Database Access via Programming Languages

5DV052 — Advanced Data Models and Systems

Umeå University
Department of Computing Science

Stephen J. Hegner
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Spring 2011

The Limitations of Stand-Alone SQL

- SQL is primarily a language for data definition, retrieval, and update.
- It is not designed for complex computation.
- Enhancements such as OLAP are useful for certain specific tasks, but still leave many important tasks difficult or impossible to achieve.

Theoretical shortcoming: Unlike most programming languages, it is not *Turing complete*.

- There are computations which cannot be expressed in SQL at all.

Interoperability shortcoming: Stand-alone SQL clients are generally vendor specific.

- Concurrent access to databases of different vendors is not possible with a single client.
- Access to multiple databases via the same client is usually awkward, requiring vendor-specific directives.

The Limitations of Stand-Alone SQL: 2

Practical shortcomings: There is also a host of practical reasons why stand-alone SQL does not suffice:

Accessibility: Most users of databases are not computer scientists.

- They need a custom interface for non-experts.
- Even experts can often work more effectively via custom interfaces.

Simplicity: Real-world database schemata are often very large and complex.

- Users often need to work with custom views which present what they need to know and are allowed to know.

Security: The correct management of access rights is a very complex task.

- It is often easier to manage access by admitting access via specific interfaces.

Concurrency: The correct management of concurrent processes is also very complex.

- It is often easier to manage concurrency via properly designed interfaces.

Database Access via Programming Languages: Desiderata

Database access via standard SQL: Ça va sans dire !

Use with:
- traditional programming languages: C, C++, Java, Python.
- languages for Web-based access: PHP, via Apache Tomcat.

Interoperability: Access to several different databases, running the systems of many different vendors, perhaps on different platforms.

The Major Players in the DBMS arena:

The "big three" commercial systems:
- Oracle Database
- IBM DB2
- Microsoft SQL Server

The major open-source systems:
- PostgreSQL
- MySQL/InnoDB

Other significant commercial vendors: Mimer SQL, Sybase

Other products with widespread usage: M$ Access

Examples of Vendor-Specific Solutions

Oracle PL/SQL: A proprietary PL/1-like imperative programming language which supports the execution of SQL queries.

Advantages:

- Many Oracle-specific features of SQL and the Oracle Database systems are supported.
- Performance may be optimized in a manner not achievable with solutions which are not vendor specific.

Disadvantages:

- Vendor lock-in: applications are tied to a specific DBMS.
- Application development is dependent upon the existence of a development environment for the language (in this case, PL/1), which may not be available on all platforms.
- Big problems arise if the vendor goes out of business or chooses to stop supporting a given platform (*e.g.*, Linux).

- VBA + MS Access under Microsoft Windows is an even stronger vendor-specific example in the desktop environment.

Embedded SQL: a Vendor-Independent Solution

- In embedded SQL, calls to SQL statements are embedded in the host programming language.
- Typically, such statements are tagged by a special marker, usually `EXEC SQL`.
- A preprocessor is invoked to convert the source program into a "pure" program in the hose language.
- The `EXEC SQL` statements are convert to statements in the host language via a preprocessor.
- In *static* embedded SQL, table and attribute names must be declared in the source program.
- In *dynamic* embedded SQL, they may be provided at run time.
- There is an ISO standard for embedded SQL.

Disadvantages of Embedded SQL

Embedded SQL has a number of distinct disadvantages:

Preprocessed: Debugging preprocessed programs is not a pleasant experience.

