Optimistic vs. Pessimistic locking

Asked 16 years, 3 months ago Modified 2 months ago

Viewed 678k times

1035

I understand the differences between optimistic and pessimistic locking. Now, could someone explain to me when I would use either one in general?

And does the answer to this question change depending on whether or not I'm using a stored procedure to perform the query?

But just to check, optimistic means "don't lock the table while reading" and pessimistic means "lock the table while reading."

database transactions locking optimistic-locking pessimistic-locking

Share

Improve this question

Follow

edited Oct 14, 2022 at 15:54

Super Kai - Kazuya Ito

1

asked Sep 24, 2008 at 19:29

Jason Baker

198k ● 138 ● 382 ● 520

3 blog.couchbase.com/… – Frank Myat Thu Jan 13, 2016 at 10:15

3 That is a good question particularly because in serializability I read `At any technique type conflicts should be detected and considered, with similar overhead for both materialized and non-materialized conflicts`. – Little Alien Dec 5, 2016 at 15:08

4 Here you can find a good explanation, here on SO, about what is the root concept of Optimistic Locking. – Diego Mazzaro Dec 21, 2019 at 9:38

2 I would recommend to read Martin Fowler's great book on patterns: martinfowler.com/books/eaa.html – koppor Oct 11, 2020 at 10:54

I think concurrency control is more accurate than locking. – Jason Law Oct 19, 2021 at 9:58

14 Answers

Sorted by: Highest score (default)

1344

Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

In the latter case you open the transaction with the TxID and then reconnect using that ID. The DBMS maintains the locks and allows you to pick the session back up through the TxID. This is how distributed transactions using two-phase commit protocols (such as XA or COM+ Transactions) work.

Share Improve this answer Follow

edited Feb 28, 2016 at 11:57

nc.
7,249 ● 6 ● 30 ● 38

answered Sep 24, 2008 at 19:40

ConcernedOfTunbridgeWells
66.5k ● 15 ● 148 ● 198

205 Optimistic locking doesn't necessarily use a version number. Other strategies include using (a) a timestamp or (b) the entire state of the row itself. The latter strategy is ugly but avoids the need for a dedicated version column, in cases where you aren't able to modify the schema. – Andrew Swan Dec 8, 2009 at 22:33

3 The concept of optimistic locking doesn't necessarily require having a 100% reliable way of knowing whether or not something has been altered; undetectable alterations aren't acceptable, but occasional false reports of alteration may not be too bad, especially if code which receives such a report rereads the data and checks whether it has actually changed. – supercat Jul 8, 2013 at 21:43

50 @supercat - Don't agree that optimistic locking is less than 100% accurate - as long as it checks all the input records for transaction that should stay unmodified for the duration, it's as accurate as pessimistic locking (select for update style) on those same records. The main difference is that optimistic locking incurs overhead only if there's a conflict, whereas pessimistic locking has reduced overhead on conflict. So optimistic is best in case where most transactions don't conflict - which I hope is usually the case for most apps. – RichVel Aug 5, 2014 at 11:33

6 @Legends - Using optimsitic locking would certainly be an appropriate strategy for a web application. – ConcernedOfTunbridgeWells Apr 2, 2015 at 7:19

4 You should mention that the choice depend also on the ratio read vs. write: if your application is mainly a read-only application by a lot of users, and sometimes you write data, than go for optimistic locking. StackOverflow, for example,

