# IT 775
# Database Technology

## Data Stores

## Integrating Databases

# Connecting to MariaDB

Client Libraries
https://mariadb.com/kb/en/client-libraries/

The MariaDB project offers client libraries to enable users to connect from a given application to MariaDB and MySQL databases.

# MariaDB Client Libraries

Client/Server Protocol
https://mariadb.com/kb/en/embedded-mariadb-interface/

Protocol Used between Clients and the MariaDB Server.

libmysqld

https://mariadb.com/kb/en/libmysqld/

# MariaDB Client Libraries

## Embedded MariaDB interface

- The embedded MariaDB server, libmysqld has the identical interface as the C client library libmysqclient. The typical usage of the embedded server is to use the normal mysql.h include file in your application and link with libmysqld instead of libmysqlclient.
- The intention is that one should be able to move from a server/client version of MariaDB to a single server version of MariaDB by just changing which library you link with. This means that the embedded C client API only changes when the normal C API changes, usually only between major releases.

# MariaDB APIs

Application Programming Interfaces

https://mariadb.com/kb/en/connectors/

- C & C++ Connectors
- Java Connector
- .NET Connector
- Node.js Connector
- ODBC Connector
- Perl DBI
- PHP
- Python
- Ruby

# MariaDB APIs

## Application Programming Interfaces

- Other Connectors & Methods

  https://mariadb.com/kb/en/other-connectors-methods/

  - Excel Add-in for MariaDB
  - Perfect-MariaDB for Swift
  - RMariaDB: MariaDB Driver for R (RMariaDB is a database interface and MariaDB driver for R)

# MariaDB Python Connector

Python Connector -

https://mariadb.com/resources/blog/how-to-connect-python-programs-to-mariadb/

MariaDB Connector/Python enables python programs to access MariaDB and MySQL databases, using an API which is compliant with the Python DB API 2.0 (PEP-249). It is written in C and uses MariaDB Connector/C client library for client server communication.

# MariaDB Python Connector

You can use the popular programming language Python to manage data stored in MariaDB Platform, including MariaDB Server, MariaDB MaxScale and MariaDB SkySQL. Here is everything you need to know about connecting to MariaDB Platform from Python for retrieving, updating and inserting information.

To get the package (in Python environment):

```
$ pip3 install mariadb
```

# MariaDB Python Connector

Example to get connection and a cursor:

```
# Module Imports
import mariadb
import sys

# Connect to MariaDB Platform
try:
    conn = mariadb.connect(
        user="db_user",
        password="db_user_passwd",
        host="192.0.2.1",
        port=3306,
        database="employees"
    )
except mariadb.Error as e:
    print(f"Error connecting to MariaDB Platform: {e}")
    sys.exit(1)

# Get Cursor
cur = conn.cursor()
```

# MariaDB Python Connector

Example – Retrieving Data:

```
cur.execute(
    "SELECT first_name,last_name FROM employees WHERE first_name=?",
    (some_name,))
```

# MariaDB Python Connector

Example – Adding Data:

```
cursor.execute(
    "INSERT INTO employees (first_name,last_name) VALUES (?, ?)",
    (first_name, last_name))
```

# MariaDB Python Connector

Example – Printing Result:

The query results are stored in a list in the cursor object. To view the results, you can loop over the cursor. Each row is passed from the cursor as a tuple containing the columns in the SELECT statement.

```
# Print Result-set
for (first_name, last_name) in cur:
    print(f"First Name: {first_name}, Last Name: {last_name}")
```

# MariaDB Python Connector

Example – Exception Catching:

For any of your SQL actions (querying, updating, deleting, or inserting records) you should try to trap errors, so you can verify that your actions are being executed as expected and you know about any problems as they occur. To trap errors, use the Error class:

```
try:
    cursor.execute("some MariaDB query"))
except mariadb.Error as e:
   print(f"Error: {e}")
```

# MariaDB Python Connector

Example – Exception Catching:

If the query in the try clause of the code fails, MariaDB Server returns an SQL exception, which is caught in the except and printed to stdout. This programming best practice for catching exceptions is especially important when you're working with a database, because you need to ensure the integrity of the information.

# MariaDB Python Connector

## Example – Entire Script:

```
#!/usr/bin/python
import mariadb

conn = mariadb.connect(
    user="db_user",
    password="db_user_passwd",
    host="localhost",
    database="employees")
cur = conn.cursor()

#retrieving information
some_name = "Georgi"
cur.execute("SELECT first_name,last_name FROM employees WHERE first_name=?", (some_name,))

for first_name, last_name in cur:
    print(f"First name: {first_name}, Last name: {last_name}")

#insert information
try:
    cur.execute("INSERT INTO employees (first_name,last_name) VALUES (?, ?)", ("Maria","DB"))
except mariadb.Error as e:
    print(f"Error: {e}")

conn.commit()
print(f"Last Inserted ID: {cur.lastrowid}")

conn.close()
```

# MariaDB Python Connector

Example – Note:

By default, MariaDB Connector/Python enables auto-commit. If you would like to manually manage your transactions, only committing when you are ready, you can disable it by setting the autocommit attribute on the connection to False.

```
# Disable Auto-Commit
conn.autocommit = False
```

Once this is done, you can commit and rollback transactions using the commit() and rollback() methods.

