Unorthodox Java: Building QuestDB

High-Performance Time-Series Database

Jaromir Hamala

QuestDB Engineering Team

About Me

- **Jaromir Hamala** - QuestDB Engineering Team
- Passion for concurrency, performance, and distributed systems
- Working on QuestDB
- Before: Hazelcast, C2B2 - the birthplace of Payara AS

What is QuestDB?

- **Open-source time-series database** (Apache License 2.0)
- SQL with time-series extensions
- PostgreSQL Wire Protocol compatible
- High-speed ingestion: InfluxDB line protocol

GitHub: https://github.com/questdb/questdb

Live Demo: https://demo.questdb.io

The Numbers

What do we mean by "high-performance"?

- **Millions** of rows ingested per second
- Query **billions** of rows efficiently

> No magic, just hard work and clever engineering

Demo

Language Breakdown

Implementation

- **90%** Java
- **10%** C/C++/Rust

But... Unorthodox Java!

- In-house standard library
- Zero GC on hot path
- JNI when needed
- SIMD optimizations

The Big Question

Why Java at all?! 🤔

Let's explore why we chose Java and how we made it work for high-performance computing...

Java is great!

It gives us

- **Community** - People who know it well
- **Tooling** - Excellent profilers, debuggers, IDEs
- **JIT Compiler** - Adaptive optimization at runtime

But also: Rust wasn’t an option yet

```
commit 95b8095427c4e2c7814ad56d06b5fc65f6685130
Author: bluestreak01 <bluestreak@gmail.com>
Date:   Mon Apr 28 16:29:15 2014 -0700

    Initial commit
```

Rust 1.0, was published on May 15, 2015

QuestDB Design Principles

Core Philosophy

1. No Allocation on Hot-Path
2. Know your memory layout
3. Core infrastructure MUST NOT allocate
4. No 3rd party Java libraries

The founder has a background in High Frequency Trading

Why Zero-GC?

We have ZGC, Shenandoah, Azul C4 but...

- Concurrent collectors still cost CPU cycles
- And memory bandwidth
- And application throughput (barriers)

What This Led To...

Three Core Disciplines

1. **Memory Discipline** - Zero allocation, off-heap patterns
2. **Execution Discipline** - Custom JIT, runtime bytecode generation
3. **Concurrency Discipline** - Lock-free algorithms, sharding

```
┌──────────────────────┐
│ Memory Discipline    │
├──────────────────────┤
│ • Zero Allocation    │
│ • Off-heap Memory    │
│ • Flyweight Pattern  │
└──────────────────────┘
         ↓
┌──────────────────────┐
│Execution Discipline  │
├──────────────────────┤
│ • Custom JIT         │
│ • Runtime Bytecode   │
│ • SIMD Operations    │
└──────────────────────┘
         ↓
┌───────────────────────┐
│Concurrency Discipline │
├───────────────────────┤
│ • Sharded GROUP BY    │
│ • Lock-free Design    │
│ • Single Writer       │
└───────────────────────┘
```

DISCIPLINE 1: MEMORY

The Foundation of Performance

> "The fastest allocation is the one that never happens"

Frontend vs Backend Memory Strategy

Frontend

- **Parser, Planner, Optimizer**
- Pure Java objects (mostly)
- Uses **Object Pooling**
  - Example: AST nodes
- Fast allocation/deallocation

Backend

- **Runtime, JIT, Storage**
- Uses **Off-heap Memory**
- Long-lived data
- Direct memory addresses
- Easy JNI communication

Different parts, different strategies!

QDB STDLIB

- network
- preferences
- std
  - bytes
  - datetime
    - microtime
    - millitime
    - AbstractDateFormat
    - AbstractTimeZoneRules
    - DateFormat
    - DateLocale
    - DateLocaleFactory
    - FixedTimeZoneRule
    - TimeZoneRuleFactory
    - TimeZoneRules
    - Transition
    - TransitionRule
  - ex
  - fastdouble
  - filewatch
  - histogram.org.HdrHistogram
  - json
  - str
  - AbstractCharSequenceHashSet
  - AbstractIntHashSet
  - AbstractLongHashSet
  - AbstractLowerCaseAsciiCharSequenceH
  - AbstractLowerCaseCharSequenceHashI
  - AbstractLowerCaseCharSequenceHashS
  - AbstractLowerCaseUtf8SequenceHashS
  - AbstractSelfReturningObject
  - AbstractUtf8SequenceHashSet
  - AssociativeCache

