DConf 2016
Berlin

DCONF 2016

std.database

(a proposed interface & implementation)

Erik Smith

THE RELATIONAL MODEL

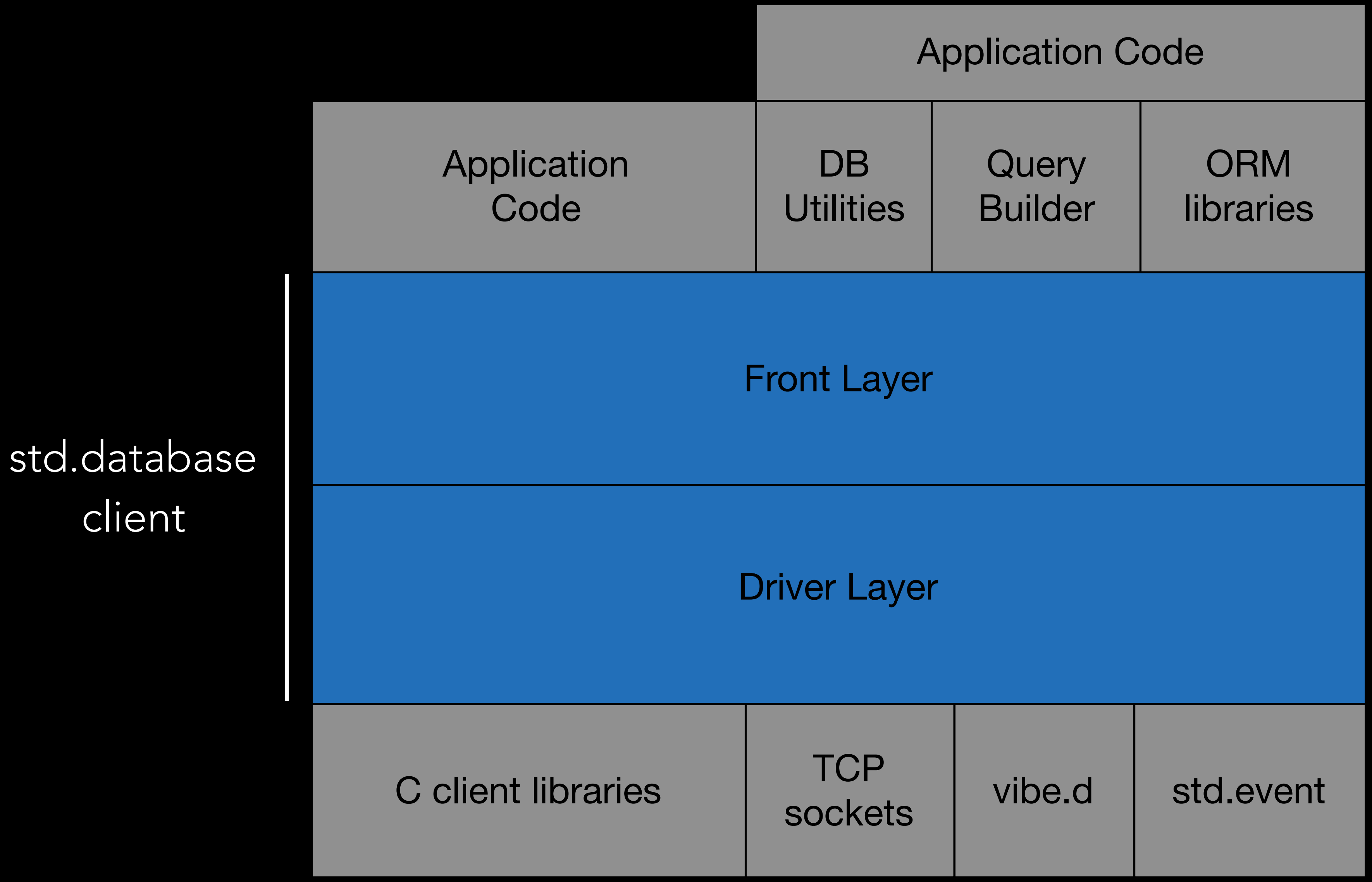
Table

	PERSON			
columns	ID INT	LAST_NAME VARCHAR(30)	FIRST_NAME VARCHAR(30)	BIRTHDATE DATE
rows	1	Yokomoto	Akiko	1990-05-03
	2	Green	Marjorie	1972-02-06
	3	Hoffman	Paul	1995-07-01

