Transaction Systems ← TumMunich база Umbra DB

Motherduck

24.04 6pm

No retake

Friday 12:00 дня
D3 в эту же неделю!!!

1,5 часа лекция, 1.5 часа сем
(придётся кодить :-))

O'Relly
Transact.
Information
System

Основа лекций

Any qustions about the course

tobias.schmidt@in.tum.de

https://db.in.tum.de/teaching/ss23/transactions/

Avia tickets

Не исп. индексы, если надо достать >10% данных (они дорогие)

w(x) - write
r(x) - read $x \in D$ - database
op - operat.   t - transaction
< - order
(op, <) - set of both $\vee$ - XOR

semantics
f takes reads in $x := f_j(v_j, \dots v_{jk})$

w(x)   x - то, что запис $x := f_j(v_j, \dots v_{jk})$

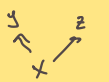
c - commit, a - abort

page model

transaction - partial order of steps (actions) of the form r(x) or w(x), where $x \in D$ and reads and writes applied to the same object are ordered.

↑ подразумевается, что было r(x) r(x) не может (себя)

PO - an arrangement on a set s.t. for certain pair of els, one precedes the other.



x перед y
x перед z
но y и z не имеют порядка м/у собой

ACID - anatomicity, consistency preserved, isolation, durability

все или ничего

↑ как будто эта транзакция одна в системе, не знаем о других

↑ даже если аборт, не остается следов, было consistent и таким ост.

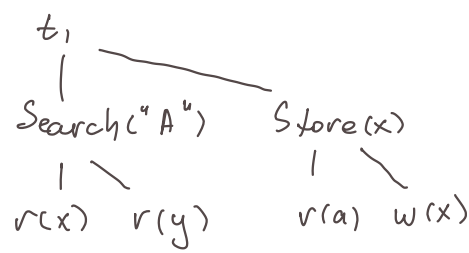
→ если success, то навсегда, даже если сервера упали

object model

transaction - (finite) tree of labeled nodes with

- the transact. identifier as the label of the root n.
- the names and params of invoked operations as labels of inner nodes
- page model r/w operations as labels of leaf nodes, along with partial order (для транз.). w-r, w-w, r-w определяем порядок!

↑ над одними данными! w(x)-r(x)



Common Problems

Lost Update:

x=1	r(x) r(x) w(x+1) "потеряли" w(x+2)
x=3	а надо 4

Inconsistent Read:

x=10 y=20	r(x) w(x-3) r(x) r(y) w(sum=x+y) невовремя прочитали r(y) w(y+3)

Dirty Read:

x=1	r(x) w(x+10) r(x) w(y=x) abort invalid read
y=11	а надо 1

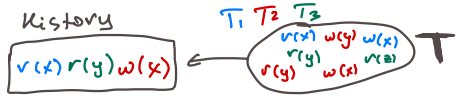
Phantom Problem:

Select(x>10)

Insert(x=11)

Select(x>10)

внезапно на один row больше, хотя операция та же внутри одной Tx



History – для сета операций, кот. сабсетом объединения всех операций всех транзакций – это пара

<сабсет этого сета + аборты и коммиты : ордеринг>

если мы помним из опред., транз. – это r и w.

такая, что:

сна • для ∀ T_i либо a_i в History, либо c_i

ord • то, что ordered внутри транзак., ordered и в History

сна • все операции транзакции t_i идут до c_i или a_i внутри order of history

ord • каждый возможный порядок "конфликтов": любые операции на одних данных, кот. write, внутри History должен быть ordered

Schedule – prefix of history

Definition 3.1 (Schedules and histories):
Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

(i) A **history** for T is a pair $s=(op(s),<_s)$ s.t.
- (a) $op(s) \subseteq \cup_{i=1..n} op_i \cup \cup_{i=1..n} \{a_i, c_i\}$
- (b) for all i, $1 \le i \le n$: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
- (c) $\cup_{i=1..n} <_i \subseteq <_s$
- (d) for all i, $1 \le i \le n$, and all $p \in op_i$: $p <_s c_i$ or $p <_s a_i$
- (e) for all p, $q \in op(s)$ s.t. at least one of them is a write and both access the same data item: $p <_s q$ or $q <_s p$

(ii) A **schedule** is a prefix of a history.

Serial Schedule – такой S, что для ∀ двух T_i и T_j, $i \neq j$, все $op \in T_i$ идут до всех $op \in T_j$ или наоборот.

Herbrand Semantics

$H_S[r_i(x)] := H_S[w_j(x)]$, where $w_j(x)$ – last write on x in S before $r_i(x)$

$H_S[w_i(x)] := f_{ix}(H_S[r_i(y_1), \dots, r_i(y_k)])$, $r(y_1) \dots r_i(y_k)$ – all reads of t_i that occur is s bef. $w_i(x)$

↑ k-ary function

Herbrand Universe – smallest set of symbols s.t.

- $f_{ox}() \in HU$ for each $x \in D$ where f_{ox} is const
- if $w_i(x) \in op_i$ for some t_i and there are m read ops that precede $w_i(x)$ in t_i & $v_1 \dots v_m \in HU$, then $f_{ix}(v_1, \dots, v_m) \in HU$.

↑ функции типа этих чтений

Herbrand Semantics of a schedule

last write ↓

$H[S]: D \to HU$ defined by $H[S](x) := H_S[w_i(x)]$

Conflict Serializability – если конфликт эквивалентен сериализуемой транзакции

$H = w_1(x) \to w_1(y) \to c_1 \to r_2(x) \to r_3(y) \to w_2(x) \to c_2 \to w_3(y) \to c_3$

$T_1 \to T_2$

$T_1 \to T_3$

Conflict equivalence

Final State Equivalence

$s \approx_f s'$, if:

1. $op(s) = op(s')$
2. $H[s] = H[s']$

$\Longleftrightarrow$

$s \approx_f s'$, if:

1. $op(s) = op(s')$
2. $LRF(s) = LRF(s')$

live-read-from

Example a:

$s = r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)$

$s' = r_3(z)\ w_3(z)\ r_2(y)\ r_2(x)\ w_2(z)\ r_1(x)\ w_1(y)\ w_1(x)$ — serial schedule

$H[s](x) = H_s[w_1(x)] = f_{1x}(f_{0x}(\,)) = H_{s'}[w_1(x)] = H[s'](x)$

$H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}(\,)) = H_{s'}[w_1(y)] = H[s'](y)$

$H[s](z) = H_s[w_2(z)] = f_{2z}(f_{0x}(\,), f_{0y}(\,)) = H_{s'}[w_2(z)] = H[s'](z)$

