGE SELECTION: $B(D+RC)$
- INSERTION: $D+C+D$ → load last page, insert, write page
- DELETION: $D+C+D$ if record identified by rid, $B(D+RC)+XC+YD$ if specified by range with X = # of records to be deleted and Y = # of pages with records to be deleted. (binary search)

Sorted (search based on key)

Suppose data are stored in pages ordered from a_1 to a_B. A basic algorith to search key K in range (h_1, h_2) is:
- if (h_1, h_2) is empty → stop with failure
- choose page address $h_1 \le i \le h_2$ and load page p_i
- if K in p_i → stop with success
- if $K <$ min value of p_i → repeat on $(h_1, i-1)$ else → repeat on $(i+1, h_2)$

Above algorithm can be changed in
binary search: i is the one in the middle of range
interpolation search: if values are numeric and uniformly distributed in (h_{min}, h_{max}) then P_K = probability that record of K has value less than or equal to $K = (K - K_{min})/(K_{max} - K_{min})$
⇒ assuming distance between addresses is analogous to distance between key values → $i = a_1 + P_K \times (a_B - a_1)$

- SCAN: $B(D+RC)$
- EQUALITY SELECTION
  - binary search: $1 + D\log_2 B + C\log_2 R$ (worst case)
  - interpolation search: $D(\log_2 \log_2 B) + C\log_2 R$ (avg. case)
    $B(D+RC)$ (worst case)
- RANGE SELECTION (or equality search on NON-KEY): $D\log_2 B + C\log_2 R + (f_s \times B - 1)(D+RC)$
- INSERTION: $1 + D\log_2 B + C\log_2 R + C + 2B(D+RC)$ → remove on avg case if insert in the middle of file
- DELETION: Similar to insertion.

Hashed:

- SCAN: $1.25\ B(D+RC)$
- EQUALITY SELECTION: $(D+RC) \times$ (number depending on overflows)
- RANGE SELECTION: $1.25\ B(D+RC)$
- INSERTION: $2D + RC$
- DELETION: cost of search $+ D + RC$

Note: sorting is not important only for sorted file eg sorted results of a query.
It is important for eliminating duplicates or for join algorithm (merge sort)
Other sorting algorithm make use of a secondary storage not possible in practice → internal sorting (main memory)
Merge sort requires $O(N \log_2 N)$ comparisons with N = # records
To use external sorting we can work at page level (not records)

2-WAY SORT

- Sort internally each page
- Merge two pages into one run (portion of file sorted) → Repeat for all — PASS 0
- Merge two runs (of two pages) into one run → Repeat for all — PASS 1
- Repeat untill we have a single run — PASS 2

⇒ Total cost $2 \times B \times (\lceil \log_2 B \rceil + 1)$ (# steps = $\lceil \log_2 B \rceil + 1$)
⇒ better than merge sort because $N \gg B$ so $O(N \log_2 B) >> O(B \log_2 B)$

2way merge sort uses 3 frames in the buffer: 2 to hold input records, 1 to hold output
but we have lot more buffer frames ⇒ each pass, read as much data as possible into buffer reducing passes and cost → $2 \times B \times$ # passes

K-WAY / MULTIPASS MERGE SORT

- Reserve F input frames and 1 output frame
- PASS 0: read file in runs of F pages and, at each run, internally sort them and write a file (called run) to disk
- PASS 1: merge F runs into one
- PASS 2: merge F runs into one

...

- SORTED!

Suppose we have F frames free to be used for sorting relation R
1) (Pass 0) repeat untill no more pages in R:
   bring F pages of R in buffer, sort their records and write corresponding pages in a file (run)
2) repeat until we have just one run
   (Pass i) repeat until we dont have runs to consider
   (merge) merge records of first $2 < F$ runs not yet analyzed into a single new run: done by reading the pages of 2 runs, using one frame for each run and writing the result in the output file (new run), one page at a time using one frame in the buffer.

Cost: $2 \times B \times (\log_2 S + 1) = 2 \times B \times (\log_2 (B/F) + 1) \longrightarrow 2 \times B \times \log_F B$ (if $2 = F-1$)

INDEX FILE ORGANIZATION

An index is whatever that let us find something quickly given one/some properties in a set of records (eg index of book or index of array)

Every index has a different "strategy" to find the record(s) and the value of one or more field that they use as base for organization is called search key

Index file:

- Data entries: each containing a value K of the search key, used to locate data records in data file related to K
- Index entries: used for the management of index file

Data file: containing data records

Properties:

- Organizations:
  - Sorted Index: the index is a sorted file
  - Tree-based: the index is a tree
  - Hash-based: the index is a function from search key to record address "ALTERNATIVES"
- Structure of data entry (Note K^* = data entry which value is K)
  1) K^* is a data record: extreme case, data entry = data record
  2) K^* is a pair (K, r): r = reference to data record with search key K
  3) K^* is a pair $(K, r\text{-list})$: r = list of reference, better use of space but variable-length data entries

Note: index > 1 on some data file → at most one of them can use ①

- Clustering or not

An index is strongly clustered when data entries are sorted according to some criterion of data records in data file. Otherwise non strongly clustered

An index is weakly clustered if ∀ value V of search key, all tuples of the indexed data file with value V appear in some page of data file

Index with structure 1 is clustered by definition. Index is clustered only if data records are sorted in data file according to search key. At most one clustered index per data file.

A primary key index is an index on relation R whose search key includes the primary key of R. Otherwise non-primary key / secondary index

Secondary index may contain duplicates. Its called unique if its search key contains a non-primary key → doesn't contain duplicates.

Index is dense if every value of search key that appears in data file appears also in at least one data entry of index, sparse otherwise.

Strongly dense if we have exactly one data entry for each data record of data file and value of search key in each data entry is the value held by the referenced data record

Index can be simple if search key is simple (single field), composite otherwise.

Single level index uses only one index structure while a multi level index uses indexes built on other index.

SORTED INDEX

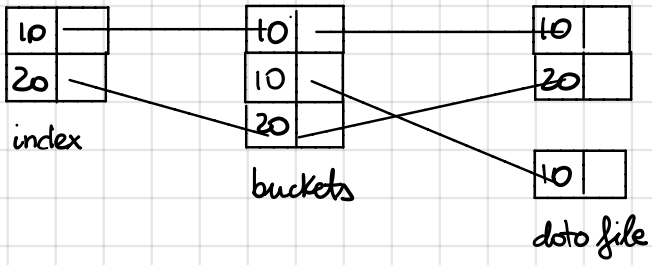
Idea: create auxiliary index sorted file which contains the value for the search key and pointers to records in data file. → binary search on index file instead of on data file.
It can be clustered or not. In both case we don't use data structure ① (go back and check)
In some way it can be primary or secondary and dense or sparse.

"Strong clustering index, greatly support range search, weakly clustering index roughly support range search, unclustering index does not support range search." - Martin Luther King -

If we need to do many page access we can add multiple level of sorted index. (eg. dense level over a sparse one).

Note: when inserting new records in clustered sorted we may have no free space available (in the right space)
To overcome this we can ① restructure the data file and index (shift records) or ② add overflow blocks: pointer to new page in different location (one per record). When to much / full → ①

Non clustering index can have duplicates and one option to deal with them is using buckets: first level index (no duplicates) points to different position on list of all key (with duplicates) that will point to corresponding data records. eg.

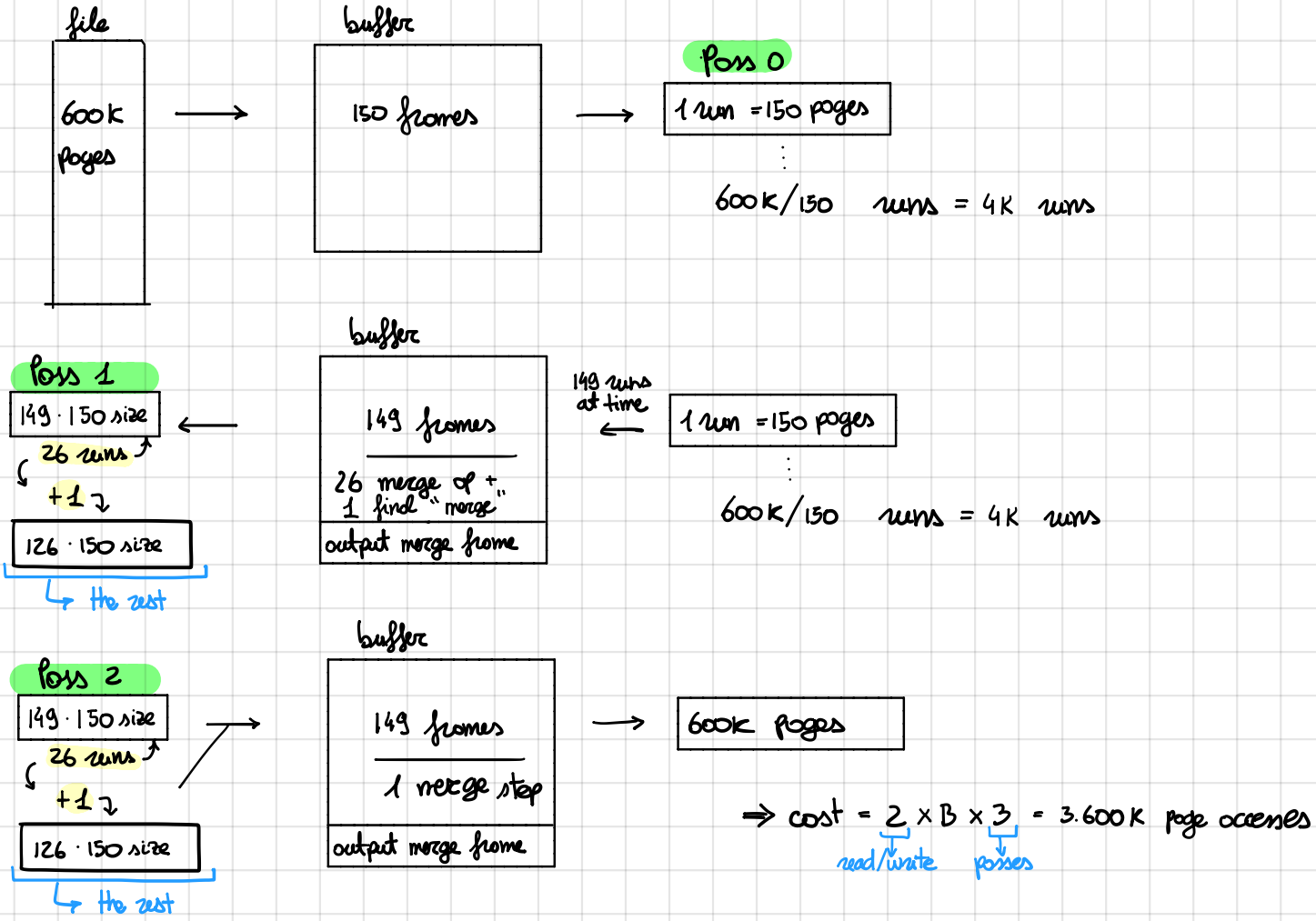


Ex

Suppose we have a file stored in 600.000 pages, and we have 150 free frames available in the buffer.

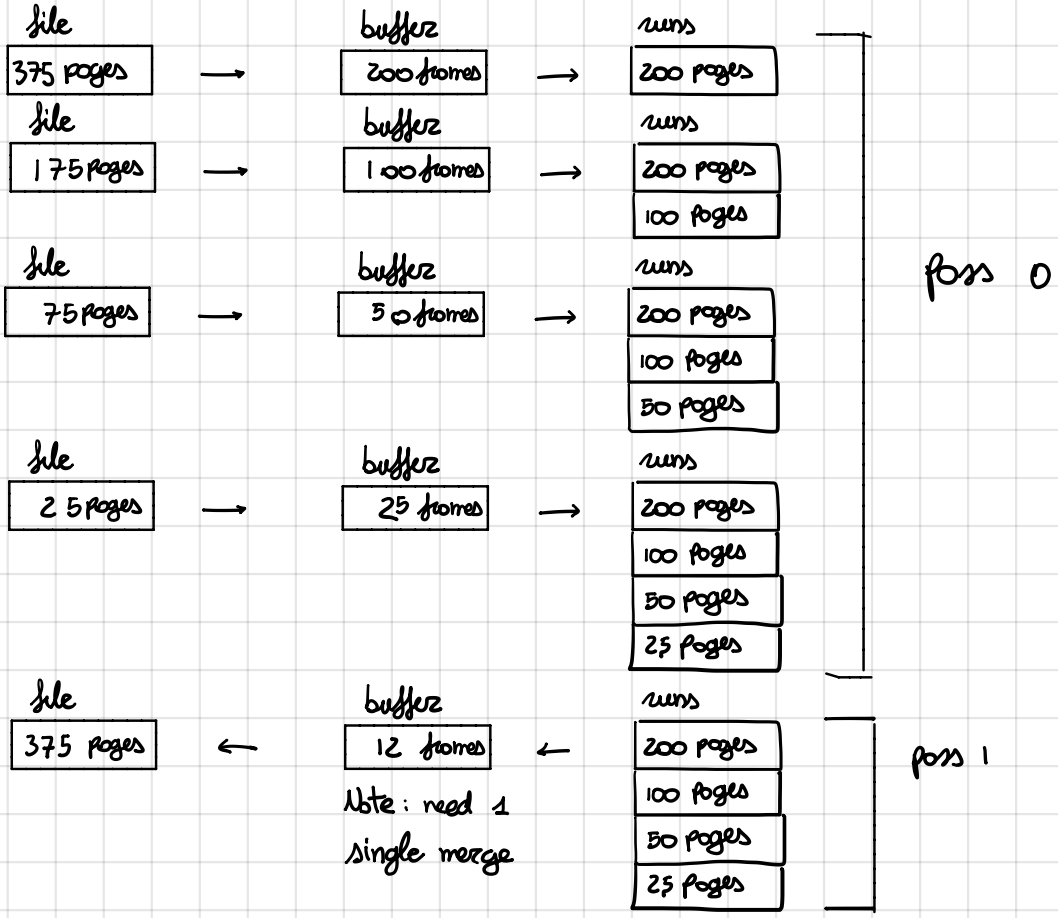
1. Illustrate in detail the algorithm for sorting the file by means of the multipass merge-sort method, specifying for each pass how many runs the algorithm produces, and which size (in terms of number of pages) have such runs.