SQL query

```
select last_name from person
where birthdate <"1991-01-01"
```

THE CLIENT STACK



THE TYPES

Database	top level context (shared)
Connection	connection to server (per thread)
Statement	query execution, prepared statements, input binding
ColumnSet	Column container, input-range
Column	meta data for result column
RowSet	RowContainer, input-range
Row	row accessor
Field	field accessor, type conversion.

EXAMPLE

```
import std.database.mysql;

auto db = createDatabase("mysql://server/db");
db
    .query("select * from person”)
    .rows
    .writeRows;
```

no explicit types

EXAMPLE IN EXPANDED FORM

```
auto db = createDatabase("mysql://server/db");
auto con = db.connection;
auto stmt = con.statement("select * from person“);
auto rows = stmt.query.rows;

foreach (row; rows) {
    for(int c = 0; c != row.width; c++) {
        auto field = row[c];
        writeln(“value: ”, field);
    }
    writeln;
}
```

EXAMPLE USING COLUMN

```
auto db = createDatabase("mysql://server/db");
auto con = db.connection;
auto stmt = con.statement("select * from person");
auto rows = stmt.query.rows;

foreach (row; rows) {
    foreach (column; columns) {
        auto field = row[column];
        writeln("name: ", column.name, ", value: “, field);
    }
    writeln;
}
```

REFERENCE TYPES

```
auto getCustomers(string zip) {
    auto db = createDatabase(“mysql://server/db”)
    db
        .query(“select from person where name=?“, zip)
        .rows;
}

auto localCustomers = getCustomers(“92122”);
```

DATABASE

```
auto db = createDatabase;
auto db = createDatabase("mysql://server/db");                    // default URI

auto con = database.connection;                                   // uses default URI
auto con = database.connection("mysql://server2/db");             // specific URI
auto con = database.connection("server");                         // named source

auto con = database.createConnection;                             // new connection
auto con = database.createConnection("mysql://server2/db");

db.query("insert into person values(1, 'joe');
```

connection: returns same connection per thread
createConnection: new connection for same thread

CONNECTION STRING

URI based string

db.connection(“mysql://server/db?username=app”)

Named source

db.connection(“mysql”)

config file

Custom Resolver →

```
 "databases": [
   {
     "name": "mysql",
     "type": "mysql",
     "server": "127.0.0.1",
     "database": "test",
     "username": "",
     "password": ""
   },
```

CONNECTION

```
auto db = connection.database;      // parent

connection.autoCommit(false);       // for transactions
connection.begin;
connection.save;
connection.commit;
connection.rollback;
connection.isolationLevel;
```

TRANSACTIONS AND SCOPE

```
auto con1 = db.connection("server1").autoCommit(false);
auto con2 = db.connection("server2").autoCommit(false);

scope(failure) con1.rollback;
scope(failure) con2.rollback;

con1.begin.query("insert into account(id,amount) values(1,-500000)”);
con2.begin.query("insert into account(id,amount) values(2, 500000)");

con1.commit;
con2.commit;
```

STATEMENT

Prepared Query / Input Binding

```
auto stmt = con
    .query(“insert into person(id, name) values(?,?)“);

stmt.query(1,”Doug”);
stmt.query(2,”Cathy”);
stmt.query(3,”Robert”);
```

STATEMENT

```
stmt.rows       // a RowSet
stmt.results    // a range of RowSets

stmt.into
```

FLEXIBLE

these are equivalent

```
db.query("select * from t”)

db.connection.query(“select * from t”)

db.connection.statement(“select * from t”).query

db.connection.statement(“select * from t”).rows
```

hard to get wrong

ROWSET

a RowSet is an InputRange

```
bool empty()
Row front()
void popFront()


row.width          // number of columns in row set
row.columns        // range of Columns
row.length         // number of results (if materialized)
```

