Enhancing A Distributed SQL Database Engine: A Case Study on Performance Optimization

Alexey Ozeritskiy

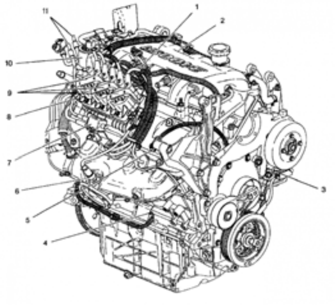
2024

About me

Alexey is a software engineer who is passionate about distributed databases. Alexey worked on Big Data Platform at Yandex for a long time. Since February 2023, he has been focusing on enhancing the SQL engine performance in the YDB database.

Outline

1. Overview & background information
2. Testing methodology
3. Investigations
4. Containerization and performance



1. Overview & background information
2. Testing methodology
3. Investigations
4. Containerization and performance

YQL: Distributed SQL Database Engine

YQL (YDB Query Language) - A library designed to parse and execute SQL queries.
Used in:

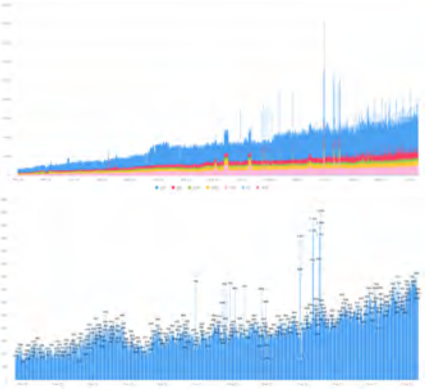
- YDB[1](Distributed Opensource SQL Database)
- YTSaurus[2](Opensource Big Data Platform)
- YQL[3](Internal Yandex Service)
- Yandex Query[4](Yandex's BigQuery-like service)

---

[1]https://ydb.tech/
[2]https://ytsaurus.tech/
[3]https://habr.com/ru/companies/yandex/articles/312430/
[4]https://cloud.yandex.ru/ru/services/query

Massive Data Handling

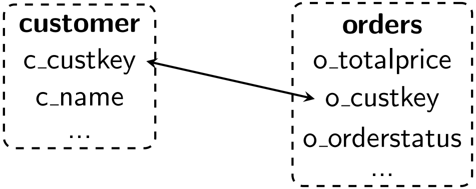
- 600000 Queries Per Day
- 800PB Per Day



YQL Architecture Overview

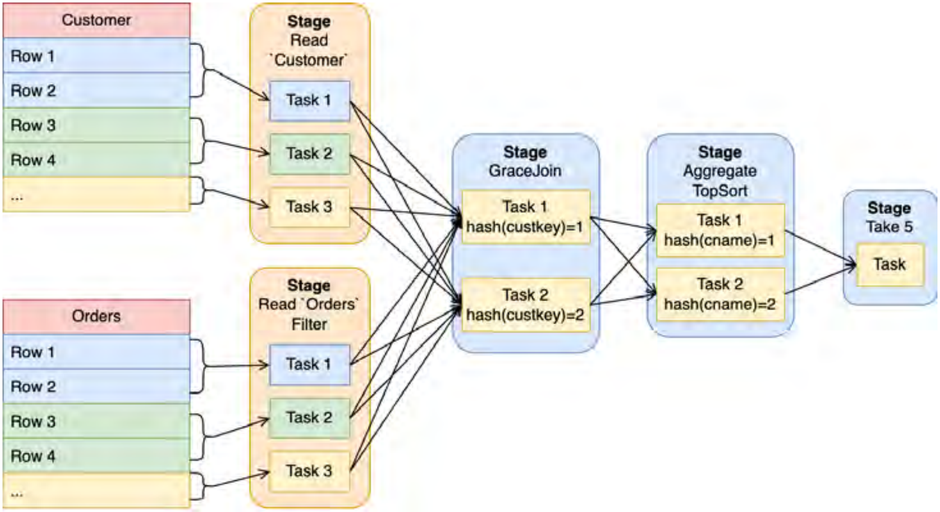
- **Parser**
  - Initial processing of queries.
  - Syntax analysis and validation.
- **Execution Plan Builder**
  - Constructs the execution plan.
  - Optimizes the query for efficient processing.
- **Execution**
  - Manages the overall execution in a distributed system.
  - Coordinates between nodes and processes.
- **Compute**
  - Handles execution of individual plan nodes.
  - Responsible for computations like filters, projections, expressions, functions, etc.

