LAST CLASS

How to inline user-defined functions into a query so that the DBMS's optimizer can understand its behavior and intention.

→ Pushing application logic into the DBMS.
TODAY'S AGENDA

Database Access APIs
Database Network Protocols
Kernel/User Bypass Methods
Client-side Optimizations
DATABASE ACCESS

All the demos in the class have been through a terminal client.
→ SQL queries are written by hand.
→ Results are printed to the terminal.

Real programs access a database through an API:
→ Direct Access (DBMS-specific)
→ Open Database Connectivity (ODBC)
→ Java Database Connectivity (JDBC)
→ Python PEP-0249
→ HTTP / REST (DBMS-specific)
DATABASE ACCESS

All the demos in the class have been through a terminal client:

- SQL queries are written by hand.
- Results are printed to the terminal.

Real programs access a database through an API:

- Direct Access (DBMS-specific)
- Open Database Connectivity (ODBC)
- Java Database Connectivity (JDBC)
- Python PEP-0249
- HTTP/REST (DBMS-specific)
DATABASE

Chapter 34. libpq — C Library

Table of Contents:
34.1 Database Connection Control Functions
34.1.1 Connection Strings
34.1.2 Parameter Key Words
34.2 Connection Status Function
34.3 Command Execution Functions
34.3.1 Main Functions
34.3.2 Preparing Query Result Information
34.3.3 Fetching Other Result Information
34.3.4 Escaping Strings for Inclusion in SQL Commands
34.4 Asynchronous Command Processing
34.5 Pipeline Mode
34.5.1 Using Pipeline Mode
34.5.2 Functions Associated with Pipeline Mode
34.5.3 When to Use Pipeline Mode
34.6 Retrieving Query Results Row-by-Row
34.7 Executing Queries in Progress
34.8 The Fast Path Interface
34.9 Nonsynchronous Invocation
34.10 Functions Associated with the CPQY Command
34.10.1 Functions for Sending CPQY Data
34.10.2 Functions for Receiving CPQY Data
34.10.3 Obsolete functions for CPQY
34.11 Coercion Functions
34.12 Processive Functions
34.13 Notice Processing
34.14 Event Splits
34.14.1 Event Types

DATABASE ACCESS

All the demos in the class have been through a terminal client.

→ SQL queries are written by hand.
→ Results are printed to the terminal.

Real programs access a database through an API:

→ Direct Access (DBMS-specific)
→ Open Database Connectivity (ODBC)
→ Java Database Connectivity (JDBC)
→ Python PEP-0249
→ HTTP/REST (DBMS-specific)
DATABASE ACCESS

All the demos in the class have been through a terminal client.
→ SQL queries are written by hand.
→ Results are printed to the terminal.

Real programs access a database through an API:
→ Direct Access (DBMS-specific)
→ Open Database Connectivity (ODBC)
→ Java Database Connectivity (JDBC)
→ Python PEP-0249
→ HTTP / REST (DBMS-specific)
OPEN DATABASE CONNECTIVITY

Standard API for accessing a DBMS. Designed to be independent of the DBMS and OS.

Originally developed in the early 1990s by Microsoft and Simba Technologies.

Every major DBMS has an ODBC implementation.
OPEN DATABASE CONNECTIVITY

ODBC is based on the device driver model. The driver encapsulates the logic needed to convert a standard set of commands into the DBMS-specific calls.
JAVA DATABASE CONNECTIVITY

Developed by Sun Microsystems in 1997 to provide a standard API for connecting a Java program with a DBMS.
→ JDBC can be considered a version of ODBC for the programming language Java instead of C.

JDBC supports different client-side configurations because there may not be a native Java driver for each DBMS.
JAVA DATABASE CONNECTIVITY

Approach #1: JDBC-ODBC Bridge ← Removed in 2014
→ Convert JDBC method calls into ODBC function calls.

Approach #2: Native-API Driver
→ Convert JDBC method calls into native calls (via JNI) of the target DBMS API.