ROW

```
auto field = row[0];                // by column index

auto field = row[“FIRST_NAME”];  // by column name

auto field = row[“first_name”];     // case insensitive

auto field = row[column];           // by column
```

FIELD

```
field.toString

field.as!T          // type T
field.as!int        // int
field.as!long       // long
field.as!string     // string
field.as!Date       // as std.datetime.Date

field.as!Variant // as std.variant.Variant  (nothrow)
field.get           // as Nullable!T  (nothrow)
field.option        // as Option!T  (nothrow)
```

FIELD ACCESSORS

```
field.isNull      // is the value null
field.name        // name of column
field.type        // type enumeration
```

SINGLE ROW QUERIES WITH into

```
string a;
int b;
Date d;
Variant d;
db
    .query("select a,b,c,d from table")
    .into(a,b,c,d);
```

ROW INDEXING

```
auto field = row[“name”];     // easy but less efficient

auto field = row[1];          // efficient but less readable
```

MIXIN HELP

mixin expansion

```
auto idIndex = rows.columns[“id”];
auto nameIndex = rows.columns[“last_name”];
```

```
mixin(rows.scatterColumns);
foreach(row; rows) {
    auto id = row[lastNameIndex];
    auto name = row[idIndex];
}
```

ROW LEVEL into

```
auto rows =
    .query("select id.name from person")
    .rows;

int id;
string name;
foreach (row; rows) row.into(id, name);
```

MORE INTO

```
query.into(range);          // output-range

query.into(myStruct);       // serialization
                            // with UFCS
```

POLYMORPHIC INTERFACE

Direct Interface

```
import std.database.mysql;
auto db = createDatabase;
auto con = db.connection("mysql://server/db");
```

Poly Interface

```
import std.database;
auto db = createDatabase;
auto con1 = db.connection(“mysql://server/db");
auto con2 = db.connection(“sqlite://file.sqlite”);
```

POLY: ADDING DRIVERS

```
import std.database;

Database.register!(std.database.sqlite.Database)();
Database.register!(std.database.mysql.Database)();
Database.register!(std.database.oracle.Database)();
```

HANDLE ACCESSORS

```
auto con = db.connection;
auto mysql = connection.handle;          // typed as MYSQL*

auto mysql = db.connection.handle;       // lifetime fail
```

TEST SUITE

```
import std.database.mysql;
import std.database.testsuite;
alias DB = Database!DefaultPolicy;
testAll!DB("mysql");
```

- Templated test framework
- Runs test twice: once direct, ones through poly driver
- Runs carefully in sandbox database

OUTPUT BINDING

ID	NAME
2	JOE

- A C level buffer interface
- RowSet handles internally

ARRAY OUTPUT BINDING

```
auto rs = con
   .query("select * from t1");

int sum;
foreach(r;rs)
   sum += r[0].as!int + r[1].as!int;
```

1000 row table

A INT	B INT
0	1
1	2
2	3
...	...
999	1000

203 ms

ARRAY OUTPUT BINDING

```
auto rs = con
    .rowArraySize(100)
    .query("select * from t1");

int sum;
foreach(r;rs)
   sum += r[0].as!int + r[1].as!int;
```

32 ms

1000 row table

A INT	B INT
0	1
1	2
2	3
…	…
999	1000

ARRAY OUTPUT BINDING

```
auto rs = con
    .rowArraySize(500)
    .query("select * from t1");

int sum;
foreach(r;rs)
   sum += r[0].as!int + r[1].as!int;
```

