Databases 2

Exam on July 1 2021

Solutions

Exercise A. Active Databases

- A sports competition database records the performances of athletes in national running competitions. Primary keys are represented with capital letters.
  - `ATHLETE (ATHL_ID, DISCIPLINE, name, personal_record, country_code, qualified)`
  - `COMPETITION (COMP_ID, discipline, number_of_enrolled, winner)`
  - `RESULT (COMP_ID, ATHL_ID, time)`
  - `COUNTRY (COUNTRY_CODE, DISCIPLINE, national_record)`
- At the end of a competition, an application inserts all the results of the athletes. Some triggers react to the insertion of the results of a competition, by:
  - a) Setting the winner of the competition;
  - b) Updating the personal record if some athlete has improved his best time for the discipline;
  - c) Updating the national record if some athlete has improved the best time for the discipline of all the athletes of the country;
  - d) Updating the qualified attribute of an athlete to true if the athlete has set the national record or has won more than 10 competitions.

Solution of exercise A

```
-- a)

create trigger UpdateWinner
After insert on result
For each row
WHEN new.time = (SELECT min(time) FROM result
     WHERE COMP_ID=new.COMP_ID)
BEGIN

   UPDATE competition
   SET winner=new.ATHL_ID
   WHERE COMP_ID=new.COMP_ID;

END
```

Solution of exercise A

```
-- b)

create trigger UpdatePersonalRecord
After insent on result
For each row
-- personal record must be for same discipline of the athlete

-- check if time is less than personal record or personal record is null

WHEN new.time < (SELECT personal_record FROM athlete WHERE
ATHL_ID=new.ATHL_ID AND DISCIPLINE = (SELECT discipline FROM competition
WHERE COMP_ID = new.COMP_ID))

OR (SELECT personal_record FROM athlete WHERE ATHL_ID=new.ATHL_ID AND
DISCIPLINE = (SELECT discipline FROM competition WHERE COMP_ID =
new.COMP_ID)) IS NULL

BEGIN

   UPDATE athlete

   SET personal_record=new.time

   WHERE ATHL_ID=new.ATHL_ID AND DISCIPLINE=(SELECT discipline FROM
competition WHERE COMP_ID = new.COMP_ID);

END
```

Solution of exercise A

```
-- c)  + d) first condition

create trigger UpdateNationalRecord
After insent on result
For each row
-- national record must be for same country and same discipline of the athlete
-- check if time is less than national record or national record is null

WHEN new.time < (SELECT national_record FROM country as c join athlete as a on c.COUNTRY_CODE
= a.country_code AND c.DISCIPLINE = a.DISCIPLINE  WHERE ATHL_ID = new.ATHL_ID AND a.DISCIPLINE
= (SELECT discipline FROM competition WHERE COMP_ID = new.COMP_ID))
OR (SELECT national_record FROM country as c join athlete as a on c.COUNTRY_CODE =
a.country_code AND c.DISCIPLINE = a.DISCIPLINE  WHERE ATHL_ID = new.ATHL_ID AND a.DISCIPLINE =
(SELECT discipline FROM competition WHERE COMP_ID = new.COMP_ID)) IS NULL

BEGIN
   DECLARE current_discipline VARCHAR(45);
   SELECT discipline INTO current_discipline FROM competition WHERE COMP_ID = new.COMP_ID;

   UPDATE country
   SET national_record=new.time
   WHERE COUNTRY_CODE = (SELECT country_code FROM athlete WHERE ATHL_ID=new.ATHL_ID LIMIT 1)
AND DISCIPLINE = current_discipline;

   UPDATE athlete
   SET qualified=true
   WHERE ATHL_ID=new.ATHL_ID AND DISCIPLINE = current_discipline;
END
```

Solution of exercise A

```
-- d) second condition

crate trigger updateQualifications
after insert on result
for each row
-- results for all athletes enrolled for a competition must be inserted before checking
qualification
WHEN (SELECT number_of_enrolled FROM competition WHERE COMP_ID = new.COMP_ID) = (SELECT
count(*) FROM result WHERE COMP_ID = new.COMP_ID)
BEGIN
   -- check only the competitions where the winner and discipline are the same and all results
have been inserted
   IF 11 = (SELECT count(*) FROM competition as c where (winner, discipline) = (select winner,
discipline from competition where COMP_ID = new.COMP_ID) and number_of_enrolled = (SELECT
count(*) FROM result WHERE COMP_ID = c.COMP_ID)) THEN
      UPDATE athlete
      SET qualified=true
      WHERE  (ATHL_ID, DISCIPLINE) = (select winner, discipline from competition where COMP_ID
= new.COMP_ID);
   END IF;
END
```

Exercise B Concurrency

Classify the following three schedules. Notice that the point where they commit is explicitly reported.

1 r2(x) w2(x) r1(y) C2 r1(x) w1(x)

2 r2(y) w2(y) r1(y) w1(y) C2 C1

3 r1(y) r2(y) w2(y) r2(x) w2(x) C2 r1(x) C1

a) Build a table like the one below that summarizes the results. Fill it in just with Yes/No.

	VSR	CSR	2PL	Strict-2PL	TS Multiversion
1					
2					
3					

b) Then, briefly motivate all the “No” (one or at maximum 2 lines of explanations for each “No”).

Exercise B Concurrency

	VSR	CSR	2PL	Strict-2PL	TS Multiversion
1	Yes	Yes	Yes	Yes	No
2	Yes	Yes	Yes	No	No
3	No	No	No	No	Yes

Each no had to be briefly motivated – in the next slides we report the complete solutions

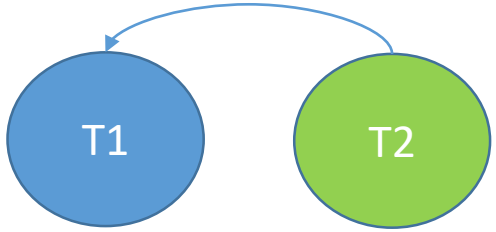
VSR

- Two schedules are view-equivalent if they have the same operations, the same reads-from relation, and the same final writes. A schedule is view-serializable if it is view-equivalent to a serial schedule of the same transactions
- r2(x) w2(x) r1(y) C2 r1(x) w1(x)
  - T1, T2 no r1(x) reads x from w2(x)
  - T2, T1 yes same as r2(x) w2(x) C2 r1(y) r1(x) w1(x)
- r2(y) w2(y) r1(y) w1(y) C2 C1
  - T1, T2: no r1(y) reads from w2(y) and final write by T1
  - T2, T1: writes OK, read from OK if read1(y) reads y written **but not commited** by w2(y)
- r1(y) r2(y) w2(y) r2(x) w2(x) C2 r1(x) C1
  - T1,T2 no r1(x) reads from w2(x)
  - T2,T1, no r1(y) does not read from w2(y)

