Fast, Flexible Iteration with Rust and Rhai

Jonathan Strong
Senior Developer, Keyrock

Keyrock

Pricing Assets at Scale

- Hundreds of markets
- Thousands of Instruments
- Millions of orders placed each day
- Billions in trading volume (USD)
- Trillions of market data events processed

What's the Price of an Asset?

- Price to buy $1?
- Price to buy $10k?
- Price to buy $1mm?
- Price to buy $10mm?
- Price to buy $10mm in 60s?
- Price to buy $10mm in 1s?

What's a *Good* Price?

- **Liquidity:** quantity of a given asset which can be bought or sold quickly
- **Exposure:** how much of an asset we have, which puts us at risk if its price goes down
- **Market Direction:** is the price going up or down?
- **Latency:** how stale is our view of the market? How long does it take to cancel orders?
- **Volatility:** how rapidly an asset price changes
- **Execution Risk:** software bugs, configuration errors, invalid/wrong data, and other illustrations of Murphy's Law in action
- **Fees:** exchange fees, typically per trade
- **SLAs:** client agreements to be quoting some percentage of each period
- **Manipulation:** Bad actors attempting to change prices artificially
- **Taxes, Regulations**
- **Opportunity Cost**

Liquidity: Daily Trade Volume

Different currency pairs exhibit starkly different trading characteristics

EUR/USD:	$1.8 trillion
BTC/USD:	$30 billion
XMR/USD:	$100 million
DOGE/USD:	$20k

Core Design Concept: Customized Application of Automated Trading to Highly Differentiated Individual Markets

Why Rust?

Python Backtesting Woes (jstrong)

- 75 updates per second
- 500μs per update

Time to process a given amount of historical data

- 1s	37.5ms
- 1h	135s
- 1d	54min
- 1w	6.3 hours

Using my first Rust program:

- 1w	33.4s (738ns per update)

Javascript Woes (Keyrock)

- Merged orderbook from multiple exchanges too slow
- 100ms GC pauses every 60s
- Type unsafety and runtime errors

Rhai: Seamless Rust Integration

Operate directly on native Rust types in scripting environment

```rust
// src/orderbook.rs

#[derive(Debug, Default, Clone)]
pub struct Level {
    pub price: f64,
    pub amount: f64,
}

#[derive(Debug, Default, Clone)]
pub struct OrderBook {
    bids: Vec<Level>,
    asks: Vec<Level>,
}

impl OrderBook {
    /// create or update a level on bids side of book at `price`
    /// to `amount`. if `amount == 0`, remove level.
    pub fn set_bid(&mut self, price: f64, amount: f64) { /* .. */ }

    /// create or update a level on asks side of book at `price`
    /// to `amount`. if `amount == 0`, remove level.
    pub fn set_ask(&mut self, price: f64, amount: f64) { /* .. */ }

    /// returns price of highest (best) bid
    pub fn best_bid(&self) -> Option<f64> { /* .. */ }

    /// price of lowest (best) ask level
    pub fn best_ask(&self) -> Option<f64> { /* .. */ }

    /// average of best bid, best ask
    pub fn mid(&self) -> Option<f64> { /* .. */ }
}
```

Rhai: Seamless Rust Integration (cont)

Operate directly on native Rust types in scripting environment

```
// src/orderbook.rs

impl rhai::CustomType for OrderBook {
    fn build(mut builder: rhai::TypeBuilder<Self>) {
        builder
            .with_name("OrderBook")
            .with_fn("set_bid", Self::set_bid)
            .with_fn("set_ask", Self::set_ask)
            .with_get("ask", |x: &mut Self| x.ask().unwrap_or(f64::NAN))
            .with_get("bid", |x: &mut Self| x.bid().unwrap_or(f64::NAN))
            .with_get("mid", |x: &mut Self| x.mid().unwrap_or(f64::NAN));
    }
}

// Rhai script

let orders = // .. (OrderBook instance)
let ask_multipliers = [1.001, 1.005];
let bid_multipliers = [0.999, 0.998, 0.997];
// return set of prices for trading engine to target
#{
    asks: ask_multipliers.map(|x| orders.mid * x),
    bids: bid_multipliers.map(|x| orders.bid * x),
}
```

