arnaud.nauwynck@gmail.com

Architecture Design

Part1 : Entity (Domain)

DTO – Entity – Model classes

& RestController/GraphQL – Service – Repository classes

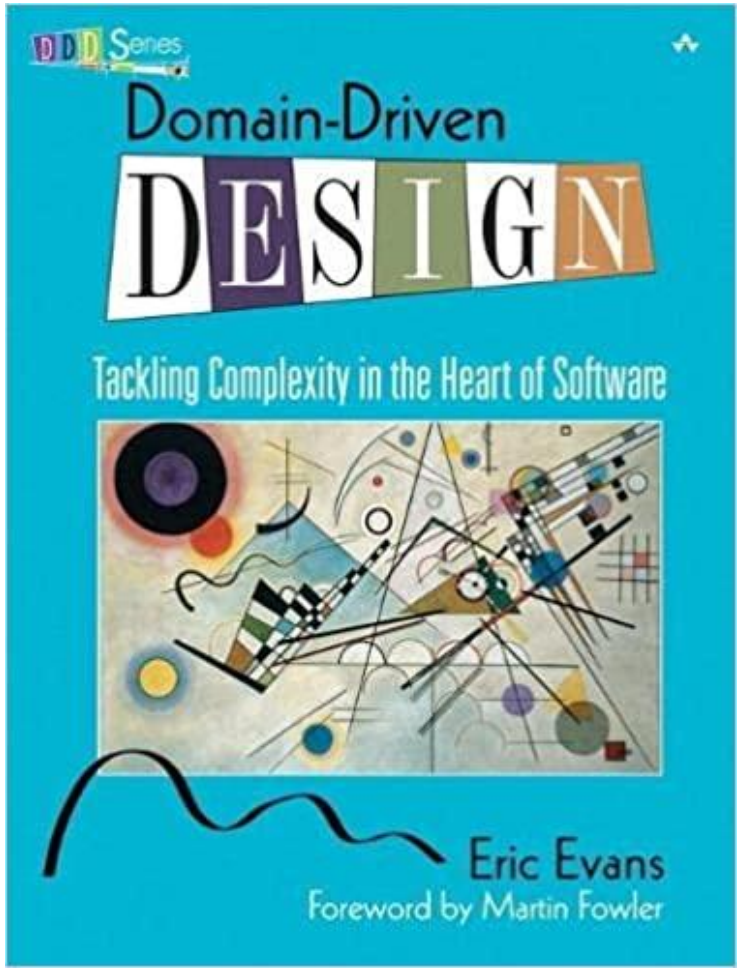
This document:
http://github.com/arnaud-nauwynck/Presentations/java/Architecture-Design-part1-Entity.pdf

Step1 :

Analysis the objectives
Define the « **Ubiquituous Langage** »

Same langage for end-users & IT

« Domain Driven Design » book



Author: Eric Evans

« DDD » : THE Reference

after introduction to UML modelisation in « Gof » = « Design Pattern » book

« THE » Domain : fit your « Bounded Context »

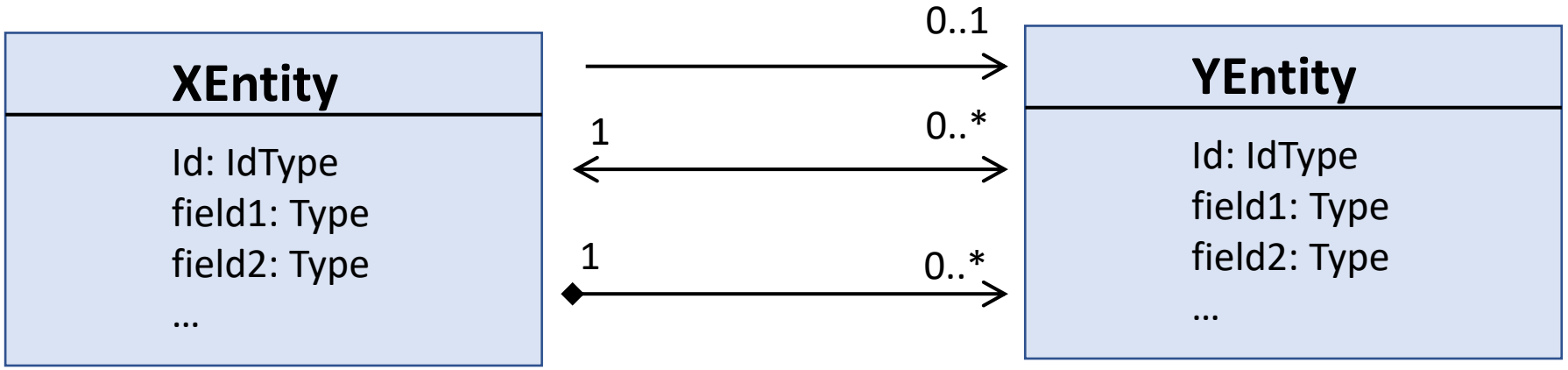
There is NOT a perfect Domain for matching all applications

1 Domain is specific to 1 App
Focus on the « kernel » / « fundamentals» of your app

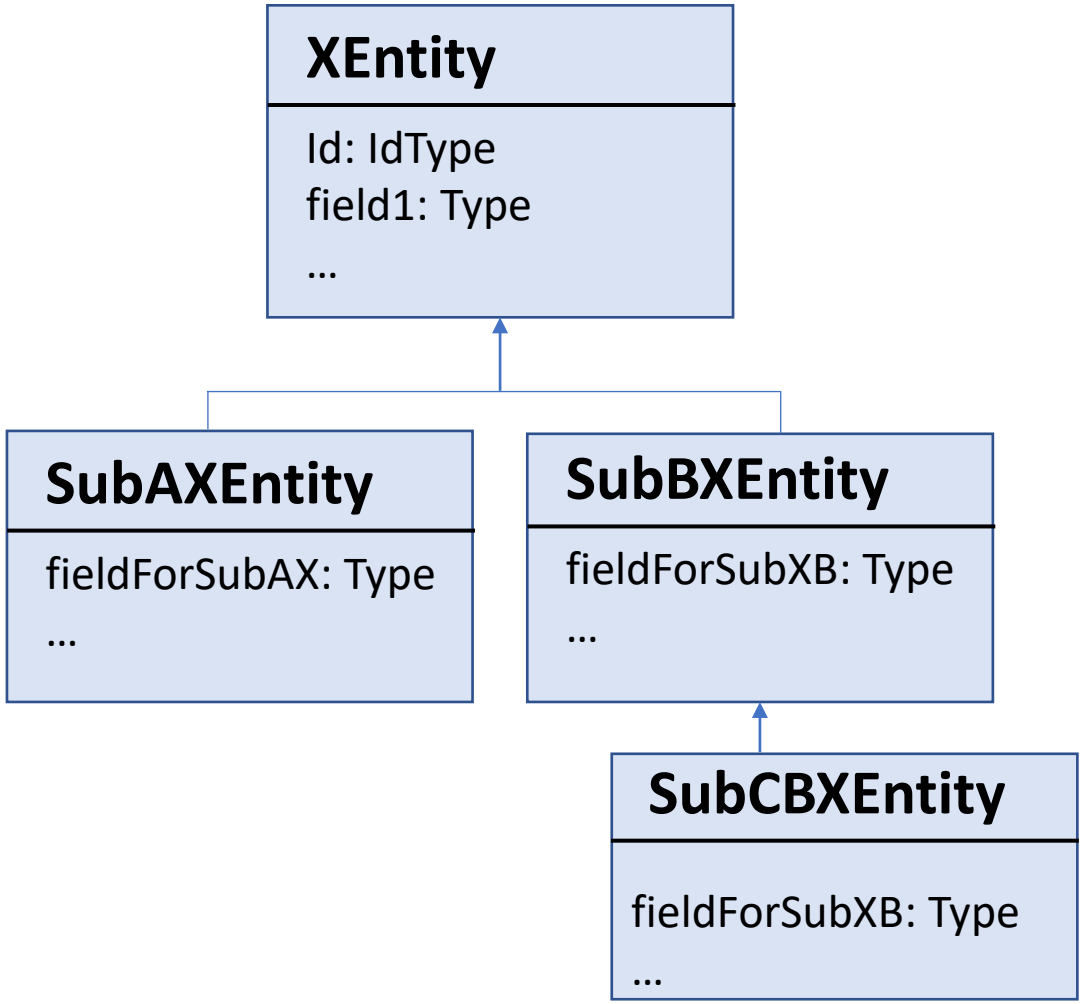
In DDD book : « Bounded Contexts »

Step 2 : Classes for Domain

Handwritten Drawings in UML
… share knowledge with team



UML … classes hierarchies? interfaces ?