$\Rightarrow s \approx_f s'$

Reads-from Relation

- $r_j(x)$ **reads** x in s **from** $w_i(x)$ if $w_i(x)$ is the last write on x, s.t. $w_i(x) <_s r_j(x)$
- The **reads-from relation** of s is:
  $RF(s) := \{(t_i, x, t_j) \mid r_j(x) \text{ reads } x \text{ from a } w_i(x)\}$
- Step p is **directly useful** for a step q, denoted $p \Rightarrow q$, if q reads from p, or p is a read step and q is a subsequent write step of the same transaction.
- Step p is **alive** in s if it is useful for some step from t_∞, and **dead** otherwise.
- The **live-reads-from** relation of s is:
  $LRF(s) := \{(t_i, x, t_j) \mid \text{an alive } r_j(x) \text{ reads } x \text{ from } w_i(x)\}$

$s = r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$

$s' = r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$

$RF(s) = \{(t_0,x,t_1), (t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

$RF(s') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

(отчуда, чво, чуда)

$LRF(s) = \{(t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

$LRF(s') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

Final State Serializability

For schedules s and s':

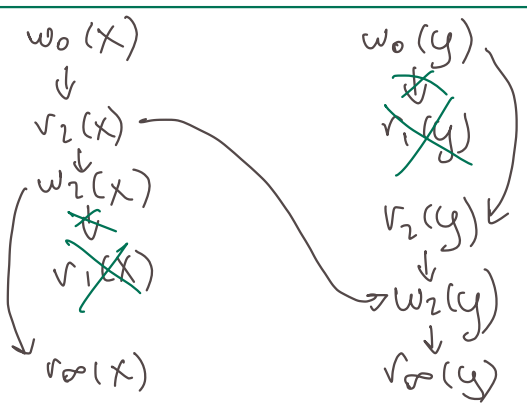
- For s let the step graph $D(s)=(V,E)$ be a directed graph with vertices $V:=op(s)$ and edges $E:=\{(p,q) \mid p \overset{\text{directly used}}{\rightarrow} q\}$, and the reduced step graph $D_1(s)$ be derived from $D(s)$ by removing all vertices that correspond to dead steps. Then:
  $LRF(s)=LRF(s')$ iff $D_1(s)=D_1(s')$
- Final-state equivalence of two schedules s and s' can be decided in time that is polynomical in the length of the two schedules.
- A schedule is final State Serializable (FSR) if there is a serial schedule s' s.t. $s \approx_f s'$. FSR denotes the class of all final-serializable histories.

↑ короче, заботимся только о том, что в конце получилось в базе и о том, как мы к этому пришли

Проверить, что s in FSR:

1) Построить граф
2) Убрать dead steps
3) Придумать serial s', у которого такой же граф.

x	y	z
$w_0(x)$	$w_0(y)$	$w_0(z)$
⋮		



Step Graph
Reduced Step Graph

$s = r_2(x) r_1(y) w_2(x) r_2(y) - r_1(x) w_2(y)$

View Serializability

$s \approx_v s'$, if:

1. $op(s) = op(s')$ — для всех x, y, z... — Hs "для базы"
2. $H[s] = H[s']$ — Hs "для всех операций", dead or alive
3. $Hs[p] = Hs'[p]$ for all read and write steps

Т.е. $s \approx_v s$ если хотя бы одно выполняется:

- $op(s) = op(s')$ и $RF(s) = RF(s')$
- $D(s) = D(s')$

Проверить, что s in VSR – построить граф и подобр. s' с такими же

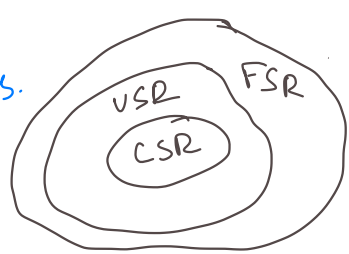
of two schedules

View Equivalence can be decided in time that is polynomial in the length of the schedules.

But: problem of deciding whether $s \in VSR$ is NP-complete

$s \not\approx_v$, $D(s) = D(s')$, а D можно предст. в CG, сжимаем до фрагмента.

$$CSR \subset VSR \subset FSR$$



Projections:

projection $\Pi_T(s)$ of s onto T - result of erasing of all steps of transact. not in T

S is monotone if:

$s \in E$: for $\forall T \subseteq trans(s)$ holds $\Pi_T(s) \in E$

↑ класс (FSR, CSR, ...) ← для любого подмножества транзакций

VSR is not monotone:

$$s = w_1(x)\, w_2(x)\, w_2(y)\, c_2\, w_1(y)\, c_1\, w_3(x)\, w_3(y)\, c_3$$

$RF(s) = \{(t_3, x, t_\infty), (t_3, y, t_\infty)\}$ $T_1 T_2 T_3$ VSR ✓

$RF(\Pi_{\{T_1, T_2\}}(s)) = \{(t_2, x, t_\infty), (t_1, x, t_\infty)\}$ $T_1 T_2$ VSR ✗

CSR monotone $s \in CSR \Leftrightarrow \Pi_T(s) \in VSR$ for all $T \subseteq trans(s)$

CSR is the largest monot. subset of VSR

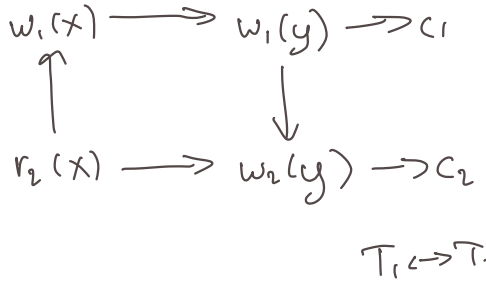
Conflict Serializability

$s \approx_c s'$, if:

1. $op(s) = op(s')$
2. $conf(s) = conf(s')$ ← conflict relations: $\{(p, q) \mid p, q$ are in confl. & $p <_s q\}$

↑ one of them is write

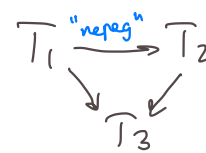
Не сериализуемый конфликт:



Проверить, что $s \in CSR$:

$S = w_1(x)\, r_2(x)\, w_3(x)$

1. Нарисовать conflict graph
2. Проверить, что нету циклов



Testing whether $s \in CSR$ can be done in polynom. time to the schedule's number of transact.

Proof of the Conflict-Graph Theorem

(i) Let s be a schedule in CSR. So there is a serial schedule s' with conf(s) = conf(s').
Now assume that G(s) has a cycle $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow t_1$.
This implies that there are pairs (p_1, q_2), (p_2, q_3), ... , (p_k, q_1)
with $p_i \in t_i$, $q_i \in t_i$, $p_i <_s q_{(i+1)}$, and p_i in conflict with $q_{(i+1)}$.
Because $s' \approx_c s$, it also implies that $p_i <_{s'} q_{(i+1)}$.
Because s' is serial, we obtain $t_i <_{s'} t_{(i+1)}$ for i=1, ..., k-1, and $t_k <_{s'} t_1$.
By transitivity we infer $t_1 <_{s'} t_2$ and $t_2 <_{s'} t_1$, which is impossible.
This contradiction shows that the initial assumption is wrong. So G(s) is acyclic.