Approach #3: Network-Protocol Driver
→ Driver connects to a middleware in a separate process that converts JDBC calls into a vendor-specific DBMS protocol.

Approach #4: Database-Protocol Driver ← Best Approach
→ Pure Java implementation that converts JDBC calls directly into a vendor-specific DBMS protocol.
DATABASE NETWORKING PROTOCOLS

All major DBMSs implement their own proprietary client wire protocol over TCP/IP.
→ Use **Unix domain sockets** if running on same box as app.
→ Andy doesn't know of any DBMS using UDP for clients.

A typical client/server interaction:
→ Client connects to DBMS and begins authentication process. There may be an SSL/TLS handshake.
→ Client then sends a query.
→ DBMS executes the query, then serializes the results and sends it back to the client.
EXISTING PROTOCOLS

Most newer systems implement one of the open-source DBMS wire protocols. This allows them to reuse the client drivers without having to develop and support them.

Just because on DBMS "speaks" another DBMS's wire protocol does not mean that it is compatible. → Need to also support catalogs, SQL dialect, and other functionality.
EXISTING PROTOCOLS

MySQL

PostgreSQL

redis

SingleStore

Clustrix

actorDB

TiDB

d0lt

CLEARDB

STARDOG

Amazon Aurora

ClickHouse

DORIS

planetscale

PolarDB

NEON

GREENPLUM DATABASE

amazon REDSHIFT

HyPer

CockroachDB

VERTICA

UMBRA

yugabyteDB

CrateDB

Materialize

Yellowbrick

QuestDB

APACHE GEODE

curiodb

roesdb

Kvrocks

Dragonfly

Tendis

KeyDB

SUMMITDB

CACHEGRAND

ZippyDB

Serverless Redis
PROTOCOL DESIGN SPACE

Row vs. Column Layout
Compression
Data Serialization
String Handling
**ROW VS. COLUMN LAYOUT**

ODBC/JDBC are row-oriented APIs.  
→ Server packages tuples into messages one tuple at a time.  
→ Client deserializes data one tuple at a time.

But switching to a column-oriented API is a bad too because client may access multiple columns for a tuple.

Solution: Vector-oriented API

```java
String sql = "SELECT * FROM xxx";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    // Do something magical row by row!
    rs.getInt(1);
    rs.getString(2);
    rs.getDate(3);
}
stmt.close();
```

```java
String sql = "SELECT * FROM xxx";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
while (rs.nextCol()) {
    while (rs.nextRow()) {
        // Do something magical per column!
        rs.getValue();
    }
}
stmt.close();
```
ROW VS.

ODBC/JDBC are row-oriented APIs.

→ Server packages tuples into messages one tuple at a time.
→ Client deserializes data one tuple at a time.

But switching to a column-oriented API is a bad too because clients may need to access multiple columns for a tuple.

Solution: Vector-oriented API

```java
String sql = "SELECT * FROM xxx";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    // Do something magical row by row!
    rs.getInt(1);
    rs.getString(2);
    rs.getDate(3);
}
stmt.close();
```

Not Real JDBC Code!
COMPRESSION

Approach #1: Naïve Compression
→ DBMS applies a general-purpose compression algo (lz4, gzip, zstd) on message chunks before transmitting.
→ Examples: Oracle, MySQL, Snowflake, BigQuery

Approach #2: Columnar-Specific Encoding
→ Analyze results and choose a specific compression encoding (dictionary, RLE, delta) per column.
→ No system implements this except with Arrow ADBC.

Heavyweight compression is better when network is slow. DBMS achieves better compression ratios for larger message chunk sizes.
DATA SERIALIZATION

Approach #1: Binary Encoding
→ Client handles endian conversion.
→ The closer the serialized format is to the DBMS's binary format, then the lower the overhead to serialize.
→ DBMS can implement its own format or rely on existing libraries (ProtoBuffers, Thrift, FlatBuffers).

