|  |  |
| --- | --- |
| 产品名称Product name | 密级Confidentiality level |
| Flint | 秘密 secret |
| 产品版本Product version | Total ?? pages 共??页 |
| V001R001 |

Flint: distributed execution framework

|  |  |  |  |
| --- | --- | --- | --- |
| Prepared by  拟制 | ??? 项目组  Parallel Exection Group | Date  日期 | 2015-10-26 |
| Reviewed by  评审人 |  | Date  日期 | 2015-11-20 |
| Approved by  批准 |  | Date  日期 | 2015-1-20 |

Revision Record

| Date  日期 | Revision Version  修订 版本 | CR ID / Defect ID CR号 | Sec No.  修改 章节 | Change Description  修改描述 | Author  作者 |
| --- | --- | --- | --- | --- | --- |
| 2015/10/26 | V1.0 |  | ALL | Write | Knizhnik.konstantin |

Catalog

[Flint: distributed execution framework 1](#_Toc435395790)

[Revision Record 2](#_Toc435395791)

[1 Introduction 6](#_Toc435395792)

[1.1 Purpose 6](#_Toc435395793)

[1.1.1 Name 6](#_Toc435395794)

[1.1.2 Operators 6](#_Toc435395795)

[1.1.3 Applications 6](#_Toc435395796)

[1.1.4 Requirement description 6](#_Toc435395797)

[1) IT production Line requirement 6](#_Toc435395798)

[2) System requirement 6](#_Toc435395799)

[2 Flint architecture 8](#_Toc435395800)

[3 Flint Design Description 11](#_Toc435395801)

[3.1 Flint API 11](#_Toc435395802)

[3.2 Flint class hierarchy 11](#_Toc435395803)

[3.3 Fint RDD methods 12](#_Toc435395804)

[4 Flint configurations 15](#_Toc435395805)

[4.1 Standalone or Spark extension 15](#_Toc435395806)

[4.2 Data formats recognized by Flint 16](#_Toc435395807)

[4.3 Flint integration with Spark 17](#_Toc435395808)

[5 Flint performance evoluation 20](#_Toc435395809)

Flint: distributed execution framework

High Level Design Document

Keywords： RDD, distributed, HDFS, Hadoop, Spark, SQL

Abstract :  This document describes the high level design of the Flint which is considered as faster elternative to Hadoop/Spark. Flint allows to write queries in C++, completely eliminating JVM interpretation/JIT cost, garbage collection and serialization overhead. Flint is optimizaed for processing data in memory, avoiding materialization of intermediate results to the disk.

Flint’s program is written in C++ and the same code is executed at each node. There is no special coordinator node in Flint. Instead of it first node is playing role of coordinator of queries. It performs merge of partial results produced by other nodes and delivers final result to the user.

Communication between nodes is performed using Unix domain/TPC sockets using propertary protocol. Each node has one thread receiving data from network and multiple threads sending data to other nodes (one sender thread for each node). The unit of transfer is large enough block (64kb by default). Messages are packed in the block and placed in queues. Queue is high level mechanism of exchanging data between nodes. Producer fills the block and places it in destination queue (for example in sender’s queue). Sender retrives blockfrom queue and send it through netwrork to the resipient. Recevier thread reads data from network and place in distination queue. Queues are identified by ID, which is assign at the moment of queue creation. As far as all nodes are executing the same code, each node is assigned ther same set of queue’s IDs.

Flints implements classical map-reduce operators (filter, project, map-reduce, reduce, aggregate,…) as well as specialized operators used by SQL (join, semijoin, sort, …). Pipelined operators (like filter and project) are executed inside pipeline locally at each node. Other operators require exchanging of data between nodes. Then can shuffle, broadcast, scattern and gather their data

List of abbreviations. Describe abbreviations in this document, full spelling of the abbreviation and Chinese explanation should be provided.

| Abbreviations | Full spelling | Explanation |
| --- | --- | --- |
| RDD | Resilient Distributed Dataset | The basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.  . |
| SQL | SQL | Non-procedural query language. De-facto standard for modern database systems. |
| RDBMS | Relational Database Management System | System responsible for storing and processing of relations (tables). |
| HDFS | Hadoop distributed file system | Distributed files system used in Apache Hadoop. |

# Introduction

## Purpose

This document describes the high level design of the Flint distributed execution framework, which can be used standalone or as part of Spark, accelerating query execution.

