# What is a query?

Currently queries only work on one single table. Joins or self joins are not supported. A query consists of two things:

* One or more **conditions** (equal, greater, less) belonging to each their **condition column**
* An **action** (find, count, find\_all), optionally belonging to an **aggregate column** that works as value source (sum, max)

The conditions can be of various kinds:

1. Requiring linear search
2. Lookup in indexed column
3. Lookup in string enum
4. ListView filter (search only in rows in a given ListView – for sub queries)
5. Searching in subtable of a column

The conditions can be related to eachother in multiple ways:

* AND
* OR
* Parantheses (begin\_group() and end\_group())

Examples of conditions are:

* greater
* less
* equal
* not\_equal
* greater\_equal
* less\_equal
* begins\_with (strings)
* ends\_with (strings)

We have various aggregate actions

* find first
* find all
* count
* sum
* max
* min
* average

And finally various column types:

* String (subtypes ArrayString, ArrayStringLong, ArrayBigBlobs)
* Integer
* Binary
* Date
* etc

# Scheduling

Assume we have 3 conditions of types b (indexed lookup), d (ListView filter) and a (linear search). Row 0 is left and the dots are matches:

b  
d  
a

The query engine “travels” from left to right in a single pass, inside a loop belonging to the condition it currently finds is fastest. It jumps frequently between the condition loops.

When a match is found in the current condition loop, the other conditions are probed at that match index, and the search resumes.

We must first note that a linear searches and indexed lookups can outperform eachother mutually, depending on match distance and bitwidth. Especially, an indexed search that matches row 1, 11, 12, 14, 25, 30, 54 is outperformed by a linear boolean search that can test row 0 - 63 in one operation.

So we introduce two statistics variables for each condition:

|  |
| --- |
| Linear search: m\_dT = (bitwidth == 0 ? 1.0 / MAX\_LIST\_SIZE : bitwidth / 8.0) Index, ListView and Enum: m\_dT = 0 |

**float m\_dD**:  
Average distance (in rowcount) between matches around current position  
  
**float m\_dT**:   
Time (in arbitrary units) it takes to test row n + 1 for a match if row n has just been tested. It depends on the condition kind:

We also define a cost function for each condition:

|  |
| --- |
| float cost(): 1.0/16.0 \* m\_dD + m\_dT |

The **cost** function is an expression of the **average time spent per table source row** when inside the given condition loop. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **Kind** | **Match distance** | **Bit width** | **cost** |
| Linear search | 64 | 1 | 0.1875 |
| Index/ListView/Enum | 64 | any | 0.1875 |
| Linear search | 20 | 4 | 1.3 |
| Linear search | infinity (no matches) | 0 | 0.001 |
| Index/ListView/Enum | 15 | any | 1.14 |
| Index/ListView/Enum | 100 | any | 0.16 |

# Call flow

Imagine a query like **first.greater(4.56).second.not\_equal(“hello”).third.equal(true)** on float, string and bool columns. This is constructed as condition objects (“nodes”) from various templated classes defined in query\_engine.hpp:

<NotEqual> StringNode  
m\_value = “hello”

m\_

<Equal> IntegerNode  
m\_value = true

m\_

<Greater, float> FloatDoubleNode  
m\_value = 4.56

m\_

The classes inherit from ParentNode which has common methods and variables. Most important are:

**Commion methods inherited from class ParentNode:**

std::vector<ParentNode\*>m\_children; // List of pointers to all other nodes so that they can call methods on eachother  
size\_t m\_condition\_column\_idx; // Column of search criteria  
double m\_dD;  
double m\_dT;  
T m\_value; // Search value

**// Main entry point of a query. Can be called on any of the nodes; yields same result. Schedules calls to   
// aggregate\_local (see below).  
// Return value is the result of the query, or Array pointer for FindAll.**  
template<Action TAction, class TResult, class TSourceColumn>  
TResult **aggregate**(QueryState<TResult>\* st, size\_t start, size\_t end, size\_t agg\_col, size\_t\* matchcount)

**// Executes start…end range of a query and will stay inside the condition loop of the node it was called on. Can be   
// called on any node; yields same result, but different performance. Returns prematurely if condition of   
// called node has evaluated to true local\_matches number of times.   
// Return value is the next row for resuming aggregating (next row that caller must call aggregate\_local on)**template<Action TAction, class TResult, class TSourceColumn>  
size\_t **aggregate\_local**(QueryState\* st, size\_t start, size\_t end, size\_t local\_limit, SequentialGetter\* source\_column)

**Pure virtual method (each node class must define their own):  
  
// Finds the first row (in start…end) that fulfills the condition of the node it was called on**

size\_t **find\_first\_local**(size\_t start, size\_t end)

There also exists a “match consuming” QueryState object. An example of call flow where row 7 is a match is given below. It is pseudo code where we only show start and end arguments for simplicity.

aggregate(0, 2000);

QueryState

bool match(size\_t index, int64\_t value)

<Greater, float> FloatDoubleNode  
m\_value = 4.56

m\_

match(7, aggregate\_source\_column.get(7))

aggregate\_local(0, 100)

<NotEqual> StringNode  
m\_value = “hello”

m\_

find\_first\_local(0, 100) = 7

find\_first\_local(7, 8) = 7

find\_first\_local(7, 8) = 7

<Equal> IntegerNode  
m\_value = true

m\_

Pseudo code:  
  
class ParentNode {

**aggregate**(start, end) {  
 while(start < end) {  
 best\_node = analyze which is the best node  
 start = best\_node->**aggregate\_local**(subrange\_start, subrange\_end);  
 }  
  