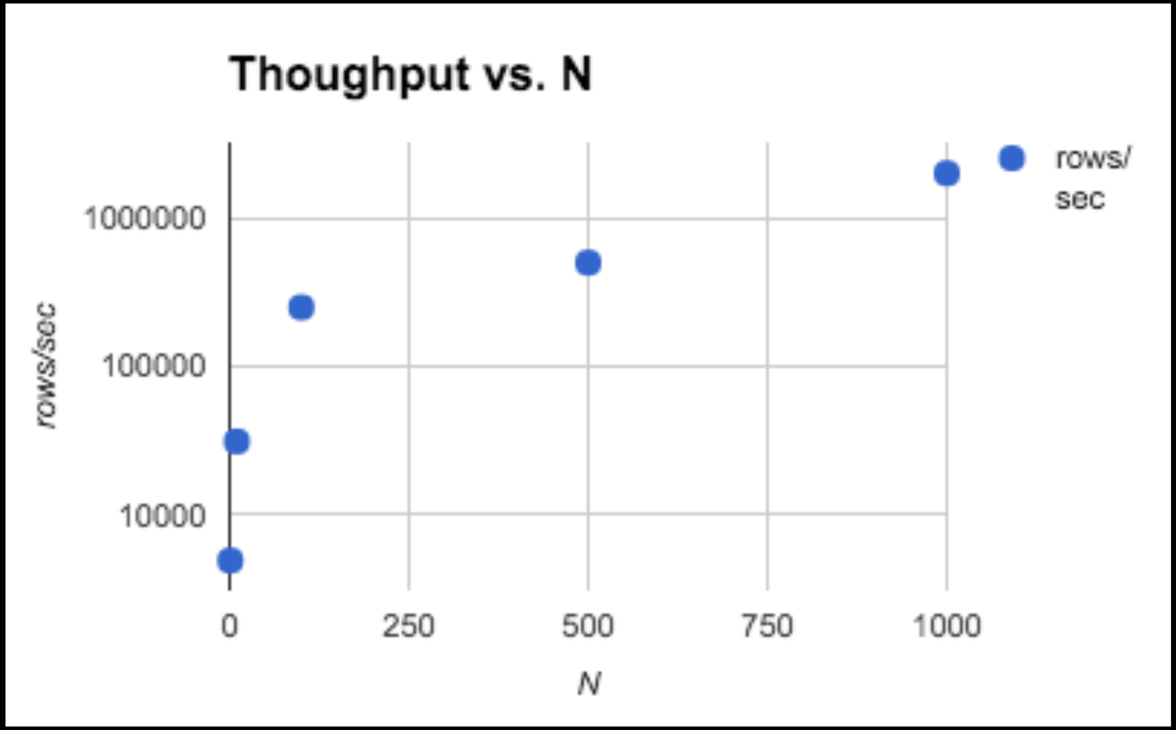
2 ms

1000 row table

A INT	B INT
0	1
1	2
2	3
...	...
999	1000

ARRAY OUTPUT BINDING

```
auto rs = con
    .rowArraySize(1000)
    .query("select * from t1");

int sum;
foreach(r;rs)
   sum += r[0].as!int + r[1].as!int;
```



601 µs

2M rows / sec

400X improvement

SKIP PARAMETERS

- Optional driver level binding mode that “stripes” bind arrays into contiguous memory

DETACHED ROWSETS

- RowSet is detachable when all rows are resident
- Detachable RowSets detach from connection
- RowSet upgraded to random-access-range
- No additional copying
- RowSet caching enabler

INPUT BINDING / PREPARED QUERY

```
auto db = createDatabase(“mysql://server/db”)
auto stmt = db
    .query(“insert into table(id, name) values(?,?)”);

foreach(d; data) stmt.query(d.id, d.name);
```

INPUT ARRAY BINDING

```
auto db = createDatabase("mysql://server/db")
auto stmt = db
    .rowArraySize(1000)
    .query("insert into table(id, name) values(?,?)");

foreach(d; data) stmt.query(d.id, d.name);
```

huge performance win

TYPE CONVERSION

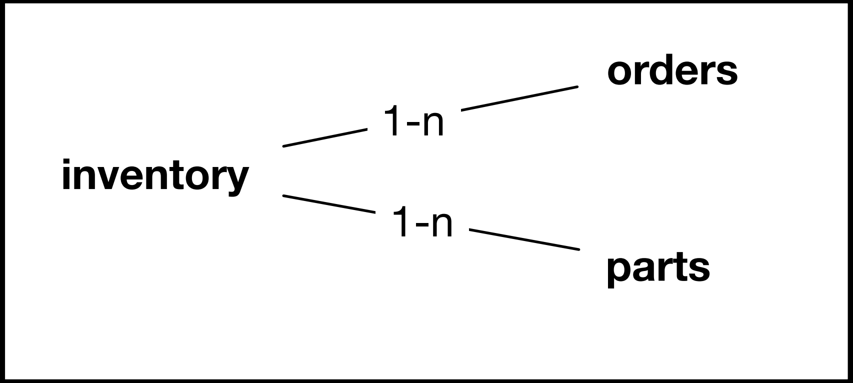
- Two layers (driver & front end)
- Native driver binding is default