### Name

Flint : Distributed execution framework

### Operators

Flint implements operators needed for execution of SQL queries and performing other distributed calculating. Also it provides framework for wrioting own operators and transports.

### Applications

Flint is intended to be used for execution of analytic queries for large volumes of data (big data). It can be considered either as competitor of Hadoop/SparkSQL, either extension/accelator of SparkSQL.

### Requirement description

#### IT production Line requirement

#### System requirement

* Must support

|  |  |  |  |
| --- | --- | --- | --- |
| **Service Requirment** | | | |
| **Requirement name** | **Challenge, problem and requirement description** | **Requirement category** | **Priority** |
| Support shared nothing configuration: data is stored locally at each node | Flint should allow to access files in local files system in internal raw format. Also it should provide utility for converting data fro CSV format to the internal repreosentation. | 01.function requirement | High |
| Support work with distributed file system | Flint must support work with HDFS | 01.function requirement | High |
| Support parquet file format | Flint should be able to work with parquet files stored either in local file system, either in DFS | 01.function requirement | High |
| Support integration with SparkSQL | It should be possible to use Flint as SparkSQL RDD, making it possible to move to Flint part of query execution plan. | 01.function requirement | High |
| Eliminate disk IO overhead | Flint should be able to process quieries without saving intermediate results to some storage device. | 02.performance requiement | High |
| Support work with dataset larger than size of memory | Decpit to the fact, then in-memory option is considered as primary Flint’s use case, it should be able to work with data sets not fitting in memory by shuffling data to the files | 02.performance requiement | Low |

* Optional support

Support high availablability: add/remove node without stopping the system

* Ignore support

Fault tolerance inside a query. It is normal to abort current query if failure happens during query execution.

* Administration utilities

Flint is not a production system so it caurrently doesn’t provide any monitoring or administration utilities..

* Technique Specification Summary
* Function：Support operators needed for distributed execution of SQL queries
* Performance: Should provide 3x better performance than SparkSQL.
* Integration: Should be available as SparkSQL RDD.
* Scalability: The performance increases with the amount of nodes.

# Flint architecture

illustrates processing of typeical SQL query. Flint builds pipeline of SQL operators which are executed concurrently at all nodes. Pipelined operators (like Project, Join) are executed independently at al nodes. Most of operations are inlines (thanks to using templates in Flint’s API). Result of each operator (row) is passed to the input of other (outer) operator. Right now Flint implements PUSH style processing (as in Volcano-style execution engine).

Fig. 1 SQL query processing pipeline

More complex SQL operators like join, group-by, aggregate, sort,… requires interaction between different nodes. Flints supports three main communication patterns:

1. Scatter-gather:

Data source is partitioned by some key and sent to the target nodes. And data nodes collect data from different nodes and process them. Scatter-gather is used to implement hash join when both joined tables are large enought. Another example of using scatter-gather is classical map-reduce which is used in SQL queries to perform aggregation with grouping.

1. Broadcast:

Data source is replicated to all nodes. Replication can be used to peform hash join when one joined table is much smaller than other.

1. Merge:

Each node performs local processing of the data and then sends partial results to coordinator where them are combined into final result. Merge is used to sort result set and to calculate grand aggregates.

shows how how scattering of data is implemented in Flint for different operators. Join needs to shuffle both inner and outer tables between all nodes. And for calculation of aggregates we need to produce partial results locally at each onde and then send them to the corrdinator where them are merged to produce final result.

Fig. 2 illustrates how communication between nodes is organized in Flint.There is separate scatter job which fetches input data source, extracts parititiong key, calculates hash code, divides it by modulo of number of nodes and and places record in the correspondent send queue. Each send query is connected with send job. So number of send queues and send threads is equals to number of nodes in the system minus one (no need to send data to self node).

Fig 2 Flint dataflow between nodes

And eachnode has single receive thread which collects data from all nodes. Recevied message includes ID of queue where data should be placed. As far as all nodes are executing the same query execution plan, sequences of obtained queue IDs are the same at all nodes. So we can specify target queue ID without any additional communication with this node.

Recevier thread enqueues received block in speficied queue connected with gather job. This job implements reduce logic – it combines responces received fro multiple node and produce single output stream of data.

