VLAD MIHALCEA

High-Performance Java Persistence and Hibernate

A beginner's guide to database locking and the lost update phenomena

SEPTEMBER 14, 2014 / VLADMIHALCEA

(Last Updated On: January 29, 2018)

Follow @vlad_mihalcea 7,641 followers

Introduction

A database is highly concurrent system. There's always a chance of update conflicts, like when two concurring transactions try to update the same record. If there would be only one database transaction at any time then all operations would be executed sequentially. The challenge comes when multiple transactions try to update the same database rows as we still have to ensure consistent data state transitions.

The SQL standard defines three consistency anomalies (phenomena):

- *Dirty reads*, prevented by Read Committed, Repeatable Read and Serializable isolation levels
- *Non-repeatable reads*, prevented by Repeatable Read and Serializable isolation levels
- *Phantom reads*, prevented by the Serializable isolation level

A lesser-known phenomenon is the *lost updates* anomaly and that's what we are going to discuss in this current article.

Isolation levels

Most database systems use Read Committed as the default isolation level (MySQL using Repeatable Read instead). Choosing the isolation level is about finding the right balance of consistency and scalability for our current application requirements.

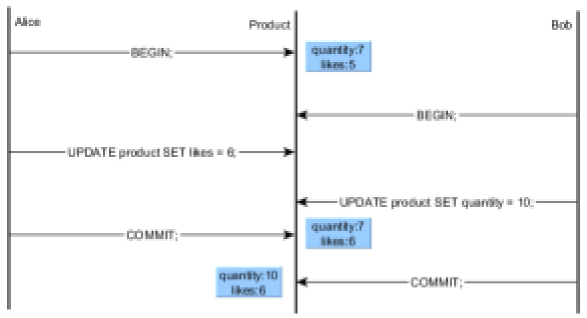
All the following examples are going to be run on PostgreSQL. Other database systems may behave differently according to their specific ACID implementation.

PostgreSQL uses both locks and MVCC (Multiversion Concurrency Control). In MVCC read and write locks are not conflicting, so readers don't block writers and writers don't block readers.

Because most applications use the default isolation level, it's very important to understand the Read Committed characteristics:

- Queries only see data committed before the query began and also the current transaction uncommitted changes
- Concurrent changes committed during a query execution won't be visible to the current query
- UPDATE/DELETE statements use locks to prevent concurrent modifications

If two transactions try to update the same row, the second transaction must wait for the first one to either commit or rollback and if the first transaction has been committed, then the second transaction DML WHERE clause must be reevaluated to see if the match is still relevant.



In this example, Bob's UPDATE must wait for Alice's transaction to end (commit/rollback) in order to proceed further.

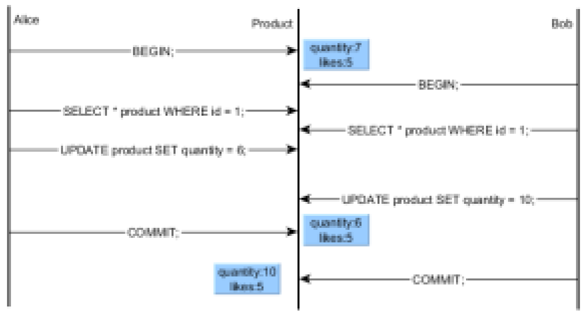
Read Committed accommodates more concurrent transactions than other stricter isolation levels, but less locking leads to better chances of losing updates.

Lost updates

If two transactions are updating different columns of the same row, then there is no conflict. The second update blocks until the first transaction is committed and the final result reflects both update changes.

If the two transactions want to change the same columns, the second transaction will overwrite the first one, therefore losing the first transaction update.

So an update is lost when a user overrides the current database state without realizing that someone else changed it between the moment of data loading and the moment the update occurs.


Alice
Product
Bob
BEGIN;
quantity:7 likes:5
BEGIN;
SELECT * product WHERE id = 1;
SELECT * product WHERE id = 1;
UPDATE product SET quantity = 6;
UPDATE product SET quantity = 10;
COMMIT;
quantity:6 likes:5
quantity:10 likes:5
COMMIT;

In this example Bob is not aware that Alice has just changed the quantity from 7 to 6, so her UPDATE is overwritten by Bob's change.