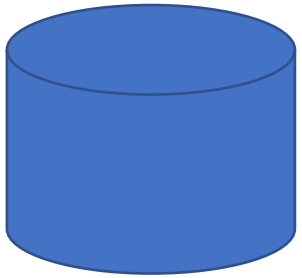
Start simple …

NO interface
NO « diamond » inheritance
NO sub-class without new fields
NOT focusing on « behaviour »
… focusing only on « data »

Favor delegation: « has »
rather than inheritance: « is »

Focus on (Persistence) Data vs Behavior

An Entity is unique via its « ID »
And completely defined via all its state data

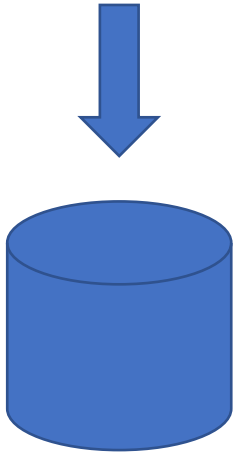


Not focusing (yet) on behavior methods on Entity
Assume only Getter/Setters
+ Add/Remove relations to other Entities

Behavior will be handle by Service classes
or corresponding Model class

Step 3 : … Your Database Technology

Focus on

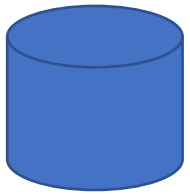


Plain Old « OLTP »
Transaction - Relational Database

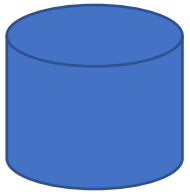
Example: Postgresql, Mysql, Oracle

Out of scope here

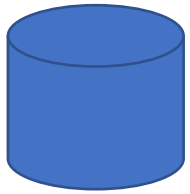
Document database
(Mongodb, CouchDb..)



Key-Value database
(HBase, Redis, ..)



Graph DB
(Neo4j..)



Object-Oriented DB
(Versant, Db4O, ? ..)
… would be fine, but not viable/mature/widespread

Object<-> Relational Mapping
… « The Vietnam of Computer Science »

the vietnam of computer science

All | Images | Videos | News | Maps | More | Tools

About 65,200,000 results (0.52 seconds)

https://blogs.tedneward.com › post › the-vietnam-of-co...

The Vietnam of Computer Science - Ted Neward's Blog

Jun 26, 2006 — Although it may seem trite to say it, Object/Relational Mapping is **the Vietnam of Computer Science**. It represents a quagmire which starts ...

https://blog.codinghorror.com › object-relational-mappi...

Object-Relational Mapping is the Vietnam of Computer Science

Jun 26, 2006 — Object-Relational Mapping is **the Vietnam of Computer Science** ... I had an opportunity to meet Ted Neward at TechEd this year. Ted, among other ...

https://www.semanticscholar.org › paper › The-Vietnam-o...

[PDF] The Vietnam of Computer Science | Semantic Scholar

Although it may seem trite to say it, Object/Relational Mapping is **the Vietnam of Computer Science**. It represents a quagmire which starts well, ...

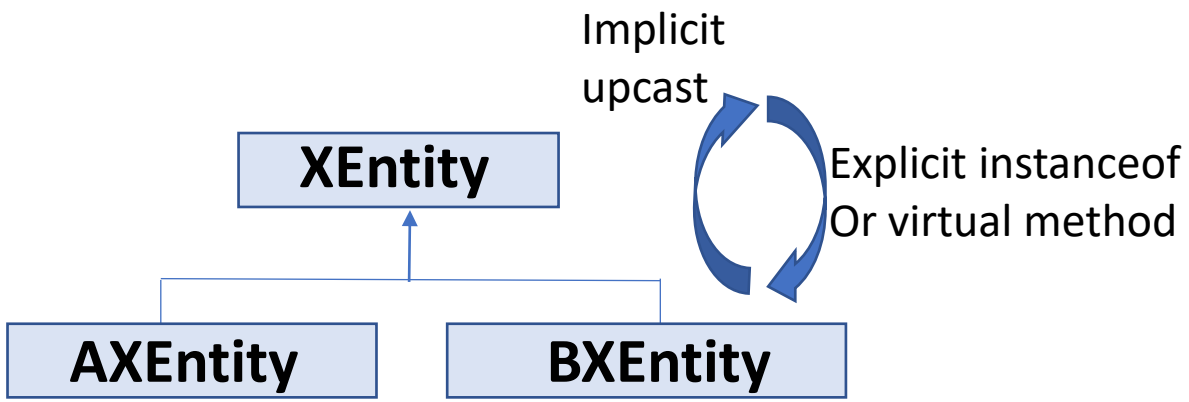
26 Jun 2006

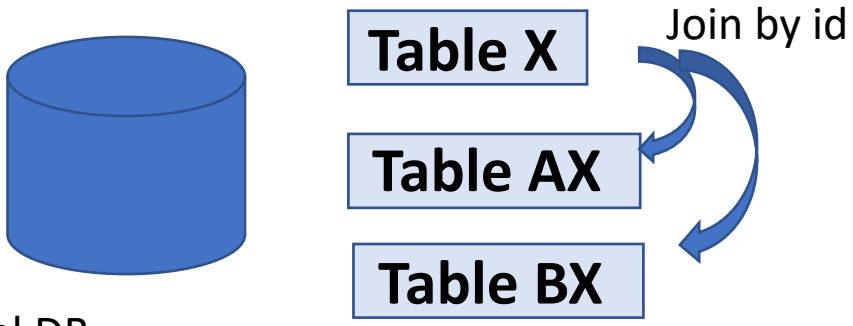
Object-Relational Mapping is the Vietnam of Computer Science

I had an opportunity to meet Ted Neward at TechEd this year. Ted, among other things, famously coined the phrase **"Object-Relational mapping is the Vietnam of our industry"** in late 2004.

It's a scary analogy, but an apt one. I've seen developers struggle for *years* with the huge mismatch between relational database models and traditional object models. And all the solutions they come up with seem to make the problem worse. I agree with Ted completely; **there is no good solution to the object/relational mapping problem**.

Mismatch in
Object (Class Hierarchy) <-> Table Storage





Relational DB
= FLAT TABULAR structure
… with some Foreign Key->PK relations

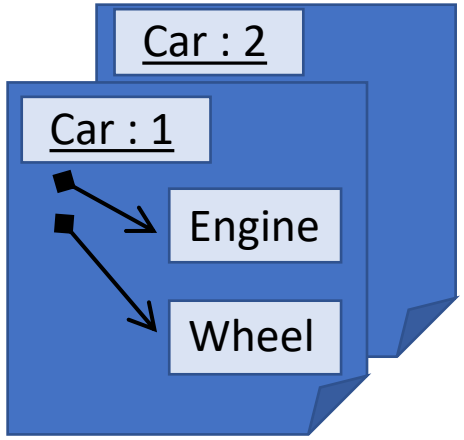
Querying « X » … including « AX » & « BX » → How to select « X » union « join A » union « join B » ??

Querying « AX » … where fieldX = … and fieldAX=… → How to efficiently search with index on table »X »
and simultaneoulsy use index on table « A »
.. with join ??

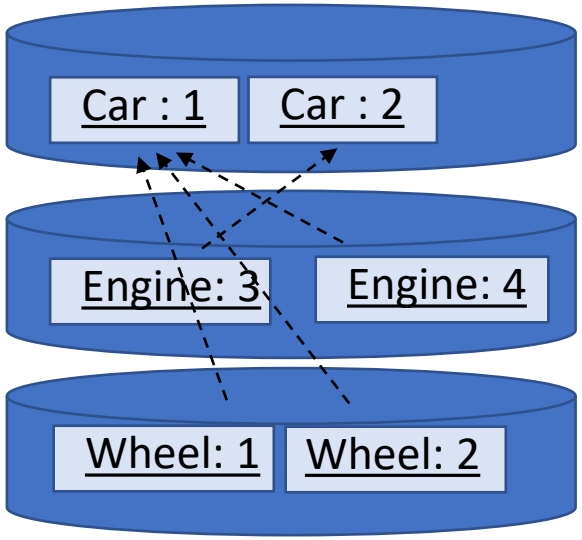
Performances Mismatch …
Object with nested Child List <-> Join global table List

```
{  carId: "abc« ,
   engine: {  type: "xx", … }
   wheels: [
      { pos:"front-left", .. },
      { pos: "front-right"}, ..
   ]
}
```

NOSql (ex:PARQUET),
or Document oriented DB



Relational DB: storage by type



…load all in 1 Query:

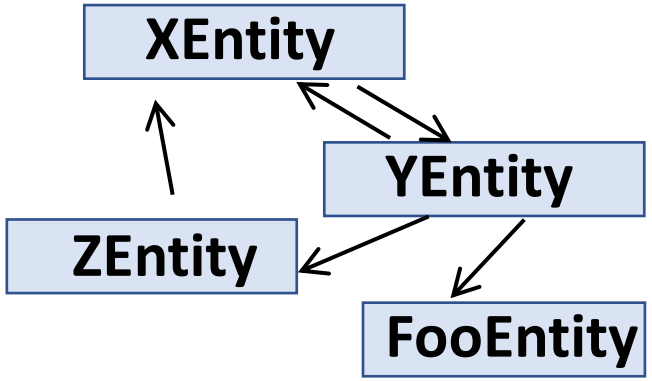
Select car, car.engine,car.wheel
From car
where id=?