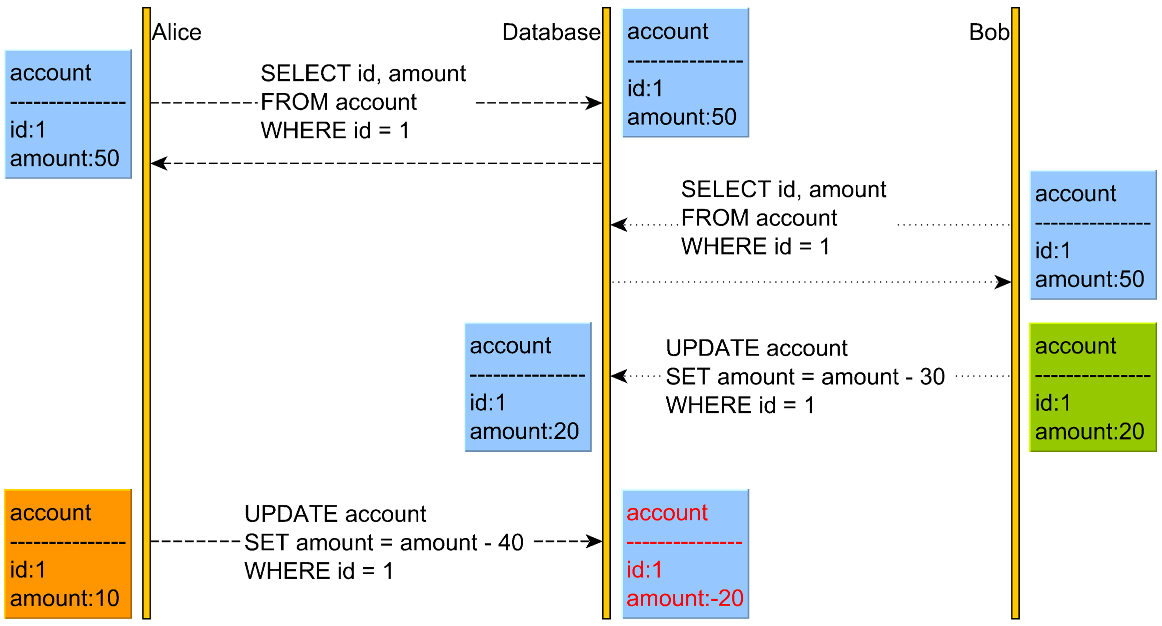
have a lot of people reading pages, and sometimes someone edit one: in pessimistic locking, who would get the lock? the first one? In optimistic locking, the person who wish to edit the page can do it as long as he have the last version of it. – jehon May 5, 2015 at 8:28

467

When dealing with conflicts, you have two options:

- You can try to avoid the conflict, and that's what Pessimistic Locking does.
- Or, you could allow the conflict to occur, but you need to detect it upon committing your transactions, and that's what Optimistic Locking does.

Now, let's consider the following Lost Update anomaly:

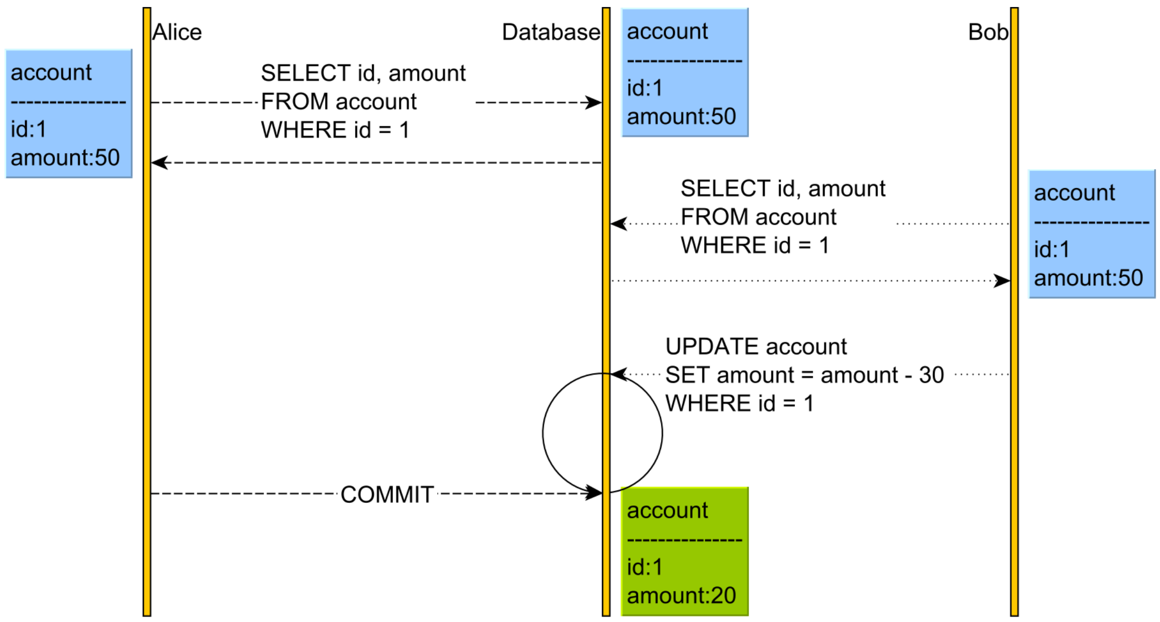


The Lost Update anomaly can happen in the Read Committed isolation level.