POLICIES

```
struct MyPolicy {…}
auto db = Database!MyPolicy;
```

- Custom allocators
- Pluggable connection pool
- assert on release build for cross/illegal type conversions
- Scoped types (no RC)
- Omit handle accessors

UTILITY EXAMPLE: JOIN



```
auto inventory = db.createConnection
      .query("select id,* inventory").rows;
auto orders = db.createConnection
      .query("select * from orders order by id").rows;
auto parts = db.createConnection
      .query("select * from parts order by id").rows;

auto joinedRows = naturalJoin(inventory, orders, parts);
foreach(r; joinedRows) {
    int id, orderId, PartId;
    inventory.into(id,...);
    orders.into(orderId,...);
    parts.into(partId,...);
}
```

an approach to the "multiple hierarchies" problem

IMPLEMENTATION DETAILS

TWO LAYER DESIGN

Front End

- Handles reference counting details for all types
- Defines all interface functions
- Consolidates calls to the driver
- Manages state
- Connection pooling

Driver

- Implement driver specific details

DRIVER INTERFACE

```
module std.database.mysql.database;

alias Database(Policy) = BasicDatabase!(Driver!Policy,Policy);

auto createDatabase()() {return Database!DefaultPolicy();}

struct Driver(Policy) {
    struct Database {...}
    struct Connection {...}
    struct Statement {...}
    struct Result {...}
}
```

Type name correspondence
between layers

DRIVER INTERFACE

```
struct Driver(Policy) {
    struct Database {…}
    struct Connection {
        this(Database *db, Source src, Allocator *a) {…}
    }
    struct Statement {
        this(Connection *con, string sql, Allocator *a) {…}
    }
    struct Result {
        this(Statement *stmt, Allocator *a) {…}
    }
}
```

POLY DRIVER: VTABLE

```
struct StmtVTable {
    void[] function(void*, string sql) create;
    void function(void*) destroy;
    void function(void* stmt) query;
}
struct StmtGenerate(Driver) {
    static auto create(void* con, string sql) {...}
    static void destroy(void* stmt) {...}
    static void query(void* stmt) {toTypedPtr!Statement(stmt).query;}
}

table = StmtGenerate!Driver.vtable;

void* stmt;
void query() {vtable.query(stmt);}
```

POLY DRIVER: VARIADIC QUERY DISPATCH

```
struct Statement {
    void* stmt;
    StmtVtable* vtable;

    this(Connection* con, string sql) {...}

    void query() {
        vtable.query(stmt, args);
    }

    void query(A...) (A args) {
        vtable.query(stmt, args);
    }
}
```

prepared version

problem

CHALLENGE

Need to transport arguments from one templates query call to another at *run time*