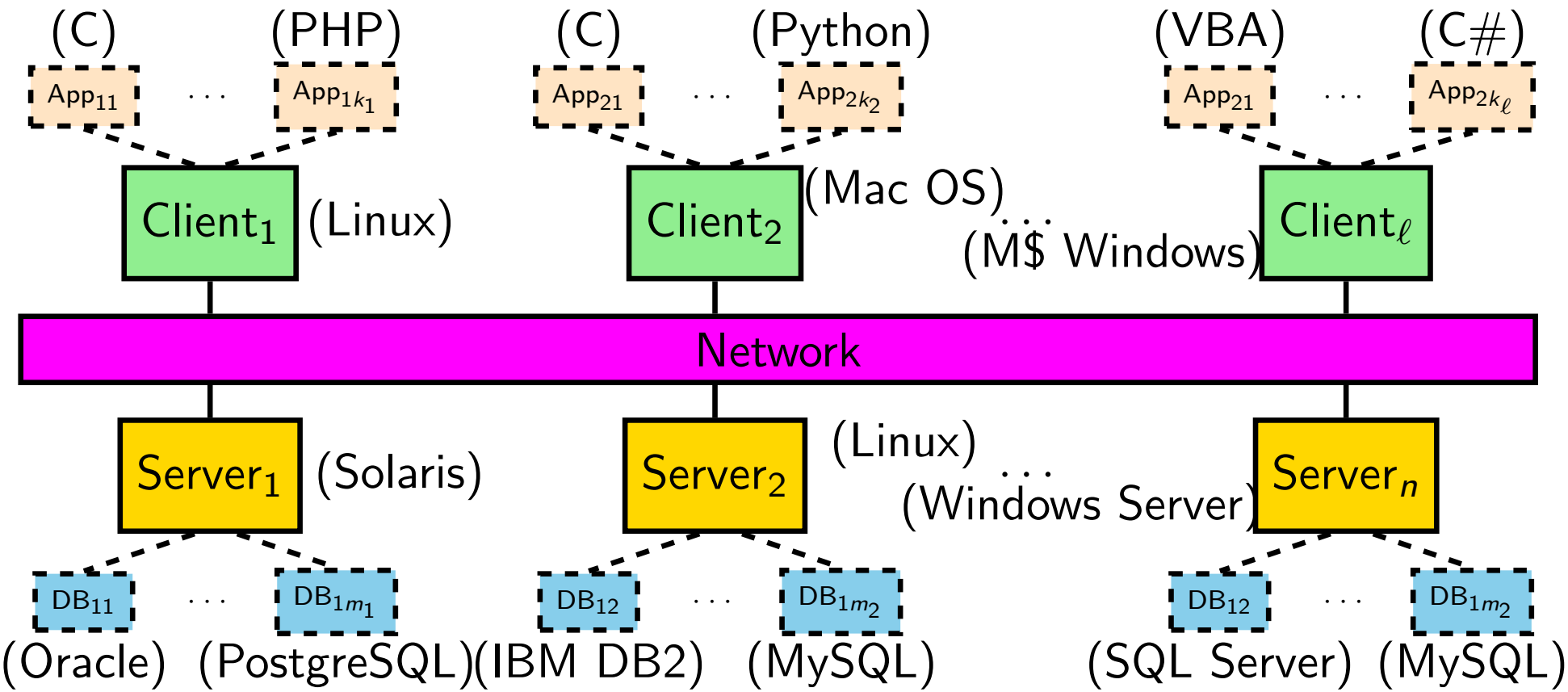
Program development environment: Because of the nature of preprocessed programs, it is not easy to provide support for the preprocessor directives within a programming environment.

Specificity: The preprocessor must be vendor specific, and at least in part, platform specific as well.

- Embedded SQL has been superseded in large part by CLI/ODBC.

A Closer Look at Interoperability

- A "real-world" situation might involve several DBMS, OSs, and PLs.
- The scenario might look something like this:



- In the ideal case, any application should be able to access any database using SQL ... subject only to limitations imposed by access rights.

The CLI/ODBC Solution to Interoperability

- There are two closely related specifications.

CLI (Call-Level Interface): An ISO/IEC standard developed in the early 1990s.

- Defined only for C and COBOL.

ODBC (Open Data Base Connectivity): A specification based upon CLI.

- Defined for many programming languages, including C, Python, Ruby, and PHP.

