CMPUT391

Overview of Transaction Processing

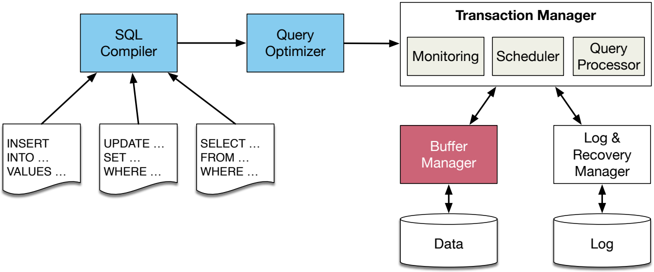
Instructor: Denilson Barbosa

University of Alberta

October 12, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

Simplified DBMS architecture



These notes look into the theory and algorithms to ensure the correct and concurrent execution of queries and updates.

First, we consider the execution of transactions in isolation. Then we look at the concurrent execution of transactions.

What is a transaction?

A sequence of operations to fulfill a user request that may involve reading and/or writing data, and computing values from the data:
- queries are “read-only” transactions that do not modify the database;
- SQL `INSERT`, `DELETE`, and `UPDATE` commands typically read some data, compute new values, and write (or overwrite) some data back to the database.

Every transaction is **logically independent** of other transactions.

We say a transaction:
- **COMMITS** if it executes in its entirety; or
- **ABORTS** if it cannot be executed in its entirety by the DBMS.

Every update command (received from the command line interface or via an API call from an application program) is *implicitly* wrapped in a transaction.

```
> INSERT INTO R VALUES (2),(1);
```

is executed as

```
BEGIN TRANSACTION;
INSERT INTO R VALUES (2),(1);
COMMIT;
```

ACID Transactions

Most relational systems implement the ACID transaction model:

A TOMICITY: every transaction must either execute in its entirety or not at all

C ONSISTENCY: every transaction must leave the database in a consistent state

I SOLATION: no transaction can interfere with the execution of another transaction

D URABILITY: if a transaction **COMMITS**, all its changes to the database must become permanent

Atomicity

Most transactions involve many read/write/compute operations and take some time to execute. Atomicity means that the DBMS cannot execute just *some* of the operations in a transaction.

In other words, the DBMS must do one of:

- Execute **all operations** in a transaction and `COMMIT` it, making its effects permanent.

- Leave the database unchanged and `ABORT` the transaction.

Why would an operation fail?

There are many reasons; e.g., attempting to update the database in a way that violates a constraint, reaching a timeout limit for user input, etc.

Consistency

Recall that a database instance is **consistent** if (and only if) it satisfies all constraints defined in its schema:

- Domain, `UNIQUE`, and `NOT NULL` constraints.
- Primary and foreign key constraints.
- Complex constraints defined using triggers.

Domain, unique, and key constraints are checked first, for each tuple inserted, deleted or modified by the transaction.

Next, the DBMS executes all triggers associated with the tables being modified by the transaction.

If no violations or exceptions are detected, the transaction is allowed to commit.

What exactly happens when some statements fail?

What happens when some statements in a transaction are successful while others fail **depends on the DBMS**.

```sql
1 CREATE TABLE T(a INT, PRIMARY KEY a);
2 BEGIN TRANSACTION;
3     INSERT INTO T VALUES (1);
4     INSERT INTO T VALUES (2);
5     INSERT INTO T VALUES (1);
6     INSERT INTO T VALUES (3);
7 COMMIT;
```

In the code above, most systems will automatically rollback[1] the transaction, leaving the table empty.

SQLite, on the other hand, **does not** automatically rollback updates[2]. Instead, it leaves it to the application to decide what to do upon the constraint violation in line 5.

[1] Rollback means reverting the database objects to their original values.

[2] `https://sqlite.org/lang_transaction.html`

Isolation

In the strictest sense, isolation means that every transaction must execute as if it was the only transaction in the system.

In other words, isolation means that once a transaction starts executing, no other transaction should be allowed to modify the data that will be used by that transaction.

Isolation and Concurrency are conflicting goals

Ensuring isolation may prevent opportunities for concurrency.

With SQL, the programmer can choose among different levels of isolation, and thus fine tune of the concurrency/isolation trade-off.

Durability

Durability means that the effects of a transaction become permanent if (and only if) it commits.

Durability is enforced by:

- Ensuring dirty buffers and the log are written to storage.
- Using reliable storage (redundant storage, RAID system, etc.).

Crash Recovery

In the event of a crash (e.g., power failure), the DBMS must be able to restore the database to reflect all changes made by all transactions that committed before the crash.
This is done by processing the entries in the log.

Transaction Execution

Transactions inside programs

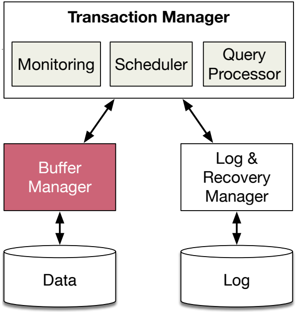
Transactions can be as simple as a command issued from the command line interface of the DBMS, or as complex as a series of SQL statements mixed with arbitrary computations by a program.

```
...
char *stmt = "UPDATE␣R␣SET␣a=?␣where␣id=?;";
sqlite3_prepare_v2(db, stmt, -1, &stmt_q, 0);

sqlite3_exec(db, "BEGIN␣TRANSACTION;", NULL, NULL, NULL);
  x = f(...); //compute some value x
  _id = input(...); //ask user for input
  sqlite3_bind_int64(stmt_q, 1, (sqlite3_int64), x);
  sqlite3_bind_int64(stmt_q, 2, (sqlite3_int64), _id);
  if (rc = sqlite3_step(stmt_q)) != SQLITE_ROW) ...
  ...
  //another query or update can go here...
sqlite3_exec(db, "COMMIT;", NULL, NULL, NULL);
...
```

Every operation of the transaction that needs to read or write data inside the DBMS is monitored and logged, to ensure the correct execution of the transaction.

The DBMS must also schedule the execution of such requests to optimize performance.



Keep in mind that many transactions can execute concurrently, and the DBMS needs to manage all of them simultaneously.