The typical find-modify-flush ORM strategy

Hibernate (like any other ORM tool) automatically translates entity state transitions to SQL queries. You first load an entity, change it and let the Hibernate flush mechanism synchronize all changes with the database.

```
public Product incrementLikes(Long id) {
    Product product = entityManager.find(Product.class, id
    product.incrementLikes();
    return product;
}

public Product setProductQuantity(Long id, Long quantity) 
    Product product = entityManager.find(Product.class, id
    product.setQuantity(quantity);
    return product;
}
```

As I've already pointed out, all UPDATE statements acquire write locks, even in Read Committed isolation. The persistence context write-behind policy aims to reduce the lock holding interval but the longer the period between the read and the write operations the more chances of getting into a lost update situation.

Hibernate includes all row columns in an UPDATE statement. This strategy can be changed to include only the dirty properties (through the @DynamicUpdate annotation) but the reference documentation warns us about its effectiveness:

> Although these settings can increase performance in some cases, they can actually decrease performance in others.

So let's see how Alice and Bob concurrently update the same Product using an ORM framework:

Alice	Bob

store=# BEGIN;
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
ID | LIKES | QUANTITY
—-+——-+———-
1 | 5 | 7
(1 ROW)
	store=# BEGIN;
	store=# SELECT * FROM PRODUCT WHERE ID = 1;
	
	ID | LIKES | QUANTITY
	—-+——-+———-
	1 | 5 | 7
	(1 ROW)

store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 7) WHERE ID = 1;	
	store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (5, 10) WHERE ID = 1;

store=# COMMIT; store=# SELECT * FROM PRODUCT WHERE ID = 1; ID | LIKES | QUANTITY —-+——-+———- 1 | 6 | 7 (1 ROW)	
	store=# COMMIT; store=# SELECT * FROM PRODUCT WHERE ID = 1; ID | LIKES | QUANTITY —-+——-+———- 1 | 5 | 10 (1 ROW)
store=# SELECT * FROM PRODUCT WHERE ID = 1; ID | LIKES | QUANTITY —-+——-+———- 1 | 5 | 10 (1 ROW)	

Again Alice’s update is lost without Bob ever knowing he overwrote her changes. We should always prevent data integrity anomalies, so let’s see how we can overcome this phenomenon.

Repeatable Read

Using Repeatable Read (as well as Serializable which offers an even stricter isolation level) can prevent lost updates across concurrent database transactions.

Alice	Bob

store=# BEGIN;
store=# SET TRANSACTION
	store=# BEGIN;
	store=# SET TRANSACTION ISOLATION

ISOLATION LEVEL REPEATABLE READ; store=# SELECT * FROM PRODUCT WHERE ID = 1; ID | LIKES | QUANTITY —-+——-+———- 1 | 5 | 7 (1 ROW)	LEVEL REPEATABLE READ; store=# SELECT * FROM PRODUCT WHERE ID = 1; ID | LIKES | QUANTITY —-+——-+———- 1 | 5 | 7 (1 ROW)
store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 7) WHERE ID = 1;	
	store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (5, 10) WHERE ID = 1;

store=# COMMIT;
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
ID | LIKES | QUANTITY
—-+——-+———-
1 | 6 | 7
(1 ROW)	

	ERROR: could not serialize access due to concurrent update store=# SELECT * FROM PRODUCT WHERE ID = 1; ERROR: current transaction is aborted, commands ignored until end of transaction block (1 ROW)

This time, Bob couldn't overwrite Alice's changes and his transaction was aborted. In Repeatable Read, a query will see the data snapshot as of the start of the current transaction. Changes committed by other concurrent transactions are not visible to the current transaction.

If two transactions attempt to modify the same record, the second transaction will wait for the first one to either commit or rollback. If the first transaction commits, then the second one must be aborted to prevent lost updates.

SELECT FOR UPDATE

Another solution would be to use the FOR UPDATE with the default Read Committed isolation level. This locking clause acquires the same write locks as with UPDATE and DELETE statements.