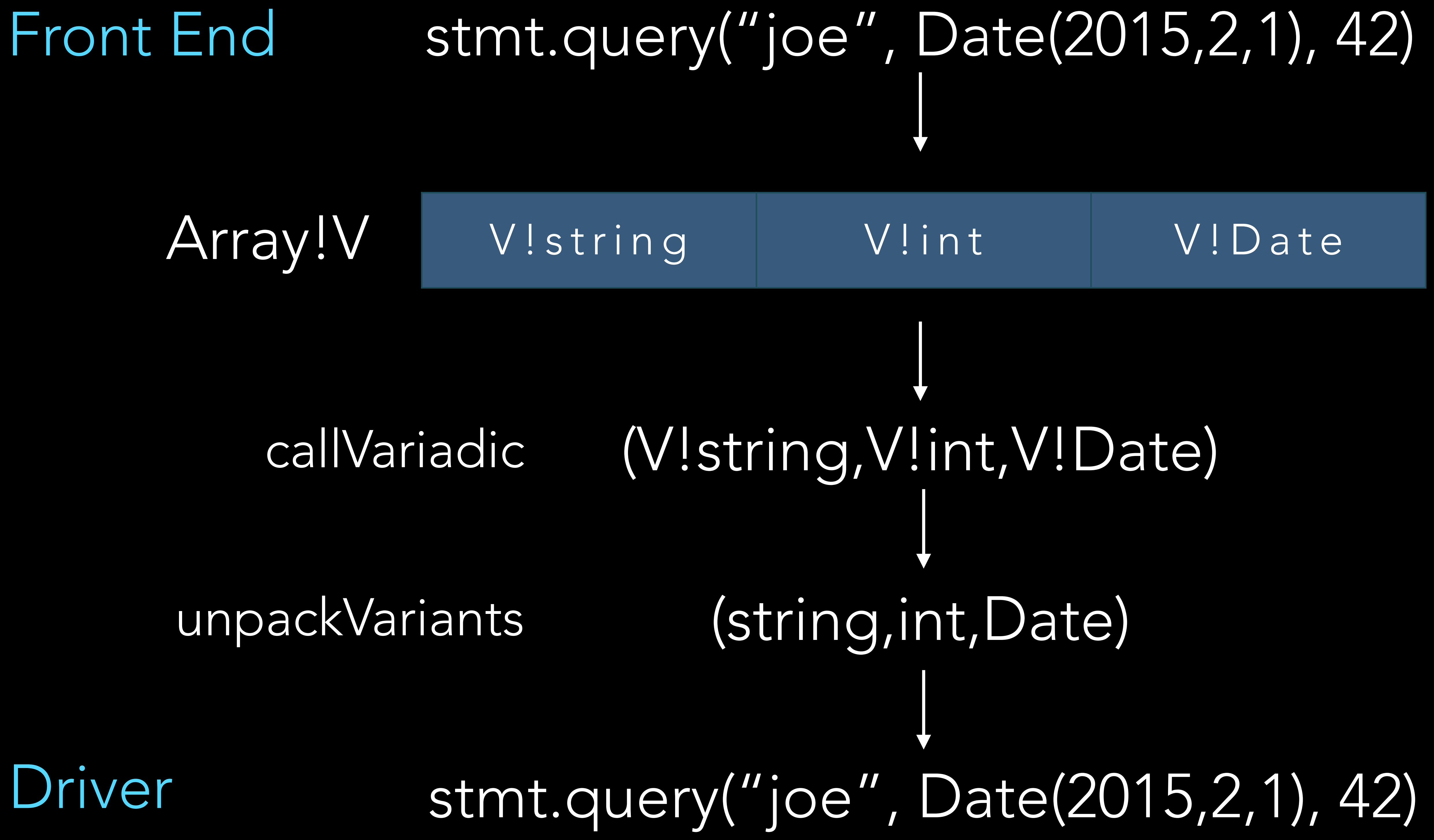
$$\text{Run time} \longrightarrow \text{Compile Time}$$

CHALLENGE #2

Avoid requiring the drivers to handle Variant arguments

APPROACH

alias V = Variant;



Front End stmt.query("joe", Date(2015,2,1), 42)

Array!V

V!string	V!int	V!Date

callVariadic (V!string,V!int,V!Date)

unpackVariants (string,int,Date)

Driver stmt.query("joe", Date(2015,2,1), 42)

PACK INTO VARIANT ARRAY

```
void query(A...) (A args) {
    auto a = Array!Variant(args);
    bindArgs.reserve(a.length);
    foreach(arg; args) a ~= Variant(arg);
    driver.stmtVtable.variadicQuery(stmt, a);
}
```