# Open Database Connectivity (ODBC)

Definition:

https://en.wikipedia.org/wiki/Open_Database_Connectivity

Open Database Connectivity (ODBC) is a standard application programming interface (API) for accessing database management systems (DBMS). The designers of ODBC aimed to make it independent of database systems and operating systems. An application written using ODBC can be ported to other platforms, both on the client and server side, with few changes to the data access code.

# Open Database Connectivity (ODBC)

Definition:

ODBC accomplishes DBMS independence by using an ODBC driver as a translation layer between the application and the DBMS. The application uses ODBC functions through an ODBC driver manager with which it is linked, and the driver passes the query to the DBMS. An ODBC driver can be thought of as analogous to a printer driver or other driver, providing a standard set of functions for the application to use, and implementing DBMS-specific functionality.

# Open Database Connectivity (ODBC)

Definition:

An application that can use ODBC is referred to as "ODBC-compliant". Any ODBC-compliant application can access any DBMS for which a driver is installed. Drivers exist for all major DBMSs, many other data sources like address book systems and Microsoft Excel, and even for text or comma-separated values (CSV) files.

# Open Database Connectivity (ODBC)

JDBC:

Java Database Connectivity (JDBC). In most ways, JDBC can be considered a version of ODBC for the programming language Java instead of C. JDBC-to-ODBC bridges allow Java-based programs to access data sources through ODBC drivers on platforms lacking a native JDBC driver, although these are now relatively rare. Inversely, ODBC-to-JDBC bridges allow C-based programs to access data sources through JDBC drivers on platforms or from databases lacking suitable ODBC drivers.

# Open Database Connectivity (ODBC)

Current State:

ODBC remains in wide use today, with drivers available for most platforms and most databases. It is not uncommon to find ODBC drivers for database engines that are meant to be embedded, like SQLite, as a way to allow existing tools to act as front-ends to these engines for testing and debugging.

# Open Database Connectivity (ODBC)

Current State:

However, the rise of thin client computing using HTML as an intermediate format has reduced the need for ODBC. Many web development platforms contain direct links to target databases – MySQL being very common. In these scenarios, there is no direct client-side access nor multiple client software systems to support; everything goes through the programmer-supplied HTML application.

# Open Database Connectivity (ODBC)

Current State:

The virtualization that ODBC offers is no longer a strong requirement, and development of ODBC is no longer as active as it once was. While ODBC is no longer a strong requirement for client-server programming, it is now more important for access, virtualization, and integration in analytics and data science scenarios. These new requirements are reflected in new ODBC 4.0 features such as semi-structured and hierarchical data, web authentication and performance improvement.

# Data Access Layer (DAL)

Definition:

https://en.wikipedia.org/wiki/Data_access_layer

A data access layer (DAL) is a layer of a computer program which provides simplified access to data stored in persistent storage of some kind, such as an entity-relational database.

For example, the DAL might return a reference to an object (in terms of o-o programming) complete with its attributes instead of a row of fields from a database table. This allows the client (or user) modules to be created with a higher level of abstraction. This model could be implemented by creating a class of data access methods that directly reference a corresponding set of database stored procedures. The DAL hides this complexity of the underlying data store from the external world.

# Data Access Layer (DAL)

Implementation:

For example, instead of using commands such as insert, delete, and update to access a specific table in a database, a class and a few stored procedures could be created in the database. The procedures would be called from a method inside the class, which would return an object containing the requested values. Or, the insert, delete and update commands could be executed within simple functions like registeruser or loginuser stored within the data access layer.

Also, business logic methods from an application can be mapped to the Data Access Layer. So, for example, instead of making a query into a database to fetch all users from several tables, the application can call a single method from a DAL which abstracts those database calls.

# Data Access Layer (DAL)

Note:

Applications using a data access layer can be either database server dependent or independent. If the data access layer supports multiple database types, the application becomes able to use whatever databases the DAL can talk to. In either circumstance, having a data access layer provides a centralized location for all calls into the database, and thus makes it easier to port the application to other database systems (assuming that 100% of the database interaction is done in the DAL for a given application).

Object-Relational Mapping tools provide data layers in this fashion, following the Active Record or Data Mapper patterns. The ORM/active-record model is popular with web frameworks.