- network
  - AbstractIODispatcher
  - AtomicSuspendEvent
  - DefaultIODispatcherConfiguration
  - Epoll
  - EpollAccessor
  - EpollFacade
  - EpollFacadeImpl
  - EventFdSuspendEvent
  - FDSet
  - HeartBeatException
  - IOContext
  - IOContextFactory
  - IOContextFactoryImpl
  - IODispatcher
  - IODispatcherConfiguration
  - IODispatcherLinux
  - IODispatcherOsx
  - IODispatchers
  - IODispatcherWindows
  - IOEvent
  - IOOperation
  - IORequestProcessor
  - JavaTlsClientSocket
  - JavaTlsClientSocketFactory
  - Kqueue
  - KqueueFacade
  - KqueueFacadeImpl
  - Net
  - NetworkError
  - NetworkFacade
  - NetworkFacadeImpl
  - NoSpaceLeftInResponseBufferException
  - PeerDisconnectedException
  - PeerIsSlowToReadException
  - PeerIsSlowToWriteException
  - PipeSuspendEvent
  - PlainSocket
  - PlainSocketFactory
  - QueryPausedException
  - SelectAccessor

- AssociativeCache
- AtomicIntList
- BiIntFunction
- BiLongFunction
- BinarySequence
- BitmapIndexUtilsNative
- BitSet
- BoolList
- BufferWindowCharSequence
- BytecodeAssembler
- ByteList
- Chars
- CharSequenceBoolHashMap
- CharSequenceHashSet
- CharSequenceIntHashMap
- CharSequenceLongHashMap
- CharSequenceObjHashMap
- ClosableInstance
- CompactCharSequenceHashSet
- CompactIntHashSet
- CompactUtf8SequenceHashSet
- ConcurrentAssociativeCache
- ConcurrentCacheConfiguration
- ConcurrentHashMap
- ConcurrentIntHashMap
- ConcurrentLongHashMap
- DefaultConcurrentCacheConfiguration
- DirectBinarySequence
- DirectByteSequenceView
- DirectIntList
- DirectIntSlice
- DirectLongList
- DirectLongLongAscList
- DirectLongLongDescList
- DirectLongLongSortedList
- DirectObjectFactory
- DoubleList
- Files
- FilesFacade
- FlyweightMessageContainer
- GenericLexer
- Hash
- ImmutableIterator
- Int3Sort
- Interval
- IntGroupSort
- IntHashSet
- IntIntHashMap
- IntList
- IntLongAssociativeCache
- IntLongHashMap
- IntLongSortedList
- IntObjHashMap
- IntShortHashMap
- IntSortedList
- IntStack
- IOURing
- IOUringAccessor
- IOURingFacade
- IOURingFacadeImpl
- IOURingImpl
- Long128
- Long256
- Long256Acceptor
- Long256FromCharSequenceDecoder
- Long256Impl
- Long256Util
- LongHashSet
- LongIntHashMap
- LongList
- LongLongHashMap
- LongLongHashSet
- LongMatrix
- LongObjHashMap
- LongSort
- LongVec
- LowerCaseAsciiCharSequenceHashSet
- LowerCaseAsciiCharSequenceIntHashMap

- LowerCaseAsciiCharSequenceObjHashMap
- LowerCaseCharSequenceHashSet
- LowerCaseCharSequenceIntHashMap
- LowerCaseCharSequenceObjHashMap
- LowerCaseUtf8SequenceIntHashMap
- LowerCaseUtf8SequenceObjHashMap
- MemoryPages
- MemoryTag
- Misc
- Mutable
- NanosecondClock
- NanosecondClockImpl
- NoOpAssociativeCache
- Numbers
- NumericException
- ObjectFactory
- ObjectPool
- ObjectStackPool
- ObjHashSet
- ObjIntHashMap
- ObjList
- ObjLongMatrix
- ObjObjHashMap
- ObjStack
- Os
- PagedDirectLongList
- Pool
- QuietCloseable
- ReadOnlyObjList
- Rnd
- Rosti
- RostiAllocFacade
- RostiAllocFacadeImpl
- Rows
- SelfReturningObjectFactory
- SimpleAssociativeCache
- SimpleReadWriteLock
- StationaryMillisClock
- StationaryNanosClock
- SwarUtils
- ThreadLocal
- Transient

What Our Stdlib Provides

1. **I/O** - Network and file operations, including io_uring
2. **Collections** - Specialized for primitives (no boxing!)
3. **Strings** - CharSequence-based, not String
4. **Numbers** - Fast parsing/printing

Memory Technique 1: Zero Allocation

Single-threaded pools

```
// SqlParser uses object pool for AST nodes
ObjectPool<ExpressionNode> expressionNodePool;

// Acquire nodes during parsing
ExpressionNode node = expressionNodePool.next();
node.of(...);

// Mass release after plan creation
expressionNodePool.clear(); // O(1) - just reset position!
```

