Introduction

Developers using an object-relational database management system (ORDBMS) get the most benefit from this technology when taking a more "holistic" approach to database analysis and design than is typical with relational DBMS technology.

Traditionally, relational DBMSs are viewed as efficient, reliable, and relatively static repositories for business data. SQL-92 provides a flexible framework for storing facts about interesting real-world situations, and a mechanism for manipulating the data recording these facts. But there are many things that SQL's simplicity make difficult for developers to achieve. For example, try to answer the following simple question using SQL-92:

"Which of my employees has a birthday this week?"

Is that so hard, you ask? Well, if the calendars we actually use were as simple as SQL-92’s DATE datatype, it would be easy. But this question is complicated by the fact that people born on the 29th of February celebrate birthdays every year on the 28th of February, except during leap years. As a result, the SQL-92 query you need to write looks like this:
The principle goal of object-relational DBMS technology is to enable developers to avoid complications like this and, in turn, create smarter databases. Later in this article we will examine what the ORDBMS solution to this exact problem looks like. But first, we will review a high-level development methodology description for developers using an ORDBMS. This methodology combines aspects of both relational and object-oriented analysis and design. Along the way, we offer practical advice about what works best, and what doesn't work at all.

Design objectives for databases

A good way to think about an ORDBMS is as a kind of software "back-plane": a framework within which you can embed software modules (object classes) corresponding to the things the application is interested in. What distinguishes an object-relational DBMS from more conventional software frameworks—such as pure object-oriented DBMSs, application-servers, or TP-monitor middleware—is that the embedded object classes are deployed within an abstracted or logical data model. As a result, object-relational database development teams need to work at two levels. One team of developers implements the objects in the database using languages like C or Java, while another team combines these objects to address the high-level requirements of the application's problem domain.

The starting point for any MIS development project is the community of users the system is intended to support, with the overriding goal of the development project being to meet their needs. We use the term problem domain to refer to the whole set of user requirements. To adequately service the needs of these users, an information system should strive to be:

- **Complete.** The system must include all information relevant to the problem domain. Another way to say this is that the database must be shareable. This sometimes means that a database should include "reference" data: facts that users have no control over, but rely upon to make decisions (maps, documentation, metadata, and so on).
- **Correct.** The system should be an accurate rendering of the problem domain. As with all human undertakings, this reality will change with time, so the system should be capable of adapting to such change.
• **Consistent.** Ambiguity in the information system—for example, if it were to display two contradictory pieces of information—leads to confusion and error among the system’s users. Information systems should be designed to minimize the risk of such problems. One important way to achieve this is to design the database schema so that there is only one place in it for any fact to be stored.

• **Flexible.** The most important lesson to learn from the success of RDBMS technology is that there is tremendous value in being able to answer questions at run-time that were not anticipated at design time. Human beings are inherently more flexible than computer systems, and users will constantly be coming up with new questions and requirements. Indeed, how well a database supports this kind of activity is a measure of its strategic value.

• **Efficient.** Time is money. A system's efficiency must be measured according to several criteria: operational performance (end user response times), time-to-market (ease of development), and administrative costs.

These are the objectives of any information system, regardless of the technology used to build it. Projects differ with respect to the weight each objective is assigned. Embedded applications that are accessed mainly by other software systems place a premium on efficiency and consistency, but are not required to be particularly flexible. On the other hand, systems supporting managerial decision-makers should above all be correct and flexible. A clear understanding of the goals and objectives of a new system's user community is very helpful when deciding between alternative designs.

**Database analysis and design**

ANSI's *Three-Tier Database Model* is twenty years old, but it is as widely used today as it ever was (although marketers have endlessly re-packaged its concepts). The three-tiered model describes three "levels of abstraction," corresponding to:

1. The end user's conceptual framework and the ways they reason about the problem domain. This is called the **Conceptual** or **External** level.

2. The software services that stores information and provides facilities whereby other programs can retrieve and manipulate this information. This corresponds to the level of abstraction the DBMS software provides. We call this the **Logical** or **Internal** level.

3. The physical structures used to structure data storage and the low-level algorithms used to perform data retrieval and manipulation. Hiding the details of this **Physical** level is the primary value-add of a relational or object-relational DBMS.

