3. Real-World Example of Pessimistic Concurrency Control
4. Pros and Cons of Pessimistic Concurrency Control
5. How it Guarantees the Read Committed Isolation Level
6. Optimistic Concurrency Control
7. Real-World Example of Optimistic Concurrency Control
8. Pros and Cons of Optimistic Concurrency Control
9. How it Guarantees the Repeatable Read Isolation Level
10. Bringing it Together

Pessimistic Concurrency Control

pessimistic – that is, it assumes that if something can go wrong, it will go wrong. This pessimism prevents conflicts from occurring by blocking them before they get a chance to start.

To prevent these conflicts, it *locks* the data that a transaction is using until the transaction is completed. This approach is 'pessimistic' because it assumes the worst-case scenario – that every transaction might lead to a *conflict*. The data is therefore locked in order to prevent conflicts from happening.

I've mentioned two technical terms here that need clarification: *locks* and *conflict*.

What are locks?

A lock is a mechanism used to control access to a database item, like a row or table. Locks ensure data integrity, if

In very simple terms, a lock is analogous to a reservation on the database item. A reservation, be it a restaurant, hotel, or a train, prevents other people from using the resource you reserved for a fixed duration of time. Locks work in a similar way.

There are two types of locks: a read lock and a write lock.

A read lock can be shared by multiple transactions trying to read the same database item. But it blocks other transactions from updating that database item.

A write lock is exclusive – that is, it can only be held by a single transaction. A transaction with a write lock on a database item blocks every other transaction from reading or updating that database item.

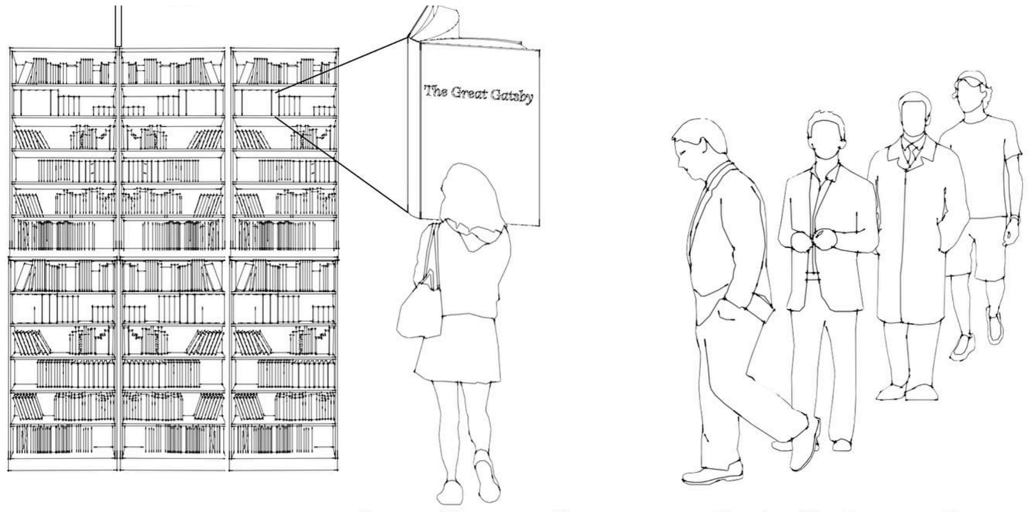
What are conflicts?

concurrently, in a way that could lead to inconsistencies or errors in the database.

A Library Analogy for Pessimistic Concurrency Control

First, let us describe an analogy for a write lock.

Imagine you're at a library, and you want to borrow a hard copy of a popular book, say, The Great Gatsby by F. Scott Fitzgerald.



Write locks are analogous to borrowing a physical book from the library

With a write lock, the librarian assumes that there can be conflicts over who gets to borrow the book. So, they implement a strict rule to avoid conflicts: only one person can hold the reservation for a physical book at a time.

returned. This is similar to how a write lock works.