2. Tell which is the cost of executing the algorithm in terms of number of page accesses.



Ex

We have to sort a relation R with 375 pages using the multipass (or, k-way) merge-sort algorithm, and initially we have 200 free frames in the buffer. However, the system is currently very busy, and every time a run is written in secondary storage during the execution of the algorithm, after such writing the number of free frames in the buffer is halved. Describe in detail what happens during the execution of the multipass merge sort algorithm in this situation, and tell how many pages are accessed during such execution.



⇒ $2 \times B \times 2 = 1500$ page accesses

Ex. We have to sort a relation R with 9.000 pages using the multipass (or, k-way) merge-sort algorithm, and we know that the number of free frames in the buffer that will be available for the algorithm is between 50 and 100. Tell which is the cost of the algorithm in terms of the number of page accesses, both in the worst case, and in the best case.

Worst case: 50 frames always
Pass 0: $F = 50$
Pass 1: $F \times (F-1) < 9000$
Pass 2: $F \times (F-1) \times (F-1) > 9000 \Rightarrow$ 3 passes needed $\Rightarrow 2 \times 9000 \times 3 = 54000$

Best case: 100 frames always
Pass 0: $F = 100$
Pass 1: $F \times (F-1) > 9000 \Rightarrow$ 2 passes needed $\Rightarrow 2 \times 9000 \times 2 = 36000$

Ex We have a relation R(A,B,C,D) with 15.000.000 tuples, where A is the primary key, and we know that every attribute and every pointer has the same size. We also know that 10 tuples of R fit in one page, and there is a primary, clustering sorted index using alternative 2 for R, with A as search key. Tell which is the number of page accesses required for answering the following query

```
Select B, C
from R
where A = 50
```

Since is clustered, index can be sparse too. We cant do an "one index evaluation" since we need to look for B and C (in addition to A).
$\log_2 N$ to find the first "50" in the index. → N depends on index dense or sparse!

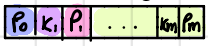
Dense: store 15.000.000 data entries → each with A value + pointer. We know 10 tuples (of 4 attributes) fit in 1 page → 1 page = 20 entries in index (because they need only 2 "arguments": A + pointer)
⇒ 15000000/20 = 750000 pages ⇒ $\log_2 750k$ = 20 page accesses + 1 (for accessing pointer) = 21

Sparse: 1 index entry for every page in R. B = # pages in R = 15000000/10 = 1.500.000 → N = B/20 = 75000
→ $\log_2 N = 17$ → cost = 17+1 page accesses

Note: in both cases seems omitted the cost to "scan" the page after pointer to find all A=50 occurrences

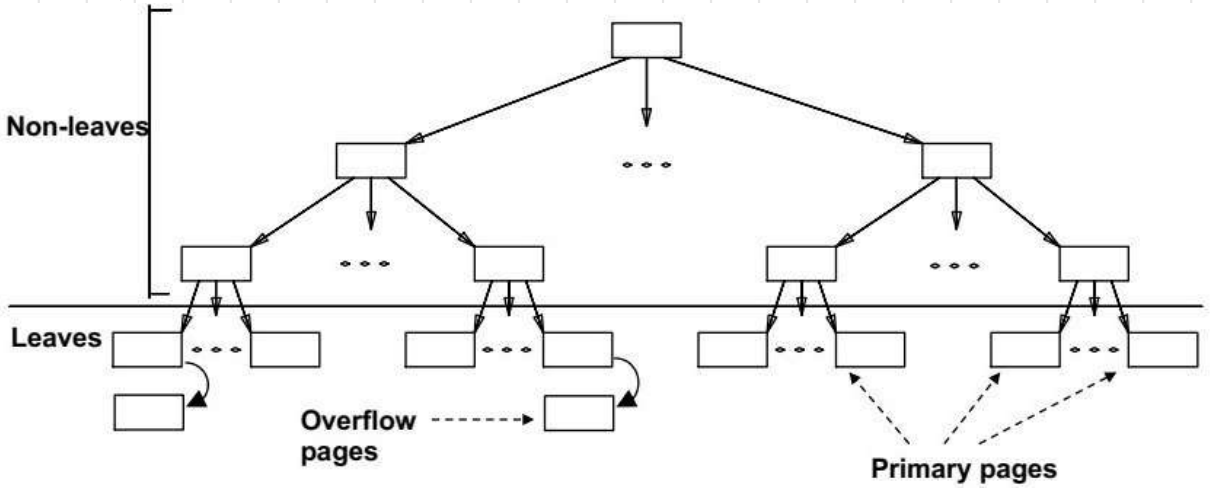
TREE INDEX

The basic idea is like having a multilevel sorted index until we get a single "entry" = the root of the tree. Searching for an entry means start from root to the leaf with the key we are looking for.
There are two kind of tree index : ISAM and B^+ , the first only for few insertion / deletion (static)

Node : 

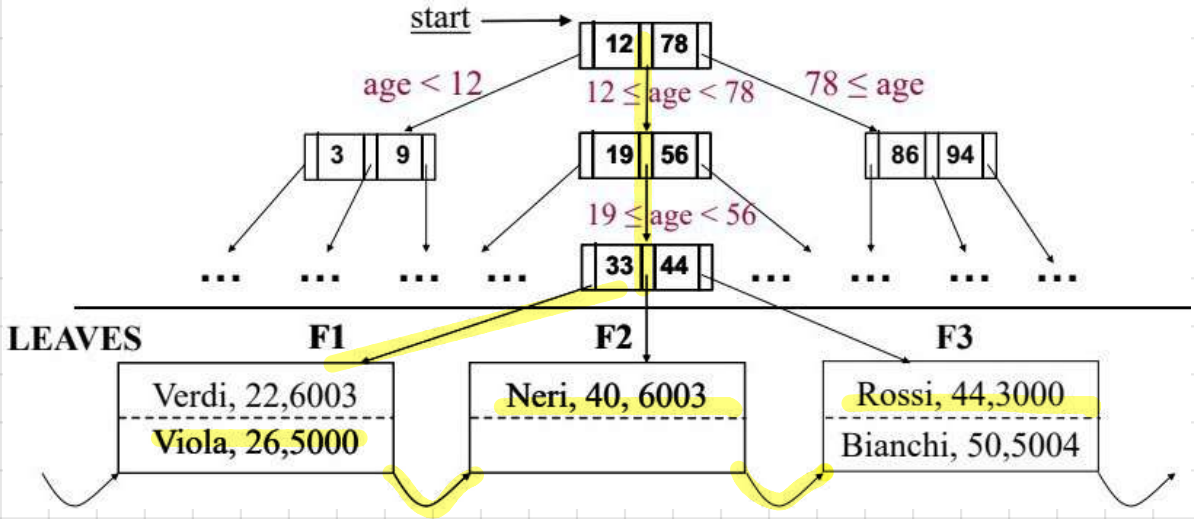
sequence of m+1 pointers P separated by different values K ordered according to search key. P_{i-1} points to subtree that contains value less than K_i . P_i instead the ones greater or equal.

ISAM : indexed sequential access method, all intermediate node have some number of children (complete tree)



The leaves contains the data entry and they can be scanned sequentially
Its a balanced tree (same heigth for every leaves)
# of children = fan-out → fan-out = F and heigth = h → F^h leaf pages
CREATION : bottom-up, allocate leaves sequentially and then create intermediate nodes.
SEARCH : from root and then compare the search key with the key of the tree until we find the leaf. Cost $\log_F N$ (F = fan-out and N = # of leaves)
INSERT : find correct leaf, overflow page if needed. Insert data record in data file.
DELETE: find the leaf and delete data record from data file, if overflow deallocate it.

B^+ : adaptation of ISAM for insertion and deletion. If each page has space for "d" search key values and "d+1" pointers then d is the rank of the tree. Every node n_i contains m_i search key values and $(d+1)/2 \leq m_i \leq d$ (except root). Leaves are pages with data entries, linked through a list based on order of search key.
Fan-out still (if happens). If intermediate nodes have different number of children we can use F = average number of children → F^h = good approximation of number of leaves → M = #leaves → $\log_F M$ = approximation of height
eg. Search $24 < age \leq 44$



Note: using alternative 1, leaves are data entries (no pointers) = data records

Equality search cost at most $\log_F N$ = height of tree → aim to have F very high (depends on block size)
INSERT: search for appropriate leaf and put there new key if there is space. Otherwise split the leaf and divide into equal parts the keys between the two new nodes. Add recursively the new pointer on father node. If we need to split the root do it and generate a new root ($h = h+1$). cost = $\log_F N$
Note: the middle value goes to the higher level
DELETE: if the node still has the minimum number of key of delete, perfect, we are done. Otherwise:
① Key redistribution: right sibling has more than minimum number of key, take the minimum one and move to the node where we had the deletion. Then change the key at parent pointing to the right sibling (cause we changed the min.)
or ②: coalesce: merge one of the siblings and then delete the pointer to it from the father (recursively adjust parent if needed).
Note: sometimes deletion just leave blank space (no coalesce). cost = $\log_F N$

CLUSTERED B+ tree index: alternative 1 and F = fan out. B = min number of pages required for storing data entries = pages for data record in alternative 1 → 1.5B = number of pages with data entries = number of physical pages forming leaves
SCAN: $1.5B(D + RC)$
SELECT (equality): $D \log_F(1.5B) + C \log_2 R$ Note B change in alternative 2
SELECT (range): same as equality but with further I/O
INSERT / DELETE: cost of search + insertion + write, $D \log_F(1.5B) + C \log_2 R + C + D$

In tree based index, performance depends on number of physical pages stored as leaves!
- Pages in leaves are occupied at 67% → physical pages are 50% more of "required leaves".
- Alternative 1 → required leaves = pages in data file
- Index dense on a key of relation → one data entry per data record → if we know data entries per pages we can compute required leaves
- Index dense, secondary non unique pay attention to how many data entries fit one page
- Index sparse → data entries in required leaves = pages in data file

Consider the relation

PROVINCE(name, area, president, population)

with 200.000 tuples, and such that 20 tuples fits in each page. Relation PROVINCE is static (essentially no insertions, no deletions), and it will be frequently queried to compute all provinces whose area is in a given range.

1. Which is the method you would choose to represent the relation PROVINCE?
2. Explain why.
3. On the basis of the selected method, compute the average cost (in terms of page accesses) of the query

   select *
   from *PROVINCE*
   where *area = <constant>*

   assuming that in the average the number of provinces with the same value for the attribute area is 18.

Range query → clustered index

Static relation → ISAM

① ISAM tree index with alternative 2 (but 1 is ok too), sparse

② because we have to compute many range query, key = area → small thanks to sparse architecture

③ How many leaves? 200.000 / 20 = 10 000 pages in PROVINCE, 40 data entries fit one page (because 20 tuples of 4 fit in one)

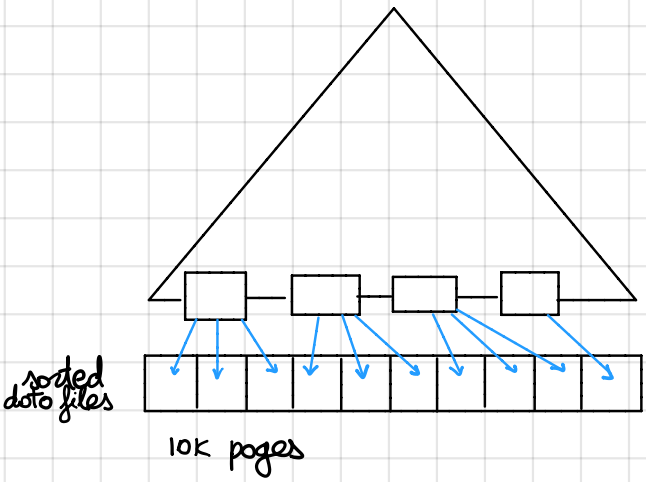
leaves = 10000 / 40 = 250

Note: "67% rule" is only on B+ tree!

F = 40 because 1 node = 1 page and index entries are <value, pointer> → 20 tuples of 4 : 1 = 40 of 2 : 1

cost = $\log_{40} 250$ + 1 + 1 = 4

$\log_{40} 250$ find leaf; + 1 access data; + 1 on other for worst case (not all 18 in same page, at least 2)



leaves - 40 data entries each

no crossed lines because it is clustered

Consider the relation WRITING(autcod, bookcod, booktitle, …), containing information on books written by authors. We know that the authors are 67.000, and each author has written 10 books in the average. Also, we know that a non clustering B+-tree index using alternative 2 is available on WRITING with search key <autcod>, and such that in each leaf we have space for 100 data entries.

1. Tell what is the average cost (in terms of page accesses) of the query

   select *
   from *WRITING*
   where *autocod = <constant>*

   assuming that the query evaluation algorithm uses the B+-tree index.
1. Explain the method used to answer question 1.
2. If the size of each data entry is 10 bytes, which is the size (in bytes) of the space occupied by all the leaves of the tree index?

Non clustering → dense

$67000 \cdot 10 = 670.000$ data record / data entries → $\frac{670000}{100} = 6700$ leaves ×

But for "67% rule" each page can store 67 entries → $\frac{670000}{67} = 10\,000$ leaves ✓

In every node we have $d/2 < n < d$ children with $d = 10000/100 = 100$ → $F = 75 = (\frac{50+100}{2})$

This is because "67% rule" apply only on leaves!