Rhai: Seamless Rust Integration (cont)

Operate directly on native Rust types in scripting environment

- Flexible, ergonomic API to define Rhai API of Rust type
- Easily put instances of native Rust types in Rhai scope for script to read, modify
- Return native Rust type from script
- No unsafe semantic mismatch between Rust and C-based scripting engine: aligns ownership, thread synchronization paradigms

Introduction to Rhai

Rhai is a scripting language written in Rust, with a syntax similar to Rust

```
// This Rhai script calculates the n-th Fibonacci number

const TARGET = 28;
const REPEAT = 5;
const ANSWER = 317_811;

fn fib(n) {
    if n < 2 {
        n
    } else {
        fib(n-1) + fib(n-2)
    }
}

print(`Running Fibonacci(${TARGET}) x ${REPEAT} times...`);

let result;
let now = timestamp();

for n in 0..REPEAT {
    result = fib(TARGET);
}

print(`Finished. Run time = ${now.elapsed} seconds.`);

print(`Fibonacci number #${TARGET} = ${result}`);

if result != ANSWER {
    print(`The answer is WRONG! Should be ${ANSWER}!`);
}
```

Dynamic Typing

`rhai::Dynamic`

```
let x = 42;             // value is an integer
x = 123.456;            // value is now a floating-point number
x = "hello";            // value is now a string
x = x.len > 0;          // value is now a boolean
x = [x];                // value is now an array
x = #{x: x};            // value is now an object map
```

Functions

Overloading, Limited Closures

```
fn f(x) {
    x * 2.0
}

let kv = #{
    abc: 123,
    def: 456,
};

// closure
let g = |x| { kv["abc"] * x };

// `this` semantics, method “dot” syntax
fn get() { this.abc }
kv.get()

// function pointer - can be returned and
called from Rust
FnPtr("f")

// function pointer - alternative syntax
let h = f;
```

Exceptions, Try/Catch, and Throw

Unlike Rust, Rhai is not an Option/Result World

- Error handling via runtime exceptions
- A thrown exception comes back to the native Rust context as a Result::Err(e)
- As someone triggered by the fehler crate (#[throws]), I like exceptions and try/catch for Rhai

Limitations

- No classes
- No traits
- No structs (but enables use of Rust native structs via CustomType)
- No tuples
- No keyword arguments
- No async
- Limited first-class functions
- Limited closures (mutating closed-over variables is difficult)
- Requires &mut self references to Rust types used in Rhai context

Performance

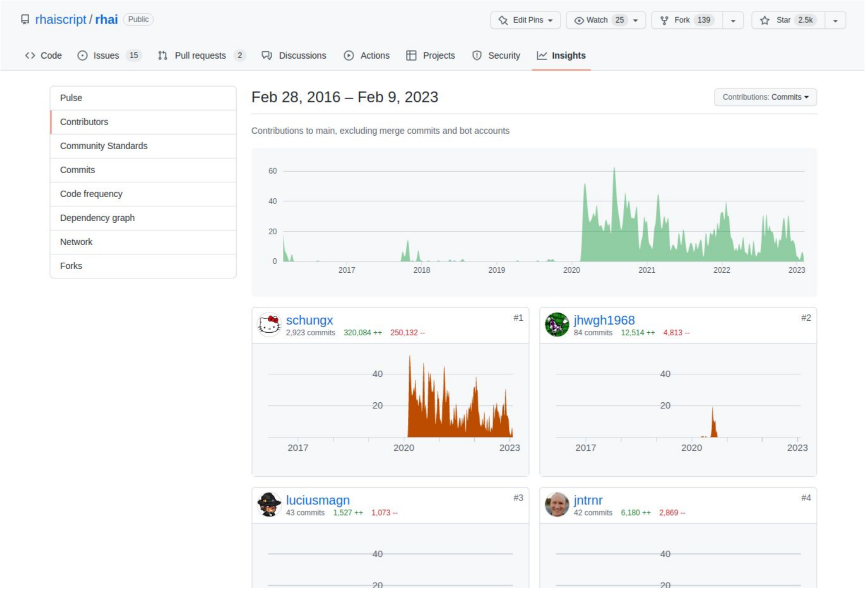
- Best used as a thin layer over Rust code
- Similar performance to Python (but most of your program will be in Rust). Slower than V8 or LuaJIT
- Can be hard to avoid .clone()s
- Compiles to AST, has optimizer, no JIT
- Code shows attention to performance
- Good documentation about performance pitfalls and how to avoid