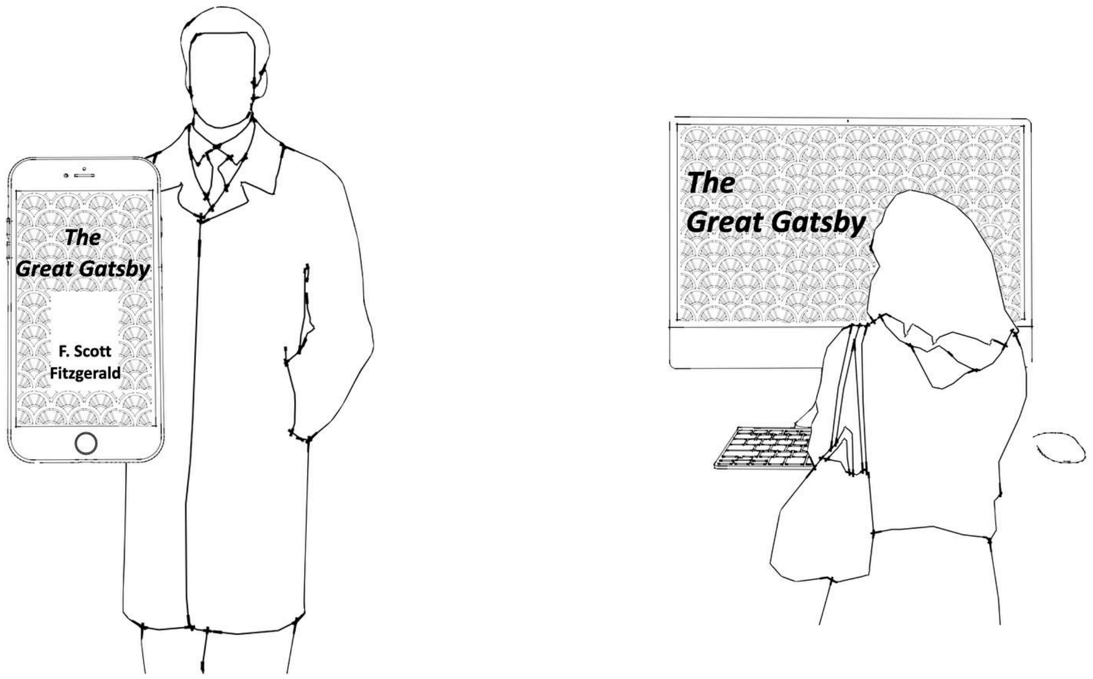
Write locks are exclusive. This means that they can only he held by a single transaction at any time. Similarly, reserving a physical book from the library means no one else has access to it. Only the person with the reservation can read the book, or write in it (although writing in a library book is bad form).

Read locks work a bit differently.

A read lock is analogous to someone making a reservation to borrow an e-book. Borrowing an e-book is not a very popular thing to do, but some libraries do have such a service.

Many people can make the same reservation for the same e-book without any conflict. One person borrowing an e-book version of The Great Gatsby does not stop others

others, for example.

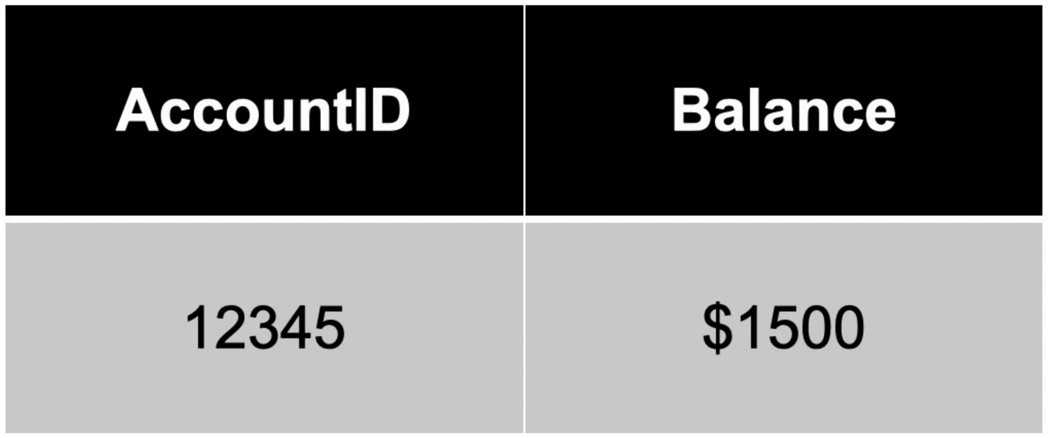


Read locks are analogous to borrowing an e-book from the library they get a chance to start. A write lock on a database item prevents other transactions from reading or updating that item while that lock is held, similar to to how a library stops more than one person from trying to borrow the same physical book at the same time.

A read lock on a database item allows other transactions to also obtain a read lock for that item, but prevents transactions from updating that item. This is analogous to borrowing an e-book, where multiple people can borrow the same e-book at the same time, but can't make any updates to it.