In the diagram above we can see that Alice believes she can withdraw 40 from her `account` but does not realize that Bob has just changed the account balance, and now there are only 20 left in this account.

Pessimistic Locking

Pessimistic locking achieves this goal by taking a shared or read lock on the account so Bob is prevented from changing the account.



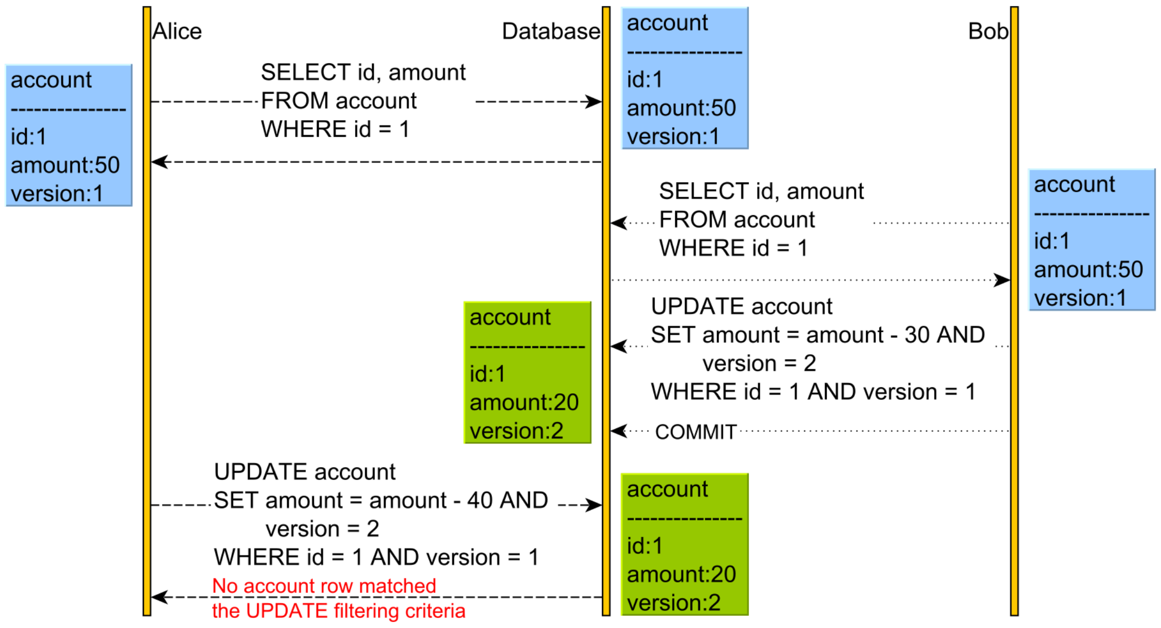
In the diagram above, both Alice and Bob will acquire a read lock on the `account` table row that both users have read. The database acquires these locks on SQL Server when using Repeatable Read or Serializable.

Because both Alice and Bob have read the `account` with the PK value of `1`, neither of them can change it until one user releases the read lock. This is because a write operation requires a write/exclusive lock acquisition, and shared/read locks prevent write/exclusive locks.

Only after Alice has committed her transaction and the read lock was released on the `account` row, Bob `UPDATE` will resume and apply the change. Until Alice releases the read lock, Bob's UPDATE blocks.

Optimistic Locking

Optimistic Locking allows the conflict to occur but detects it upon applying Alice's UPDATE as the version has changed.


Alice
Database
Bob
account
----
id:1
amount:50
version:1
SELECT id, amount FROM account WHERE id = 1
UPDATE account SET amount = amount - 30 AND version = 2 WHERE id = 1 AND version = 1
COMMIT
account
----
id:1
amount:20
version:2
UPDATE account SET amount = amount - 40 AND version = 2 WHERE id = 1 AND version = 1
No account row matched the UPDATE filtering criteria