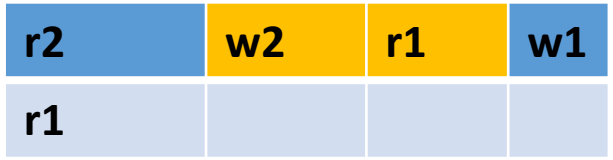
CSR

1 r2(x) w2(x) r1(y) C2 r1(x) w1(x)
2 r2(y) w2(y) r1(y) w1(y) C2 C1
3 r1(y) r2(y) w2(y) r2(x) w2(x) C2 r1(x) C1

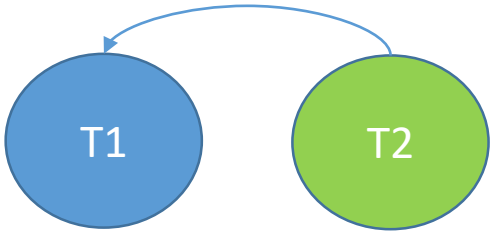
1: OK



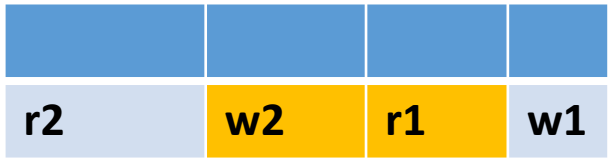
X
Y



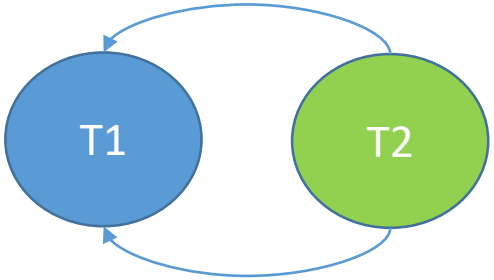
2: OK



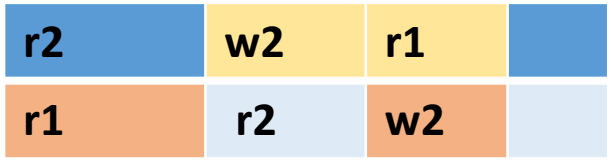
X
Y



3: KO



X
Y



2PL

1 r2(x) w2(x) r1(y) C2 r1(x) w1(x)
2 r2(y) w2(y) r1(y) w1(y) C2 C1
3 r1(y) r2(y) w2(y) r2(x) w2(x) C2 r1(x) C1

1	2	3	4	5	6	7	8	9	10
r2-lock(x) ↗ granted	w2-lock(x) ↗ (upgrade)	r1-lock(y) ↗ granted	**C2** r2-lock(x) release ↘ w2-lock(x) release ↘	r1-lock(x) ↗ granted	w1-lock(x) ↗ **upgrade**	C1 T1 Release all locks			2PL 2PL strict
r2-lock(y) ↗ granted	w2-lock(y) ↗ (upgrade)	w2-lock(y) (downgrade) ↘	r1-lock(y) ↗ granted	r2-lock(y) released ↘	w1-lock(y) ↗ upgrade	**C2** **C1**			2PL Not 2PL strict
r1-lock(y) ↗ granted	r2-lock(y) ↗ granted	r1-lock(y) release ↘	w2-lock(y) ↗ (upgrade)	r2-lock(x) ↗ granted	w2-lock(x) ↗ (upgrade)	**C2** Release all locks	r1-lock(x) ↗ granted		Not 2PL nor 2PL strict

TS-multi

1 r2(x) w2(x) r1(y) C2 r1(x) w1(x)
2 r2(y) w2(y) r1(y) w1(y) C2 C1
3 r1(y) r2(y) w2(y) r2(x) w2(x) C2 r1(x) C1

Request	Response	New Values
r2(x)	OK	RTM(x)=2
w2(x)	OK	WTS(x)=2
r1(y) -C2	OK	Multiversion accepts all reads
r1(x)	OK	Multiversion accepts all reads
w1(x)	KO	1 < RTM(x)
r2(y)	OK	RTS(y) = 2
w2(y)	OK	WTS(y) = 2
r1(y)	OK	Multiversion accepts all reads
w1(y) C2 C1	KO	1 < RTM(x)
r1(y)	OK	
r2(y)	OK	RTM(y) = 2
w2(y)	OK	WTM(y)=2
r2(x)	OK	RTM(x)=2
w2(x) C2	OK	WTM(x)=2
r1(x) C1	OK	

Exercise C. Physical DB

Consider again tables COMPETITION, ATHLETE and RESULT from Ex. A.

Table COMPETITION stores 1000 tuples in a B+ primary storage built on attribute Discipline, with 3 levels and 100 leaf nodes.

Table ATHLETE stores 100K tuples in 5K blocks in a hash primary storage with the hash function defined on ATHL_ID. The average cost for a lookup is 1.2 i/o ops.

Table RESULT stores 1M tuples in 100K blocks in a primary entry-sequenced sequential structure.

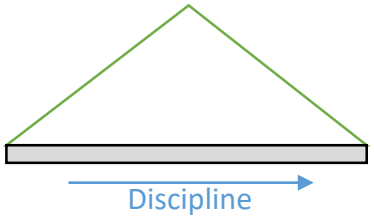
On table RESULT there are a secondary B +-tree index built on attribute COMP_ID, with three levels and 2.5k leaf nodes and a hash-index built on attribute ATHL_ID with (the same function as that of the primary hash on Athlete and) negligible overflow chains.

Consider the query that extracts the athletes who won the 5000m with a time that is less than 13 minutes. Describe a reasonably efficient plan (and estimate its cost) in the above scenario, knowing that val(Discipline)=50.