Rhai is best used as a thin layer over Rust, with “heavy lifting” done on the native Rust side

Fine-Grained Control

- Extensive (extreme?) use of feature-gating/conditional compilation
- Provides additional, fine-grained control over how scripts are executed at runtime
- Single expression-only mode
- “Don’t Panic Guarantee - Any Panic is a Bug”
- Robust sandboxing limits: length of strings/arrays/maps, number of operations, number of modules, max call stack depth, max expression depth
- Rust ownership rules applied to objects shared between Rhai, native Rust scopes

Active Development



Led by Stephen Chung

Additional Tooling, Resources

- Rhai Book
- Online Playground
- Language Server
- rhai-doc
- REPL
- Vim Plugin
- Sublime Text Package
- VS Code Plugin
- Discord
- Zulip
- Reddit

Case Study

Custom Pricing Indicators

Indicators

- Indicators are transformations of raw market data used to inform pricing decisions
- In our usage, a broad term that encompasses prevailing prices, our current exposure to given assets, statistical aggregations, and other categories of price-related signals
- Generally not rocket scientist-invented, stochastic calculus-derived triumphs of elegant genius. Think diligent application of market expertise to constantly changing conditions

Design Goals

Why?

- Flexibility: apply customized logic to specific markets
- Hot Reloading: modify running system without bringing it down
- Empowering Non-Developers: modifying the underlying Rust system is not something we expect our trading team to do, but creating a custom indicator should be accessible/learnable

Design Constraints

Why not?

- Performance: speed kills in high frequency trading
- Traceability/Auditability: must be able to track, debug, and understand what the state of the system is and what it was at some point in the past

Custom Indicators

- Our internal term-of-art which refers to a Rhai-based indicator that applies custom logic to one or more other indicators
- In more palpable terms, a custom indicator consists of a initialization script, run once when the custom indicator is created or modified, as well as a single expression that is executed whenever its inputs change

Custom Indicator Example

```
// initialization script
let scope = #{
    max_skew: 8.0,
    max_delta: 200_000.0
};

// update script
if delta_exposure_eur_330 > scope.max_delta {
    0.0
} else if abs(balance_exposure_eur_469 / 100_000.0) > scope.max_skew {
    -sign(balance_exposure_eur_469) * scope.max_skew
} else {
    -balance_exposure_eur_469 / 100_000.0
}
```

Indicators Composition

- Internal name of our service that
  - Listens to raw market data feeds
  - Computes updated outputs for the set of existing custom indicators when their inputs change
  - Publishes those updated outputs to any subscribers
- Used heavily in production
- Thousands of custom indicators that handle billions of market events per day

Challenges

- Performance
- Tracing/Debugging: Adds Layer of State
- Human Error: NaN Propagation
- Error and Missing Data Handling
- Closure limitations make stateful event handling tricky, especially when compared to Lua/Javascript
- Striking the right balance of limited execution environment (for performance/security purposes) while not inhibiting users
- Rhai, as a new language, is still relatively arcane and unknown