… Load in several Queries
(several index lookups + physical IOs)

1/ Select car … where id=?
2/ select engine… where car_id=?
3/ select wheel .. where car_ id=?
…

Or LEFT JOIN

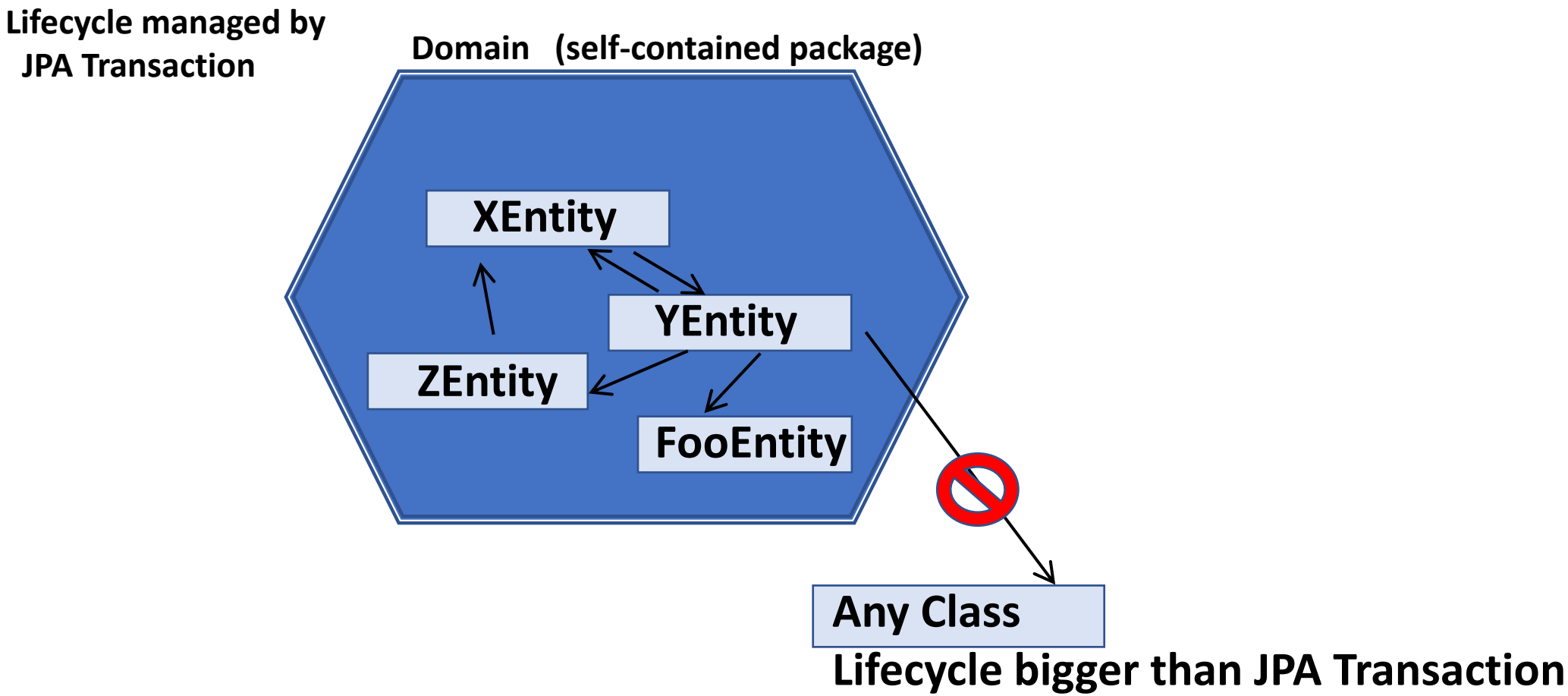
Relations (Graph) between Entity



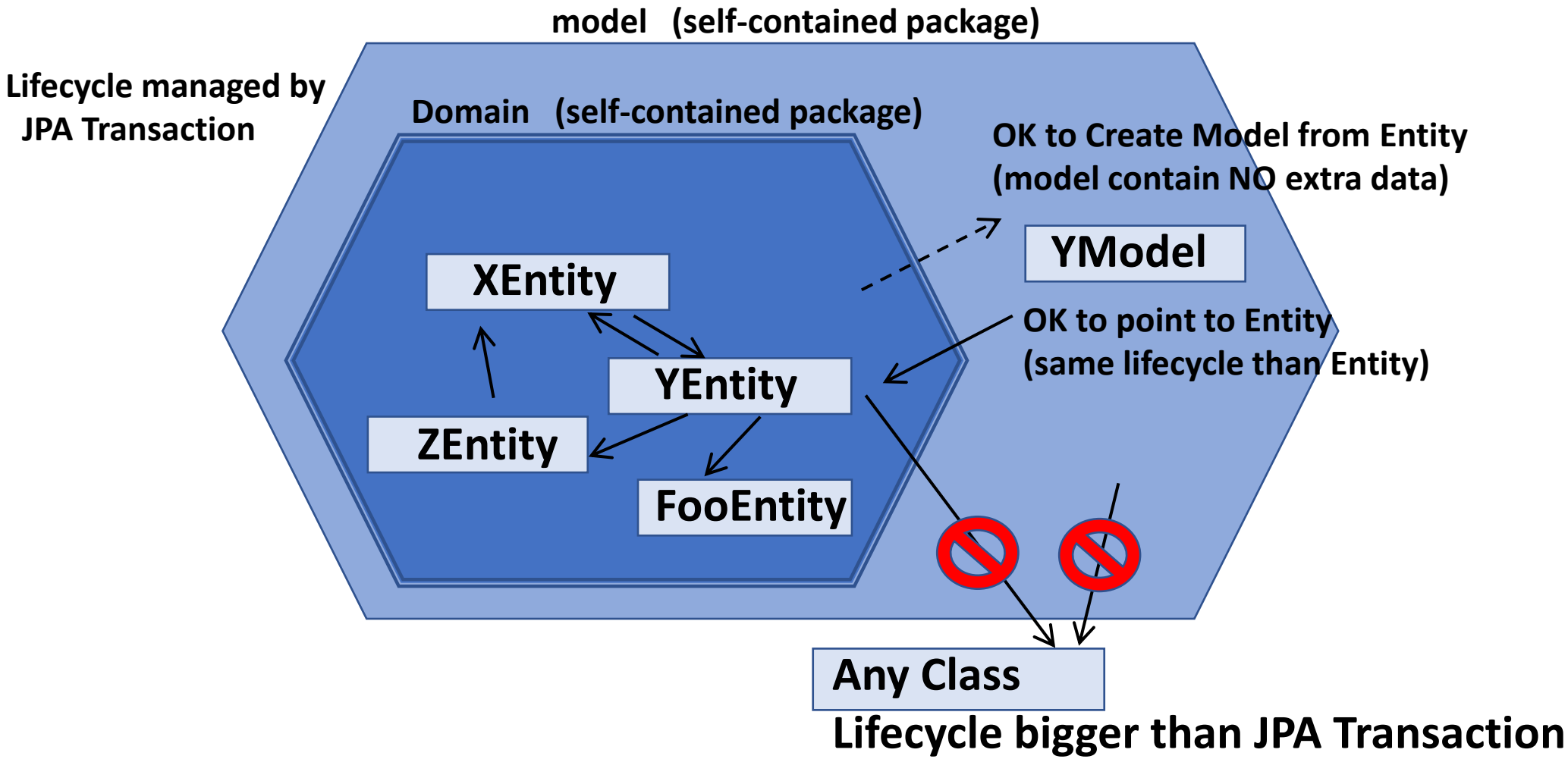
Relations: maybe cyclic / lazy loaded:

in-memory pointers for @ManyToOne
+
List<> pointers for @OneToMany

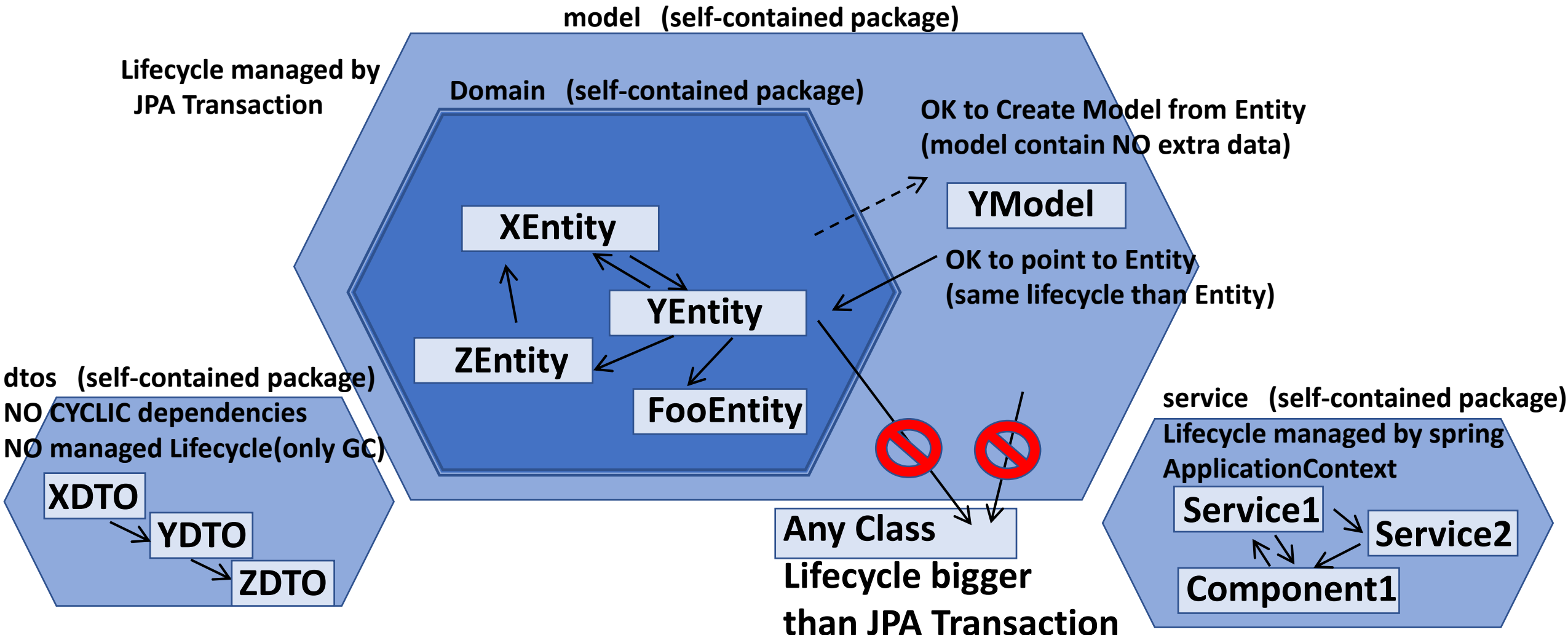
Relations (Graph) between Entity restrict to kernel « domain »