(ii) Let G(s) be acyclic. So it must have at least one source node.
The following topological sort produces a total order < of transactions:
a) start with a source node (i.e., a node without incoming edges),
b) remove this node and all its outgoing edges,
c) iterate a) and b) until all nodes have been added to the sorted list.
The total transaction ordering order < preserves the edges in G(s);
therefore it yields a serial schedule s' for which $s' \approx_c s$.

Транзакция T_i reads from T_j:

$w_j(x) < r_i(x)$ - запись в j раньше, чем чтение в i
$a_j \notin r_i(x)$ - не было аборта в j до чтения в i
if $\exists w_k(x): w_j(x) < w_k(x) < r_i(x)$, then $a_k < r_i(x)$,

если что-то записали из другой транзакции в x до чтения в i, то оборвут и откатим

Commutativity Rules

- $r_i(x) r_j(y) \sim r_j(y) r_i(x)$, if $i \neq j$
- $r_i(x) w_j(y) \sim w_j(y) r_i(x)$, if $i \neq j, x \neq y$
- $w_i(x) w_j(y) \sim w_j(y) w_i(x)$, if $i \neq j, x \neq y$

Можно менять:
- родом стоящие рядом разн. транз.
- р.с. ряды играй, если они вообще никак не связ.
- р.с. играть, если они из разн. тр. над разн. эл-тами

Commutativity-based reducible все, что в CSR.
— т.е. есть s' s.t. $s \approx^* s'$

Order Preserving Conflict Serializability

$s \in OCSR$, if:

1. $s \in CSR$
2. $\forall t \in trans.(s)$, if t complitely preceedes t' in s, then the same holds in s'!

Потому что не было, что в s они идут в одном порядке и завершены, а в s' — в другом

$s = w_1(x) r_2(x) c_2 w_3(y) c_3 w_1(y) c_1$

$1 \to 2$ $\quad$ $3 \to 1$ $\quad$ $T_3 T_1 T_2$ - in CSR

$T_2 \to T_3$ $\quad$ а тут нет $\Rightarrow$ not in OCSR

Commit-order Preserving Conflict Serializability

- $s \in COCSR$, if:

  $\forall t_i, t_j \in trans(s)$: if there are $p \in t_i, q \in t_j$ with $(p, q) \in conf(s)$, then $c_i <_s c_j$.

  Т.е. для заданных транзакций, порядок конфликтов определяет порядок коммитов.

- $s \in COCSR$, if:

  1. $\exists s' : s \overset{CSR}{\approx_c} s'$
  2. $\forall t_i, t_j \in trans(s): t_i <_{s'} t_j \Leftrightarrow c_i <_s c_j$

  ▷ сериал. sched' транзакции расположены так, как были ↗ расположены их коммиты ▷ сериал. S.

Ясно, что если бы $t_i c_s t_j$ и $s \in COCSR$ и $s \notin OCSR$, то $t_i \not<_s t_j$, а $= t_i >_s t_j$, а тогда $c_j >_s c_i$ .....

$$COCSR \subset OCSR \subset CSR$$

(по опред.)

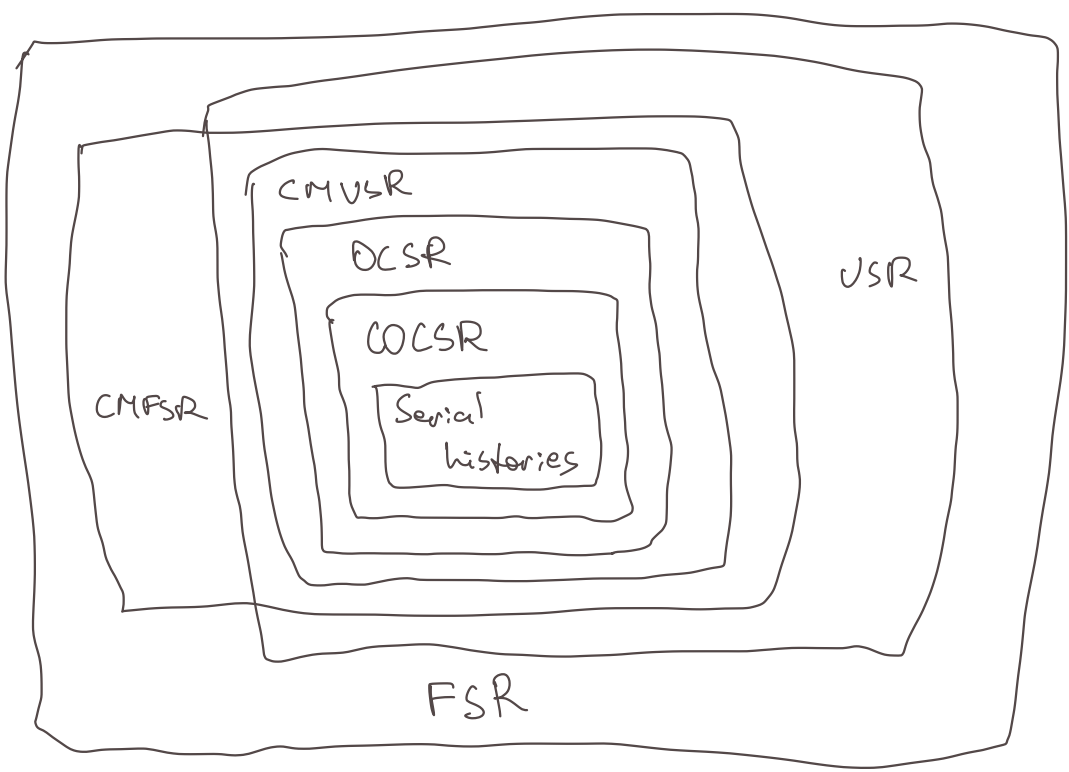
потому что если в CG $T_i \to T_j \Rightarrow c_i \to c_j$, а если бы был цикл, то $c_1 \to c_2 \to .. c_k \to c_1$ ← такое невозможно

prefix-closed – когда свойство S выполн. на префиксе S
commit-closed – когда свойство S выполн. на проекции
S на всех закоммиченных транзак.

Commit-Θ-serializability

S is C-Θ-S, if:

C(P(p)) is Θ-serial. for each prefix p of S, Θ ∈ {FSR, USR, CSR}
↑ only commited trans.