# Database Abstraction Layer (DBAL)

Definition:

https://en.wikipedia.org/wiki/Database_abstraction_layer

A database abstraction layer (DBAL or DAL) is an application programming interface which unifies the communication between a computer application and databases. Traditionally, all database vendors provide their own interface that is tailored to their products. It is up to the application programmer to implement code for the database interfaces that will be supported by the application. Database abstraction layers reduce the amount of work by providing a consistent API to the developer and hide the database specifics behind this interface as much as possible. There exist many abstraction layers with different interfaces in numerous programming languages. If an application has such a layer built in, it is called database-agnostic.

# Database Abstraction Layer (DBAL)

## Levels of Abstraction - Physical:

The lowest level connects to the database and performs the actual operations required by the users. At this level the conceptual instruction has been translated into multiple instructions that the database understands. Executing the instructions in the correct order allows the DAL to perform the conceptual instruction.

Implementation of the physical layer may use database-specific APIs or use the underlying language standard database access technology and the database's version SQL.

Implementation of data types and operations are the most database-specific at this level.

# Database Abstraction Layer (DBAL)

## Levels of Abstraction – Conceptual or Logical:

The conceptual level consolidates external concepts and instructions into an intermediate data structure that can be devolved into physical instructions. This layer is the most complex as it spans the external and physical levels. Additionally it needs to span all the supported databases and their quirks, APIs, and problems.

This level is aware of the differences between the databases and able to construct an execution path of operations in all cases. However the conceptual layer defers to the physical layer for the actual implementation of each individual operation.

# Database Abstraction Layer (DBAL)

## Levels of Abstraction – External or View:

The external level is exposed to users and developers and supplies a consistent pattern for performing database operations. Database operations are represented only loosely as SQL or even database access at this level.

Every database should be treated equally at this level with no apparent difference despite varying physical data types and operations.

# Database Abstraction Layer (DBAL)

Abstraction in the API:

Libraries unify access to databases by providing a single low-level programming interface to the application developer. Their advantages are most often speed and flexibility because they are not tied to a specific query language (subset) and only have to implement a thin layer to reach their goal. As all SQL dialects are similar to one another, application developers can use all the language features, possibly providing configurable elements for database-specific cases, such as typically user-IDs and credentials. A thin-layer allows the same queries and statements to run on a variety of database products with negligible overhead.

# Database Abstraction Layer (DBAL)

Abstraction in the API:

Popular use for database abstraction layers are among object-oriented programming languages, which are similar to API-level abstraction layers. In an object-oriented language like C++ or Java, a database can be represented through an object, whose methods and members (or the equivalent thereof in other programming languages) represent various functionalities of the database. They also share advantages and disadvantages with API-level interfaces.

# Database Abstraction Layer (DBAL)

## Language-level Abstraction:

An example of a database abstraction layer on the language level would be ODBC that is a platform-independent implementation of a database abstraction layer. The user installs specific driver software, through which ODBC can communicate with a database or set of databases. The user then has the ability to have programs communicate with ODBC, which then relays the results back and forth between the user programs and the database. The downside of this abstraction level is the increased overhead to transform statements into constructs understood by the target database.Alternatively, there are thin wrappers, often described as lightweight abstraction layers, such as OpenDBX and libzdb. Finally, large projects may develop their own libraries, such as, for example, libgda for GNOME.

# Database Abstraction Layer (DBAL)

Pros:

- Development: Software developers only have to know the database abstraction layer's API instead of all APIs of the databases their application should support. The more databases should be supported the bigger is the time saving.
- Wider potential install-base: using a database abstraction layer means that there is no requirement for new installations to utilise a specific database, i.e. new users who are unwilling or unable to switch databases can deploy on their existing infrastructure.
- Future-proofing: as new database technologies emerge, software developers won't have to adapt to new interfaces.
- Testing: a production database may be replaced with a desktop-level implementation of the data for developer-level unit tests.
- Added Database Features: depending on the database and the DAL, it may be possible for the DAL to add features to the database.

# Database Abstraction Layer (DBAL)

## Cons:

- Speed: any abstraction layer will reduce the overall speed more or less depending on the amount of additional code that has to be executed. The more a database layer abstracts from the native database interface and tries to emulate features not present on all database backends, the slower the overall performance. This is especially true for database abstraction layers that try to unify the query language as well like ODBC.
- Dependency: a database abstraction layer provides yet another functional dependency for a software system, i.e. a given database abstraction layer, like anything else, may become obsolete, outmoded or unsupported.
- Masked operations: database abstraction layers may limit the number of available database operations to a subset of those supported by the supported database backends. In particular, database abstraction layers may not fully support database backend-specific optimizations or debugging features. These problems magnify significantly with database size, scale, and complexity.