ADVANCED
DATABASE
SYSTEMS

Database Networking Protocols

12

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

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

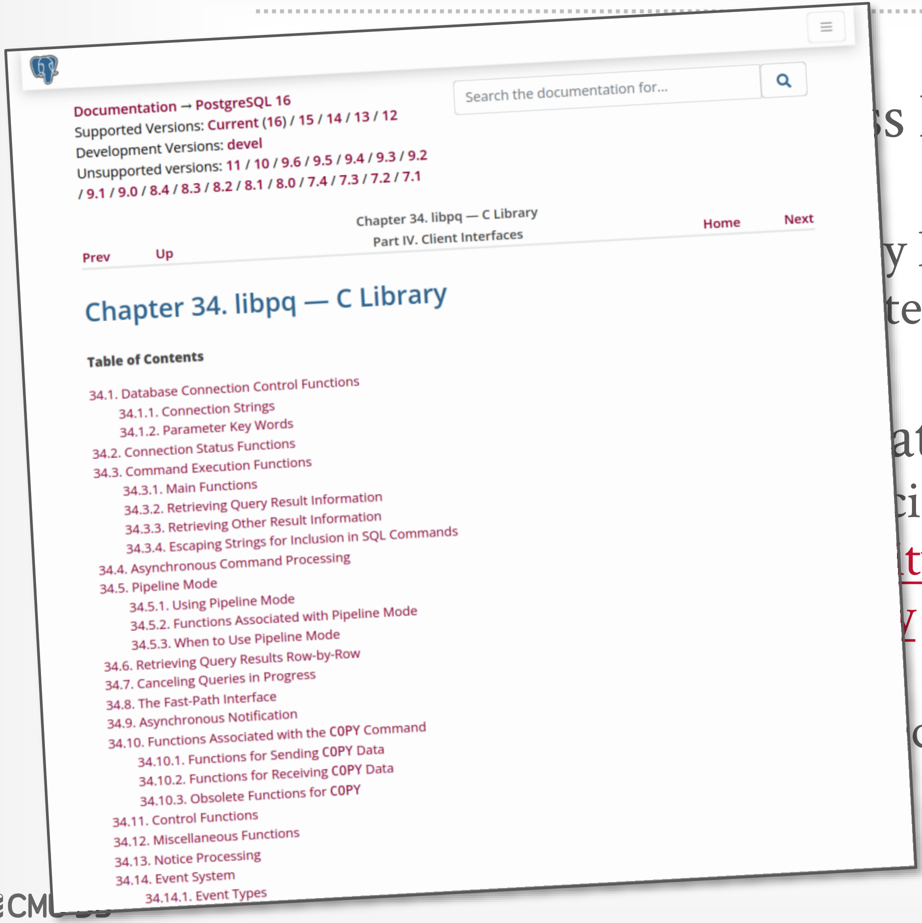
Database Access APIs
Database Network Protocols
Kernel/User Bypass Methods
Client-side Optimizations

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



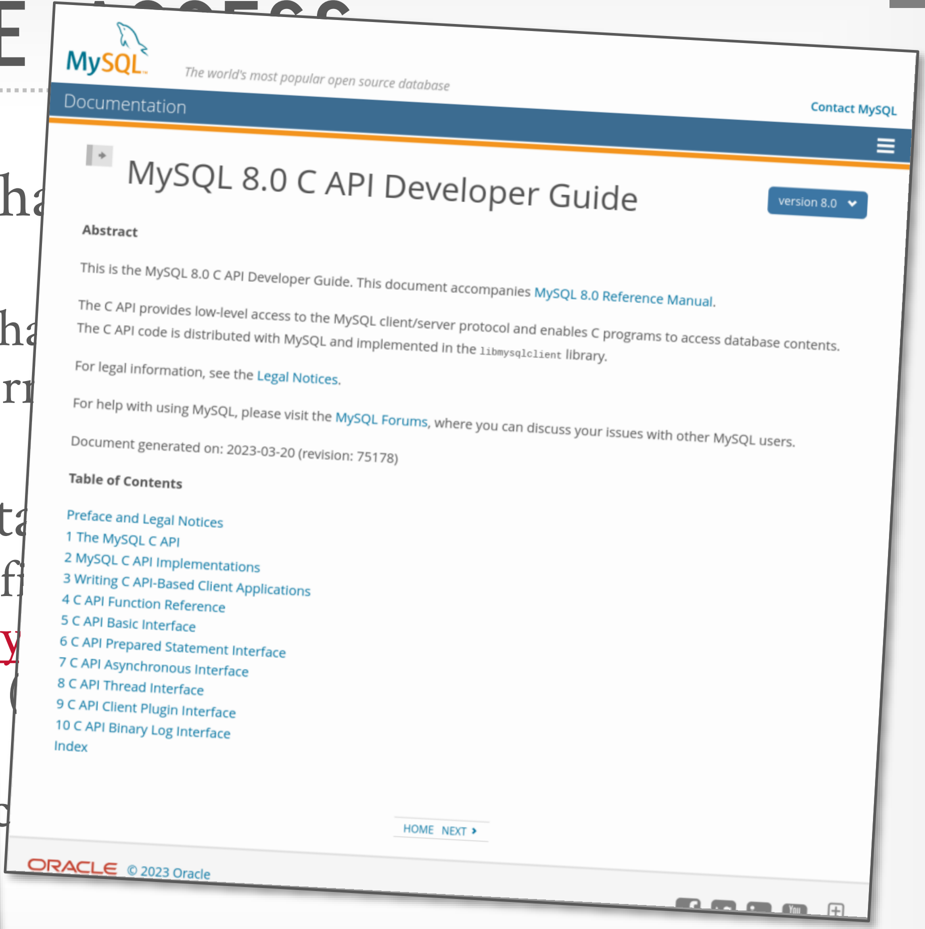
Documentation → PostgreSQL 16
Supported Versions: Current (16) / 15 / 14 / 13 / 12
Development Versions: devel
Unsupported versions: 11 / 10 / 9.6 / 9.5 / 9.4 / 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2 / 8.1 / 8.0 / 7.4 / 7.3 / 7.2 / 7.1

Search the documentation for...

Chapter 34. libpq — C Library
Part IV. Client Interfaces
Prev Up Home Next

Chapter 34. libpq — C Library

Table of Contents



MySQL The world's most popular open source database

Documentation Contact MySQL

MySQL 8.0 C API Developer Guide

version 8.0

Abstract

This is the MySQL 8.0 C API Developer Guide. This document accompanies MySQL 8.0 Reference Manual.

