Architecture and Implementation of Database Systems (Summer 2018)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Summer 2018

Part VIII

Concurrency Control

The “Hello World” of Transaction Management

- My bank issued me a debit card to access my account.
- Every once in a while, I’d use it at an ATM to draw some money from my account, causing the ATM to perform a **transaction** in the bank’s database.

```
$bal \leftarrow$ read_bal ($acct\_no$) ;
$bal \leftarrow bal - 100\,\text{CHF}$ ;
write_bal ($acct\_no$, $bal$) ;
```

- My account is properly updated to reflect the new balance.

Concurrent Access

The problem is: My wife has a card for the account, too.

- We might end up using our cards at different ATMs at the **same time**.

me	**my wife**	**DB state**
$bal \leftarrow \texttt{read}\,(acct)\,;$		1200
	$bal \leftarrow \texttt{read}\,(acct)\,;$	1200
$bal \leftarrow bal - 100\,;$		1200
	$bal \leftarrow bal - 200\,;$	1200
$\texttt{write}\,(acct\,,\,bal)\,;$		1100
	$\texttt{write}\,(acct\,,\,bal)\,;$	1000

- The first update was **lost** during this execution. Lucky me!

Another Example

- This time, I want to **transfer** money over to another account.

```
   // Subtract money from source (checking) account
1  chk_bal ← read_bal (chk_acct_no) ;
2  chk_bal ← chk_bal − 500 CHF ;
3  write_bal (chk_acct_no , chk_bal) ;
   // Credit money to the target (saving) account
4  sav_bal ← read_bal (sav_acct_no) ;
5  sav_bal ← sav_bal + 500 CHF ;
6  write_bal (sav_acct_no , sav_bal) ;
```

- Before the transaction gets to step **6**, its execution is **interrupted/cancelled** (power outage, disk failure, software bug, . . .). My money is **lost** ☹.

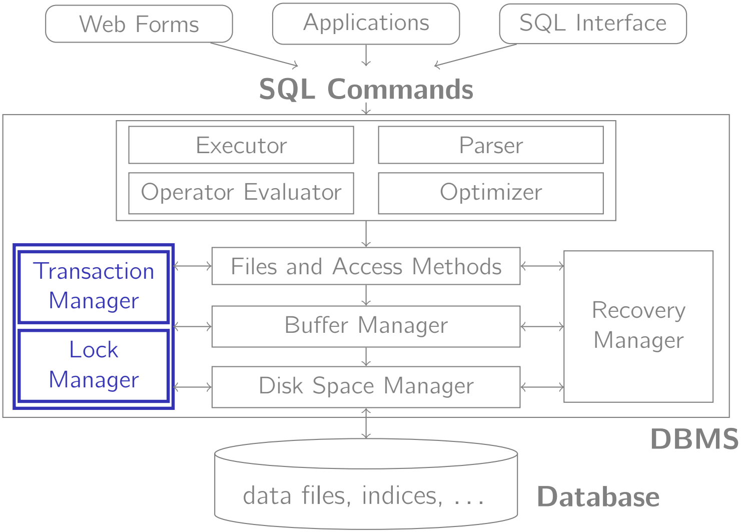
ACID Properties

One of the key benefits of a database system are the **transaction properties** guaranteed to the user:

A	Atomicity	Either **all** or **none** of the updates in a database transaction are applied.
C	Consistency	Every transaction brings the database from one **consistent** state to another.
I	Isolation	A transaction must not see any effect from other transactions that run in parallel.
D	Durability	The effects of a **successful** transaction maintain persistent and may not be undone for system reasons.

A challenge is to preserve these guarantees even with **multiple users** accessing the database **concurrently**.

Concurrency Control



Anomalies: Lost Update

- We already saw a **lost update** example on slide 296.
- The effects of one transaction are lost, because of an uncontrolled overwriting by the second transaction.

Anomalies: Inconsistent Read

Consider the money transfer example (slide 297), expressed in SQL syntax:

```
     Transaction 1                      Transaction 2
UPDATE Accounts
  SET balance = balance - 500
  WHERE customer = 4711
    AND account_type = 'C';
                                   SELECT SUM(balance)
                                     FROM Accounts
                                    WHERE customer = 4711;
UPDATE Accounts
  SET balance = balance + 500
  WHERE customer = 4711
    AND account_type = 'S';
```

- Transaction 2 sees an **inconsistent** database state.

Anomalies: Dirty Read

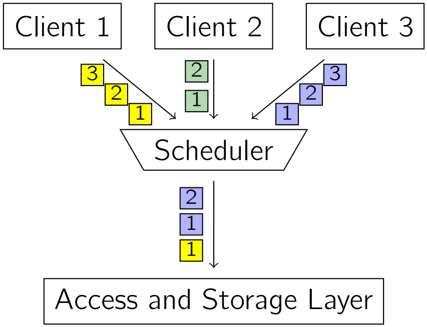
At a different day, my wife and me again end up in front of an ATM at roughly the same time:

me	**my wife**	**DB state**
$bal \leftarrow$ `read` $(acct)$;		1200
$bal \leftarrow bal - 100$;		1200
`write` $(acct, bal)$;		1100
	$bal \leftarrow$ `read` $(acct)$;	1100
	$bal \leftarrow bal - 200$;	1100
`abort` ;		1200
	`write` $(acct, bal)$;	900

- My wife's transaction has already read the modified account balance before my transaction was **rolled back**.

Concurrent Execution

- The **scheduler** decides the execution order of concurrent database accesses.



Database Objects and Accesses

- We now assume a slightly simplified model of database access:
  1. A database consists of a number of named **objects**. In a given database state, each object has a **value**.
  2. Transactions access an object o using the two operations `read` o and `write` o.
- In a **relational** DBMS we have that

$$\text{object} \equiv \text{attribute} \quad .$$

Transactions

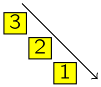
A **database transaction** T is a (strictly ordered) sequence of **steps**. Each **step** is a pair of an **access operation** applied to an **object**.

- Transaction $T = \langle s_1, \dots, s_n \rangle$
- Step $s_i = (a_i, e_i)$
- Access operation $a_i \in \{\texttt{r(ead)}, \texttt{w(rite)}\}$

The **length** of a transaction T is its number of steps $|T| = n$.

