A pgv3 Server

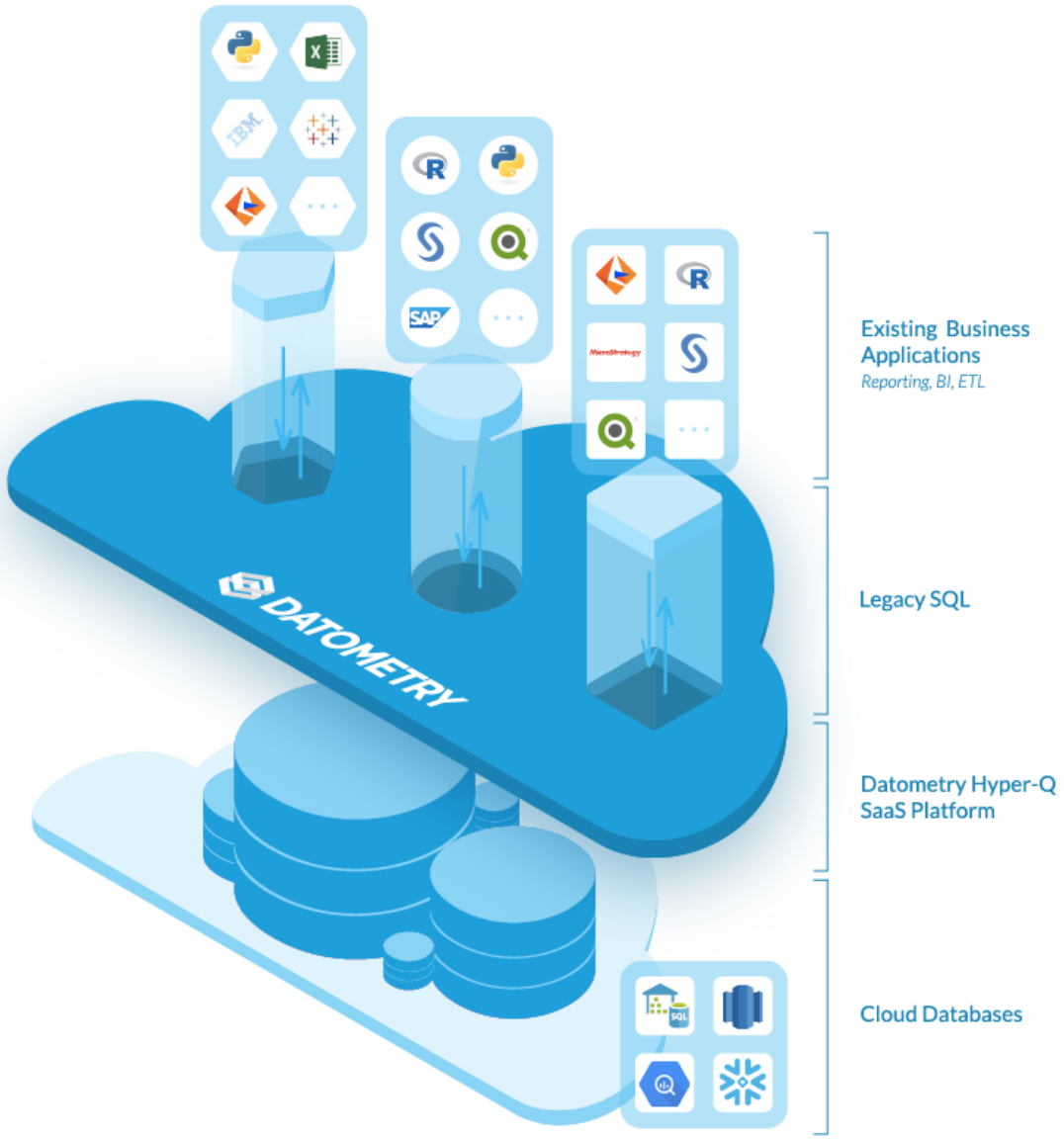
Marc Sugiyama

Software Engineer, Datometry

What is Datometry?

Datometry Hyper-Q™ is the industry's first database virtualization platform that makes databases and applications fully interoperable.

- Translate network protocols
- Cross-compile SQL to emulate features
- (almost) All Erlang



Technology Stack

- Any SQL Dialect
- ODBC/JDBC/libpq
- Erlang server-side

Why?