Example




```sql
select c_name, sum(o_totalprice) as totalprice from orders
join customer on o_custkey = c_custkey
where o_orderstatus = 'O' group by c_name
order by totalprice desc limit 5
```

Execution Plan



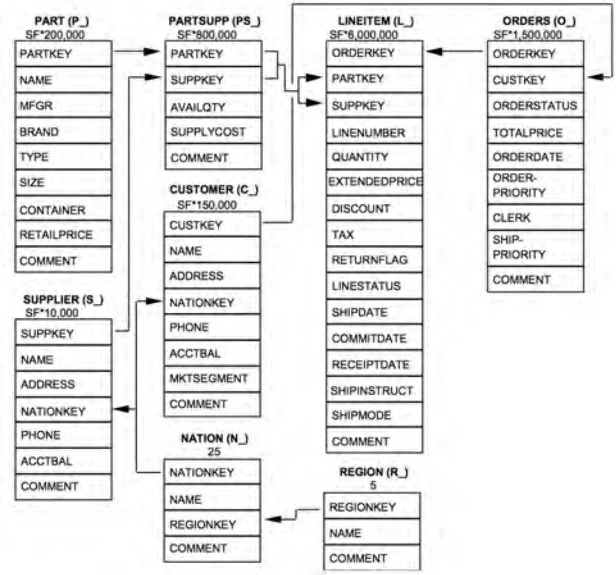
1. Overview & background information
2. Testing methodology
3. Investigations
4. Containerization and performance

Benchmark-driven approach

- Metrics
- Bottleneck Identification
- Scalability Testing
- Real-world Simulation
- Vendor Neutral comparison

TPC-H Benchmark

- Benchmark for OLAP systems
- 22 SQL Queries
- 9 Tables
- Data Generator



TPC-H Benchmark Data Generation

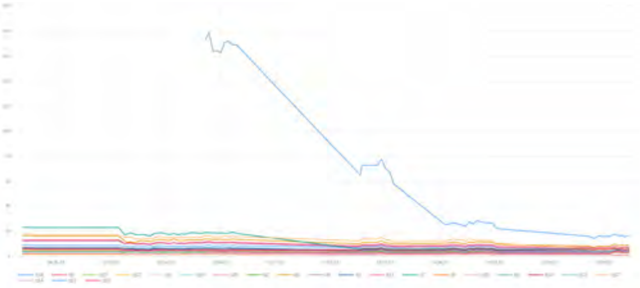
```
dbgen -s 100 -C 100 -S 88
```

- **-s 100**: Scale factor 100, approximating 100GB of data
- **-C 100**: Generate data in 100 parallel jobs
- **-S 88**: Specifies this as job number 88
- **-C, -S**: Used for large-scale data generation

- Generate Everything on MapReduce
- Convert/Upload to S3/Parquet
- Parquet: Compression: Snappy, RowGroup: 10^5, Table Split: 60 parts.

Continuous Integration (CI)

- Use VM and small scale (10)
- Daily Run
- Use Parquet Files
- Per-Commit Run
- Commit-Commit Comparison



Run Distributed Engine in One Process

- `dqrun`[5] is a utility for local debugging of a distributed SQL engine.
- Can run all components of a distributed engine in a single process for debugging.

Example of Usage

```
dqrun -s -p query.sql \
      --gateways-cfg examples/gateways.conf \
      --fs-cfg examples/fs.conf \
      --bindings-file examples/bindings_tpch.json
```

[5] https://github.com/ydb-platform/ydb/tree/main/ydb/library/yql/tools/dqrun

Run Distributed Engine in Multi-Process Configuration

- service_node and worker_node[6] are testing utilities for debugging a distributed SQL engine in a distributed configuration.

Running service_node

```
service_node --id 1 --port 5555 --grpcport 8080
```

Running worker_node

```
worker_node --id 2 --port 5556 --service_addr localhost:8080 --workers 4
```

dqrun as a Client

```
dqrun --dq-host localhost --dq-port 8080 -s -p query.sql \
      --gateways-cfg examples/gateways.conf \
      --fs-cfg examples/fs.conf --bindings-file examples/bindings_tpch.json
```