Frontend optimization - parse without allocation

Demo: `SqlParser.parseTableName()`

Memory Technique 1b: Zero Allocation

Postgres Wire Protocol and `double` columns

DEMO - `PGConnectionContext.appendRecord()`

Memory Technique 1c: Zero Allocation

IPv4 to String Conversion

```sql
SELECT ip_address, CAST(ip_address AS STRING) as ip_str
```

```java
// Traditional: Creates garbage for each row
public String getIPv4String(Record rec) {
    int ipv4 = arg.getIPv4(rec)
    String s = IPUtils.formatIPv4(ipv4); // New String per row!
    return s;
}

// QuestDB: Reusable StringSink
private final StringSink sinkA = new StringSink();
public CharSequence getStrA(Record rec) {
    sinkA.clear();  // Reset, don't allocate!
    Numbers.intToIPv4Sink(sinkA, arg.getIPv4(rec));
    return sinkA;
}
```

Memory Technique 2: Off-Heap Memory

Direct Memory Access

```
// Allocate off-heap memory
long ptr = Unsafe.malloc(size);

// Direct memory operations
Unsafe.getUnsafe().putLong(ptr + offset, value);

// Manual memory management
Unsafe.free(ptr);
```

Benefits:

- No GC pressure
- Predictable memory layout
- Cache-friendly access patterns

Memory-Mapped Files (mmap)

```
// Map file directly into memory
long ptr = Files.mmap(fd, size, Files.MAP_RW);

// Read data directly from memory address
long value = Unsafe.getUnsafe().getLong(ptr + offset);

// Write data directly to memory address
Unsafe.getUnsafe().putLong(ptr + offset, newValue);
```

Benefits:

- **Zero-copy** - No data copying between kernel/userspace
- **Simple** - Kernel handles paging

Flyweight Pattern with Off-heap

```java
public class FlyweightDirectUtf16Sink implements CharSequence {
    private long ptr;  // Start of memory region
    private long lo;   // Current position
    private long hi;   // End of memory region

    // Point to existing memory - no allocation!
    public FlyweightDirectUtf16Sink of(long start, long end) {
        this.ptr = start;
        this.lo = start;
        this.hi = end;
        return this;
    }

    // Direct memory access
    public char charAt(int index) {
        return Unsafe.getUnsafe().getChar(ptr + index * 2L);
    }

    [...]
}
```

Memory Discipline Recap

- **No GC pressure** - Off-heap data is not tracked by the GC
- **Memory layout control** - Cache-friendly, predictable access
- **Zero allocation** - Reuse, don't recreate

DISCIPLINE 2: EXECUTION

Making Every CPU Cycle Count

> "Let the machine do what it does best"

Memory Layout Matters

Row vs Columnar Storage

Traditional Row Storage

```
┌──────────────────────────────────────────────────────────┐
│ Row 1: [id=1, sensor='A', temp=23.5, ts=t1]              │
├──────────────────────────────────────────────────────────┤
│ Row 2: [id=2, sensor='B', temp=24.1, ts=t2]              │
├──────────────────────────────────────────────────────────┤
│ Row 3: [id=3, sensor='A', temp=23.8, ts=t3]              │
├──────────────────────────────────────────────────────────┤
│ Row 4: [id=4, sensor='C', temp=22.9, ts=t4]              │
└──────────────────────────────────────────────────────────┘
```

Problem for analytics:

- To read all temperatures, must skip over other fields
- Poor cache utilization
- Can't use SIMD effectively

Columnar Storage

```
id column:     [1, 2, 3, 4, ...]
sensor column: ['A', 'B', 'A', 'C', ...]
temp column:   [23.5, 24.1, 23.8, 22.9, ...]
ts column:     [t1, t2, t3, t4, ...]
```

Benefits:

- Read only what you need
- Sequential memory access
- Cache-friendly
- SIMD operations on entire columns

QuestDB's Secret: Time Ordering

```
Traditional Columnar (unordered):
temp: [24.1, 22.9, 23.5, 23.8, ...]  ← Random time order

QuestDB Columnar (time-ordered):
temp: [22.9, 23.5, 23.8, 24.1, ...]
       ↑                        ↑
    Oldest                   Newest
```

Key Invariant: All columns are **physically sorted by time**

Why Time Ordering Matters

Efficient Time Filtering

```sql
SELECT avg(temp) FROM sensors
WHERE ts > now() - '1h'
```

- **Binary search** to find time range start
- Sequential read of recent data
- No index needed!

Cache Locality

- Recent data (most queried) stays hot in cache
- Natural prefetching for sequential access

What is SIMD? Single Instruction, Multiple Data

Traditional (Scalar):

```
a[0] + b[0] = c[0]  ← One operation
a[1] + b[1] = c[1]  ← One operation
a[2] + b[2] = c[2]  ← One operation
a[3] + b[3] = c[3]  ← One operation
```

SIMD (Vectorized):

```
┌───────────────────┐   ┌───────────────────┐   ┌───────────────────┐
│a[0]│a[1]│a[2]│a[3]│ + │b[0]│b[1]│b[2]│b[3]│ = │c[0]│c[1]│c[2]│c[3]│
└───────────────────┘   └───────────────────┘   └───────────────────┘
                ↑ One instruction processes 4 values! ↑
```

AVX2: 256-bit registers = 8 ints or 4 doubles at once!