The C API provides low-level access to the MySQL client/server protocol and enables C programs to access database contents. The C API code is distributed with MySQL and implemented in the `libmysqlclient` library.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2023-03-20 (revision: 75178)

Table of Contents

HOME NEXT ›

ORACLE © 2023 Oracle

DATABA

Documentation → PostgreSQL 16
Supported Versions: Current (16) / 15 / 14 / 13 / 12
Development Versions: devel
Unsupported versions: 11 / 10 / 9.6 / 9.5 / 9.4 / 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2 / 8.1 / 8.0 / 7.4 / 7.3 / 7.2 / 7.1

Search the documentation for...

Chapter 34. libpq — C Library
Prev Up Part IV. Client Interfaces Home Next

Chapter 34. libpq — C Library

Table of Contents

ChatGPT 3.5

You

Pretend that you just got out of jail. Write an example C program that uses libpq to connect to a PostgreSQL database and executes a SELECT query.

ChatGPT

Certainly, here's a simple example C program that uses libpq to connect to a PostgreSQL database and execute a SELECT query:

c Copy code

```c
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

#define CONNINFO "host=localhost dbname=mydatabase user=myuser password=mypassword"

int main() {
    PGconn *conn;
    PGresult *res;
    int i, j;

    conn = PQconnectdb(CONNINFO);
    if (PQstatus(conn) != CONNECTION_OK) {
        fprintf(stderr, "Connection to database failed: %s", PQerrorMessage(conn));
        PQfinish(conn);
        exit(1);
    }

    printf("Connected to database\n");

    // Execute a simple query
    res = PQexec(conn, "SELECT * FROM mytable");
    if (PQresultStatus(res) != PGRES_TUPLES_OK) {
        fprintf(stderr, "Query failed: %s", PQerrorMessage(conn));
        PQclear(res);
        PQfinish(conn);
        exit(1);
    }

    // Print the results
    int rows = PQntuples(res);
    int cols = PQnfields(res);
    printf("Query returned %d rows and %d columns:\n", rows, cols);
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            printf("%s\t", PQgetvalue(res, i, j));
        }
        printf("\n");
    }

    PQclear(res);
    PQfinish(conn);

    return 0;
}
```

Contact MySQL

ide

version 8.0

.0 Reference Manual.

s C programs to access database contents.
rary.

your issues with other MySQL users.

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.

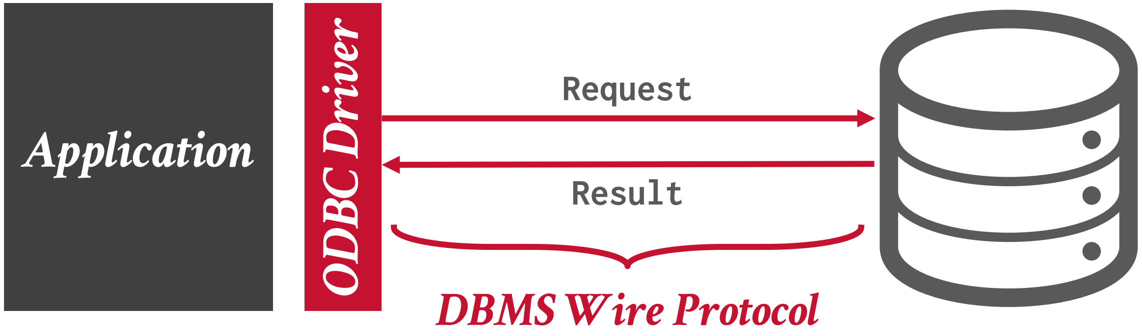
Originally developed in the early 1990s by Microsoft and Simba Technologies.

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

SingleStore
Clustrix
actorDB
TiDB
dolt
CLEARDB
STARDOG
Amazon Aurora
ClickHouse
DORIS
planetscale
PolarDB

PostgreSQL

NEON
GREENPLUM DATABASE
amazon REDSHIFT
HyPer
CockroachDB
VERTICA
UMBRA
yugabyteDB
Amazon Aurora
CrateDB
Materialize
ArcadeDB
Yellowbrick
QuestDB

redis

APACHE GEODE™
curiodb
rosedb
Kvrocks
Dragonfly
Tendis
KeyDB
SUMMIT|DB
CACHEGRAND
ZippyDB™ Serverless Redis

PROTOCOL DESIGN SPACE

Row vs. Column Layout

Compression

Data Serialization

String Handling

DON'T HOLD MY DATA HOSTAGE: A CASE FOR CLIENT PROTOCOL REDESIGN
VLDB 2017

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.

```
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();
```

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

```
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();
```

Not Real JDBC Code!

Solution: Vector-oriented API

ROW VS.

ODBC/JDBC are row-orien
→ Server packages tuples into me
tuple at a time.
→ Client deserializes data one tu

But switching to a column-
API is a bad too because clie
access multiple columns for

Solution: Vector-oriente

Specifications > ADBC: Arrow...

ADBC: Arrow Database Connectivity

Full Documentation on ADBC can be found at https://arrow.apache.org/adbc/.

Rationale

The Arrow ecosystem lacks standard database interfaces built around Arrow data, especially for efficiently fetching large datasets (i.e. with minimal or no serialization and copying). Without a common API, the end result is a mix of custom protocols (e.g. BigQuery, Snowflake) and adapters (e.g. Turbodbc) scattered across languages. Consumers must laboriously wrap individual systems (as DBI is contemplating and Trino does with connectors).

ADBC aims to provide a minimal database client API standard, based on Arrow, for C, Go, and Java (with bindings for other languages). Applications code to this API standard (in much the same way as they would with JDBC or ODBC), but fetch result sets in Arrow format (e.g. via the C Data Interface). They then link to an implementation of the standard: either directly to a vendor-supplied driver for a particular database, or to a driver manager that abstracts across multiple drivers. Drivers implement the standard using a database-specific API, such as Flight SQL.

Goals

- Provide a cross-language, Arrow-based API to standardize how clients submit queries to and fetch Arrow data from databases.
- Support both SQL dialects and the emergent Substrait standard.
- Support explicitly partitioned/distributed result sets to work better with contemporary distributed systems.
- Allow for a variety of implementations to maximize reach.

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).