Running example

Consider this oversimplified banking situation:

Holder

id	Name
1234	Hurit
7564	Nuttah
7890	Kimi

Account

	holder	type	balance
t_1	7564	checking	75
	1234	savings	50
t_2	7564	investment	40

And assume Nuttah requests to transfer money from her checking account into her investments.

```sql
BEGIN TRANSACTION; -- transfer between accounts

UPDATE Account SET balance = balance - 20
WHERE holder = 7564 AND type = 'checking';

UPDATE Account SET balance = balance + 20
WHERE holder = 7564 AND type = 'investment';

COMMIT;
```

Doing the transaction **atomically** prevents her losing money!

The attributes modified by the transaction are called the **database elements** of the transaction. In some texts, these are defined as entire tuples.

Holder

id	Name
1234	Hurit
7564	Nuttah
7890	Kimi

Account

	holder	type	balance
t_1	7564	checking	75
	1234	savings	50
t_2	7564	investment	40

These are the database elements that need to be modified by our example transaction:

- The `balance` attribute on tuple (with id) t_1.
- The `balance` attribute on tuple (with id) t_2.

Database elements, buffers, I/O, ...

Recall the transfer unit for I/O operations in the DBMS are *disk blocks*. Thus, before a database element can be modified, the block containing it must be retrieved from disk.

DBMS I/O primitives:

- `input(X)`: copies the disk block containing element `X` to a memory buffer
- `output(X)`: copies the memory buffer with element `X` to disk
- `v = read(X)`: copies the value of element `X` from the memory buffer into local variable v in the transaction program
- `write(X,v)`: copies the value of variable v in the local address space of the transaction to element `X` in the memory buffer

A call to either `v = read(X)` or `write(X,v)` calls `input(X)` if the database element `X` isn't already in memory.

Holder

id	Name
1234	Hurit
7564	Nuttah
7890	Kimi

Account

	holder	type	balance
t_1	7564	checking	75
	1234	savings	50
t_2	7564	investment	40

Focusing only on the I/O and data modification operations, the transaction is abstracted as the following "program" that will be executed by the DBMS.

Here we are using the $\rightsquigarrow$ symbol to abstract the "query" part of the update statement that finds the tuple with the corresponding id.

```
BEGIN TRANSACTION;
A ⇝ t1.balance;
B ⇝ t2.balance;
v = read(A);
v := v - 20;
write(A,v);
v = read(B);
v := v + 20;
write(B,v);
output(A);
output(B);
COMMIT;
```

Logging and Crash Recovery

Atomicity and Durability with logging

There are two ways the DBMS can violate the atomicity principle:

- When some write operation of a transaction that commits is not made persistent.
- When some write operation by a transaction that aborts is made persistent.

Because transactions take time to complete, and computers do crash, the DBMS keeps a log of the operations performed by a transaction.

If the DBMS crashes during the execution of a transaction, the crash recovery system uses the log to ensure atomicity.

Idea: save the values the database elements had before the transaction started in the log. If there is a problem, revert to them.

Rules

(1) On a call to `write(X,v)` by transaction T_j, write to the log the **old** value of database element `X`.

(2) Only write `<COMMIT` T_j`>` to the log **after** all new values are written (through a call to `output(·)`)

Write Ahead Logging (WAL)

Write all old values to the log before writing any new values to disk.

Undo Logging Example

Step	Action	v	Memory		Disk		Log
			A	B	A	B	
1					75	40	<START T_1>
2	`v = read(A)`	75	75		75	40	
3	`v := v-20`	55	75		75	40	
4	`write(A,v)`	55	55		75	40	<T_1, A, 75>
5	`v = read(B)`	40	55	40	75	40	
6	`v := v+20`	60	55	40	75	40	
7	`write(B,v)`	60	55	60	75	40	<T_1, B, 40>
8	`output(A)`	–	55	60	55	40	
9	`output(B)`	–	55	60	55	60	
10	**COMMIT**	–	55	60	55	60	<COMMIT T_1>

log has old values

table has new values

Crash Recovery with Undo Logging

Algorithm 1 database recovery from Undo Log after system crash

Read the log from the end towards the beginning
for each log entry $\langle e_i \rangle$ **do**
    if $\langle e_i \rangle$ = `<COMMIT` T_j`>` or $\langle e_i \rangle$ = `<ABORT` T_j`>` **then** *% T_j has been accounted for*
        mark T_j as completed
    else
        if $\langle e_i \rangle$ = `<START` T_j`>` and T_j is not completed **then** *% must undo T_j*
            read the log forward from this point
            for each log entry <T_j, `X`, v> **do**
                write v as the value of `X` on the database
            write `<ABORT` T_j`>` in the log *% so we don't re-do it after another crash*

Atomicity with Undo Logging

Proving that the Undo Logging protocol ensures atomicity requires showing that, with or without a system crash, no database element X has a value v written by a transaction T_j that **did not commit.**[3]

Case 1: there was no system crash

- T_j aborted because of some of its logical operations failed, **before** any `output(X)` **could have been called**.

Case 2: there was a system crash

- Even if the new value of X was flushed to disk before the crash, line 9 of the crash recovery algorithm reverts X to the original value.

[3] Completing the proof to show that all writes of committed transactions are made persistent is left as an exercise.

Undo Logging is I/O Intensive

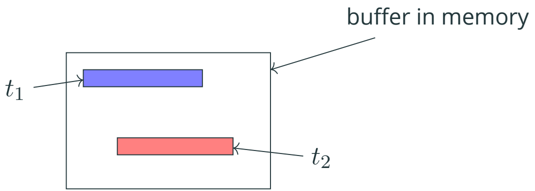
Step	Action	v	Memory		Disk		Log
			A	B	A	B	
1					75	40	<START T_1>
2	`v = read(A)`	75	75		75	40	
3	`v := v-20`	55	75		75	40	
4	`write(A,v)`	55	55		75	40	<T_1, A, 75>
5	`v = read(B)`	40	55	40	75	40	
6	`v := v+20`	60	55	40	75	40	
7	`write(B,v)`	60	55	60	75	40	<T_1, B, 40>
8	`output(A)`	–	55	60	55	40	
9	`output(B)`	–	55	60	55	60	
10	`COMMIT`	–	55	60	55	60	<COMMIT T_1>

I/O before commit

Undo logging is I/O intensive

- Transactions only become "permanent" after they commit in the log;
- but undo logging requires **writing the data before** committing in the log.