All DBMS analysis and design methodologies describe techniques for representing the problem domain at different levels of abstraction, and algorithms for transforming different models between layers. Figure 2, below, provides an idealized illustration of how the three models are arranged, and how they interact.
The methodology we describe in this article is fairly conventional, in that it proceeds from left to right within the figure above. Initially it focuses on capturing a complete description of how the system's users think about the information they access. Such conceptual views consist of a list of the objects users work with and the facts described using these objects. Note that in many Web sites "users" are idealized. That is, the site's overall architect "channels" them for you, deciding what they ought to see, where they ought to see it, and how they ought to behave.

The conceptual model drives the design of the logical database, which is implemented as an object-relational database schema. In an object-relational DBMS, the schema concept is extended to include datatypes and their behavior, in addition to tables, views, and common business processes. Finally, the ORDBMS is sufficiently flexible that developers and administrators can fine-tune the logical schema to optimize system performance. This involves working at the physical layer--creating indices, partitioning data, and re-implementing embedded logic to achieve adequate performance.

Conceptual data modeling

Different users have different perspectives on the problem domain; a development team's first task is to analyze and document them all. Having said that, it is very important to begin this process in the sure and certain knowledge that conceptual models change. Historically, it was hard to modify user-interfaces. Upgrades to client-side software were made in large, staged releases. But because HTML and scripting languages make it so easy to modify an end-user's experience on a Web site, and because "Web time" is such an accelerated frame of reference, today's user-interface (UI) developers place tremendous pressure on their back-end database to evolve as rapidly as they do.

One thing that has not changed, however, is that the best way to document conceptual models is with diagrams. Graphical user-interfaces make drawing complex figures easy. And a plethora of semantic data models exist that formalize the meanings of different figures, permitting a team of developers to communicate clearly about what users think. Some of these semantic models include:
• Extended Entity-Relationship Modeling (EER). EER extends traditional Entity-Relationship Modeling with notions like inheritance and abstract datatypes (domains or classes of objects). Excellent handbooks for EER analysis include Toby Theorey's *Database Modeling and Design* (Morgan Kaufmann) and Fleming and von Halle's *Handbook of Relational Database Design* (Addison Wesley).

• More advanced relational modeling techniques have been recently developed. The best known of these is Object-Role Modeling or ORM. Terry Halpin's *Conceptual Schema and Relational Database Design* (Prentice Hall) is a superb introduction to this technique, and to conceptual analysis in general.

• Object-oriented analysis and design has produced several extremely useful diagramming languages. The best known of these is the Universal Modeling Language or UML. While intended primarily for developers working with object-oriented programming languages, UML can be readily adapted for working with object-relational DBMSs. Many books and tools describe and use UML.

Numerous Computer Assisted Software Engineering (CASE) programs exist to help developers automate conceptual modeling. Unfortunately, most of these tools do not, at this time, reflect the full capacities of the ORDBMS. They do not, for example, capture information about the palette of datatypes that are combined within database tables, or the functions implementing the behavior of these objects. Over time, these tools will improve.

**Extended entity relationship modeling**

For clarity and simplicity, we use the EER diagramming model in our example. EER doesn't pretend to cover every possible modeling situation, but it is simple, intuitive, and works quite well for a majority of cases.

Where the use of EER in ORDBMS methodology diverges from the traditional approach is that we analyze each entity's contents somewhat differently. Having arrived at a high-level overview of a user's conceptual model using EER, we use object-oriented techniques to decompose entities into a minimal set of object classes. Sometimes we treat an entire entity as an object class. But more typically, the elements that the entity is composed from--the kind of each element, and the role the type plays in the entity--can be effectively modeled as indivisible units of meaning within the database (using the techniques of object-oriented analysis).

