**Hibernate – dynamic-insert attribute example**

**What is dynamic-insert**

The dynamic-insert attribute tells Hibernate whether to include null properties in the SQL INSERT statement. Let explore some examples to understand more clear about it.

**Dynamic-insert example**

**1. dynamic-insert=false**

The default value of dynamic-insert is false, which means **include null properties**in the Hibernate’s SQL INSERT statement.

For example, try set some null values to an object properties and save it.

StockTransaction stockTran = **new** StockTransaction();

//stockTran.setPriceOpen(new Float("1.2"));

//stockTran.setPriceClose(new Float("1.1"));

//stockTran.setPriceChange(new Float("10.0"));

stockTran.setVolume(2000000L);

stockTran.setDate(**new** Date());

stockTran.setStock(stock);

session.save(stockTran);

Turn on the Hibernate “show\_sql” to true, you will see the following insert SQL statement.

Hibernate:

insert

into

stock\_transaction

(DATE, PRICE\_CHANGE, PRICE\_CLOSE, PRICE\_OPEN, STOCK\_ID, VOLUME)

values

(?, ?, ?, ?, ?, ?)

**Hibernate will generate the unnecessary columns** (PRICE\_CHANGE, PRICE\_CLOSE, PRICE\_OPEN) for the insertion.

**2. dynamic-insert=true**

If set the dynamic-insert to true, which means **exclude null property values** in the Hibernate’s SQL INSERT statement.

For example, try set some null values to an object properties and save it again.

StockTransaction stockTran = **new** StockTransaction();

//stockTran.setPriceOpen(new Float("1.2"));

//stockTran.setPriceClose(new Float("1.1"));

//stockTran.setPriceChange(new Float("10.0"));

stockTran.setVolume(2000000L);

stockTran.setDate(**new** Date());

stockTran.setStock(stock);

session.save(stockTran);

Turn on the Hibernate “show\_sql” to true. You will see the different insert SQL statement.

Hibernate:

insert

into

stock\_transaction

(DATE, STOCK\_ID, VOLUME)

values

(?, ?, ?)

**Hibernate will generate only the necessary columns** (DATE, STOCK\_ID, VOLUME) for the insertion.

**Performance issue**

In certain situations, such as a very large table with hundreds of columns (legacy design), or a table contains extremely large data volume, insert something not necessary definitely will drop down your system performance.

**How to configure it**

You can configure the dynamic-insert properties value through annotation or XML mapping file.

**1. Annotation**

@Entity

@Table(name = "stock\_transaction”

@org.hibernate.annotations.Entity(dynamicInsert = **true**)

**public** **class** StockTransaction **implements** java.io.Serializable {

## 2. XML mapping

<class . table=*"stock\_transaction"* dynamic-insert=*"true"*>

<id name=*"tranId"* type=*"java.lang.Integer"*>

<column name=*"TRAN\_ID"* />

<generator class=*"identity"* />

</id>

## Conclusion

This little “**dynamic-insert**” tweak may increase your system performance, and highly recommends to do it. However, one question in my mind is why Hibernate set it to false by default?

**Hibernate – dynamic-update attribute example**

**What is dynamic-update**

The dynamic-update attribute tells Hibernate whether to include unmodified properties in the SQL UPDATE statement.

**Dynamic-update example**

**1. dynamic-update=false**

The default value of dynamic-update is false, which means **include unmodified properties** in the Hibernate’s SQL update statement.

For example, get an object and try modify its value and update it.

Query q = session.createQuery("from StockTransaction where tranId = :tranId ");

q.setParameter("tranId", 11);

StockTransaction stockTran = (StockTransaction)q.list().get(0);

stockTran.setVolume(4000000L);

session.update(stockTran);

Hibernate will generate the following update SQL statement.

Hibernate:

update

stock\_transaction

set

DATE=?,

PRICE\_CHANGE=?,

PRICE\_CLOSE=?,

PRICE\_OPEN=?,

STOCK\_ID=?,

VOLUME=?

where

TRAN\_ID=?

Hibernate will update all the unmodified columns.

**2. dynamic-update=true**

If set the dynamic-insert to true, which means **exclude unmodified properties** in the Hibernate’s SQL update statement.

For example, get an object and try modify its value and update it again.

Query q = session.createQuery("from StockTransaction where tranId = :tranId ");

q.setParameter("tranId", 11);

StockTransaction stockTran = (StockTransaction)q.list().get(0);

stockTran.setVolume(4000000L);

session.update(stockTran);

Hibernate will generate different update SQL statement.

Hibernate:

update

stock\_transaction

set

VOLUME=?

where

TRAN\_ID=?

Hibernate will update the modified columns only.

**Performance issue**  
In a large table with many columns (legacy design) or contains large data volumes, update some unmodified columns are absolutely unnecessary and great impact on the system performance.

## How to configure it

You can configure “dynamic-update” properties via annotation or XML mapping file.

**1. Annotation**

@Entity

@Table(name = "stock\_transaction", catalog = "mkyong")

@org.hibernate.annotations.Entity(

dynamicUpdate = **true**

)

**public** **class** StockTransaction **implements** java.io.Serializable {

**2. XML mapping**

<class . table=*"stock\_transaction"* catalog=*"mkyong"*

dynamic-update=*"true"*>

<id name=*"tranId"* type=*"java.lang.Integer"*>

<column name=*"TRAN\_ID"* />

<generator class=*"identity"* />

</id>

**Conclusion**

This little “**dynamic-update**” tweak will definitely increase your system performance, and highly recommended to do it.