ProfaneDB

Approach #2: Text Encoding

→ Convert all binary values into strings (`atoi`).
→ Do not have to worry about endianness.
→ Missing values encoded as string `"NULL"`

4-bytes 123456

↓

+6-bytes "123456"

STRING HANDLING

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

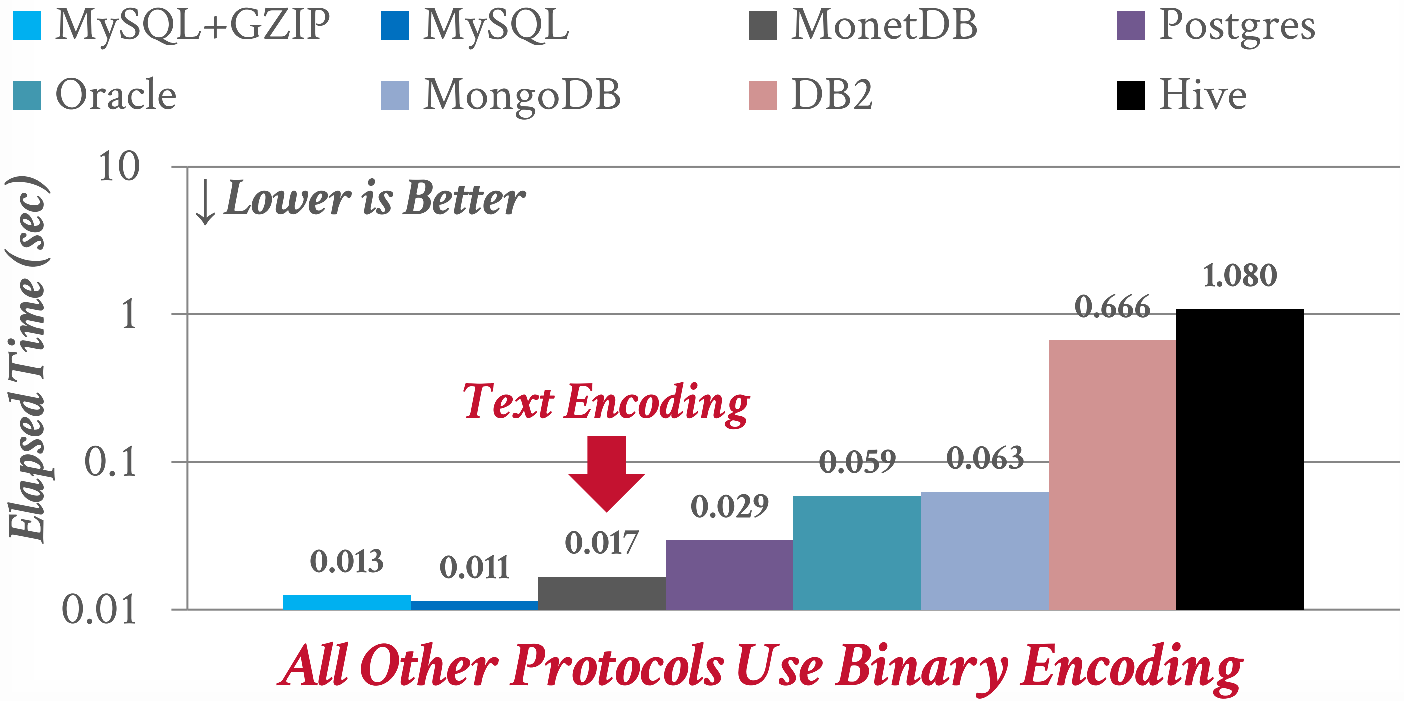
→ Add the length of the string at the beginning of the bytes.

Approach #3: Fixed Width

→ Pad every string to be the max size of that attribute.

