The core of IBM Cloudscape™ is the Apache Derby open source database. Derby's lightweight, embeddable architecture makes it the ideal database engine for deploying database-driven Java™ applications. With Derby, the database engine becomes part of the application, so the user never has to install or manage it. This article provides an overview of Derby technical features.

Cloudscape support

On March 13, 2007, IBM announced that it would no longer produce IBM Cloudscape and would be phasing out support for releases 10.0 and 10.1 of the software. The IBM Cloudscape referral page contains links intended to help you find information on Apache Derby sites as an alternative to the information previously posted at IBM.

About the names Derby and Cloudscape

Cloudscape is the original zero-admin, embeddable, 100% Java relational database that entered the marketplace in 1996. In August 2004 IBM contributed Derby, a copy of its Cloudscape 10.0 relational database product, to the Apache Software Foundation (ASF) to help accelerate innovation around data-driven Java applications. IBM continues its Cloudscape commercial offering at no-charge, which adds features to the core Derby engine. The most current release is Cloudscape 10.1, which includes Apache Derby 10.1.

Not surprisingly, the existence of two names side-by-side (Cloudscape and Derby) sometimes causes confusion. With that in mind, this section clarifies what is in the Apache Derby software and what is in the IBM Cloudscape product. We mention features you probably won't recognize yet, but rest assured that they are described later in this technical article.

This technical article uses "Derby" when referring to any feature that is part of the open source database engine, including:

- The core RDBMS engine.
- The embedded JDBC Driver.
- The Network Server.
- The Derby Network Client JDBC Driver for use with the Network Server.
The IBM Cloudscape product includes Derby without any modification whatsoever to the underlying source code. This technical article uses "Cloudscape" when referring to the features IBM adds to the core Derby engine, such as:

- IBM Technical Support.
- IBM DB2 UDB Universal JDBC Driver for use with the Network Server.
- The IBM Cloudscape manuals, translated manuals, and the Cloudscape Information Center.
- Installers and sample databases that jump start developers who are new to Java and/or Derby.

At times, you might notice Derby and Cloudscape being used as synonyms, and you might see the core database engine referred to as Cloudscape, especially in other IBM developerWorks technical articles and when referring to older Cloudscape releases. We apologize for any confusion. Moving forward, think **Derby** for the core database features and **IBM Cloudscape** for the supported commercial release. And if you find yourself using the two terms synonymously, don't worry; we understand what you mean.

**Introduction**

Derby is a lightweight, embeddable relational engine in the form of a Java class library. Its native interface is Java Database Connectivity (JDBC), with Java-relational extensions. It implements the SQL92E standard as well as many SQL 99 extensions. The engine provides transactions and crash recovery, and allows multiple connections and multiple threads to use a connection. Derby can be embedded easily into any Java application program or server framework without introducing other programming languages because it is a Java class library. Derby's support for complex SQL transactions and JDBC allows your applications to migrate to other SQL databases, such as IBM DB2® Universal Database™ (UDB), when they need to grow.

The Derby Network Server provides connectivity to Derby databases within a single system or over a network. The Derby Network Server receives and replies to queries from clients using a standard network protocol. Databases are accessed through the Derby Network Server using the Derby Network Client JDBC driver or IBM DB2 Universal JDBC Driver. (Not all Derby functionality is supported when using the Network Server. See the **Server and Administration Guide** for details.)

Several technical aspects differentiate Derby from other database systems:

- Derby is easy to administer. When embedded in a client application, a Derby system requires no administration at all.
- Derby is embeddable. Applications can embed the Database Management System (DBMS) engine in the application process, eliminating the need to manage a separate database process or service.
- Derby can run as a separate process, using the Network Server framework or a server framework of your choice.
- Derby is a pure Java class library. This is important to Java developers who are trying to maintain the advantages of Java technology, such as platform independence, ease of configuration, and ease of installation.
• Derby needs no proprietary Java Virtual Machine (JVM). Written entirely in the Java language, it runs with any certified Java 2 Platform, Standard Edition (J2SE) JVM at a release of 1.3 or higher. Beginning with Derby 10.1, there is new Java 2 Platform, Micro Edition (J2ME) support for JSR 169, the JDBC API defined for the Connected Device Configuration (CDC) / Foundation Profile (FP).