- Well Seasoned, Open source, Documented
  - https://www.postgresql.org/docs/current/protocol-flow.html
  - https://www.postgresql.org/docs/current/protocol-message-formats.html
- ODBC, JDBC, Reference Server-side implementation
- Prototype:
  - Better understand pgv3 protocol
  - Learn gen_statem

Typical pgv3 Messages

- Network byte order
- One byte message tag/type
- 32-bit length
- Payload depends on message type

```
<<Type:8, Length:32, Payload:Length/binary>>
```

Except that…

pgv3 Messages

- Initial messages use a request code:
  - **SSLRequest**: `<<8:32, 80877103:32>>`
  - **StartupMessage**: `<<Length:32, 196608:32, Payload:Length/binary>>`

pgv3 Messages

- Short message:

```
decode(<<Type:8, Length:32, Data/binary>>)
  when size(Data) < (Length - 4) ->
    incomplete_message.
```

- Long message:

```
decode(<<Type:8, Length:32, Data/binary>>) ->
    PayloadLength = Length - 4,
    <<Payload:PayloadLength/binary, Rest/binary>> = Data,
    {Type, Payload, Rest}.
```

Message Decoder

- Assume payload of proper length
- Shape of payload depends on message type
  - except for authentication message
- Pattern match on Type:

```
decode(<<$C, Length:32, Item:8, CloseParams/binary>>) -> ...
decode(<<$D, Length:32, DescribeParams/binary>>) -> ...
...
```

Common Patterns: NUL terminated string

```
take_string(Binary) ->
    [String, Rest] = binary:split(Binary, <<0>>),
    {String, Rest}.

take_strings(<<0, Rest/binary>>, Acc) ->
    {lists:reverse(Acc), Rest};

take_strings(Binary, Acc) ->
    {String, Rest} = take_string(Binary),
    take_strings(Rest, [String | Acc]).
```

Common Pattern: List of int16, int32

```
decode_list16(<<Cnt:16,
                Vals:Cnt/binary-unit:16,
                Rest/binary>>) ->
    {take_int16_list(Vals), Rest}.

take_int16_list(Binary) ->
    [Value || <<Value:16>> <= Binary].
```

Example: Bind

Byte1('B') – Bind command, **Int32** – Message length
String – Portal name, **String** – Prepared statement name
Int16 – Count of parameter format codes
List of Int16 - The parameter format codes

For each parameter:

Int32 – Length of parameter value
Bytes – parameter value

Int16 – Count of result column format codes
List of Int16 – The result-column format codes

Complex Payloads, E.g., Bind

```
decode2($B, Payload) ->
    {Portal, Rest1} = take_string(Payload),
    {Prepd, Rest2} = take_string(Rest1),
    <<FmtCnt:16, FmtB:FmtCnt/binary-unit:16, ValCnt:16,
        Rest3/binary>> = Rest2,
    ParamFmts = take_int16_list(FmtB),
    {Values, Rest4} = take_values(ValCnt, Rest3),
    <<ResFmtCnt:16,
        ResFmtsB:ResFmtCnt/binary-unit:16>> = Rest4,
    ResFmts = take_int16_list(ResFmtsB),
    {'Bind', Portal, Prepd, ParamFmts, Values, ResFmts};
```

Ambiguous Messages

- pgv3 Authentication response
- Resolved by knowing protocol handshake state
- Decode payload in the Protocol Handler

Encoding Responses

- Response messages structured like request messages:

```
<<Type:8, Length:32, Payload/binary>>
```

- Variable length messages:

```
set_length(Type, Payload) ->
    Length = size(Payload) + 4,
    <<Type:8, Length:32, Payload/binary>>.
```

