Supersonic Query Engine API

A short introduction

(onufry@google.com,
tkaftal@google.com)

Introduction

Supersonic is a query engine library for efficient columnar data processing.

Supersonic is all about ***speed.***

The library uses many low-level, cache-aware optimisations in order to achieve this goal.

It also has a robust and conveniet API - the aim of this document is to provide a quick overview of how Supersonic should be used.

Operations

The basic Supersonic class used to perform any operation is called (surprise) "Operation".

Examples of Operations include Compute, Filter, Sort, HashJoin and more.

Operations

The basic Supersonic class used to perform any operation is called (surprise) "Operation".

Examples of Operations include Compute, Filter, Sort, HashJoin and more.

In the typical case to create an operation we need:

- a child operation (or children), that supply data
- a projector (a description which of the output columns we should pass on)
- usually some more stuff (an Expression in Compute, an ordering description in Sort)

Operations - how to use them

Run an operation factory

Create a Cursor out of the operation

Pull results using Next

Operations - how to use them

Run an operation factory	(input: child (i.e. data source), projector, usually specification) *returns: Operation*
Create a Cursor out of the operation	(ties the columns of the child to the cursor, preparation for evaluation) *returns: a Failure or Cursor*
Pull results using Next	(pulls data from the cursor) *returns: a Failure or View*

Computation example

```
Expression* expression = (...);
Operation* data_source = (...);

Operation* computation =
    Compute(expression, data_source);

FailureOrOwned<Cursor> cursor =
    computation->CreateCursor();
// error handling.
PROPAGATE_ON_FAILURE(cursor);

ResultView out_data = cursor.get()->Next();
// (now consume the results somehow).
```

Filtering example, with column selection

```
Expression* predicate = (...);
Operation* data_source = (...);
Projector* projector = (...);

Operation* filter = Filter(
    predicate, projector, data_source);
// memory management.
filter->SetBufferAllocator(
    HeapBufferAllocator::Get(), false);
FailureOrOwned<Cursor> bound_filter =
    filter->CreateCursor();

ResultView data = bound_filter.get()->Next();
```

Where does the data come from?

Supersonic does not *currently* provide a built-in data storage format.

There is a strong intention of developing one.

As of now, data can be persisted in memory, using `Tables`.

Where does the data come from?

```
TupleSchema schema = (...);
Table table = Table(
    schema, HeapBufferAllocator::Get());
```

```
Operation* table_input = new ScanView(
     table->view());
```

Where does the data come from?

A more low-level approach:

```
TupleSchema schema = (...);
Table table = Table(
    schema, HeapBufferAllocator::Get());
// Now you can add rows to this table using
// AppendRow, the specifics are in
// supersonic/cursor/infrastructure/row.h
FailureOrOwned<Cursor> cursor =
    table.CreateCursor();
```

Failure recognition

Operation creation does not fail (in general).

Creating a Cursor can fail. Thus, a `FailureOr<Cursor>` is returned from CreateCursor. This is a lightweight template.

Success is checked for by `.is_failure()`, and the result retrieved through `.get()`. On failure we can analyze the problem by `.exception()`.

The typical "propagation" case is handled through the `PROPAGATE_ON_FAILURE(result);` macro.
(that's a macro because we want to handle stack traces).

There's also a FailureOrOwned<T>, the equivalent of scoped_ptr<T> (just as FailureOr<T> is the equivalent of T).

Expressions

Operation	Expression
CreateCursor	Bind
Cursor	BoundExpression
Next	Evaluate
ResultView	FailureOr<View>

Expression - where do I get the data?

In Operations, we assumed the child Cursor was the supplier of the data.

However, in Expressions usually we supply data ourselves, View by View (as an Expression is a part of a Cursor, and we "hold" the root of the Expression tree, and not the leaves).

Thus, while `Next()` had no arguments, `Evaluate(const View&)` takes an input View, which is then passed down, and appropriate expressions (for instance `NamedAttribute`) extract the appropriate View columns as their result Views.

Evaluating Expressions

```
Expression* column_a = NamedAttribute("a");
Expression* column_b = NamedAttribute("b");
Expression* plus = Plus(column_a, column_b);

TupleSchema input;
input.AddAttribute("a", INT32, NOT_NULLABLE);
input.AddAttribute("b", UINT32, NULLABLE);
FailureOrOwned<BoundExpression> bound_plus =
    plus->Bind(input,
               HeapBufferAllocator::Get(),
               1000);

FailureOrReference<const View> output =
    bound_plus->Evaluate(some_view);
```

Computation example with Expression

```
Expression* expression = Plus(
   NamedAttribute("a"), NamedAttribute("b"));
Operation* data_source = (...);

Operation* computation = Compute(expression,
                                 data_source);

computation->SetBufferAllocator(
    HeapBufferAllocator::Get(), false);

FailureOrOwned<Cursor> cursor =
    computation->CreateCursor();

ResultView out_data = cursor.get()->Next();
```

Object and data lifetime

Each supersonic Cursor/Expression/Operation/etc. owns its children, and takes care of freeing them when it is itself destroyed.

After calling Evaluate, the new View that is created overwrites the old one - an Expression has space for only one block of Evaluation results (this might change in the structured API).

Similarly, calling Next on a Cursor invalidates the old data.

Object and data lifetime

```
Expression* expression =
  Plus(AttributeAt(0), AttributeAt(1));
BoundExpression* bound = expression.Bind(...);
ResultView result = bound.Evaluate(input1);

const View* view_pointer = &result.get();

bound.Evaluate(input2);
```

Now view_pointer points to a View containing the results of the second Evaluation. The view encapsulates data that is physically stored in a block that is a member of the BoundExpression and gets overwritten with every Evaluate call.
The same goes for Cursors and calling Next().

How to copy data for reusal?

```
const View& result =
  bound.Evaluate(some_view).get();
TupleSchema schema = result.result_schema();
// Prepare new block, same schema as our View
Block* new_space = new Block(schema,
  HeapBufferAllocator::Get());
// Allocate same number of rows as our View.
new_space->Reallocate(result.row_count());
// Prepare a copier (the last true denotes a
// deep copy.
ViewCopier copier(schema, schema,
                  NO_SELECTOR, true);
// And copy the data.
copier.Copy(result.row_count(), result, NULL,
  0, new_space);
```

What next?

To see some fully-fledged working examples **download the source** and go to

`supersonic/test/guide`