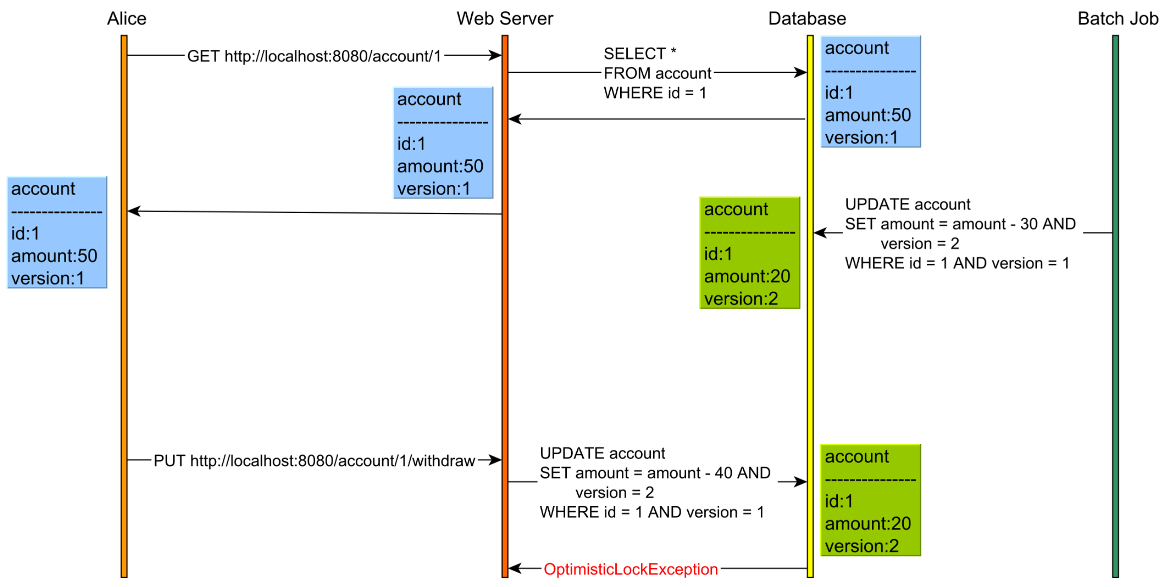
This time, we have an additional `version` column. The `version` column is incremented every time an UPDATE or DELETE is executed, and it is also used in the WHERE clause of the UPDATE and DELETE statements. For this to work, we need to issue the SELECT and read the current `version` prior to executing the UPDATE or DELETE, as otherwise, we would not know what version value to pass to the WHERE clause or to increment.

Application-level transactions

Relational database systems have emerged in the late 70's early 80's when a client would, typically, connect to a mainframe via a terminal. That's why we still see database systems define terms such as SESSION setting.

Nowadays, over the Internet, we no longer execute reads and writes in the context of the same database transaction, and ACID is no longer sufficient.

For instance, consider the following use case:


Alice
Web Server
Database
Batch Job
GET http://localhost:8080/account/1
SELECT * FROM account WHERE id = 1
account --------------- id:1 amount:50 version:1
account --------------- id:1 amount:50 version:1
account --------------- id:1 amount:50 version:1
UPDATE account SET amount = amount - 30 AND version = 2 WHERE id = 1 AND version = 1
account --------------- id:1 amount:20 version:2
PUT http://localhost:8080/account/1/withdraw
UPDATE account SET amount = amount - 40 AND version = 2 WHERE id = 1 AND version = 1
account --------------- id:1 amount:20 version:2
OptimisticLockException

Without optimistic locking, there is no way this Lost Update would have been caught even if the database transactions used Serializable. This is because reads and writes are executed in separate HTTP requests, hence on different database transactions.

So, optimistic locking can help you prevent Lost Updates even when using application-level transactions that incorporate the user-think time as well.

Conclusion

Optimistic locking is a very useful technique, and it works just fine even when using less-strict isolation levels, like Read Committed, or when reads and writes are executed in subsequent database transactions.

The downside of optimistic locking is that a rollback will be triggered by the data access framework upon catching an `OptimisticLockException`, therefore losing all the work we've done previously by the currently executing transaction.

The more contention, the more conflicts, and the greater the chance of aborting transactions. Rollbacks can be costly for the database system as it needs to revert all current pending changes which might involve both table rows and index records.

For this reason, pessimistic locking might be more suitable when conflicts happen frequently, as it reduces the chance of rolling back transactions.