① cost: $\log_{75} 10000 + 1 + 10 = 14$

find leaf access all pointers

CHECK clustered / non clustered index difference

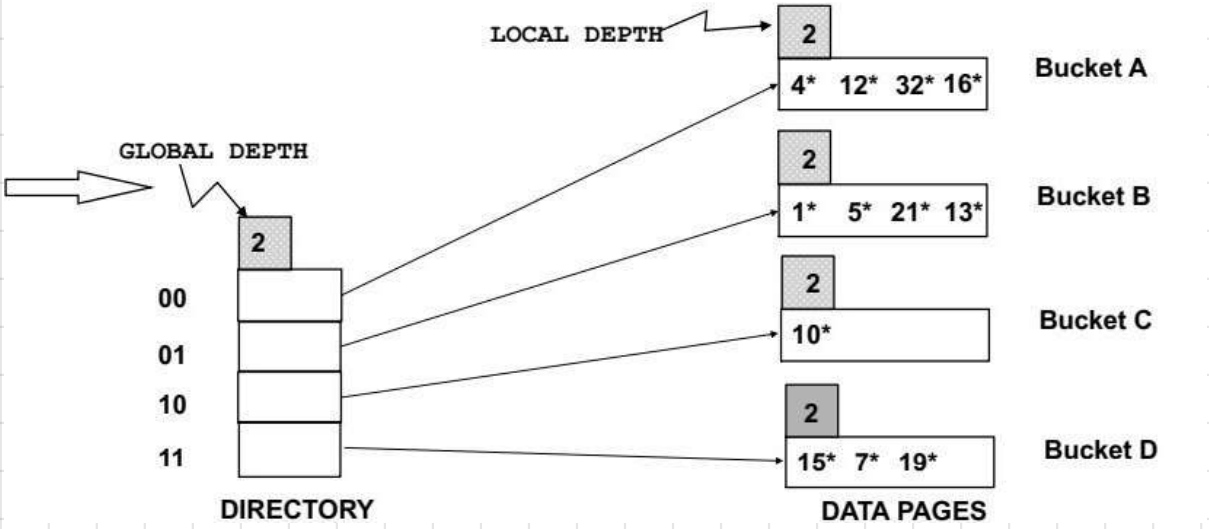
HASH INDEX

Static Hash : coincides with hashed file (see simple file organization)

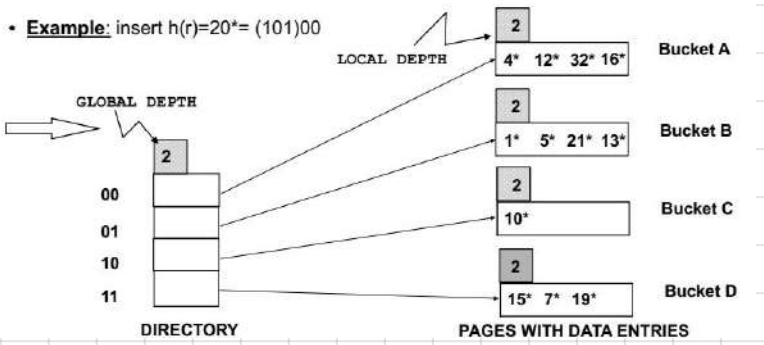
Extendible Hash: when primary page is full (bucket), double number of bucket? too costly! → double a directory of pointers to buckets (+ adapt hash function)

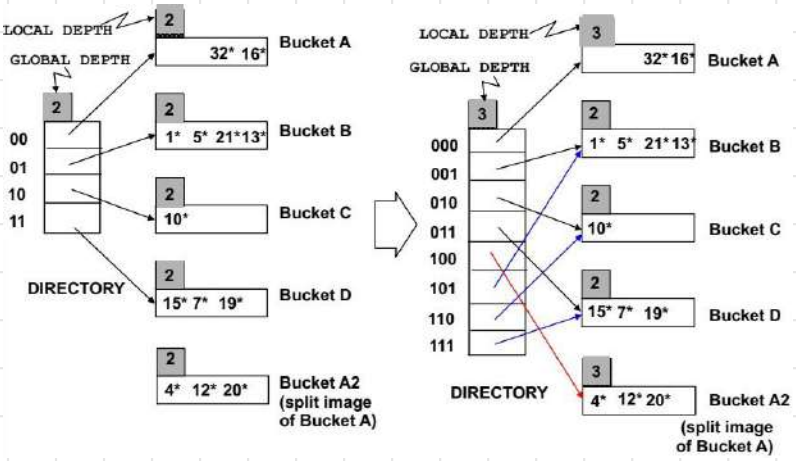
e.g.

- The directory is an array of 4 items -- in the picture, k is denoted by h(k)*
- To find the bucket for *k*, consider the last g bits of **h**(*k*), where g is the *global depth* (note that, with this method, the buckets for k_1 and k_2 can be the same, even if k_1 and k_2 do not collide -- i.e., even if $h(k_1) \neq h(k2)$)
- If **h**(*k*) = 5 = binary 101, then the bucket for 5* is the one pointed by 01



INSERT: if bucket B is full but the search key value k is such that h(k) is different from at least one entry in B in the last c+1 bits (c = local depth) then split it and re-distribute the entries in B and its split image according to the same hash function h, but using c+1 bits. If needed double the directory. In particular the decision on whether to double or not the directory is based on comparing global depth and local depth of bucket to be split: if global = local





Global depth: max number of bits needed to decide which is the bucket of given entry

local depth: number of bits used to decide whether an entry belongs to a bucket.

Bucket Split: when inserting, if bucket full and we can distribute data entries in two buckets using c+1 bits

Directory Doubled: when inserting, if bucket split and if local depth = global depth → insertion would make local > global Then directory must be doubled = copy it and setting up pointers to split image.

EVALUATION OF OPERATORS

QUERY PROCESSING

Sql query is first compiled by SQL engine : parsed and optimized. Then it is executed on data to retrieve the result.

Optimization = logical query plan + physical query plan = identify operators + (order of evaluation of query subparts + choose method to evaluate operators)

Operators :

- Union, Intersection and Difference (for sets and bags too) Note. Bags = sets with duplicates
- Selection and Projection
- Cartesian product and Join
- Duplicate elimination (from bag to set)
- Grouping
- Sorting

We are interested in algorithms to implement these operators. Operators can be applied in two ways :

Materialization - based : system computes result and store it in a temporary relation (file)

Pipeline - based : use iterator, one single tiple (of result) on every execution (eg. iterator.next())

↳ Open(), GetNext() and Close()

Algorithms can be classified in :

Sequential : classification change according to number of passes of reading from secondary storage

1) One Pass : algorithm read data only once
2) Nested - loop : loop inside loop (to analyze an operation)
3) Two Pass : read data twice, one for preprocessing and then write back and the second for further procesing
4) Multi Pass : 3 or more passes, generalization of two pass.
5) Index Based : use one or more indexes.

Parallel : use many processor in shared-nothing architecture.

Another classification :

Sorting Based : one or both operands need to be sorted. Two/Multi Pass used

Hash Based : one or more hash function needed. Two/Multi Pass used

Index Based : one or more index needed.

B = B(R) = number of pages used to store relation R in secondary storage (block)] B ≥ QB

QB = QB(R) = number of pages required to store records of R (min value)]

SR(R) = average size of space per page occupied by each record of R]

SP = size of page] QB(R) = NR(R) × SR(R) / SP

NR = number of records]

- No free space → QB = B → clustered relation (Not the file!)
- C% of free space in each page → B(R) = QB(R) × 100 / (100 - C)
- R is represented by a clustered file and R is the "child" relation → C% not occupied by R → B(R) = QB(R) × 100 / (100 - C)
- Relation is a clustered file and R is the "father" → C% occupied by father → B(R) = QB(R) × 100 / C

From now, if not stated, we assume clustered relations.

Note : in cost analysis in term of page accesses, we ignore the cost of page accesses due to the need of writing the result in secondary storage

ONE-PASS ALGORITHMS

- **Projection on R** (with no index on projected attributes):
  read pages of R one by one into input frame, perform operations on tuples and write them on output buffer. When output frame is full, write in secondary storage
  Space: input and output frame
  Cost: B = number of pages
  Note: we don't need B input frames because we don't need to store data all at some time in buffer!
  Note: we need to buffer output (output frame) because during projection some attributes are deleted generating free space in pages in input frame → output frame compact them.
- **Selection on R** (R not sorted and without relevant index)
  Similar to projection, write selected tuples in secondary storage
  Space: input and output frame
  Cost: B = number of pages
- **Selection on R** (R sorted on attributes used to search without relevant index):
  it's the case where the "where" condition refers to search key on which R is sorted → binary / interpolation search
  Space: input and output frame
  Cost: $O(\log_2 B)$ with binary search. If search key is not key of R we have to add as many page access as number of pages of interest.
- **Duplicate elimination on R** (R not sorted and without relevant index):
  read each page one at time into input frame and keep track of tuples using hashtable or balanced binary tree → $O(N^2)$ for comparison
  Space: 1 input and M-1 output → R without duplicates must fit at most M-1 frames (to store hashtable / tree)
  Cost: B = number of pages
- **Duplicate elimination** (R sorted without relevant index):
  duplicates now are close so we only need to "remember" the last tuple we encountered so far.
  Read pages one at time into input frame and keep in buffer (without duplicates) the group of tuples with value equal to last value seen. For each tuple in reading page we check if equal to one of tuples in buffer: yes ignore, no add to group. When tuple belongs to new group, write previous group into result
  Space: 1 frame to read and M-1 to store group → groups must fit M-1 frames
  Cost: B = number of pages
  Note: duplication is meant on multiple attributes eg PERSONS (birth year, name ...) so the group is on "birth year".
- **Grouping R** (R not sorted and no relevant index):
  read one page at time in input frame and create an entry in buffer for each grouping attribute + value of aggregation ie SUM / AVG ... Write when read all tuples!
  Space: 1 input frame and M-1 output frame
  Cost: B = number of pages
- **Grouping R** (R sorted and no relevant index):
  same group members are close and we can calculate one group at time. When new group, write the previous one
  Space: 1 input and 1 output frame
  Cost: B = number of page
- **Bag Union:**
  having bag R and S we simply read every page of R, write them and repeat on S
  Space: 1 frame
  Cost: B(R) + B(S)

- **Other binary operator** (R and S not sorted):
  read the smaller into buffer and build data structure suitable for it (find/insert quickly)
  Space: $\min(B(R), B(S)) \leq M-2$. 1 input and 1 output frame
  Cost: $B(R) + B(S)$
  Note: from now on S is the smaller
- **Set Union**:
  Load S in $M-2$ frames into a data structure and also copy them in result. One by one read R's pages and check if tuples are in S: no, add to output, yes, ignore. Write output when full.
  Space: 1 input, 1 output and $M-2$ frames to store S
  Cost: $B(R) + B(S)$
- **Set Intersection**
  as before, if tuple is in S write in output, otherwise ignore
  Space: input, output and $M-2$ for S
  Cost: $B(R) + B(S)$
- **Set difference** $(R - S)$
  as before, if in S ignore, write otherwise
  Space: input, output and $M-2$ for S
  Cost: $B(R) + B(S)$
- **Set difference** $(S - R)$
  as before, if in S remove from data structure. At end copy data structure $(M-1)$ to output
  Space: input frame and $M-1$ for S
  Cost: $B(R) + B(S)$
- **Bag Intersection**
  Note: $R = \{2,2,3,4\}$ $S = \{2,3,3\} \rightarrow R \cap S = \{2,3\}$ → repetition must be the same on both bags
  Load S in $M-2$ frames and build data structure counting tuples occurrences. Read R's pages one by one in input frames and check if in S: yes, add to output and decrement counter (occurrences), no, ignore. When counter = 0 delete
  Space: input, output and $M-2$ frames for S. Note: we can avoid to store duplicates of S (we have counter!)
  Cost: $B(R) + B(S)$
- **Bag difference** $(S - R)$
  Read S in $M-1$ frames and build some data structure. Read pages of R one by one in input: if in S decrement counter, when 0 delete. At the end write all to result.
  Space: input + $M-1$ frames for S
  Cost: $B(R) + B(S)$
- **Bag difference** $(R - S)$
  Read S in $M-2$ frames with usual structure. Read R in input frame: if in S decrement its counter, otherwise copy to output buffer. When full, write to result.
  Space: input, output and $M-2$ frames for S
  Cost: $B(R) + B(S)$
- **Cartesian Product**
  Read S in $M-2$ frames and pages of R one by one in input. For every tuple concatenate it with every tuple of S and write to output. When full write to result.
  Space: input, output and $M-2$ frames
  Cost: $B(R) + B(S)$

- **Natural Join**

We assume that R(X,Y) is joined with S(Y,Z) where Y = all attributes that R and S have in common. S still the smaller one.

1) Read S in M-2 frames into data structure with Y as search key
2) Read R in input frame and for each tuple we find the ones of S that match with all attributes in Y and fuse them in output. When full, write into result.

Space: input, output and M-2 frames for S
Cost: B(R) + B(S)

NESTED-LOOP ALGORITHMS

- **Duplicate elimination**

Consider an order on pages of R and read them one page P at time. For each of them read all subsequent pages in an other input frame: if tuple t is not in any other page, write it in output frame otherwise ignore copies (copy in output only once)

Space: 3 frames
Cost: B(B+1)/2

Suppose we can use more than 3 frames (M + 2 frames in total): load M pages in these M frames instead of one single page and scan subsequent pages as before. In this way we can eliminate duplicates in the block too! ⇒ block-nested-loop algorithm

Space: M+2 frames
Cost: $B \times (3/2 + B/M) + M$ (approximation)
⇒ with two operands: $B(S) + B(R) \times (1 + B(S)/M)$

Very high cost, not often used but can ALWAYS being executed ($M \gg 1$). Join operator uses this schema

- **Join**

One of the arguments is read once but the second multiple times → one and half pass

p_x = average number of tuples of relation X

eg B(R) = 1000 B(S) = 500 p_R = 100 p_S = 10 and 10 ms per page access

- Tuple-based nested-loop JOIN

Scan S
↳ for each page G and for each tuple E in it
↳ scan R looking for tuples "joining" E

Cost: $B(S) + (p_S \times B(S) \times B(R))$ ~ 140 hours

- Page-based nested-loop join

Scan S and for each page G
↳ scan R looking for tuples joining those in G

Cost: $B(S) + B(S) \times B(R)$ ~ 1.5 h

- block nested-loop JOIN

Outer relation fits M frames (of the M + 2 we have)
For each chunk of M pages
↳ read pages into buffer and organize tuples in data structure with search key = common attributes of R and S
For each page b of R
↳ read b and for each of its tuples find the ones matching in the data structure and store them in output

Cost: $B(S) + B(R) \times (1 + B(S)/M)$ ~ 1 min with M = 102

TWO-PASS ALGORITHMS

Deal with situations where one pass is not enough. Read into main memory, process, write to disk and re-read from disk to complete operation.