Case Study

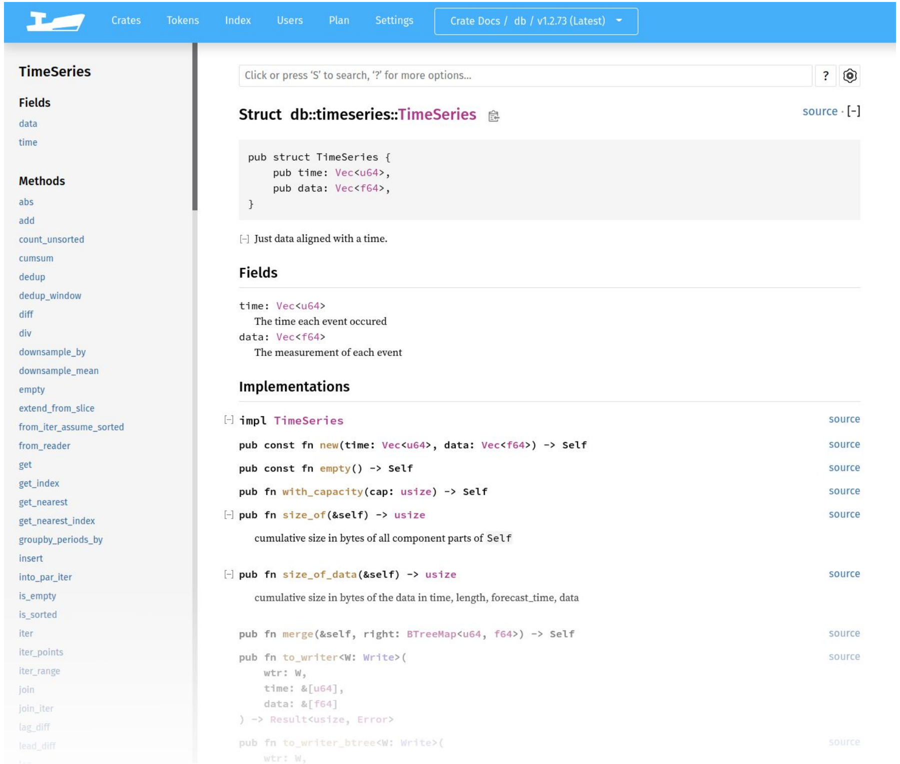
Time Series Query Language

Scenario: Rust Program with Time Series Data

Scenario: complex, high-performance Rust system which performs intensive analysis on large volume time series data

Extensive Rust functionality already exists

Rust code has what you need, but how to apply it to data, especially at runtime, is a more difficult task



Core idea: provide means to interact directly with data inside a running Rust program, without any layer in between