AVX512: 512-bit registers

Explicit SIMD in Java I

```
JEP 338: Vector API (Incubator)
Authors Vladimir Ivanov, Razvan Lupusoru, Paul Sandoz, Sandhya Viswanathan
Owner   Paul Sandoz
Type    Feature
Scope   JDK
Status  Closed / Delivered
Release 16
Component       hotspot / compiler
Discussion      panama dash dev at openjdk dot java dot net
Effort  M
Duration        M
Relates to      JEP 414: Vector API (Second Incubator)
Reviewed by     John Rose, Maurizio Cimadamore, Yang Zhang
Endorsed by     John Rose, Vladimir Kozlov
Created 2018/04/06 22:58
Updated 2021/08/28 00:15
Issue   8201271
```

Explicit SIMD in Java II

```
JEP 508: Vector API (******Tenth****** Incubator)
Owner   Ian Graves
Type    Feature
Scope   JDK
Status  Closed/Delivered
Release 25
Component       core-libs
Discussion      panama dash dev at openjdk dot org
Effort  XS
Duration        XS
Relates to      JEP 489: Vector API (Ninth Incubator)
Reviewed by     Jatin Bhateja, Sandhya Viswanathan, Vladimir Ivanov
Endorsed by     Paul Sandoz
Created 2025/03/31 18:19
Updated 2025/05/21 21:28
Issue   8353296
```

Execution Technique 1: JIT compiled SIMD filters

Not Java Vector API - Our Own JIT!

Built with:

- **asmjit** library for code generation
- C++ backend, Java frontend
- AVX2 instructions for vectorization
- Processes 8 rows simultaneously (256-bit registers)

Example: Filter on INT column processes 8 values at once

Execution Technique 1b: JIT Architecture

SQL JIT Architecture

Frontend (Java):

- Analyzes filter suitability
- Serializes AST to IR

Backend (C++):

- Uses **asmjit** library
- Emits x86-64 machine code
- AVX2 vectorization

```
SQL Query: WHERE total_amount > 150
                    ↓
┌──────────────────────────────────────────┐
│            Frontend (Java)               │
├──────────────────────────────────────────┤
│ 1. Parse filter expression               │
│ 2. Build Abstract Syntax Tree (AST)      │
│ 3. Analyze suitability for JIT           │
│ 4. Serialize to Intermediate Rep (IR)    │
└──────────────────────────────────────────┘
                  │ IR
                  ↓
┌──────────────────────────────────────────┐
│            Backend (C++)                 │
├──────────────────────────────────────────┤
│ 1. Parse IR from Java frontend           │
│ 2. Generate x86-64 machine code          │
│ 3. Use AVX2 SIMD instructions            │
│ 4. Return function pointer to Java       │
└──────────────────────────────────────────┘
                  │ Native function pointer
                  ↓
    Vectorized filter processes 8 rows at once!
```

```java
/**
 * Writes IR of the filter described by the given expression tree to memory.
 *
 * @param node       filter expression tree's root node.
 * @param scalar     set use only scalar instruction set execution hint in the returned options.
 * @param debug      set enable debug flag in the returned options.
 * @param nullChecks a flag for JIT, allowing or disallowing generation of null check
 * @return JIT compiler options stored in a single int in the following way:
 * <ul>
 * <li>1 LSB - debug flag</li>
 * <li>2-4 LSBs - filter's arithmetic type size (widest type size): 0 - 1B, 1 - 2B, 2 - 4B, 3 - 8B, 4 - 16B</li>
 * <li>5-6 LSBs - filter's execution hint: 0 - scalar, 1 - single size (SIMD-friendly), 2 - mixed sizes</li>
 * <li>7 LSB - flag to include null checks for column values into compiled filter</li>
 * </ul>
 * <p>
 * Examples:
 * <ul>
 * <li>00000000 00000000 00000000 00100100 - 4B, mixed types, debug off, null checks disabled</li>
 * <li>00000000 00000000 00000000 01000111 - 8B, scalar, debug on, null checks enabled</li>
 * </ul>
 * @throws SqlException thrown when IR serialization failed.
 */
public int serialize(ExpressionNode node, boolean scalar, boolean debug, boolean nullChecks) throws SqlException {
    inPredicateTraverseAlgo.traverse(node,  visitor: this);
    putOperator(RET);

    ensureOnlyVarSizeHeaderChecks();
    TypesObserver typesObserver = predicateContext.globalTypesObserver;
    int options = debug ? 1 : 0;
    int typeSize = typesObserver.maxSize();
    if (typeSize > 0) {
        // typeSize is 2^n, so number of trailing zeros is equal to log2
        int log2 = Integer.numberOfTrailingZeros(typeSize);
        options = options | (log2 << 1);
    }
    if (!scalar && !forceScalarMode) {
```