• The Derby DBMS engine is lightweight. It is about 2MB of class files, and it uses as little as 4MB of Java heap.

• Derby provides the ability to write stored procedures and functions in Java. Derby does not have a proprietary stored procedure language; it uses JDBC.

Derby is also like other relational database systems. Derby implements the SQL92E language standard and the JDBC API standard. It has transactions (commit and rollback), supports multiple connections with transactional isolation, and provides crash recovery. Like other databases, it allows multiple threads to share the same connection, and Derby implements many SQL99 features, with extensions for Java technology.

This unique combination of technical capabilities allows application developers to build data-driven applications that are pervasive (run anywhere), deployable (downloadable), manageable, extensible, and connectable. These technical features are discussed in this article, organized as shown below:

• General architecture
  • Embedded
  • Network Server
  • Easy Integration
  • Data Persistence
  • Flexibility
  • Programmer’s application program interface (API)
  • JDBC drivers
  • CLI/ODBC driver

• Relational Database Management System (RDBMS) capabilities

• Security features
  • User authentication
  • User authorization
  • Java 2 Security Manager
  • Encrypted databases

• Java extensions
  • Java functions
  • Using Java functions in triggers
  • Java stored procedures
  • Loading Java classes from the database

• IBM Cloudscape Features

• Summary
General architecture

Derby is a relational database management engine implemented as a Java class library. Applications use the JDBC API to access data managed by the Derby engine.

Embedded

The database engine is embeddable. This means that rather than running as a separate process, the database engine software can be part of the application so that the application and the database engine run in the same JVM. With an embedded database engine, the application uses the JDBC API to access the database. The embedded JDBC driver transfers data to and from the database engine without the need for network communication. Whether or not it is embedded, the database engine supports multiple simultaneous connections and access from multiple application threads.

![Figure 1. Derby embedded in an application](image)

Network Server

It is also possible to deploy Derby as a traditional client/server database server. To accomplish this, use Derby embedded in a server framework (that is, a piece of software that can accept and process network communication). Derby includes a Network Server, but you can also run Derby in a server framework of your choice.
Figure 2. Derby acts as a client/server DBMS within a server framework

Easy Integration

It is just as easy to embed Derby inside a Java Web server, such as IBM WebSphere® Application Server. Figure 3 shows Derby embedded in a Java Web server, with clients accessing the database through Hypertext Transfer Protocol (HTTP) requests to servlets.

Figure 3. Derby embedded in a Web server

Data Persistence

Derby provides persistence of data by storing data in disk files. A Derby engine can manage one or more database files, but each database file can only be accessed by a single Derby engine.
In a client/server configuration, the engine provides multi-user access to the databases under its control. All threads that access the database do so through the database engine.

**Figure 4. Thread access to Derby**

![Thread access to Derby diagram]

The data in the database disk files is stored in a portable format, so that databases can be easily transported from machine to machine regardless of the CPU architecture of each machine. For example, you can directly copy a Derby database you developed on a Linux machine to a Windows machine -- or even to a small device running J2ME/CDC/FP. Derby can also handle data files on read-only media. These characteristics make it easy to download Derby applications together with their database or run them from a CD-ROM. The combination of portable database formats and the pure Java DBMS engine makes it possible to send a data-centric application anywhere, either on media or over a network.

**Flexibility**

Derby provides a great deal of flexibility for system designers. Each Derby instance can manage multiple databases, the databases can live on various media, and there's nothing to stop the application from connecting to other DBMS systems.

**Figure 5. Flexible application architecture**

![Flexible application architecture diagram]
Programmer’s API

Applications access the database with the industry standard JDBC API. When a client application needs to store or retrieve data in the database, it submits a request through the JDBC API to the Derby engine (either over a network or directly to the embedded engine). The client program is not required to use any Derby-specific APIs; clients only use JDBC. The Derby driver is selected by supplying a Derby JDBC connection Uniform Resource Locator (URL), as shown in Listing 1.