Alice	Bob

store=# BEGIN;
store=# SELECT * FROM PRODUCT WHERE ID = 1 FOR UPDATE;
 	
ID | LIKES | QUANTITY
—-+——-+———-
1 | 5 | 7
(1 ROW)
	store=# BEGIN;
	store=# SELECT * FROM PRODUCT WHERE ID = 1 FOR UPDATE;

store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 7) WHERE ID = 1;
store=# COMMIT;
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
ID | LIKES | QUANTITY
—-+——-+———-
1 | 6 | 7
(1 ROW)	

	id | likes | quantity
	—-+——-+———-
	1 | 6 | 7
	(1 row)
	
	store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 10) WHERE ID = 1;
	UPDATE 1
	store=# COMMIT;
	COMMIT
	store=# SELECT * FROM PRODUCT WHERE ID = 1;

	id | likes | quantity —-+——-+———- 1 | 6 | 10 (1 row)

Bob couldn't proceed with the SELECT statement because Alice has already acquired the write locks on the same row. Bob will have to wait for Alice to end her transaction and when Bob's SELECT is unblocked he will automatically see her changes, therefore Alice's UPDATE won't be lost.

Both transactions should use the FOR UPDATE locking. If the first transaction doesn't acquire the write locks, the lost update can still happen.

Alice	**Bob**

store=# BEGIN;
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
id | likes | quantity
—-+——-+———-
1 | 5 | 7
(1 row)	

	store=# BEGIN; store=# SELECT * FROM PRODUCT WHERE ID = 1 FOR UPDATE id | likes | quantity —-+——-+———- 1 | 5 | 7 (1 row)
store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 7) WHERE ID = 1;	

	store=# UPDATE PRODUCT SET (LIKES, QUANTITY) = (6, 10) WHERE ID = 1;
	store=# SELECT * FROM PRODUCT WHERE ID = 1;

	id | likes | quantity —-+——-+———- 1 | 6 | 10 (1 row) store=# COMMIT;
store=# SELECT * FROM PRODUCT WHERE ID = 1; id | likes | quantity —-+——-+———- 1 | 6 | 7 (1 row) store=# COMMIT;	
	store=# SELECT * FROM PRODUCT WHERE ID = 1; id | likes | quantity —-+——-+———- 1 | 6 | 7 (1 row)

Alice’s UPDATE is blocked until Bob releases the write locks at the end of his current transaction. But Alice’s persistence context is using a stale entity snapshot, so she overwrites Bob changes, leading to another lost update situation.

Optimistic Locking

My favorite approach is to replace pessimistic locking with an optimistic locking mechanism. Like MVCC, optimistic locking defines a versioning concurrency control model that works without acquiring additional database write locks.

The product table will also include a version column that prevents old data snapshots to overwrite the latest data.

Alice	Bob

store=# BEGIN;
BEGIN
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
id | likes | quantity | version
—-+——-+———-+———
1 | 5 | 7 | 2
(1 row)
	store=# BEGIN;
	BEGIN
	store=# SELECT * FROM PRODUCT WHERE ID = 1;
	
	id | likes | quantity | version
	—-+——-+———-+———
	1 | 5 | 7 | 2
	(1 row)

store=# UPDATE PRODUCT SET (LIKES, QUANTITY, VERSION) = (6, 7, 3) WHERE (ID, VERSION) = (1, 2);
UPDATE 1	

	store=# UPDATE PRODUCT SET (LIKES, QUANTITY, VERSION) = (5, 10, 3) WHERE (ID, VERSION) = (1, 2);

store=# COMMIT;
store=# SELECT * FROM PRODUCT WHERE ID = 1;
 	
id | likes | quantity | version
—-+——-+———-+———
1 | 6 | 7 | 3
(1 row)	

	UPDATE 0
	store=# COMMIT;
	store=# SELECT * FROM PRODUCT WHERE ID = 1;
	
	id | likes | quantity | version
	—-+——-+———-+———
	1 | 6 | 7 | 3
	(1 row)

Every UPDATE takes the load-time version into the WHERE clause, assuming no one has changed this row since it was retrieved from the database. If some other transaction manages to commit a newer entity version, the UPDATE WHERE clause will no longer match any row and so the lost update is prevented.