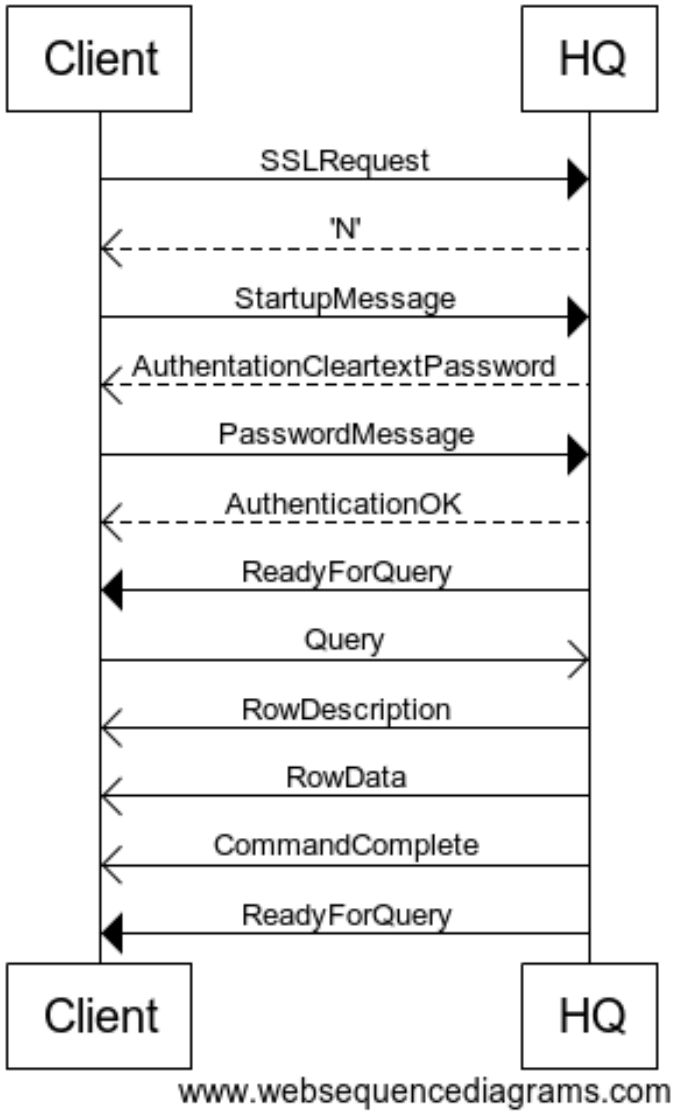
- Binary is a contiguous block of memory
  - avoid **memcpy**
  - Use **iolists**

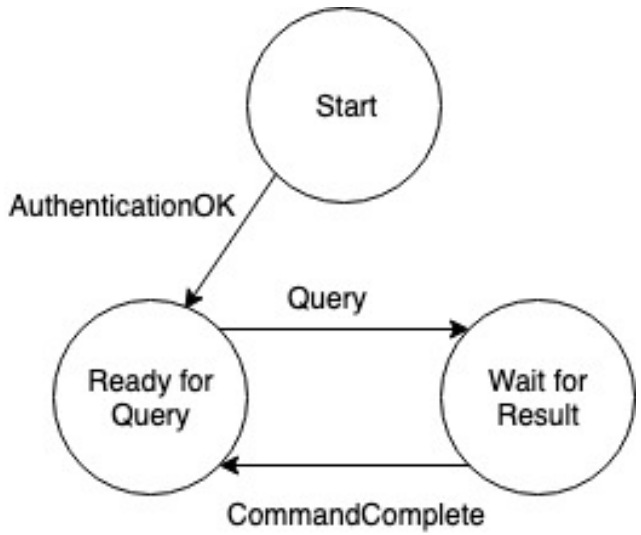
Connection Handler

```
gen_tcp:listen, gen_tcp:accept,
handle_info({tcp, Socket, Data}, ...)
```

- Coalesces multiple **gen_tcp** messages into complete pgv3 messages
- Splits **gen_tcp** message into individual pgv3 messages
- Successfully decodes pgv3 message
- Passes messages one-by-one to Protocol Handler

pgv3 Protocol





gen_statem

- OTP behavior for finite state machines
- Terminology: **State** of State Machine versus Loop **Data**
- Two modes:
  - 'state_functions' – State is function callback
  - 'handle_event_function' – Single handle_event function callback
- Cleaner API than **gen_fsm**

gem_fsm and gen_statem

gen_fsm caller	gen_fsm handler	gen_statem caller	gen_statem handler
send_event	State(Event, Data)	cast	handle_event(cast, Event, State, Data)
sync_send_event	State(Event, From, Data)	call	handle_event({call, From}, Event, State, Data)
erlang:send	handle_info	erlang:send	handle_event(info, Event, State, Data)
send_all_state_event	handle_event	cast	handle_event(cast, Event, _AnyState, Data)
sync_send_all_state_event	handle_sync_event	call	handle_event({call, From}, Event, _AnyState, Data)