A Simple Real-World Example of Pessimistic Concurrency Control in Action

Let's illustrate how pessimistic concurrency control works using a simple example involving a bank balance database



AccountID	Balance
12345	$1500

Database columns for AccountID and Balance

Two transactions, T1 and T2, intend to update the balance of account 12345. T1 wants to withdraw $300, and T2 wants to deposit $400. At the end of these two transactions, the account balance should read $1600

1. Start of T1 (Withdrawal): T1 requests to update the balance of AccountID 12345. The database system places an exclusive write lock on the row for AccountID 12345, preventing other transactions from reading or writing to this row until T1 is completed. T1 reads the balance ($1500).
2. T1 Processing: T1 calculates the new balance as $1200 ($1500 - $300).
3. Commit T1: T1 writes the new balance ($1200) back to the database. Upon successful commit, T1 releases the exclusive lock on AccountID 12345.
4. Start of T2 (Deposit) After T1 Completes: Now that T1 has completed and the lock is released, T2 can start. T2 attempts to read and update the balance for AccountID 12345. The database system places an exclusive lock on the row for AccountID 12345 for T2, ensuring no other transactions can interfere. T2 reads the updated balance ($1200).

6. Commit T2: T2 writes the new balance ($1600) back to the database. Upon successful commit, T2 releases the exclusive lock on AccountID 12345.
7. Result: The Accounts table is updated using locks
   After T1: $1200 After T2: $1600

Without a write lock in this example, T1 and T2 could read the original balance of $1500 at the same time. So, instead of a balance of $1200 after T1 has committed, T2 still reads the original balance of $1500 and adds $400. This would cause the final balance to be $1500 + $400 = $1900 (instead of $1600).

Absence of locking has created free money, which is never a bad thing for a customer. But, if money can be conjured out of thin air because of these conflicts, it can also vanish, and accidentally shrinking bank balances are a quick way to make customers unhappy.

Concurrency Control

Just like reserving a book ensures that it's set aside for one person, pessimistic concurrency control locks data for a single transaction. Other transactions cannot access or modify this data until the lock is released.

This method prevents two people from trying to take out the same popular book at the same time, thereby avoiding disputes. Similarly, in databases, it stops conflicts due to concurrent transactions before they get a chance to start.

But this approach can be inefficient. The reserved book might sit on the reserved shelf for a while, stopping other people from reading it.

In databases, this locking mechanism can lead to underutilisation of resources and a slowdown in the speed transactions take to complete, since a subset of the data is locked and inaccessible to other transactions.

Guarantee the Read Committed Isolation Level

So, how exactly does pessimistic concurrency control work in ensuring the isolation guarantee, that is the “I” in ACID? The implementation details can vary across different DBMS. But the explanation here shows the general approach.

Recall that the read committed isolation level prevents dirty writes and dirty reads.

Preventing Dirty Writes

Overwriting data that has already been written by another transaction but not yet committed is called a dirty write. A common approach to preventing dirty writes is to use pessimistic concurrency control. For example, by using a write lock at the row level.

complete. Recall that write locks can only be held by a single transaction. This prevents another transaction from acquiring a lock to modify that row.

Preventing Dirty Reads

Reading data from another transaction that has not yet been committed is called a dirty read. Dirty reads are prevented using either a read or write lock. Once a transaction acquires a read lock on a database item, it will prevent updates to that item.

But what happens if you are trying to read something that is already being updated but the transaction has not yet committed? In this instance, the write lock saves the day again.

Since write locks are exclusive (can’t be shared with other transactions), any transaction wanting to read the same prevents other transactions from reading uncommitted changes.

Optimistic Concurrency Control

With optimistic concurrency control, transactions do not obtain locks on data when they read or write. The "Optimistic" in the name comes from assuming that conflicts are unlikely to occur, so locks are not needed. If something does go wrong though, conflicts will still be prevented and everything will be OK.

Unlike pessimistic concurrency control – which prevents conflicts from occurring by blocking them before they get a chance to start – optimistic concurrency control checks for conflicts at the end of a transaction.