We could write the money transfer transaction as

$$T = \langle\ (\texttt{read}, \mathit{Checking}), (\texttt{write}, \mathit{Checking}),\ (\texttt{read}, \mathit{Saving}), (\texttt{write}, \mathit{Saving})\ \rangle$$



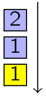
or, more concisely,

$$T = \langle r(C), w(C), r(S), w(S) \rangle \enspace .$$

Schedules

A **schedule** S for a given set of transactions $\mathbf{T} = \{T_1, \ldots, T_n\}$ is an arbitrary sequence of execution steps

$$S(k) = (T_j, a_i, e_i) \qquad k = 1 \ldots m \ ,$$



such that

1. S contains all steps of all transactions an nothing else and
2. the order among steps in each transaction T_j is preserved:

$$(a_p, e_p) < (a_q, e_q) \text{ in } T_j \Rightarrow (T_j, a_p, e_p) < (T_j, a_q, e_q) \text{ in } S \ .$$

We sometimes write

$$S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$$

to mean

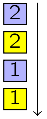
$$\begin{array}{ll} S(1) = (T_1, \texttt{read}, B) & S(3) = (T_1, \texttt{write}, B) \\ S(2) = (T_2, \texttt{read}, B) & S(4) = (T_2, \texttt{write}, B) \end{array}$$

Serial Execution

One particular schedule is **serial execution**.

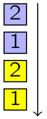
- A schedule S is **serial** iff, for each contained transaction T_j, all its steps follow each other (no interleaving of transactions).

Consider again the ATM example from slide 296.



- $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- This schedule is **not** serial.

If my wife had gone to the bank one hour later, "our" schedule probably would have been serial.



- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$

Correctness of Serial Execution

- Anomalies such as the "lost update" problem on slide 296 can **only** occur in multi-user mode.
- If all transactions were fully executed one after another (no concurrency), no anomalies would occur.
- **Any serial execution is correct.**

- Disallowing concurrent access, however, is **not practical**.
- Therefore, allow concurrent executions if they are **equivalent** to a serial execution.

Conflicts

What does it mean for a schedule S to be equivalent to another schedule S'?

- Sometimes, we may be able to **reorder** steps in a schedule.
  - We must not change the order among steps of any transaction T_j ($\nearrow$ slide 306).
  - Rearranging operations must not lead to a different **result**.
- Two operations (a, e) and (a', e') are said to be **in conflict** $(a, e) \nleftrightarrow (a', e')$ if their order of execution matters.
  - When reordering a schedule, we must not change the relative order of such operations.
- Any schedule S' that can be obtained this way from S is said to be **conflict equivalent** to S.

Conflicts

Based on our `read`/`write` model, we can come up with a more machine-friendly definition of a conflict.

- Two operations (T_i, a, e) and (T_j, a', e') are **in conflict** in S if
  1. they belong to two **different transactions** $(T_i \neq T_j)$,
  2. they access the **same database object**, *i.e.*, $e = e'$, and
  3. at least one of them is a `write` operation.
- This inspires the following conflict matrix:

	`read`	`write`
`read`		×
`write`	×	×

- **Conflict relation $\prec_S$:**

$$(T_i, a, e) \prec_S (T_j, a', e')$$
$$:=$$
$$(a, e) \not\leftrightarrow (a', e') \wedge (T_i, a, e) \text{ occurs before } (T_j, a', e') \text{ in } S \wedge T_i \neq T_j$$

Conflict Serializability

- A schedule S is **conflict serializable** iff it is conflict equivalent to **some** serial schedule S'.
- **The execution of a conflict-serializable S schedule is correct.**
  - S does **not** have to be a serial schedule.
- This allows us to **prove** the correctness of a schedule S based on its **conflict graph** $G(S)$ (also: **serialization graph**).
  - **Nodes** are all transactions T_i in S.
  - There is an **edge** $T_i \rightarrow T_j$ iff S contains operations (T_i, a, e) and (T_j, a', e') such that $(T_i, a, e) \prec_S (T_j, a', e')$.
- S is conflict serializable if $G(S)$ is **acyclic**.[18]

[18] A serial execution of S could be obtained by sorting $G(S)$ **topologically**.

Serialization Graph

Example: ATM transactions ($\nearrow$ slide 296)

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, A) \prec_S (T_2, \mathtt{w}, A)$
  $(T_2, \mathtt{r}, A) \prec_S (T_1, \mathtt{w}, A)$
  $(T_1, \mathtt{w}, A) \prec_S (T_2, \mathtt{w}, A)$

$$\begin{tikzcd}
 & T_2 \arrow[dl, bend left] \\
T_1 \arrow[ur, bend left] \arrow[ur, phantom, "\circlearrowright"] &
\end{tikzcd}$$

$\rightarrow$ **not** serializable

Example: Two money transfers ($\nearrow$ slide 297)

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$
- Conflict relation:
  $(T_1, \mathtt{r}, C) \prec_S (T_2, \mathtt{w}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{r}, C)$
  $(T_1, \mathtt{w}, C) \prec_S (T_2, \mathtt{w}, C)$
  $\vdots$

$$\begin{tikzcd}
 & T_2 \\
T_1 \arrow[ur, bend left] &
\end{tikzcd}$$

$\rightarrow$ serializable

Query Scheduling

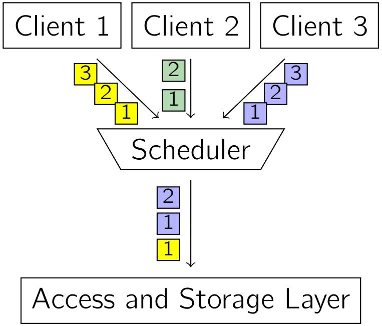
Can we build a scheduler that **always** emits a serializable schedule?

Idea:

- Require each transaction to obtain a **lock** before it accesses a data object o:

```
lock o ;
...access o ...;
unlock o ;
```

- This prevents **concurrent** access to o.



Locking

- If a lock cannot be granted (*e.g.*, because another transaction T' already holds a **conflicting** lock) the requesting transaction T_i gets **blocked**.
- The scheduler **suspends** execution of the blocked transaction T.
- Once T' **releases** its lock, it may be granted to T, whose execution is then **resumed**.
- Since other transactions can continue execution while T is blocked, locks can be used to **control the relative order of operations**.

Locking and Serializability

✎ **Does locking guarantee serializable schedules, yet?**

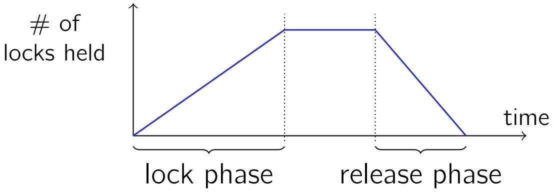
ATM Transaction with Locking

Transaction 1	**Transaction 2**	**DB state**
`lock` (*acct*) ;		1200
`read` (*acct*) ;		
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`read` (*acct*) ;	
	`unlock` (*acct*) ;	
`lock` (*acct*) ;		
`write` (*acct*) ;		1100
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`write` (*acct*) ;	1000
	`unlock` (*acct*) ;	

Two-Phase Locking (2PL)

The **two-phase locking protocol** poses an additional restriction:

- Once a transaction has **released** any lock, it must **not** acquire any new lock.