[6] https://github.com/ydb-platform/ydb/tree/main/ydb/library/yql/tools/dq

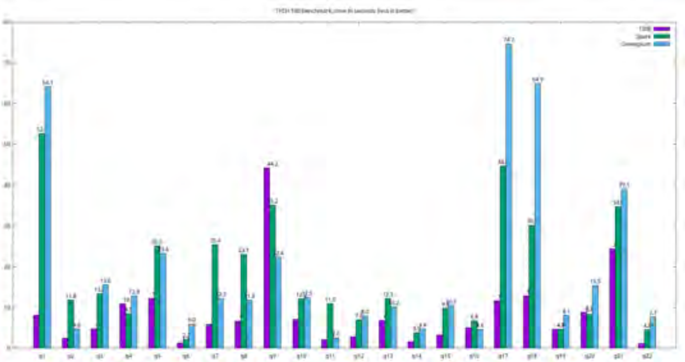
UnixBench's Style Measures

- Execute the test suite N times.
- Apply the UnixBench[7]technique to determine the final value:
  - Discard the lowest third of the results.
  - Calculate the final value using the geometric mean of the remaining results.

[7]https://github.com/kdlucas/byte-unixbench

TPC-H 100: Target Values

- TPC-H 100 Benchmark Results: TiDB v5.1, Greenplum 6.15.0, Apache Spark 3.1.1
- Details[8]: Xeon E5-2630 (120 cores total), 3 nodes, NVMe
- Execution Times: TiDB: 189s, Greenplum: 436s, Spark: 388s



[8] https://docs.pingcap.com/tidb/v5.1/v5.1-performance-benchmarking-with-tpch

Hardware

- 2x Xeon Gold 6338 (64 cores, 128 threads)
- 512GB RAM
- `taskset` for thread pinning

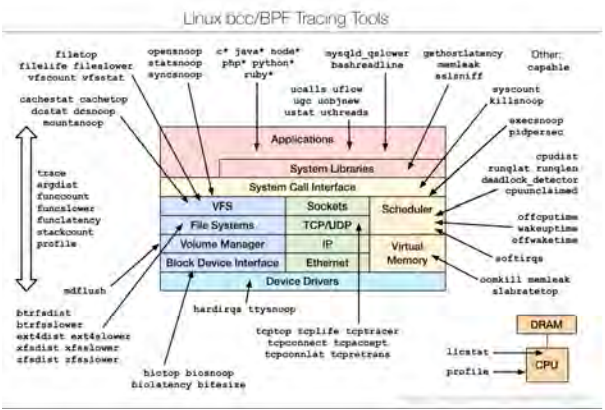
1. Overview & background information
2. Testing methodology
3. Investigations
4. Containerization and performance

Linux Performance Tools

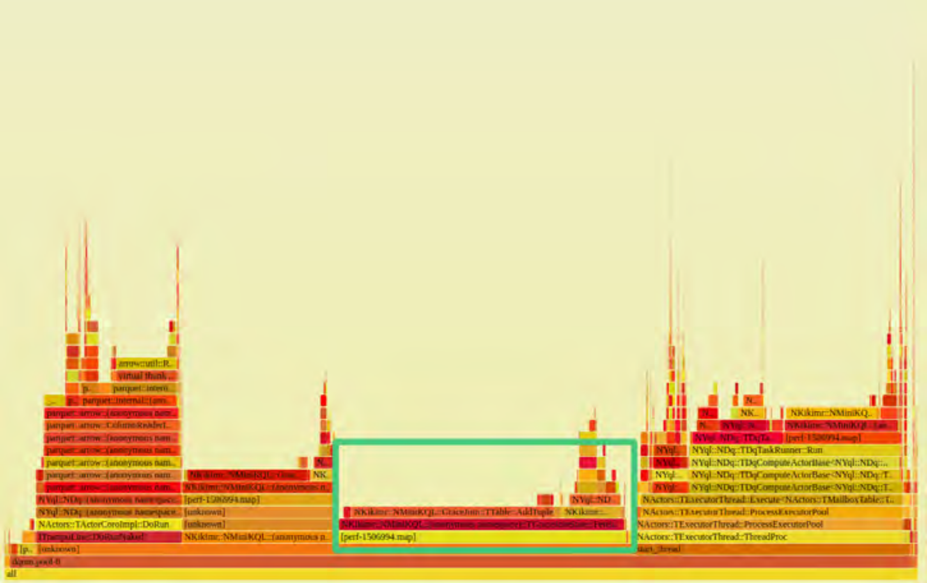
- `perf` - A versatile performance analyzing tool.
- `stackcount` - Tracks function call counts and stack traces.
- `memleak` - Identifies potential memory leaks in applications.