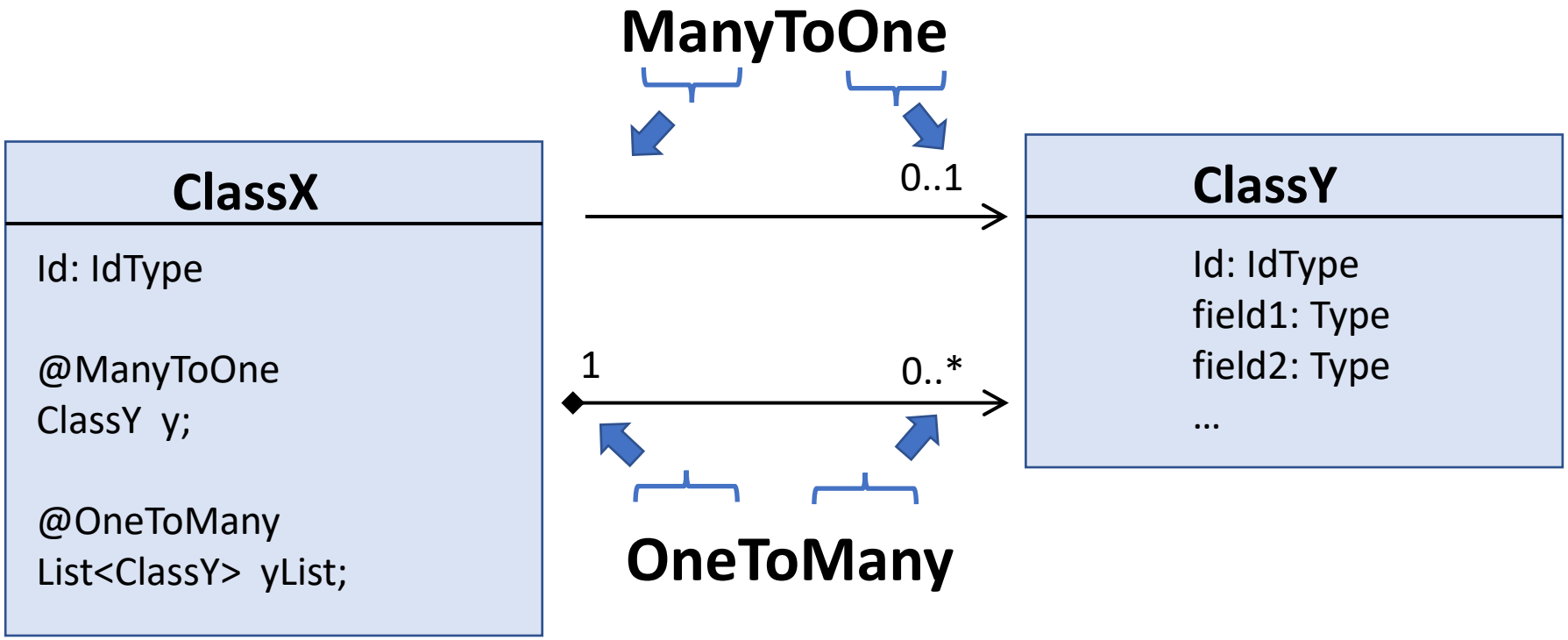
Entity – Model … same managed Lyfecycle



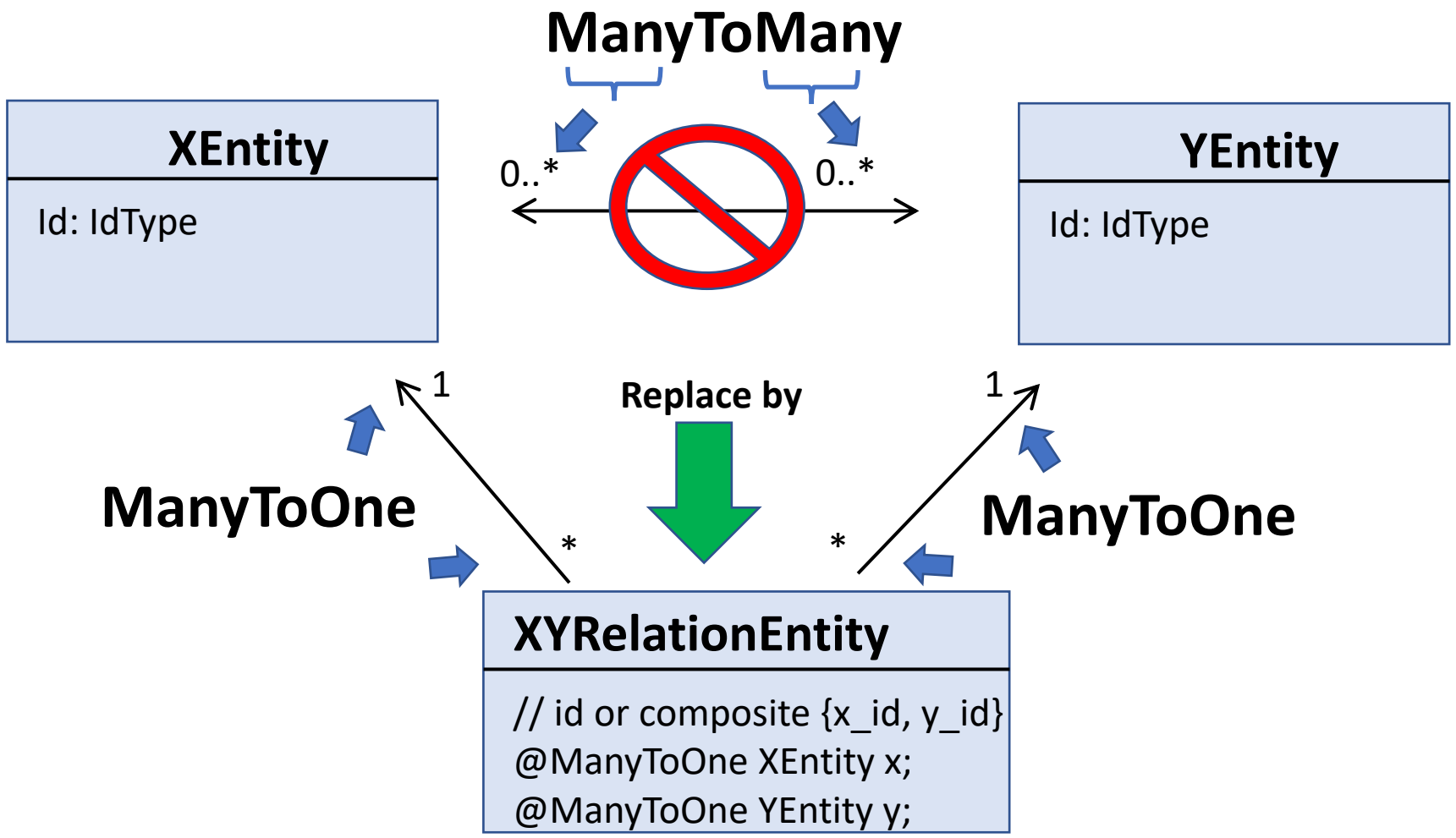
DTO - Entity – Model – Service
… different managed Lyfecycle