Buffer management with Undo Logging



Suppose transactions T_1, T_2 modify **different tuples** t_1 and t_2 in the **same block** (loaded to the same memory buffer), and that T_1 is ready to write its changes to disk and then commit.

Because I/O is performed one page at a time, writing the changes of T_1 will also write **dirty** data created by T_2.

The DBMS must hold off on T_1's commit until T_2 is ready to commit. Otherwise, if T_2 aborts later on, the DBMS will have to bring the data back from disk to undo the change.

Redo Logging

Idea: save the results of the transaction to the log and commit the transaction as soon as the log has all new values. If there is a problem, *replay* the log.

Rules:

(1) on a call to `write(X,v)` by the transaction, write to the log the **new** value of database element `X`

(2) only call `output(X)` after the transaction is committed on the log and the log has been *flushed*

Write Ahead Logging (WAL)

Write all new values of the database elements to the log before writing them to disk.

Redo Logging Example

Step	Action	v	Memory		Disk		Log
			A	B	A	B	
1					75	40	<START T_1>
2	`v = read(A)`	75	75		75	40	
3	`v := v-20`	55	75		75	40	
4	`write(A,v)`	55	55		75	40	<T_1, A, 55>
5	`v = read(B)`	40	55	40	75	40	
6	`v := v+20`	60	55	40	75	40	
7	`write(B,v)`	60	55	60	75	40	<T_1, B, 60>
8	**COMMIT**		55	60	75	40	<COMMIT T_1>
9	`output(A)`		55	60	55	40	
10	`output(B)`		55	60	55	60	

log has new values

table has new values

Crash Recovery with Redo Logging

Algorithm 2 database recovery from Redo Log after system crash

Scan the log to find all transactions that started and all transactions that committed

Read the log from the beginning towards the end

for each log entry $\langle e_i \rangle$ **do**

    if $\langle e_i \rangle$ = <T_j, X, v> **and** T_j has committed **then** *% must redo T_j*

        write v as the value of X on the database

for each uncommitted transaction T_i **do**

    write <ABORT T_i> in the log *% so that we ignore this transaction in the future*

flush the log

Atomicity with Redo Logging

Proving that the Redo Logging protocol ensures atomicity requires showing that, with or without a system crash, no database element X has a value v written by a transaction T_j that **did not commit.**[4]

Case 1: there was no system crash

- T_j aborted because some of its logical operations failed, **before** any `output(X)` **could have been called**.

Case 2: there was a system crash

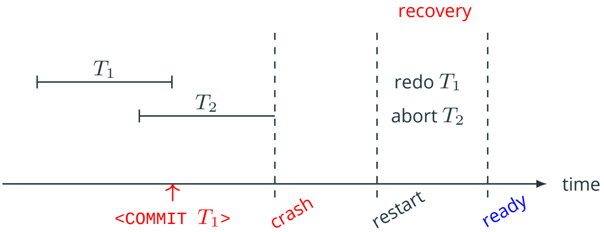
- Line 5 of the crash recovery algorithm, which does the writing, does not apply to transaction T_j because it did not commit in the log.

[4] Completing the proof to show that all writes of committed transactions are made persistent is left as an exercise.

Re-doing a transaction already done?

Because (with REDO logging) transactions commit in the log **before** writing changes to disk, the DBMS can never know (based on the log alone) if the results of a transaction are on disk or not.

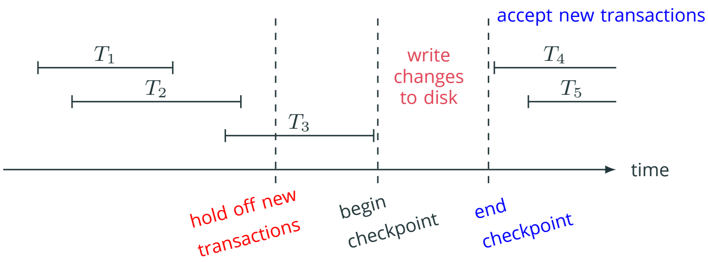
Example:



The DBMS **must redo** T_1 (and every other committed transaction in the log) after a crash, **even if it had written its changes to disk**.

Periodic checkpointing, discussed next, solves this problem.

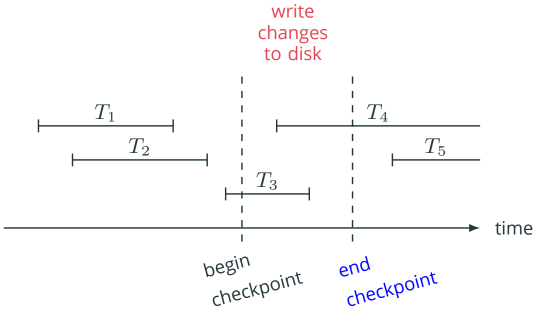
Quiescent[5] checkpointing



When the DBMS decides to do a checkpoint, it: (1) **stops** accepting transactions; (2) writes to disk all changes by previous transactions; and then (3) resumes.

[5] To **quiesce** is to pause or alter a device or application to achieve a consistent state, usually in preparation for a backup or other maintenance.
`https://en.wikipedia.org/wiki/Quiesce`

A **Nonquiescent** checkpoint applies only to transactions that **committed before the checkpoint started**.

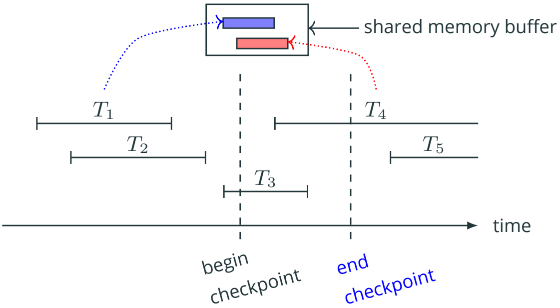


In the example above, changes by T_1 and T_2 are forced to disk, and they become “100%” permanent.