- Two-phase locking is **the** concurrency control protocol used in database systems today.

Again: ATM Transaction

Transaction 1	**Transaction 2**	**DB state**
`lock` (*acct*) ;		1200
`read` (*acct*) ;		
`unlock` (*acct*) ;		
	`lock` (*acct*) ;	
	`read` (*acct*) ;	
	`unlock` (*acct*) ;	
`lock` (*acct*) ; ↯		
`write` (*acct*) ;		1100
`unlock` (*acct*) ;		
	`lock` (*acct*) ; ↯	
	`write` (*acct*) ;	1000
	`unlock` (*acct*) ;	

A 2PL-Compliant ATM Transaction

- To comply with the two-phase locking protocol, the ATM transaction must not acquire any new locks after a first lock has been released.

```
lock (acct) ;                      } lock phase
bal ← read_bal (acct) ;
bal ← bal − 100 CHF ;
write_bal (acct, bal) ;
unlock (acct) ;                    } unlock phase
```

Resulting Schedule

Transaction 1	Transaction 2	DB state
`lock` (*acct*) ;		1200
`read` (*acct*) ;		
	`lock` (*acct*) ;	
`write` (*acct*) ;	↓ Transaction	1100
`unlock` (*acct*) ;	blocked	
	`read` (*acct*) ;	
	`write` (*acct*) ;	900
	`unlock` (*acct*) ;	

- The use of locking lead to a correct (and serializable) schedule.

Lock Modes

- We saw earlier that two **read** operations do not conflict with each other.
- Systems typically use different types of locks (**"lock modes"**) to allow read operations to run concurrently.
  - **read locks** or **shared locks**: mode `S`
  - **write locks** or **exclusive locks**: mode `X`
- Locks are only in conflict if at least one of them is an `X` lock:

	shared (`S`)	exclusive (`X`)
shared (`S`)		×
exclusive (`X`)	×	×

- It is a safe operation in two-phase locking to **convert** a shared lock into an exclusive lock during the lock phase.

Deadlocks

- Like many lock-based protocols, two-phase locking has the risk of **deadlock** situations:

Transaction 1	**Transaction 2**
`lock` (A) ;	
⋮	`lock` (B)
do something	⋮
⋮	do something
`lock` (B) [wait for T_2 to release lock]	⋮
	`lock` (A) [wait for T_1 to release lock]

- Both transactions would wait for each other **indefinitely**.

Deadlock Handling

A typical approach to deal with deadlocks is **deadlock detection**:

- The system maintains a **waits-for graph**, where an edge $T_1 \rightarrow T_2$ indicates that T_1 is blocked by a lock held by T_2.
- Periodically, the system tests for **cycles** in the graph.
- If a cycle is detected, the deadlock is **resolved** by **aborting** one or more transactions.
- Selecting the **victim** is a challenge:
  - Blocking **young** transactions may lead to **starvation**: the same transaction is cancelled again and again.
  - Blocking an **old** transaction may cause a lot of investment to be thrown away.

Deadlock Handling

Other common techniques:

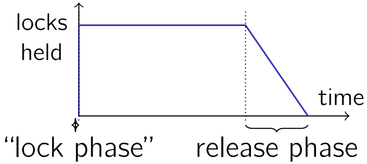
- **Deadlock prevention:** *e.g.*, by treating handling lock requests in an **asymmetric** way:
  - **wait-die**: A transaction is never blocked by an **older** transaction.
  - **wound-wait**: A transaction is never blocked by a **younger** transaction.
- **Timeout:** Only wait for a lock until a timeout expires. Otherwise assume that a deadlock has occurred and **abort**.

E.g., IBM DB2 UDB:

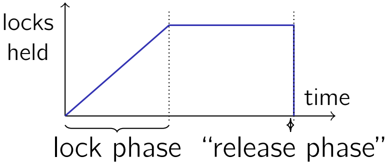
```
db2 => GET DATABASE CONFIGURATION;
  :
  :
Interval for checking deadlock (ms)       (DLCHKTIME) = 10000
Lock timeout (sec)                      (LOCKTIMEOUT) = -1
```

Variants of Two-Phase Locking

- The two-phase locking protocol does not prescribe exactly when locks have to acquired and released.
- Possible variants:



preclaiming 2PL

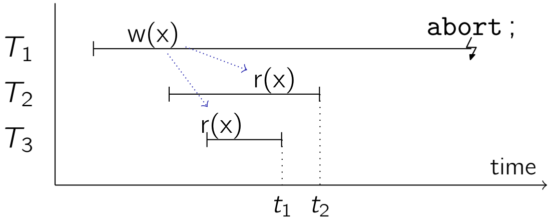


strict 2PL

- ✎ **What could motivate either variant?**

Cascading Rollbacks

Consider three transactions:



- When transaction T_1 aborts, transactions T_2 and T_3 have already read data written by T_1 ($\nearrow$ dirty read, slide 302)
- T_2 and T_3 need to be **rolled back**, too.
- T_2 and T_3 **cannot** commit until the fate of T_1 is known.
- two-phase locking vs. strict two-phase locking

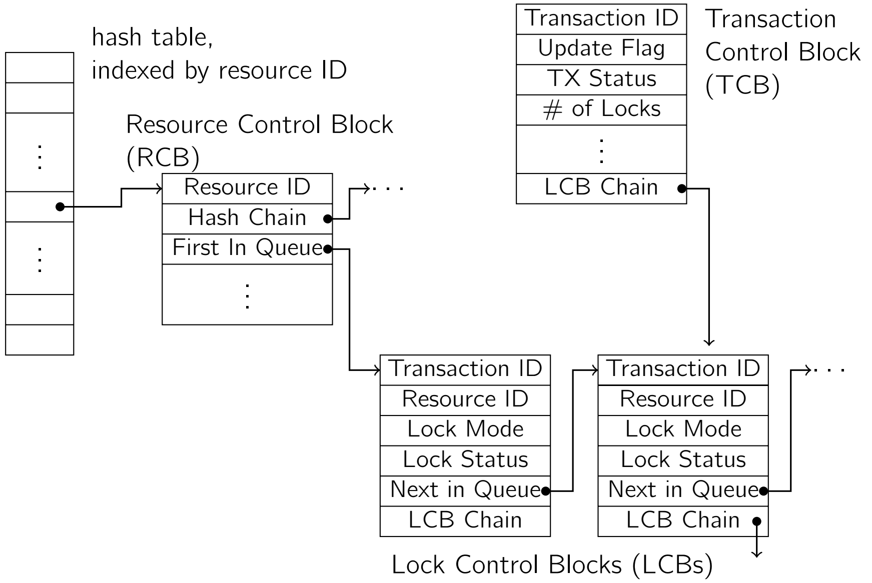
Implementing a Lock Manager

We'd like the Lock Manager to do three tasks very efficiently:

1. Check which locks are currently held for a given **resource** (in order to decide whether another lock request can be granted).
2. When a lock is released, **transactions** that **requested** locks on the **same resource** have to be identified and granted the lock.
3. When a transaction **terminates**, all held locks must be released.