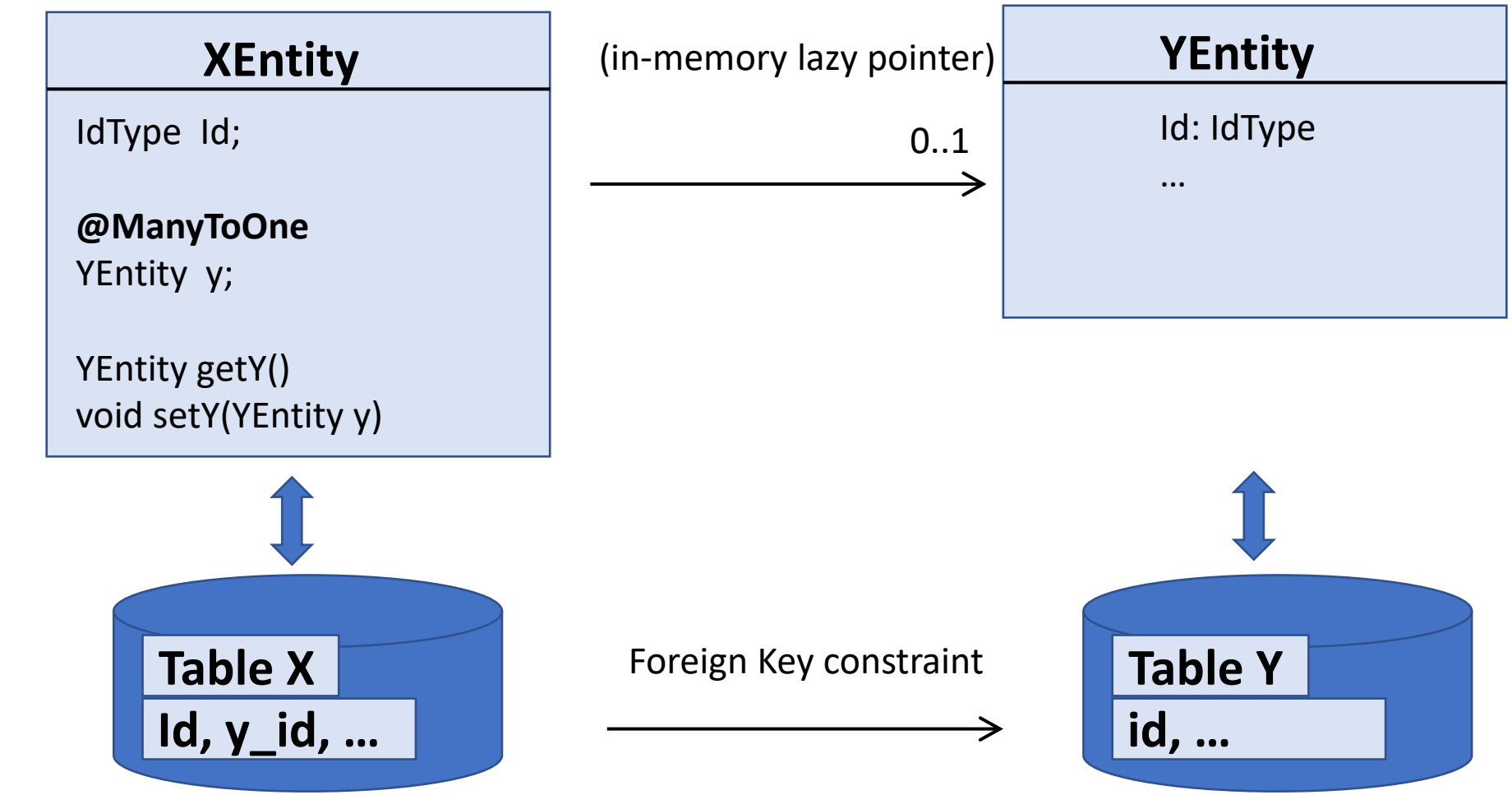
Relations Cardinality
OneToMany & ManyToOne



Split (materialize) ManyToMany relations



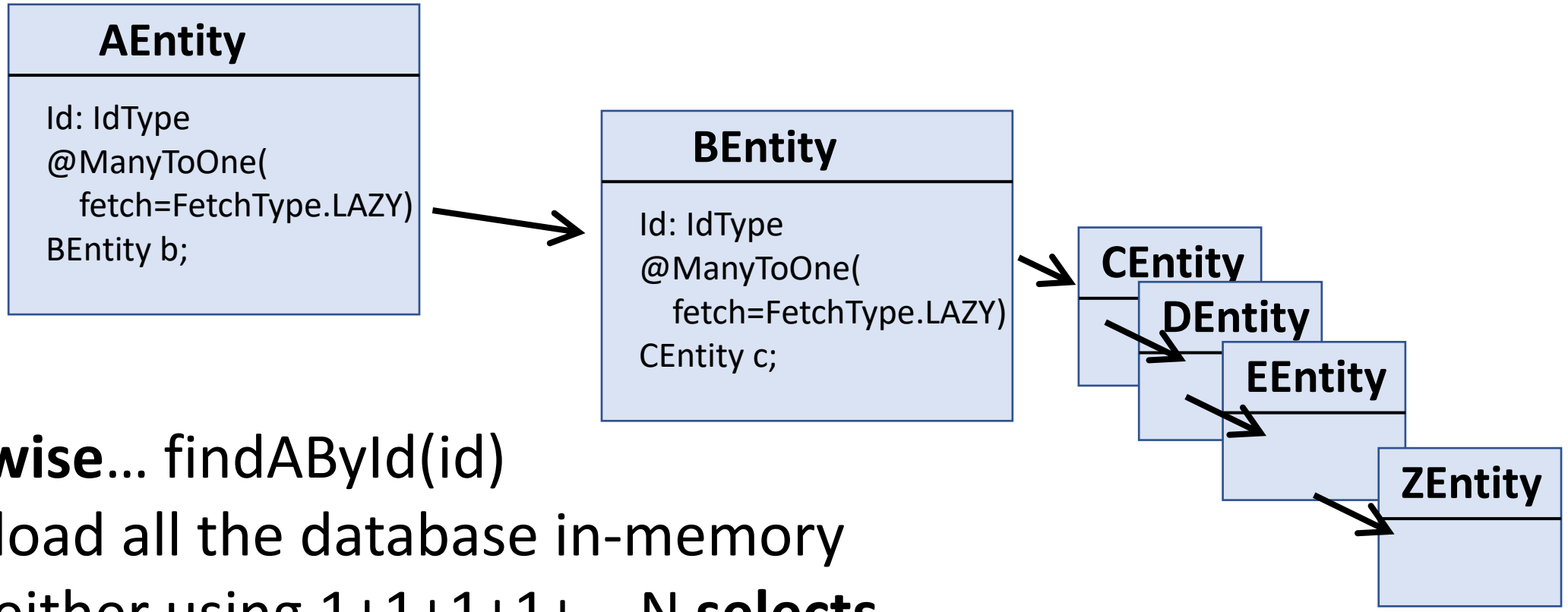
ManyToOne = pointer/reference/ Foreign Key



column « Y_ID »: Foreign Key to « Y.ID »
can be nullable, but valid y when set

Column « ID » : Primary Key
(+ unique index « PK_ID »)

ManyToOne => always(?) use non default fetch = FetchType.LAZY



Otherwise… findAById(id)

=> load all the database in-memory

=> either using 1+1+1+1+… N **selects**

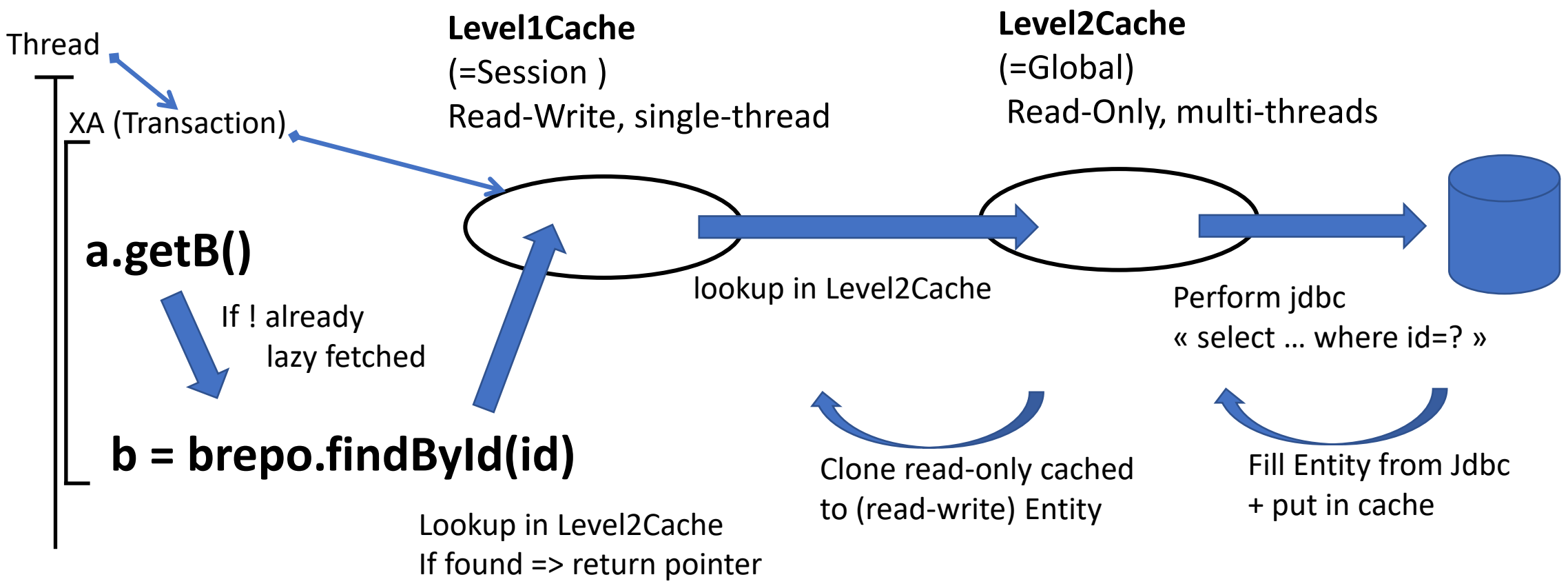
or using 1+1+1+…N **LEFT JOIN** + deduplications

Lazy Loading: load only ONCE, on first access (instrumented code)