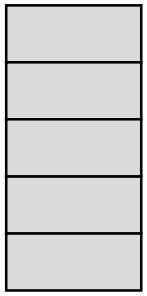
```
select *
from ATHLETE
where Discipline = "5000m" and
        (ATHL_ID, Discipline) in
            ( select ATHL_ID, Discipline
              from COMPETITION join RESULT on
                      ATHL_ID = Winner
              where Time < "13:00.00" )
```

ATHLETE (ATHL_ID, DISCIPLINE, Name, Personal_record, Country_code, Qualified)
COMPETITION (COMP_ID, Discipline, Number_of_enrolled, Winner)
RESULT (COMP_ID, ATHL_ID, Time)

```
select *
from ATHLETE
where Discipline = "5000m" and
        (ATHL_ID, Discipline) in
            ( select ATHL_ID, Discipline
              from COMPETITION join RESULT on  ATHL_ID = Winner
              where Time < "13:00.00" )
```

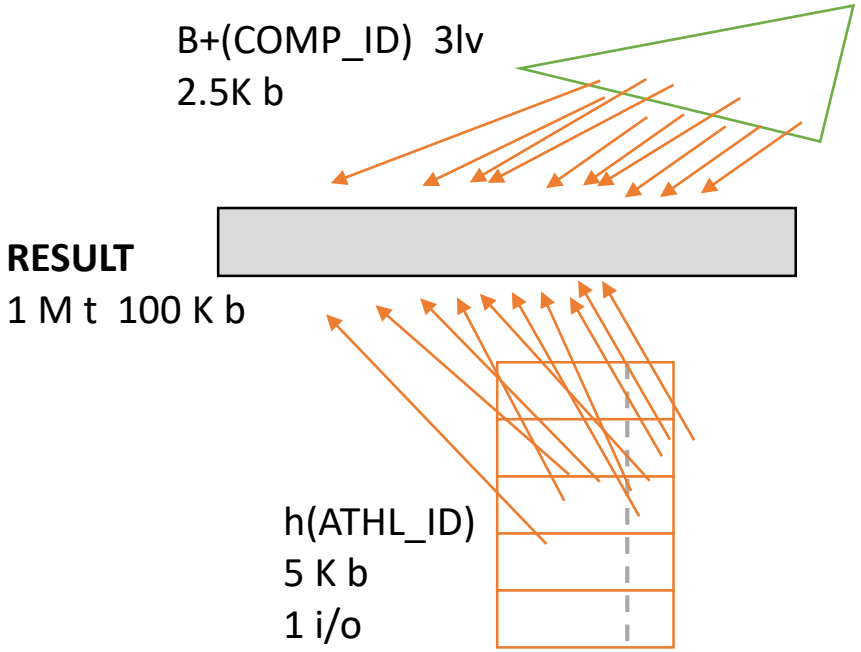


COMPETITION
100 b
1 K t



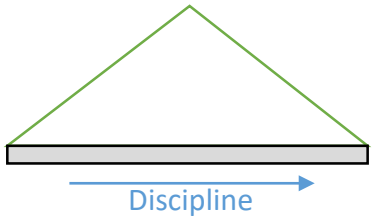
ATHLETE
5 K b
100 K t

h(ATHL_ID)
1.2 i/o



RESULT
1 M t 100 K b

B+(Discipline) 3lv



COMPETITION
100 b
1 K t

Given the discipline *5000m*, in this B+ you can get all the competions by navigating 2 intermediate levels and with a further access the first leaf node containing a *5000m* competition.

How many leaf nodes are needed to store all the *5000m* competitions?
How many *5000m* competitions are stored in the table?

We know that val(Discipline) = 50, which means that there are 50 different disciplines in the table. Since it stores 1000 tuples, there will be in average 1000/50=20 competition tuples for each discipline (2% of the tuples)

How many blocks are needed to store 20 tuples?

From the data we know that 100 blocks contain 1000 tuples, therefore each block contains in average 1000/100=10 tuples

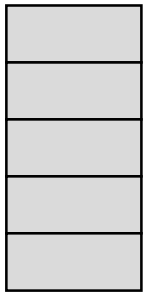
To store 20 tuples, we need exactly 2 blocks. But the probability that a discipline starts exactly at the beginning of the block is very low. It is more probable that the first tuple of a discipline starts at some point in the block. For this reason, in average, to get all the competitions of a discipline, we need to read 3 blocks.

The cost to access all the *5000m* competions is:

2(intermediate levels) + 3 (blocks)

→ If needed, these 3 blocks can be cached

Depending on the plan you may compute also these data:



ATHLETE
5 K b
100 K t

h(ATHL_ID)
1.2 i/o

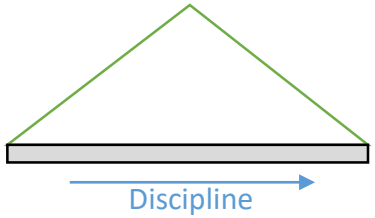
How many athletes satisfy this part of the query?

```
select *
from ATHLETE
where Discipline = "5000m“
```

Since val(Discipline)=50, which means that there are 50 different disciplines, we can assume that the athletes are uniformely distributed over the different disciplines and therefore that there are

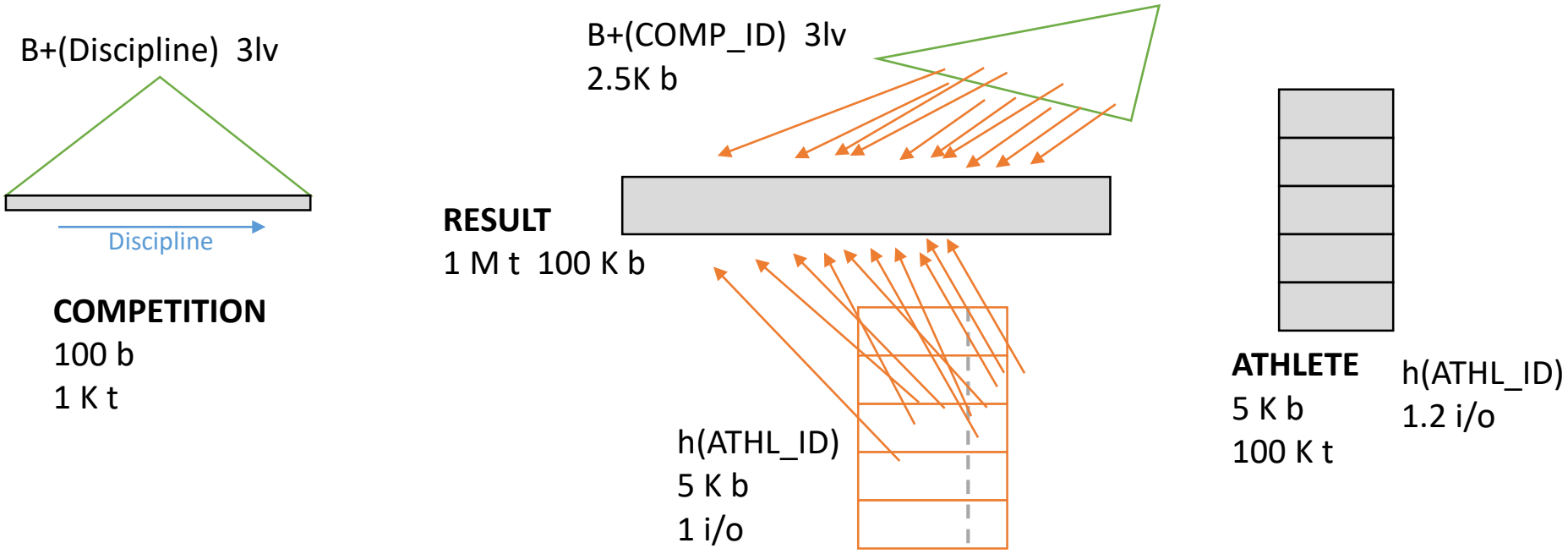
100K / 50 = 2000 athletes (2% of the total)
with discipline *5000m*