Two types: based on sorting and based on hashing

BASED ON SORTING

Relation such that $B(R) > M$.

1) Repeatedly do the following (with multiple relations do it for all)
   - Read M pages into buffer
   - Sort them in main memory (sublist)
   - Write sublist into M pages in secondary storage
2) Process the sorted sublist to execute the desired operation using one buffer frame for the output

- **Duplicate elimination**

After building sublists we use available frames to hold one page for each sublist (1 frame - 1 sublist) and repeatedly copy into output the minimum tuple among the ones under consideration and ignoring duplicates

Space: $M-1$ frames for the $M-1$ sublist (each of M pages) and 1 output frame $\sim \sqrt{B(R)} + 2$ frames

Cost: $3B(R)$ = create sublists, write them, read each page from them.

- **Grouping and aggregation**

In pass 1 build sublists using grouping attributes as sort key

In pass 2 repeatedly do the following: find least value of sort key among first available tuples in buffer. This value v becomes next group, prepare corresponding tuple in output and
a) examinate each tuples sorted according to v and accumulate the needed aggregates (eg sum/avg..)
b) if buffer becomes empty read other page from sublist

When no more tuples with sort key v the group is complete and when all tuples in output are complete we write to secondary storage.

Space: $\sim \sqrt{B(R)} + 2$ frames

Cost: $3B(R)$

- **Set union**

In pass 1 build sublists for both R and S

Use one frame for each sublist of R and S and repeatedly find the first remaining tuple among all frames. Copy it to output and skip all its copies

Space: $\sim \sqrt{B(R)+B(S)} + 3$ frames

Cost: $3(B(R)+B(S))$

- **Simple sort-based JOIN**

Pass 1: sort $R(X,Y)$ and $S(Y,Z)$ based on y with two-pass multi-way merge-sort ($B(R) \leq (M-1)M$ and $B(S) \leq (M-1)M$)

Pass 2: assume for no value y of Y appearing in both R and S, $B(R)_y \cup B(S)_y > M-1$ frames!

Then we repeatedly do the following using one frame for R and one for S
1) find the least value y of Y currently at front of both frames
2) if y is only in one of them ignore all its tuples, otherwise use $M-3$ frames to identify all tuples with y and put in output the join tuples.

Note: if for a given y the corresponding tuples occupy more page we have to load them in more frame i.e. we always have to have in buffer all tuples of current value y (for both R and S)

Space: $B(R) \leq M(M-1)$ and $B(S) \leq M(M-1) \rightarrow \sqrt{\max(B(R),B(S))}$ ion

Cost: $4(B(R)+B(S))$ for pass 1 and $B(R)+B(S)$ for pass 2 $= 5(B(R)+B(S))$

Otherwise if the assumption is not valid anymore we do the following:
1) if the tuples of one relation with value y fit $M-2$ frames then we load them and one by one we load pages

of second relation with value y i.e. one-pass join on only the tuples with y value
2) otherwise we use M-1 frames to perform nested-loop join of tuples with value y
Note: in both case might be necessary to read both relation to know if the fit or not.

- **Sort-merge JOIN**

If we know in advance the "too many pages" case is very rare we can use this algorithm
First create sorted sublists for R and S and then
1) bring first page of each sublist into buffer
2) repeatedly find the least y among all sublist. Identify all tuples in both relations with value y and join them in output frame

Space: $\sqrt{B(R)+B(S)}+3$ frames
Cost: $3(B(R)+B(S))$

BASED ON HASHING

If data is too big to be processed in one pass, hash all tuples of operand(s).
For all common operations there is a way to select hash key so that all the tuples that need to be considered together has the same hash and are in same (pair of) bucket
Then perform operation working one bucket at time. With M frames we can pick M-1 buckets and handle relations of size M with respect to the case of one-pass algorithm.

Pass 1: partitioning relations by hashing
Initialize M-1 bucket/frames
For each page b of R
↳ read b into M^{TH} frame.
For each tuple t of b
↳ if frame h(t) has no space for t → copy frame in secondary storage and initialize a new empty page in it
Copy t in h(t)
For each bucket
↳ if corresponding frame is not empty write to secondary storage

- **Duplicate elimination**

Pass 1 as said before and then proceed as follow: we can examine one bucket at time since same tuples will hash to same bucket and produce R_i = portion of R that hash to i^{TH} bucket and then union all partitions. We can use one-pass algorithm on each backed if R_i fit in buffer.

Space: each $R_i = B(R)/M-1$ pages → $size(R_i) \leq M$
Cost: access pages for read and write and then re-read buckets → $3B(R)$

- **Grouping and aggregation**

Partition R using h depending only on grouping attributes. Pass 2 is a one-pass algorithm that process each bucket b forming the tuple of result derived from each group stored in b knowing that all such tuples are there

Space: we can store each bucket in buffer if $B(R)/M-1 \leq M-1$ but in second pass we only need one tuple per group
Even if $size(b) > M$ we can handle it in one pass if all tuples for all groups in it fit M-1 pages.
So pages needed to store all group in each bucket $\leq M-1$. Pages need to store one tuple per group in each bucke are $(B(R)/M-1)/A$ (A = avg tuples per group) → $(B(R)/M-1)/A < M \rightarrow B(R) \leq (M-1)(M-1)A$ ~ dont know how

Cost: $3B(R)$

- **Union, Intersection and Difference**

Partition R and S → 2(M-1) buckets then load in buffer the various pair of buckets R_i and S_i and apply appropriate one-pass algorithm

Space: one-pass algorithms require S in at most M-2 pages → $\min(B(R), B(S)) \leq (M-1)(M-2) \leq (M-2)^2$
Cost: $3(B(R)+B(S))$

- **Hash-Join**

  Partition R and S. Load R_i and S_i and apply one-pass JOIN

  Space: $\min(B(R), B(S)) \leq (M-1)(M-2) \leq (M-2)^2$

  Cost: $3(B(R)+B(S))$

Note: Hashbased algorithm buffer requirement depends only on smaller of the two operand while sort-based depends on the sum of them.

MULTIPASS ALGORITHMS

Two pass algorithms can be generalized to use as many passes as necessary (no limit to size of relations)

MULTIPASS SORT-MERGE (SORT-BASED)

- Recursive
  - base step: if R fits M frames, sort it in buffer with main memory and write to secondary storage
  - inductive step: if not, partition R in M-1 groups and recursively sort them. Then merge the M-1 sublists using M-1 buffer and write to secondary storage

Unary operations

- First phase: partition R in M-1 parts and sort them using (multipass) merge-sort
- Second phase: load one page at time of sorted sublists and perform tuples operations using output frame

Binary operations

- First phase: partition R and S in M_R and M_S such that $M_R + M_S \leq M-1$ and sort with (multipass) merge-sorte
- Second phase: same as before

If K = # passes → Unary cost: $2(K-1)B(R) + B(R) = (2K-1)B(R)$

↳ Binary cost: $2(K-1)(B(R)+B(S)) + B(R) + B(S) = (2K-1)(B(R)+B(S))$

To use K passes we need $B(R)^{1/K} + 1$ buffer frames for unary operation and $(B(R)+B(S))^{1/K} + 1$ for binary. ($\leq M$)

MULTIPASS HASH-BASED

Hash relation(s) into M-1 buckets. Apply operation on each bucket individually if unary operation. If binary apply to each pair at time. Result is the union of results on buckets.

BASIC STEP: for unary operations, if relation fit buffer, read it and apply operation. For binary, if one of the relations fit M-1 frames read it and perfor operation reading second relation one page at time into M^{TH} frame

INDUCTIVE STEP: if none of them fits then hash each into M-1 buckets and recursively perform operation on each bucket (unary) or pair (binary) and accumulate output.

Unary Cost: $(2K-1)B(R)$

Binary Cost: $(2K-1)(B(R)+B(S))$

To use K passes we need $B(R)^{1/K} + 1$ buffer frames for unary operation and $\min(B(R), B(S))^{1/K} + 1$ for binary. ($\leq M$)

- Relation R is stored in 8.000 pages, relation S is stored in 24.000 pages, and our DBMS has 500 free buffer frames. We want to compute the bag difference $R -_B S$. Consider the following two questions:
  1. Among the one-pass and the two-pass algorithm based on sorting, which one would you choose, and why?
  2. Describe in detail the algorithm you have chosen, compute the size and the number of the sublists produced, and tell how many page accesses it requires for computing $R -_B S$.
- Answer the above two questions in two scenarios: (A) R and S are not sorted; (B) R is not sorted, but S is sorted on all attributes.

First scenario:

A: we cant use one pass because they are not sorted and in this case they should fit in buffer which they dont we can use two pass because $\frac{8000}{500} + \frac{24000}{500} \leq 499$

B: Create $8000/500 = 16$ sublists for R and $24000/500 = 48$ for S, all of them sorted. Compute the bag difference reading each sublists one page at time in corresponding buffer frame + output frame for result. (16+48 fits in 499). Cost: $3 \times (B(R) + B(S))$

Second scenario:

A: we can do better (no need to sort S), but still two-pass

B: we create $8000/500 = 16$ sorted sublists for R and compute bag difference using 16 frames for R and reading S one page at time + output buffer for result. Cost: $B(S) + 3B(R)$

A relation R(A,B,C) is stored in 8.000 pages (with 100 tuples per page), a relation S(D,E) is stored in 5.000 pages (with 100 tuples per page), and our DBMS has 122 free buffer frames. We want to compute the join of R and S on the condition C = D, and we know that the 400 different values stored in the attribute C are uniformly distributed in the tuples of R, and the 160 different values in the attribute D are uniformly distributed in the tuples of S.

1. Can we use the block nested-loop algorithm?
2. Can we use a two-pass algorithm based on sorting?

For each of the two cases (1 and 2), (i) if the answer is no, then explain the answer; (ii) if the answer is yes, then describe in detail how the algorithm (or, for case 2, the variant chosen) works, and tell which is the cost of the algorithm for computing the above join in terms of number of page accesses.

1) Yes because block nested-loop algorithm can always be used. → cost: $B(S) + (B(S)/M-2)B(R)$

2) Since $B(R) \leq (M-1)M$ and $B(S) \leq (M-1)M$ and $B(R)/M + B(S)/M \leq M-1$
we could use both variants of two-pass join algorithm. Which one?
The best is sort-merge join but the condition "for no value of joining attributes the size of tuples with that value exceed buffer" should be satisfied!
R: #tuples with some value $= \frac{8000 \cdot 100}{400} = 2000 \rightarrow \frac{2000}{100} = 20 + 1$ pages
S: ... = 32 pages
For this algorithm we need: $\left(\frac{8000}{122} = 66\right) + \left(\frac{5000}{122} = 41\right)$ frames to store the sorted sublists + 20 frames to store the tuples with some value + 1 output frames = 128 > 122 → CANT DO IT!
⇒ Use simple-sort join because for every value of C, all 21 pages of R with tuples with that value will fit the buffer (20 < 122-2)
Cost: $3(B(R) + B(S))$

Consider the relation CAR(code,owner,type,year), storing information about cars, each one with its type, its owner, and the year of its construction. CAR has 500.000 tuples stored in a heap file, where each page contains 50 tuples. Consider the aggregate query Q that, for each owner o, computes the number of cars owned by o, and assume that we have a good hash function on owner that distributes the tuples of CAR uniformly, and that we have 101 free buffer frames.

1. Which algorithm would you use for computing Q?
2. Describe in detail the algorithm chosen.
3. Tell which is the cost of executing the algorithm in terms of number of page accesses.

Use hashbased algorithm and we can do it in two passes cause $\frac{500000}{50} \leq (M-1)(M-1)$.
First pass: distribute with hash function tuples in $M-1$ buckets in secondary storage.
Second Pass: one bucket at time, read pages and store in buffer one tuple for each owner value (with counter)
When done, write on result. Cost: $3 \cdot \frac{500000}{50} = 3 \cdot 10000$

In the two-pass algorithms based on sorting for duplicate elimination, we assumed that we know a priori the number of buffer frames available, and that such number never changes during execution of the algorithm. In reality, the number of buffer frames available may change during the execution of the algorithm.

Tell how would you change the algorithm in order to cope with this problem.

If the number of frames decreases after producing one sublist then in second pass we will have more sublists than available frames (with different size).
To solve this we can recursively choose to sort-merge together K sublists ($K <$ current M)
Stop when we have a number of lists that fit in buffer ($\leq M-1$) and continue as always.
Cost ofcourse increases.

INDEX-BASED ALGORITHMS

Algorithms that use indexes in one-pass and nested-loop schema.

An index is said to **conform** to attr op value if:

- it is a tree index, the search key is attr and op is $<, \leq, <>, \geq, >$
- it is a hash index, the search key is attr and op is $=$

Prefix of search key: initial non-empty segment of attributes for a composite search key

eg. $\langle a,b,c\rangle \rightarrow \langle a\rangle$ or $\langle a,b\rangle$, $\langle a,c\rangle$ and $\langle b,c\rangle$ not!

An index is said to conform to the conjunction (att op value) and (att op value) and ... if:

- it is a tree index and exists a prefix P of search key s.t. for each attribute in P there is a term of the form (att op value) in the conjunction (= primary terms)
- it is a hash index and for each attribute of search key there is a term (att = value) in conjunction

eg. hash index with search key $\langle r,b,s\rangle$

→ (r = "J" and b = 5 and s = 3) ✓ conform

→ (r = "J" and b = 7) ✗ not conform s missing

eg tree index with search key $\langle r,b,s\rangle$

→ (r > "J" and b = 7 and s = 5) ✓ conform

→ (r = "H" and s = 9) ✓ conform prefix $P = \langle r\rangle$

→ (b = 8 and s = 10) ✗ not conform $\langle b,s\rangle$ not a prefix

SELECTION

Special one pass algorithms, in general more efficient than classical ones.

Select * from R where <att op value>

Case 1: att is a key of relation, op is $=$ and we have index on that att

- hash index conforms to the condition and we can use it. Cost: 1 (2 if one access for bucket and one for overflow page)
- tree index conforms to the condition and we can use it. Cost: we refer to equality search cost, if we dont know # pages in leaves or for out we assume 3-4 page accesses

Case 2: att not key, op is $=$ and we have weakly clustering hash index on att.