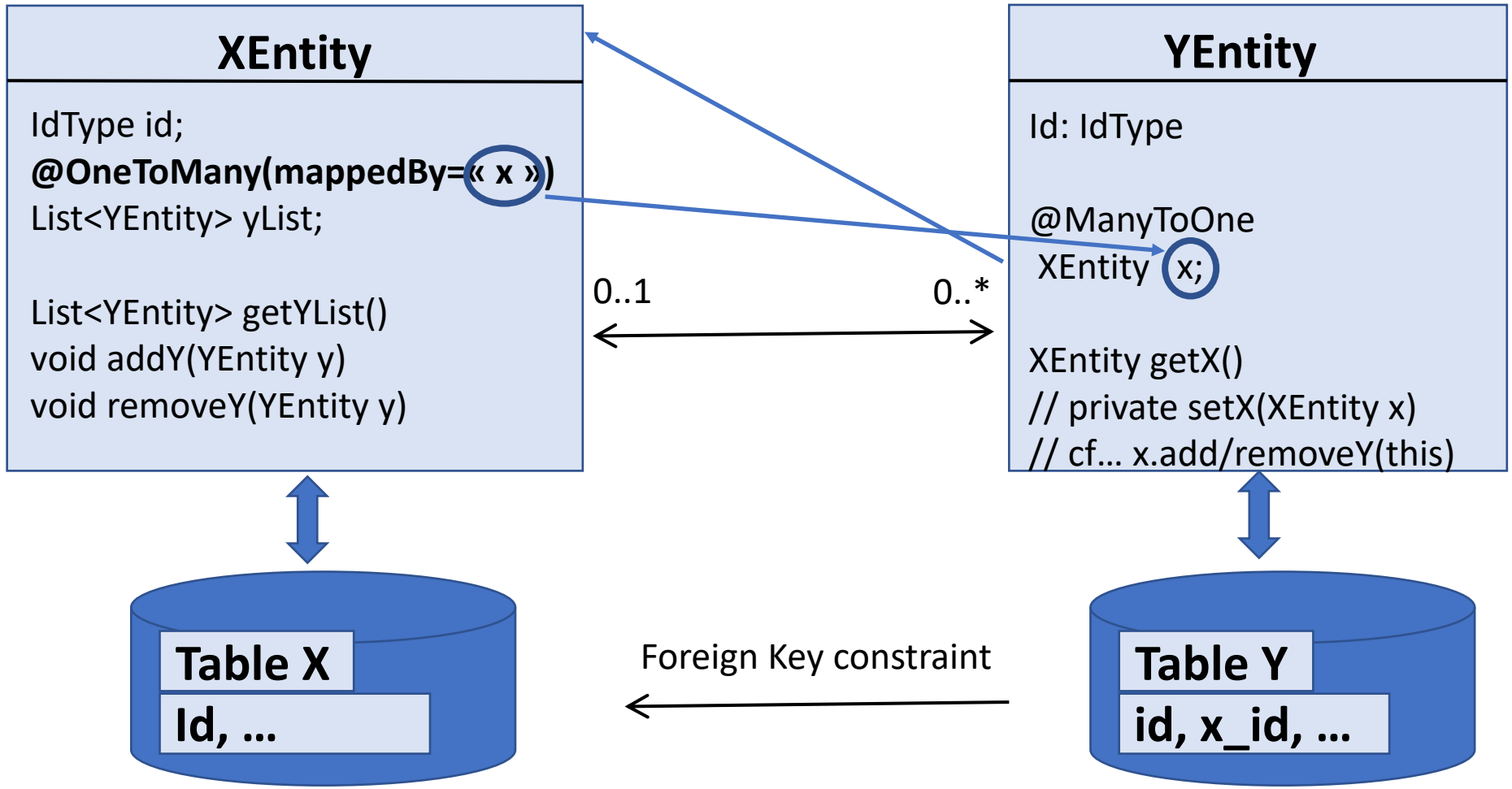
```
@ManytoOne( fetch = FetchType.LAZY )
private B Bentity b;

// code you write
// standard @Getter + @Setter

// != real code INSTRUMENTED by JPA
private Long $bId;  private EntityManager $em;
public BEntity getB() {
   if (this.b == null && this.$bId != null) {
      this.b = this.$em.findById(BEntity.class, this.$bId);
   }
   return this.b;
}
```

GENERATED CODE

findById()
remote « select » calls … only if caches 1+2 miss



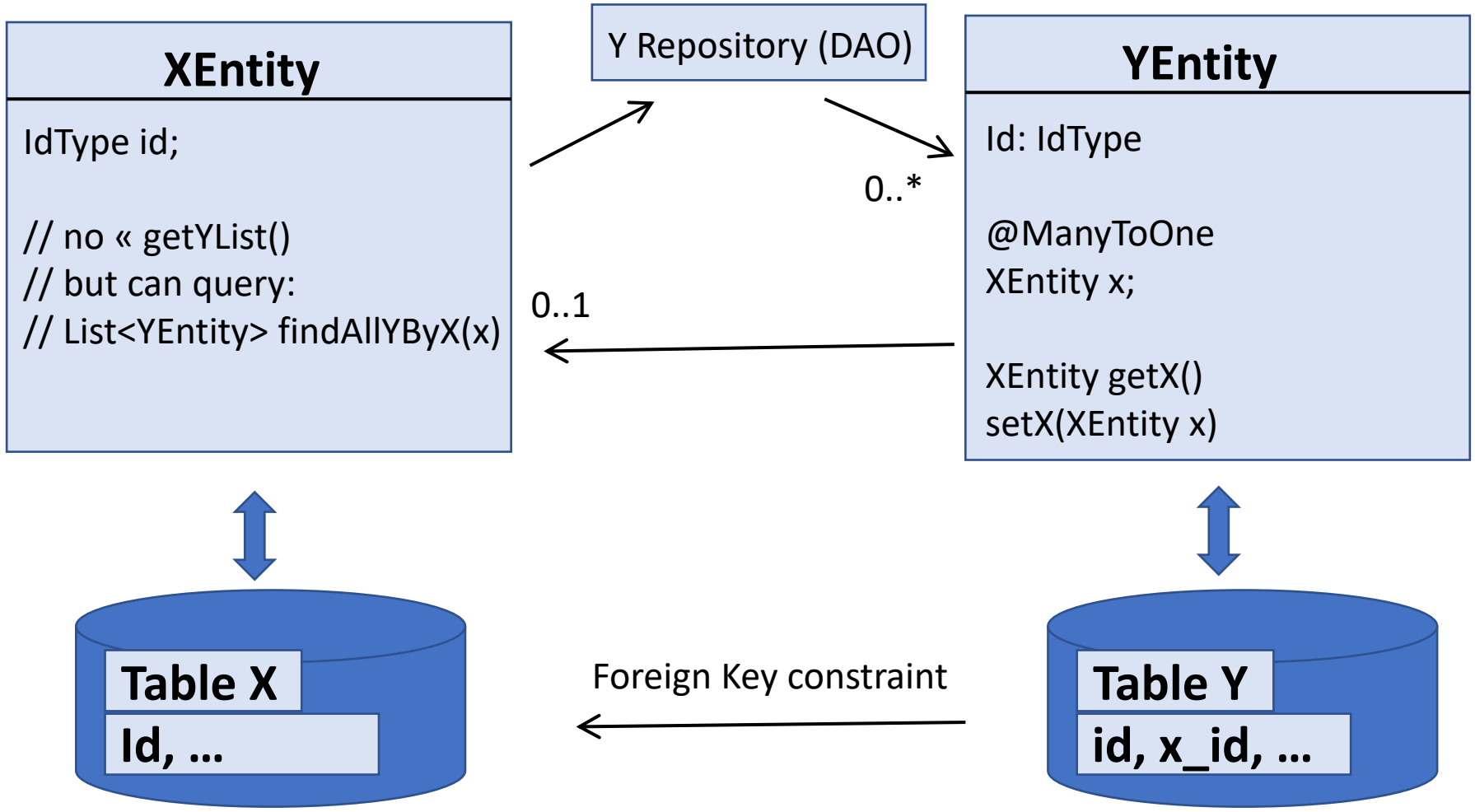
OneToMany : List with mappedBy relation



Column « ID »: Primary Key
Standard table
… no constraint relation with « Y »

Column « ID » : Primary Key (+ unique index « PK_ID »)
Column « X_ID »: Foreign Key to « X.ID » (nullable)

Unmapped « Hidden » OneToMany
.. Same DB, different JPA mapping



Column « ID »: Primary Key
Standard table
… no constraint relation with « Y »

Column « ID » : Primary Key (+ unique index « PK_ID »)
Column « X_ID »: Foreign Key to « X.ID » (nullable)

Owner.List <-> Element.owner
redundant data to be consistent

```
addY(YEntity y) {
  if (y.getXOwner() != null)
    throw new Ex (« already in list »);
  this.yList.add(y);
  y._inv_setXOwner(this);
}
removeY(YEntity y) {
  if (y.getXOwner() != this)
    throw new Ex (« not in list »);
  this.yList.remove(y);
  y._inv_setXOwner(null);
}
```

a « Y » object can only be in 0..1 « X list »

y._inv_setXOwner(x)
… should not be called directly

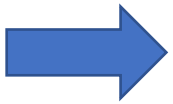
(redundant with x.addY(this))

Caching Invalidations for Mapped @OneToMany

x.addY(y);

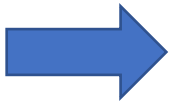
2 side-effects:

Entity « x » is modified
: field « yList » contains 1 more element
(if already lazy Loaded)



… on commit,
must broascast change
for invalidating cached « X »
Or change in-memory « X »

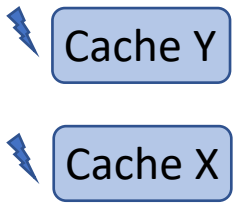
Entity « y » is modified
: field « xOwner » is changed null -> « x »
(if already lazy Loaded)



… on commit,
must broascast change
for invalidating cached « Y »
Or change in-memory « Y »

Caching Invalidation Difference between mapped @OneToMany / inverse @ManyToOne

@ManyToOne

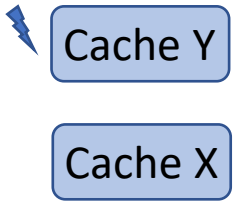


when changing Y
Need to invalidate both caches
« X » and « Y »

cache
- X
- X.yList (in X)
- Y

Can use « x.getYList() » method
.. So caching for get « Y list »