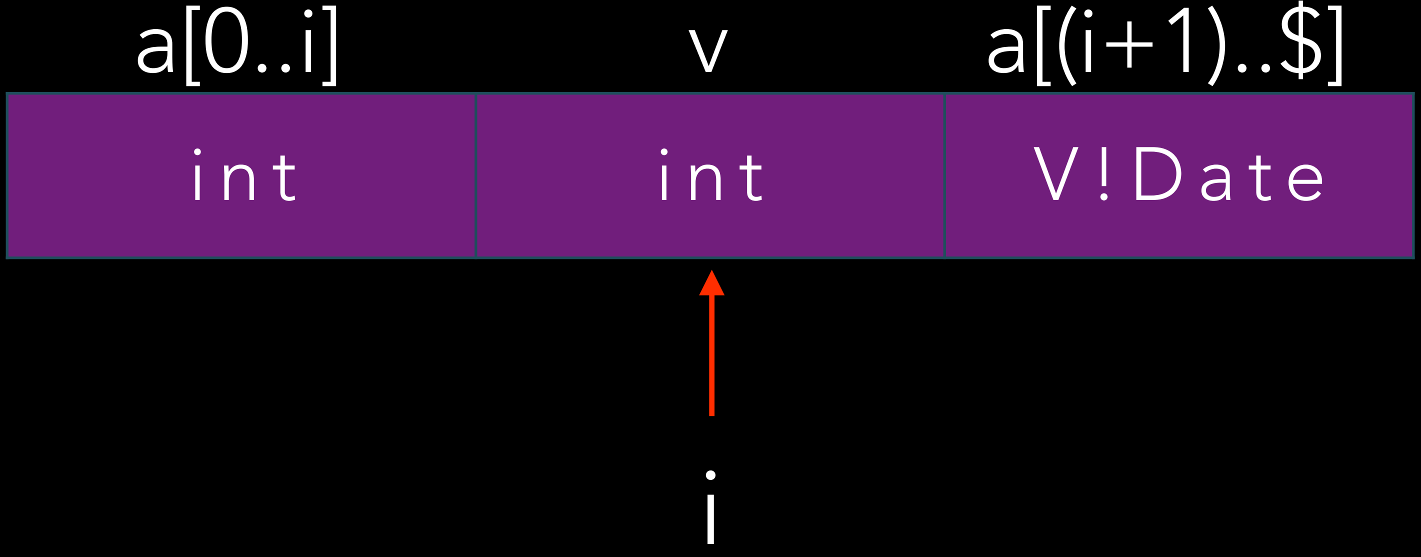
```
static void variadicQuery(void[] stmt, ref BindArgs a) {
    auto s = toTypedPtr!Statement(stmt);
    callVariadic!F(a, s);
}
```

ARRAY TO VARIADIC CALL

```
static void callVariadic(alias F,S,A...) (ref S s, A a) {
    switch (s.length) {
        case 0: break;
        case 1: F(a,s[0]); break;
        case 2: F(a,s[0],s[1]); break;
        case 3: F(a,s[0],s[1],s[2]); break;
        case 4: F(a,s[0],s[1],s[2],s[3]); break;
        case 5: F(a,s[0],s[1],s[2],s[3],s[4]); break;
        default: throw new Exception(“arg overload");
    }
}
```

