Lecture #12: Networking Protocols

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Database Access

Applications/users can interact in various ways with a Database Management System. For example, one approach is to interact with the Database Management System via a command line interface (CLI). However, in real-world scenarios, applications must pull data from the database management system and process it in the application code. Therefore, a CLI interface is not suitable for such cases. There are three broad categories of Access Method APIs. These include *Open Database Connectivity (ODBC)*, *Java Database Connectivity (JDBC)*, and *Direct Access (DBMS-Specific, low-level C library)*.

In practice, people use ODBC and JDBC since they are agnostic to the database management system. Therefore, the logic in the application code does not have to change if the database management system is changed.

Open Database Connectivity (ODBC)

ODBC was developed in the early 1990s by Microsoft and Simba Technologies in an effort to create one more level of abstraction when interacting with the DBMS. It was mainly used for C/C++ applications. Nowadays, ODBC is the standard API for accessing a DBMS, and it is agnostic to the DBMS and the OS. Every DBMS provides an implementation of ODBC for various programming languages.

Before ODBC, the application code was written using the library the DBMS (Direct Access) exposed. For example, in a scenario where Postgresql is used, the application code had to use the libpq library exposed by this specific DBMS (Postgresql). When changing the DBMS, the application code had to be rewritten using the proprietary library exposed by the new system (e.g., MySQL). Therefore, the ODBC was created to hide all these problems.

ODBC is based on the device *driver* model. The database system vendor is responsible for providing a driver based on the ODBC specification to interact with the DBMS. Essentially, an ODBC driver (e.g., in Postgres) will use the libpq to be implemented.

The ODBC driver is also responsible for delivering results to the client in the correct format. For example, the database server and client might be running on CPUs with different architectures, and endianness might be a issue. The ODBC driver is responsible for handling scenarios like those.

The ODBC spec might support various features that the DBMS might not support inherently (e.g. Postgres does not support cursors). Therefore, it is also responsible for emulating that behavior.

Java Database Connectivity (JDBC)

Developed by Sun Microsystems in 1997, JDBC provides a standard API for connecting a Java program with a DBMS. At that time, ODBC was Windows-specific and used in C/C++ applications. Similar to ODBC, JDBC exposes a standard set of APIs and uses drivers to communicate with the DBMS.

There are four different approaches to implementing JDBC driver:

- **JDBC-ODBC Bridge**: A wrapper that translated JDBC calls to ODBC calls. This approach is deprecated nowadays.
- **Native-API Driver**: Converts JDBC method calls into native C calls via JNI.
- **Network-Protocol Driver**: The driver connects to a middleware in a separate process that converts JDBC calls into a vendor-specific DBMS protocol.
- **Database Protocol Driver**: Pure Java implementation that converts JDBC calls directly into a vendor-specific DBMS protocol. This is the most common implementation nowadays.

2 Database Networking Protocols (Wire Protocol)

The wire protocol is unique to each database management system and defines how data should be transmitted over the network between the DBMS and the client. It is implemented over TCP/IP. Essentially, ODBC/JDBC drivers hide the complexity of the wire protocol.

Client/Server Interaction

The client connects to the database management system via ODBC/JDBC, and after the authentication process, the client can send queries to the DBMS. After executing the query, the DBMS has to serialize the results and send them back to the client via the network. The serialization process is expensive and is considered a bottleneck [3]. The serialization is part of the wire protocol and has an extensive design space.

Existing Protocols

Many new systems use one of the existing open-source wire protocols. The most widely popular open-source protocols are Postgres, MySQL, and Redis.

Just using an open-source wire protocol (e.g., Postgres) does not make the DBMS compatible with Postgres. The new DBMS should support catalogs, have the same SQL dialect, etc.

3 Protocol Design Space

Different design choices can be made to improve the wire protocol [3].

Row vs. Column Layout

ODBC and JDBC are inherently row-oriented APIs. The DBMS packages tuples into messages one tuple at a time. Therefore, the client has to deserialize data one tuple at a time.

However, modern data analysis software (e.g., Tensorflow, **Spark**) operates on matrices and columns. Serializing/Deserializing one tuple at a time makes it inefficient for OLAP queries.

To improve performance, one potential solution is send data in vectors via the ADBC(Arrow Database Connectivity) which uses PAX layout (insted of row-store). PAX is a hybrid format which is used extensively in modern file formats and in the Arrow Project.

Compression

Client drivers are conservative since they have to implement the features that the DBMS will support. For example, if the DBMS sends the data compressed, then the driver must support decompression. Moreover, when using different programming languages, it is preferable to use a native implementation of ODBC instead of implementing ODBC by calling C functions (which increases the overhead). Therefore, if the wire protocol has specific features, then every ODBC driver for every programming language has to implement these features, which increases the complexity.