Inverse @OneToMany

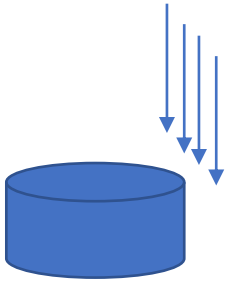


when changing « Y »
… referential data « X » do not change
=> Can cache « X », no need invalidation

No « x.getYList() » method

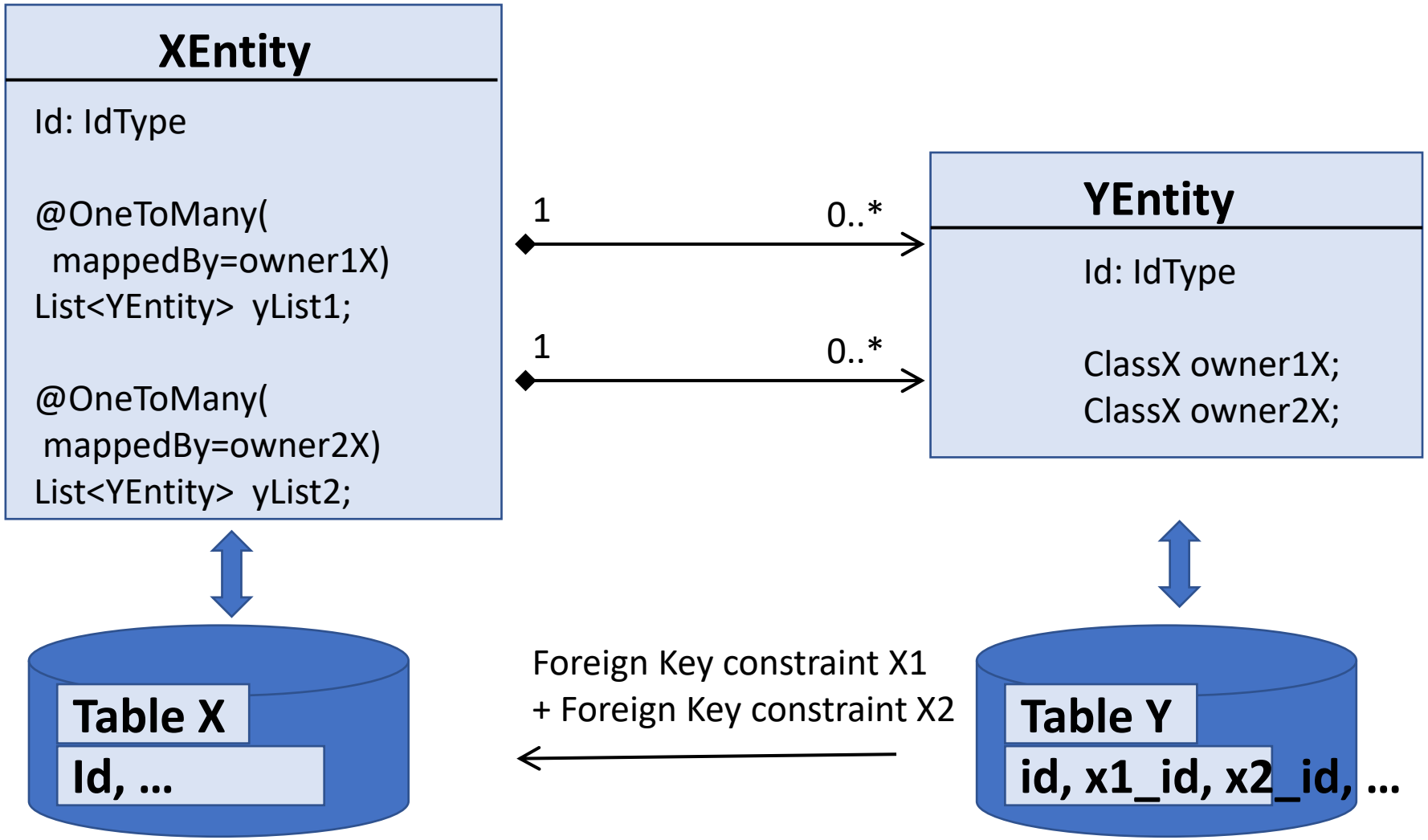
cache
- X (no yList)
- Y

.. So no field caching for get « Y list »
Need to perform many queries
« dao.findAllYByX(x) »



REPEAT sql
select y .. xhere x_id=?

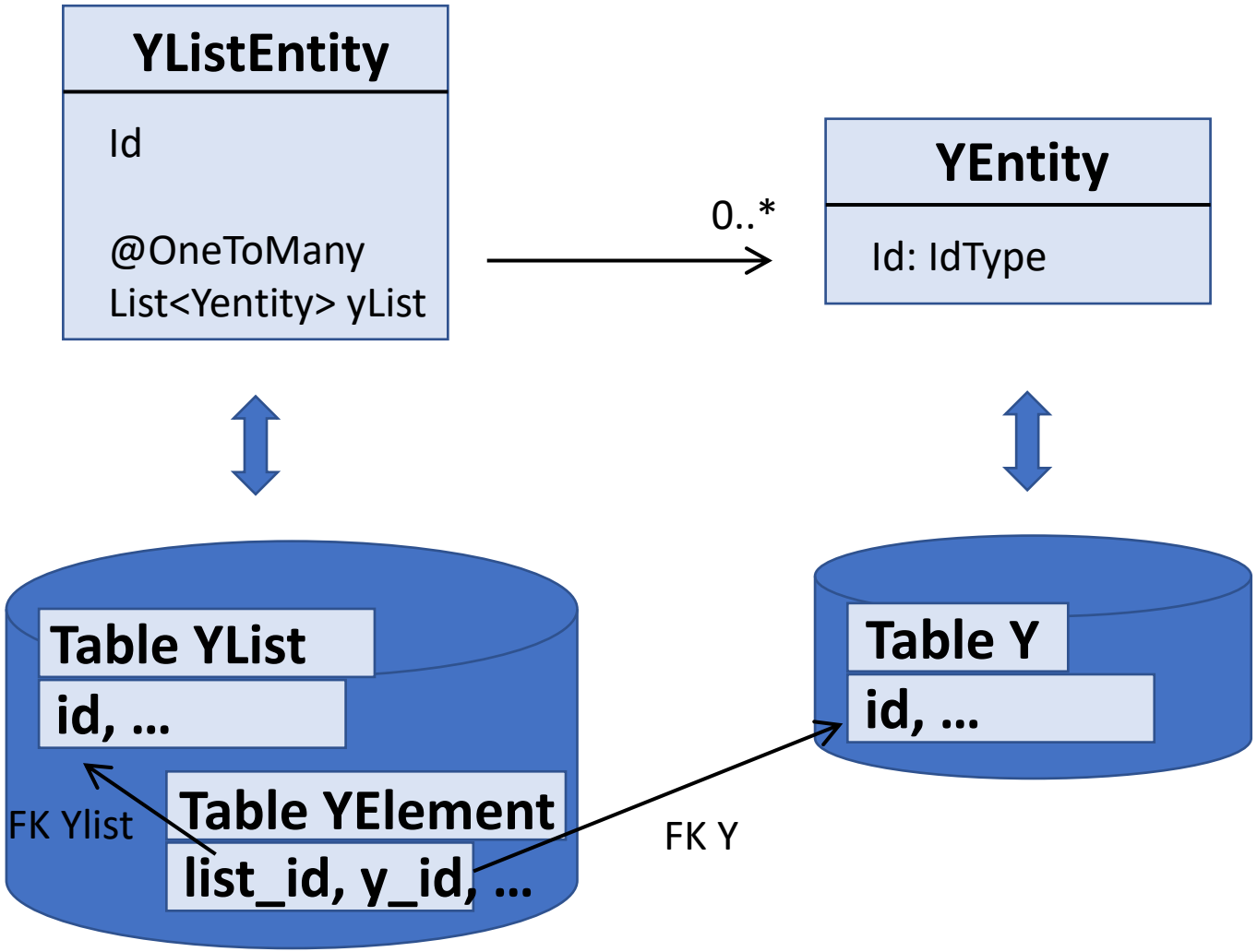
Multiple (Fixed) List1,List2 for same Entities ?