Indivisible unit of t_i relative to t_j, $t_i \neq t_j$ – кусок операций из t_i, в кот. нельзя вклинить никакую операцию из t_j

подстрока

q depends on p ($p \rightsquigarrow q$) – если они в одной транз. или в конфликте, и $p <_S q$

Relatively serial S – даже если там есть не-serial куски – опер. $p \in t_j$ вклинивается в неделимую подстроку t_i, то никакая из операций куска не зависит от p и p не зависит ни от одной операции из куска.

CSR, FSR, VSR, OCSR, COCSR

$S_1 = w_3(y)\, c_3\, w_1(x)\, r_2(x)\, c_2\, w_1(y)\, c_1$

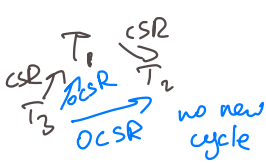
$S_2 = w_1(y)\, w_2(x)\, w_2(y)\, c_2\, w_1(y)\, c_1\, w_3(x)\, w_3(y)\, c_3$

$S_1 = t_3\, t_1\, t_2$ – CSR $\Rightarrow$ FSR & VSR.

Дальше: OCSR и COCSR

not

$t_1 <_{t_2}$ c_{t_2}



S_2 = $t_1\, t_2\, t_3$ – not CSR

VSR ✓

SR ✓

not CSR $\Rightarrow$ not OCSR & COCSR

$t_2 \to t_1$

$t_3 \to t_2$

$t_3 \to t_1$

$t_1 \to t_2$

x	y
	w_1
r_2	w_2
	w_1
w_2 ↓	w_3 ↓

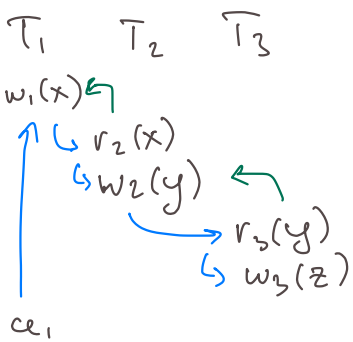
Recoverability

History is recoverable if:

$\forall T_i : T_i$ reads from T_j, $i \neq j$, $c_i \in H$ $\quad c_j < c_i$

$H = w_1(x)\ r_2(x)\ w_2(y)\ c_2\ a_1$ — NOT recoverable

Cascading aborts



Все три транзакции откатятся

Strictness

H is strict if:

$\forall w_j(x), p_i(x), \quad w_j(x) < p_i(x) \quad a_j < p_i(x)$ or $c_j < p_i(x)$

p_i: w_i or r_i

Если никакие незакоммит. или незаборт. данные не читаются и не перезаписываются

Schedulers.

Gen(S) - all schedules produced by S

Optimistic - aggressive, выполняет все, но в конце может где-то застрять
Pessimistic - conservative, откладывает некоторые операции, что может привести к сериалу. S (наприм. Locking Schedulers)

Two-Phase locking Protocol (2PL)

2 типа локов:
- S - shared - read lock
- X - exclusive - write lock

Gen(2PL) ⊂ CSR
Gen(2PL) ⊂ OCSR

2 фазы: лок и анлок, причем вторая фаза наступает как можно раньше
В лок фазу нельзя анлокать и наоборот.
Фазы не зависят друг от друга м/у транзакц.

Пример ниже.

Strict 2PL

Фаза анлоков модифиц.: exclusive локи (xl) держатся до коммита (или аборта)

Strong 2PL

То же, что и S2PL, но еще и с Sl.

Gen(SS2PL) ⊂ Gen(S2PL) ⊂ Gen(2PL)
Gen(SS2PL) ⊂ COCSR

$s = r_1(x)\, r_3(y)\, w_3(y)\, r_2(z)\, w_2(x)\, r_4(y)\, c_3\, w_4(z)\, c_4\, c_2\, c_1$

t_1	t_2	t_3	t_4
$sl_1(x)$			
$r_1(x)$			
$u_1(x)$			
		$sl_3(y)$	
		$r_3(y)$	
		$xl_3(y)$	
		$w_3(y)$	
		$u_3(y)$	
	$sl_2(z)$		
	$r_2(z)$		
	$xl_2(x)$		
как можно раньше начин. фазу unlock'ов →	$u_2(z)$		
	$w_2(x)$		
	$u_2(x)$		
			$sl_4(y)$
			$r_4(y)$
			$xl_4(z)$
		-//- →	$u_4(y)$
			$w_4(z)$
			$u_4(z)$

2PL

Static (conservative) 2PL - сначала берем все-все locks, а потом уже выполняем

$s = r_1(x)\, r_3(y)\, w_3(y)\, r_2(z)\, w_2(x)\, r_4(y)\, c_3\, w_4(z)\, c_4\, c_2\, c_1$

t_1	t_2	t_3	t_4
$sl_1(x)$			
$r_1(x)$			
$u_1(x)$			
		$sl_3(y)$	
		$r_3(y)$	
		$xl_3(y)$	
		$w_3(y)$	
	$sl_2(z)$	waits	
	$r_2(z)$		
	$xl_2(x)$		
	$u_2(z)$		
	$w_2(x)$		
	waits		$sl_4(y)$
			blocked
		$u_3(y)$	
		c_3	$r_4(y)$
			$xl_4(z)$
			$u_4(y)$
			$w_4(z)$
			$u_4(z)$
			c_4
	$u_2(x)$		
	c_2		
c_1			

S2PL

$s = r_1(x)\, r_3(y)\, w_3(y)\, r_2(z)\, w_2(x)\, r_4(y)\, c_3\, w_4(z)\, c_4\, c_2\, c_1$

t_1	t_2	t_3	t_4
$sl_1(x)$ $r_1(x)$ waits			
		$sl_3(y)$ $r_3(y)$ $xl_3(y)$ $w_3(y)$ waits	
	$sl_2(z)$ $r_2(z)$ $xl_2(x)$ locked		
			$sl_4(y)$ locked
		$u_3(y)$ c_3	
			$r_4(y)$ $xl_4(z)$ locked
$u_1(x)$ c_1			
	$w_2(x)$ $u_2(z)$ $u_2(x)$ c_2		
			$w_4(z)$ $u_4(y)$ $u_4(z)$ c_4

SS2PL

Deadlock Resolution

- Victim selection strategies: Choose victim to abort & repeat until no more cycles
  - Last blocked $t_1 \xrightarrow{1} t_2 \xrightarrow{2} t_3 \xleftarrow{3}$ ждет t_3
  - Random
  - Youngest та, которая началась позже всех
  - Minimum locks та, у кот. взято меньше всего локов
  - Minimum work та, которая потребила пока что наим. кол-во ресурсов системы (CPU, ...)
  - Most cycles та, при удал. кот. прервется наиб. кол-во циклов
  - Most edges та, при удал. кот. удалится наиб. кол-во ребер

You can restrict lock waits to ensure acyclic WFG. (waits for graph)

- Deadlock prevention strategies: Restrict lock waits to ens. acyclic WFG
  - wait-die $t_i \xrightarrow{\text{waits for}} t_j$, if t_i started before t_j, wait, else abort.
    Короче, если пришла новая молодая транзакция, и видит старую, то аборт новую, иначе ждем
  - wound-wait $t_i \rightarrow t_j$, если t_i старше t_j, аборт t_j; else wait
  - immediate restart $t_i \rightarrow t_j \Rightarrow$ abort t_i сразу же
  - running priority $t_i \rightarrow t_j$, если t_j уже сама заблок., abort t_j, else wait
  - timeout abort waiting tr. when timer expires

Altruistic locking

Отдаем лок другой трань. и не должны с ним больше ничего делать (по уроде мы не и должны описывать)

Допис.