compiler.cpp × avx2.h

```cpp
struct Function {
    void avx2_loop(const instruction_t *istream, size_t size, uint32_t step, bool null_check, int unroll_factor = 1) {

        for (int i = 0; i < unroll_factor; ++i) {
            questdb::avx2::emit_code(c, istream, size, values, null_check, data_ptr, varsize_aux_ptr, vars_ptr, input_index);

            auto mask = values.pop();

            //mask compress optimization for longs
            bool is_slow_zen = CpuInfo::host().familyId() == 23; // AMD Zen1, Zen1+ and Zen2
            if (step == 4 && !is_slow_zen) {
                Ymm compacted = questdb::avx2::compress_register(c, row_ids_reg, mask.ymm());
                c.vmovdqu(ymmword_ptr(rows_ptr, output_index, 3), compacted);
                Gp bits = questdb::avx2::to_bits4(c, mask.ymm());
                c.popcnt(bits, bits);
                c.add(output_index, bits.r64());
                c.vpaddq(row_ids_reg, row_ids_reg, row_ids_step);
            } else {
                Gp bits = questdb::avx2::to_bits(c, mask.ymm(), step);
                questdb::avx2::unrolled_loop2(c, bits.r64(), rows_ptr, input_index, output_index, rows_id_start_offset,
                                              step);
            }
            c.add(input_index, step); // index += step
        }

        c.cmp(input_index, stop);
        c.jl(l_loop); // index < stop
        c.bind(l_exit);

        scalar_tail(istream, size, null_check, rows_size);
        c.ret(output_index);
    }
```

JIT Performance Impact

Real Query Example

```sql
SELECT * FROM trips
WHERE total_amount > 150
AND pickup_datetime IN ('2009-01')
```

Single-thread scanning 13.5M rows (out of 1.6B total rows):

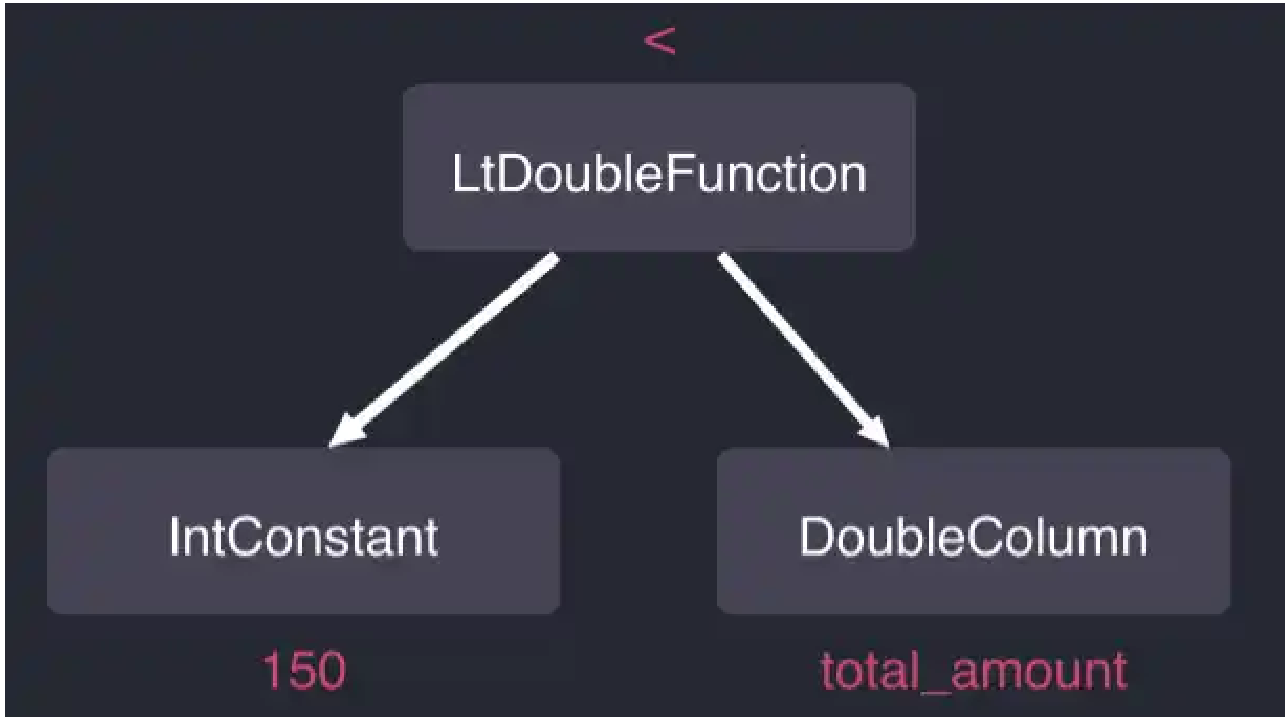
- Without JIT: 150ms (hot run)
- With JIT: 35ms (hot run)
- **76% reduction** in execution time
- **3.3 GB/s** filtering rate