Listing 1. Derby JDBC connection URL

```java
Connection conn = DriverManager.getConnection("jdbc:derby:greetdb");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT message FROM greetings");
while (rs.next())
{
    String greeting = rs.getString(1);
    System.out.println(greeting);
}
rs.close();
stmt.close();
conn.close();
```

Since the application is only using JDBC APIs, it is portable, and will run against JDBC-compliant databases from other vendors. This standards compliance enables a developer to write an application against Derby and deploy it against another DBMS.

JDBC drivers

To support the embedded architecture, Derby includes an embedded JDBC driver for accessing Derby from Java client programs. The embedded driver is used to communicate with Derby when it is running in the same JVM as the application program. To support the client-server architecture, the Derby Network Client works with the Derby Network Server. IBM Cloudscape includes the IBM DB2 UDB JDBC Universal Driver, which also works with the Derby Network Server. These JDBC drivers are Type 4 drivers: pure Java technology on the client and connect directly to the database engine without any intermediate translation.

JDBC connection URLs to Derby databases all start with the prefix `jdbc:derby:`. Given a database called `mydb` that is on the file system in `/home/databases`, Table 1 below summarizes what the connection URL might look like for the various JDBC drivers:

<table>
<thead>
<tr>
<th>JDBC Driver</th>
<th>Connection URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded</td>
<td><code>jdbc:derby:/home/databases/mydb</code></td>
</tr>
<tr>
<td>Derby Network Client</td>
<td><code>jdbc:derby://localhost:1527/home/databases/mydb</code></td>
</tr>
<tr>
<td>IBM DB2 UDB JDBC</td>
<td><code>jdbc:derby:net://localhost:1527/home/databases/mydb</code></td>
</tr>
</tbody>
</table>

Derby JDBC drivers support JDBC 2.0 and 3.0; they work with JVM 1.3.1 and beyond. The standard extensions for JNDI data sources, connection pooling, and XA are also provided. These features enable Derby to be a resource manager in a distributed J2EE system.
CLI/ODBC driver

The IBM DB2 CLI/ODBC driver for Derby is available as a free download from IBM developerWorks. In addition to supporting standard ODBC clients, it also makes it possible to develop PHP/Perl and .NET applications over ODBC.

RDBMS capabilities

Derby implements the SQL92E language standard and many features of SQL99, with extensions for Java. SQL language support includes the following:

- Basic types: CHAR, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, NUMERIC, REAL, SMALLINT.
- Datetime data types (from SQL92T): DATE, TIME, TIMESTAMP (with JDBC date/time escape syntax).
- Other types: BIGINT, VARCHAR, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, BLOB, CLOB.
- Basic math operations: +, *, -, /, unary +, unary -.
- Basic comparisons: <, >, <=, >=, <>, =.
- Datetime literals
- Datetime arithmetic using JDBC escape functions
- Built-in functions: ABS or ABSVAL, CAST, LENGTH, concatenation (||), NULLIF and CASE expressions, CURRENT_DATE, CURRENT_ISOLATION, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, DATE, DAY, HOUR, IDENTITY_VAL_LOCAL, LOCATE, LCASE or LOWER, LTRIM, MINUTE, MOD, MONTH, RTRIM, SECOND, SESSION_USER, SQRT, SUBSTR, TIME, TIMESTAMP, UCASE or UPPER, USER, YEAR.
- Basic predicates: BETWEEN, LIKE, NULL
- Quantified predicates: IN, ALL, ANY/SOME, EXISTS
- CREATE and DROP SCHEMA
- CREATE and DROP TABLE
- Check constraints
- ALTER TABLE: ADD COLUMN and ADD/DROP CONSTRAINT
- CREATE and DROP VIEW
- CREATE and DROP SYNONYM
- Constraints: NOT NULL, UNIQUE, PRIMARY KEY, CHECK, FOREIGN KEY
- Cascade delete
- Column defaults
- Delimited identifiers
- Updatable cursors (through JDBC)
- Dynamic SQL (through JDBC)
- INSERT, UPDATE, and DELETE statements
- Positioned updates and deletes
- WHERE qualifications
- GROUP BY
- HAVING
- ORDER BY
• UNION, UNION ALL, INTERSECT, and EXCEPT
• Subqueries as expressions (from SQL92F)
• Joins in the WHERE clause
• Joins (SQL92T): INNER, RIGHT OUTER, LEFT OUTER, named column join, condition join
• Aggregate functions (with DISTINCT): AVG, COUNT, MAX, MIN, SUM
• Select expressions
• SELECT *, SELECT table.* (SQL92T), SELECT DISTINCT, select expressions
• Named select columns
• SQLSTATE
• UNION in views (SQL92T)
• CAST (SQL92T)
• INSERT expressions (SQL92T): insert into T2 (COL) select col from T1
• VALUES expressions: select * from (values (1, 2)) as foo(x, y), etc.
• Triggers