All other transactions that commit will be re-done if there is a crash before the next checkpoint.

Buffer management with Redo Logging

What if a committed transaction shares a buffer with an uncommitted transaction excluded from the checkpoint?



The checkpoint cannot proceed until T_4 commits! Alternatively, the DBMS may hold off **starting** T_4 (only) until the checkpoint is complete.

Summary

The logging strategies we've seen so far...

- Undo logging is **I/O intensive**: write changes before committing.
- Redo logging is **memory intensive**: write changes after committing.

In both cases, sharing buffers can be difficult, forcing the DBMS to hold off the execution of some transactions.

Long-running transactions are especially problematic: they need a lot of memory with Redo logging, and may hold off other transactions for a long time.

Undo/Redo Logging

Arriving at the best of both worlds...

Rules:

(1) on `write(X,v)`, write **both** the old and the new values of `X` on the log

(2) flush the log immediately after the transaction commits

(3) flush dirty buffers anytime

Write Ahead Logging (WAL)

For every database element, the log must have its old and new values **before** the dirty buffer with that element is flushed.

Undo/Redo Logging Example

Step	Action	v	Memory A	Memory B	Disk A	Disk B	Log
1					75	40	<START T_1>
2	`v = read(A)`	75	75		75	40	
3	`v := v-20`	55	75		75	40	
4	`write(A,v)`	55	55		75	40	<T_1, A, 75, 55>
5	`v = read(B)`	40	55	40	75	40	
6	`v := v+20`	60	55	40	75	40	
7	`write(B,v)`	60	55	60	75	40	<T_1, B, 40, 60>
10	`output(B)`		55	60	55	60	
9	**COMMIT**		55	60	55	60	<COMMIT T_1>
10	`output(A)`		75	40	55	60	

log has old values

log has new values

table has new values

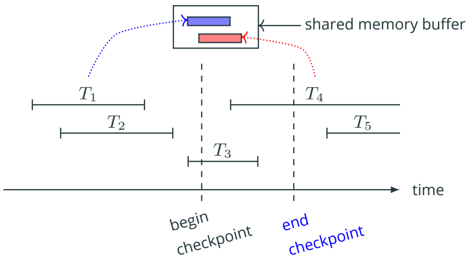
Crash Recovery with Undo/Redo Logging:

(1) read the log backwards until a checkpoint;
(2) find out which transactions completed, and which didn't;
(3) **undo** all transactions that did not complete; mark them as aborted in the log;
(4) **redo** all transactions that committed but started since the most recent *checkpoint*.

Atomicity with Undo/Redo Logging:

A similar argument to the previous protocols works.

Checkpointing with Undo/Redo logging



With Undo/Redo logging we are free to write to disk anytime we want, regardless of the state of the transaction.

We can flush the shared buffer with the uncommited changes of T_2 during the checkpoint of T_1, because if T_2 aborts we have the old value in the log to perform the undo.

What else about logging?

Backups

The log can be used to enable incremental backups of the database that work pretty much like checkpointing, except that instead of flushing dirty buffers to disk, the system *replays* the log on the backup database.

Reusing the log

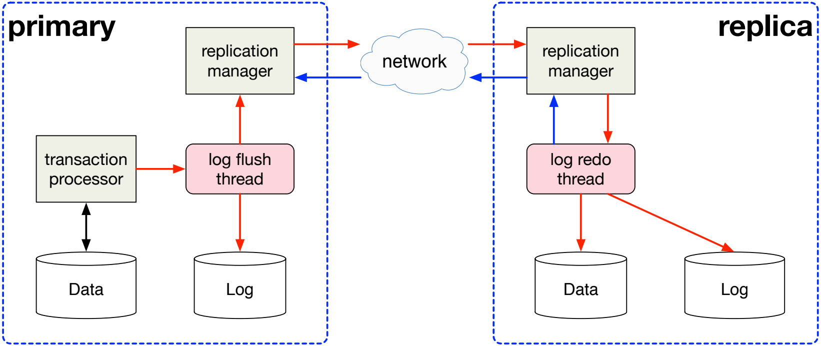
The log file can be truncated periodically, once all transactions are safely preserved (e.g., after backups or checkpoints).

Some systems use a fixed-sized circular buffer for the log.

Replication with Undo/Redo Logging

Redo Logs can be used to keep a replica database up-to-date:

- the master sends the log entries while the replica periodically acknowledges which transactions have been replicated
- the replica can take over on a failure of the master system



Logging and Durability

Two sides to the story:

(1) Recovering from a crash (logging).
(2) Preventing/recovering from media failure.

Media **reliability** can be improved by:

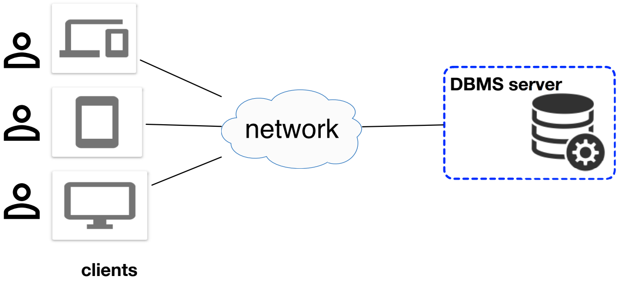
(1) Buying better hardware.
(2) Adding **redundancy**.

Many forms of redundancy:

(1) RAID: having the same data written to multiple disks.
(2) Replication: having a DBMS act as a hot backup of another.

Isolation

Concurrent access to the database



Most databases are used concurrently by many users (e.g., think of Bear Tracks), posing a mix of queries and updates at the same time.

The DBMS must decide the order in which the transactions (queries or updates) are executed, ensuring the integrity of the data at all times.

Running example

Assume we have two transactions now:

- T_1 transfers from checking into investment as before; and
- T_2 accrues 10% interest in the investment.

The correct execution of the transactions now goes beyond ensuring atomicity with the help of the log.

Because the two transactions modify the same database element (the balance in the investment account), we say there is a race condition between them.

If both transaction attempt to modify it concurrently, the DBMS needs to decide who gets to modify that element first, in a way that the account holder does not lose money.

Serial Schedule

A schedule of two or more transactions is **serial** if every transaction executes to completion before the next transaction starts.