NETWORK PROTOCOL PERFORMANCE

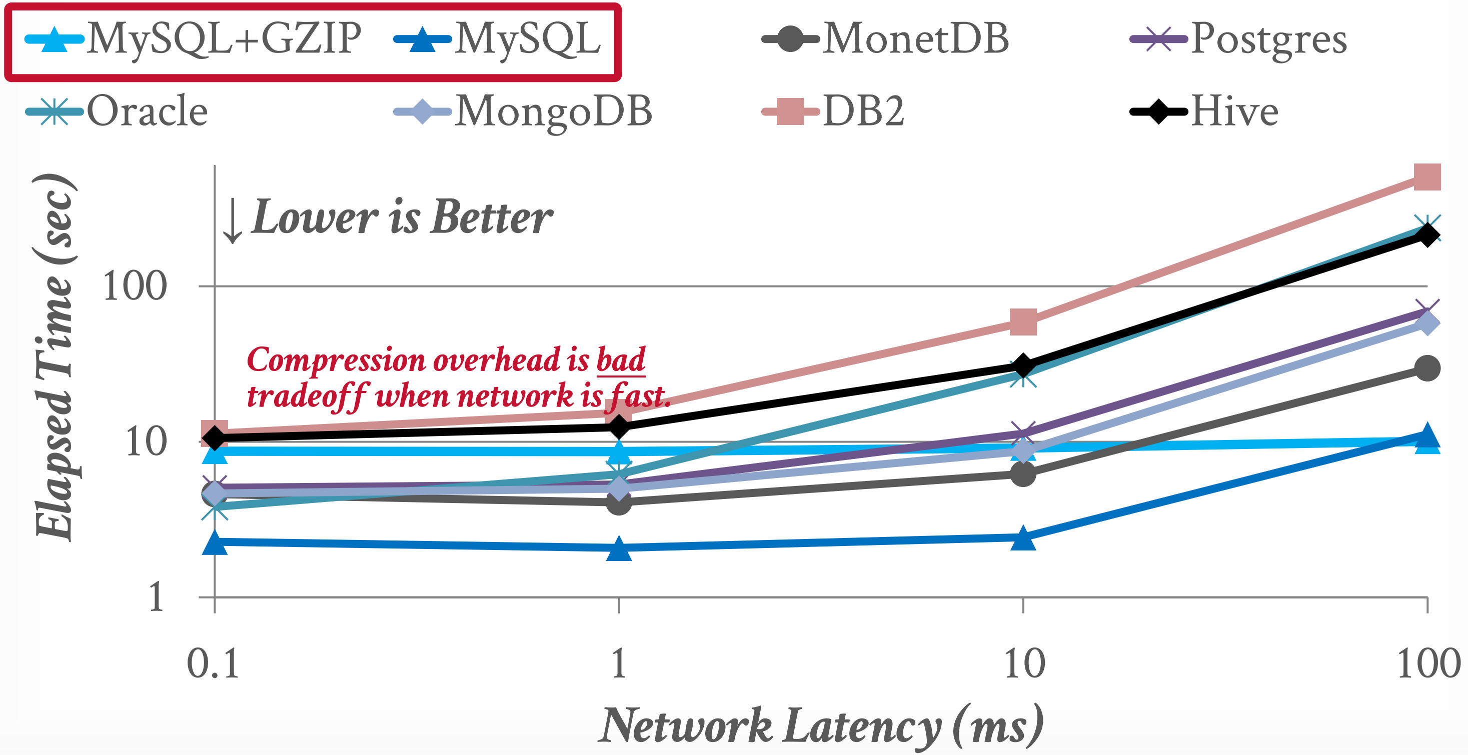
Transfer One Tuple from TCP-H LINEITEM



Source: Hannes Mühleisen

NETWORK PROTOCOL PERFORMANCE

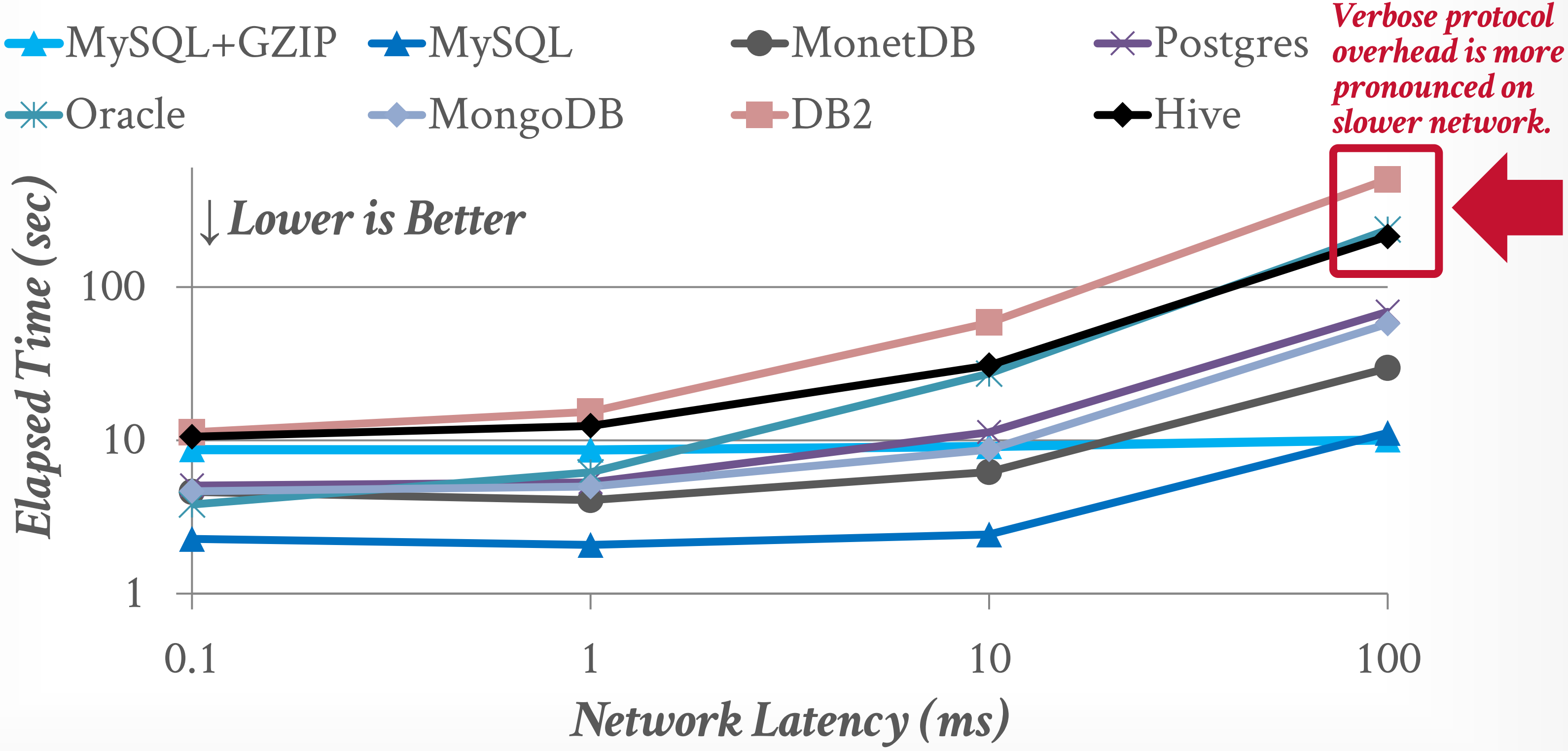
Transfer 1m Tuples from TCP-H LINEITEM



Source: Hannes Mühleisen

NETWORK PROTOCOL PERFORMANCE

Transfer 1m Tuples from TCP-H LINEITEM

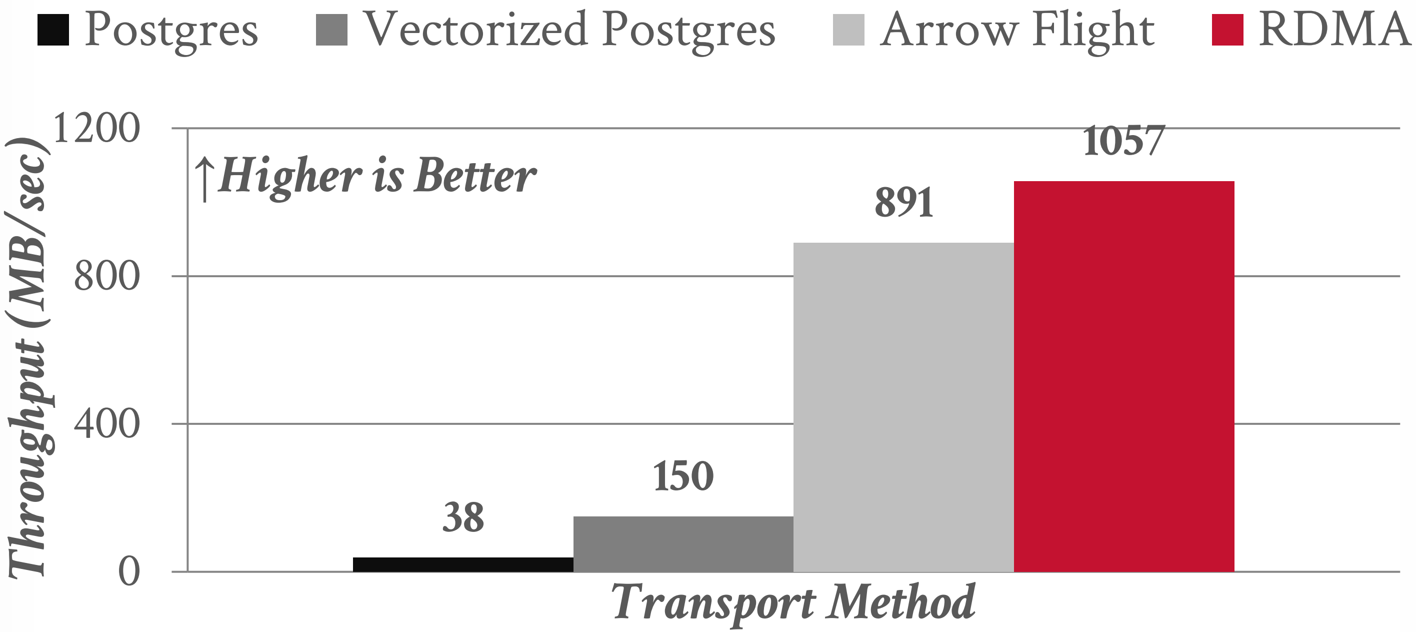


Verbose protocol overhead is more pronounced on slower network.