 // This is often overloaded for performance, e.g. for IntegerNode  
 **aggregate\_local**(start, end) {  
 for(;;) {  
 start = find\_first\_local(start, end);  
 if(start == not\_found)  
 return end;  
   
 For each other node, test if other\_node->**find\_first\_local**(start, start + 1)   
 is a match, and if they are, then call:  
 match(start, aggregate\_column->get(start));  
 }  
 }  
}

We have a node class for each column type:

class **ParentNode**   
template <class TConditionValue, class TConditionFunction> class **IntegerNode**: public ParentNode  
class **ListviewNode**: public ParentNode   
class **SubtableNode**: public ParentNode   
template <class TConditionFunction> class **StringNode**: public ParentNode   
template<> class **StringNode<Equal>**: public ParentNode   
template <class TConditionValue, class TConditionFunction> class **FloatDoubleNode**: public ParentNode   
template <class TConditionFunction> class **BinaryNode**: public ParentNode   
class **OrNode**: public ParentNode   
template <class TConditionValue, class TConditionFunction> class **TwoColumnsNode**: public ParentNode   
class **ExpressionNode**: public ParentNode

# New datatype?

Super simple! Make a class that provides at least following methods:

NewTypeNode(size\_t column) {  
}

void init(const Table& table) {  
}

size\_t find\_first\_local(size\_t start, size\_t end) {   
}  
  
For performance, also overload aggregate\_local(), but this is optional.

# IntegerNode

For conditions on integer columns (int, date, bool, etc) we utilize SSE to search in 128 bits (16 bytes) at a time, for both equal, notequal, greater, less, etc.

Assume we have an Array object of byte sized integers and perform a find\_all(). The stars are matches. We could first call Array::find(0, -1):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** |  |  |  |  | **5** |  |  | **7** |  |  |  |  | **12** |  | **15** |  |  |  |  |  |  |  | **23** |  | **25** |  |  |  |  |  | **31** |  |  |  |  |  |  |  | **39** |  |  |  |  |
|  |  |  |  |  | **\*** |  |  |  |  |  |  |  | **\*** |  |  |  |  |  |  |  |  |  |  |  | **\*** |  |  |  |  |  |  |  |  |  |  |  |  |  | **\*** |  |  |  |  |

SSE will first compare index 0…15 and return 5.

To find the next match, we could call Array::find(6, -1):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** |  |  |  |  | **5** |  |  | **7** |  |  |  |  | **12** |  | **15** |  |  |  |  |  |  |  | **23** |  | **25** |  |  |  |  |  | **31** |  |  |  |  |  |  |  | **39** |  |  |  |  |
|  |  |  |  |  | **\*** |  |  |  |  |  |  |  | **\*** |  |  |  |  |  |  |  |  |  |  |  | **\*** |  |  |  |  |  |  |  |  |  |  |  |  |  | **\*** |  |  |  |  |

This requires a manual search loop for the unaligned part, because SSE can only read at aligned data (SSE 4.2 introduced a nonaligned load instruction, but its performance has not yet been tested).

This is slow and also performs redundant work, especially when match density is high.

To solve that, we have extended Array::find(); it can call match() directly for each match, provided that 1) the query has no other conditions, and 2) aggregate column (if any) is the same as condition column.

Array::find() can also call a callback method if there are other conditions that need to be tested.

So inside the IntegerNode class, we have overloaded aggregate\_local to following:

**pseudo code**  
class IntegerNode {

template <Action TAction, class TSourceColumn> bool **match\_callback**(int64\_t v) {  
 // v is row index  
 Test remaining conditions in m\_children with other\_node->find\_first\_local(v, v + 1)  
 and call **match**(v, aggregate\_column.get(v)) if all conditions are true for row v  
}

// Overloading of general aggregate\_local described in previous chapter  
size\_t **aggregate\_local**(start, end) {  
 if(there are no other conditions than us, and aggregate column is the same as ours) {

m\_array.**find**<TAction>(start, end, NULL, &query\_state); // query\_state has the match() method

}  
else {

m\_array.**find**<act\_CallbackIdx>(start, end, &match\_callback, &query\_state); //query\_state contains match()

}

}

# Query State

The QueryState is defined in array.hpp

It consumes matches and performs aggregate actions on them. Different actions use either index, value or nothing:

|  |  |  |
| --- | --- | --- |
| **Action** | **Uses index** | **Uses value** |
| find | + |  |
| find\_all | + |  |
| count |  |  |
| average |  | + |
| max |  | + |
| min |  | + |
| sum |  | + |

It has the method:

template <Action action, bool pattern> inline bool match(size\_t index, uint64\_t indexpattern, int64\_t value)

If value or index are unused by the action, the caller will most often just pass 0 for that argument (compilers are bad at detecting that the return value of get() is unused and that it has no side effects and that it can thus be elided).

Both SSE and some “integer bithacks” we have developed are very fast at generating a bit-pattern of matching rows. You can see an example of that in Array::FindGTLT\_Fast() in array.hpp.

These bitpatterns can be utilized to provide a whole chunk of matches to the QueryState at once in a single call. Set the ‘bool pattern’ template argument to true and provide it as ‘indexpattern ‘ argument.

The match() can decide wether or not the indexpattern is sufficient information for the given action and return ‘false’ if it rejects it (e.g. for the max/min actions). If match returns false, you must call manually one time per match with normal index and value arguments:

uint64\_t matchpattern = BitHacks(data\_chunk);  
bool accepted = match<Taction, true>(0, matchpattern, 0);

if(!accepted) {  
 for(int I = 0; I < 64; I++)  
 …  
 extract index and values for each match  
 match<Taction, false>(index, 0, value);  
}

All this is reduced at compile time.