T_1	T_2	A	B
		75	40
`v = read(A); v := v-20`			
`write(A,v)`		55	
`v = read(B); v := v+20`			
`write(B,v)`			60
COMMIT		55	60
	`t = read(B); t := t*1.1`		
	`write(B,t)`		66
	COMMIT	55	66

Ideal form of isolation: if no two transactions overlap, they cannot interfere with one another.

Serial execution of transactions is not the same as *deterministic* execution of transactions:

- If there are two or more transactions ready for execution, the order in which they are picked determines the outcome.

T_1	T_2	A	B
		75	40
	`t = read(B); t := t*1.1`		
	`write(B,t)`		44
	`COMMIT`	75	44
`v = read(A); v := v-20`			
`write(A,v)`		55	
`v = read(B); v := v+20`			
`write(B,v)`			64
`COMMIT`		55	64

Why add concurrency?

Allowing multiple transactions to run concurrently brings opportunities for parallelism, which leads to faster response times for the users:

- E.g., while one transaction waits for user input (e.g., credit card information), another transaction can use the CPU.

Conflicting Goals:

(1) Users want as much parallelism and concurrency as possible.

(2) As long as it does not interfere with their work.

Problems with uncontrolled concurrency

#1: the lost update problem.

Happens when a transaction overwrites the changes made by another.

T_1	T_2	A	B
		75	40
`v = read(A); v := v-20`			
`write(A,v)`		55	
`v = read(B); v := v+20`			
	`t = read(B); t := t*1.1`		
	`write(B,t)`		44
	`COMMIT`		
`write(B,V)`			60
`COMMIT`		55	60

lost transaction

 # 2: the uncommitted dependency problem.

Happens when a transaction is allowed to update an element using an uncommitted value of another element written by a transaction that later one aborts.

T_1	T_2	A	B
		75	40
`v = read(A); v := v-20`			
`write(A,v)`		55	
dirty read	`t = read(B); t := t*1.1`		
	`write(B,t)`		44
`v = read(B); v := v+20`			
	ABORT		
`write(B,v)`			64
COMMIT			
		55	64

 # 3: the inconsistent analysis problem.

Happens when a transaction and a query both read/write the same elements concurrently.

Example: transaction T_3, a query that returns the sum of all funds owned by Nuttah.

```
SELECT SUM(balance)
FROM Account
WHERE holder=7564
```

T_1	T_3	A	B	SUM
		75	40	
`v = read(A); v := v-20`				
	`t = read(A); sum := t`			75
`write(A,v)`		55		
`v = read(B); v := v+20`				
`write(B,v)`			60	
`COMMIT`		55	60	
	`t = read(B); sum := sum+t`			135

Serializability

Since each transaction individually cannot cause the database to become inconsistent, a concurrent schedule of transactions $T_1, \ldots, T_k$ that does not cause inconsistencies must be equivalent to a serial schedule of the same transactions[6].

Such a schedule is called **serializable**.

Testing serializability of schedule S

- Draw a graph where every node is a transaction in S.
- Add an edge from transaction T_i to T_j if:
  - T_i and T_j access the same element;
  - at least one of them writes that element in common;
  - T_i performs its operation before T_j.
- If the graph has a cycle, S **is not serializable**.

[6] As we will see soon, this is true only if no transaction aborts.

Examples

Serializable schedule with funds transfer (T_1) before interest calculation (T_2).

T_1	T_2
`v = read(A); v := v-20`	
`write(A,v)`	
`v = read(B); v := v+20`	
`write(B,v)`	
COMMIT	
	`t = read(B); t := t*1.1`
	`write(B,t)`
	COMMIT

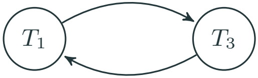
$$
\begin{tikzcd}
T_1 \arrow[r] & T_2
\end{tikzcd}
$$

Serializable schedule with funds transfer (T_1) concurrently with query (T_3).

T_1	T_3
	`t = read(A); sum := t`
`v = read(A); v := v-20`	
`write(A,v)`	
	`t = read(B); sum := sum+t`
`v = read(B); v := v+20`	
`write(B,v)`	
`COMMIT`	

Non-serializable schedule (inconsistent analysis):

T_1	T_3
`v = read(A); v := v-20`	
	`t = read(A); sum := t`
`write(A,v)`	
`v = read(B); v := v+20`	
`write(B,v)`	
COMMIT	
	`t = read(B); sum := sum+t`



Serializability is **necessary** but not sufficient to avoid database inconsistency. If a transaction aborts, even in a serializable schedule, we can arrive in an inconsistent database.

For example, the "dirty read" sechedule from before is serializable:

T_1	T_2
`v = read(A); v := v-20`	
`write(A,v)`	
	`t = read(B); t :`
	`write(B,t)`
`v = read(B); v := v+20`	
	ABORT
`write(B,v)`	
COMMIT	

Moral of the story… so far

To avoid inconsistencies in a concurrent schedule we need to:

(1) Ensure the schedule is serialiazable; and

(2) Prevent dirty reads.

Next we will look into concurrency control mechanisms that achieve both goals.

Later we will see other kinds of concurrency problems that may also arise, and how to fix them.

Concurrency Control strategies

Transaction processing is an important aspect of database applications and different strategies exist, with trade-offs depending on the kind of application:

(1) On Line Transaction Processing (**OLTP**), such as commerce and financial applications (e.g., credit card transactions) deal with many concurrent transactions and often require dedicated hardware and design strategies specific for transaction processing.

(2) On Line Analytical Processing (**OLAP**) applications use data science on archival databases (e.g., monthly data) and don’t require as much.

In general, the concurrency control strategies in use by popular DBMSs fall into these categories:

Pessimistic: assume the worst and use mechanisms that prevent undesirable interactions.

- Pros: no problems due to concurrency.
- Cons: computationally expensive.

Optimistic: assume there will be very few race conditions; monitor the execution of the transactions and intervene only when needed.

- Pros: less overhead compared to preventive methods.
- Cons: some transactions might be delayed or restarted

Preventive Concurrency Control with Locks

The transaction scheduler uses locks[7] to prevent undesirable access to the database elements.