What is a good data structure to accommodate these needs?

Bookkeeping

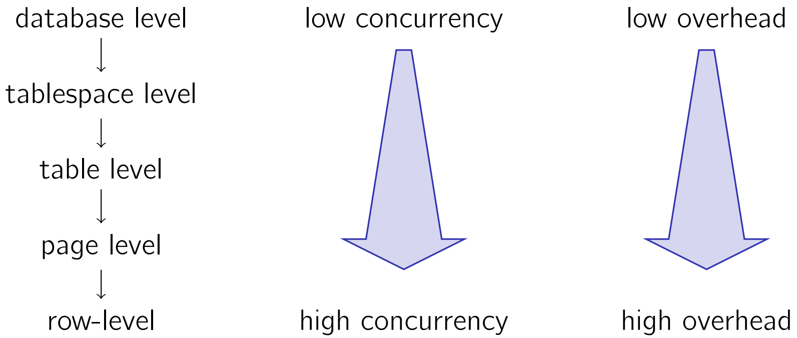


Implementing Lock Manager Tasks

1. The locks held for a given **resource** can be found using a **hash lookup**.
   - Linked list of Lock Control Blocks via ‘First In Queue’/‘Next in Queue’
   - The list contains **all** lock requests, granted or not.
   - The transaction(s) at the **head** of the list are the ones that currently hold a lock on the resource.
2. When a lock is **released** (*i.e.*, its LCB removed from the list), the next transaction(s) in the list are considered for granting the lock.
3. All locks held by a single **transaction** can be identified via the linked list ‘LCB Chain’ (and easily released upon transaction termination).

Granularity of Locking

The **granularity** of locking is a trade-off:



- **Idea: multi-granularity** locking

Multi-Granularity Locking

- Decide the granularity of locks held **for each transaction** (depending on the characteristics of the transaction).
  - A row lock, *e.g.*, for

```
SELECT * FROM CUSTOMERS                                   Q1
 WHERE C_CUSTKEY = 42
```

    and a table lock for

```
SELECT * FROM CUSTOMERS                                   Q2
```

- How do such transactions know about each others' locks?
  - Note that locking is **performance-critical**. Q_2 doesn't want to do an extensive search for row-level conflicts.

Intention Locks

Databases use an additional type of locks: **intention locks**.

- Lock mode **intention share**: `IS`
- Lock mode **intention exclusive**: `IX`
- Conflict matrix:

	S	X	IS	IX
S		×		×
X	×	×	×	×
IS		×		
IX	×	×		

- A lock `I`□ on a coarser level means that there's some □ lock on a lower level.

Intention Locks

Protocol for multi-granularity locking:

1. A transaction can lock any granule g in $\Box \in \{\texttt{S}, \texttt{X}\}$ mode.
2. Before a granule g can be locked in $\Box$ mode, it has to obtain an $\texttt{I}\Box$ lock on **all** coarser granularities than contain g.

Query Q_1 would, *e.g.*,

- obtain an `IS` lock on **table** `CUSTOMERS`
  (also on on tablespace and database) and
- obtain an `S` lock on the **tuple(s)** with `C_CUSTKEY = 42`.

Query Q_2 would place an

- `S` lock on `table CUSTOMERS`
  (and an `IS` lock on tablespace and database).

Detecting Conflicts

Now suppose a write query comes in:

```
UPDATE CUSTOMERS                                    Q3
  SET NAME = 'John Doe'
  WHERE C_CUSTKEY = 17
```

It'll want to place

- an `IX` lock on **table** `CUSTOMER` (and . . .) and
- an `X` lock on the **row** holding customer `17`.

As such it is

- **compatible** with Q_1
  (there's no conflict between `IX` and `IS` on the table level),
- but **incompatible** with Q_2
  (the `S` lock held by Q_2 is in **conflict** with Q_3's `IX` lock).

Consistency Guarantees and SQL 92

Sometimes, some degree of inconsistency may be acceptable for specific applications:

- "Mistakes" in few data sets, *e.g.*, will not considerably affect the outcome of an aggregate over a huge table.
  - $\rightsquigarrow$ Inconsistent read anomaly
- SQL 92 specifies different **isolation levels**.
- *E.g.*,

```
SET ISOLATION SERIALIZABLE;
```

- Obviously, less strict consistency guarantees should lead to increased throughput.

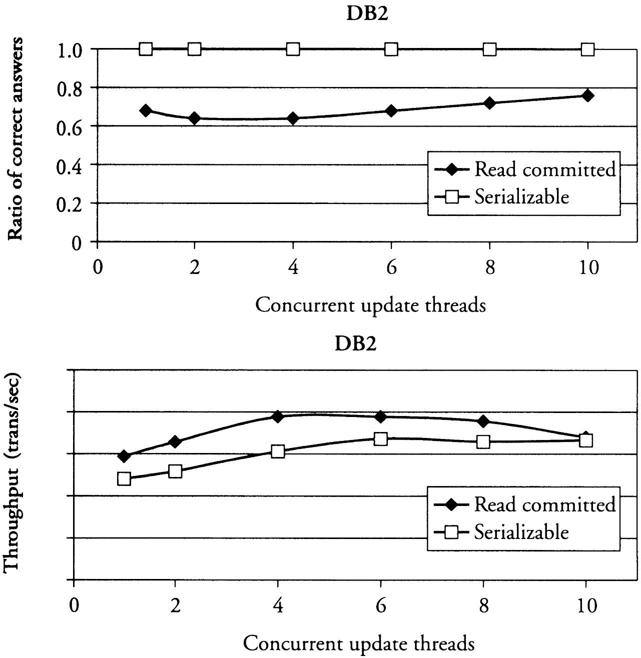
SQL 92 Isolation Levels

read uncommitted (also: ‘dirty read’ or ‘browse’)
: Only **write locks** are acquired (according to strict 2PL).