$Gen(2PL) \subset Gen(AL)$
$Gen(AL) \subset CSR$

Non-locking Schedulers

Basic Timestamp Ordering

Timestamp ordering rule (TO rule):
Each transaction t_i is assigned a **unique timestamp ts(t_i)** (e.g., the time of t_i's beginning).
If $p_i(x)$ and $q_j(x)$ are in conflict, then the following must hold:
$p_i(x) <_s q_j(x)$ iff ts(t_i) < ts(t_j) for every schedule s.

Theorem 4.15:
Gen (TO) $\subseteq$ CSR.

Basic timestamp ordering protocol (BTO):
- For each data item x maintain max-r (x) = max{ts(t_j) | $r_j(x)$ has been scheduled} and max-w (x) = max{ts(t_j) | $w_j(x)$ has been scheduled}.
- Operation $p_i(x)$ is compared to max-q (x) for each conflicting q:
  - if ts(t_i) < max-q (x) for some q then abort t_i
  - else schedule $p_i(x)$ for execution and set max-p (x) to ts(t_i)

$$s_1 = r_1[x]w_1[x]w_1[y]c_1r_2[y]r_3[z]w_3[z]c_3r_2[z]c_2$$

$$s_2 = r_1[x]w_2[x]r_3[x]w_2[z]c_2w_3[z]c_3r_1[z]c_1$$

Solution. Let's first show the **BTO** output for both schedules:

$$s_{1out} = r_1[x]w_1[x]w_1[y]c_1r_2[y]r_3[z]w_3[z]c_3a_2$$

Timestamps history:
$\text{Ts}[r_x] = \emptyset\ 1$
$\text{Ts}[w_x] = \emptyset\ 1$
$\text{Ts}[r_y] = \emptyset\ 2$
$\text{Ts}[w_y] = \emptyset\ 1$
$\text{Ts}[r_z] = \emptyset\ 3$
$\text{Ts}[w_z] = \emptyset\ 3$ (> 2 for $r_2 \Rightarrow$ abort T_2)

$$s_{2out} = r_1[x]w_2[x]r_3[x]w_2[z]c_2w_3[z]c_3a_1$$

Timestamps history:
$\text{Ts}[r_x] = \emptyset\ \not{1}\ 3$
$\text{Ts}[w_x] = \emptyset\ 2$
$\text{Ts}[r_z] = 0$
$\text{Ts}[w_z] = \emptyset\ \not{2}\ 3$ (> 1 for $r_1 \Rightarrow$ abort T_1)

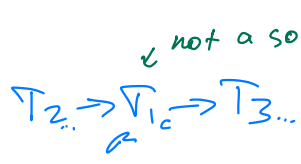
Serialization Graph Testing

1. creates a new node for transaction t_i in the current graph G if $p_i(x)$ is the first operation it sees from t_i;
2. inserts edges of the form (t_j, t_i) into G for each operation $q_j(x)$ that is in conflict with $p_i(x)$, $i \neq j$, and that has been output previously; now two cases can arise:
   (a) The resulting graph G is cyclic. If $p_i(x)$ were executed, the resulting schedule would no longer be serializable. Thus, $p_i(x)$ is rejected and t_i aborted, and the node for t_i and all its incident edges are removed from G.
   (b) G is (still) acyclic. Then $p_i(x)$ can be output—that is, added to the schedule already output—and the tentatively updated graph is kept as the new current one.

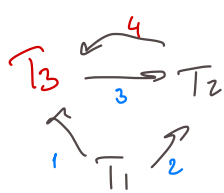
Gen(SGT) = CSR

Ноды из графа можно удалять, когда они коммитятся и при этом к ним не идут ребра

можно удалить



нельзя удалить, пока T_2 не закоммитится



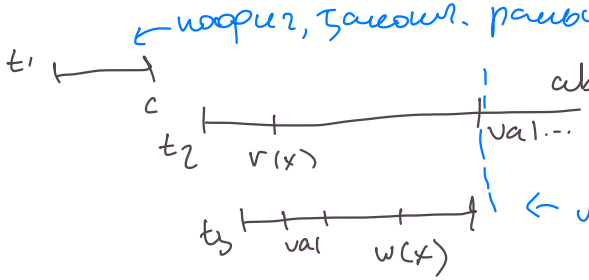
Optimistic protocols

read - validate - write

BOCC

Our reads against their previously commited writes

↑ Аборт, если прочитали то, что закоммитилось в это время



- мы. Аборт себя

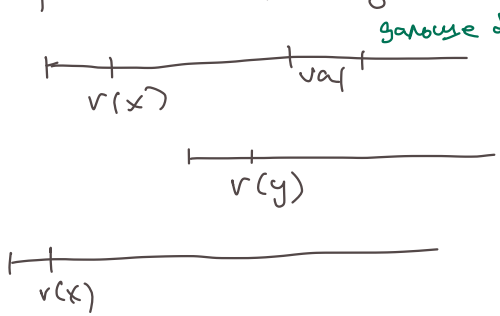
validation - critical phase, никто другой в нее ничего не делает

FOCC

Gen (BOCC) с CSR

FOCC

Our future writes against their reads.



1. Аборт себя
2. Аборт других
3. Ждем, пока конфл. ридит закоммит. и сами валидируем

Read-only трансы вообще можно не валидир.

Multiversion Schedules

- Multiversion
  $r_1(x_0)\ w_2(x_2)\ r_3(x_0)$ – читает любой свет
- Monoversion
  $r_1(x_0)\ w_2(x_2)\ r_3(x_2)$ – читает только текущий свет

$<<_x$ – порядок версий x

Multiversion View Serializability

Reads from $\{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$ $\quad w_2(x_2)\ r_{10}(x_2)$

VSR $\subset$ MVSR

Deciding if mv history in MVSR is NP-complete

Multiversion conflict serialisation

MV conflict: $r_i(x_j)$ и $w_k(x_k)$, s.t. $r_i(x_j) <_m w_k(x_k)$

Проверка – подбираем по кайфу $w_0[x_0]\ r_1[x_0]\ w_1[x_1]\ r_2[x_0]\ w_2[x_2]$

не сериально.