More Linux Performance Tools

- https://github.com/iovisor/bpftrace
- https://github.com/iovisor/bcc
- https://github.com/brendangregg/FlameGraph

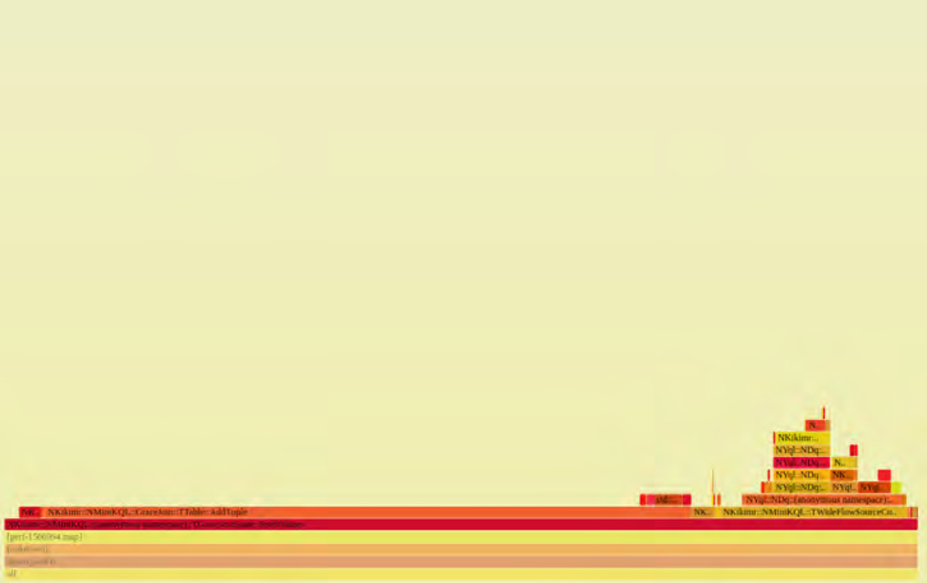


Slow Join



NKikimr::NMiniKQL::GraceJoin::TTable::AddTuple
[perf-1506994.map]
NYql::NDq::TDqTaskRunner::Run
NActors::TExecutorThread::ProcessExecutorPool
NActors::TExecutorThread::ThreadProc
NActors::TActorCoroImpl::DoRun
[unknown]
all

Slow Join: Closer Look



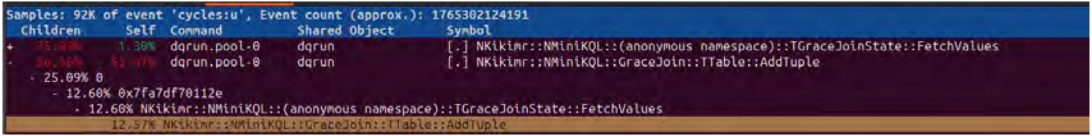
Slow Join: Closer Look

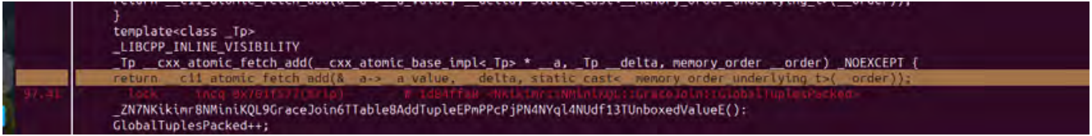
`perf report`

```
Samples: 92K of event 'cycles:u', Event count (approx.): 1765302124191
  Children      Self  Command          Shared Object       Symbol
+   75.87%     1.30%  dqrun.pool-0     dqrun               [.] NKikimr::NMiniKQL::(anonymous namespace)::TGraceJoinState::FetchValues
-   56.56%    51.97%  dqrun.pool-0     dqrun               [.] NKikimr::NMiniKQL::GraceJoin::TTable::AddTuple
   - 25.09% 0
      - 12.60% 0x7fa7df70112e
         - 12.60% NKikimr::NMiniKQL::(anonymous namespace)::TGraceJoinState::FetchValues
              12.57% NKikimr::NMiniKQL::GraceJoin::TTable::AddTuple
```

Slow Join: Closer Look

perf report





Slow Join: Atomics!