B+(Discipline) 3lv



COMPETITION
100 b
1 K t

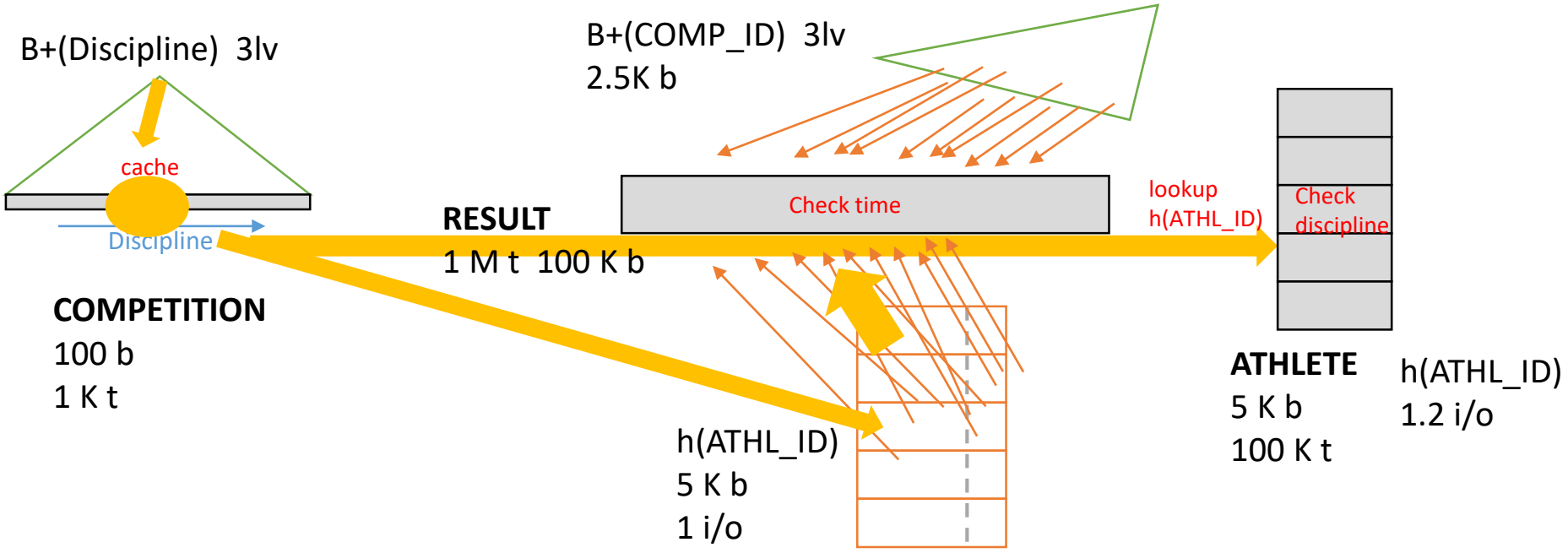
How many winners do we have in COMPETITION?
In the previous slide we computed the number of competitions: 20 → there will be 20 winners



We need to join RESULT with
a) COMPETITION on ATHL_ID = Winner and
b) ATHLETE on ATHL_ID

a) Can be done exploting the 3 blocks of COMPETITION identified in slide 15: we just need to scan the RESULT table (through the hash table since we have the Winner ID)
b) We can exploit the two hash structures defined with the same hash function

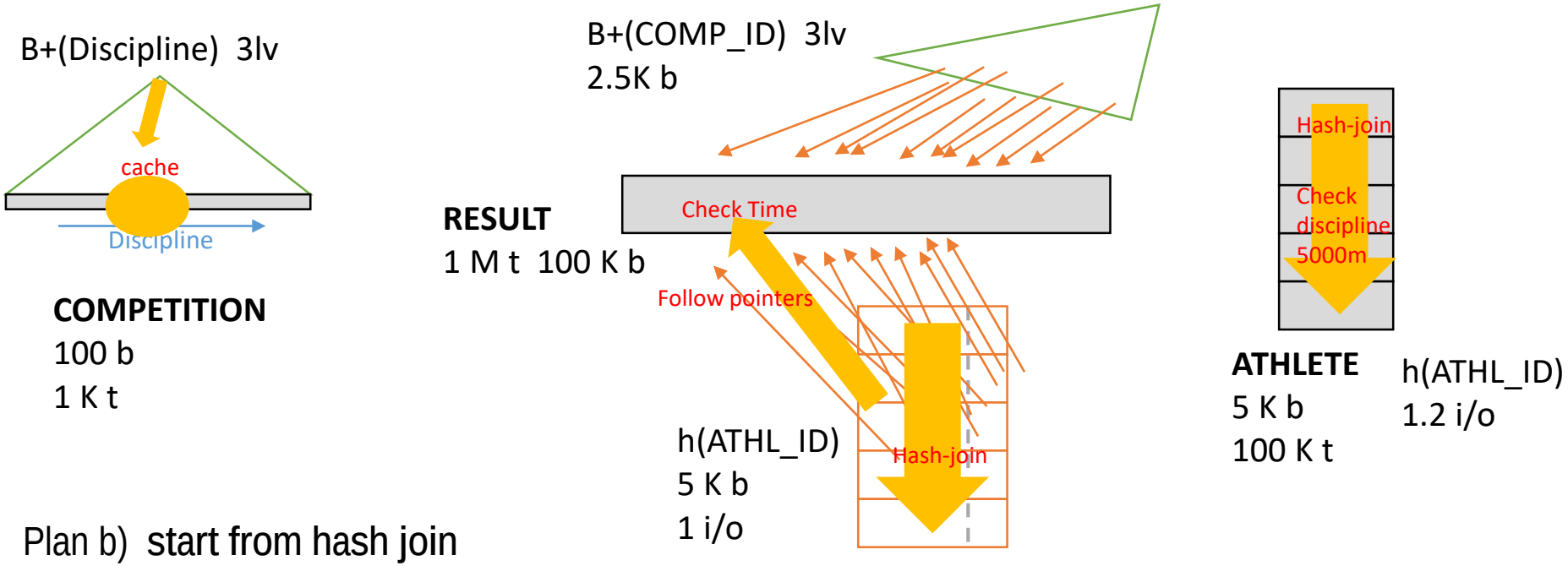
In both cases, the Discipline in Competition and Athlete must be the same, and the Time in RESULT must be checked.