Enter Rhai

Easily inspect/transform/modify program state, data

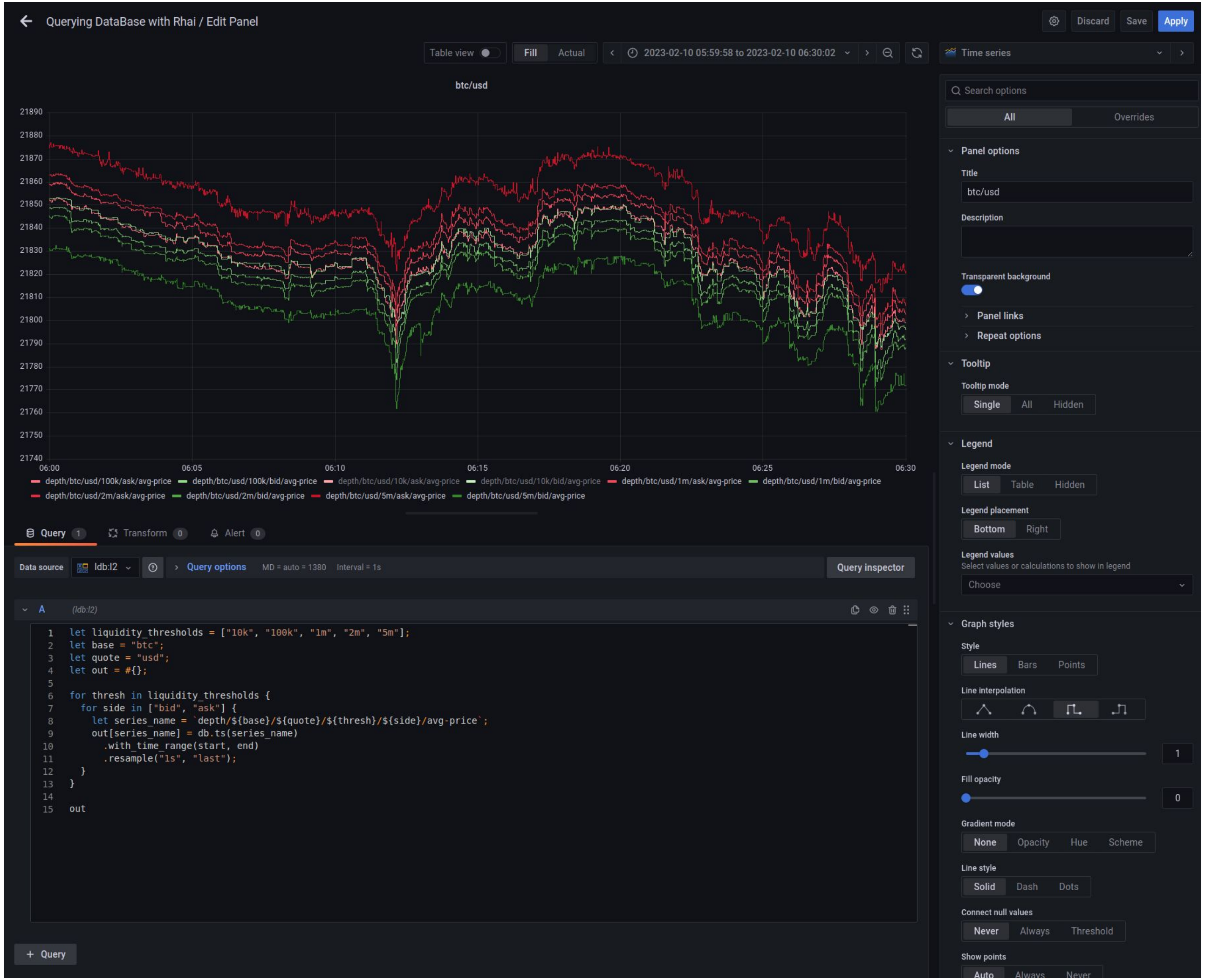
- Extending a Rhai interface to native Rust code is easy and fast
- Interact with the state of your Rust program without needing any layer in between
- Hot reloading: fast feedback cycle
- Performance issues of using a dynamic scripting language mitigated by performing heavy lifting in native Rust, which is blazing fast and guarantees thread safety
- Flexible: run source-controlled, code reviewed scripts, enable REPL- or query-like interactive functionality for ad hoc analysis, allow users to specify a single expression, etc.