Performance Improvement: Q21 TPC-H 10 from 15s to 7s.

```diff
4 ydb/library/yql/minikql/comp_nodes/mkql_grace_join_imp.cpp
@@ -14,13 +14,10 @@ namespace NMiniKQL {

 namespace GraceJoin {

-static std::atomic<ui64> GlobalTuplesPacked = 0;
-static std::atomic<ui64> GlobalTuplesDeleted = 0;

 void TTable::AddTuple(  ui64 * intColumns, char ** stringColumns, ui32 * stringsSizes, NYql::NUdf::TUnboxedValue * iColumns ) {

     TotalPacked++;
-    GlobalTuplesPacked++;

     TempTuple.clear();
     TempTuple.insert(TempTuple.end(), intColumns, intColumns + NullsBitmapSize_ + NumberOfKeyIntColumns);
@@ -1174,7 +1171,6 @@ TTable::TTable( ui64 numberOfKeyIntColumns, ui64 numberOfKeyStringColumns,
 }

 TTable::~TTable() {
-    GlobalTuplesDeleted += TotalPacked;
 };
```

`perf top`

perf top

osq_lock in perf top - what is it?

stackcount

```
@[
  osq_lock+1
  rwsem_optimistic_spin+66
  rwsem_down_write_slowpath+155
  down_write_killable+82
  vm_mmap_pgoff+162
  ksys_mmap_pgoff+273
  do_syscall_64+72
  entry_SYSCALL_64_after_hwframe+68
]: 330025
```

Memory Allocator

- Optimized for Concurrency
- Allocator Per Query
  - Isolation of Memory Usage
  - Efficient Memory Allocation and Release
  - Simplified Debugging and Profiling

Memory Allocator

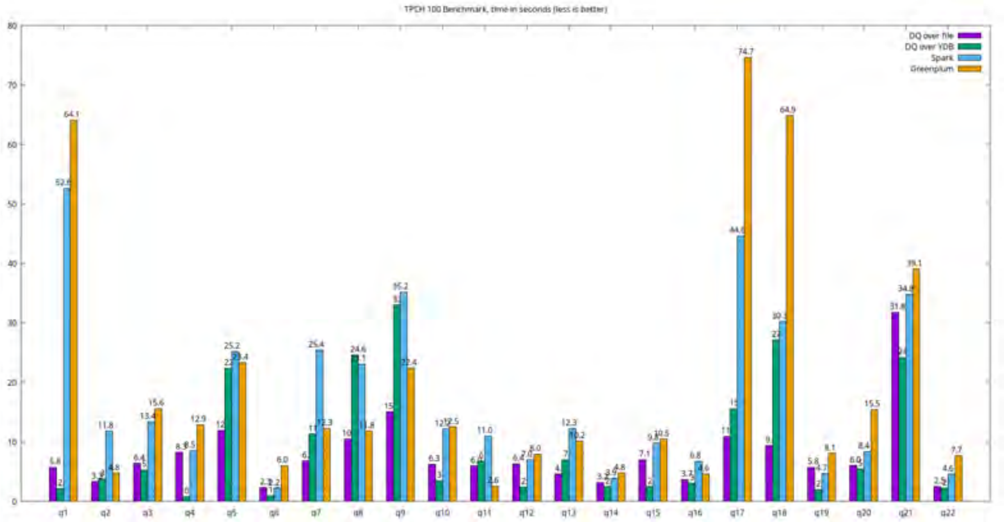
- Allocate 32 pages at once for improved efficiency[9].
- Q20 TPC-H 100 performance improved from 36s to 27s.

```
-        void* mem = ::mmap(nullptr, size + POOL_PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, 0, 0);
+        auto allocSize = size + ALLOC_AHEAD_PAGES * POOL_PAGE_SIZE;
+        void* mem = T::Mmap(allocSize);
```

[9] https://github.com/ydb-platform/ydb/commit/b7e0a08cab9583cb83546494333d0c0f87260be2