Plan a) – use the ID of the winner and exploit the hash functions

- Apply the selectivity on the Competition B+ and access the 3 blocks containing 5000m competitions
- We have 20 tuples for *5000m* in these blocks, corresponding to 20 competitions. In each tuple, we have the ID of the winner!!
- For each competition, given the winner ID do a look-up in the hash index (cost 1) and follow the pointers to the corresponding results. How many pointers do we have? For each athete in average we have 1M/100K = 10 results.
- Check that the time is < 13 minutes.
- Then lookup the Athlete (we have at most 20 winners) only if the time is <13' using the hash function

Cost = 2 + 3 (tree nodes) + 20(winners)*(1 (hash cost) + 10 (pointers)) + 20*1.2 = **229 i/o**

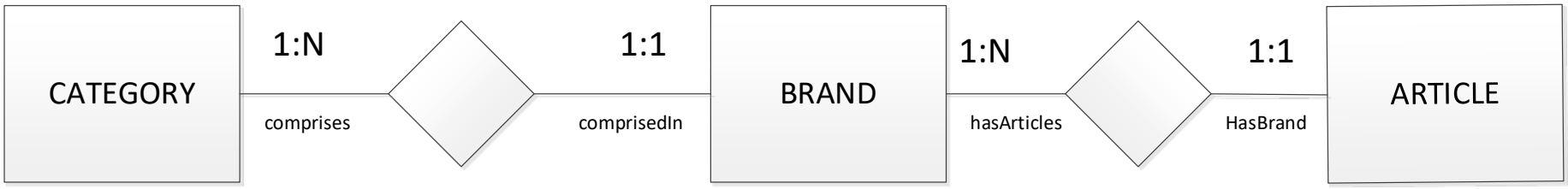


Plan b) start from hash join

- Apply the selectivity on the Competition B+ and cache 3 blocks like in the first plan
- Do a hash join on ATHL_ID. Select only the "5000m" athletes (cost = 5K + 5K)
- Since val(Discipline)=50 there will be 100K/50 =2k athletes for each discipline. For these athletes we need to access all their results to check if the time is less than 13 minutes. The number of pointers to follow is 1M/100K=10, which is the average number of competitions to which each athlete participates.
- For the athletes that have a time less than 13 minutes, check if they won that competition, by comparing the ATHL_ID and the COMP_ID (in the RESULT tuples) with the Winner and COMP_ID in the cached data (no additional costs)

Cost = 2 + 2 (tree nodes) + 5K+5K (hash join) + 2000 (athletes)*10(pointers) = **30K**

Exercise D. JPA in presence



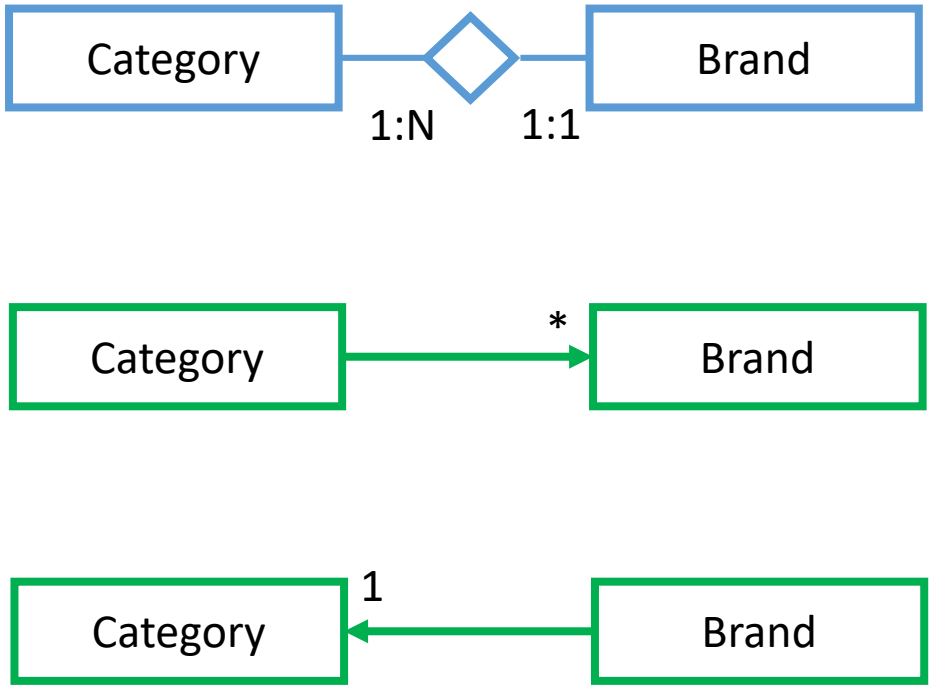
An e-commerce web application manages data that adhere to the following conceptual model.

The user can access a HOME page where he can start to drill down the catalogue. In the HOME page he can select one category (e.g., “sport apparel”) from a list of categories to see its brands, which are shown in a BRANDS page. For the list of brands in the BRANDS page he can select one brand (e.g., “Adidas”) of the chosen category to see its articles, which are listed in an ARTICLES page. By choosing one article from the list in the ARTICLES page finally he can see the details of the chosen article in the ARTICLE page. The details of an article include: code, name, price, picture and the brand it belongs to. Clicking on the bands attribute of an article leads back the ARTICLES page showing all the articles of that brand.