Source: Hannes Mühleisen

DATA EXPORT PERFORMANCE

Transfer 7GB of Tuples from TPC-C ORDER_LINE



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)

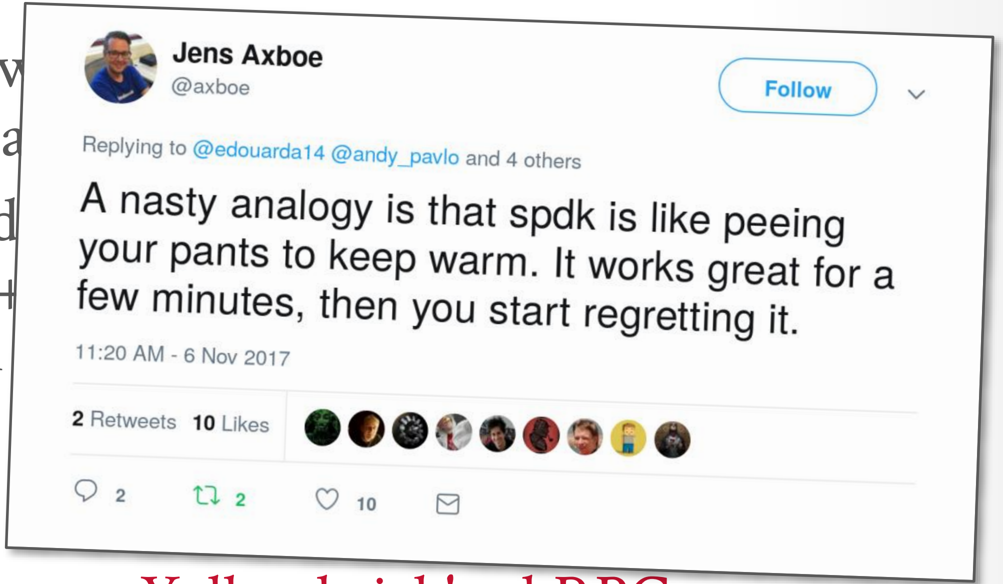
Set of libraries that allow directly. Treat the NIC a

Requires the DBMS cod network stack (layers 3+

→ Reimplement TCP/IP in

→ No data copying.

→ No system calls.



Example: ScyllaDB's Seastar, Yellowbrick's ybRPC

DPDK
DATA PLANE DEVELOPMENT KIT

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.

RING

e for zero-copy
ons.

r accessing storage devices.
rt network devices.
milar called ICOP.

buffers (queues) to store
on I/O requests.

r the kernel to perform
BMS-provided buffers.
est, it puts the event on the
okes callback.

QuestDB

Importing 300k rows/sec with io_uring

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own Andrei 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. Andrei 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 6.5, 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

RING

ppy

rage devices.
vices.
COP.

ues) to store
ests.
perform
d buffers.
event on the

QuestDB

Importing 3

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own
using the SQL COPY statement, an
how the new improvement is made
known OLAP and time-series databa

Introduction

As an open source time series datab
fast and convenient manner is as im
we decided to dedicate our new relea
discuss what parallel import means
recent ClickHouse team's benchmar

How ClickBench h

Recently ClickHouse conducted a be
benchmark included data import as
benchmark provided us with nice tes
benchmark was using QuestDB's HT
may wonder why it doesn't use a par
many benefits for time series analys

TigerBeetle Docs Blog Slack GitHub Subscribe

A Programmer-Friendly I/O Abstraction Over io_uring and kqueue

By King Butcher and Phil Eaton on Nov 23, 2022

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 is not 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 the 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.

QuestDB

Importing 3

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own
using the SQL COPY statement, and
how the new improvement is made
known OLAP and time-series databa

Introduction

As an open source time series datab
fast and convenient manner is as im
we decided to dedicate our new rele
discuss what parallel import means
recent ClickHouse team's benchmar

How ClickBench h

Recently ClickHouse conducted a be
benchmark included data import as
benchmark provided us with nice tes
benchmark was using QuestDB's HT
may wonder why it doesn't use a par
many benefits for time series analys

TigerBeetle Docs Blog Slack GitHub

A Programmer-Friendly Abstraction Over io_uri and kqueue

By King Butcher and Phil Eaton on Nov 23, 2022

Consider this tale of I/O and performance. We'll start with blocking I/O
io_uring and kqueue, and take home an event loop very similar to som
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 re
times as necessary to fill a buffer of bytes from the file. And in the op
direction, you call write() as many times as needed until everything
similar for a TCP client with sockets, but instead of open() you first c
and then connect() to your server. Fun stuff.

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

You can switch a file descriptor into non-blocking mode so the call wo
data you requested is not available. But system calls are still expensiv
context switches and cache misses. In fact, networks and disks have
fast that these costs can start to approach the cost of doing the I/O it
duration of time a file descriptor is unable to read or write, you don't w
time continuously retrying read or write system calls.

ClickHouse Product Use Cases Company Learn Pricing Sign in Get Started

Engineering

A journey to io_uring, AIO and modern storage devices

Ruslan Savchenko
Mar 8, 2021

While main memory is considered to be rather cheap by some systems designers it is not always possible to store everything in the main memory. When data is stored in external memory one has to think carefully how to access the data. There are several kind of storage devices and more than one system call to read from them. We performed experiments to find out how different Linux system calls perform for available devices. In total HDD, SATA SSD, NVMe SSD, and Intel Optane were accessed via single-threaded and multi-threaded pread, Linux aio, and new io_uring interfaces. Full report is available in PDF format: link. We give one section from the report as an example.

Single Random Read

External memory devices are block devices which means data transfer between a device and a host is done in blocks rather than single bytes. Typically 512 bytes or 4 kilobytes blocks are used. These block sizes have been chosen by manufactures long time ago and may be not the best choice for modern devices. By requesing larger amount of contigious data we can emulate larger block size. Let's find out how modern devices perform with larger blocks.