Results

- Execution times: YQL - 154s, YDB - 209s, Greenplum - 436s, Spark - 388s



1. Overview & background information
2. Testing methodology
3. Investigations
4. Containerization and performance

User Code Isolation

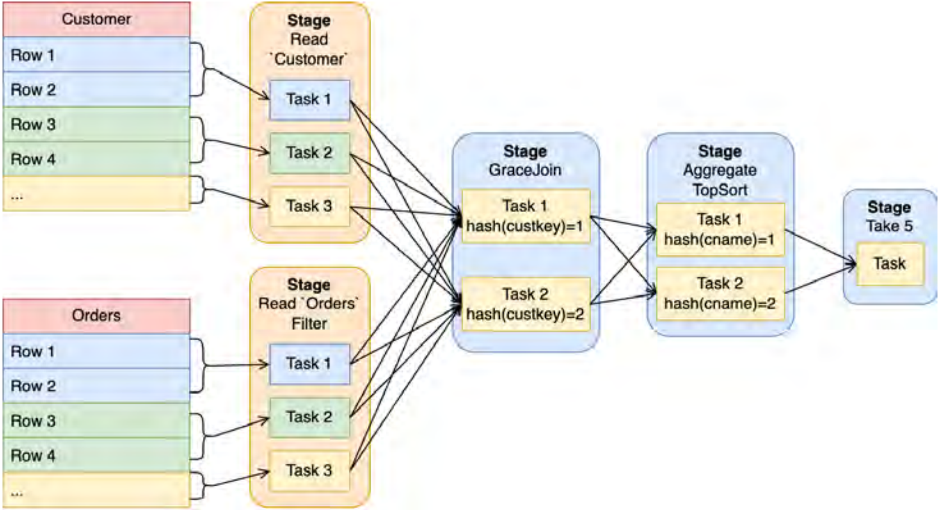
```
$f=Python3::f(@@
def f(x):
    """
    Callable<(Int32)->Int32>
    """
    import ctypes
    print(ctypes
        .cast(1, ctypes.POINTER(ctypes.c_int))
        .contents)
    return 0
@@);

select $f(0);
```

User Code Isolation

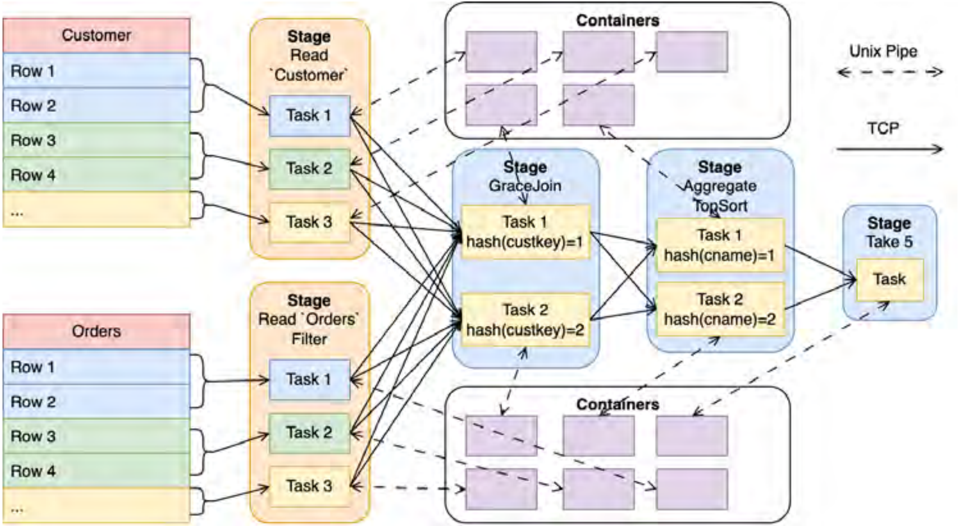
```
$f=Python3::f(@@
def f(x):
    """
    Callable<(Int32)->Int32>
    """
    import ctypes
    print(ctypes
       .cast(1, ctypes.POINTER(ctypes.c_int))
       .contents)
    return 0
@@);

select $f(0);
```

```
Container killed by signal: 11 (Segmentation fault)
?? at .../b4382c8e-78fcb74c-519140b6-33:0:0
Simple_repr at .../_ctypes.c:4979:12
PyObject_Str at .../object.c:492:11
PyFile_WriteObject at .../fileobject.c:129:17
builtin_print at .../bltinmodule.c:2039:15
cfunction_vectorcall... at .../methodobject.c:443:24
PyObject_Vectorcall at .../pycore_call.h:92:11
_PyEval_EvalFrameDefault at .../ceval.c:0:0
...
```