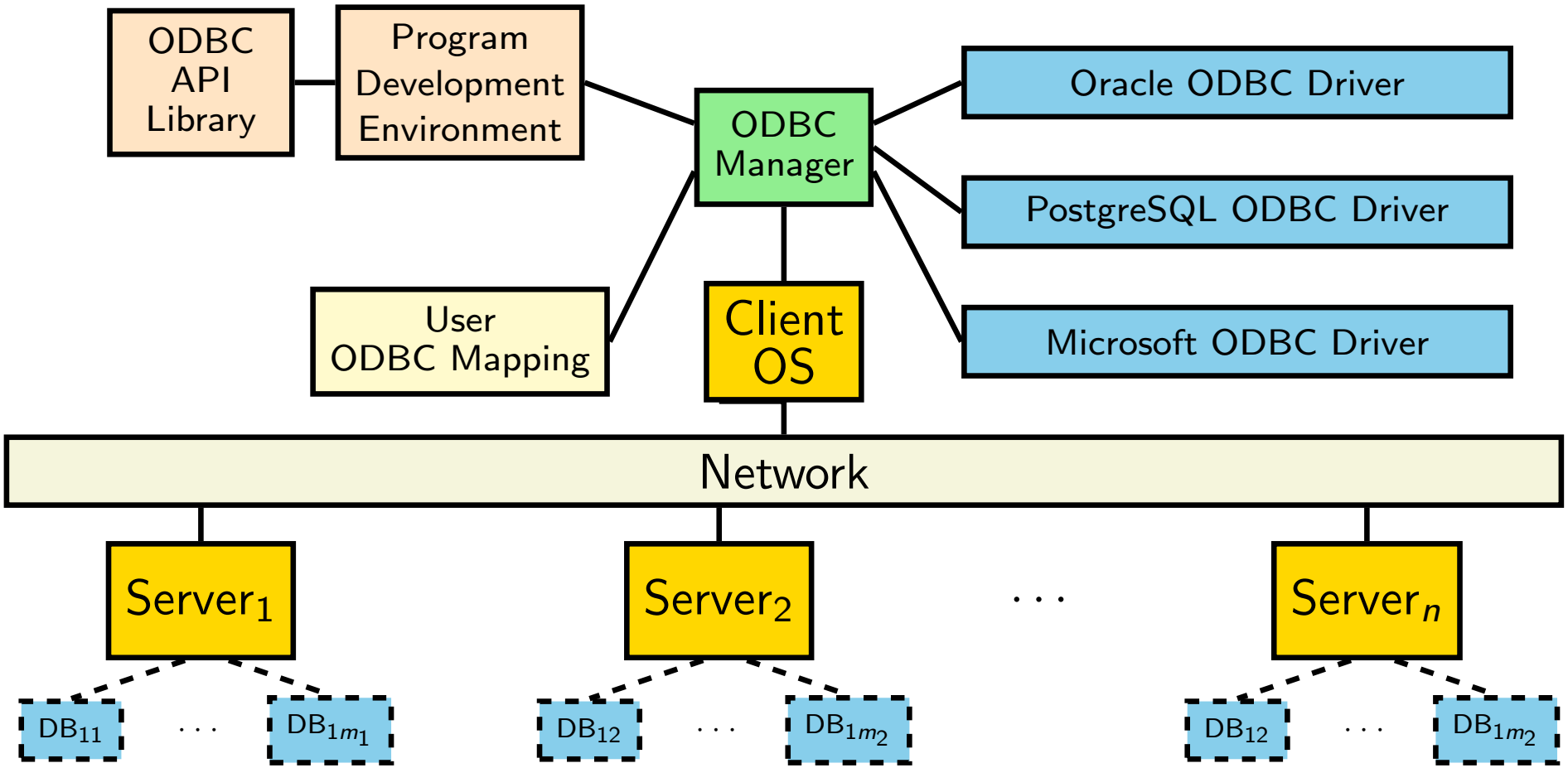
JDBC: An ODBC-like specification for Java.

- All of these solutions exhibit a large degree of interoperability.

ODBC is not platform, OS, or DBMS specific.
OS: Unix, Linux, MacOS, MS Windows, IBM
DBMS: You name it.

- Interestingly, the major player which promoted ODBC was ... Microsoft!

The Architecture of ODBC for a Single Client



Color code:

Operating system | OS-specific utility | DBMS-specific module

Development-environment-specific module | User configuration file

Using ODBC in the Linux Environment

- The main ideas are presented via a set of five annotated programs in C.
- These slides provide only supporting information.
- The specific context is the gcc compiler under Linux and Solaris.
- Although these examples are intended to be generic, they have been tested only with the PostgreSQL DBMS, using Debian Linux and Sun/Oracle Solaris as the client-side operating system.
- The on-line reference manual for ODBC may be found on a Microsoft Web site. Use your favorite search engine to find it.
- Good books on ODBC are difficult to find.
- These notes and the example programs will provide enough to do the ODBC laboratory exercise.