Share Improve this answer Follow

edited Mar 24, 2021 at 10:01

answered Nov 20, 2019 at 10:23

Vlad Mihalcea
153k ● 84 ● 588 ● 975

3 For what scenarios would you suggest to choose OptimisticLocking and PessimisticLocking? Does it depend

on how often a OptimisticLockException occurs?
– Stimpson Cat Jun 3, 2020 at 10:34

3 @StimpsonCat from what I read from his conclusion, yes, if u get exception often then it is better to go pessimistic locking. Like in my case, the chance of exception occur is very small so i will go for Optimistic locking. – Dave Cruise Oct 15, 2020 at 11:58

2 Upvoted. Although the material isn't novel, well-explained answers are becoming a rarity in SO as more and more one-off homework questions flood the system. – Abhijit Sarkar Sep 25, 2021 at 1:46

1 I paused your youtube video to search for sth online, bumped into your answer.. I know this is against the comment guidelines but nevertheless very helpful contribution!
– EralpB Dec 23, 2021 at 8:39

6 @EralpB Google, StackOverflow, YoutTube, GitHub, Twitter, LinkedIn, you'll find me everywhere 😉 – Vlad Mihalcea Dec 23, 2021 at 10:36

278

Optimistic locking is used when you don't expect many collisions. It costs less to do a normal operation but if the collision DOES occur you would pay a higher price to resolve it as the transaction is aborted.

Pessimistic locking is used when a collision is anticipated. The transactions which would violate synchronization are simply blocked.

To select proper locking mechanism you have to estimate the amount of reads and writes and plan accordingly.

Share Improve this answer Follow

edited May 5, 2019 at 11:32
Hearen
7,778 ● 4 ● 57 ● 73

answered Sep 24, 2008 at 19:37
Ilya Kochetov
18.4k ● 6 ● 47 ● 62

In normal case, the statement is perfect but in special cases where you could manage the **CAS** operation allowing inaccuracy as @skaffman mentioned in the answer, I would say that really depends. – Hearen May 5, 2019 at 11:33

108

Optimistic assumes that nothing's going to change while you're reading it.

Pessimistic assumes that something will and so locks it.

If it's not essential that the data is perfectly read use optimistic. You might get the odd 'dirty' read - but it's far less likely to result in deadlocks and the like.

Most web applications are fine with dirty reads - on the rare occasion the data doesn't exactly tally the next reload does.

For exact data operations (like in many financial transactions) use pessimistic. It's essential that the data is accurately read, with no un-shown changes - the extra locking overhead is worth it.

Oh, and Microsoft SQL server defaults to page locking - basically the row you're reading and a few either side. Row locking is more accurate but much slower. It's often worth setting your transactions to read-committed or no-lock to avoid deadlocks while reading.

Share Improve this answer Follow

edited Dec 13, 2016 at 14:06

answered Sep 24, 2008 at 19:34

Keith
155k ● 82 ● 306 ● 446

JPA Optimistic Locking allows you to guarantee read-consistency. – Gili Sep 24, 2008 at 19:43

5 Read-consistency is a separate concern - with PostgreSQL, Oracle and many other databases, you get a consistent view of data regardless of any updates not yet committed, and aren't affected even by exclusive row locks. – RichVel Aug 1, 2014 at 13:52

1 I have to agree with @RichVel. On the one hand, I can see how pessimistic locking could prevent dirty reads if your transaction isolation level is READ UNCOMMITTED. But it is misleading to say that optimistic locking is susceptible to dirty reads without mentioning out that most databases (including apparently MS SQL Server) have a default isolation level of "READ COMMITTED", which prevents dirty reads and makes optimistic locking just as accurate as pessimistic. – antinome Oct 7, 2014 at 20:17

Eric Brower says that bankers, unlike others, prefer dirty operations. Your gurus seem absolutely out of trolleys. – Little Alien Dec 5, 2016 at 15:18

2 Eric Brewer is the guru who gave the the CAP theorem says about consistency in banking. It is opposite to what you honor it for. – Little Alien Dec 5, 2016 at 22:05

30

I would think of one more case when pessimistic locking would be a better choice.

For optimistic locking every participant in data modification must agree in using this kind of locking. But if someone modifies the data without taking care about the version column, this will spoil the whole idea of the optimistic locking.

Share Improve this answer Follow

answered Apr 4, 2013 at 8:55

Nikolay
1,121 ● 2 ● 12 ● 17

1 People attempting to use optimistic and pessimistic locking can also step on each others' feet, so to speak. Imagine a scenario where an optimistic session reads a record and is doing some calculations while a pessimistic session updates the record, then the optimistic session comes back and updates that same record without noting any changes made. Select ... for update only works if every session is using that same syntax. – lusional Aug 4, 2016 at 22:09

30

There are basically two most popular answers. The first one basically says

> Optimistic needs a three-tier architectures where you do not necessarily maintain a connection to the database for your session whereas Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking you need either a direct connection to the database.