Hibernate uses the PreparedStatement#executeUpdate result to check the number of updated rows. If no row was matched, it then throws a StaleObjectStateException (when using Hibernate API) or an OptimisticLockException (when using JPA).

Like with Repeatable Read, the current transaction and the persistence context are aborted, with respect to atomicity guarantees.

Conclusion

Lost updates can happen unless you plan on preventing such situations. Other than optimistic locking, all pessimistic locking approaches are effective only in the scope of the same database transaction, when both the SELECT and the UPDATE statements are executed in the same physical transaction.

Email Address *

10 000 readers have found this blog worth following!

If you **subscribe** to my newsletter, you'll get:

- A **free sample** of my Video Course about running Integration tests at warp-speed using Docker and tmpfs
- **3 chapters** from my book, **High-Performance Java Persistence**,
- a **10% discount** coupon for my book.

Get the most out of your persistence layer!

Subscribe

Advertisements

Share this:

Tweet Share 1 Share submit G+

Like this:

Loading...

Related

A beginner's guide to the Phantom Read anomaly, and how it differs between 2PL and MVCC
January 31, 2017
In "Database"

A beginner's guide to ACID and database transactions
January 5, 2014
In "Transactions"

A beginner's guide to read and write skew phenomena
October 20, 2015
In "Java"

Categories: Hibernate, SQL, Transactions

Tags: database locks, hibernate, lost updates, MVCC, PostgreSQL, read committed, repeatable read, Training, transactions, Tutorial

← How does the bytecode enhancement dirty checking mechanism work in Hibernate 4.3

How to prevent lost updates in long conversations →

20 thoughts on “A beginner’s guide to database locking and the lost update phenomena”

Guy Pardon says:
APRIL 23, 2018 AT 9:44 AM

Nice post!

For optimistic locking: what always confuses me is whether the UPDATE … WHERE VERSION = (load-time version) needs repeatable read too, or not.

For instance: I guess it depends on the DBMS in question whether simple UPDATE … WHERE … statements are safe against concurrent executions / lost updates: what if 2 concurrent updates specify the same load-time version and execute in parallel – they could both see the same version, and update it, with lost updates as a result.

What do you think?

Thanks

★ Loading...

REPLY

vladmihalcea says:
APRIL 23, 2018 AT 10:59 AM

Optimistic locking does not require REPEATABLE READ on the database side. That's because, no matter what isolation level is chosen, writes are always executed sequentially. Once a Tx modifies a record, no other Tx can modify the same record until the first Tx commits or rolls back.

If you use REPEATABLE READ of the DB side, and you previously read a certain row that you now want to update, when doing the UPDATE, the predicate goes against the latest row state and if the Tx snapshot differs than the tuple snapshot, the UPDATE will fail with a "could not serialize access due to concurrent update" exception.

If you didn't read the record before, and the row has not been modified, the UPDATE will work just fine.

If you didn't read the record before, and the row was modified by a concurrent Tx, the row is now locked by the concurrent Tx and your UPDATE will wait for the first Tx that locked the record to release the row-level lock (after commit or rollback).

★ Loading...

REPLY

Guy Pardon says:
APRIL 23, 2018 AT 11:06 AM

Thanks!

Not sure if UPDATE … WHERE … is a simple write, since it also contains a query-party (the predicate). So in a way, it's a read-write scenario like the ones you describe in the article, no?

Do you have any pointers to the specs that say that predicates have to be re-evaluated after concurrent modifications?

★ Loading...

vladmihalcea says:
APRIL 23, 2018 AT 11:09 AM

Sure thing. Just search for "The search condition of the command (the WHERE clause) is re-evaluated to see if the updated version of the row still matches the search condition." in this PostgreSQL manual page. It's the same for other DBs relying on MVCC.

★ Loading...

SpaceMiomio says:
SEPTEMBER 15, 2017 AT 7:14 AM

This confused me: If Read Comitted has the character "Queries only see data committed before the query began and also the current transaction uncommitted changes", then how could it leads to non-repeatable read?

★ Loading...