The categories are in the order of tens, the brands in the order of hundreds, and the articles in the order of hundreds of thousands.

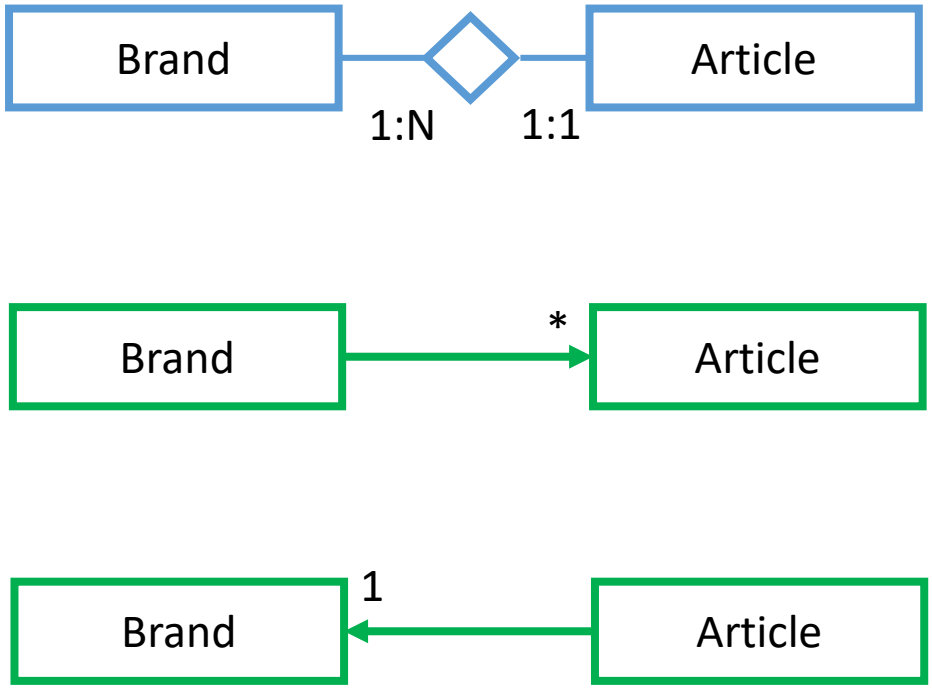
Show the JPA entities that map the domain objects of the conceptual model, taking into account the above mentioned access paths of the application. When designing the annotations for the relationships, specify the owner side of the relationship, the mapped-by attribute, and the cascading policies you consider more appropriate to support the access required by the web application.

Relationship comprises/comprisedIn



- Category → Brand
  @OneToMany is necessary show the list of brands in the BRANDS page
  - Owner = brand
  - FetchType can be EAGER
  - Remove can be cascaded
- Brand → Category
  @ManyToOne non necessary, can be mapped for consistency

Relationship has Articles/hasBrand



- Brand → Article
  @OneToMany is necessary to show the list of articles in the ARTICLES page
  - Owner = article
  - FetchType can be LAZY
  - Remove can be cascaded
- Article → Brand
  @ManyToOne can be used to navigate from the article back to the brand for filling the ARTICLES page of the brand (in alternative a named query can be used)

```
SELECT a FROM Article a WHERE a.brand = :brand
```

Entity Category

```
@Entity
public class Category{
 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int categoryId;
 private String categoryName;

 // Brands are few can be loaded eagerly
 // Removing the category removes the brands too
 @OneToMany(mappedBy="category“, fetch = FetchType.EAGER,
                                 cascade =  CascadeType.REMOVE)
 private List<Brand> brands;

 //getters and setters...

}
```

Entity Brand

```
@Entity
public class Brand{

 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int brandId;

 private String brandName;

 @ManyToOne
 @JoinColumn(name=“category")
 private Category category; // owner of the relation

 @OneToMany(mappedBy="brand", fetch = FetchType.LAZY,
                              cascade=CascadeType.REMOVE)
  private List<Article> articles;
       //getters and setters...
}
```

Entity Article

```
@Entity
public class Article{
 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int articleId;
 private String articleCode;
 private String articleName;
 private double price;

 // the picture is loaded only on demand
 @Basic(fetch = FetchType.LAZY)      @Lob
 private byte[] picture;

 @ManyToOne
 @JoinColumn(name=“brand")
 private Brand brand;

        //getters and setters...
}
```

Exercise D. JPA online

An e-commerce web application manages data about catalogues of articles belonging to different brands and categories. After logging in, the user can access a HOME page where he can start to drill down the catalogue. In the HOME page, he can select one category (e.g., “sports apparel”) from a list of categories to see its brands, which are shown in a BRANDS page. For the list of brands in the BRANDS page, he can select one brand (e.g., “Adidas”) of the chosen category to see its articles, which are listed in an ARTICLES page. By choosing one article from the list in the ARTICLES page he can see the details of the chosen article in the ARTICLE page. The details of an article include: code, name, price, picture and the brand it belongs to. Clicking on the brand attribute of an article leads back to the ARTICLES page showing all the articles of that brand. When a user displays the details of an article, the application creates a relationship between the user and the article’s brand, to record that the user may have a preference for such a brand.

The categories are in the order of tens, the brands in the order of hundreds, and the articles in the order of hundreds of thousands. Given the specifications

1) Design the Entity-Relationship diagram of the data model and write the SQL DDL code or draw the graphical model of the logical schema corresponding to the ER diagram.