Live DEMO (actually faster, because multithreaded)

Pre-JIT Filtering

- Operator function call tree
- Row-by-row processing
- Virtual method calls
- Interpreted execution

```java
public boolean hasNext() {
    while (base.hasNext()) {
        if (filter.getBool(record)) {
            return true;
        }
    }
    return false;
}
```



JIT Filtering

- Direct machine code
- Vectorized (8 rows at once)
- No virtual calls
- **But**: x86-64 specific, ARM not supported yet

11K lines of code, 250+ commits to build it!

```java
public class CompiledFilter implements Closeable {  Eugene Lysiuchenko +4

    private static final ThreadLocal<FiltersCompiler.JitError> tlJitError

    private long fnAddress;  5 usages

    public long call(  Andrei Pechkurov +2
            long dataAddress,
            long dataSize,
            long varSizeAuxAddress,
            long varsAddress,
            long varsSize,
            long rowsAddress,
            long rowsSize,
            long rowsStartOffset
    ) {
        return FiltersCompiler.callFunction(
                fnAddress,
                dataAddress,
                dataSize,
                varSizeAuxAddress,
                varsAddress,
                varsSize,
                rowsAddress,
                rowsSize,
                rowsStartOffset
        );
    }
```

Execution Technique 2: Runtime Bytecode Generation

Custom Comparators for ORDER BY

```sql
SELECT * FROM readings
ORDER BY sensor_id, batch_id DESC, timestamp
```

Problem: Generic comparator with virtual calls is slow!

Solution: Generate specialized bytecode at runtime

Traditional Generic Comparator

```
public int compare(Record a, Record b, int[] columns, int[] types) {
    for (int i = 0; i < columns.length; i++) {
        int col = Math.abs(columns[i]) - 1;
        boolean desc = columns[i] < 0;
        int cmp = 0;

        switch (types[col]) {
            case STRING:
                cmp = compareString(a.getStr(col), b.getStr(col));
                break;
            case LONG:
                cmp = Long.compare(a.getLong(col), b.getLong(col));
                break;
            case DOUBLE:
                cmp = Double.compare(a.getDouble(col), b.getDouble(col));
                break;
            // ... 20+ more types!
        }
        if (desc) cmp = -cmp;
        if (cmp != 0) return cmp;
    }
    return 0;
}
```

QuestDB: Runtime Bytecode Generation

Demo: `RecordComparatorCompiler`

```java
int maxStack = sz + (fieldIndices.size() > sz ? 4 : 0) + 3;
asm.startMethod(nameIndex, descIndex, maxStack, maxLocal: 3);

int fieldIndex = 0;
for (int i = 0; i < sz; i++) {
    if (i > 0) {
        asm.iload( value: 2);
        // last one does not jump
        branches.add(asm.ifne());
    }
    asm.aload( value: 0);
    asm.getfield(fieldIndices.getQuick(fieldIndex++));
    asm.iload( value: 1);
    int index = keyColumns.getQuick(i);
    int columnIndex = (index > 0 ? index : -index) - 1;
    asm.iconst(col
    asm.invokeInte

    if (columnTypes.getColumnType(columnIndex) == ColumnType.LONG128 || columnTypes.getColumnType(columnIndex) == ColumnType.UUID) {
        asm.aload( value: 0);
        asm.getfield(fieldIndices.getQuick(fieldIndex++));
        asm.aload( value: 1);
        asm.iconst(columnIndex);
        asm.invokeInterface(fieldRecordAccessorIndicesB.getQuick(i), argCount: 1);
    }

    asm.invokeStatic(comparatorAccessorIndices.getQuick(i));
    if (index < 0) {
        asm.ineg();
```

Generated Class Structure

```
// Generated class for ORDER BY sensor_id, batch_id DESC, timestamp
public class GeneratedComparator implements RecordComparator {

    // Fields to cache left record values
    private int f0;    // sensor_id column
    private int f1;    // batch_id column
    private long f2;   // timestamp column

    // Cache left record values
    public void setLeft(Record record) {
        this.f0 = record.getInt(0);
        this.f1 = record.getInt(1);
        this.f2 = record.getLong(2);
    }

    // Compare cached left with right record
    public int compare(Record right) {
        int cmp = Integer.compare(this.f0, right.getInt(0));
        if (cmp != 0) return cmp;
        cmp = -Integer.compare(this.f1, right.getInt(1)); // DESC!
        if (cmp != 0) return cmp;
        return Long.compare(this.f2, right.getLong(2));
    }
}
```

QuestDB: Runtime Bytecode Generation

```
// Generated at runtime for ORDER BY sensor_id, batch_id DESC, timestamp
public int compare(Record r) {
    int cmp = Integer.compare(this.f0, r.getInt(0));
    if (cmp != 0) return cmp;
    cmp = -Integer.compare(this.f1, r.getInt(1));
    if (cmp != 0) return cmp;
    return Long.compare(this.f2, r.getLong(2));
}
```

RecordComparatorCompiler generates:

- Custom class per query
- Type-specific comparison inlined
- No switches, no virtual calls, no boxing
- JVM can optimize this perfectly!