Approach #2: Text Encoding
→ Convert all binary values into strings (atoi).
→ Do not have to worry about endianness.
→ Missing values encoded as string "NULL"
Approach #1: Null Termination
→ Store a null byte ('\0') to denote the end of a string.
→ Client scans the entire string to find end.

Approach #2: Length-Prefixes
→ Add the length of the string at the beginning of the bytes.

Approach #3: Fixed Width
→ Pad every string to be the max size of that attribute.
NETWORK PROTOCOL PERFORMANCE

Transfer One Tuple from TCP-H LINEITEM

MySQL+GZIP  MySQL  MonetDB  Postgres
Oracle  MongoDB  DB2  Hive

Lower is Better

Text Encoding

All Other Protocols Use Binary Encoding

Source: Hannes Mühleisen
Compression overhead is a bad tradeoff when network is fast.
NETWORK PROTOCOL PERFORMANCE

Transfer 1m Tuples from TCP-H LINEITEM

- MySQL+GZIP
- MySQL
- MonetDB
- Postgres
- Oracle
- MongoDB
- DB2
- Hive

Lower is Better

Elapsed Time (sec)

Network Latency (ms)

Verbose protocol overhead is more pronounced on slower network.

Source: Hannes Mühleisen
DATA EXPORT PERFORMANCE
Transfer 7GB of Tuples from TPC-C ORDER_LINE

Throughput (MB/sec)

Transport Method

- Postgres
- Vectorized Postgres
- Arrow Flight
- RDMA

Higher is Better

Throughput (MB/sec)

- 38
- 150
- 891
- 1057

Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats
VLDB 2020
OBSERVATION

The DBMS's network protocol implementation is not the only source of slowdown.

The OS's TCP/IP stack is slow…
→ Expensive context switches / interrupts
→ Data copying
→ Lots of latches in the kernel

How to avoid the OS entirely or work with it to make our DBMS run faster.
KERNEL-BYPASS METHODS

Allows the system to get data directly from the NIC into the DBMS address space.
→ No unnecessary data copying.
→ No OS TCP/IP stack.

Approach #1: Data Plane Development Kit
Approach #2: Remote Direct Memory Access
Approach #3: io_uring
DATA PLANE DEVELOPMENT KIT (DPDK)

Set of libraries that allows programs to access NIC directly. Treat the NIC as a bare metal device.

Requires the DBMS code to do more to manage network stack (layers 3+4), memory, and buffers.
→ Reimplement TCP/IP in usercode (e.g., F-Stack).
→ No data copying.
→ No system calls.

Example: ScyllaDB's Seastar, Yellowbrick's ybRPC
DATA PLANE DEVELOPMENT KIT (DPDK)

Set of **libraries** that allows programs to access NIC directly. Treat the NIC as a bare metal device.

Requires the DBMS code to manage network stack (layers 3+), memory, and buffers.

→ Reimplement TCP/IP in usercode (e.g., F-Stack).

→ No data copying.

→ No system calls.

Example: **ScyllaDB's Seastar**, **Yellowbrick's ybRPC**
REMOTE DIRECT MEMORY ACCESS

Read and write memory directly on a remote host without going through OS.
→ 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 (i.e., no callbacks).
→ InfiniBand vs. RoCE

Examples: Oracle Exadata, Microsoft FaRM
**IO_URING**

Linux system call interface for zero-copy asynchronous I/O operations.
→ Originally added in 2019 for accessing storage devices.
→ Expanded in 2022 to support network devices.
→ Windows has something similar called **ICOP**.

OS exposes two circular buffers (queues) to store submission and completion I/O requests.
→ DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers.
→ When OS completes request, it puts the event on the competition queue and invokes callback.
Importing 300k rows/sec with io_uring

September 12, 2022 - 18 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB’s very own Andrey Pechkurov presents how to ingest large CSV files a lot more efficiently using the SQL `COPY` statement, and takes us through the journey of benchmarking. Andrey also shares insights about how the new improvement is made possible by `io_uring` and compares QuestDB’s import versus several well-known OLAP and time-series databases in Clickhouse’s ClickBench benchmark.