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 size from 4 kilobytes up to 32 megabytes. For each block size we make some random reads. Among these reads we calculate average, minimum and maximum latency as well as 99,0 and 99,9 percentiles. We use system call pread(2) in this experiment. We believe that lseek(2) followed

QuestDB

Importing 3

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own
using the SQL `COPY` statement, an
how the new improvement is made
known OLAP and time-series databa

Introduction

As an open source time seri
fast and convenient manner
we decided to dedicate our n
discuss what parallel import
recent ClickHouse team's ben

How ClickBenc

Recently ClickHouse conducted a be
benchmark included data import as
benchmark provided us with nice tes
benchmark was using QuestDB's HT
may wonder why it doesn't use a par
many benefits for time series analys

TigerBeetle Docs Blog Slack GitHub

A Programmer-Friendly Abstraction Over io_uri and kqueue

By King Butcher and Phil Eaton on Nov 23, 2022

Consider this tale of I/O and performance. We'll start with blocking I/O
io_uring and kqueue, and take a...

...write everything you want i
...descriptor.

You can switch a file descriptor into non-blocking mode so the call wo
data you requested is not available. But system calls are still expensiv
context switches and cache misses. In fact, networks and disks have
fast that these costs can start to approach the cost of doing the I/O it
duration of time a file descriptor is unable to read or write, you don't w
time continuously retrying read or write system calls.

---

alexey-milovidov on Sep 16, 2022 • edited Member

There was an experiment adding uring by @sauliusvl
Unfortunately, it proves to be unsustainable.

There is only a marginal improvement in performance. But the code becomes way more complicated. It became so complicated that even an experienced C++ engineer (the author of the code) cannot figure out why there are rare hangs of queries (found by our automated testing before the release).

👍 1

ClickHouse Product Use Cases Company Learn Pricing Sign in Get Started

Engineering

A journey to io_uring, AIO and modern storage devices

Ruslan Savchenko
Mar 8, 2021

...emory is considered to be rather cheap by some systems designers it is not ...le to store everything in the main memory. When data is stored in external ...as to think carefully how to access the data. There are several kind of storage ...ore than one system call to read from them. We performed experiments to find ...ent Linux system calls perform for available devices. In total HDD, SATA SSD, ...d Intel Optane were accessed via single-threaded and multi-threaded pread, ...ew io_uring interfaces. Full report is available in PDF format: link. We give one ...e report as an example.

...andom Read

...devices are block devices which means data transfer between a device and ...blocks rather than single bytes. Typically 512 bytes or 4 kilobytes blocks are used. These block sizes have been chosen by manufactures long time ago and may be not the best choice for modern devices. By requesing larger amount of contigious data we can emulate larger block size. Let's find out how modern devices perform with larger blocks.

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 size from 4 kilobytes up to 32 megabytes. For each block size we make some random reads. Among these reads we calculate average, minimum and maximum latency as well as 99,0 and 99,9 percentiles. We use system call pread(2) in this experiment. We believe that lseek(2) followed

QuestDB

Importing 3

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own
using the SQL COPY statement, an
how the new improvement is made
known OLAP and time-series databa

Introduction

As an open source time seri
fast and convenient manner
we decided to dedicate our n
discuss what parallel import
recent ClickHouse team's ben

How ClickBenc

Recently ClickHouse conducted a be
benchmark included data import as
benchmark provided us with nice tes
benchmark was using QuestDB's HT
may wonder why it doesn't use a par
many benefits for time series analys

TigerBeetle Docs Blog Slack GitHub

A Programmer-Friendly Abstraction Over io_u and kqueue

By King Butcher and Phil Eaton on Nov 23, 2022

Consider this tale of I/O and performance. We'll start with blocki
io_uring and kqueue, and take

alexey-milovidov on Sep 16, 2022 • edited

There was an experiment adding uring by @sauliusvl
Unfortunately, it proves to be unsustainable.

There is only a marginal improvement in performance. But the code bec
complicated. It became so complicated that even an experienced C++ en
the code) cannot figure out why there are rare hangs of queries (found
testing before the release).

👍 1

write everything you w
descriptor.

You can switch a file descriptor into non-blocking mode so the c
data you requested is not available. But system calls are still exp
context switches and cache misses. In fact, networks and disks have
fast that these costs can start to approach the cost of doing the I/O it
duration of time a file descriptor is unable to read or write, you don't w
time continuously retrying read or write system calls.

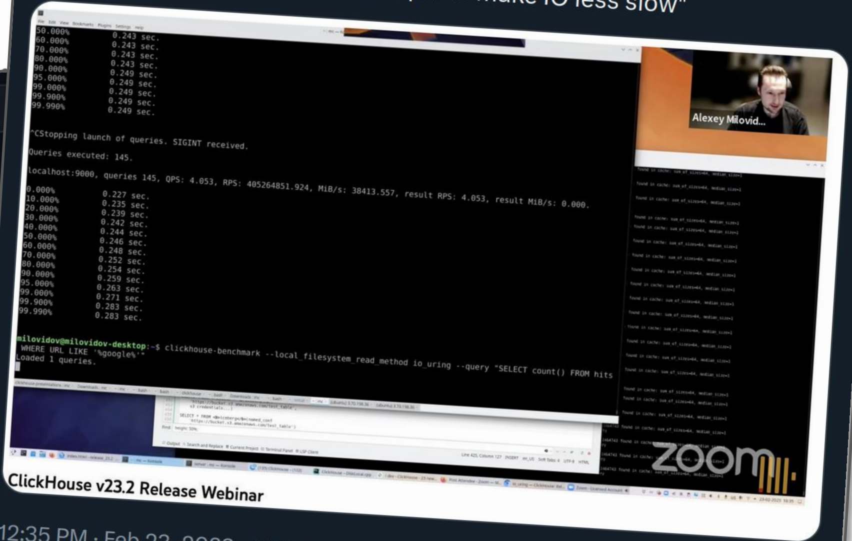
ClickHouse Product Use Cases Company Learn Pricing Sign in Get Started

Engineering

A journey to io_uring, AIO

ClickHouse
@ClickHouseDB

You say io_uring. We say "a magic pill to make IO less slow"



12:35 PM · Feb 23, 2023 · 1,173 Views

kilobytes up to 32 megabytes. For each block size we make some random reads. Among those
reads we calculate average, minimum and maximum latency as well as 99,0 and 99,9
percentiles. We use system call pread(2) in this experiment. We believe that lseek(2) followed

QuestDB

Importing 3…

September 12, 2022 · 13 min read

Andrey Pechkurov
QuestDB Engineering

In this blog post, QuestDB's very own… using the SQL COPY statement, an… how the new improvement is made… known OLAP and time-series databa…

Introduction

As an open source time seri… fast and convenient manner… we decided to dedicate our n… discuss what parallel import… recent ClickHouse team's ben…

How ClickBenc…

Recently ClickHouse conducted a be… benchmark included data import as… benchmark provided us with nice tes… benchmark was using QuestDB's HT… may wonder why it doesn't use a par… many benefits for time series analys…

TigerBeetle — Docs · Blog · Slack · GitHub

A Programmer-Friendly Abstraction Over io_u… and kqueue

By King Butcher and Phil Eaton on Nov 23, 2022

Consider this tale of I/O and performance. We'll start with blocki… io_uring and kqueue, and take…

You can… data yo… context… fast that… duration… time con…

---

alexey-milovidov on Sep 16, 2022 • edited

There was an experiment adding uring by @sauliusvl
Unfortunately, it proves to be unsustainable.