Weakly clustered hash ⇒ given value V for attribute, the tuples with that v will be is as few pages as possible (1 or 2).

T(R) = # tuples of R. V(R,A) = # distinct tuples in projection of R on attribute A.

If the $=$ condition is on A and there is a hash index on A, then the index conforms to the condition

Selection cost is (1 or 2) + $B(R)/V(R,A)$ because $T(R)/B(R)$ is the average size of each page and $T(R)/V(R,A)$ is the average number of tuples with some value of A ⇒ $(T(R)/V(R,A))/(T(R)/B(R)) = B(R)/V(R,A)$ is the average number of accesses

Case 3: att not key, op is $=$ and we have a clustering tree index on att.

clustering index → data file ordered same as index

We have to compute the number B of leaves required. → average cost = $\log_{F\cdot 1.5} B + B(R)/V(R,A)$.

If we dont know pages in leaves or for out we assum (3 or 4) + $B(R)/V(R,A)$

Case 4: att not key, op is $=$ and we have nonweakly index on att.

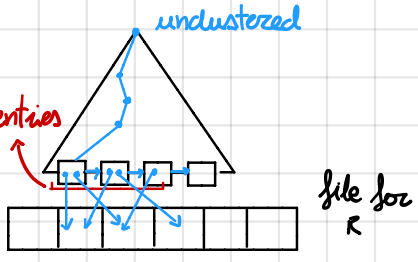
nonweakly clustering → each index entry can point to different page and cost could be one page per record.

eg. R(A, B, C)

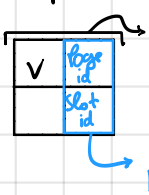
Select *

from R

where A = 10



To improve this remember that:



⇒ sort all data entries with value v (10 in our case) according to page id → different page only if all pages before already read/skipped → avoid multiple page loading.

Case 5: condition involves one attribute, op is < or > or range equality and we have clustering index on att. Analysis similar to tree based index

Case 6: condition is complex

condition can be splitted in atomic condition using index-based selection if possible

PROJECTION

Suppose R(A,B,C) and query "Select B,C from R" with hash based index on R with search key A. In this case index is an additional structure that doesn't help us at all.

Instead suppose tree based index with search key <B,C>, now we can avoid to scan R and just follow the index cause leaves are less than data pages.

Note: search key must includes all attributes of projection!

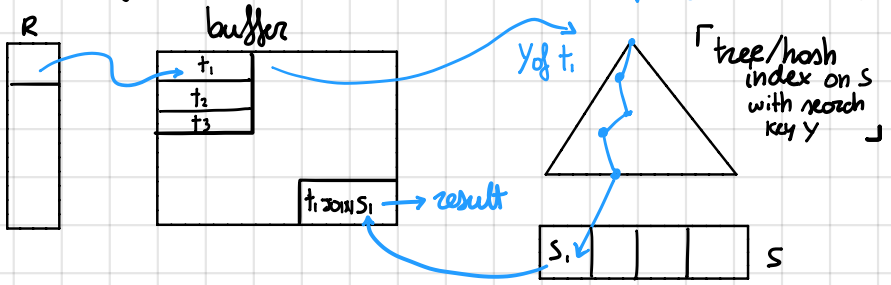
This is the so called index-only scan.

If the query was "Select distinct B from R" this kind of scan also let us to "remove" duplicates due to the fact that search key is <B,C> and so all similar Bs are close to each other.

Selection and projection algorithms can be used as special cases for: duplicate elimination, aggregation, union, intersection, difference and join.

JOIN

First case of index based join is index-nested loop algorithm. eg. R(x,y) Join S(y,z)



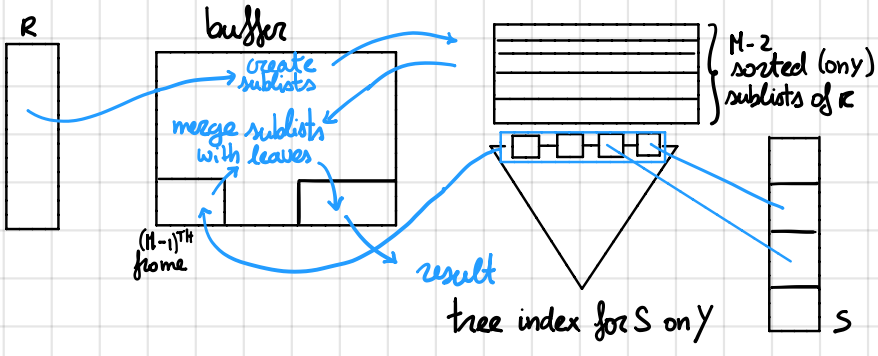
One page of R at time, for every tuple we find on index the pairing tuples of S on y and join them in output

Cost: B(R) + T(R)·(cost of using index for one value + B(S)/V(S,y))

↳ retrieve as many pages of S with given y if y not the key of S

We should avoid this case due to its high cost, but not if R << S and V(S,y) is high (eg y key of S)

The second case assumes that at least one relation has a sorted index (tree or hash).

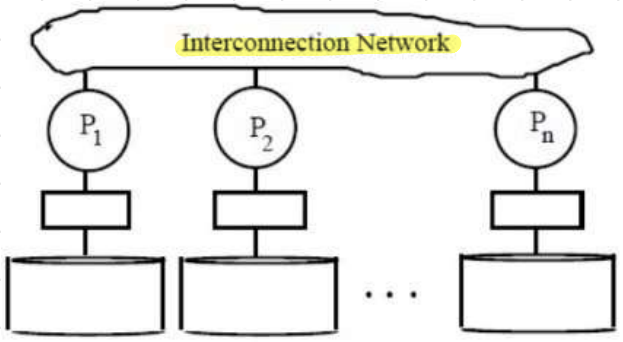


First compute sorted sublists of R on y and then merge them using leaves ("array of leaves") as the (M-1)TH sublist

If we have sorted index for each relation then we talk about zig-zag join cause we jump from on index to the other (and not between relations)

PARALLEL ALGORITHMS

We refer to the "shared-nothing" architecture where all processors have their own memory and secondary storage and dont share any resource.
Communication and data sharing between processor is done through the communication network
Communication cost is so low that we can ovoid to take it into account.



Horizontal data partitioning: R is split into P chunks stored in P nodes. We case use round robin (tuple t_i to chunk (i mod P), range based partitioning on attribute A (tuple t to chunk i if $v_{i-1} < t.A < v_i$) or hash based partitioning.
Hash based partitioning works with hash function based on ALL values of tuple.
eg. if we want P buckets we might convert each component of tuple t somehow to integer [0,P-1], add it for all components, divide by P and take remainder as bucket number for t.

SELECTION

If tuples are distributed evenly among the P processor's disks then the elapsed time on parallel databases is B(R)/P in all case of data partitioning
What change is who does the job: RR → all servers, Range-based → one server only, hash-based → if h works on A then one server for equality selection, all servers otherwise or for range selection
Note: page accesses are still B(R) in total but since R is divided in P chunks scanned in parallel the time needed is B(R)/P

PROJECTION

Similar to selection. B(R)/P elapsed time in all cases of partitioning. Projection cant change distribution of tuples.

DUPLICATE ELIMINATION

If we use hash-based partitioning all duplicates are in same processor. In this case if we can use one pass algorithm for each processor then elapsed time is B(R)/P. Otherwise we consider (max cost)/ P

GROUPING

Again if we use hash-based partitioning then some group tuples will be in some bucket. → B(R)/P with one pass algorithms (if hash function depends only on grouping attributes)

BINARY OPERATORS

We have to use same hash function for both relation (a new one if necessary) to compute union, difference and intersection

UNION / DIFFERENCE / INTERSECTION

Some tuple from different relation will go in same bucket and then we can easily unify (or diff...) in parallel.

JOIN

Same as before but hash function must depends only on joining attribute.

Performances in terms of page accesses and CPU times are lower then single processor case but elapsed time instead is better in parallel case ('/P of single processor)
Suppose we have R join S with R and S in same processor. We now apply the distributing hashing function.

To do this we must read $B(R) + B(S)$ pages. Then we must "ship" them to right bucket
→ $[(B(R) + B(S))] \cdot (P-1) / P$.

Two-pass sort-merge sort at each processor:

- $B(R) + B(S)$ for reading and hashing tuples (ignore shipment cost)
- $[B(R) + B(S)] / P$ access per processor to account for the writing in local disk during distribution
- $3[B(R) + B(S)] / P$ for local join at each processor

→ Total: $3[B(R) + B(S)]$ for uniprocessor to $B(R) + B(S) + 4[B(R) + B(S)]/P = [B(R) + B(S)] \cdot (P+4)/P$

Consider the relation CONCERT(band,year,city,cost,people) that stores information about concerts, with the band that gave the concert, the year when the concert was held, the city where the concert was held, the cost of the concert, and the number of people that attended the concert. The relation occupies 300.000 pages, each of 800 KB. We assume that all fields and pointers in any record have the same size of 10 KB, independently of the field type. There is a tree-based index on CONCERT with search key ⟨band,year,city⟩, using alternative 2. Consider the query

select band,city
from CONCERT

and tell which algorithm you would use for executing the query, and how many page accesses such algorithm needs for computing the answer in the following cases: (1) the index is a B+-tree, and is sparse, (2) the index is a B+-tree, and is dense, (3) the index is an ISAM index, and is dense.

In case 1 the index is sparse so we cant use the index-only scan → scan the relation directly → cost 300 000 page accesses

In case 2 we can use the index (one entry for each record). Compute the number of leaves and how many page for each: each page = 800 kb and each field = 10Kb → every page has space for 80 values
→ every leaf has space for 80/4 = 20 data entries (4 because pointer + 3 search key values)
→ "67% rule" → 13 data entries for each leaf.
Relation has 800/50 (50 = 5 field · 10kb) = 16 tuples per page → 16 · 300.000 = 4800000 tuples total
→ 4800000/13 = number of leaves = 369.230
We can scan the index with search key band and city with 369.230 page access > 300.000
Scanning relation is still better
In case 3 we have only 1 difference: no "67% rule"! → 4800000/20 = 240000 leaves < 300 000 → one pass algorithm index only scan

Consider the relation
WORK(employeeNo,lastName,firstName,birthYear,company,salary)
stored in a heap file with 850.000 tuples. Assume that the size of each value is 10 Bytes, the size of each page is 600 Bytes, and the number of different companies in the relation WORK is no more than 2.700. Suppose we have 300 free buffer frames available.

- Consider the query
  select max(salary) from WORK group by company

  computing, for each company, the maximum salary given to its employees. Describe in detail the algorithm you would use to compute the answer to the query and the cost of the execution of such algorithm in terms of number of page accesses.

- Consider the query
  select * from WORK order by employeeNo

  and describe in detail the algorithm you would use to compute the answer to the query and the cost of the execution of such algorithm in terms of number of page accesses.

Query 1

1 tuple = 6 · 10 bytes → 600/60 = 10 tuples per page → 850000/10 = 85000 pages
Even though we have only 300 free buffer we could still use one pass algorithm because we only have to store one tuple for each group (company + max salary) → 2700 · 2 · 10 bytes = 54000 bytes in buffer
54000/600 = 90 frames (<300 ok)
→ cost 85000 page accesses = B(WORK)

Query 2

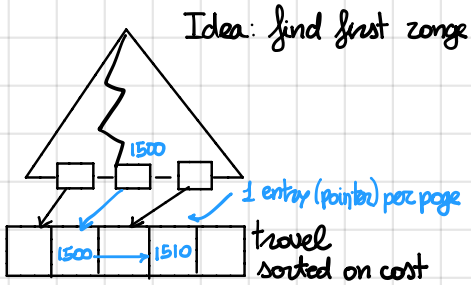
We just have to sort the relation → multipass-merge sort algorithm → 85000 ≤ 300 · 299 → 2 passes
→ 85000 · 3 = 255 000 page accesses
(3 = 2 + 1)

Consider the relation TRAVEL(code,person,nation,cost) that stores in a sorted file information about travels of people, with the code of the travel, the person who traveled, the nation visited, and the cost of the travel. The relation has 3.000.000 tuples, stored in 300.000 pages, and has 10.000 different values in the attribute "cost", uniformly distributed in the relation. We assume that all fields and pointers have the same length, independently of the attribute. There is a sparse, clustering B+-tree index on TRAVEL with search key "cost", using alternative 2. Consider the query

```
select code, nation
from TRAVEL
where cost > 1500 and cost <= 1510
```

and tell both the algorithm and the cost (in terms of the number of page accesses) for computing the answer to the query.

Note: sorted on cost since the clustered index is on "cost"



Idea: find first range condition and then scan until the second one on data file

① Compute #leaves: depends on how many data entries each leaf stores → sparse = one data entry per page = 300000 of relation

The data entries we want are composed by two values: key and pointer that is half the space of a tuple (4 values)

So 300000/300000 = 10 tuples per page → 20 data entries per leaf

"67% rule" → 13 data entries in each leaf

→ 300000/13 = 23.077 leaves

② Fan-out = $\frac{20+10}{2}$ = 15

③ How many pages to scan: 3000000/10000 = 300 tuples with same cost → we need 10 different costs:

1500, 1501 ... 1510 → 300 · 10 = 3000 tuples → $\frac{3000}{10}$ = 300 pages (↳ entries per page)

Cost: $\log_{15} 23.077 + 1 + 300 = 305$ (↳ reach first leaf)

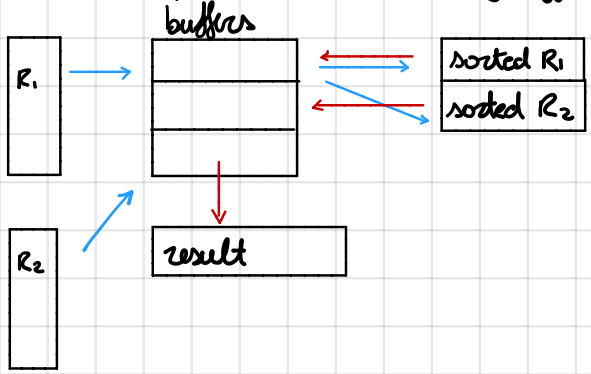
Suppose that we have only 3 buffer frames available, and we have to answer the query

```
select * from R1
minus
select * from R2
```

where R1 and R2 have 5000 and 9000 pages respectively, and are stored in heap files that may contain duplicates, and the result is without duplicates. Which is the most efficient algorithm for answering the query? Which is the cost of the algorithm in terms of number of page accesses?

i.e. bag difference with duplicates elimination

Consider multipass algorithm for bag difference and during it eliminate duplicates.