REPLY

vladmihalcea says:
SEPTEMBER 15, 2017 AT 7:19 AM

It's simple.

i. Alice opens a new transaction and reads the account balance and the value is 100.
ii. Bob opens a new transaction, withdraws 50 dollars and commits.
iii. Alice reads again the account balance and the value is 50 now.

There you go! You've just got a non-repeatable read.

★ Loading...

REPLY

Tomasz Ulinski says:
AUGUST 25, 2017 AT 4:16 PM

Great article Vlad, I greatly appreciate your passion for sharing knowledge.

.

All the best!

★ Loading...

REPLY

vladmihalcea says:
AUGUST 25, 2017 AT 4:20 PM

Thanks for your fine words.

★ Loading...

REPLY

← OLDER COMMENTS

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

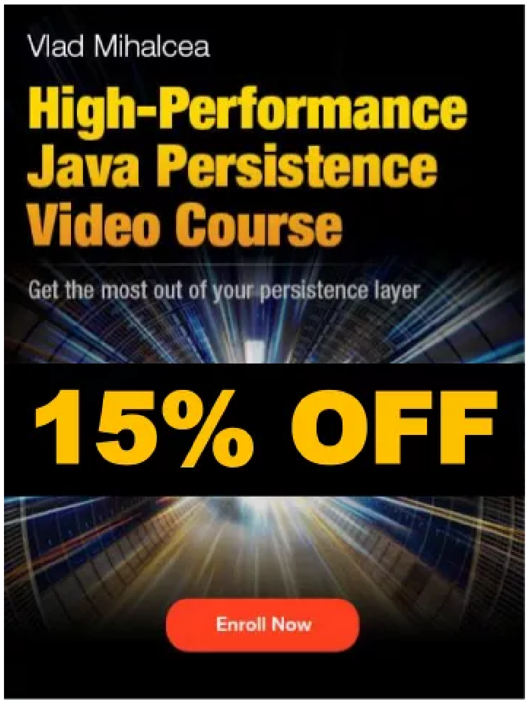
Website

☐ Sign me up for the newsletter!

Post Comment

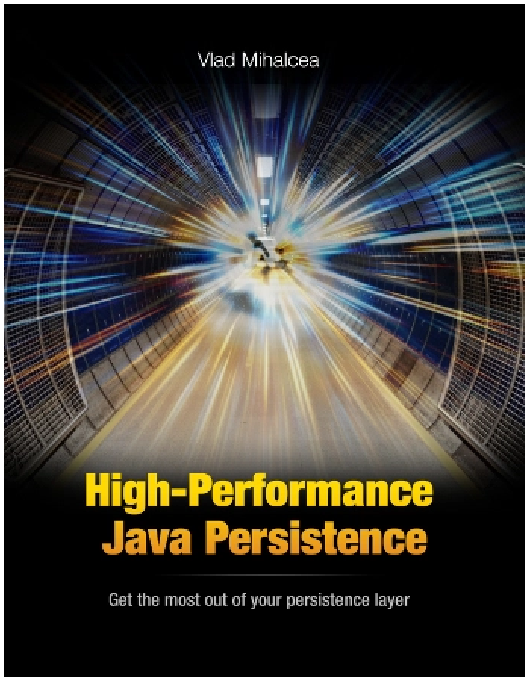
☐ Notify me of follow-up comments by email.