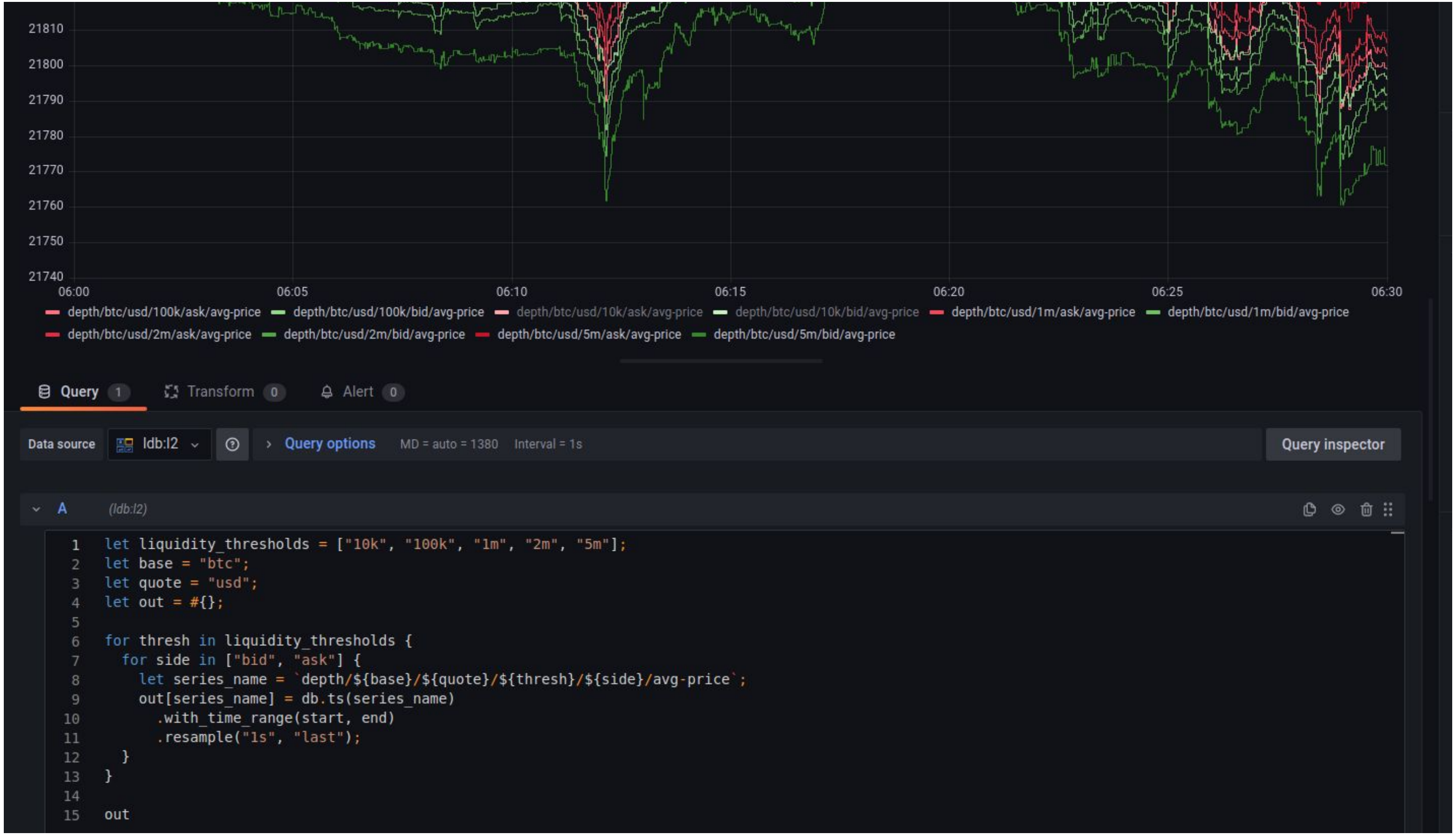
Example: Embedded Time Series Database

Simplified design for demonstration purposes

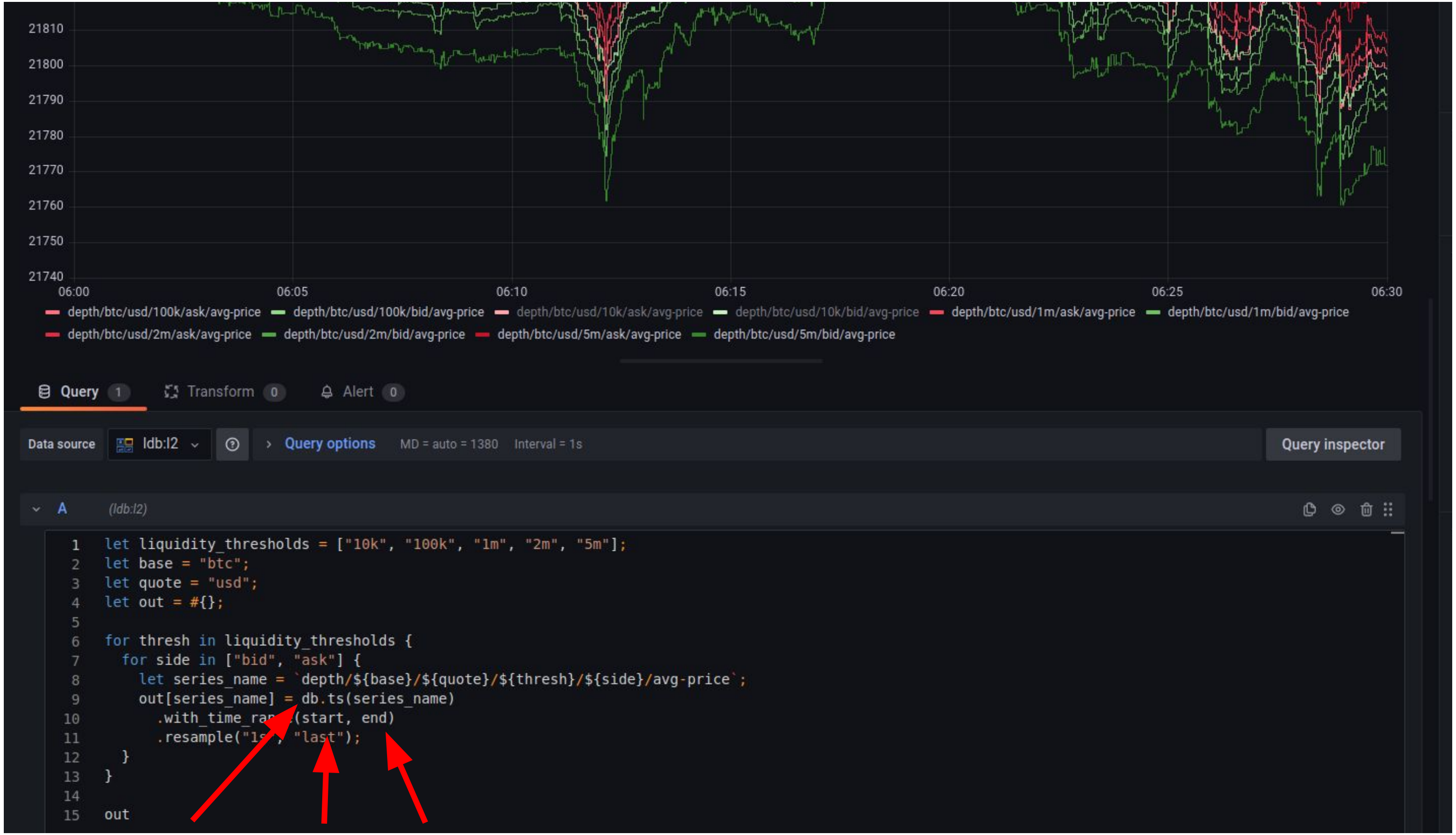
```rust
/// fetches `TimeSeries` by name from disk/cache
pub struct DataBase {
    config: Config,
    cache: Arc<DashMap<String, Arc<TimeSeries>>>,
    pool: Vec<IoWorker>,
    logger: slog::Logger,
    // ..
}

/// `self.time.len() == self.data.len()`
pub struct TimeSeries {
    /// time of event
    time: Vec<u64>,
    /// value of measurement
    data: Vec<f64>,
}
```