Introduction

As an open source time-series database company, we understand that getting your existing data into the database in a fast and convenient manner is as important as being able to `ingest` and `query` your data efficiently later on. That’s why we decided to dedicate our new release, QuestDB 6.8, to the new parallel CSV file import feature. In this blog post, we discuss what parallel import means for our users and how it’s implemented internally. As a bonus, we also share how recent Clickhouse team’s benchmark helped us to improve both QuestDB and its demonstrated results.

How ClickBench helped us improve

Recently ClickHouse conducted a benchmark for their own database and many others, including QuestDB. The benchmark included data import as the first step. Since we were in the process of building a faster import, this benchmark provided us with nice test data and baseline results. So, what have we achieved? Let’s find out. The benchmark was using QuestDB’s HTTP import endpoint to ingest the data into an existing non-partitioned table. You may wonder why it doesn’t use a partitioned table, which stores the data sorted by the timestamp values and provides many benefits for time series analysis. Most likely, the reason is terrible import execution time. Both HTTP-based
IO_URING

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

Originally added in 2019 for accessing storage devices.

Expanded in 2022 to support network devices.

Windows has something similar called ICOP.

OS exposes two circular buffers (queues) to store submission and completion I/O requests.

DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers.

When OS completes request, it puts the event on the completion queue and invokes callback.

Consider this tale of I/O and performance. We'll start with blocking I/O, explore io_uring and kqueue, and take home an event loop very similar to some software you may find familiar.

This is a twist on King's talk at Software You Can Love, Milan '22.

Classical approach

When you want to read from a file you might open(), and then call read() as many times as necessary to fill a buffer of bytes from the file. And in the opposite direction, you call write() as many times as needed until everything is written. It's similar for a TCP client with sockets, but instead of open(), you first call socket() and then connect() to your server. Fun stuff.

In the real world though you can't always read everything you want immediately from a file descriptor. Nor can you always write everything you want immediately to a file descriptor.

You can switch a file descriptor into non-blocking mode so the call won't block while data you requested isn't available. But system calls are still expensive, incurring context switches and cache misses. In fact, networks and disks have become so fast that these costs can start to approach the cost of doing I/O itself. For the duration of time a file descriptor is unable to read or write, you don't want to waste time continuously retrying read or write system calls.
A Programmer-Friendly Abstraction Over io_uring and kqueue

Consider this tale of I/O and performance. We'll start with blocking I/O against io_uring and kqueue, and take home an event loop very similar to something you may find familiar.

This is a twist on King's talk at Software You Can Love, Milan '22.

Classical approach

When you want to read from a file you might open() and then call read() as many times as necessary to fill a buffer of bytes from the file. And in the opposite direction, you can write() as many times as needed until everything is written, similar for TCP client with sockets, but instead of open() you first connect() to your server. Fun stuff.

In the real world though you can't always read everything you want in one go. You request data from a file descriptor. Nor can you always write everything you want in one go. You request data from a file descriptor.

You can switch a file descriptor into non-blocking mode so the call will block only when there is no data to be read. System calls are still expensive because the CPU needs to switch context switches and cache lines. In fact, networks and disks have shown that these costs can start to approach the cost of doing the I/O itself. So in duration of time a file descriptor is unable to read or write, you don't want to spend a lot of time continuously retrying read or write system calls.
IO_URING is a Linux system call interface for zero-copy asynchronous I/O operations. It was originally added in 2019 for accessing storage devices and expanded in 2022 to support network devices. Windows has something similar called ICOP.

OS exposes two circular buffers (queues) to store submission and completion I/O requests. DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers. When OS completes a request, it puts the event on the completion queue and invokes a callback.

There was an experiment adding using @sauliusvl. Unfortunately, it proved to be unsustainable.