MCSR – (не) с помощью конфл. графа!
Подбир. сериал., т.ч. порядок конфликт-пар не нарушен!

Multiversion Serialization Graph (MVSG)

Граф транзакций и след. ребрами:

- $T_j \to T_k$, если есть $r_k[x_j]$ (конфликт графа)
- $T_i \to T_j$, если есть $w_i[x_i]$ и $r_k[x_j]$, при этом $x_i << x_j$ ($w_0[x_0]$ перед $w_j[x_j]$)
- $T_k \to T_i$, если есть $r_2[x_j]$ и $w_i[x_i]$, при этом $x_j << x_i$ ($r_2[x_1]$ перед $w_3[x_3]$)

↑ Этим проверяем MVSR

Если есть цикл — не в MVSR!

1) $T_w \to T_r$
2) $T_{w_1} \to T_{w_2} \dots T_r$
3) $T_w, \; T_r \to T_{w_2}$

Multiversion Timestamp Ordering

- принимаем $r_i(x_k)$, if $k < i$
- принимаем $w_k(x_k)$, если не было $r_i(w_j)$, т.ч. $j < k < i$
- откладываем коммит до тех пор, пока не закоммитят трань., кот. записали то, что мы прочитали

$Gen(MVTO) \subset MVSR$

Multiversion 2PL

- Non-Final steps:

Для non-final steps только 2 версии - committed и одна uncommited

$r(X)$ читает закоммич., или незакоммич.-ка выбор

$w(X)$ заблок., пока есть для версии, откуда пишем

- Final step:

Final step delayed, пока не закоммитятся все, кто прочитали "current version" того, кто мы записали и все те, кто записали что-то, что мы прочитали

T.e:

Модем, если: • прочитали то, что кто-то записал
• перезаписали то, что кто-то прочитал

↑ короче, предыдущ. $V_i(X_0)$ и $W_i(X_i)$

↑ как в MVTO

$r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$

Final step ↑ Final step

$r_1(x_1)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)$ ↑ $w_1(y_1)\ c_1\ w_2(x_2)\ c_2$

x: 0 (c), 1 (u)

y: 0 (c), 2 (u)

↑ может прочитать любую

1) не может создать новую версию x (их уже 2)

2) должен быть delayed как final step, потому что прочитала x_1 и y_1, кот. еще не закоммитил.

2V2PL

before and after image

wl – to write uncommited & support only 2 versions
rl – to read only commited version
cl – certify lock for final step of tr. for all written data

Multiversion Serialization Graph Testing Scheduler

Делаем подбором:

$r_i(x) \to r_i(x_j)$ с условиями: ⇒ т.к. можем прочитать из любого T_j

- После $w_j(x_j)$ не должно быть $w_k(x_k)$, т.е. мы не должны читать старые данные
- Не должно быть ребра $t_i \to t_j$ (а то будет цикл)

Добавляем $t_j \to t_i$ и ребро, которое определено для всех остальных $w_k(x_k)$, кот. не подпад. под условия ↑ (т.е. которых мы могли выбрать как j), где они стоят: до t_j или после t_i, т.е. рисуем $t_k \to t_j$ или $t_i \to t_k$

Не можем прочитать из транз., которая следует за нашей, т.е. из нашей T_i уже есть в нее путь

$w_i(x_i)$:

Добавляем все MV conflicts, т.е. $r_j(x_k)$, которые были до этого write

ROMV Protocol

updates – 2PL on both r u w (Strict or Strong) (deps race)

w creates a new version with the tr. commit time

read-only – tr. timestamped with its begin time

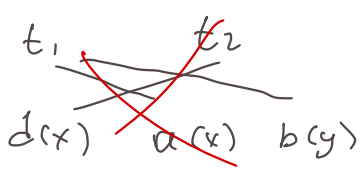
$r_i(x) \rightarrow r_i(x_j)$, x_j – largest timestamp (no deps race)

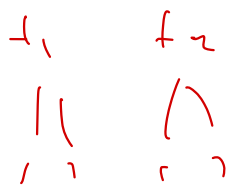
Object Model

layered

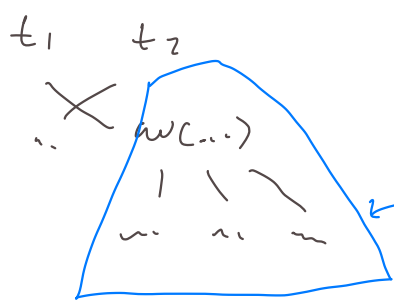
acyclic n (n-layered)

serial



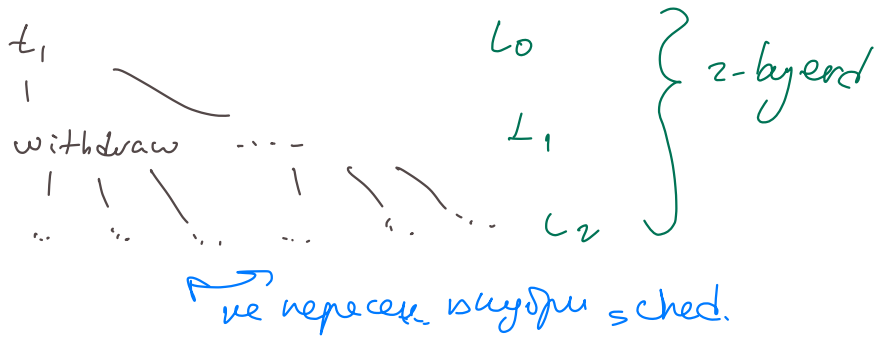


isolated



leaves of dep.

flat - 2-level schedule



не пересек. внутри sched.

Смотрим, serializable ли по перестановке (т.е. среднему) слою, когда все subtree $T_1 \dots T_k$ isolated

Каждый layer должен быть serialised in the same order

L_i-to-L_{i-1} schedule - 2-level sched. derived from a tree

если каждое в OCSR, то и все дерево tree-reducible (т.е. можно полностью удалить прим.)

Tree reducible - can be transformed to the total order of its roots.