Compiling a C Program with ODBC Calls

- One of the ODBC client-side ODBC C libraries must be included.
- One of the following should work to compile `program.c`.:

```
cc -lodbc program.c
cc -liodbc program.c
```

- Although they are functionally equivalent, the libraries `odbc` and `iodbc` cannot co-exist on the same system.
- Try one; if a list of error messages appears, try the other.
- There may be some small differences between the two which may require changes in `scanf` and `printf` formats.
- Currently, use `iodbc` on the systems of the department.
- The actual structure of C programs which contain ODBC calls is illustrated via accompanying example programs, with some basic principles discussed in these slides.

Data-Source Configuration Under the Linux Installation

- Every data source which is to be reached via ODBC calls must be declared in the `.odbc.ini` file in the home directory of the user.
- A minimal example file is shown below for connection to PostgreSQL databases on the `postgres` server using Linux.

```
# The ODBC data source names are are not used by PostgreSQL.
[ODBC Data Sources]
mydb1 = database1
mydb2 = database2
# The name in square brackets is the ODBC DB name; may be chosen arbitrarily.
[database1]
Description = PostgreSQL test database 1
Driver      = /usr/lib/odbc/psqlodbca.so
# The name on the next line is the PostgreSQL DB name.
Database    = hegner1
Servername  = postgres
# Here is the definition of a second database.
[database2]
Description = PostgreSQL test database 2
Driver      = /usr/lib/odbc/psqlodbca.so
Database    = hegner2
Servername  = postgres
```

A More Complete Specification of a Data Source

```
[ODBC Data Sources]
mydb3 = database3
[database3]
Description = PostgreSQL test database 1
Driver      = /usr/lib/odbc/psqlodbca.so
Database    = hegner1
Servername  = postgres
Port        = 5432
ReadOnly    = 0
Username    = hegner1
Password    = "badidea"
Trace       = No
TraceFile   = /tmp/odbc.log
```

- The fields not shown on the previous slide are optional.
- `Port` and `ReadOnly` need be specified only if they differ from the defaults.
- `Trace` and `Tracefile` need only be specified if tracing is desired.
- The `UserName` and `Password` fields are not used in the access methods used in this course. .

Variations for the Solaris Installation

- Under the Solaris installation, use the `unixodbc` library, and specify `gcc` and the load path explicitly.

```
gcc -L/usr/local/lib -lodbc program.c
```

- Also, the driver is in a different location.

```
[ODBC Data Sources]
mydb1 = database1
mydb2 = database2
[database1]
Description = PostgreSQL test database 1
Driver      = /usr/local/lib/psqlodbc.so
Database    = hegner1
Servername  = postgres
[database2]
Description = PostgreSQL test database 2
Driver      = /usr/local/lib/psqlodbc.so
Database    = hegner2
Servername  = postgres
```

Combining Linux and Solaris in one `.odbc.ini` file

- To use a single .odbc.ini file for both Linux and Solaris, a simple solution is to use different ODBC names.

```
[ODBC Data Sources]
mydb1Solaris = database1S
mydb1Linux = database1L
[database1S]
Description = PostgreSQL test database 1
Driver      = /usr/local/lib/psqlodbc.so
Database    = hegner1
Servername  = postgres
[database1L]
Description = PostgreSQL test database 2
Driver      = /usr/lib/odbc/psqlodbca.so
Database    = hegner1
Servername  = postgres
```

- This requires using a different ODBC name, for the same database, when using Linux access than for Solaris.

Some Basics of ODBC Calls in C

Identifiers:

- Most ODBC identifiers begin with `SQL` (uppercase letters).
- It is therefore good practice to avoid using this sequence of characters as the beginning of user-defined identifiers.

API calls:

- ODBC contains a large number of functions (around 80).
- They have names such as `SQLAllocHandle` and `SQLCloseCursor`.
- Only a few will be used in this course.
- Most (all?) return a value of type `SQLReturn`.
- The returned value is zero if the the execution was normal, and nonzero if it was not.

Includes: To use ODBC API calls, the following two includes must be in the program header:

```
#include <sql.h>
#include <sqlext.h>
```

Matching SQL to Host-Language Data Structures

- SQL and the host language (in this case `C`) each have their own data types.
- To use ODBC, there must be a mechanism for translation between these types.
- To effect this, the primitive types which occur in SQL are assigned corresponding types in the host language.
- The definitions are found in the header file `sqltypes.h`, which is loaded by `sql.h`.
- A table of some of the principal associations is shown on the next slide.
- For API calls, these types, rather than the associated types of `C`, should be used.