Other traditional database features include:

• Cost-based query optimizer: join order, index selection, bulk fetching, join strategies (nested loop or hash), sort avoidance, lock escalation, subquery flattening, transitive closure, and many other query transformations. It uses a unique sampling technique that requires no intervention for statistical gathering, and also provides query plan overrides and statistics on actual query results.
• Multi-column B-Tree indexes
• Unlimited-length rows, and the capability of "streaming" column values to the client
• Data import and export with fast-load and fast-create index
• Transaction commit and rollback
• Transactional isolation (serializable, repeatable read, read committed, or dirty read)
• Support for global transactions spanning multiple databases
• Crash recovery
• Multiple concurrent connections
• Multi-threaded connections
• Row locking with escalation to table locks
• User and connection authentication: built-in SH3, application-defined, or through external user authentication systems (LDAP, JNDI)
• Diagnostics and consistency checks
• Online database backup
• Locales
• Database encryption
• Script tool, ij
• DDL examination tool, dblook
• Modify column to change its length or nullability
• Identity default for automatic sequence numbers for data
• Import values into an identity column
Security features

Unlike high-end enterprise databases housed in secure facilities, you might find Apache Derby databases anywhere you look -- on servers, workstations, notebooks, laptops, kiosks, or, beginning in Derby 10.1, even on smaller devices running J2ME/CDC/FP. Derby provides features for securely deploying databases to mobile or remote sites, including authentication using a variety of schemes, user authorization, support for running with a Java 2 Security Manager, and database encryption. Each of these security features is introduced below.

User authentication

Derby supports a simple, built-in mode for maintaining an internal list of authorized users. It also supports integrating external user authentication schemes; for example, it can be configured to check with an external authentication service. Lightweight Directory Access Protocol (LDAP) support is provided, and custom schemes are supported with user-defined Java Naming and Directory Interface (JNDI) classes.

Not having the user names and passwords maintained in the database means less administrative overhead to transfer names into the database. This is especially important in deployed server applications, which must be deployed with as little administrative overhead as possible.

LDAP is an emerging Internet standard that provides an open directory access protocol running over TCP/IP. Windows NT™ domain user authentication can be provided through LDAP by using the Netscape NT Synchronization Service.

User authorization

Since most access to deployed Derby databases is through applications, Derby provides simple user authorization controls. Users can be restricted to read-only access or restricted from any access on a per-system or per-database level. This ensures that only permitted, authenticated users access or modify a database.

Java 2 Security Manager

Derby can be run under a Java 2 security manager. In addition to supporting policy files, Derby can detect digital signatures on JAR files. Before loading a class from a signed JAR file stored in the database, Derby will verify the JAR was signed with a X.509 certificate and verify the validity of the signature.

Encrypted databases

When a database is deployed to a remote or mobile location, it is not possible to use physical security to prevent unauthorized access to data. If the data files can be read, a sophisticated user could decode the information they contain. The only way to secure data in this environment is to encrypt it on disk. That way, simply being able to read the database files does not reveal the data.

Derby supports secure remote data database encryption. All data in such a database is decrypted by the database engine when read and re-encrypted when written back to disk. No data exists in
clear-text form in the database files. Since the entire database is encrypted, the structure of the
database schema is also hidden.

For an encrypted database to be usable, a boot password must be provided when the database
is first started. This is a separate password from the usual database connection username and
password, which must also be supplied to access the database. Without the boot password, the
database will not start.