Transactions can only read or write a database element if they are granted an appropriate lock from by the DBMS.

Kinds of locks:

- A **shared** lock can be granted to many transactions, which can only read the element.
- An **exclusive** lock can be granted to a single transaction only, which can read and write the element.

Transactions release the locks once they are no longer needed.

[7] https://en.wikipedia.org/wiki/Record_locking

Locking in database applications follow the **Two Phase Locking** (2PL) protocol:

- **Phase 1:** the transaction can acquire as many locks as needed.
- **Phase 2:** once the transaction releases a lock, it can no longer acquire new ones.

The **Strict 2PL** protocol adds one more restriction, which requires the transaction *to hold on to all locks until it is ready to commit* or when it aborts.

Here's how Strict 2PL handles the "inconsistent analysis" problem:

T_1	T_3	A	B	SUM
		75	40	
	`sh_lock(A);`			
	`t = read(A); sum := t`			75
`xl_lock(A)`				
wait	`sh_lock(B)`			
wait	`t = read(B); sum := sum+t`			115
wait	`unlock(A); unlock(B)`			
`v = read(A); v := v-20`				
`write(A,v)`		55		
`xl_lock(B)`				
`v = read(B); v := v+20`				
`write(B,v)`			60	
`unlock(A); unlock(B)`				
COMMIT		55	60	

T_1 and T_2 can no longer access the elements concurrently, since T_1 needs exclusive access.

Here’s how Strict 2PL handles the “lost update” problem:

T_1	T_2	A	B
		75	40
`xl_lock(A)`			
`v = read(A); v := v-20`			
`write(A,v)`		55	
`xl_lock(B)`			
`v = read(B); v := v+20`			
	`xl_lock(B)`		
`write(B,V)`	wait		60
`unlock(A); unlock(B)`	wait		
COMMIT		55	60
	`t = read(B); t := t*1.1`		
	`write(B,t)`		66
	`unlock(B)`		
	COMMIT		

Here's how Strict 2PL avoids the dirty read:

T_1	T_2	A	B
		75	40
`xl_lock(A);`			
`v = read(A); v := v-20`			
`write(A,v)`		55	
	`xl_lock(B)`		
	`t = read(B); t := t*1.1`		
	`write(B,t)`		44
`xl_lock(B)`			
wait	⋮		
wait	ABORT		40
`v = read(B); v := v+20`			
`write(B,v)`			60
`unlock(A); unlock(B)`			
COMMIT			
		55	60

Deadlocks

When two or more transactions get stuck waiting for one another; each holding an element another needs.

T_1	T_4
`xl_lock(A);`	
`v = read(A); v := v-20`	
`write(A,v)`	
	`xl_lock(B)`
	`t = read(B); t := t-10`
	`write(B,t)`
`xl_lock(B)`	
wait	`xl_lock(A)`
wait	wait
⋮	⋮

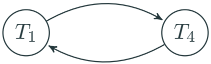
No transaction involved in a deadlock can continue.

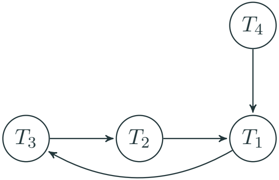
Detecting deadlocks with the **"waits-for" graph** of a set of running transactions:

- Nodes are transactions.
- Add an edge $T_i \to T_j$ if T_i is locked waiting for an element that T_j holds.

Theorem: a set of transactions is in deadlock if (and only if) their waits-for graph contains a cycle.

T_1	T_4
`xl_lock(A);`	
`v = read(A); v := v-20`	
`write(A,v)`	
	`xl_lock(B)`
	`t = read(B); t := t-10`
	`write(B,t)`
`xl_lock(B)`	
wait	`xl_lock(A)`
wait	wait
⋮	⋮



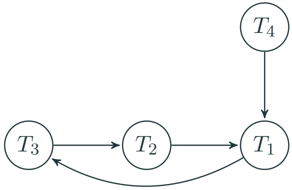


Note that a transaction can be in deadlock even if it is **not** part of the cycle.

In the example above, T_4 is also deadlocked because it cannot proceed until T_1 releases the locks on the elements it needs.

Breaking a deadlock

Once a deadlock is detected the DBMS must pick one transaction in the cycle to **rollback**, allowing the others to continue (hopefully to completion).



But which one?

- The one holding up the most other transactions?
- The one **closest to completion**?
- The one **most recently started**?
- The one from the user with **lowest priority**?

It is up to the **database administrator** to implement whichever policy the organization decides to adopt.

Practical issues with locking

Deadlocks can be avoided if the DBMS can impose a strict ordering on the database elements. But this is hard to do in practice, especially with transactions that insert new elements.

Maintaining the "waits-for" graph can be costly (e.g., when there are thousands of concurrent transactions running).

- Some systems monitor the CPU usage of the transactions; if a transactions stops using the CPU, assume it is deadlocked.

An exclusive lock on a database element implies locks on *all indexes* defined over that element.

- Locking just some nodes of the B+-tree is possible, but tricky to implement.

Optimistic Concurrency Control with timestamps

The approach with optimistic concurrency control is to **monitor** the transactions as they execute, intervening only when a problem has occurred or might occur.

The monitoring is based on when the transactions are accepted by the system.

- Each transaction gets a unique and immutable timestamp TS(T_i) based on the time it started.

Serializable schedules and timestamps

Let $T_1, \ldots, T_n$ be a schedule where TS(T_1) $<$ $\cdots$ $<$ TS(T_n).

Note that the schedule will be *seriazable* if whenever transactions T_i and T_j perform conflicting operations on the same element (i.e., at least one of them writes the element), if $i < j$ then T_i performs its operation **before** T_j.

Data structures needed for monitoring

Recall that each transaction T_i gets a *unique* and immutable timestamp `TS`(T_i), e.g., corresponding to its start time.

In order to know what transactions have been doing, the transaction monitor records the following information, for every element in use by an active transactions:

- `RT(X)`: timestamp of the transaction that most recently read `X`;
- `WT(X)`: timestamp of the transaction that most recently wrote `X`;
- `C(X)`: bit indicating if the most recently written value of `X` has been committed to disk.