The Principal ODBC-C Data-Type Associations

- Some of the most commonly used associations are shown below.

ODBC Type	C Type
SQLCHAR	char
SQLSCHAR	signed char
SQLINTEGER	long int
SQLUINTEGER	unsigned long int
SQLSMALLINT	short int
SQLUSMALLINT	unsigned short int
SQLREAL	float
SQLDOUBLE,SQLFLOAT	double
SQLDATE	a large struct

- There are many others (*e.g.*, for time).
- The exact associations may vary from system to system.
- Therefore, for API calls, the ODBC types, rather than the C types, should be used.

Integer Encodings of ODBC Data Types

- Each ODBC type has an integer encoding.
- These encodings are used in the arguments to API calls, to indicate which data type is to be used.
- Each encoding also has a symbolic name, so the programmer need not know (and should not use) the actual integer.
- The associations of numbers to symbolic names are found in `sqlext.h`.
- A table of some of the most common ones is shown below.

ODBC Type	Name for Integer Encoding
SQLCHAR	SQL_C_CHAR
SQLSCHAR	SQL_C_STINYINT
SQLINTEGER	SQL_C_SLONG
SQLUINTEGER	SQL_C_ULONG
SQLSMALLINT	SQL_C_SSHORT
SQLUSMALLINT	SQL_C_USHORT
SQLREAL	SQL_C_FLOAT
SQLDOUBLE,SQLFLOAT	SQL_C_DOUBLE
SQLDATE	SQL_C_TYPE_DATE

- These need not be remembered; only the concept is important.

Integer Encodings for SQL Data Types

- There is also an integer encoding (and an associated symbolic identifier) for each SQL type.
- These associations are found in sqlext.h.
- The table below gives only typical associations, found in the on-line documentation for ODBC.

SQL Type	Name for Integer Encoding
char(n)	SQL_CHAR
varchar(n)	SQL_VARCHAR
smallint	SQL_SMALLINT
integer	SQL_INTEGER
real	SQL_REAL
numeric(p,s)	SQL_DECIMAL
decimal(p,s)	SQL_NUMERIC
date	SQL_TYPE_DATE

- Consult the local documentation for exact usage.

ODBC Handles

- A *handle* is a numerical value with is associated with a certain object.
- File handles are familiar in systems programming.
- In ODBC, there are four types of handles.

  Environment handles: In order to access a database via ODBC, an ODBC environment must be established.
    - There is normally only one such environment per program.

  Connection handles: Just as there must be a file handle for every open file in an operating system, so too must there be a connection handle for each connection to an ODBC database.

  Statement handles: A statement handle is associated with an SQL statement which is to be issued to a database for execution.

  Descriptor handles: Descriptors are metadata which describe formats associated with SQL statements.
    - Descriptor handles will not be used in this course.

Declaration and Use of ODBC Handles

- Handles are declared using the type `SQLHandle`.
- Handles are allocated using the function `SQLAllocateHandle`.
- Handles are freed using the function `SQLFreeHandle`.
- These are all illustrated in the example programs.
- The current version of ODBC is 3.xx.
- The (much) older version 2.xx used a different syntax for handles.
- That syntax is deprecated and should not be used, even though it may still work.

Other Key ODBC Directives

- The following are just indicators of the key operations in ODBC.
- All are illustrated in the example programs.

`SQLPrepare` Prepare ("compile") an SQL statement for execution.

`SQLExecute` Executed a compiled SQL statement.

`SQLExecDirect` Compile and then execute an SQL statement in one step.

- Appropriate in situations in which an SQL statement is to be executed only once.

`SQLBindParameter` Bind an input-parameter index in an SQL statement to a variable in the program.

`SQLBindCol` Bind a column of a query result (output parameter) to a variable in the program.

`SQLFetchTuple` Fetch the next tuple from the result of a query.

`SQLCloseCursor` Close the cursor on a given query, so that the statement handle may be used to collect the results of a new query.

Other Classes of API Calls

- A few other major classes of ODBC API calls are the following.

  Catalog queries: Find out which relations are in a given database, what the types of the columns are, what the constraints are, and so forth.

  Optimization directives: Process large sets of queries with efficient batch operations.

  Error management: If something goes wrong, find out what the problem is.

- In all, there are over 80 API calls in ODBC.