① Sort both with two-way sorting algorithm

② Merge them eliminating duplicates.

Cost: sort R_1 + sort R_2 + merge

$= [2B(R_1) \cdot \log_2(B(R_1)+1)] + [2B(R_2) \cdot \log_2(B(R_2)+1)] + [B(R_1)+B(R_2)]$

= 424000 page accesses

QUERY PROCESSING

SQL query → Compilation → Physical query plan → Query execution → Result

Compilation: query parsing → query optimization; Physical query plan: executable but not optimized

Parsing: parse (generate a parse tree to check errors), convert (converted to relational algebra tree)
Optimization: select logical query plan applying laws/operations
Select Physical query plan: estimate result size, consider possible solutions, estimate cost, select best one
Execute: give the results, but can also retrieve statistics for database management for next queries.

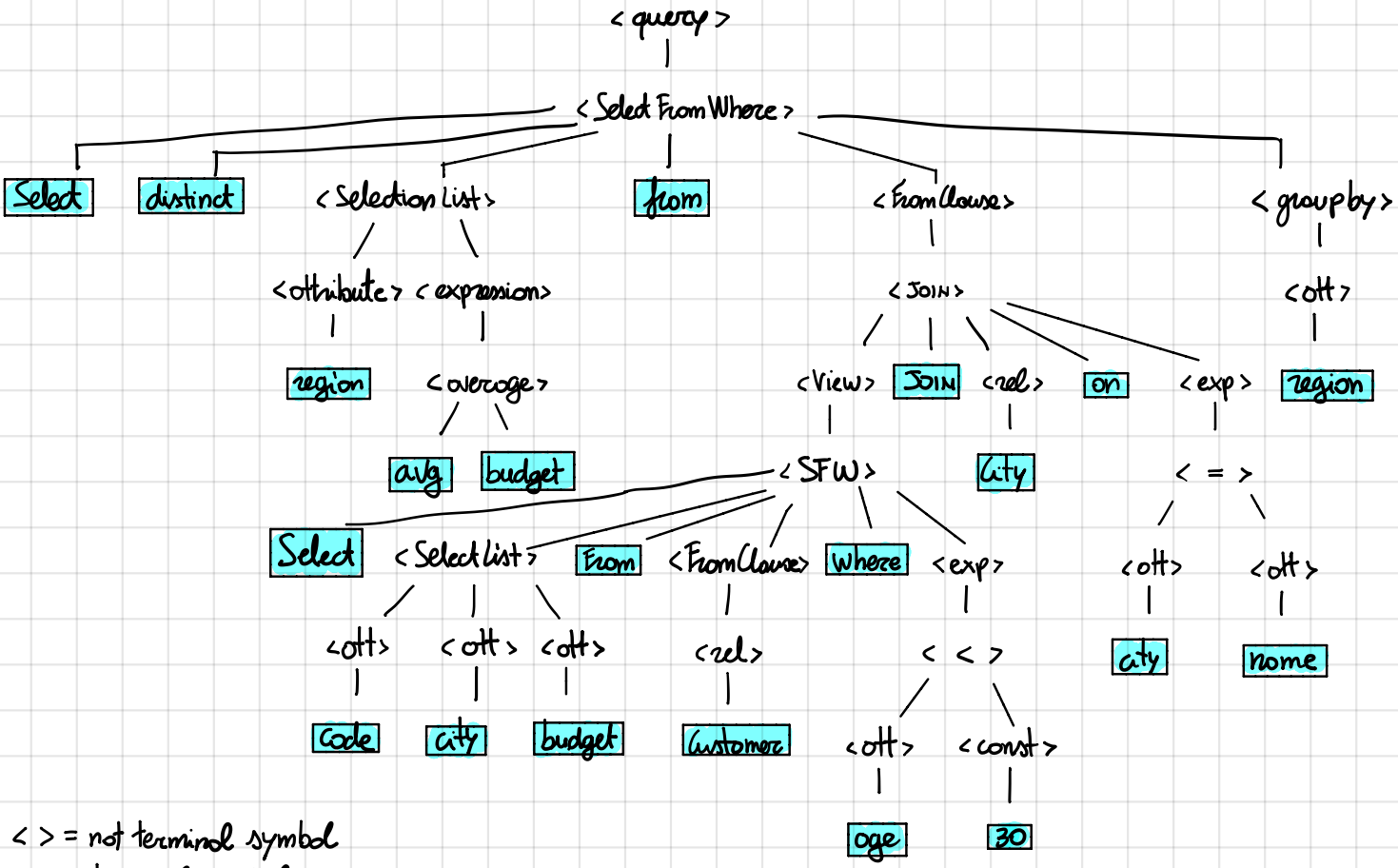
PARSING

Parse: ① sql query analyzed and represented as sparse tree ② the tree is pre-processed with following goals:
- views substituted by their definition
- every element of query should be valid element for schema
- resolve ambiguities of attributes
- type checking

Convert: transform to an extended relational algebra expression tree with internal node constituted by operators (union, intersection, difference, selection σ, projection π, cartesian product $\times$, JOIN $\bowtie$, $\ltimes$, duplicates elimination δ, grouping γ and sorting)

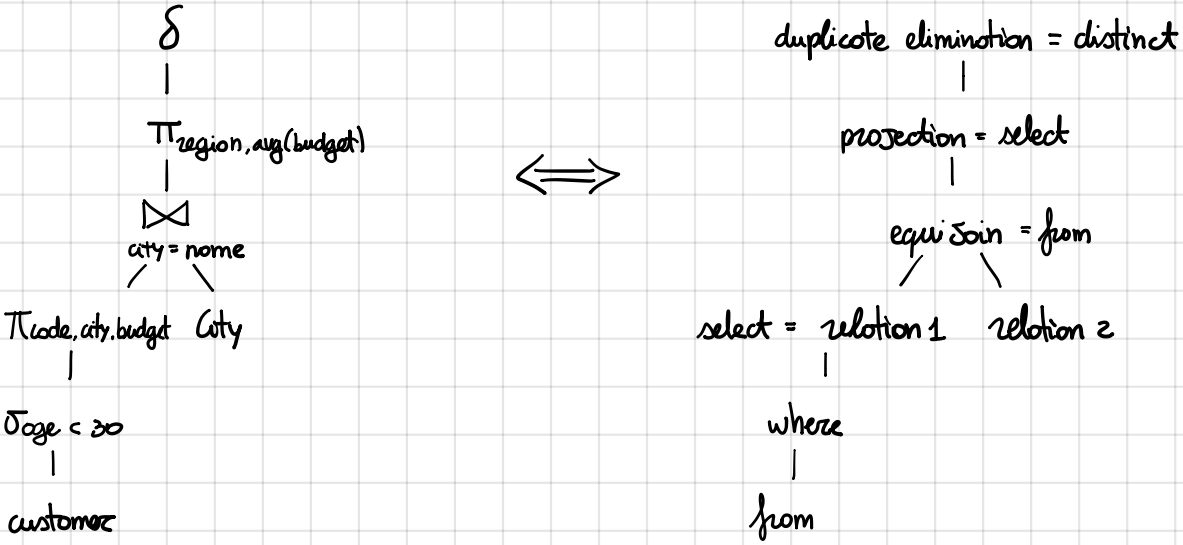
eg. [Customer (code, city, age, budget), City (name, region),] relations
[create view YCust as
Select code, city, budget View
from Customer where age < 30]

Query: Select distinct region, avg(budget) from YCust JOIN City on city = name groupby region



< > = not terminal symbol
▭ = terminal symbol

Relational Algebra Tree:



SELECT LOGICAL QUERY PLAN

Apply relational algebra rules: "equivalence-preserving" rules

- $R \bowtie S = S \bowtie R$
- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- $R \times S = S \times R$
- $(R \times S) \times T = R \times (S \times T)$
- $R \cup S = S \cup R$
- $R \cup (S \cup T) = (R \cup S) \cup T$
- $R \cap S = S \cap R$
- $R \cap (S \cap T) = (R \cap S) \cap T$
- $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$ Only for sets
- $\sigma_{P_1}[\sigma_{P_2}(R)] = \sigma_{P_2}[\sigma_{P_1}(R)]$ In this way we can move the operator in the tree
  - ↳ $\sigma_{P_1 \vee P_2}(R) = \sigma_{P_1}(R) \cup_s \sigma_{P_2}(R)$ Only for sets
- $\pi_{XY}(R) = \pi_X[\pi_Y(R)]$ Only if X subset of Y otherwise π_Y could extract some X tuples.
- $\pi_X[\sigma_P(R)] = \{\sigma_P[\pi_X(R)]\}$ if Z subset of X $\pi_X[\sigma_P(R)] = \pi_X\{\sigma_P[\pi_{XZ}]\}$ otherwise

  (X subset of R attributes, Z attributes in predicate P = subset of R attributes)

eg $R(A,B,C,D,E)$ $X = \{E\}$ $P: (A=3) \wedge (B = \text{"cat"})$

$\pi_X\{\sigma_P(R)\}$ vs $\pi_E\{\sigma_P\{\pi_{ABE}(R)\}\}$ which better?

↳ this pushes projection down the tree

But with second tree we can't use index on single attributes → we need a new one on "ABE".

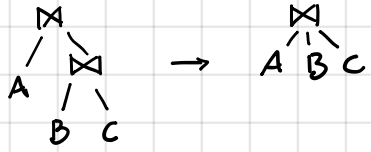
- $\sigma_P(R \times S) = R \bowtie_P S$
- $\sigma_P(R \bowtie S) = [\sigma_P(R)] \bowtie S$
- $\sigma_q(R \bowtie S) = R \bowtie [\sigma_q(S)]$
- $\sigma_q(R \bowtie S) = \sigma_q(R) \bowtie \sigma_q(S)$
- $\sigma_P(R \cup S) = \sigma_P(R) \cup \sigma_P(S)$
- $\sigma_P(R - S) = \sigma_P(R) - S = \sigma_P(R) - \sigma_P(S)$
- $R \bowtie_E S = \sigma_E(R \times S)$
- $R \bowtie S = \pi_L(\sigma_E(R \times S))$ E = condition that equates each pair of attributes from R and S with same name; L = list of attributes that contains union of attributes of S and R
- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$
- $\delta(R \bowtie_E S) = \delta(R) \bowtie_E \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$
- $\delta(R \cap_B S) = \delta(R) \cap_B \delta(S)$
- $\delta(R \cap_B S) = \delta(R) \cap_B S$
- $\delta(R \cap_B S) = R \cap_B \delta(S)$

Guidelines

1) Repeat if and until possible
   - push selections over projections (split selection if possible)
   - group selections
   - push selections over cartesian product

2) Eliminate useless projection eg $\pi_X(R) = R$ if X = all R attributes
3) Push projections or add new ones avoiding to make indexes useless
4) Remove duplicates or move them if necessary
5) Combine selection with product to obtain a more convenient equijoin.

For n-ary operators:
each node with some associative/commutative operator (eg union/intersection/join) is grouped as child of some node. This new node can be computed whenever the system thinks is time to.

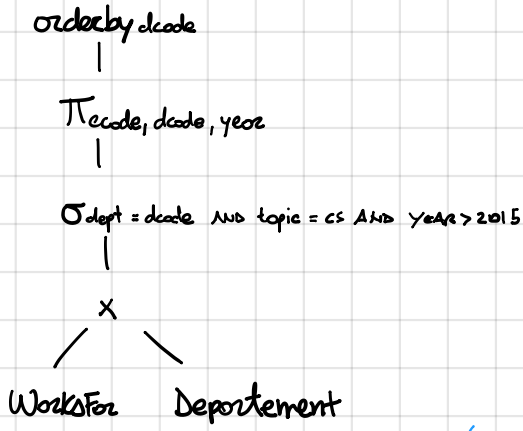
eg 

We have a buffer with 100 frames available, and the relations: WorksFor(ecode, dept, year, salary), stored in 80.000 pages (with 12 tuples each), and Department(dcode, topic), stored in 12.000 pages (with 24 tuples each), where each value in dept appears in dcode, each employee is associated with no more than 100 departments, year has 40 values (from 1973 to 2020) uniformly distributed in the tuples of WorksFor, and topic has 20 values uniformly distributed in the tuples of Department.