DISCIPLINE 3: CONCURRENCY

Scaling Without Contention

> "Share nothing, merge something"

Parallel GROUP BY Evolution

The Journey to Scale

From single-threaded to massively parallel execution

Single-threaded GROUP BY

```
SELECT sensor, max(temperature) FROM readings GROUP BY sensor
```

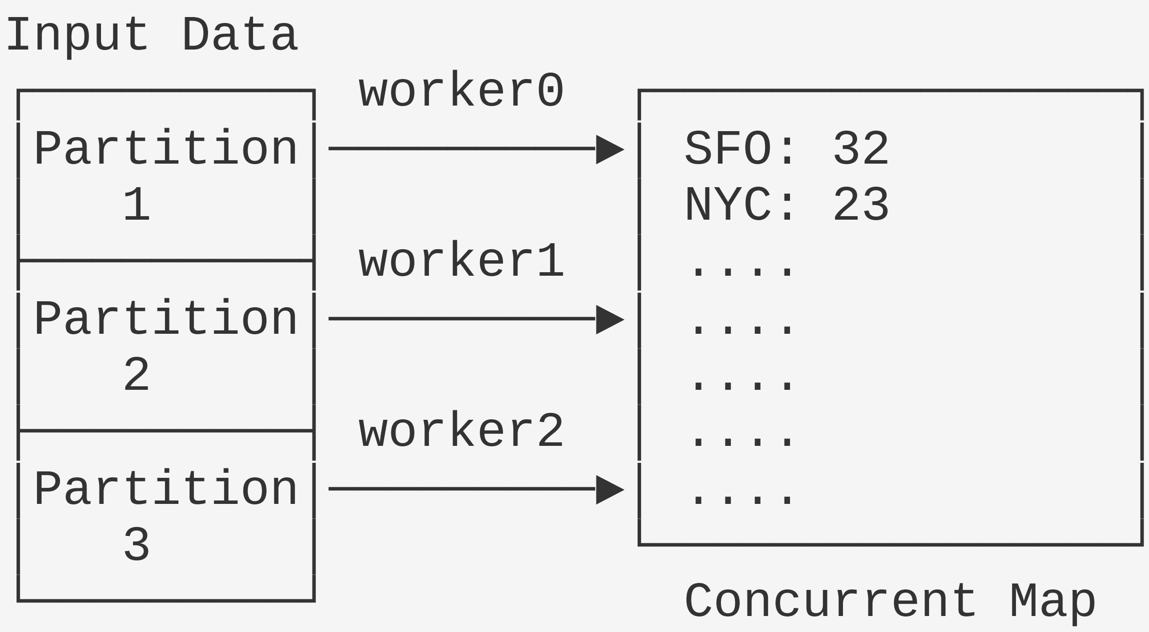
```
Input Data                         Output Map

┌─────────┐                        ┌──────────┐
│ NYC, 23 │ ─────────────────────▶ │ NYC: 23  │
│ SFO, 32 │   Single Worker        │ SFO: 32  │
│ NYC, 21 │                        │ ...      │
│ NYC, 22 │                        │ ...      │
│ ...     │                        │ ...      │
│ Row N   │                        └──────────┘
└─────────┘
```

Problem: Only uses one CPU core!

Concurrent GROUP BY?

```sql
SELECT sensor, max(temperature) FROM readings GROUP BY sensor
```



Problem: Contention on shared map

Single Writer Principle Violated!

Naive Parallel GROUP BY I

Problem: The same key is now in multiple maps!

Naive Parallel GROUP BY II

```
Input Data
┌──────────┐  worker0    ┌──────────┐
│Partition │────────────▶│HashMap 1 │  \
│    1     │             └──────────┘   \
├──────────┤  worker1    ┌──────────┐    \    merge     ┌──────────┐
│Partition │────────────▶│HashMap 2 │     ──────────────│  Result  │
│    2     │             └──────────┘    /              └──────────┘
├──────────┤  worker2    ┌──────────┐   /
│Partition │────────────▶│HashMap 3 │  /
│    3     │             └──────────┘
└──────────┘
```

Problem: Merge becomes bottleneck with high cardinality!