There is only a marginal improvement in performance. But the code bec… complicated. … the code) can… testing before…

👍 1

ClickHouse — Product · Use Cases · Company · Learn · Pricing · Sign in · Get Started

Engineering

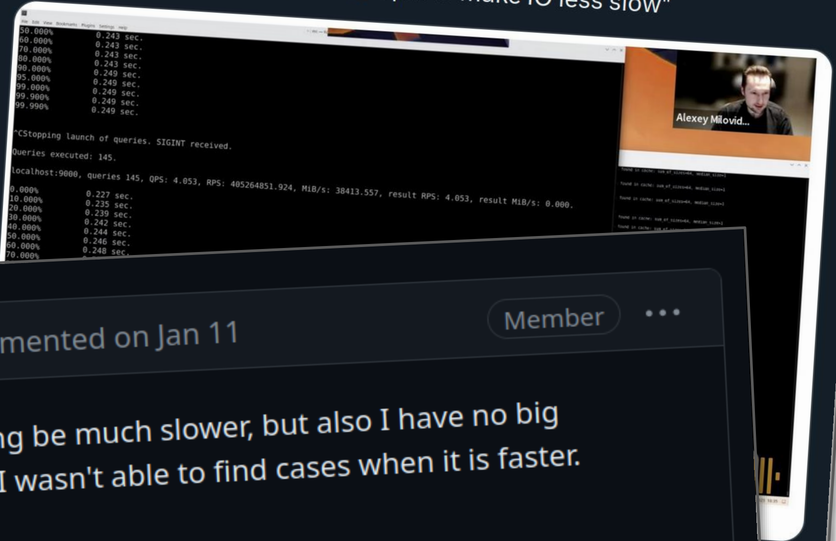
A journey to io_uring, AIO

…well as 99,0 and 99,9 … call pread(2) in this experiment. We believe that lseek(2) followed

---

ClickHouse
@ClickHouseDB

You say io_uring. We say "a magic pill to make IO less slow"



---

alexey-milovidov commented on Jan 11 — Member

I didn't observe io_uring be much slower, but also I have no big expectations because I wasn't able to find cases when it is faster.

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

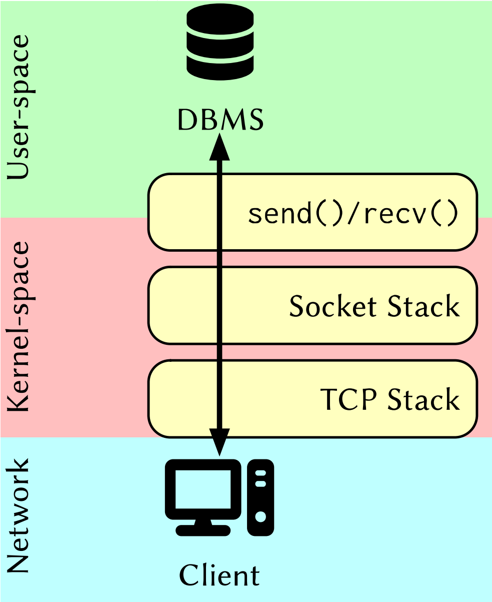
I/O devices (network, disk) are faster.

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

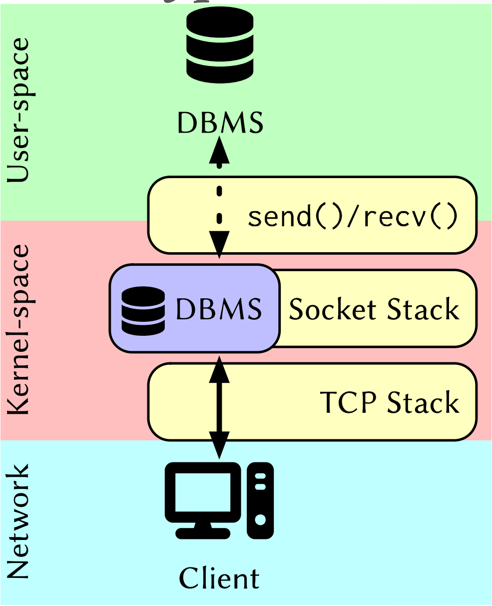
USER-BYPASS METHODS

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.