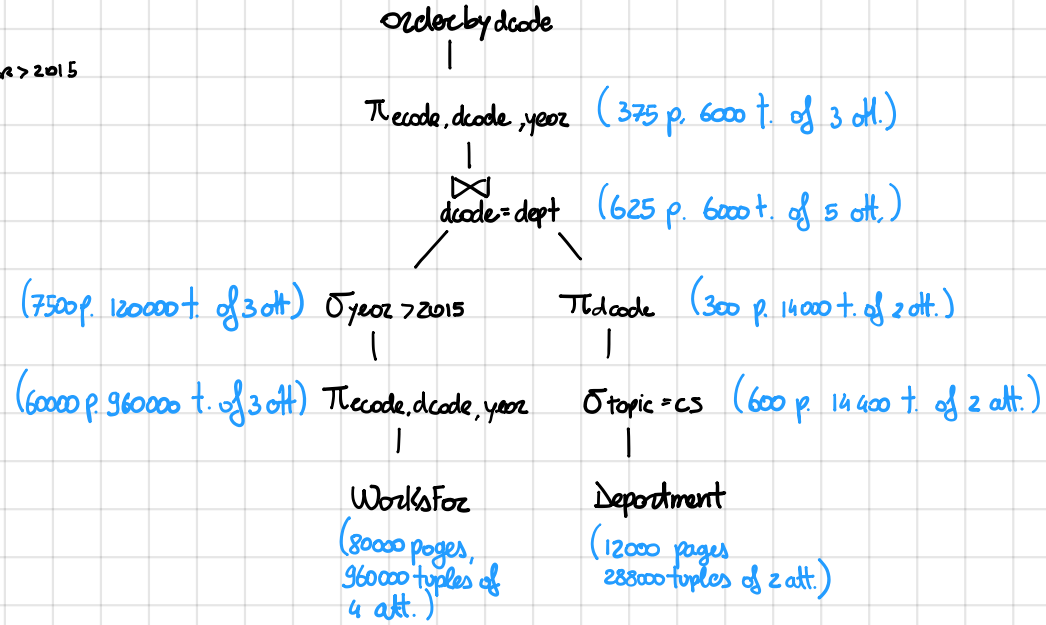
Consider the following query

```
select ecode, dcode, year
from WorksFor, Department
where dept=ecode and topic = 'CS' and year >= 2015
order by dcode
```

1) show the logical query plan corresponding to the query
2) show the logical query plan obtained by applying suitable relational algebra equivalence rules

1) 

2) • $\times$ + $\sigma_{dept = dcode}$ → equijoin $\bowtie_{dept = dcode}$
• push bottom $\sigma_{topic = cs}$ and $\sigma_{year > 2015}$
• push bottom $\pi_{dcode, ecode, year}$



SELECT PHYSICAL QUERY PLAN

1) Estimate result size
2) Consider several physical query plan alternatives
3) Estimates their costs
4) Pick the best one

In physical plan we do transformations that are effective with respect to current volume of data (statistics), not indipendently anymore.

Note: $S(R)$ = # bytes in each tuple of R, $V(R,A)$ = distinct value for attribute A in R

CARTESIAN PRODUCT

$W = R1 \times R2$

$T(W) = T(R1) \times T(R2)$

$S(W) = S(R1) + S(R2)$

PROJECTION

Number of tuples doesn't change but size of each may decrease → less pages → calculate factor f by which relation shrinks $(f \times T(R))$

SELECTION

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

$W = \sigma_{A=dog}(R)$

$T(W) = T(R)/V(R,A)$ considering uniform distribution

$= 5/3$

Inequality: $\sigma_{z>val}$

- Approach 1: assume 1/3 of tuples satisfy it → $T(R)/3$
- Approach 2: estimate values in range min and max

eg min = 1 max = 20 $\sigma_{A>15}$ $V(R,A) = 10$

f = fraction of range = $\frac{20-15+1}{20-1+1} = \boxed{6/20}$ $= \frac{max - val + 1}{max - min + 1}$

$T(W) = f \cdot V(R,A) \cdot T(R)/V(R,A) = f \cdot T(R)$

Disequality: $\sigma_{z \neq val}(R)$

- Approach 1: assume is rare to satisfy equality → $T(W) = T(R)$
- Approach 2: $T(R)/V(R,z)$ tuples fail the condition → $T(W) = T(R) - T(R)/V(R,z)$

With multiple condition (AND) just treat them as a cascade of simple selections

With OR an (over) estimation is the sum of number of tuples that satisfy each.

JOIN

We use two assumptions (valid if JOIN on foreign key)

1) Containment of value sets: if $V(R1,A) \leq V(R2,A) \Rightarrow$ every A value in $R1$ is also in $R2$ $(R1[A] \subseteq R2[A])$
2) Preservation of value sets: if A is an attribute in R but not in S then $V(R \bowtie S, A) = V(R,A)$ i.e. A doesn't loose values

Case 1: $R1$ and $R2$ have NO common attributes → $T(W)$ = same as cartesian product

Case 2: single attribute in common → remember assumption 1 (eg $V(R1,A) \leq V(R2,A)$)

1 tuple of $R1$ matches $T(R2)/V(R2,A)$ tuples in $R2$ → $T(W) = T(R2) \cdot T(R1)/V(R2,A)$

In general: $T(W) = T(R2) \cdot T(R1) / max[V(R1,A), V(R2,A)]$

Case 3: multiple attributes in common (also valid for equijoin) → estimate number of tuples multiplying $T(R1)$ and $T(R2)$ and then dividing by the larger of $V(R1,y)$ and $V(R2,y)$ for each y in common

eg. $R(A,B,C)$ $S(D,E,F)$ on $B=D$ $C=E$ [equijoin]

$T(R) = 1000$ $T(S) = 2000$ $V(R,B) = 20$ $V(S,D) = 50$ $V(R,C) = 100$ $V(S,E) = 60$

$T(W) = T(R \bowtie S) = 1000 \cdot 2000 / (50 \cdot 100) = 400$

$\max(B,D)$ $\max[C,E]$

If we have n-ary join always start with product of number of tuples in each relation, the divide for all the $V(R,X)$ for each X appearing at least twice. (but the least)

eg $R(a,b,c)$ $S(b,c,d)$ $U(b,e)$ $W = R \bowtie S \bowtie U$

$T(R) = 1000$ $T(S) = 2000$ $T(U) = 5000$

- $V(R,A) = 100$ $V(R,B) = 20$ $V(R,C) = 160$
- $V(S,B) = 50$ $V(S,C) = 100$ $V(S,D) = 400$
- $V(U,B) = 200$ $V(U,E) = 500$

$\longrightarrow$

$T(W) = (1000 \cdot 2000 \cdot 5000) / (50 \cdot 200 \cdot 160) = 6250$

$\max$ of C

$\max(S)$ of B

Theta-join can be treated as selection following a product with with these observations:

- equality condition estimated using method for natural join
- inequality of type R.a < S.b can be treated as R.a < constant $\rightarrow$ selective factor of 1/3. Similar for disequality

OTHER OPERATORS

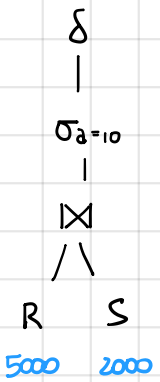
- Bag union = sum of size of the two relations
- Set union = good estimated as largee one + half of smaller one
- Intersection: half of the smaller
- Difference: $R - S \rightarrow T(R) - T(S)/2$
- Duplicate elimination: $\min\{T(R)/2, \text{product of all } V(R,a_i)\}$
- Grouping and Aggregation: $\min\{T(R)/2, \text{product of } V(R,a_i) \forall \text{ grouping attributes}\}$

Let R(a,b) and S(b,c) two relations, with

T(R) = 5000	T(S) = 2000
V(R,a) = 50	V(S,b) = 200
V(R,b) = 100	V(S,c) = 100

```
select distinct *
from R join S
where R.a = 10
```

- compute the logical query plan corresponding to the query expression, and analyze possible alternative logical query plans,
- show for each logical query plan an estimation of the number of tuples in all the nodes except for the root.

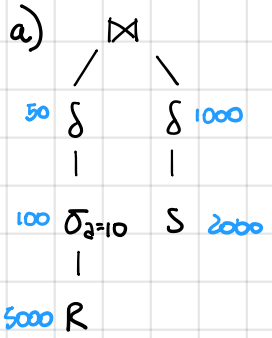


1000 = $T(\text{JOIN}) / V(\text{JOIN}, A)$

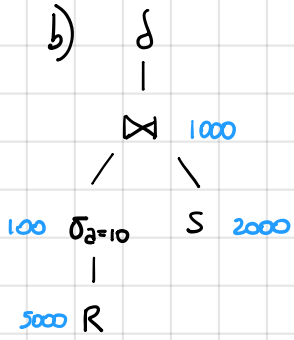
50000 = $T(R) \cdot [T(S) / \max\{V(R,b), V(S,b)\}]$

We can push selection:

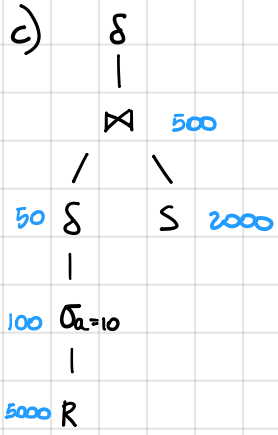
a)



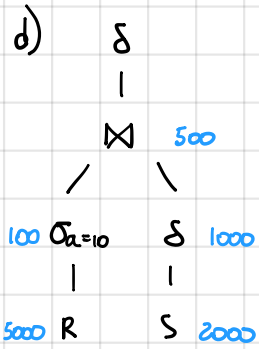
b)



c)



d)



How to find the correct physical query plan from the logical one?
Select:
1) Order and grouping for associative-commutative operators (eg JOIN)
2) An algorithm for each operator (or group of them)
3) Additional operators / transformation
4) The way in which each argument pass from an operator to the following.

⇒ between all possible plans, pick the "cheaper"
Problem: TOO MANAY POSSIBILITIES → use HEURISTICS
1) Use indexes for selection of form A=c on stored relation if available
2) If selection is A=c + other conditions, use index if available and then apply new operator filter (a selection for physical plan)
3) If R in the join has an index on joining attributes then check if its good to use index-based join with R in inner loop
4) If R in JOIN is sorted in JOIN attribute then prefer a join algorithm based on sorting
5) When computing union/intersection of 3 or more relations, group the smaller ones first
6) Apply specific algorithms for deciding JOIN order.

JOIN ORDER

If binary → choose smaller relation as left argument
If more than 2 operands: greedy algorithms which makes one decision at time with no backtracking
The greedy algorithm should select a left-deep tree where greediness is based on idea that we want to keep the intermediate relations as small as possible at each level of tree. (NOT ALWAYS OPTIMAL BUT WHO CARES)
G.A.:

BASIS → start with pair of relations estimated to join in smallest relation, this will be the current tree
INDUCTION → Find a new relation that joined with current tree forms the smallest relation. New current tree will have old current tree as left argument and this relation as rigth argument. Repeat.

Now we have decide if intermediate results should be written in secondary storage (materialized) or not (pipelined) and then build the final tree with appropriate notation for the query execution engine.

PIPELINING VS MATERIALIZATION

Unary operators are treated by pipelining:
eg Select A from R(A,B,C,D,E) where B=3 → We would never store the temporary results!
Binary operators instead may need to store part of relations (or all of them).

PHYSICAL PLAN TREE NOTATION

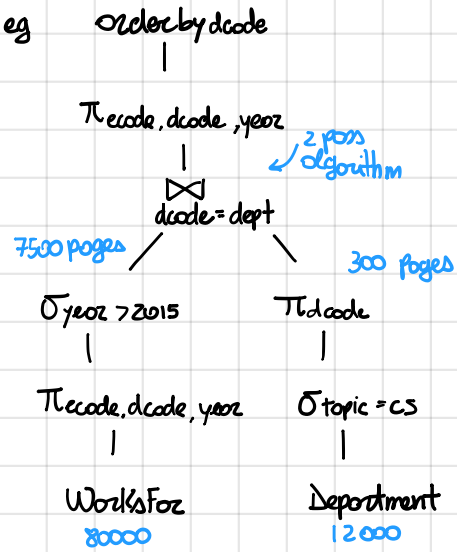
- For each internal node indicate the algorithm used in it
- Indicate materialized intermediate relation double line crossing the edge between relation and consumer. eg =
- Indicate in leaves how we access relations of database

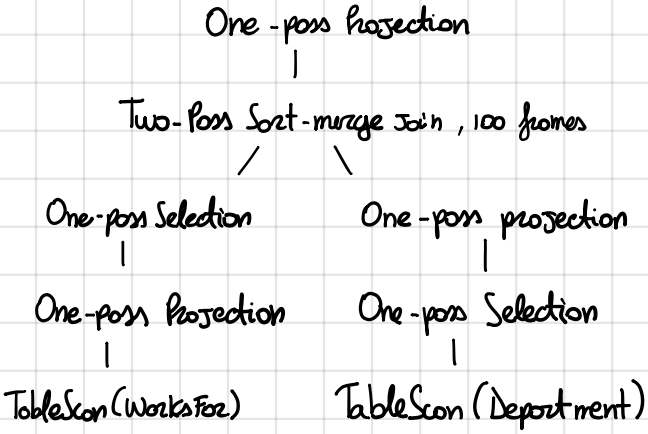
↓

1) TableScan(R): scan the relation R
2) SortScan(R,L): R scanned and sorted according to attribute list L
3) IndexScan(R,A,C): R accessed with index on attributes A, where the index follow condition C
4) IndexScan(R,A): R is retrieved by scanning index on attributes A (index-only scan)

- For selection use Filter (c)
- For sorting intermediate relations use Sort(L) where L = list of sorting attributes
- X-pass Sort Partition(A) with F = #available buffers indicate first phase (compute sublists) of a x passes algorithm based on sorting with search key A.
- X-pass Hash Partition(A) as before but for hashing
- Merge(op) indicates last phase of multipass algorithm, merge sublists/buckets, according to operator op

eg





Cost of Physical query plan: 80000 + 2·7500 + 12000 + 2·300

Note: General "cost rule" of two pass algorithm is 80000 + 12000 + 3·(7500 + 300) but the final cost is x2

Why? Suppose we materialize the relations as follow

Two-Pass Sort-merge Join, 100 frames

One-pass Selection ⋮ One-pass projection ⋮

materialization

→ Read WorksFor (TableScan) + Write One pass Selection (7500)
+ Read Department (TableScan) + Write One pass Projection (300)
+ 3·(7500 + 300)
two pass Sort merge Join

In other words: 80000 + 12000 for scanning Work and Department
7500 + 300 for computing sublists
7500 + 300 for the last phase of merging