Database encryption is useful for applications that distribute databases to locations where
physical security of the files cannot be guaranteed. For example, mobile databases on notebook
computers can be stolen if the notebook computer is stolen. Applications that are installed on
remote multi-user machines are subject to unauthorized access if the remote administrator does
not appropriately protect the files.

Database encryption adds less than 10% performance overhead, and takes no additional disk
space. It is based on the 1.2.1 version of the Java Cryptographic Extension (JCE). Data Encryption
Standard (DES) is used as a default encryption method, or the user can configure which algorithm
to use.

**Java extensions**

Derby supports the following Java extensions:

- Java functions
- Using Java functions in triggers
- Java stored procedures
- Loading Java classes from the database

The examples in the following sections assume that the `ADDRESS` table exists in a database along
with the `ZIPMAP` table that maps ZIP codes to cities. Listing 2 shows the schema for the tables.

**Listing 2. Sample schema**

```sql
CREATE TABLE address (name VARCHAR(60), street VARCHAR(255), zipcode INTEGER)
CREATE TABLE zipmap  (zipcode INTEGER, city VARCHAR(255))
```

**Java functions**

A Java function consists of Java code that is callable from SQL. A Java function can be invoked
anywhere in a SQL statement that a SQL expression is allowed. It can also be invoked with the
`VALUES` keyword, and it can be invoked inside a trigger.

For example, Listing 3 creates a function in the ZipMap class that looks up the name of a city given
its ZIP code (error and exception-handling code are ommitted for brevity).
Listing 3. Sample function source code

```java
public class ZipMap {
    public static String cityFromZipFunc (int zipcode) {
        String city;
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        String query = "SELECT city FROM zipmap WHERE zipcode = ?";
        PreparedStatement ps = conn.prepareStatement(query);
        ps.setInt(1, zipcode);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            city = "Unknown City";
        else
            city = rs.getString(1);
        rs.close();
        ps.close();
        conn.close();
        return city;
    }
}
```

The SQL statement that creates a function for this Java code is shown in Listing 4:

**Listing 4. Sample CREATE FUNCTION**

CREATE FUNCTION cityFromZip (zipcode INTEGER)
RETURNS VARCHAR(255)
LANGUAGE JAVA
PARAMETER STYLE JAVA
READS SQL DATA
EXTERNAL NAME 'com.acme.sales.ZipMap.cityFromZipFunc'

Once the `cityFromZip` function has been created, it can be invoked using the `VALUES` keyword or by referencing it in the SQL statement as shown in Listing 5.

**Listing 5. Sample cityFromZip function usage**

```
VALUES cityFromZip(94105)
SELECT name, street, cityFromZip(zipcode), zipcode
FROM address

SELECT *
FROM address
WHERE cityFromZip(zipcode)='San Francisco'
```

Using Java functions in triggers

A SQL trigger specifies the actions that should occur when a SQL statement inserts into, updates, or deletes from a table. Here’s an example of a trigger that calls a Java function to send an e-mail alert when a new record is inserted into the `ADDRESS` table.

**Listing 6. Sample function in a trigger**

```
CREATE TRIGGER newaddress AFTER INSERT ON address
REFERENCING NEW as NEW
FOR EACH ROW MODE DB2SQL
    VALUES(sendEmailAlert(NEW.name))
```
Java stored procedures

A Java stored procedure consists of Java code that is callable from SQL, runs in the database server, and accesses the database.

Derby supports the `CALL` statement for invoking a Java stored procedure. Derby supplies JDBC `OUT` parameters so that values can be returned in parameters; this is done by having the Java parameter be an array of the type desired. For example, an `int` parameter would be declared as `int[]` and a `String` parameter would be declared as `String[]` so a value could be returned in it. If the stored procedure includes any `OUT` parameters, it needs to be invoked from a client application using the `CallableStatement` interface. If the stored procedure does not include any `OUT` parameters, it can also be invoked using `CALL` in a tool that executes dynamic SQL statements, such as the `ij` tool.