Commut на листьях
Order. : если δ_{ij} <, значит <
Prun -> isolated subtree -> root

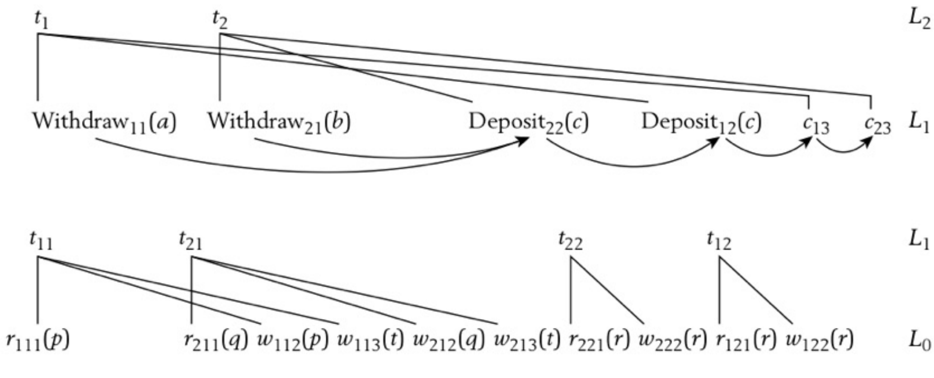
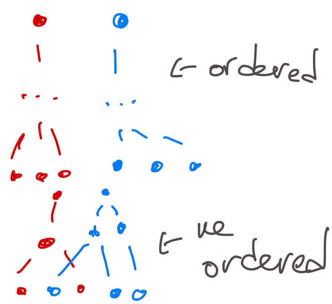


Figure 6.7 Level-to-level schedules constructed from the layered object model schedule of Figure 6.1.

Conflict-faithfullness – если все операции в конфл., то их дети тоже

Nodes p and q are ordered if все их leaf-level дети ordered в том же порядке



- **Leaf Nodes**:
  - p' and q' are leaf nodes (individual operations).
  - Suppose $p' \leq_s q'$ (meaning p' occurs before q').
- **Higher-Level Nodes**:
  - p is a higher-level node that has descendants, including p'.
  - q is a higher-level node that has descendants, including q'.

To determine if $p \leq_s q$:

- Check all the leaf descendants of p and q.
- If every leaf descendant of p (like p') is ordered before every leaf descendant of q (like q') in the leaf ordering, then $p \leq_s q$.

Layered 2PL

Берем лок перекидываем. t_i — withdr ← тут

Если отпустили, то больше не берем локи (2PL)

Когда закончили операцию, на кот. нужен лок, отпускаем локи у детей (а лок на опер. держим)

Gen(Layered 2PL) ⊂ Tree Reducible

General Object-Model 2PL

Обычные и retained локи:

Выполняем $f(x)$ – локаем x

Закончили $f(x)$ – лок стал retained (пока кто-то может его забрать (нужно?), но только если это safe)

Precision locking

- predicate locking на все query +1
- insert, delete, update локают конкретные rows
  ↑ прием update проверяется на пересеч с predicate и до, и после

Functional dependencies

a	b	c
1	5	3
2	7	4
1	5	7

a determines b one depend. - primary key

IDM Final State Serializability Testing

NP-complete. Nepeбopan

IDM Conflict Serializability

$S = w_2(1;2)\, w_1(2;3)\, w_2(3;2)$ $t_1 \rightleftarrows t_2$

$S \approx w_1(2;3)\, w_2(1;2)\, w_2(3;2) \approx t_1 t_2$ ← FSR

IDM extended Conflict Graph

Extended Confl. Serial. if $EG(S)$ is acyclic

$CSR \subset ECSR \subset FSR$

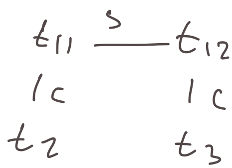
Transaction Chopping

Делим t_i на куски $t_{i_1} \ldots t_{i_k}$ 

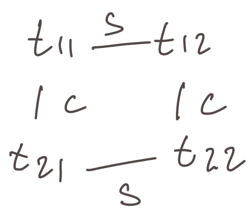
Chopping is correct - выполн. кусков confl. equiv. to a ser. hist и:
piece - неделимо исп.

Chopping graph - undirected graph без циклов

confl.
c edge - разн. тр. в конфл.
s edge - одна тр.
safe



Cycle - cycle, that involves both c и s edges



Key-Range Locking

ins, del, search – lock single key

range_search – lock interv [low, high]

table scan locks $(-\infty; +\infty)$

+page locks required during sub-trans.

Incremental Key-Range Locking

Lock [found_key, next-ex-key) + page locks during sub-trans.

search(x) – lock x or largest $k < x$

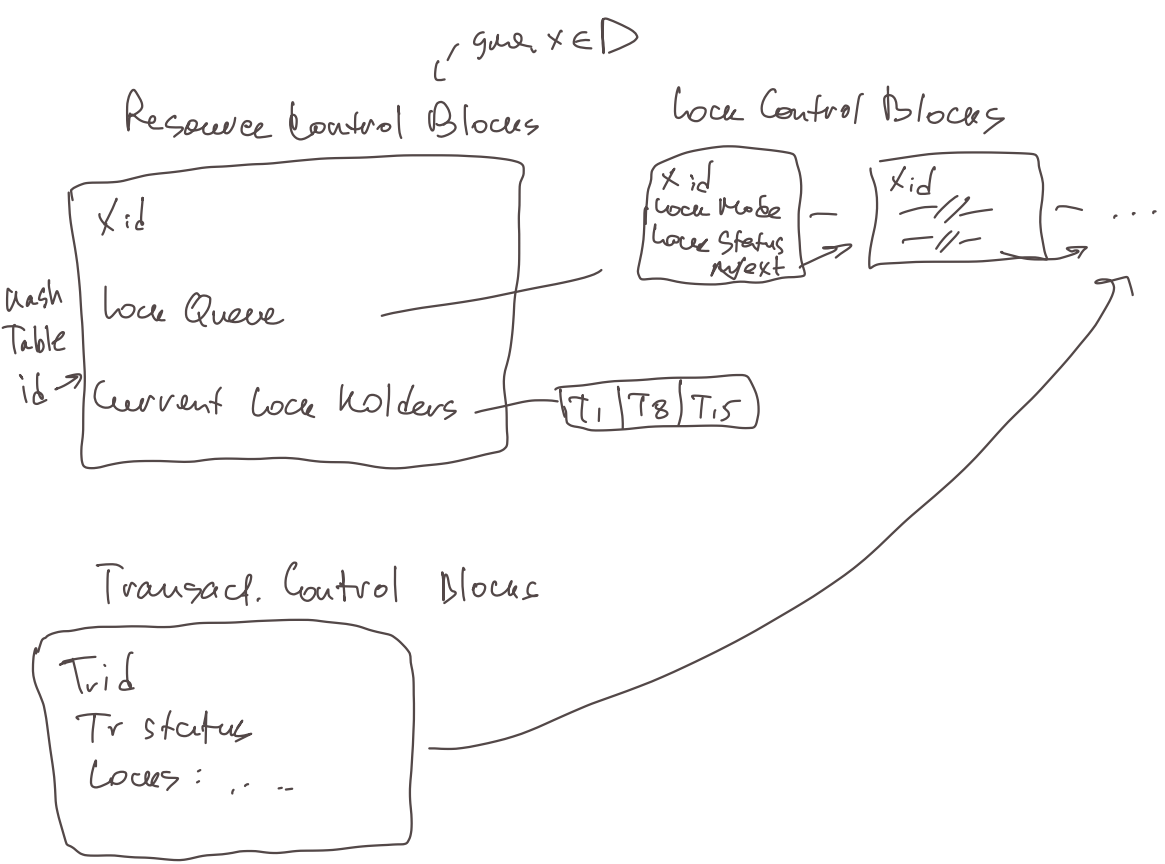
next(current_key, ...) – request rl on current key