Consider this tradeoff of I/O and performance. It’s hard to start everything in main memory; data is stored in external storage. And it needs to be thought of carefully how to access the data. We have a multi-format of storage; more than one system calls to read from them. We performed experiments to find out how many times the Linux system calls perform for available devices. In total, the Sata SSD, Intel Optane were accessed via single-threaded and multi-threaded read.

A journey to io_uring, AIO and modern storage devices

Our goal is to pick the best block size for a random read. An application (or filesystem) can pick any block size and access data with respect to this block size. We vary block sizes from 4 kelbytes up to 32 megabytes. For each block size, we make some random reads across. Among these reads, we calculate average, minimum and maximum latency as well as 90.0 and 99.9 percentiles. We use system call profiling to this experiment. We believe that block size followed...
IO_URING

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

→ Originally added in 2019 for accessing storage devices.
→ Expanded in 2022 to support network devices.
→ Windows has something similar called ICOP.

OS exposes two circular buffers (queues) to store submission and completion I/O requests.

→ DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers.

→ When OS completes request, it puts the event on the completion queue and invokes callback.
IO_URING: Linux system call interface for zero-copy asynchronous I/O operations.

→ Originally added in 2019 for accessing storage devices.
→ Expanded in 2022 to support network devices.

Windows has something similar called ICOP.

OS exposes two circular buffers (queues) to store submission and completion I/O requests.

→ DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers.

→ When OS completes request, it puts the event on the competition queue and invokes callback.
IO_URING

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

→ Originally added in 2019 for accessing storage devices.
→ Expanded in 2022 to support network devices.
→ Windows has something similar called ICOP.

OS exposes two circular buffers (queues) to store submission and completion I/O requests.

→ DBMS submits requests for the kernel to perform read/write operations to DBMS-provided buffers.
→ When OS completes request, it puts the event on the competition queue and invokes callback.
I/O BOTTLENECKS

I/O devices (network, disk) are faster.

OS logic has also gotten faster.

Max Achievable Throughput:
42Gbps per CPU core

→ A 2021 study found that over 50% of CPU cycles are spent on memcpy

User-space DBMS

Source: Matt Butrovich
Instead of pulling DBMS data into user-space, push DBMS logic down into kernel-space.

→ Avoids copying buffers, scheduling user threads, and system call overhead.

Only useful for parts of the DBMS that operate on I/Os that the system does not retain for long periods of time.

Source: Matt Butrovich
USER-BYPASS METHODS

Execute DBMS logic inside of the OS kernel via extended-Berkeley Packet Filters (eBPF) to avoid having to communicate with user-space code.

Dynamically load safe, event-driven programs in kernel-space.
→ Write in C and compile to eBPF
→ Programming model is limited (no malloc, restricted # of instructions).
CONNECTION POOLING THROUGHPUT

Amazon EC2 Instances running PostgreSQL v14.5
YCSB Workload

Throughput (k TPS)

Higher is Better

pgBouncer               Odyssey               CMU Tigger

large    xlarge    2xlarge    4xlarge    8xlarge
33       31        37        37        39
32       39        44        44        47
33       48        51        50        52

Source: Matt Butrovich
**OBSERVATION**

It's great that we optimized the DBMS's server-side networking stack and the DBMS wire protocol. But what about optimizing the client-side when it receives data from the DBMS?

![Diagram showing time comparison between MySQL and PostgreSQL](image-url)

*Source: Xiaoying Wang*
**ConnectorX** is a client-side library that provides fast and memory-efficient loading of data from a DBMS into Dataframes.

→ Integrated in Polars.

Divides data into chunks to allow multiple threads to populate Dataframe arrays in parallel.
A DBMS's networking protocol is an often-overlooked bottleneck for performance.

Kernel bypass methods greatly improve performance but require more bookkeeping. → Probably more useful for internal DBMS communication.

User bypass is an interesting direction for ephemeral I/Os in DBMSs.
Query Optimization for the next two weeks.
→ I will update reading list tonight!