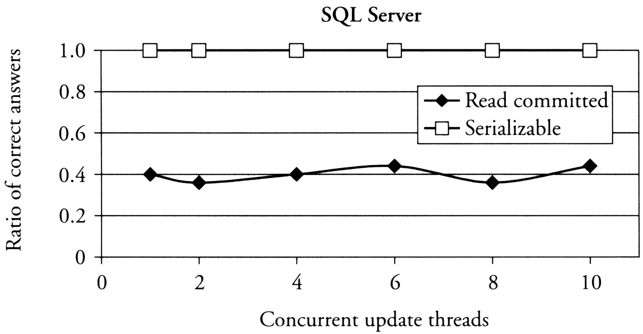
read committed (also: ‘cursor stability’)
: **Read locks** are only held for as long as a cursor sits on the particular row. **Write locks** acquired according to strict 2PL.

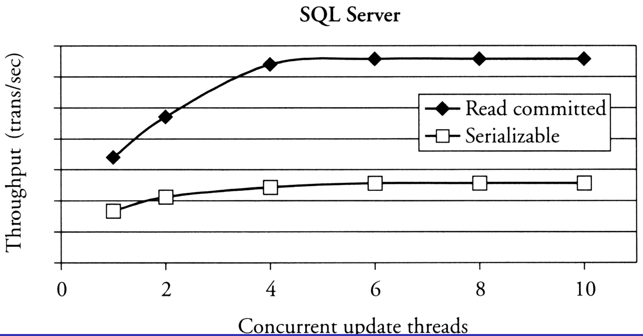
repeatable read (also: ‘read stability’)
: Acquires **read** and **write locks** according to strict 2PL.

serializable
: Additionally obtains locks to avoid **phantom reads**.

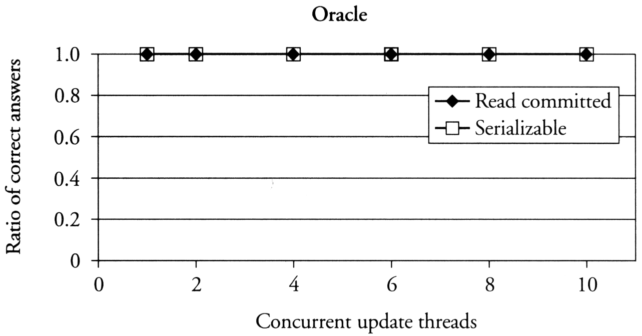


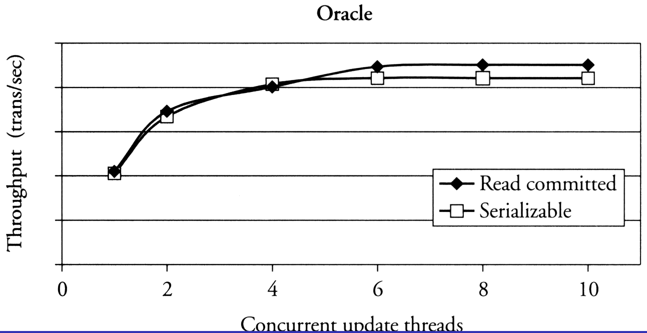
Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.





Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.





Dennis Shasha nad Philippe Bonnet. Database Tuning. Morgan Kaufmann, 2003.

Resulting Consistency Guarantees

isolation level	**dirty read**	**non-repeat. rd**	**phantom rd**
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

- Some implementations support more, less, or different levels of isolation.
- Few applications really need serializability.

Locking and B-Trees

Transaction 1	**Transaction 2**	**Result**

`SELECT COUNT (*)`
`FROM Customers`
`WHERE Name = 'Sam'`
		2

	`INSERT INTO Customers`
	`VALUES (..., 'Sam', ...)`
		ok

`SELECT COUNT (*)`
`FROM Customers`
`WHERE Name = 'Sam'`
		3 $\lightning$

Transaction 1 "sees" the concurrent insert done by Transaction 2.

$\rightarrow$ **Isolation property violated.**

This is an instance of the **phantom problem**.

Avoiding Phantoms

Locking only tuples cannot avoid the phantom problem.

- The tuple added by T_2 is new; T_1 could never have locked it before.
- To avoid the phantom problem, we also have to lock **absent** tuples.

Phantoms can be avoided with:

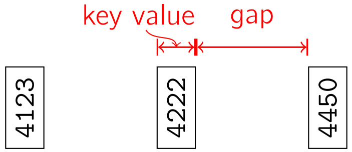
- **Predicate Locking:** For each query, lock the predicates that it uses.

  Representing, finding, and comparing predicates can be difficult and inefficient.

- **Key-Range Locking:** Lock **index entries** that match the predicate.
  - *E.g.*, in the previous example, lock the index key `Sam`.

Key-Range Locking

- Use B-trees to lock **key values**, **not tuples**!
  - → This is somewhat orthogonal to regular data locking.
- In general, we want to lock **ranges** of key values.
  - → Including **absence** of key values.
  - → Lock existing **key values** and **gaps**.

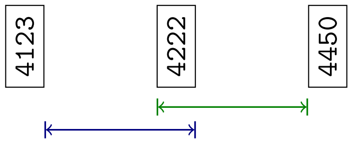


→ The current index content determines which ranges can be locked.

Key-Range Locking

Typically:

- Acquire **one lock** to mean a key value **and** its neighboring gap:



→ **Previous key locking:** $\vdash\!\leftrightarrow\!\dashv$
Lock covers key value *x* and the gap that **follows** *x*.

→ **Next key locking:** $\vdash\!\leftrightarrow\!\dashv$
Lock covers key value *x* and the gap that **precedes** *x*.

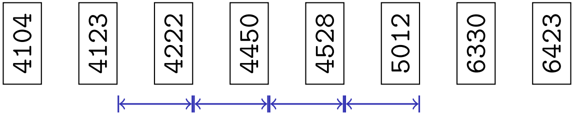
This way, existing key values can be used as lookup keys in the system's **lock manager** (which is typically organized as a hash table).

Reading Transactions

Idea:

- Queries acquire `S` locks for all key ranges that **intersect** with ranges in query predicates.

E.g., scan range [4200, 5000]:

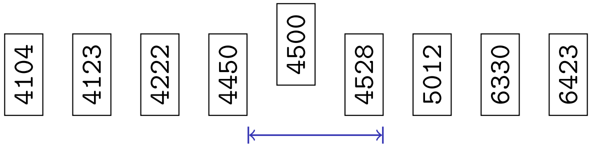


→ Ranges]4123, 4200[and]5000, 5012] locked "too much"!

Inserts

- Inserts need to acquire a lock on the **gap** into which they want to insert.
- Thus, with next key locking: acquire lock on next-largest key.