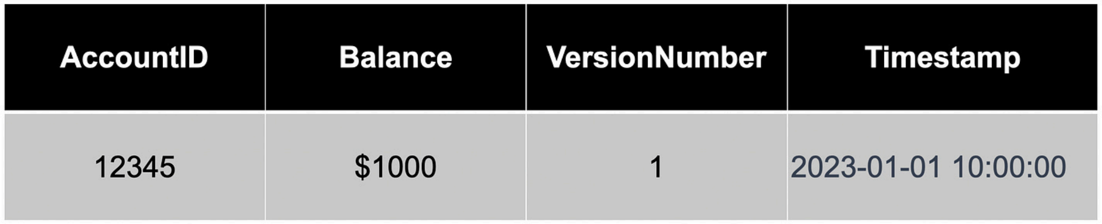
With optimistic concurrency control, multiple transactions can read or update the same database item without

Every time a transaction wants to update a database item, say a row, it will also read two additional columns added to every table by the DBMS – the timestamp and the version number. Before that transaction is committed, it checks if another transaction has made any change(s) to that row by confirming if the version number and timestamp are the same.

If they have changed, that means another transaction has updated that row, so the initial transaction will have to be retried.

A Simple Real-World Example of Optimistic Concurrency Control in Action

Let's illustrate how optimistic concurrency control works using a simple example involving a bank balance database table. Assume we have a table named Accounts with the



AccountID	Balance	VersionNumber	Timestamp
12345	$1000	1	2023-01-01 10:00:00

Table showing AccountID, Balance, VersionNumber, and Timestamp columns

Two transactions, T1 and T2, intend to update the balance of account 12345 at the same time. T1 wants to withdraw $200, and T2 wants to deposit $300. At the end of these two transactions, the account balance should read $1100

Here are the steps of how this will work:

Simultaneously, T2 reads the same row with the same balance, version number, and timestamp.

2. Processing: T1 calculates the new balance as $800 ($1000 - $200) but does not write it back immediately. T2 calculates the new balance as $1300 ($1000 + $300) but also waits to commit.

3. Attempt to Commit T1: Before committing, T1 checks the current VersionNumber and Timestamp of AccountID 12345 in the database. Since no other transaction has modified the row, T1 updates the balance to $800, increments the VersionNumber to 2, updates the Timestamp, and commits successfully.

4. Attempt to Commit T2: T2 attempts to commit by first verifying the VersionNumber and Timestamp. T2 finds that the VersionNumber and Timestamp have changed (now VersionNumber is 2, and Timestamp is updated), indicating another

realises there was a conflict.

5. Resolution for T2: T2 must restart its transaction. It re-reads the updated balance of $800, the new VersionNumber 2, and the updated Timestamp. T2 recalculates the new balance as $1100 ($800 + $300), updates the VersionNumber to 3, updates the Timestamp, and commits successfully.

Result: The Accounts table is updated sequentially and safely without any locks: After T1: $800, VersionNumber: 2. After T2: $1100, VersionNumber: 3.

Benefits and Challenges of Optimistic Concurrency Control

On the positive side, avoiding locks allows for high levels of concurrency. This is particularly beneficial in read-heavy workloads where transactions are less likely to conflict, allowing the system to handle more transactions in a given

But in scenarios where conflicts are frequent, the cost of repeatedly rolling back and retrying transactions can outweigh the benefits of avoiding locks, making optimistic concurrency control less efficient

How Optimistic Concurrency Controls Guarantee the Repeatable Read Isolation level

The repeatable read is more strict isolation level in that it has the same guarantees as read committed isolation, plus it guarantees that reads are repeatable.

A repeatable read guarantees that if a transaction reads a row of data, any subsequent reads of that same row of data within the same transaction will yield the same result, regardless of changes made by other transactions. This

How can a repeatable read be achieved? Pessimistic control using a read lock can help with this, since a transaction with a read lock on a database item will prevent that item from being updated. But this can be inefficient, since a long running read transaction can block updates from happening to that database item.

Multi-Version Concurrency Control (MVCC) is a concurrency control method used by some DBMS to allow multiple transactions to access the same data simultaneously without locking the data. This makes it a popular choice for reducing lock contention and improving the scalability of databases.

MVCC achieves this by keeping multiple versions of data objects, which helps to manage different visibility levels for transactions depending on their timestamps or version numbers.

A lock is a mechanism used to control access to a database item, like a row or table. In very simple terms, it is analogous to a reservation on a database item.

Pessimistic concurrency control assumes the worst. It assumes that conflicts are likely to happen, so locks are used to block transactions that can cause conflicts before they even get a chance to start.

In situations where conflicts are common, such as a write heavy application, this approach can prevent the overhead associated with frequent rollbacks and retries (which happens in optimistic concurrency control) by ensuring exclusive access to database items during transactions.

Optimistic concurrency control assumes the best. It assumes that conflicts are unlikely to occur, so locks are not needed to stop transactions before they start. Instead,