Execution Plan

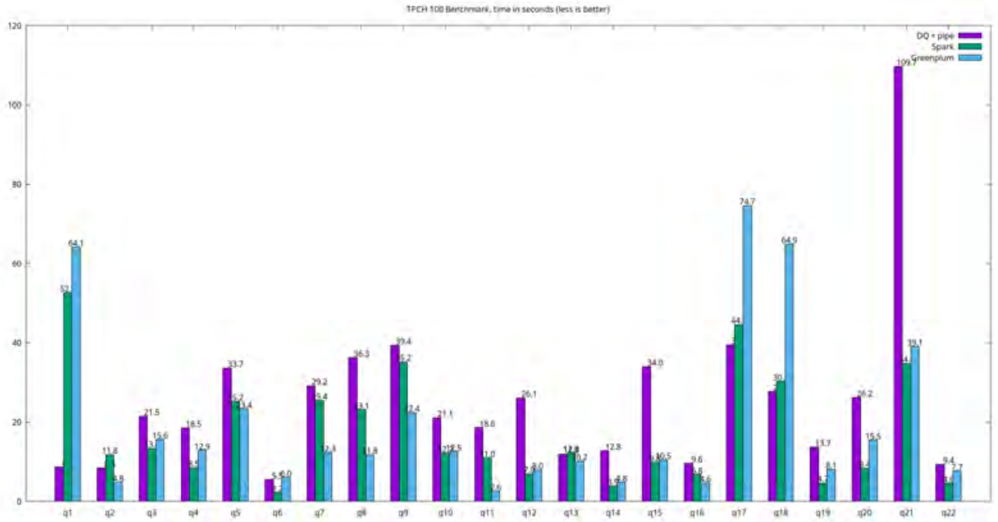


Execution Pipeline with Unix Pipe



Slow Results

- Execution times: Pipe - 561s, Spark - 388s, Greenplum - 436s



Linux IPC Performance

Analysis by Peter Goldsborough: IPC Benchmarks[10]

Method	100 Byte Messages	1 Kilo Byte Messages
Unix Signals	–broken–	–broken–
ZeroMQ (TCP)	24,901 msg/s	22,679 msg/s
Internet sockets (TCP)	70,221 msg/s	67,901 msg/s
Domain sockets	130,372 msg/s	127,582 msg/s
Pipes	162,441 msg/s	155,404 msg/s
Message Queues	232,253 msg/s	213,796 msg/s
FIFOs (named pipes)	265,823 msg/s	254,880 msg/s
Shared Memory	4,702,557 msg/s	1,659,291 msg/s
Memory-Mapped Files	5,338,860 msg/s	1,701,759 msg/s

Table: Comparison of Message Passing Performance

[10] https://github.com/goldsborough/ipc-bench

Pipe Performance

Francesco Mazzoli's article on fast pipes: Fast Pipes Analysis[11]

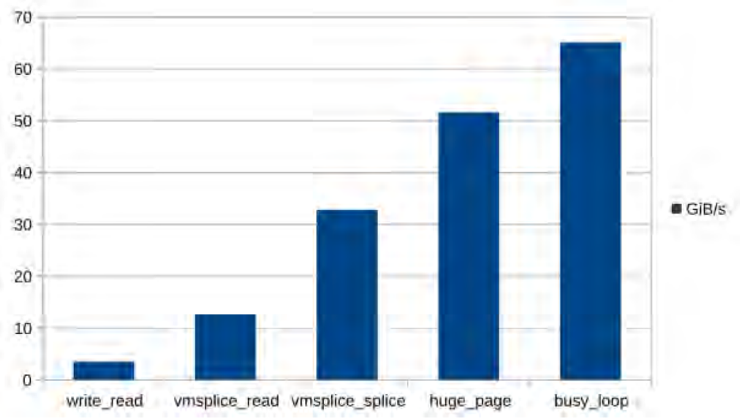
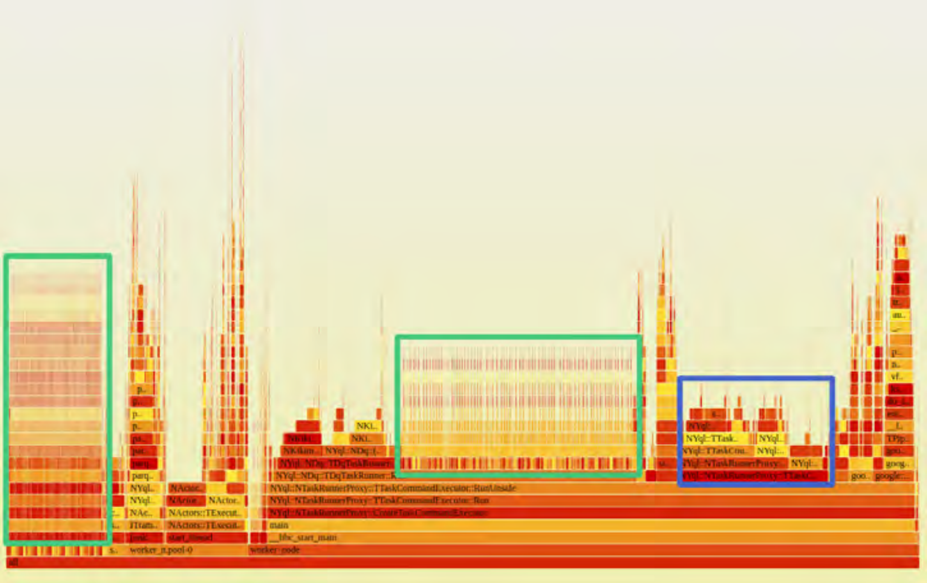


