**iBATIS and Hibernate are quite different beasts.**

The way I tend to look at it is this: Hibernate works better if your view is more object-centric. If however you view is more database-centric then iBATIS is a much stronger choice.

If you're in complete control of your schema and you don't have an extremely high throughput requirement then Hibernate can work quite well. The object model makes for fairly convenient code but at a huge complexity cost.

If you're dealing with a "legacy" database schema where you need to write fairly complicated SQL queries then chances are iBATIS will work better.

HQL (Hibernate Query Language) is another language you'll have to learn and even then you'll probably find cases where you still need to write SQL. What's more, chances are you will at some spend half a day figuring out the right combination of XML, properties, annotations, etc to get Hibernate to generate a performant SQL query.

There is no universal "A is better than B" answer for this question.

**The reason is that you're trying to do one of two things typically:**

1. Create/Update/Delete some complex domain entities
2. Run analytic fetch queries (i.e. summation/aggregation queries)

**Hibernate** works well for case 1 allowing you to just make a POJO and persist/update it. It also does this quickly, unless your domain is quite large.

**myBatis** is great for fetch queries (case 2) where you just want an answer. Hibernate would attempt to load the entire object graph and you'd need to start tuning queries with LazyLoading tricks to keep it working on a large domain. Conversely if you just want some analytic POJO page, the myBatis implementation of the same query would be trivial.

Because of this, myBatis [is faster than Hibernate](http://arxiv.org/ftp/arxiv/papers/0710/0710.1404.pdf) at SELECTS.

These two cases are the difference between **Commands** where you want to change the domain data and **Responses** where you just want to fetch some data.

Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or the mappedBy) side.

**Joins:**

<https://stackoverflow.com/questions/38549/what-is-the-difference-between-inner-join-and-outer-join>

**Hibernate/JPA**

<https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html>

<https://stackoverflow.com/questions/2749689/what-is-the-owning-side-in-an-orm-mapping>

**Why is the notion of a owning side necessary:**

The idea of a owning side of a bidirectional relation comes from the fact that in relational databases there are no bidirectional relations like in the case of objects. In databases we only have unidirectional relations - foreign keys.

**What is the reason for the name 'owning side'?**

The owning side of the relation tracked by Hibernate is the side of the relation that *owns* the foreign key in the database.

**What is the problem that the notion of owning side solves?**

Take an example of two entities mapped *without* declaring a owning side:

@Entity

@Table(name="PERSONS")

public class Person {

@OneToMany

private List<IdDocument> idDocuments;

}

@Entity

@Table(name="ID\_DOCUMENTS")

public class IdDocument {

@ManyToOne

private Person person;

}

From a OO point of view this mapping defines not one bi-directional relation, but *two* separate uni-directional relations.

The mapping would create not only tables PERSONS and ID\_DOCUMENTS, but would also create a third association table PERSONS\_ID\_DOCUMENTS:

CREATE TABLE PERSONS\_ID\_DOCUMENTS

(

persons\_id bigint NOT NULL,

id\_documents\_id bigint NOT NULL,

CONSTRAINT fk\_persons FOREIGN KEY (persons\_id) REFERENCES persons (id),

CONSTRAINT fk\_docs FOREIGN KEY (id\_documents\_id) REFERENCES id\_documents (id),

CONSTRAINT pk UNIQUE (id\_documents\_id)

)

Notice the primary key pk on ID\_DOCUMENTS only. In this case Hibernate tracks both sides of the relation independently: If you add a document to relation Person.idDocuments, it inserts a record in the association table PERSON\_ID\_DOCUMENTS.

On the other hand, if we call idDocument.setPerson(person), we change the foreign key person\_id on table ID\_DOCUMENTS. Hibernate is creating *two* unidirectional (foreign key) relations on the database, to implement *one* bidirectional object relation.

**How the notion of owning side solves the problem:**

Many times what we want is only a foreign key on table ID\_DOCUMENTS towards PERSONSand the extra association table.

To solve this we need to configure Hibernate to stop tracking the modifications on relation Person.idDocuments. Hibernate should only track the *other* side of the relation IdDocument.person, and to do so we add **mappedBy**:

@OneToMany(mappedBy="person")

private List<IdDocument> idDocuments;

**What does it mean mappedBy ?**

This means something like: "modifications on this side of the relation are already **Mapped By** the other side of the relation IdDocument.person, so no need to track it here separately in an extra table."

**Are there any GOTCHAs, consequences?**

Using **mappedBy**, If we only call person.getDocuments().add(document), the foreign key in ID\_DOCUMENTS will **NOT** be linked to the new document, because this is not the owning /tracked side of the relation!

To link the document to the new person, you need to explicitly call document.setPerson(person), because that is the ***owning side*** of the relation.

When using **mappedBy**, it is the responsibility of the developer to know what is the owning side, and update the correct side of the relation in order to trigger the persistence of the new relation in the database.

ou can imagine that the owning side is the entity that has the reference to the other one. In your excerpt, you have an one-to-one relationship. Since it's a symmetric relation, you'll end up having that if object A is in relation with object B then also the vice-versa is true.

This means that saving into object A a reference to object B and saving in object B a reference to object A will be redundant: that's why you choose which object "owns" the other having the reference to it.

When you have got an one-to-many relationship, the objects related to the "many" part will be the owning side, otherwise you would have to store many references from a single object to a multitude. To avoid that, every object in the second class will have a pointer to the single one they refer to (so they are the owning side).

For a many-to-many relationship, since you will need a separate mapping table anyway there won't be any owning side.

In conclusion the **owning side** is the entity that has the reference to the other.

<https://stackoverflow.com/questions/12493865/what-is-the-difference-between-inversedby-and-mappedby>

In bidirectional relationship has both an owning side and an inverse side

**mappedBy** : put into The inverse side of a bidirectional relationship To refer to its owning side

**inversedBy** : put into The owning side of a bidirectional relationship To refer to its inverse side

**AND**

**mappedBy** attribute used with the OneToOne, OneToMany, or ManyToMany mapping declaration.

**inversedBy** attribute used with the OneToOne, ManyToOne, or ManyToMany mapping declaration.

**Notice** : The owning side of a bidirectional relationship the side that contains the foreign key.

there two reference about inversedBy and mappedBy into Doctrine Documentation : [First Link](https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/association-mapping.html),[Second Link](https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/unitofwork-associations.html)