These data structures allow the transaction monitor to detect non-serializable schedules!

Transactions send the following requests to the transaction monitorblue:

- `read(X)` / `write(X)`:
- `commit()` / `abort()`

The transaction monitor responds with one of the following:

- **grant** the request: the operation proceeds;
- **deny** the request: the transaction is rolled back (restarted, from scratch, with a new timestamp);
- **delay** the request: the transaction is put on hold until the scheduler can decide if it is safe to grant the request.

Note that a request is denied only when the transaction monitor knows that granting the request would cause problems (e.g., break serializability).

Timestamp-based scheduling

Algorithm 3 Handling `read(X)` request by transaction T_i

- **if** TS(T_i) $<$ WT(X) **then** *% read too late!*
  - deny request; rollback T_i
- **else**
  - **if** C(X) is true **or** TS(T_i)=WT(x) **then**
    - grant request;
    - **if** TS(T_i) $>$ RT(X) **then**
      - RT(X) = TS(T_i)
  - **else**
    - delay request; put T_i on hold

Algorithm 4 Handling `write(X)` request by transaction T_i

- **if** TS(T_i) $<$ RT(X) **then** *% write too late!*
  - deny request; rollback T_i
- **else**
  - **if** TS(T_i) $\geq$ WT(X) **then**
    - grant request; write X
    - set WT(X) = TS(T_i) and C(X) = false
  - **else**
    - **if** C(X) is true **then**
      - do nothing; let T_i continue
    - **else**
      - delay request; put T_i on hold

Algorithm 5 Handling `commit()` request by transaction T_i

1: **for each** element X written by T_i **do**
2: set C(X) = true
3: **for each** transaction T_j waiting to read/write X **do**
4: grant T_j's request; let T_j continue

Algorithm 6 Handling `abort()` request by transaction T_i

1: **for each** element X written by T_i **do**
2: **for each** transaction T_j waiting to read/write X **do**
3: re-evaluate T_j's request with the current timestamps

Here's how timestamping avoids the "lost update" problem:

T_1 — TS(T_1) = 100	T_2 — TS(T_2) = 150	A	B	
		75	40	
`v = read(A); v := v-20`				RT(A)=100
`write(A,v)`		55		WT(A) = 100; C(A)=false
`v = read(B); v := v+20`				RT(B)=100
	`t = read(B); t := t*1.1`			RT(B)=150
	`write(B,t)`		44	WT(B)=150; C(B)=false
	COMMIT		44	C(B)=true
`write(B,V)`				DENY
rollback		75		

Because TS(T_1) $<$ TS(T_2), once T_2 writes B, T_1 can no longer overwrite it.

Also, note that the new value of A written by T_1 is not yet committed, so that change is not made permanent.

Recall that with Strict 2PL, during a race condition, the transaction that acquires the lock first is allowed to continue (slide 59).

T_1 — TS(T_1) = 100	T_2 — TS(T_2) = 150	A	B	
		75	40	
`v = read(A); v := v-20`				RT(A)=100
`write(A,v)`		55		WT(A) = 100; C(A)=false
`v = read(B); v := v+20`				RT(B)=100
	`t = read(B); t := t*1.1`			RT(B)=150
	`write(B,t)`		44	WT(B)=150; C(B)=false
	COMMIT		44	C(B)=true
`write(B,V)`				DENY
rollback		75		

With timestamping, on the other hand, every request is evaluated individually: Even though T_1 was granted the read on B before T_2 started, that does not mean T_1 has priority.

For this reason, the request by T_1 is often called a “write too late” or “write out of order”.

But what if T_2 had not yet committed?

T_1 — TS(T_1) = 100	T_2 — TS(T_2) = 150	A	B	
		75	40	
`v = read(A); v := v-20`				RT(A)=100
`write(A,v)`		55		WT(A) = 100; C(A)=false
`v = read(B); v := v+20`				RT(B)=100
	`t = read(B); t := t*1.1`			RT(B)=150
	`write(B,t)`		44	WT(B)=150; C(B)=false
`write(B,V)`				DELAY
⋮				

In this case, the transaction monitor does not know yet if the write requested by T_1 is valid or not: If T_2 later on aborts, T_1 should be allowed to write its value. If T_2 commits, T_1 is rolled back.

Here's how timestamping avoids the "inconsistent analysis" problem:

T_1 — TS(T_1) = 100	T_3 — TS(T_3) = 200	A	B	SUM	
		75	40		
`v = read(A);`					RT(A)=100
`v := v-20`					
	`t = read(A);`				RT(A)=200
	`sum := t`			75	
`write(A,v)`					DENY
rollback					
	`t = read(B);`				RT(B)=200
	`sum := sum+t`			115	

Again, even though T_1 started before T_2 (note their timestamps), that does not mean it has priority.

Transactions are allowed to run until they make an invalid request.

Here's how timestamping avoids the "dirty read" problem:

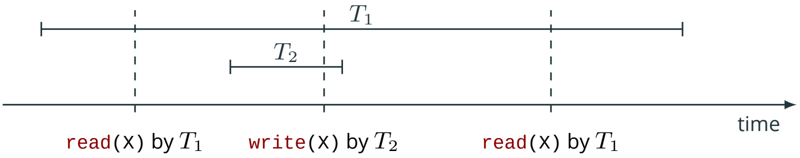
T_1 — TS(T_1) = 100	T_2 — TS(T_2) = 150	A	B	
		75	40	
`v = read(A); v := v-20`				RT(A)=100
`write(A,v)`		55		WT(A)=100; C(A)=false
	`t = read(B); t := t*1.1`			RT(B)=150
	`write(B,t)`		44	WT(B)=150; C(B)=false
`v = read(B)`				DELAY
	ABORT		40	
`v = read(B); v := v+20`				RT(B)=100;
`write(B,v)`			60	WT(B)=100; C(B)=false
COMMIT				
		55	60	C(A)=true; C(B)=true

When T_1 attempts to read B (again, too late), the transaction manager puts it on hold until the fate of T_2 is decided.

When T_2 aborts, T_1 is allowed to continue, reading the old value for that element.