handle_event/4

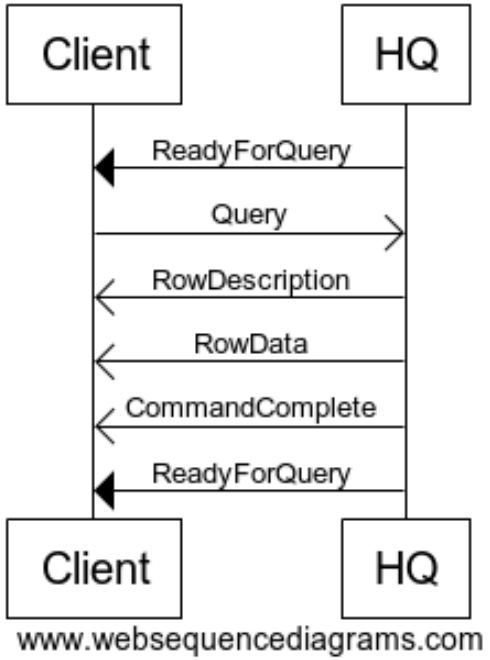
```
handle_event(EventType, EventContent, State, Data)
```

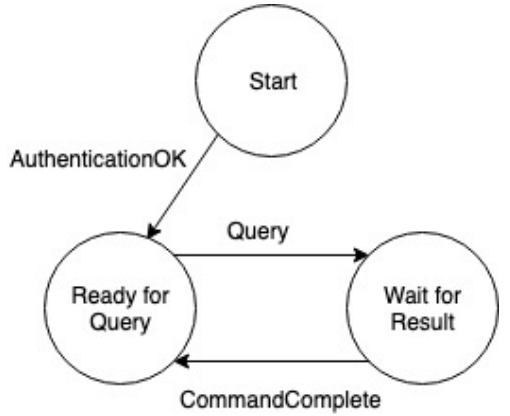
- EventType
  - `gen_statem:call -> {call, From}`
  - `gen_statem:cast -> cast`
  - `erlang:send -> info`

```
handle_event(EventType, EventContent, _AllState, Data)
```

Example: Simple Query

1. gen_tcp:send('ReadyForQuery')
2. handle_event('Query',
   Ready For Query) -> Wait for Result
3. handle_event({send, 'RowDescription'},
   Wait for Result) -> Wait for Result
4. handle_event({send, 'RowData'},
   Wait for Result) -> Wait for Result
5. handle_event({send, 'CommandComplete'},
   Wait for Result) -> Ready For Query





Receive Query

```
handle_event(cast,
  {message, {'Query', Query}},
  ready_for_query, LS) ->
    opg_server:query(self(), Query),
    {next_state, simple_reply_until_complete, LS};
```

Send Row Description

```
handle_event(cast,
  {reply, {row_description, Row, EncodeType}},
  simple_reply_until_complete, LS0) ->
    RowDesc = row_description(Row, EncodeType),
    LS1 = respond({'RowDescription', RowDesc}, LS0),
    {next_state, simple_reply_until_complete, LS1};
```

Send Row Data

```
handle_event(cast,
    {reply, {row_data,
      Rows, RowDescription, EncodeType}},
    simple_reply_until_complete, LS0) ->
    LS1 = lists:foldl(
     fun(Row, LS) ->
      respond(
         {'DataRow', Row, RowDescription, EncodeType},
         LS)
     end, LS0, Rows),
    {next_state, simple_reply_until_complete, LS1};
```

Send Command Complete

```
handle_event(cast,
    {reply, {complete, {select, Count}}},
    simple_reply_until_complete, LS0) ->
    LS1 = respond(
        {'CommandComplete', select, Count}, LS0),
    {next_state, ready_for_query, LS1};
```

'enter' event type

- Server sends 'ReadyForQuery' when ready for next query
- Use 'enter' event type

```
handle_event(enter, _OldState, ready_for_query, LS0) ->
    LS1 = flush({'ReadyForQuery', idle}, LS0),
    {next_state, ready_for_query, LS1};
```

gen_tcp, gen_statem, messages, protocol

- Messages are the vocabulary
- Protocol is the grammar
- Connection Handler – get messages from **gen_tcp**, send decoded pgv3 messages to the Protocol Handler
- Protocol Handler – tracks pgv3 protocol handshakes, drives the "server"

OCBC, JDBC, pqlib

- PG SQL leaks into pgv3 implementation
  - Metadata lookup
  - Parameterize Queries
  - Session settings

Lessons Learned

- Ambiguous message tags
- Inconsistent message tags
- Unspecified field length
- Difficult to combine connection and protocol handlers
- Message chunking
- Synchronous acknowledgements and network performance

Thank You

marc@datometry.com

Visit Datometry to learn about our technology

https://datometry.com/