Column « ID »: Primary Key
Standard table
… no constraint relation with « Y »

Column « ID » : Primary Key (+ unique index « PK_ID »)
Column « X1_ID »: Foreign Key to « X.ID » (nullable)
Column « X2_ID »: Foreign Key to « X.ID » (nullable)

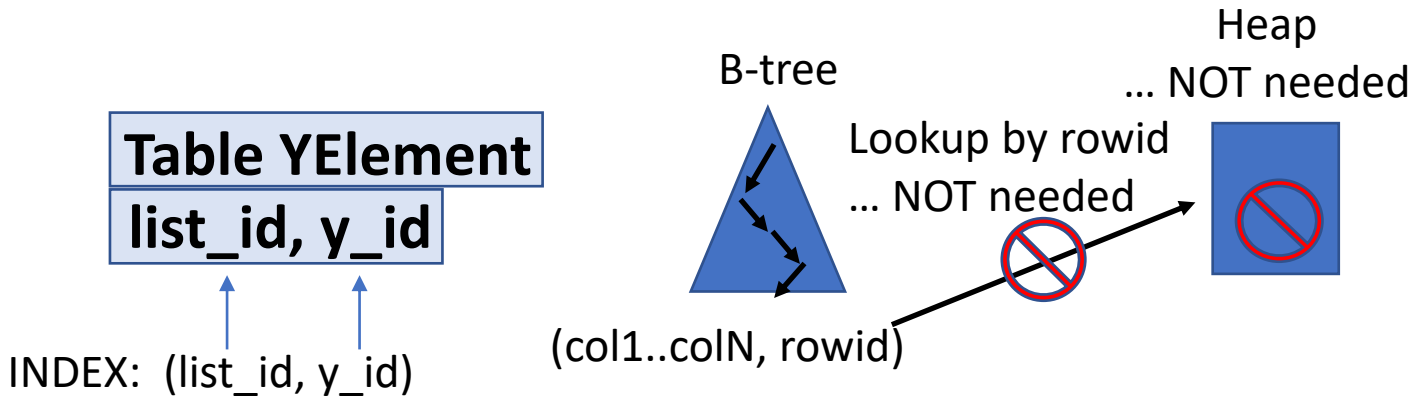
« List » Entity


YListEntity
Id
@OneToMany
List<Yentity> yList
0..*
YEntity
Id: IdType
Table YList
id, …
Table YElement
list_id, y_id, …
FK Ylist
FK Y
Table Y
id, …

Notice.. IOT Tables for « list elements »

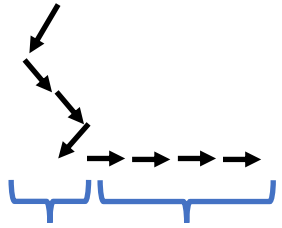
IOT = Indexed Organized Table

Contains only a single B-Tree Index,
but no Head allocation
(… all columns contained in index)



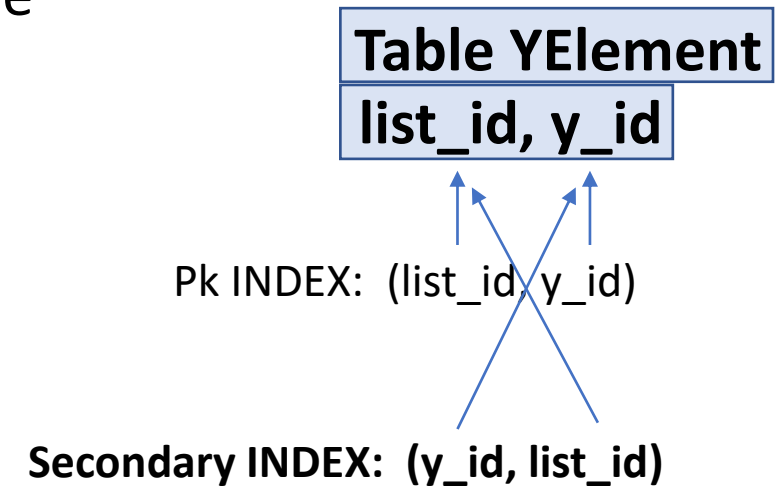
IOT Table: take ~1 / 2 of disk space, perform ~2x faster

FAST query: « select y_id from ylist_element where list_id=? »

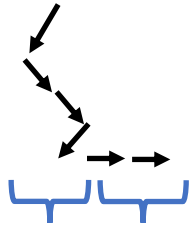


Notice.. Reverse INDEX for fast lookup « Y » are in « lists » : memberships

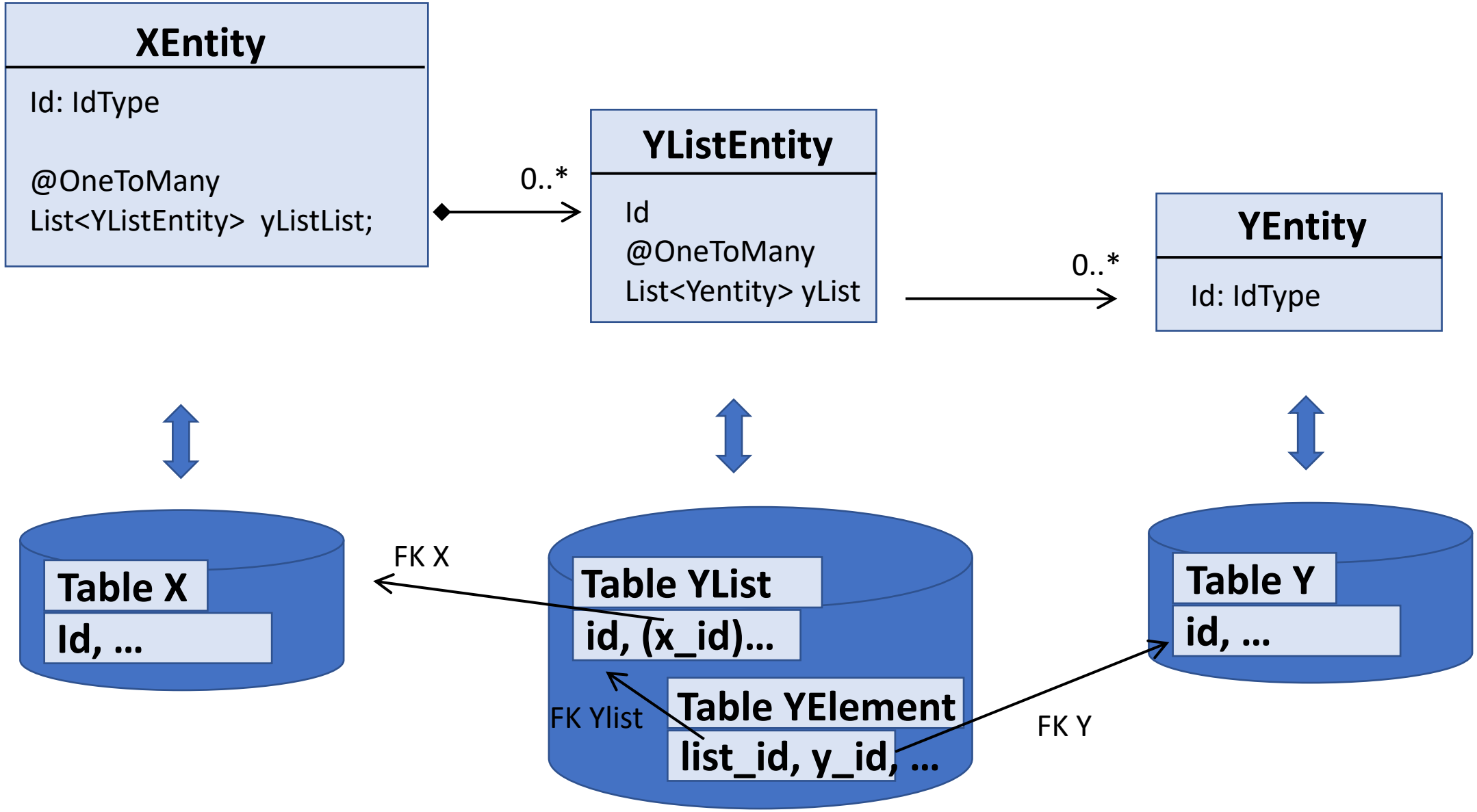
IOT = Indexed Organized Table
+ extra index..



FAST query: « select list_id from ylist_element where y_id=? »



Multiple (Dynamic) Lists for same Entities ?



Mapping for List
allow Duplicates? preserve Order? Composite PK?

Table YElement

```
// composite embedded @Id
//  .. See next
long list_id;
Set<Yentity> y;
```

⟷ **list_id, y_id**

No duplicate (Set instead of List)
No order
PK is composite {list_id,y_id}

```
@Id @GenereatedValue
long id; // useless in DB !!
long list_id;
Set<Yentity> y;
```

⟷ **Id, list_id, y_id**

Idem… but extra useless « technical » ID
No duplicate (Set instead of List)
No order

```
long list_id;
List<Yentity> y;
```

⟷ **list_id, order, y_id**

List with order .. Possibly duplicates
PK is composite {list_id,order}

Importance of correct Database Model

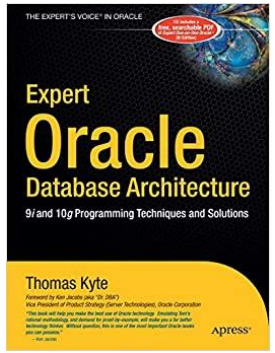
Database performs well when early deployed to PROD
(contains ~100 rows)

Doing « FULL SCAN » of few rows is OK, database fit in-memory

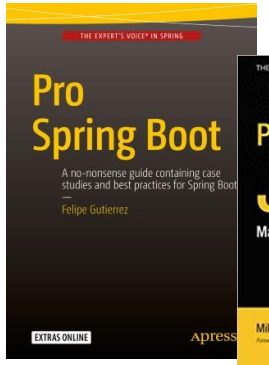
Problems arise late when >= 100 000 000 rows

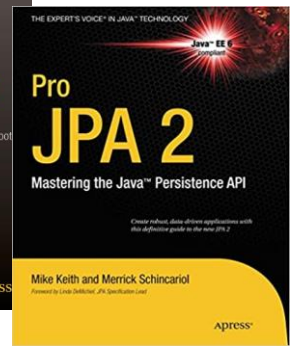
Good Architecture =

Optimized Database Architecture



AND Correct JPA Mapping





AND Correct Config (Cache/Invalidation)

AND Correct Code

Part 1 : Entity (Domain UML, Database, JPA mapping)

Next parts

Part 2 : extra Model class for Entity ?

Part 3 : DTO classes for Entity

Part 4 : Rest Controller, GraphQL, Service