There are two main approaches for compression in OLAP systems:

- **Naive Compression**: The DBMS can apply general-purpose compression algorithms (lz4, gzip, zstd). One advantage of using the general purpose compression algorithms is that they are agnostic to the data format (row vs column vs PAX). Moreover, from an engineering point of view, the client can use an existing library to decompress. However, one disadvantage of general-purpose compression algorithms is their slower decompression speed (compared to columnar-specific encodings). Since networks are getting fast, using a heavy-weight compression algorithm can become the new bottleneck.
- **Columnar-Specific Encoding**: These encodings leverage the fact that data is stored in a columnar format, and each value has the same data type. The DBMS can use different encodings (Delta, Dictionary, RLE, Frame of Reference) to compress the data in these cases. The columnar-specific encodings sacrifice compression ratio for decompression speed.

Data Serialization

Data are serialized into packets to be transferred on the wire. There are two ways to serialize the data:

- **Binary Encoding**:
  Represent data in its binary form. The DBMS can implement its own binary encoding format or rely on existing libraries (e.g., ProtoBuf, Thrift). The closer the serialized format is to the DBMS's binary format, the lower the overhead to serialize. Since endianness can differ between the DBMS and the client, the client has to handle endian conversion as well.
- **Text Encoding**: Convert all binary values into a string(ASCII/UTF8). Since everything is a string, the client does not have to worry about endianness. The downside of the method is more data storage and does not leave much room for optimizations.

String Handling

There are three approaches to handling strings for the DBMS and driver:

- **Null Termination**: Store a null byte ('\0') to denote the end of a string. Client has to scan entire string to find the end of the string. Therefore jumping to the next value is impossible without more metadata.
- **Length Prefixes**: Add the length of the string at the beginning of the string (similar to Strings in Java). The DBMS needs additional bytes to store the length. However, skipping values by jumpoing to the next string is easier.
- **Fixed Width**: Pad every string to be the max size of the attribute.

There is no particular string handling approach that stands out above the others; all three approaches perform differently in different settings.

4 Kernel Bypass

Implementing the DBMS network protocol is one of many sources of slowdown but not the only one. A lot of overhead and cycles are spent in the TCP/IP stack inside the OS's kernel. The interference with the TCP/IP stack slows the DBMS since the networking stack makes expensive interrupts and spends a lot of cycles for context switching. Moreover, when sending data to the client, the operating system has to copy the data to the NIC. However, it should first copy the data from the DBMS to its internal buffers. Additionally, other threads that are running in the OS have to be scheduled as well, and the internal data structures inside

the OS have to be appropriately locked, which adds significant overhead in the critical path of sending data from the DBMS to the client.

The philosophy of database management systems is to interact with the operating system as little as possible.

Therefore, some DBMS try to avoid the OS. We use kernel bypass methods to allows the system to get data directly from the Network Interface Controller (NIC) into the DBMS address space. This saves the need for data copying or OS TCP/IP stack since the DBMS will be talking directly to the hardware layer. There are two kernel bypass methods:

Data Plane Development Kit (DPDK)

DPDK is a set of libraries that allows programs to access NIC directly. It treats the NIC as a bare metal device. This approach requires the DBMS code to do more to manage memory and buffers. An example of a DBMS that uses this is **ScyllaDB**.

Remote Direct Memory Access (RDMA)

RDMA libraries allow reading and writing memory directly on a remote host without going through the OS [2]. The client needs to know the correct address of the data that it wants to access; the server is unaware that memory is being accessed remotely. **Oracle RAC** and **Microsoft FaRM** are DBMSs that use RDMA.

IO_URING

Linux system call interface for zero-copy asynchronous I/O operations.

5 User Bypass

Another approach is to push the DBMS logic into the kernel to avoid copying data from kernel space to user space [1]. Kernel modules are one approach to extending the kernel. The downside is that if a kernel module crashes, the whole OS will crash. In addition, in many environments, loading a kernel module is not allowed due to security reasons. The solution to this problem is eBPF. eBPF allows the user to write high-level code that gets compiled to eBPF IR and verified (therefore is safe). Code verified by eBPF can be loaded on the fly as if it were a kernel module. The safety comes from the limited API that eBPF provides(no malloc, restricted number of instructions, etc.).

References

[1] M. Butrovich, K. Ramanathan, J. Rollinson, W. S. Lim, W. Zhang, J. Sherry, and A. Pavlo. Tigger: A database proxy that bounces with user-bypass. *Proc. VLDB Endow.*, 16(11):3335–3348, jul 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611530. URL https://doi.org/10.14778/3611479.3611530.

[2] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 355–370, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2882949. URL https://doi.org/10.1145/2882903.2882949.

[3] M. Raasveldt and H. Muhleisen. Don't hold my data hostage: a case for client protocol redesign. In *Proceedings of the VLDB Endowment*, volume 10, pages 1022–1033, June 2017. doi: https://dl.acm.org/citation.cfm?id=3115408.