Another answer is

> optimistic (versioning) is faster because of no locking but (pessimistic) locking performs better when contention is high and it is better to prevent the work rather than discard it and start over.

or

> Optimistic locking works best when you have rare collisions

As it is put on this page.

I created my answer to explain how "keep connection" is related to "low collisions".

To understand which strategy is best for you, think not about the Transactions Per Second your DB has but the duration of a single transaction. Normally, you open trasnaction, performa operation and close the transaction. This is a short, classical transaction ANSI had in mind and fine to get away with locking. But, how do you implement a ticket reservation system where many clients reserve the same rooms/seats at the same time?

You browse the offers, fill in the form with lots of available options and current prices. It takes a lot of time and options can become obsolete, all the prices invalid between you started to fill the form and press "I agree" button because there was no lock on the data you have accessed and somebody else, more agile, has intefered changing all the prices and you need to restart with new prices.

You could lock all the options as you read them, instead. This is pessimistic scenario. You see why it sucks. Your system can be brought down by a single clown who simply starts a reservation and goes smoking. Nobody can reserve anything before he finishes. Your cash flow drops to zero. That is why, optimistic reservations are used in reality. Those who dawdle too long have to restart their reservation at higher prices.

In this optimistic approach you have to record all the data that you read (as in mine Repeated Read) and come to the commit point with your version of data (I want to buy shares at the price you displayed in this quote, not current price). At this point, ANSI transaction is created, which locks the DB, checks if nothing is changed and commits/aborts your operation. IMO, this is effective emulation of MVCC, which is also associated with Optimistic CC and also assumes that your transaction restarts in case of abort, that is you will make a new reservation. A transaction here involves a human user decisions.

I am far from understanding how to implement the MVCC manually but I think that long-running transactions with option of restart is the key to understanding the subject. Correct me if I am wrong anywhere. My answer was motivated by this Alex Kuznecov chapter.

Share Improve this answer Follow

edited May 23, 2017 at 11:33
Community Bot
1 ● 1

answered Dec 8, 2016 at 0:24
Little Alien
1

21

In most cases, optimistic locking is more efficient and offers higher performance. When choosing between pessimistic and optimistic locking, consider the following:

- Pessimistic locking is useful if there are a lot of updates and relatively high chances of users trying to update data at the same time. For example, if each operation can update a large number of records at a time (the bank might add interest earnings to every account at the end of each month), and two applications are running such operations at the same time, they will have conflicts.

- Pessimistic locking is also more appropriate in applications that contain small tables that are frequently updated. In the case of these so-called hotspots, conflicts are so probable that optimistic locking wastes effort in rolling back conflicting transactions.
- Optimistic locking is useful if the possibility for conflicts is very low – there are many records but relatively few users, or very few updates and mostly read-type operations.

Share Improve this answer Follow

answered Jul 16, 2017 at 7:49

Koenigsegg
615 ● 1 ● 10 ● 23

15

Let's say in an ecommerce app, a user wants to place an order. This code will get executed by multiple threads. In `pessimistic locking`, when we get the data from the DB, we lock it so no other thread can modify it. We process the data, update the data, and then commit the data. After that, we release the lock. Locking duration is long here, we have locked the database record from the beginning till committing.

In `optimistic locking`, we get the data and process the data without locking. So multiple threads can execute the code so far concurrently. This will speed up. While we update, we lock the data. We have to verify that no other thread updated that record. For example, If we had 100 items in inventory and we have to update it to 99 (because your code might be `quantity=queantity-1`) but if another thread already used 1 it should be 98. We had `race condition` here. In this case, we restart the thread so we execute the same code from the beginning. But this is an expensive operation, you already came to end but then restart. if we had a few race conditions, that would not be a big deal, If the race condition was high, there would be a lot of threads to restart. We might run in a loop. In the race condition is high, we should be using `pessimistic locking`

Share Improve this answer Follow

edited Oct 12 at 21:06

answered Sep 10, 2022 at 4:19

Yilmaz

48.7k ● 18 ● 210 ● 264

9

One use case for optimistic locking is to have your application use the database to allow one of your threads / hosts to 'claim' a task. This is a technique that has come in handy for me on a regular basis.