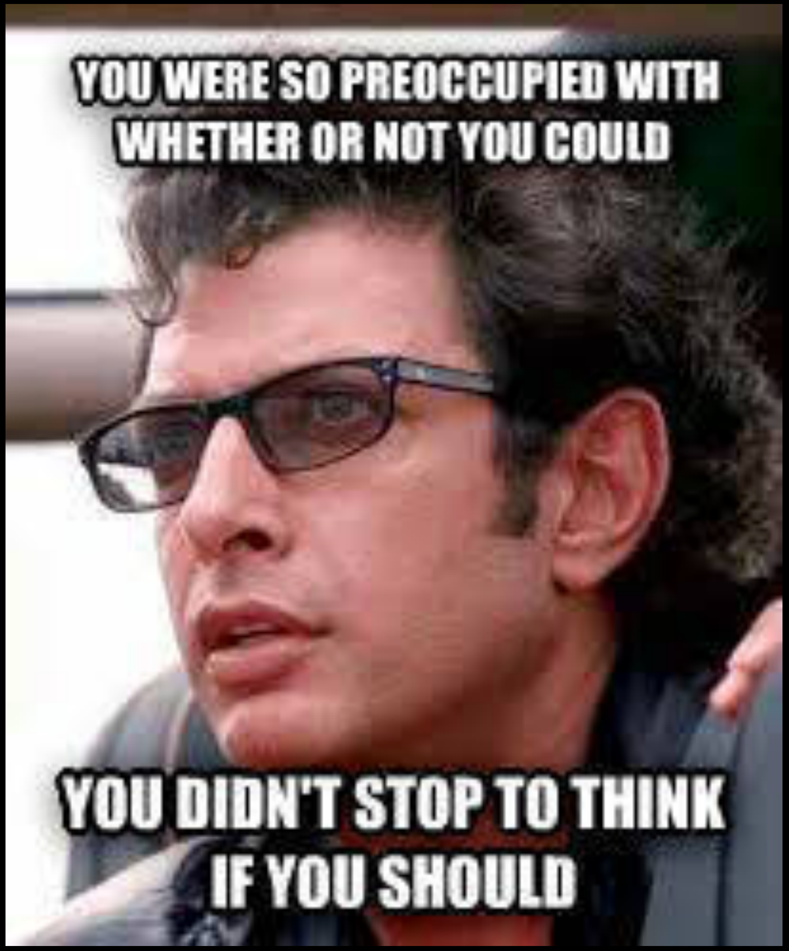
UNPACK VARIANTS

```
static void unpackVariants(alias F, int i=0, A...)(A a) {
    alias Types = AliasSeq!(byte, ubyte, string, char, dchar, int, uint, long, ulong, Date);

    static void call(int i, T, A...)(T v, A a) {
        unpackVariants!(F,i+1)(a[0..i], v, a[(i+1)..$]);
    }

    static if (i == a.length) {
        F(a);
    } else {
        foreach(T; Types) {
            if (a[i].convertsTo!T) {
                call!i(a[i].get!T,a);
                return;
            }
        }
        throw new Exception("unknown type: " ~ a[i].type.toString);
    }
}
```



THE FAIL IS STRONG
WITH THIS ONE
memegenerator.net

REALITY CHECK


YOU WERE SO PREOCCUPIED WITH WHETHER OR NOT YOU COULD
YOU DIDN'T STOP TO THINK IF YOU SHOULD

P(10,5) = 30240

Drivers *must* implement an additional call
two options:

```
query(ref Array!Variant args)
query(Args..)(A.. args)            // accept variants
```

Fiber based Driver Approaches

1) Modify socket calls in driver source, if available, to use a D (or vibe.d) non-blocking socket layer.
2) Implement from scratch a non-blocking driver using a known wire protocol
3) Adapt an existing non-blocking interface for use with vibe.d / std.event.

FIBER ASYNC QUERY EXAMPLE

vibe.d calls
async driver calls

```
auto descriptor = con.descriptor();

  auto event = createFileDescriptorEvent(
    descriptor, FileDescriptorEvent.Trigger.any);

  nonBlockingQuery(sql);

  while (true) {
     if (poll) {
        bool complete = readData;
        if (complete) break;
     }
     event.wait(FileDescriptorEvent.Trigger.read);
  }
}
```

NON-BLOCKING MYSQL CLIENT

WebScaleSQL

“We're Gonna Need A Bigger Database”

SUPPORTED DATABASES

current support (WIP)

SQLite ODBC PostgreSQL ORACLE MySQL FreeTDS *Making the leap to SQL Server*

Up next

WebScaleSQL Microsoft SQL Server

UPCOMING WORK

- Fiber based drivers for Postgres & Webscalesql
- Asynchronous push models (Observables)
- Query builder
- Schema metadata
- Callable statements
- Blob support
- Operation timing
- Simulated binding (freetds)
- Quote escaping
- Expose more features of underlying drivers
- Multiple result support
- NoSql support
- Test suite improvement
- Utilities
- More…

GETTING STARTED (OSX)

```
$ brew install dmd dub mysql
$ mkdir -p ~/src/demo && cd ~/src/demo
```

dub.json

```
{
    "name": "demo",
    "libs" : ["mysqlclient"],
    "dependencies": {"dstddb":"*"},
    "targetType": “executable”,
    "versions": [“StdLoggerDisableLogging"]
}
```

demo.d

```
import std.database.mysql;
import std.database.util;
void main() {
    auto db = createDatabase("mysql://127.0.0.1/test");
    db.query("select * from person”).rows.writeRows;
}
```

```
$ dub
```

QUESTIONS

https://github.com/cruisercoder/dstddb

DUB: dstddb