For example, consider the classic Employee/Department/Product/Customer database schema beloved of university professors everywhere. Naturally, our example is updated a little and includes some typical real-world details that complicate matters, and are hard to handle in traditional RDBMSs. Figure 3, below, presents an Extended ER diagram describing the high-level conceptual model for this example.
A brief description of our application might look like this:

"Boxes-R-Us Inc. (the company) manufactures and sells a range of cardboard box Products of various colors and sizes to Customers. Its manufacturing is distributed among a set of Branches. Employees of the company each work at a Branch, but might transfer between Branches over time. Employees are classified into different subtypes: Contractors, and two kinds of Full_Time employees (Production workers and Sales staff). This is done because Employees are paid according to different schedules depending on what kind of employee they are. Sales staff coordinate the sale of a quantity of Product to a Customer."

All of the modeling concepts that characterize EER modeling remain valid when working with an ORDBMS. And as is the case with traditional EER models, it is a very good idea to document the following in your diagram:

- Estimate the parity of has_a relationships between entities. In our example, we use a "crows feet" technique to indicate that each Branch makes many Products, but that no Product is made at multiple branches. Other concepts, like compulsory relationships (where the existence of an instance of a weak entity is dependent upon the existence of a related strong entity) and business rules, are also useful.
- Describe any Keys. The key concept is important for two reasons. First, keys drive the normalization process, which is how a database designer can ensure that each fact in their schema is represented only once. And second, keys often indicate important rules about data that should be enforced to make sure that the data being stored is correct.
- Model relationships carefully. The figure above includes a pair of relationships that are more complex than simple has_a relationships between entities. The first of these is reflected in the Employees hierarchy. Such inheritance hierarchies can be represented more directly in an ORDBMS than in a RDBMS. The second kind of relationship is qualified. For example, Employees work at Branches for periods of time.

Let us examine the internal structure of a couple of these entities in more detail. Note that although we use a fairly standard Entity diagram here, you are probably better off using a UML style Class Diagram for this more detailed description. An important difference between the way you use EER
modeling with an RDBMS and an ORDBMS is that the ORDBMS model supports strong typing. That is, all things that are of a like kind (in relational theory, we say domain) should be identified as such. Cataloging all these domains as part of your EER model requires that you pay particular attention to the kind of data actually comprising each entity’s elements. For example:

**Figure 4. Detailed structure of two entities from EER model**

<table>
<thead>
<tr>
<th>Employee</th>
<th>Attribute_Name</th>
<th>Attribute_Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Employee_Num</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Person_Name</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Mail_Address</td>
<td></td>
</tr>
<tr>
<td>Resume</td>
<td>Document</td>
<td></td>
</tr>
<tr>
<td>Lives_At</td>
<td>Geo_Point</td>
<td></td>
</tr>
<tr>
<td>Date_of_Birth</td>
<td>DATE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer</th>
<th>Attribute_Name</th>
<th>Attribute_Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Customer_Num</td>
<td></td>
</tr>
<tr>
<td>Connect</td>
<td>Person_Name</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Deliver_Address</td>
<td></td>
</tr>
<tr>
<td>Bill_Address</td>
<td>Mail_Address</td>
<td></td>
</tr>
</tbody>
</table>

There are several things to point out about these entities.

First, notice how many different kinds of data there are, and how the strong typing captures more semantic information in the conceptual model. Using RDBMS technology it would be typical to decompose each attribute into a set of atomic components and assign each an SQL-92 type. We might, for example, use INTEGER instead of Employee_Num and Customer_Num. This has the unfortunate consequence that the attribute’s name must be used to distinguish kinds of information. With an ORDBMS, it is easy to create separate types corresponding to each kind of data object, and reserve the attribute’s name for its role in the entity.

Strong typing is a very good programming practice. In a schema that observes strong typing, it becomes possible to use the system catalogs to discover all kinds of information about how different parts of the schema fit together. Suppose, for example, you wanted to know everything that the database knows about Employees. In our schema, Employees are identified by Employee Numbers (the primary key of the employees' entity is an attribute named Id, which is specified to be an Employee_Num). All "tables" in the schema that have a column of this type can therefore be said to have something to do with Employees, even if none of the usual foreign key constraints are defined.

Strong typing is particularly effective in decision support databases. For data warehousing purposes, a developer might create several tables containing data extracted from several operational systems and loaded into a single database. Typically, data objects like Employee_Num are values used to cross-check records between different systems. Once all of this data is loaded into a single database, the strong typing allows developers who may be unaware of what data exists to answer important metadata questions, like "What information do we have about employees?"