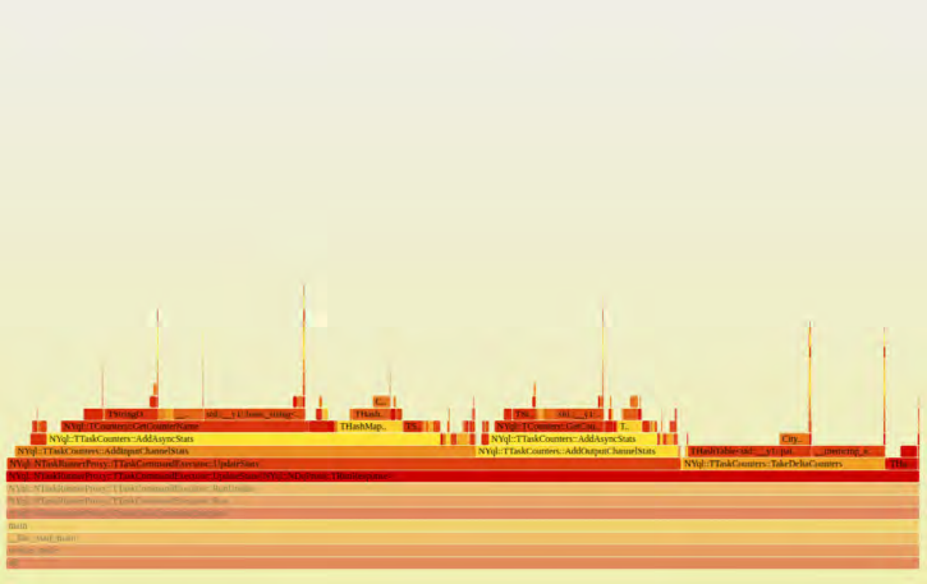
Figure: Detailed analysis of the fast pipe system from Mazzoli's research

[11] https://mazzo.li/posts/fast-pipes.html

Pipe Performance and Flamegraph

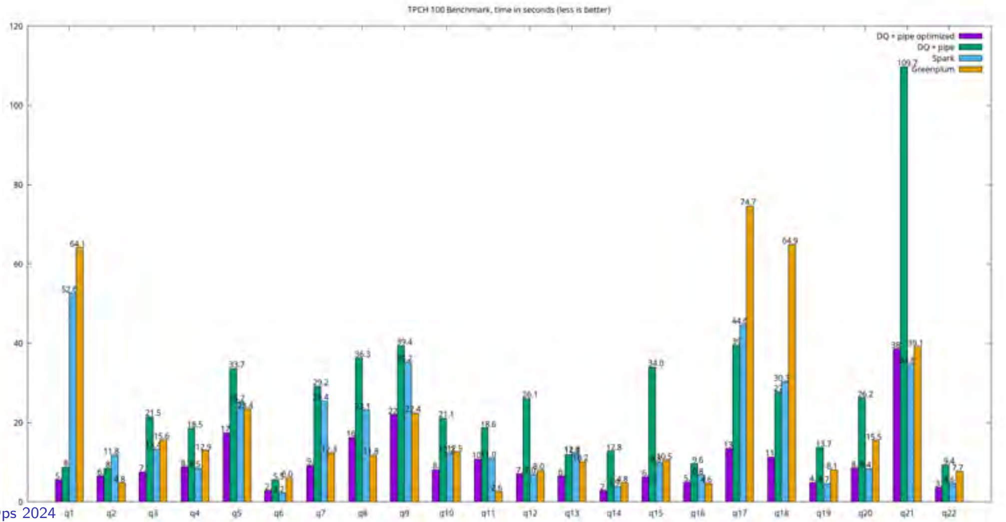


Pipe Performance and Flamegraph



Results: DQ + PIPE

- Execution times: Pipe Optimized - 223s, Pipe - 561s, Spark - 388s, Greenplum - 436s



What's Next?

- TPC-H Terabyte Scales
  - 1Tb, 10Tb, 100Tb
- TPC-DS
  - Reflects contemporary OLAP systems
  - Emphasizes the importance of plan-level optimizations

Thank You

Contact Information

GitHub: resetius

LinkedIn: alexey-ozeritskiy

YouTube: mormeoi