If we use materialization we need one more pass to write on disk

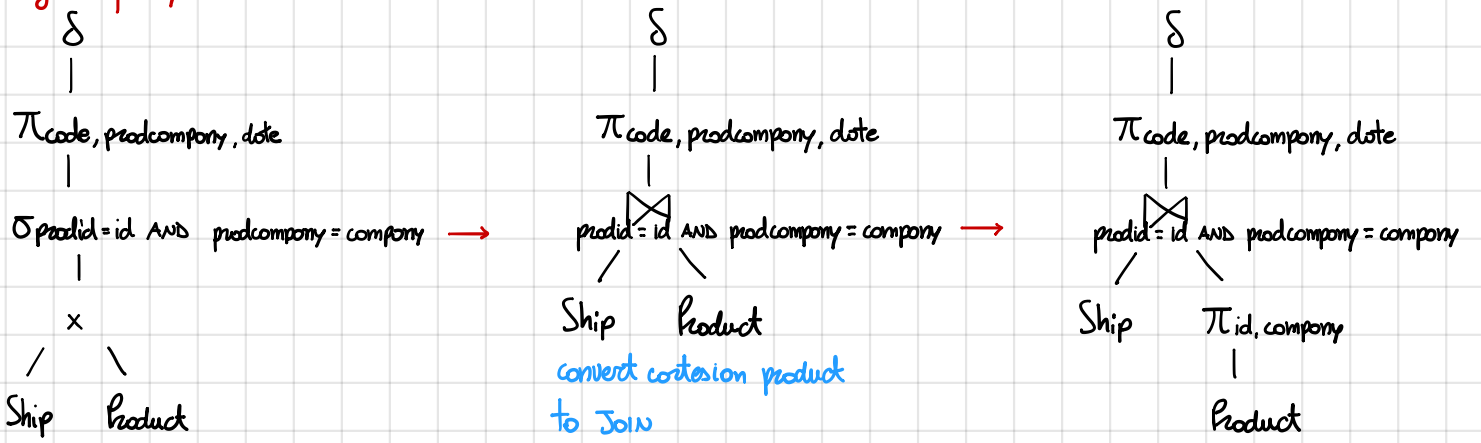
- Ship(code,prodid,prodcompany,date), 200.000 pages (10 tuples per page)
- Product(id,company,name,type,year), 850.000 pages with a tree-based index on <id,company> with 10.000 leaves
- we have 500 buffer frames available

You are asked to:
- show the logical query plan
- show the physical query plan you would select

Consider the query:

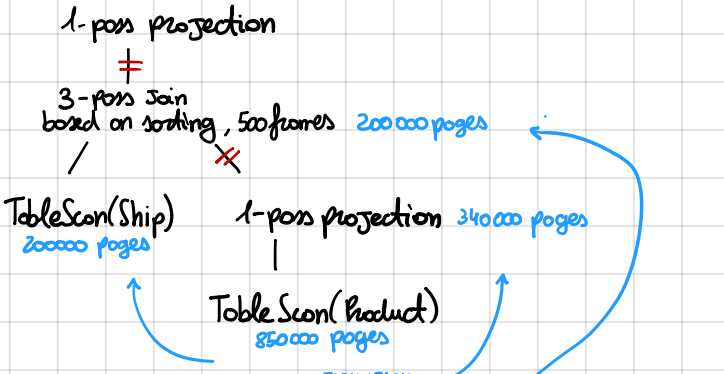
```
select distinct code, prodcompany, date
from Ship, Product
where prodid = id and prodcompany = company
```

Logical query Plan



Physical query Plan(s)

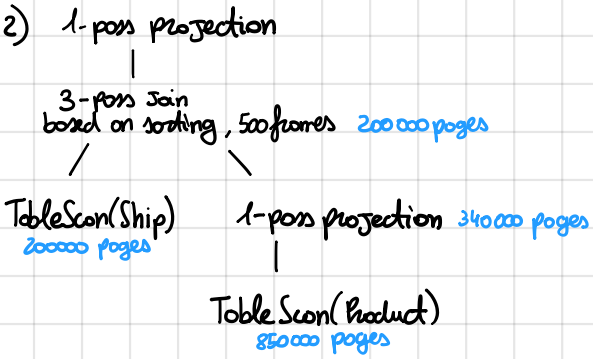
1)



Cost: 850000 + 340 000 + 5 · 540000 + 200000 = 4 090 000 p.a.
(200K + 340K)

All materialized!

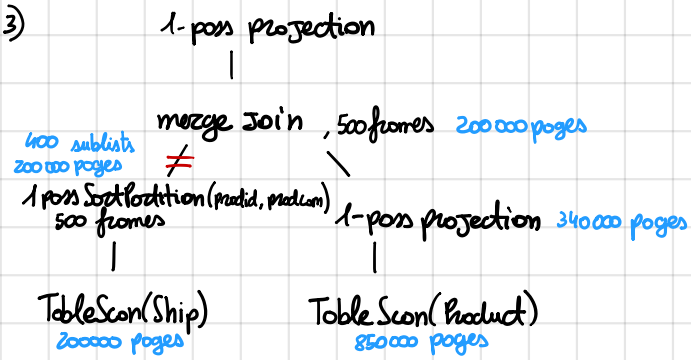
2)



Product not sorted on <id, company>

Cost: 1050000 + 4 · 540000 = 3210000 p.a.

3)

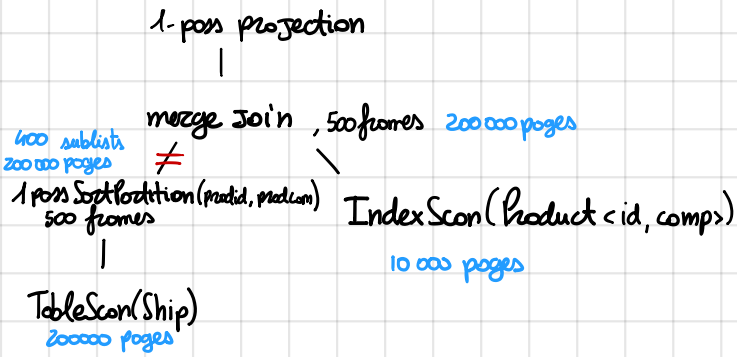


Product is sorted on <id, company>

Cost: 3·(200 000) + 850000 = 1450000 p.a.

↳ read Ship + materialization + reading of merge

4) by pushing projection we loose the possibility to use the index on product... so now we start from middle logical query plan



Cost: 2 · 200 000 + 210000 = 610 000 p.a.

BEST PLAN!

Given a binary relation R(A,B), and a unary relation S(C), the S-*portion* of R is the unary relation defined as follows:

$$\{a \in R[A] \mid \forall b\langle a,b\rangle \in R \rightarrow \langle b\rangle \in S\}$$

In other words, a value a is in the S-portion of R if a appears in the projection R[A] of R on attribute A, and every value b related to a by means of R appears in the only attribute C of S.

Assuming that R is stored in a file sorted on ⟨A, B⟩ with 50.000 tuples in 5.000 pages, S is stored in a heap file with 6.000 tuples in 400 pages, and the buffer has 3 frames available,

1. design an algorithm that, given relations R and S as specified above, computes the S-portion of R;
2. tell which is the cost of the algorithm written for item 1 in terms of number of page accesses.

1) 3 frames: load R (page by page) in one of them and for each page scan (page by page) S in a second frame. If find a value B which is not in S write corresponding A to result. At the end compute R - result.
2) Cost equal to nested loop algorithm = $5000 + 5000 \cdot 400$ (5000: outer, 400: inner)

Consider the relations Tournament(name,year,city,director,prize,winner), which contains 60 pages storing information about 3.000 tennis tournaments, and Player(code,lastname,yearOfBirth,cityOfBirth), which stores information about 150.000 tennis players. Assume that every attribute of every relation and every pointer has the same size, and that there is a B+-tree-based, primary, unclustered index on Player with search key code.

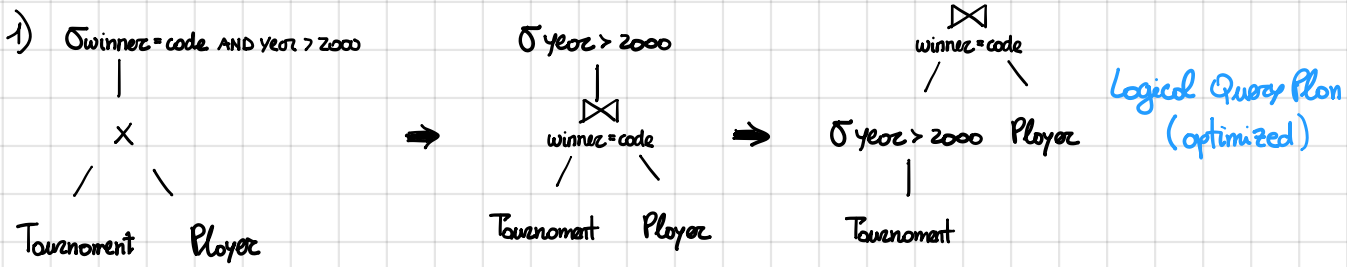
Consider the query

```
select *
from Tournament, Player
where winner = code and year > 2000
```

show the associated logical query plan, and, assuming that the statistics tell us that virtually all the tuples satisfies the condition year > 2000, and that you can only use three buffer frames (two buffers, and one needed for handling the output),

1. show the logical query plan associated to the query expression and the logical query plan chosen for the query evaluation;
2. tell which physical query plan would you use for computing the answer to the above query;
3. which is the cost of computing the answer to the above query using the chosen query plan.

1)



Logical Query Plan (optimized)

2) @ Simple Sort based JOIN (3 frames)

Filter year > 2000 (60) — TableScan(Player) (2000)

TableScan(Tournament) (60)

Cost: $60 \cdot [\log(60+1) + 2000[\log(2000)+1]$ (sorting) $+ 2000 + 60 = 26480$

Note: log base 2

b) Since Tournament has a small number of tuples we can use the index-based JOIN: we scan Tournament and for each tuple remove the ones with year ≤ 2000 and use index to find Player's tuples with winner = code. Only two buffer frames.

Index-based JOIN (3 frames)

Filter(year > 2000) (60) — IndexScan(Player, code, winner = code) (?)

TableScan(Tournament) (60)

Spoiler! The second one is the best physical query plan. How to find out? Compute number of page accesses to find a tuple of Player given the attribute code.

Compute number of leaves (assume alternative 2): need to store 150000 entries ⟨value, pointer⟩ → 50 tuples (of 6 attributes) of Tournament fit one page so 150 tuples of two attributes (value and pointer) fit in one page → 67% rule: 100 entries in one page → 150000/100 = 1500 leaves

Compute fan out: $\frac{\frac{150}{2} + 150}{2} = 112$ → Page accesses to reach correct entry = $\log_{112} 1500 = 3$ + 1 to access the data record

Cost: $60 + 3000 \cdot 4 = 12060$

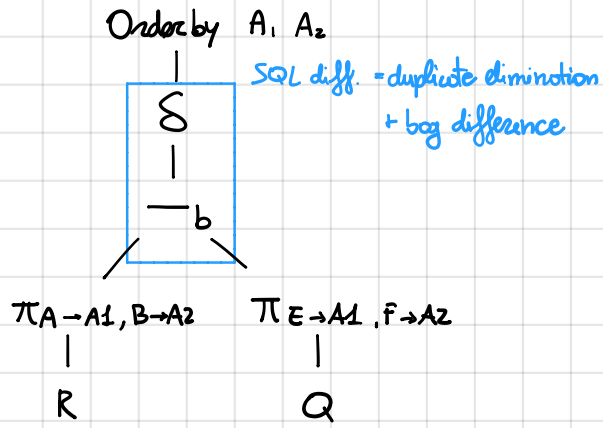
The SQL table R(A,B,C) has 1.400.000 tuples stored in a heap file, the SQL table Q(E,F,G,H,L,M,N) has 2.400.000 tuples stored in a heap file, and there is a B+-tree index on (E,F), where (E,F) is the key of Q.

We know that each attribute or pointer occupies 20 Bytes, the size of each page is 600 Bytes, and the buffer has 400 free frames. Consider the following SQL query (we remind the student that the minus clause in SQL computes the difference by eliminating duplicates, i.e., by taking the distinct rows of the first operand and returning the rows that do not appear in a second operand):

```
select T.A1, T.A2
from (select A as A1, B as A2 from R
      minus
      select E as A1, F as A2 from Q) as T
order by T.A1, T.A2
```

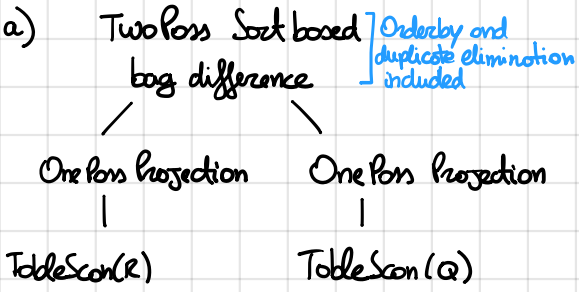
Describe in detail both the logical and the physical query plan, and tell which is the cost of executing the chosen physical query plan.

Logical Query Plan:



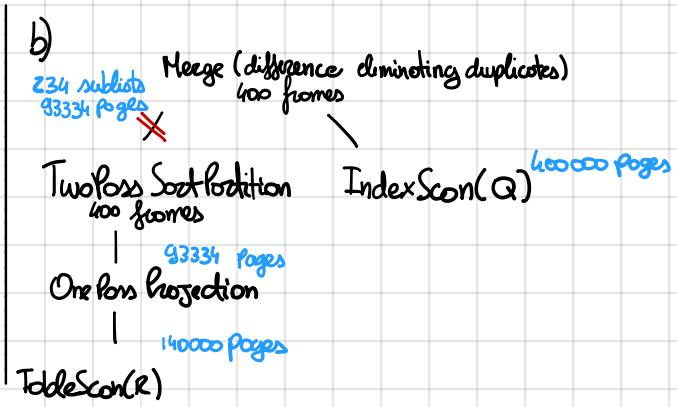
Physical Query Plan

a)



Good but high cost

b)



Explaination:

In R we have $600/(20 \cdot 3) = 10$ tuples per page $\rightarrow$ $1400000/10 = 140000$ pages for R

In some way we have 600 000 pages for O

Projection on R is $1400000/600/40 = 93334$ because is done on 2 attributes $(2 \cdot 20)$

Index on Q is unclustering $\rightarrow$ dense. It has 3 values (E,F, pointer) so each page has $600/60 = 10$ data entries + 67% rule $= 6$ data entries $\rightarrow 2400000/6 = 400000$ leaf pages

We cant use onePass algorithm for bag difference because they are not sorted (THEY ARE BAGS) and if they were they should fit in buffer. We can use the twoPass algorithm because $93334/400 < 399$ (Q is sorted so we can use directly the index)

We first compute the sorted sublists of R and then apply second pass (merge)

Cost: $140000 + 2 \cdot 93334 + 400000$