When performing analysis at this level it is a good idea not to go into too much detail about the kinds of data you are working with. If you were using an RDBMS, the two Address attributes would likely be decomposed into First_Line, Second_Line, City, State, and Zip. Getting the most out of your ORDBMS means taking a slightly different approach. Once you have identified every place
within the problem domain that a particular kind of data is used, you are in a better position to understand what they all have in common. Then you can develop the structure and behavior of the object once and re-use it throughout the schema.

Attributes that include the employee’s resume and the geographic location where the employee lives are alien to RDBMS developers. These are, however, kinds of data that many object-relational databases manage. Under these circumstances, a developer can purchase a bundle of extensions--called IBM Informix®; DataBlade® modules--to manage this kind of data.

**Figure 5. Internal structure of Products entity**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Attribute Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Product_Number</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td>Dimensions</td>
<td>Physical_Size</td>
</tr>
<tr>
<td>Capacity</td>
<td>Mass</td>
</tr>
<tr>
<td>Available_In</td>
<td>SET(Color)</td>
</tr>
<tr>
<td>Price</td>
<td>Currency</td>
</tr>
</tbody>
</table>

There are two things to point out about the elements making up the Products entity.

The ORDBMS data model supports non-first normal form attributes. We see an example of its use here, to store the range of colors in which a particular box is available. At this stage, don’t worry too much about how this kind of structure will be physically represented. Although an ORDBMS table can store a set of data values in a single column, as we shall see, this may or may not be the best way to do it.

The boxes our company manufactures each have certain physical specifications; the most important of these are the box's dimensions (how long, how wide, and how tall) and how strong it is constructed (the amount of mass it can hold without tearing). For Boxes-R-Us, their business efficiency is bound to how easy they can make it for customers to place orders with them. This means that it would be ideal if their database allowed them to reason directly about things like physical mass, and whether or not an object that is "12 INCHES x 10 INCHES x 8 INCHES" can fit inside a box "31 CENTIMETERS x 31 CENTIMETERS x 31 CENTIMETERS." ORDBMS extensibility makes it possible to embed objects like this directly into the query language, which is the biggest advantage of the technology when building business applications.

So why is all of this activity useful? Because it helps you create a minimal catalog of the different kinds of data in the application. As we shall see, part of the development process involves implementing these various kinds of data using the *user-defined type* and *user-defined function* features of the ORDBMS.

**Object-oriented analysis and design**

Central to object-oriented analysis and design is the idea that an object’s interface (the means by which it is handled) should be separate from the details about its implementation (the data structures and logic within it). The role OO analysis plays in developing ORDBMS databases is as
a conceptual framework for working with the extensible type system. New objects are implemented using user-defined type mechanisms.

The behavior of these types is implemented within user-defined functions.

The first task is to create a list of all of the various kinds of data identified in the EER model. This includes both the entities and the various kinds of data making up attributes of these entities. The following table consists of a partial list, in no particular order, of various kinds of data identified in our example:

**Figure 6. List of domains from Boxes-R-Us EER schema**

<table>
<thead>
<tr>
<th>Product_Number</th>
<th>String</th>
<th>Physical_Size</th>
<th>Mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Employee_Num</td>
<td>Person_Name</td>
<td>Mail_Address</td>
</tr>
<tr>
<td>Geo_Point</td>
<td>Document</td>
<td>Customer_Num</td>
<td>Delivery_Address</td>
</tr>
<tr>
<td>Currency</td>
<td>Bonus_Plan</td>
<td>Period</td>
<td>Quantity</td>
</tr>
<tr>
<td>Product</td>
<td>Employees</td>
<td>Full_Time_Emp</td>
<td>Sales</td>
</tr>
</tbody>
</table>

An important virtue of this approach is that you only need to develop shared data objects once, even though they might be used in many places throughout the schema. This way, if you discover an error or an oversight in an earlier phase of analysis, the error is relatively easy to fix. If the definitions of these objects were scattered across multiple places throughout the schema, correcting a problem could become a large task indeed.