E.g., insert 4500:



→ Acquire `X` lock on 4528 (which covers range $]4450, 4528]$).

→ If the reading transaction from the previous slide still holds its locks, a conflict on 4528 will be detected (and the insert will have to wait).

→ Insert new key and `X` lock it immediately.

Lock Duration

Readers:

- Keep the range locked until the transaction commits.
  This is to make sure the range can be re-read at any time without seeing phantoms.

Inserts:

- Keep **newly inserted entry** `X` locked until commit time.
  → This prevents others from reading un-committed data.
- The lock on the **next key** (4528 here), however, can be released **immediately**.
  → Acquiring the lock with "instant duration" ensures there is no co-running reader for that range.
  → Once the new key is inserted, readers (or writers) are free to lock the next key (4528), since its associated range (]4500, 4528] now) only covers the gap **without** the newly inserted key.

Why Bother?

This ability to lock with instant duration is very relevant in practice.

- Inserts at the **right end** of a B-tree are a very common pattern.
  - → Next key locking requires an extra $+\infty$ index entry, by the way.
  - → All append queries will lock this $+\infty$ entry.
  - → When the lock on $+\infty$ is an instant lock, other inserts can proceed immediately.

→ Note how this also favors next key locking over previous key locking.

Deletions

To **delete** an entry *x*, the transaction has to obtain

- an `X` lock on the to-be-deleted entry *x*,
  - → Make sure no other transaction still depends on *x*.
  - → The lock is effectively instant, since the transaction is about to remove *x* anyway.
- an `X` lock on *x*'s **next key** until **commit time**.
  - → ✎ **Why?**

Key-Value Locks in Practice

IBM DB2 does not lock index entries explicitly.

- Instead, DB2 performs **data-only locking**.
- A locked tuple **implies** a key-range lock in **all** indexes on the table.
- When checking for lock compatibility, DB2 looks for already held locks, but also considers the **isolation level** of the lock holder.

Data-only locking may lead to unexpected **side effects**:

- *E.g.*, a scan criterion on one column may lead to locks in scattered regions of other attributes.

On the positive side, deriving key-range locks from row locks reduces the number of locks to maintain (and thus the complexity of the lock manager).

Ghost Records

Support for **ghost records** may ease key-range locking considerably.

- **Deletes** will not actually remove the index entry, but only turn the record into a ghost.
- The ghost still represents a valid range boundary (locks can be acquired on ghosts just as on normal records).
- Flipping the ghost bit is merely a form of **value update** of the record.
  - → Value updates do not need range locks as long as they do not modify the key value.

Ghost Records and Inserts

The same advantages also hold for **inserts** if a ghost with the right key value already exists.

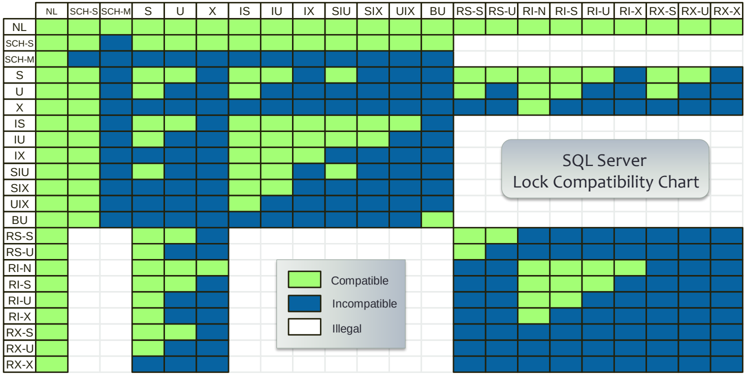
→ Need to lock only the key value itself (neighboring range is often implicit, but not strictly required).

Existence of a matching ghost need **not** be a coincidence.

Trick:

- Invoke a short, **separate transaction** that creates the ghost for us.
- The transaction will have to acquire range locks. But it will commit immediately (and release its locks).

Locking in Practice—SQL Server



SQL Server Lock Compatibility Chart

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
SCH-S	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
SCH-M	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
S	🟩	🟩	🟦	🟩	🟩	🟦	🟩	🟩	🟦	🟩	🟦	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟦	🟩	🟩	🟦
U	🟩	🟩	🟦	🟩	🟦	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟦	🟩	🟩	🟦	🟦	🟩	🟦	🟦
X	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟦
IS	🟩	🟩	🟦	🟩	🟩	🟦	🟩	🟩	🟩	🟩	🟩	🟩	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
IU	🟩	🟩	🟦	🟩	🟦	🟦	🟩	🟩	🟩	🟩	🟩	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
IX	🟩	🟩	🟦	🟦	🟦	🟦	🟩	🟩	🟩	🟦	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
SIU	🟩	🟩	🟦	🟩	🟦	🟦	🟩	🟩	🟦	🟩	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
SIX	🟩	🟩	🟦	🟦	🟦	🟦	🟩	🟩	🟦	🟦	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
UIX	🟩	🟩	🟦	🟦	🟦	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
BU	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
RS-S	🟩	⬜	⬜	🟩	🟩	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦
RS-U	🟩	⬜	⬜	🟩	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟩	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
RI-N	🟩	⬜	⬜	🟩	🟩	🟩	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟩	🟩	🟩	🟩	🟦	🟦	🟦
RI-S	🟩	⬜	⬜	🟩	🟩	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟩	🟩	🟩	🟦	🟦	🟦	🟦
RI-U	🟩	⬜	⬜	🟩	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟩	🟩	🟦	🟦	🟦	🟦	🟦
RI-X	🟩	⬜	⬜	🟩	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟦
RX-S	🟩	⬜	⬜	🟩	🟩	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
RX-U	🟩	⬜	⬜	🟩	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
RX-X	🟩	⬜	⬜	🟦	🟦	🟦	⬜	⬜	⬜	⬜	⬜	⬜	⬜	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦

🟩	Compatible
🟦	Incompatible
⬜	Illegal

NL	No Lock	IX	Intent Exclusive	RI-S	Insert Range-Shared
SCH-S	Schema Stability Lock	SIU	Share with Intent Update	RI-U	Insert Range-Update
SCH-M	Schema Modification Lock	SIX	Share with Intent Exclusive	RI-X	Insert Range-Exclusive
S	Shared	UIX	Update with Intent Exclusive	RX-S	Exclusive Range-Shared
U	Update	BU	Bulk Update	RX-U	Exclusive Range-Update
X	Exclusive	RS-S	Shared Range-Shared	RX-X	Exclusive Range-Exclusive
IS	Intent Shared	RS-U	Shared Range-Update		
IU	Intent Update	RI-N	Insert Range-Null		

sqlbits READPAST & Furious: Locking Blocking and Isolation · Mark Broadbent · sqlcloud.co.uk

Multi-User and Multi-Thread Support

So far we looked at ill effects **between user transactions**.