The best example I can think of is for a task queue implemented using a database, with multiple threads claiming tasks concurrently. If a task has status 'Available', 'Claimed', 'Completed', a db query can say something like "Set status='Claimed' where status='Available'. If multiple threads try to change the status in this way, all but the first thread will fail because of dirty data.

Note that this is a use case involving only optimistic locking. So as an alternative to saying "Optimistic locking is used when you don't expect many collisions", it can also be used where you expect collisions but want exactly one transaction to succeed.

Share Improve this answer Follow

answered Mar 27, 2017 at 16:59

Charlie Camp
146 ● 1 ● 2

8

Lot of good things have been said above about optimistic and pessimistic locking. One important point to consider is as follows:

When using optimistic locking, we need to cautious of the fact that how will application recover from these failures.

Specially in asynchronous message driven architectures, this can lead of out of order message processing or lost updates.

Failures scenarios need to be thought through.

Share Improve this answer Follow

answered Oct 30, 2019 at 4:24

Abhishek Shinde
121 ● 1 ● 3

6

On a more practical note, when updating a distributed system, optimistic locking in the DB may be inadequate to provide the consistency needed across all parts of the distributed system.

For example, in applications built on AWS, it is common to have data in both a DB (e.g. DynamoDB) and a storage (e.g. S3). If an update touches both DynamoDB and S3, an optimistic locking in DynamoDB could still leave the data in S3 inconsistent. In this type of cases, it is probably safer to use a pessimistic lock that is held in DynamoDB until the S3 update is finished. In fact, AWS provides a locking library for this purpose.

Share Improve this answer Follow

answered Aug 22, 2021 at 22:52
Big Pumpkin
4,447 ● 1 ● 30 ● 19

1 FWIW, AWS DynamoDB also supports optimistic locking. docs.aws.amazon.com/amazondynamodb/latest/developerguide/… – Big Pumpkin Aug 22, 2021 at 22:55

6

Optimistic locking means ***exclusive lock is not used when reading a row*** so **lost update** or **write skew** is not prevented. So, use **optimistic locking**:

- If **lost update** or **write skew** doesn't occur.
- Or, if there are no problems even if **lost update** or **write skew** occurs.

Pessimistic locking means ***exclusive lock is used when reading a row*** so **lost update** or **write skew** is prevented. So, use **pessimistic locking**:

- If **lost update** or **write skew** occurs.
- Or if there are some problems if **lost update** or **write skew** occurs.

In **MySQL** and **PostgreSQL**, you can use **exclusive lock** with `SELECT FOR UPDATE`.

You can check [my answer of the lost update and write skew examples]() with **optimistic locking(without `SELECT FOR UPDATE`)** and **pessimistic locking(with `SELECT FOR UPDATE`)** in **MySQL**.

Share Improve this answer Follow

edited Feb 9, 2023 at 5:34
Henry Ecker ♦
35.6k ● 19 ● 45 ● 64

answered Oct 14, 2022 at 15:43
Super Kai - Kazuya Ito
1

3

> **Optimistic locking and Pessimistic locking are two models for locking data in a database.**

Optimistic locking : where a record is locked only when changes are committed to the database.

Pessimistic locking : where a record is locked while it is edited.

Note : In both data-locking models, the lock is released after the changes are committed to the database.

Share Improve this answer Follow

answered Dec 28, 2021 at 8:32

Amit kumar
2,694 ● 11 ● 31 ● 43

0

I would like to add the following to the Vlad Mihalcea post above:

i believe that for the optimistic concurrency, done in the classical UPDATE way shown in the article, there is an edge case where 2 transactions in Read Commited level, would read the same version, say 3, and would also write the next version, 4, back, and the where clause on the update, version=3 would be satisfied for both concurrent transactions, and the commit would also be accepted for both transactions.

There is another way, more costly but safer, of doing optimistic concurrency, namely to store versions in another table, into which we do inserts for each next version, and have a unique constraint on the (entity id, version) tuple, so that even if the two concurrent transctions would be able to insert both, and of course this is an edge case, only one commit will be accepted by the database, the other commit would become a rollback.

What do you think?

Share Improve this answer Follow

answered Jul 16 at 15:37

NicuMarasoiu
806 ● 11 ● 25

The first session to update the row will block the second session until the first transaction ends. At that point, the second session will see an updated value in the database and update 0 rows if the first transaction was commited, or 1 row if the first transaction was rolled back. – MrBackend Sep 25 at 8:28