Frequently, users will use different terms to refer to the same thing (*synonyms*) or the same term to refer to different things (*antonyms*). Using object-oriented techniques helps you to sort out these ambiguities. Object-oriented software engineering techniques revolve around the concept of an "object" as an atomic unit of meaning encapsulating both state and behavior. Although no one may agree on the names, if the two objects are sufficiently similar in terms of how they are handled, then it is highly likely that they are the same. Similarly, two kinds of data with the same name but completely different definitions are probably different objects.

For example, the Mail_Address and Delivery_Address domains are candidates for being synonyms. The following figure presents the interface for each of these objects using the UML standard for presenting object interfaces. UML class diagrams express the static structure of an object, including whether or not the elements of that structure can be addressed directly by developers using the type. What is clear from these figures is that these two kinds of data are very closely related but differ in that instances of a Delivery_Address may include a document giving detailed delivery instructions.
When turning this analysis into an implementation plan, the developer has two choices. First, they can develop each of these as completely unrelated types. Alternatively, they can use type inheritance to re-use the implementation of the Mail_Address type in the Delivery_Address. By using type inheritance to address this problem, a developer would reduce the total amount of code they had to write. The code used to format an address label from a Mail_Address might be re-used with the Delivery_Address. And, of course, they always have the option of overloading the behavior if necessary.

Some kinds of data are examples of patterns. A pattern is a repeated structure that many types use. Patterns are useful because they allow you to flesh out a complete design based on a few characteristics.

Enumerated objects are an example of a very simple pattern. In our database, it might be the case that the range of Colors in which boxes can be manufactured is quite small—maybe only black, white, and brown. To keep the data as consistent as possible, a developer can use the implementation of the enumerated type to enforce these rules. Internally, instances of a Color object might simply be stored as a 2-byte integer. But in the code that returns values to a client program, or inserts values into the table, a developer can write logic to ensure that invalid instances are simply impossible.

Another way to describe this is to say that a new object must implement certain behaviors to be used in certain ways. For example, when a customer or sales staff member is browsing the product catalog, they may search by certain criteria. They may only want boxes capable of holding an object weighing three pounds. In other words, they want boxes whose capacity exceeds three pounds. For efficient browsing, they would like their query to return matching boxes in ascending order of capacity.

Such a query would look like this:

"Show me all Boxes with a Capacity in excess of ‘3 Pounds’?"
Figure 8. Example of workload query

```sql
SELECT P.Id, 
P.Capacity  
FROM Products P  
WHERE P.Capacity > '3 POUNDS'  
ORDER BY P.Capacity ASC
```

By knowing that this kind of query is important, you can infer that users want to compare instances of this object with one another, sort them, index them, and so on. To support this kind of functionality, a new type needs to implement what are known as the *ordinal operator functions* (LessThan, LessThanOrEqual, Equal, GreaterThanOrEqual, GreaterThan, NotEqual) and *B-tree support functions* (Compare()).

In the **following figure**, we present the UML diagram for the Mass datatype. Note how in this diagram we use the full definition of each of the object's behaviors. This helps us to differentiate among them, and identify "reserved" behaviors that are common for multiple types.

Figure 9. Example of UML class diagram with ordinal support

Other types, that use other indexing, need to supply different support functions. For example, consider the Period datatype used to store the duration of a particular employee's association with a branch. Periods are objects with beginning and ending dates or dates and times. Efficiently supporting concepts like overlapping periods requires that the type include behaviors that the R-tree access method can use.

Let us examine the employee hierarchy in more detail. Recall that the main reason that Employees is divided into sub-categories is that different kinds of employees are paid differently (their remuneration is calculated differently) based on the kind of work they perform at Boxes-R-Us. The figure **below** illustrates the differences between Production workers and Sales staff members.
Notice the degree of overlap in these definitions. With the exception of the attributes relating to how different kinds of employees are paid, these two entities are identical. Note also how these entities share common behaviors. When we examine the other kinds of employees we observe a similar pattern. However, the algorithm used to calculate remuneration varies, depending on the employee category.