insert(y, RID) – req. wl on y & largest $k < y$
delete –//–

Page level – read & write

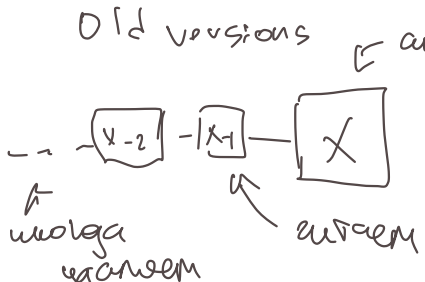
Access Level – search, ret.,

Gen(Lock Coupling) $\subset$ OCSR

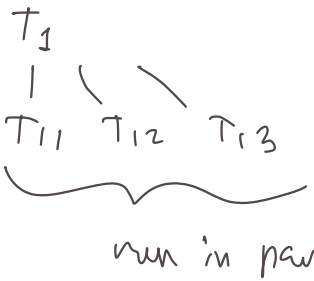


Granularity levels – db, table, page, row

Intention locks – хотим более мелкий лок - shared or excl.



Nested Transactions



Conflict-ratio-driven Overload Control

Adjust tr. processing based on the ratio of confl. locks.
to prevent thrashing

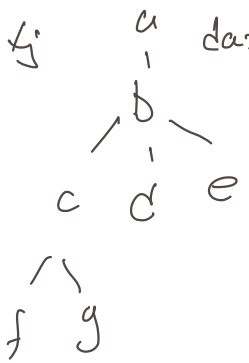
Write-only Tree locking

- Request node, request each parent of node locked
- After $WL_i(x)$ no further $WL_i(x)$ is allowed (on the same x)

If T_i locks x before T_j
for all successors v locked by both T_i and T_j
$$WL_i(v) <_s WU_i(v) <_s WL_j(v)$$

$Gen(WTL) \subseteq CSR$

WTL is deadlock free



Read-write Tree Locking

Pitfall of t is a set

$\{ \cup \{x \in WS(t) \mid x$ is a child or parent of some $y \in C_i\}$

O_2PL

$wl_1(x)\ vl_2(x)$ ← $w_1(x)\ r_2(x)$

↑ usually we wait, но

▷ O_2PL может дать ряд пок

но еще нужно соблюдать порядок unlock'ов

$r_1(x)$ $w_2(x)$ $w_1(x)$

$rl_1(x)$ $r_1(x)$ $w_2l(x)$ $w_2(x)$ $wl_1(x)$

→ cycle, поэтому abort

Altruistic locking

"donating" locks to other transactions

↓

отдали и забыли

~~Первая транз. должна отпускать пок~~

$p_j(x)$ is in the wake of t_i : if $d_i(x) < p_j(x) < o_{unlock}(x)$

$s = r_1(x)\ w_3(x)\ c_3\ r_2(x)\ r_1(y)\ r_1(z)\ c_1\ w_2(z)\ c_2$

t_1	t_2	t_3
$sl\,x$		
rx		
dx		
		$xl\,x$
		wx
		c_3 < ux
	$sl\,x$	
	~~waits~~	
$sl\,y$	rx	
ry	$xl\,z$	
$sl\,z$	ux	
ux	wz	
rz	c_2	
uz		
c_1		

$$o_s = r|_1(x)\, r_1(x)\, d_1(x)\, w|_3(x)\, w_3(x)\, wu_3(x)\, c_3$$
$$r|_2(x)\, r_2(x)\, r|_1(y)\, r_1(y)\, r|_1(z)\, r_1(z)$$
$$r u_1(x)\, r u_1(y)\, r u_1(z)\, c_1\, w|_2(z)\, w_2(z)\, r u_2(x)$$
$$w u_2(t)\, c_2$$

MVSG

$m = w_0[x_0] r_1[x_0] w_1[x_1] r_2[x_0] w_2[x_2]$

$\forall r[x_j], w_i[x_i]$:

if $x_i \ll x_j$: $T_i \to T_j$

if $x_j \ll x_i$: $T_j \to T_i$

conflicts:

$r_1[x_0], w_2[x_2]$

Если одновременно

тогда, то

$r_1[x_0] \leftrightarrow w_2[x_2]$

$r_1[x_0], w_3[x_3]$

T_0 T_1

T_2

ROMV, update - strict 2PL / strong 2PL

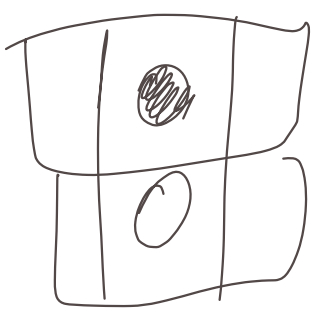
Commutative:
deq-ok, enq-ok
deq-empt, deq-empt

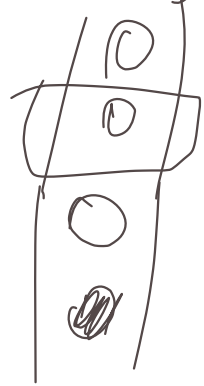
– не важно, что первое

Not com:
enq-ok, enq-ok
deq-ok, deq-empt

не можем поменять

Semi-queue – enq. as usual, but deq – любой возможн. элемент, не удаляя, нашей





Escrow locking

x	inf	sup
[0,100]	40	60
$inc(x,40)$	40	100
C_1	80	100

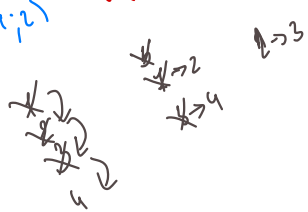
$inc_2(x,50)$ – rejected as out of the range

$inc_3(x,20)$ - - - wait on all other tr, that inc. x

e-e-e

$d(A=3)$ $d(A=3)$ $m(A=1, A=2)$ $m(A=2, A=3)$ $m(A=3, A=4)$

$r(1)w(1,2)$ $r(2)w(2,3)$ $r(3)w(3,4)$



{1;2;3;4}

d(1) m(2->3)

d(1) m(4->5)

m(3->4)

m(5->6)

1,2,3,4

m(2->3) d(3) m(3->4) d(5)

1 2 3

d(6)

m(2->3) m(3->4) d(4) d(5) m(4->5) m(5->6)

1 2 3 3 2 1

1,2,3

d(4) d(5) d(6)

2,1

1,2