VIDEO COURSE

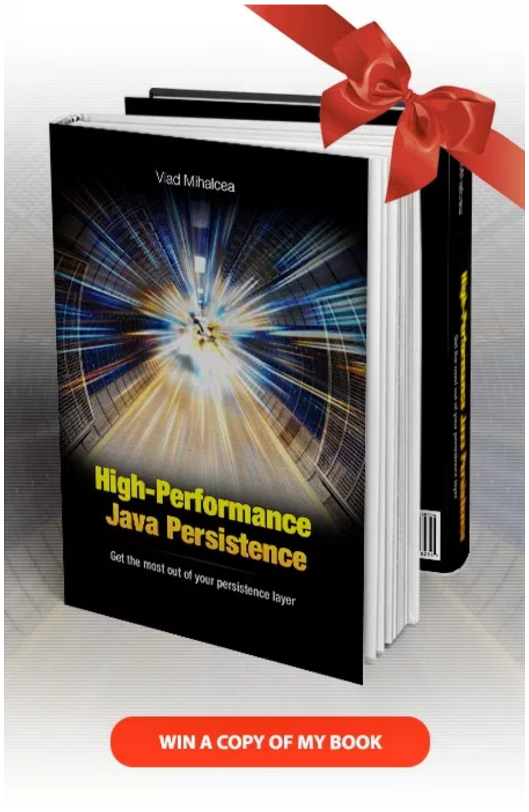

Vlad Mihalcea
High-Performance Java Persistence Video Course
Get the most out of your persistence layer
15% OFF
Enroll Now

JDK.IO WORKSHOP – COPENHAGEN

HIGH-PERFORMANCE JAVA PERSISTENCE


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer

WIN A COPY OF MY BOOK!


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer
WIN A COPY OF MY BOOK

FACEBOOK PAGE

vladmihalcea.com
403 likes

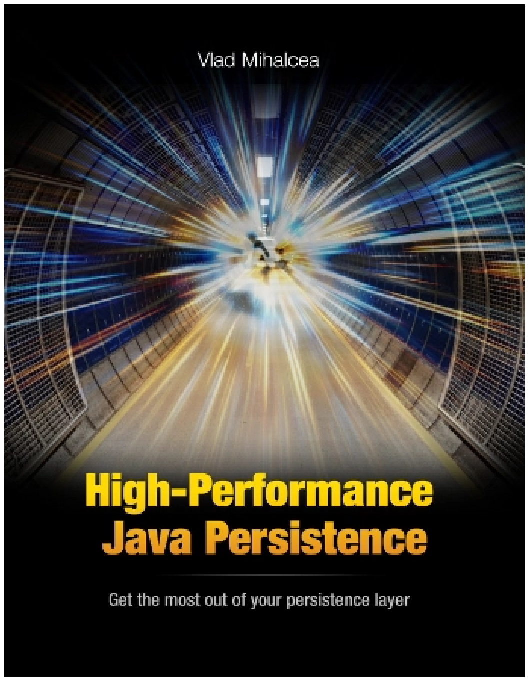
Like Page

HIBERNATE PERFORMANCE TUNING TIPS

HYPERSISTENCE

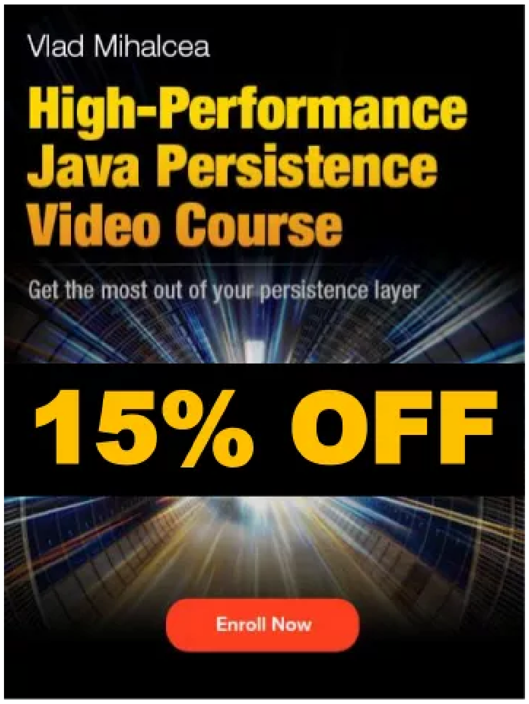
WIN A COPY OF MY BOOK!

HIGH-PERFORMANCE JAVA PERSISTENCE


Vlad Mihalcea
High-Performance Java Persistence
Get the most out of your persistence layer

JDK.IO WORKSHOP – COPENHAGEN

VIDEO COURSE


Vlad Mihalcea
High-Performance Java Persistence Video Course
Get the most out of your persistence layer
15% OFF
Enroll Now

Search ...

RSS - Posts

RSS - Comments

ABOUT

About

Privacy Policy

Terms of Service