Formalizing behaviors is obviously an important task at this stage. All of the classes introduced so far possess some kind of behavior. In some cases, it is not obvious how the behavior can be expressed. For example, the Mass class might need to cater to both metric and imperial units, in which case it will also need to address conversion. Sometimes, you will observe that the behavior might even vary. The exact algorithm used to calculate a Sales representative’s annual salary varies from year to year, as management attempts to direct the sales force towards different goals.

In this presentation of the methodology, we have described a top-down approach. Larger structures are identified and analyzed first, before decomposing the problem down to its smallest parts. Another equally useful approach is to begin with an analysis of the smallest components of the problem and find out how they are combined. This approach, called bottom-up analysis, has the advantage that higher level structures (like tables and the relationships between them) tend to change more rapidly, and are simpler to modify. By focusing first on assembling a comprehensive palette of object types within the database, you will often find that questions about higher level structures are simpler to answer. Deciding which is appropriate for your project is a matter of taste and experience.

Data flow, workload, and queries

Another aspect of the overall application that is important to consider at this stage is the workload: the set of common queries and business processes that the information system supports. A description of the workload is useful in several ways.

First, developers can embed logic-implementing business operations into the ORDBMS and invoke them directly from a client program. Of course, this is nothing new. Relational DBMSs have supported such database procedures for some time. But many developers were wary of using the technique, because although it offered improved performance, it required them to develop
code in proprietary procedural languages. With an ORDBMS, these database procedures can be implemented in open languages using standard APIs, such as Java and JDBC.

Second, one of the experiences of early adopters of ORDBMS technology has been that by introducing logic that was traditionally linked into external programs into the ORDBMS they can improve the performance and flexibility of their overall system. For example, it isn't too much of a stretch to suggest that Boxes-R-U's might want to send each of their employees a card or note on their birthday. As we saw earlier, supporting an apparently simple operation like "Given a date-of-birth, is today a birthday?" in SQL-92 is impossible, because matching birthdays has to take into account the situation of people born on the 29th of February during a leap year. During non-leap years, these people celebrate their birthday on 28th of February. Ideally, you would like to be able to run the following query, letting it deal with the complexity of the problem.

**Figure 11. Birthday query**

```sql
SELECT E.Name
FROM Employees E
WHERE Birthday ( E.Date_of_Birth ) = Birthday ( TODAY );
```

In an external program it would be obvious to create a "Birthday" class in a language like C++ or Java. Using it would require that you write a query retrieving the names and dates of birth of everyone born during the current month (about 1/12th of the total data). Then on the client side, programming logic in the Birthday class would determine matches. Such an object class might look like this:

**Figure 12. UML class diagram for birthday**

An important principle of the ORDBMS data model is that types like this do not have to be used to define a table's structure or a column. The behavior of this object can be used to implement complex query operations over otherwise conventional data, as depicted in Figure 11. Also, most object-oriented methodologies acknowledge that sometimes what you want is not really an object, but simply a function or procedure. Readers for whom object-oriented everything sounds like babble can take comfort in the fact that it is entirely possible to think about the design of your ORDBMS database in an entirely functional or modular mode just as easily.

Third, a workload catalog gives you the basis for writing a performance and scalability test environment. One of the most common causes of technical failure in new software systems is that developers do not test what they implement in anything like a production environment.
Conclusion

In this article, we have examined the initial development methodology for object-relational database applications. We have seen that, in order to get the most from their ORDBMS, developers need to gather more information about the application's problem domain than is typical in relational DBMS development. Because an ORDBMS permits them to embed modules of software to represent the state and behavior of the objects in their problem domain, developers must pay attention to aspects of their application that were traditionally the task of external programs, middleware, or the client-interface.

Using a combination of Extended Entity-Relationship modeling and the Universal Modeling Language's approach to expressing class definitions pictorially, we were able to take an application and reduce it to:

1. The minimal, yet complete list of the objects of interest in the problem domain.
2. The means whereby these objects are structured into a correct and consistent representation of the facts we wish to record.

In the final installment of this series, we will describe a procedure whereby the semantic model we have described here can be represented using the various features of the ORDBMS data model. As part of this detailed design process, we review the performance and flexibility tradeoff between different mechanisms.

© Copyright IBM Corporation 2002
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)