Custom Grafana Plugin

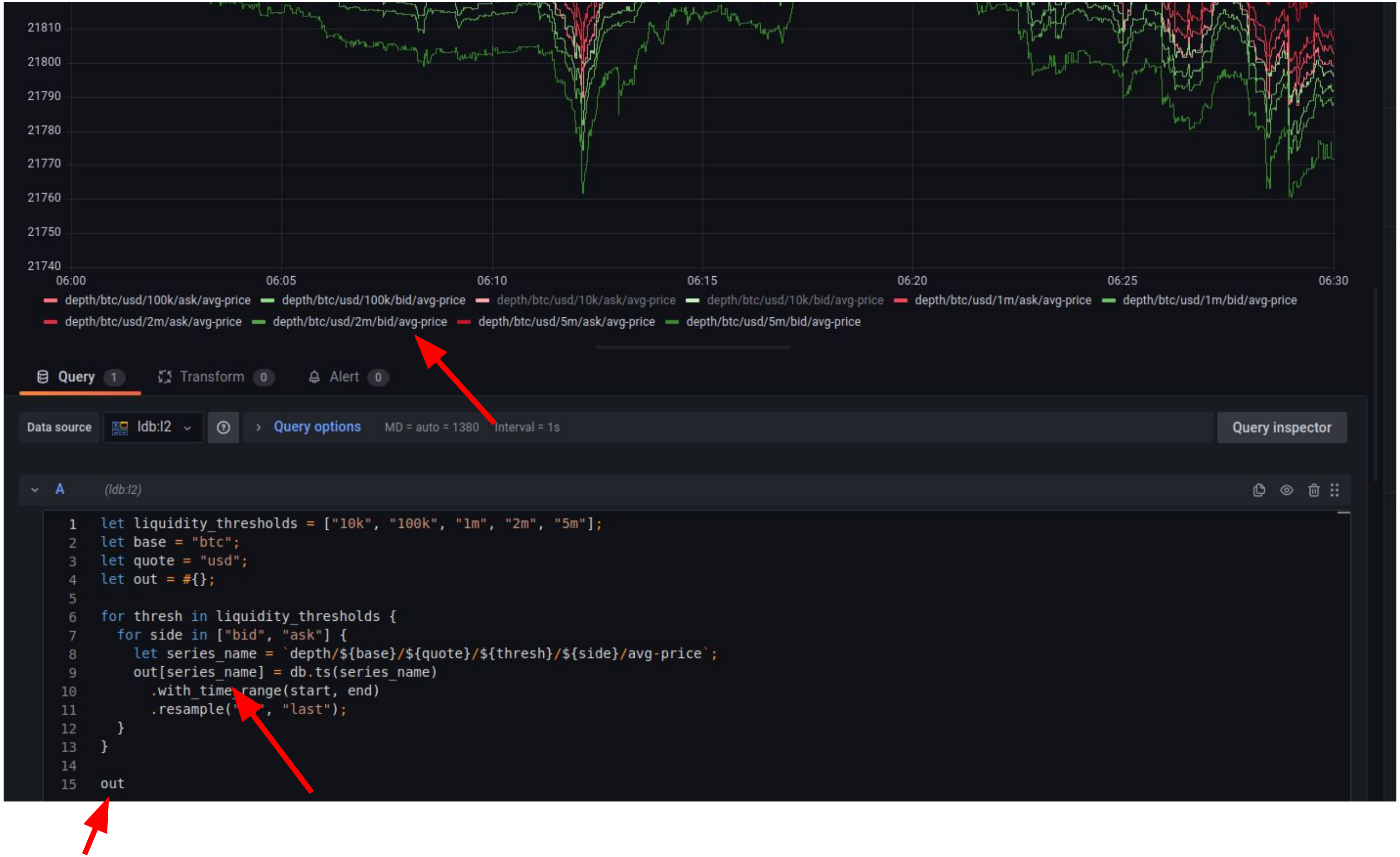




Rhai as an imperative query language



db, `start`, end in global scope by convention, expected by script



Script returns a Map<String, TimeSeries>, which is sent back to Grafana in Arrow IPC format

Query 1 | Transform 0 | Alert 0

Data source | ldb:l2 | Query options | MD = auto = 1380 | Interval = 1s | Query inspector

A (ldb:l2)

```
let liquidity_thresholds = ["10k", "100k", "1m", "2m", "5m"];
let base = "btc";
let quote = "usd";
let out = #{};

for thresh in liquidity_thresholds {
  for side in ["bid", "ask"] {
    let series_name = `depth/${base}/${quote}/${thresh}/${side}/avg-price`;
    out[series_name] = db.ts(series_name)
      .with_time_range(start, end)
      .resample("1s", "last");
  }
}

out
```

Pass name of native Rust function, so work on large collection happens entirely in native Rust code

Conclusions

Rhai’s Superpower is its Tight Rust Integration

Seamless interoperability

For Best Results, Keep as Much Work in Rust as Possible

- Design the bridge layer between Rust and Rhai to keep the heavy lifting on the Rust side
- Example: instead of calling a Rhai function on each item in a large collection, pass a function name (as a String) to Rust and use the native Rust function to perform the work