Solution: Sharded GROUP BY

```
Each worker creates multiple small maps (shards):

  Worker 0            Worker 1          Worker 2          Worker 3
 ┌──────────┐        ┌──────────┐      ┌──────────┐      ┌──────────┐
 │Map Shard0│        │Map Shard0│      │Map Shard0│      │Map Shard0│
 │Map Shard1│        │Map Shard1│      │Map Shard1│      │Map Shard1│
 │Map Shard2│        │Map Shard2│      │Map Shard2│      │Map Shard2│
 │Map Shard3│        │Map Shard3│      │Map Shard3│      │Map Shard3│
 └──────────┘        └──────────┘      └──────────┘      └──────────┘

Key → Shard: hash(key) % 4
```

Key property: Each key always maps to the same shard number!

Sharded GROUP BY - Parallel Merge

```
 Thread 0    Thread 1     Thread 2     Thread 3       Final Result
┌──────┐    ┌──────┐     ┌──────┐     ┌──────┐       ┌─────────┐
│Shard0│────│Shard0│─────│Shard0│─────│Shard0│──────▶│ Result0 │
└──────┘    └──────┘     └──────┘     └──────┘       └─────────┘
┌──────┐    ┌──────┐     ┌──────┐     ┌──────┐       ┌─────────┐
│Shard1│────│Shard1│─────│Shard1│─────│Shard1│──────▶│ Result1 │
└──────┘    └──────┘     └──────┘     └──────┘       └─────────┘
┌──────┐    ┌──────┐     ┌──────┐     ┌──────┐       ┌─────────┐
│Shard2│────│Shard2│─────│Shard2│─────│Shard2│──────▶│ Result2 │
└──────┘    └──────┘     └──────┘     └──────┘       └─────────┘
┌──────┐    ┌──────┐     ┌──────┐     ┌──────┐       ┌─────────┐
│Shard3│────│Shard3│─────│Shard3│─────│Shard3│──────▶│ Result3 │
└──────┘    └──────┘     └──────┘     └──────┘       └─────────┘
```

4 parallel merges instead of 1! No key appears in multiple results.

Result: No single-threaded bottleneck!

The Three Disciplines - Recap

Memory Discipline ✓

- Zero allocation patterns
- Off-heap memory management
- Flyweight views

Execution Discipline ✓

- Custom JIT compiler
- Runtime bytecode generation
- SIMD vectorization

Concurrency Discipline ✓

- Sharded algorithms
- Lock-free design
- Single writer principle

Result

Millions of rows/sec ingestion

Billions of rows queries

Near-zero GC pauses

JNI is NOT Slow!

The Secret: Pass Primitives Only!

```
// SLOW: Passing object references
native void processData(String[] data);  // Object refs = slow!

// FAST: Passing memory addresses
native void processData(long address, int length);  // Just primitives!
```

Off-heap data enables fast JNI

- Direct memory addresses (long)
- No object references
- No GC coordination needed

This is why we use off-heap memory!

Real-World Performance

Full Table Scan Example

1.6 billion rows - All taxi trips data

```sql
SELECT * FROM trips
WHERE total_amount > 150
AND passenger_count = 1
```

With JIT: Significant speedup even on cold runs!

Peak filtering rate: 9.4 GB/s (single thread)

Time-Series Specifics

SAMPLE BY Query

```sql
SELECT timestamp, avg(temperature)
FROM sensors
WHERE device_id = 'sensor1'
SAMPLE BY 1h
```

Optimized for:

- Time-ordered access
- Recent data queries
- High-cardinality aggregations

Lessons Learned

1. **Java CAN be fast** - With the right approach
2. **Hardware sympathy** - Know your CPU and memory
3. **Question conventions** - Standard library isn't sacred
4. **Batch operations** - Amortize costs
5. **Measure everything** - Benchmarks guide optimization

Why Java After All?

The Good

- **Excellent tooling** - Profilers, debuggers, IDEs
- **JIT compiler** - Adaptive optimization
- **Developer productivity** - Fast iteration

The Trade-offs

- Required deep JVM knowledge
- Built our own infrastructure
- Careful coding discipline - **super important!**

Is it worth it?

- **Maybe?** It works for us!
- **Depends on your use case** - High-performance, low-latency systems
- **Java is not the bottleneck** - It's how you use it!
- **Unorthodox Java** - Yes, but it can be done!

Community & Resources

Get Involved!

- **GitHub:** https://github.com/questdb/questdb
- **Community:** Active contributors welcome
- **Use cases:** Finance, IoT, Monitoring, Analytics

Learn More

- QuestDB documentation
- Performance blog posts
- Benchmark results

Q&A

Thank you! 🙏

Jaromir Hamala

QuestDB Engineering Team

Questions?

- Specific optimization techniques?
- Architecture decisions?
- Performance measurements?
- Getting started with QuestDB?

Bonus: Code Examples

Want to try QuestDB?

Visit: https://demo.questdb.io/index.html

Or run locally:

```
# Docker
docker run -p 9000:9000 questdb/questdb
```

Visit: http://localhost:9000