$\rightarrow$ **Locks on data objects** helped to **isolate transactions**.

Parallel threads might cause additional problems:

$\rightarrow$ Two writers, different data objects, same page $\leadsto$ **corrupted data**.

$\rightarrow$ Locks will not isolate threads that belong to the **same transaction**.

$\rightarrow$ How do we protect **internal data structures** (lock table, buffer pool, etc.)?

- Lock manager can only lock user data objects!

This calls for a mechanism to **isolate threads** (not transactions).

$\rightarrow$ Short-lived, in-memory "locks" or **latches**.
(The term "lock" is reserved for transaction-level locking.)

Page Latches

Latches protect data at a **page granularity**.

→ This has also been called **storage-layer concurrency**.

To achieve **high concurrency**:

- Hold latches **as short as possible**.
- Hold **few latches only** (and/or latch at fine granularities).

In addition:

- Choose a **fast implementation** for latches.
  - → no frills like **deadlock checking**
  - → instead: avoid deadlocks by **coding discipline**

Page Latches (Data Pages)

Example:

- Latches on data pages make page modifications appear as an **atomic operation**.
  - → Protect from, *e.g.*, observing a corrupt page.
- Latching is **in-memory only**.
  - → No I/O while holding a page latch.
  - → Latches are not flushed to disk.
- Only hold **one latch at a time**.
  - → ✎ **Why?**

Latches and B-Trees (Search)

✎ **Order for latch acquisition/release during a B-tree search?**

Latch Coupling

With latch coupling, a thread may hold more than one latch at a time.

→ A **deadlock** still cannot occur:

- Every thread will navigate/acquire latches top-down.
- All threads acquire latches in same order → no deadlock.

Latches and B-Trees (Updates)

Updates to B-trees operate **bottom-up**.

Possible strategy:

- Acquire read latches as during search, but **keep** all latches.
  - → Ensure that the parent (grandparent, ...) is still the parent during bottom-up processing.
- Acquire **write latches bottom-up**.
  - → Latch conversion: read latch $\rightsquigarrow$ write latch.
  - → Write-latch parent before splitting a child.
- Release write latches when all necessary changes to the page are applied; release ancestor read latches when no more splits are necessary.

If the B-tree implementation uses **sibling pointers**, additional locks may have to be acquired on **sibling nodes**.

Latches and B-Trees (Updates)

The strategy on the previous slide guarantees **correctness**.

- All tree modifications are write-latched, and released latches always leave behind a consistent B-tree.

But:

The strategy entails a danger of **deadlocks**.

- **Searches** acquire their latches **top-down**.
- **Updates** acquire their (write) latches **bottom-up**.

Remember: We want latches to be lightweight $\rightarrow$ no deadlock checking.

Latching B-Trees Without Deadlocks

Deadlocks can be **avoided** when **all** operations acquire latches either top-down or bottom-up.

Thus:

- Let insert operations acquire write latches right away.

✎ **What do you think of this strategy?**

Latching B-Trees Without Deadlocks

Chances that a write latch on a parent is actually needed are **really low**.

$\rightarrow$ *E.g.*, B-tree with up to 100 entries/node $\rightarrow$ chance of a split: 2 %

Idea: (Try to) keep write latch only when really necessary.

- During tree descent, observe **space utilization** in visited nodes.
- When a node *n* has **enough space** to hold another entry, *n* definitely won't have to be split.
- For such nodes *n*, the **parent node** *p* will not have to be updated.
  - $\rightarrow$ *p* is then called **split safe**.
- The latch on that parent *p* can be released safely.

Lock Coupling Protocol (Variant 1)

readers

```
place S lock on root ;
current ← root ;
while current is not a leaf node do
    place S lock on appropriate son of current ;
    release S lock on current ;
    current ← son of current ;
```

writers

```
place X lock on root ;
current ← root ;
while current is not a leaf node do
    place X lock on appropriate son of current ;
    current ← son of current ;
    if current is safe then
        release all locks held on ancestors of current ;
```

Increasing Concurrency for Common Scenarios

- Even with lock coupling there's a considerable amount of locks on inner tree nodes (reducing concurrency).
- Chances that inner nodes are actually affected by updates are **very small**.
  - Back-of-the-envelope calculation:
    $d = 50 \Rightarrow$ every 50th insert causes a split ($2\,\%$ chance).
- An insert transaction could thus optimistically assume that no leaf split is going to happen.
  - On inner nodes, only read locks acquired during tree navigation (plus a write lock on the affected leaf).
  - If assumption is wrong, re-traverse the tree and obtain write locks.

Lock Coupling Protocol (Variant 2)

Modified protocol for **writers**:[19]

```
place S lock on root ;
current ← root ;
while current is not a leaf node do
    son ← appropriate son of current ;
    if son is a leaf then
        place X lock on son ;
    else
        place S lock on son ;
    release lock on current ;
    current ← son ;
if current is unsafe then
    release all locks and repeat with protocol Variant 1 ;
```

[19]Reader protocol remains unchanged.

B-Tree Latching and High Concurrency

- Deciding **split safety** can be difficult for **variable-length keys**.
- The strategy on the previous slide thus has to be **very conservative**.
- Effectively, many latches are still held **unnecessarily**.

Ways to improve concurrency (by holding fewer latches):

- **split proactively**: When a node is not split safe, split it right away. At least the system then suffers the unnecessary latch only once.
- **repeated root-to-leaf passes**: Descend with only read latches first. Re-traverse the tree with full write latches when a split is necessary.
- **giveup technique**: hold only single-node read latches (and risk inconsistencies); detect conflicts and re-traverse in case of a conflict.
- **B$^{\text{link}}$-trees**: slightly relax some B-tree rules.

Giveup Technique

A deadlock can only arise when a thread acquires (or tries to) a new latch before releasing an old one.

→ A thread that always only holds a **single latch** at a time can never deadlock.

Search routine with only a single latch held at any time:

```
$n \leftarrow$ root page ;
while $n$ is not a leaf do
    read-latch $n$ ;
    determine child $n'$ of $n$ ;
    un-latch $n$ ;
    $n \leftarrow n'$ ;
read-latch $n$ ;
return matching records (if any)
  ;
un-latch $n$ ;
```

Giveup Technique

There is a **risk of inconsistencies** when only a single latch is held.

- Between determining the child page n' and latching it, a **concurrent update** might have split n'.
- The search might **miss** an entry that is now on a new page.