A Java stored procedure that looks up the name of a city given a ZIP code might look like this (again, code for handling errors and exceptions is omitted for brevity):

**Listing 7. Sample stored procedure source code**

```java
public class ZipMap {
    public static void cityFromZipSp (int zipcode, String[] outCity) {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        String query = "SELECT city FROM zipmap WHERE zipcode = ?";
        PreparedStatement ps = conn.prepareStatement(query);
        ps.setInt(1, zipcode);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            outCity[0] = "Unknown City";
        else
            outCity[0] = rs.getString(1);
        rs.close();
        ps.close();
        conn.close();
        return;
    }
}
```

The stored procedure would be created as shown in Listing 8:

**Listing 8. Sample CREATE PROCEDURE**

```
CREATE PROCEDURE cityFromZip ( IN zipcode INTEGER, OUT city VARCHAR(255) )
PARAMETER STYLE JAVA
LANGUAGE JAVA
READS SQL DATA
EXTERNAL NAME 'com.acme.sales.ZipMap.cityFromZipSp'
```

As mentioned above, if a stored procedure includes `OUT` parameters, a client application executes the stored procedure using the `CallableStatement` method, as shown in Listing 9.

**Listing 9. Sample client calling stored procedure**

```java
public static String callOutCity(int zipCode, Connection con) {
```
```java
String outCity = "";
try {
    // prepare the CALL statement
    String procName = "CITYFROMZIP";
    String sql = "CALL " + procName + "(?, ?)";
    CallableStatement callStmt = con.prepareCall(sql);

    // Set the input parameter to the zipcode
    callStmt.setInt(1, zipCode);

    // Register the output parameter
    callStmt.registerOutParameter(2, Types.CHAR);

    // call the stored procedure
    System.out.println();
    System.out.println("Call stored procedure named " + procName);
    callStmt.execute();

    // retrieve output parameters
    outCity = callStmt.getString(2);
    System.out.println("City name is " + outCity);
} catch (SQLException e) {
    System.out.println(e);
} return(outCity);
}
```

### Loading Java classes from the database

The classes for SQL procedures and functions implemented in Java can be stored in the database along with the data, and the database can be programmed to load the classes from there. This makes the database data and its logic into a single, self-contained application package. This simplifies application deployment, because it reduces the potential for problems with a user’s class path.

Storage and loading of application classes in the database is provided by two mechanisms: the ability to add, remove, and replace JAR files in the database, and the ability to add these JAR files to the class path.

Once the application code has been added to the database, moving or copying the database ensures that the appropriate application logic is moved along with the data. This means that object data will never be separated from its methods.

### IBM Cloudscape features

The Derby software includes so many capabilities, what more could you possibly want?

The IBM Cloudscape commercial product ships Derby and additionally includes:

- Technical Support available for purchase
- IBM DB2 UDB Universal JDBC Driver for use with the Derby Network Server
- Translated manuals included with the IBM Cloudscape manuals
• The Cloudscape Information Center (see Related topics)
• Features to jump start developers who are new to Java and/or Derby, including:
  • Product installers for Windows and Linux that install the Java Runtime Environment (JRE)
  • Sample databases

Summary

In this article you learned about the fundamental architecture of Derby and how it can be embedded inside a client or a server application. You learned about the use of standard JDBC calls to manipulate the data. You received a summary of the security mechanisms available in Derby. You also learned how to take advantage of Java integration to create Java functions and stored procedures.

Derby provides developers with a small footprint, standards-based Java database that can be tightly embedded into any Java-based solution. With its combination of robust SQL features, support for Java technology, security, and embeddable, pure Java architecture, Derby is the data management product of choice for data-driven Java applications. The value that the IBM Cloudscape commercial offering adds makes it easier than ever for Java developers to deploy their Java-based solutions.
Related topics

- The IBM Cloudscape Information Center will provide you with in-depth details on Derby and Cloudscape.
- Read "Cloudscape frequently asked questions" (developerWorks, updated August 2005).
- Learn more about JDBC at http://java.sun.com/jdbc.
- "IBM Cloudscape: Understanding Java class path" (developerWorks, August 2004) helps you avoid class path problems for a trouble-free Cloudscape installation.
- Download IBM DB2 CLI/ODBC for Derby at no-charge.
- The Apache Derby site contains a wealth of information about Derby and hosts mail lists for developers and users.
- Learn about Application Development with DB2.

© Copyright IBM Corporation 2004, 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)