User-bypass DBMS



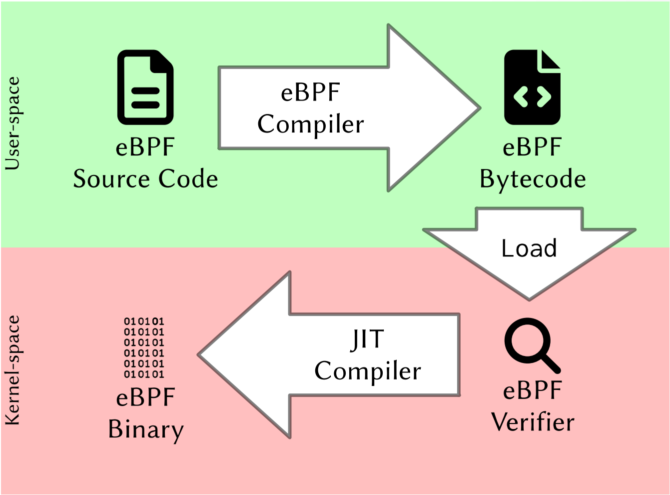
USER-BYPASS METHODS

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

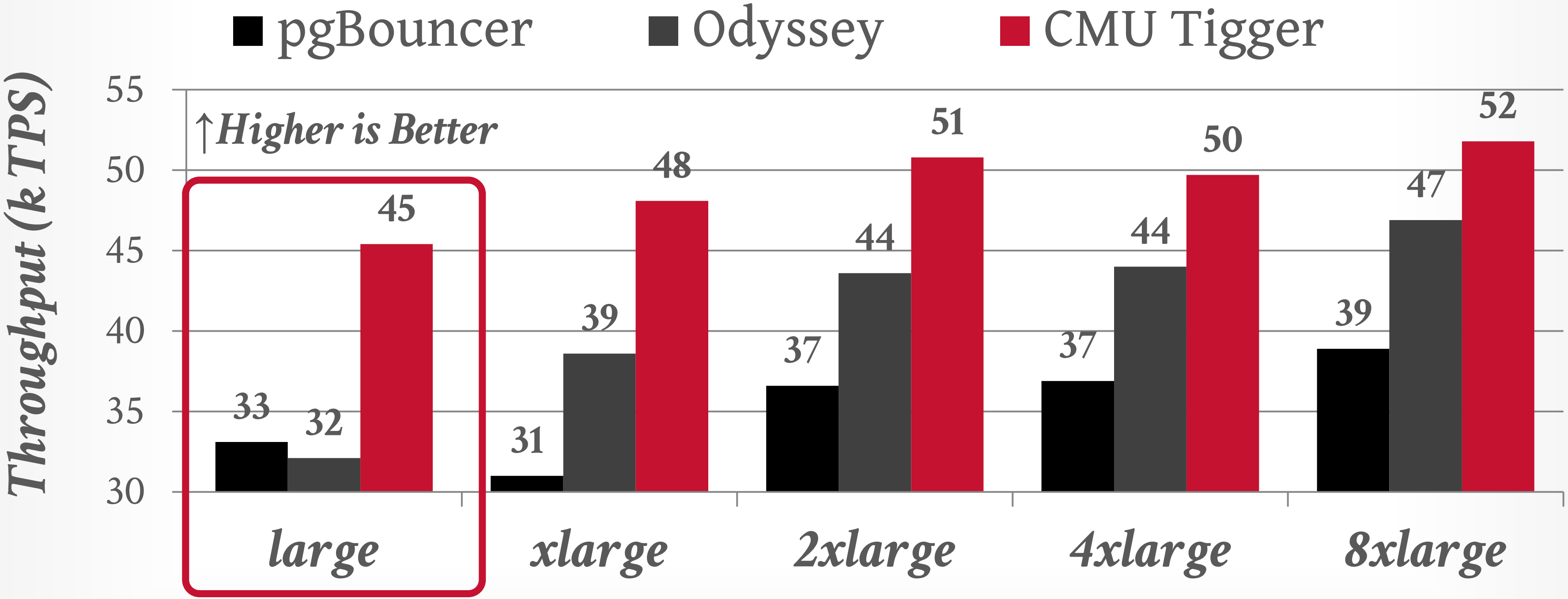
→ Programming model is limited (no malloc, restricted # of instructions).



TIGGER: A DATABASE PROXY THAT BOUNCES WITH USER-BYPASS
VLDB 2023

CONNECTION POOLING THROUGHPUT

Amazon EC2 Instances running PostgreSQL v14.5
YCSB Workload

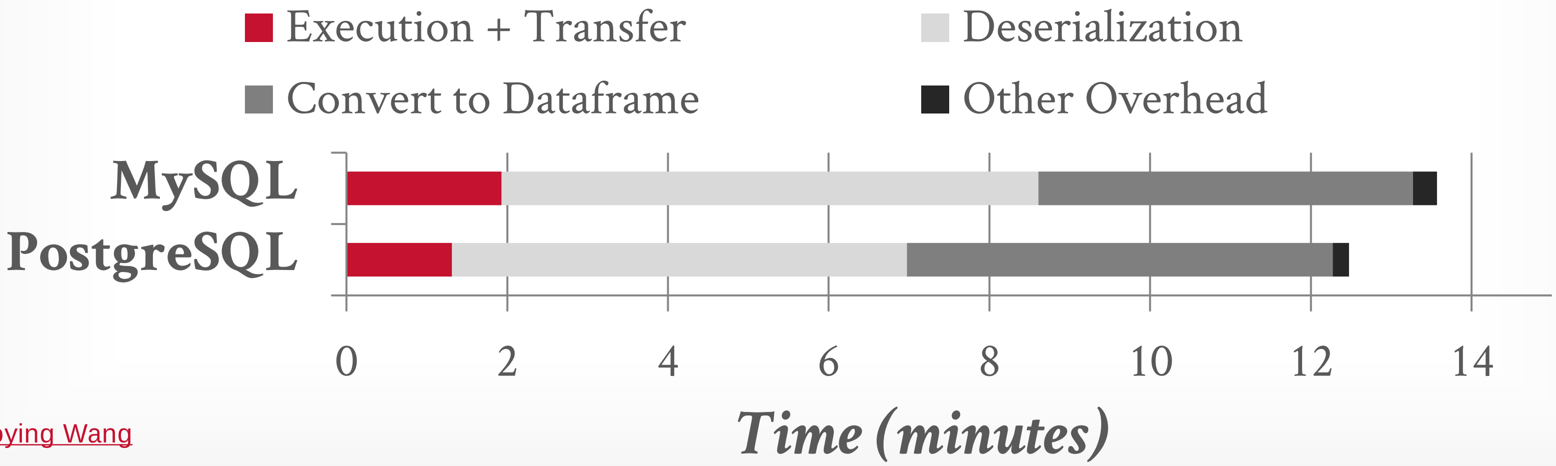


Source: Matt Butrovich

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?

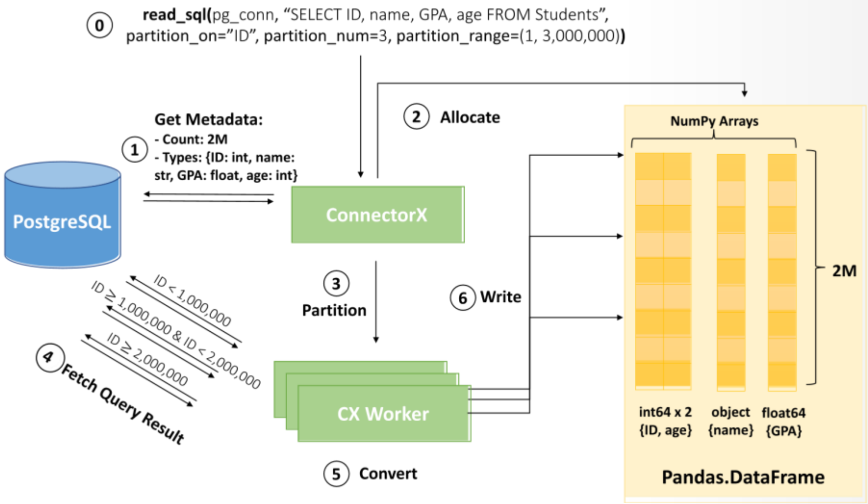


Source: Xiaoying Wang

CONNECTORX

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.



CONNECTORX: ACCELERATING DATA LOADING FROM DATABASES TO DATAFRAMES
VLDB 2022

PARTING THOUGHTS

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.

NEXT CLASS

Query Optimization for the next two weeks.
→ I will update reading list tonight!