Thus: Detect when a conflicting update has happened.

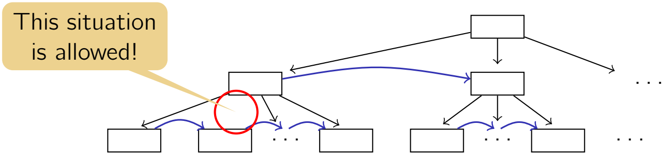
- When descending, remember the two **separator keys** k_{min} and k_{max} in n that guided to n'.
- When looking at n', first check whether k_{min} and k_{max} are still the correct separator keys for that page.
  - $\rightarrow$ Keep copies of parent's separator keys in each node.
  - $\rightarrow$ Such copies are also called **fence keys**.
- If a conflict is detected, **abort and re-try** a moment later.

B$^{\text{link}}$-Trees

Lehman and Yao[20] proposed a B-tree variant, usually referred to as **B$^{\text{link}}$-tree**, where writes must latch at most two nodes at a time.

Idea:

- Assume a B-tree with **forward sibling pointers**.
- **Relax B-tree structure**: Allow parent $\rightarrow$ child to be missing when the child is reachable via the sibling pointer of its predecessor.



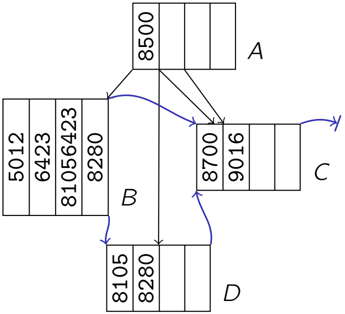
[20]Lehman and Yao. *Efficient Locking for Concurrent Operations on B-Trees*, TODS 6(4), 1981.

B^{link}-Trees

With the relaxation, node splitting and parent updates can be separated.

```
1  latch & read page $B$ ;
2  create new page $D$ and latch it
     ;
3  populate page $D$ ;
4  set next pointer $D \rightarrow C$ ;
5  un-latch $D$ ;
6  set next pointer $B \rightarrow D$ ;
7  adjust content of $B$ ;
8  un-latch $B$ ;

9  latch & read $A$ ;
10 adjust content of $A$ ;
11 un-latch $A$ ;
```



→ Lines 9–11 can be deferred to a later time.

B^{link}-Trees

With the relaxation stated before, lines 1–8 already represent a correct B^{link}-tree.

- Lines 9–11 are, in a sense, only applied for performance reasons.

The parent could be updated also at a **later time**:

- As a “clean-up process” triggered when the update has completed.
- When the next search traverses the tree.
- During database maintenance.

In fact, even the page latches can be avoided when pointer updates and record deletions can be done **atomically**.

PostgreSQL, *e.g.*, uses B^{link}-Trees.

Optimistic Concurrency Control

- So far we've been rather **pessimistic**:
  - we've assumed the worst and prevented that from happening.
- In practice, conflict situations are not that frequent.
- **Optimistic concurrency control:** Hope for the best and only act in case of conflicts.

Optimistic Concurrency Control

Handle transactions in **three phases**:

1. **Read Phase.** Execute transaction, but do **not** write data back to disk immediately. Instead, collect updates in a **private workspace**.
2. **Validation Phase.** When the transaction wants to **commit**, test whether its execution was correct. If it is not, **abort** the transaction.
3. **Write Phase.** Transfer data from private workspace into database.

Validating Transactions

Validation is typically implemented by looking at transactions'

- **Read Sets** $RS(T_i)$**:** (attributes read by transaction T_i) and
- **Write Sets** $WS(T_i)$**:** (attributes written by transaction T_i).

backward-oriented optimistic concurrency control (BOCC):

Compare T against all **committed** transactions T_c.
Check **succeeds** if

$$T_c \text{ committed before } T \text{ started} \quad \textbf{or} \quad RS(T) \cap WS(T_c) = \emptyset \ .$$

forward-oriented optimistic concurrency control (FOCC):

Compare T against all **running** transactions T_r.
Check **succeeds** if

$$WS(T) \cap RS(T_r) = \emptyset \ .$$

Multiversion Concurrency Control

Consider the schedule

$$r_1(x), w_1(x), r_2(x), \overset{\color{red}{t}}{\downarrow} w_2(y), r_1(y), w_1(z) \quad .$$

✎ **Is this schedule serializable?**

- Now suppose when T_1 wants to read y, we'd still have the "old" value of y, valid at time t, around.
- We could then create a history equivalent to

$$r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z) \quad ,$$

which is **serializable**.

MVCC in Practice—Read-Only MVCC

A simple form of MVCC is the **Read-Only MVCC**:

- **Read/write transactions** use concurrency control as before (*e.g.*, 2PL)
- **Read-only transactions** do not acquire any locks. For each read operation $r(x)$ of a read-only transaction T_{RO}, read the version of x that existed when T_{RO} started.

That is, read-only transactions see a **snapshot** of the database as of the time when they started.

Problem:

- Must mark each data object with **commit time** of transaction.

“Read-Consistency” MVCC

Oracle implements “read committed” ($\nearrow$ slide 336) using the **“Read-Consistency” protocol**:

- **read-only transactions** are treated as in the Read-Only protocol.
- **writes in read/write transactions** acquire long-duration write locks.
- **reads in read/write transactions** do not acquire read locks; they read the most recent version of any data object.

$\rightarrow$ Reads only return committed values ($\leadsto$ read committed).

$\rightarrow$ Read-only transactions see consistent state (unlike in read committed).

$\rightarrow$ Readers never block writers and vice versa.

Snapshot Isolation

A modification of the same idea yields **snapshot isolation**.

- All **reads** of any transaction T see the version that was current when T started.
- All **writes** must satisfy the **"first committer wins"** property. A transaction T is allowed to commit only if there is no other transaction T' such that
  - (a T' committed between the start and commit time of T **and**
  - (b T' updated a data object that T also updated.

  Otherwise, T aborts.

To test "first committer wins," compare write sets of T and T'.

- Snapshot isolation is implemented, *e.g.*, in Oracle, SQL Server, PostgreSQL

Wrap-Up

ACID and Serializability

To prevent from different types of **anomalies**, DBMSs guarantee **ACID properties**. **Serializability** is a sufficient criterion to guarantee **isolation**.

Two-Phase Locking

Two-phase locking is a practicable technique to guarantee serializability. Most systems implement **strict 2PL**. SQL 92 allows explicit **relaxation** of the ACID isolation constraints in the interest of performance.

Concurrency in B-trees

Specialized protocols exist for concurrency control in B-trees (the root would be a locking bottleneck otherwise).