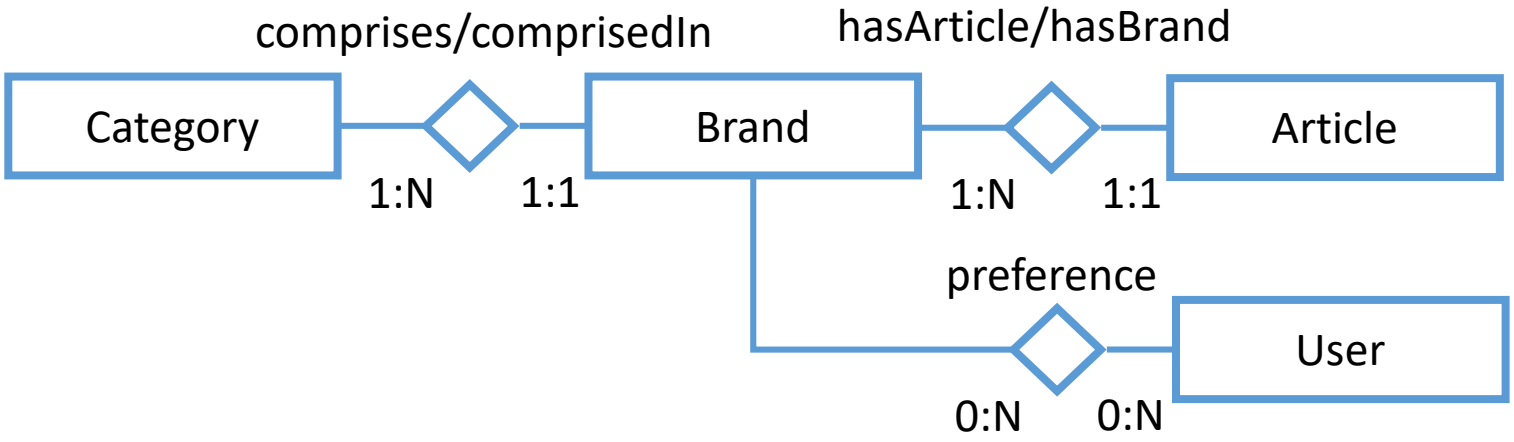
2) Write the entity classes of the ORM mapping, including annotations for the attributes and for the relationships, fetch type of attributes and of relationships, and operation cascading policies for relationships (when not by default). Motivate the design choices. Specify the named queries used by the methods of the business objects.

3) List the components of the application. For the data access services in the business tier, specify the type of the EJB component and write the complete signature of all the business methods. Motivate the design choices.

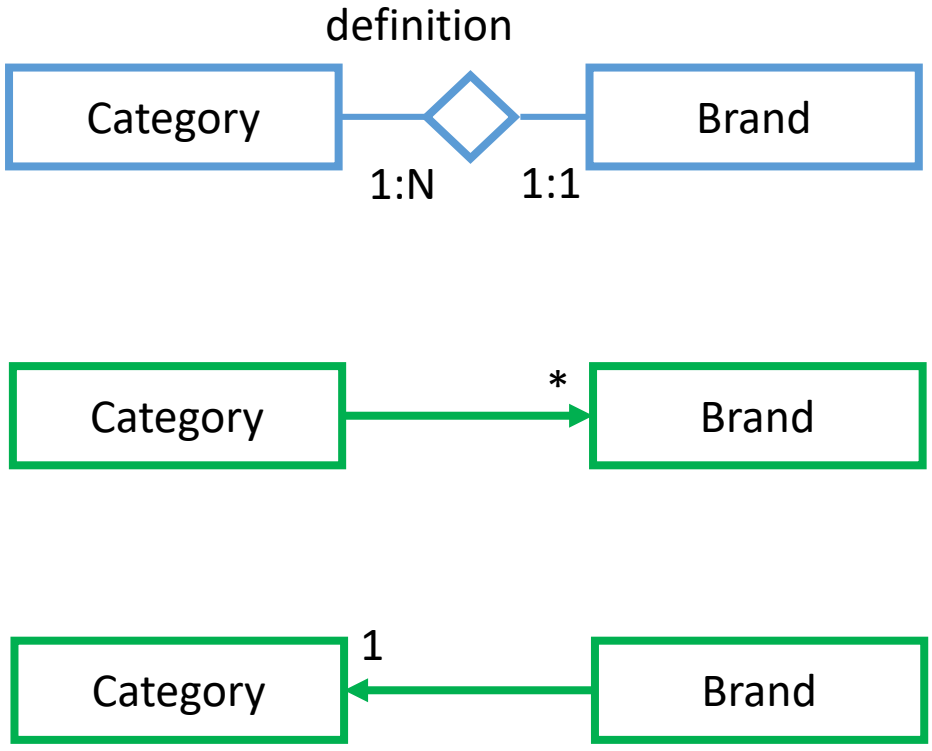
4) Write the Java code of the entity and business methods that respond to the selection of an article in the ARTICLES page.

--

ER

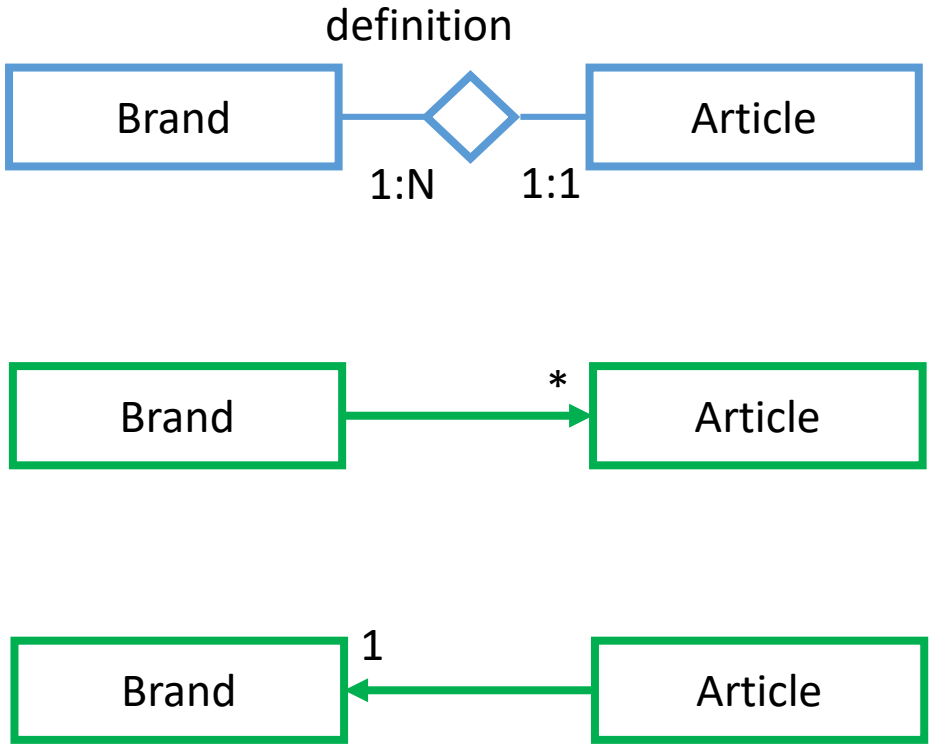

comprises/comprisedIn
hasArticle/hasBrand
Category
Brand
Article
1:N
1:1
1:N
1:1
preference
User
0:N
0:N