Long-running transactions and snapshot isolation

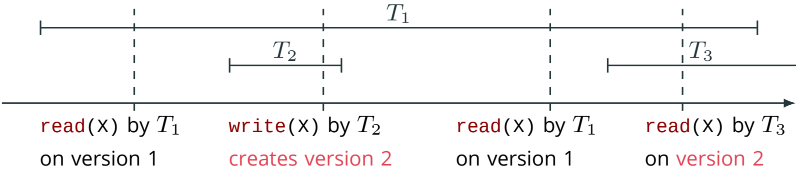
Consider transactions T_1 and T_2 that both need to write to a database element x, and the following timeline corresponding to running them **without** any concurrency control:



If we use locking, T_2 must wait a long time (as T_1 started first).

If we use time-stamping, T_1 will be rolled back, after computing for a long time.

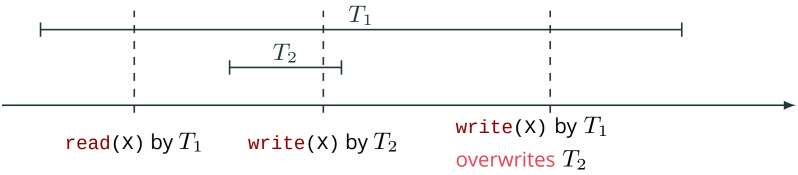
Instead of overwriting the element, a write operation creates a **new version**, used by transactions with that timestamp (or higher).



In this method, each transaction runs on a “snapshot” of the database, consistent with the time the transaction started[8].

[8] https://en.wikipedia.org/wiki/Snapshot_isolation

But snapshot isolation re-introduces the “lost update” problem:



If T_1 **writes** element x that will overwrite the update by T_2.

This can lead to inconsistencies, as other values written by T_2 could have been calculated based on the value of x in its snapshot.

Different DBMSs handle this problem differently. Often, the solution is to let the DBA or the application decide how to handle the problem.

Practical Considerations

Locking? Timestamping?

There is no **single** model that is best for all applications.

The choice depends on the **mix of queries and updates** in the workload.

> The only low hanging fruit here: if the workload is 100% of queries, there is no need for concurrency control!

Many systems use a **combination** of locking and timestamping:

- 2PL with shared locks to execute updates
- timestamping with multiple versions for queries

Overhead of ensuring Isolation

$\boxed{E\text{: number of database elements; } T\text{: number of transactions}}$

Space cost of implementing isolation.

Locking: $O(E \cdot T)$

- lock table grows linearly with the number of locked elements

Timestamping: $O(E) + O(T)$

- read/write times for each element and timestamps for each transaction

Timestamping + versioning: $O(E \cdot T)$

- every transaction may create a new version of every element

In all concurrency control models, some steps must be done **atomically** by the scheduler:

- updating the lock table
- updating the timestamp table
- creating/deleting versions

These operations require **synchronization** and cannot be done in parallel to avoid race conditions leading to inconsistency, which can limit the system throughput.

Isolation Levels in SQL

SQL allows the programmer **to chose the level of isolation** necessary for each transaction in the application.

In SQL:1999 these isolation levels are possible:

- `READ UNCOMMITTED`: the transaction is allowed to perform dirty reads.
- `READ COMMITTED`: the transaction can only read elements that have been committed.
- `REPEATABLE READ`: database elements read by the transaction cannot change after it starts.
- `SERIALIZABLE`: the transaction runs in total isolation from other transactions.

SQL isolation levels and the problems they prevent

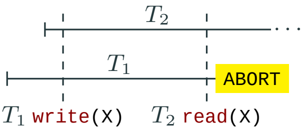
Level	dirty reads	non-repeatable reads	phantoms
READ UNCOMMITTED	allows	allows	allows
READ COMMITTED	prevents	allows	allows
REPEATABLE READ	prevents	prevents	allows
SERIALIZABLE	prevents	prevents	prevents

Preventing Dirty Reads

As we saw, both locking and timestamp-based concurrency control prevent dirty reads.

An even simpler (and faster) solution would be to use just the **commit bits** `commit()` of the timetamp concurrency control approach.

Aside: are **dirty reads** a real problem?



DEFINITELY YES for applications like banking:

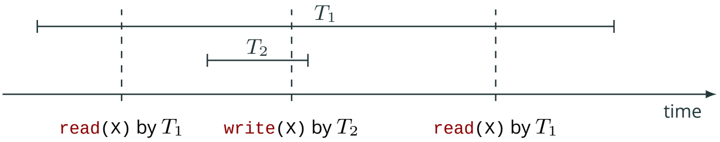
- T_1 is a deposit; T_2 is a withdrawal from the same account.
- The dirty read may authorize the ATM to dispense money that the customer does not have

MAYBE? for OLAP workloads ("analytics", business intelligence):

- T_1 is a sale of *items*; T_2 generates a monthly sales report
- The dirty read will overestimate the monthly sales of those items, which might not be a problem

Non-Repeatable Read

A **Non-Repeatable Read** (NRR) happens when two reads of the same database element by the same transaction return different values.



As we saw, all methods can detect and prevent NRRs.

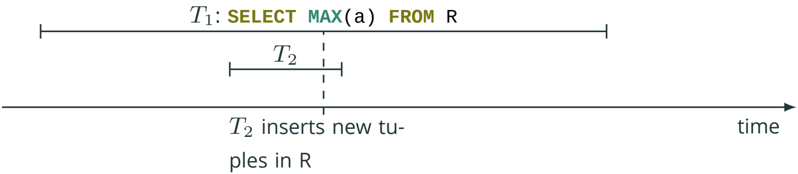
Although snapshot isolation (MVCC) prevents NRRs at the expense of potential inconsistencies.

Phantom tuples

Happen when a transaction T_1 reads from a relation that another transaction T_2 is inserting new tuples to.

Typical example

T_1 computes an aggregate over a column of relation R while T_2, at the same time, inserts new tuples in R.



Locking and timestamp based concurrency control work at the level of database elements (e.g., tuples or attributes of tuples). However...

- T_1 cannot lock the tuples inserted by T_2 because it does not even know they exist!
- There are no read/write violations involving them either.

Solution: T_2 must lock the <u>entire relation</u> before it can go through with the insertion.