Relationship comprises/comprisedIn



- Category → Brand
  @OneToMany is necessary show the list of brands in the BRANDS page
  - Owner = brand
  - FetchType can be EAGER
  - Remove can be cascaded
- Brand → Category
  @ManyToOne non necessary, can be mapped for consistency

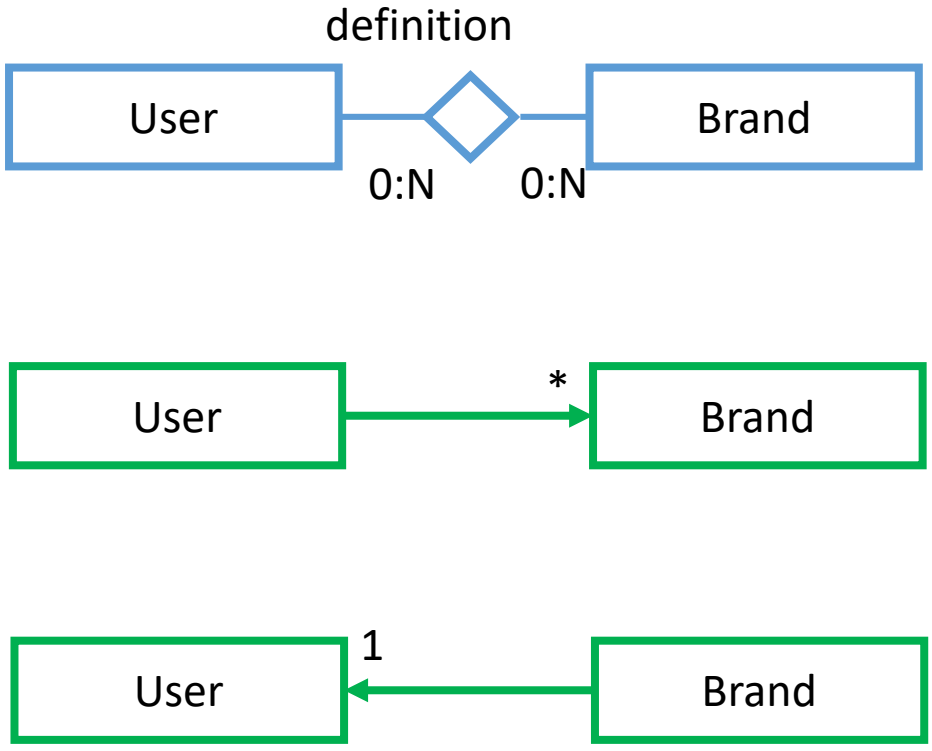
Relationship has Articles/hasBrand



- Brand → Article
  @OneToMany is necessary to show the list of articles in the ARTICLES page
  - Owner = article
  - FetchType can be LAZY
  - Remove can be cascaded
- Article → Brand
  @ManyToOne can be used to navigate from the article back to the brand for filling the ARTICLES page of the brand (in alternative a named query can be used)

```
SELECT a FROM Article a WHERE a.brand = :brand
```

Relationship preference



- User → Brand
  @ManyToMany may be used to display preferred content
  - Owner can be either user or brand
  - FetchType can be EAGER (only a few brands associated)
- Brand → User
  @ManyToMany not strictly necessary, can be mapped for symmetry, in this case
  - FetchType must be LAZY (many users could be associated)

Entity Category

```
@Entity
public class Category{
 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int categoryId;

 private String categoryName;

 // Brands are few can be loaded eagerly
 // Removing the category removes the brands too
 @OneToMany(mappedBy="category“, fetch = FetchType.EAGER,
                                 cascade =  CascadeType.REMOVE)
 private List<Brand> brands;
 //getters and setters...
}
```

Entity Brand

```
@Entity
public class Brand{
 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int brandId;
 private String brandName;

 @ManyToOne
 @JoinColumn(name=“category")
 private Category category; // owner of the relation

 @OneToMany(mappedBy="brand", fetch = FetchType.LAZY,
                              cascade=CascadeType.REMOVE)
  private List<Article> articles;

  @ManyToMany(fetch = FetchType.LAZY)
        @JoinTable( name=“usr_brand", joinColumns={
                          @JoinColumn(name=“brandid")}
                , inverseJoinColumns={
                          @JoinColumn(name=“userid")})
  private List<User> interestedUsers;

  //getters and setters...

}
```

Entity Article

```
@Entity
public class Article{
 @Id @GeneratedValue(strategy=GenerationType.AUTO)
 private int articleId;

 private String articleCode;
 private String articleName;
 private double price;

  // the picture is loaded only on demand
  @Basic(fetch = FetchType.LAZY)     @Lob
  private byte[] picture;

  @ManyToOne
  @JoinColumn(name=“brand")
  private Brand brand;

        //getters and setters...
}
```

Entity User

```
@Entity
public class User implements Serializable {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 // some attributes
 private String password;
 private String username;

 // bi-directional many-to-many association to Brand
 @ManyToMany(mappedBy = "interestedUsers", fetch = FetchType.EAGER)
 private List<Brand> preferredBrands;

 // setters and getters

}
```

Components

- Client components
  - Login/Logout
  - GoToHomePage: extracts all the categories
  - GoToBrandsPage: extracts all the brands of a category
  - GoToArticlesPage: extracts all the articles of a brand
  - GoToArticlePage: extracts all the details of an article and associates its brand to the user
  - Home.html: displays the list of all categories
  - Brands.html: displays the list of brands of a category
  - Articles.html: displays the list of articles of a brand
  - Article.html: displays the details of an article
- Business Components
  - UserService
    - Integer checkCredentials (string u, string pwd)
    - Void associateBrandToUser(brandId, userId)
  - CategoryService
    - List<Category> findCategories()
  - BrandService
    - List<Brand> findBrandsByCategory(catId)
    - Brand findBrandByID(brandID)
  - ArticleService
    - List<Article> findArticlesByBrand(brandId)
    - Article findArticleById(artId)

Business methods

```
// In entity Brand

        public void  addInterestedUser(User u) {
                getInterestedUser().add(u);
                u.getpreferredBrands().add(this);
        }



// In User Service


        public void associateBrandToUser(int brandId, int usrId) {
                User u = em.find(User.class, usrId);
                Brand b = em.find(Brand.class, brandId);
                if (!brand.getInterestedusers().contains(u))
                        b.addInterestedUser(u);
        }
```