In Flint data flow is completely controlled by RDD implementation, which can choose one or another plan depending on input data sources. For example join can be performed either by shuffling inner and outer tables, either be replication of inner table if it is small enough (see Fig 3).

Node 0

|  |
| --- |
| 5 |
| 1 |
| 7 |

|  |
| --- |
| 3 |
| 6 |
| 2 |

|  |
| --- |
| 4 |
| 1 |
| 9 |

Node 1

|  |
| --- |
| 8 |
| 5 |
| 2 |

|  |
| --- |
| 7 |
| 5 |
| 8 |

Node 2

|  |
| --- |
| 3 |
| 9 |
| 6 |

Fig 3a Shuffle join

Node 0

|  |
| --- |
| 5 |
| 1 |
| 7 |

|  |
| --- |
| 3 |
| 6 |
| 2 |

|  |
| --- |
| 4 |
| 1 |
| 9 |

Node 1

|  |
| --- |
| 8 |
| 5 |
| 2 |

|  |
| --- |
| 7 |
| 5 |
| 8 |

Node 2

|  |
| --- |
| 3 |
| 9 |
| 6 |

Fig 3b Broadcast join

size >inmemJoinThreshold

size < broadcastjpinJoinThreashold

Shuffle to files

In memory shuffle join

Broadcast join

Fig 3c Join kind selection algorithm

# Flint Design Description

## Flint API

The main idea of Flint API is to provide API similar with Spark API but using C++ language. C++ templates are widely used fir efficient inlining of functions. Unlike Spark API, Flint API doesn’t provide abstraction of data partitioning. The same code is executed at all nodes and whenever exchange of data between nodes is needed, it is up to concerete operator’s implementation. For example, JOIN operator can be implemented using broadcast of one joined table or using shuffle join.

So there is no RDD.compute method inFlint RDD which returns iterator though records belonging to the partition. Instead of it Flint RDD class serves like iterator, providing single **bool next(Record&)** method, which is used for traversing RDD records. Just one method is used for moving current position in RDD forward and retirieving next record. It is done to minimize virtual function call overhead.

Flint RDD as well as Spark RDD is using *pull* model of data flow and lazy evaluation. Inocation of most of RDD methods doesn’t actually perform any calculations. Instead of it exexution pipe is build. Execution is started when you call **RDD::execute** method or start iteration yourself.

Using polymorphic (virtual **next** method) greatly simplifies Flint design and provides highest flexibility, allowing to easily implement custom RDDs. But it prevents from complete inlining of query execution code and adds extra overhead of virtual functions call. Also some researches shows that *push* model can provide better performance under certain assumptions.

## Flint class hierarchy

Fig 4. shows main Flint class hiearchy. There is abstract RDD class which has several implementations corresponding to different operations with RDD. Right now there RDD operations are mostly priented on SQL queries execution. So there are RDD representing such relational operations as join, filter, project, group-by, aggregate. But it is possible to define own RDDs, implementing arbitrary data processing schemas. For example matrix multiplication nd other linear algebra operations.

Unline Spark, which comtrols tasks execution and data distribution, in Flint it is reposnibiluily of RDD implementation to organize data exchange and processing. Arbitrary distributes algorithms can be implemented by RDD. For example matrix multiplication can perform N iterations where each node multiples rows and columns locally available at this node and then sends its part of one of the matrixes to neighbour node. Or it can use broadcast method which is more efficient if broadcast is internally supported by underlying network layer.

Fig 4. Flint class hierarchy

## Fint RDD methods

template<class T >

template<bool(\*)(T const &) predicate>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< T > \*** [**RDD**](classRDD.html)**< T >::**[**filter**](classRDD.html#ae9eb98d74fdbe7a52af2434f69abbff3) | **(** |  | **)** |  |

Filter input [**RDD**](classRDD.html)

**Returns:**

[**RDD**](classRDD.html) with records matching predicate

template<class T >

template<class I , class K , void(\*)(K &key, T const &outer) outerKey, void(\*)(K &key, I const &inner) innerKey>

|  |  |  |  |
| --- | --- | --- | --- |
| [**RDD**](classRDD.html)**<** [**Join**](structJoin.html)**< T, I > > \*** [**RDD**](classRDD.html)**< T >::**[**join**](classRDD.html#ae418938416c501dc29e652661ddee845) | **(** | [**RDD**](classRDD.html)**< I > \*** | **with,** |
|  |  | **size\_t** | **estimation,** |
|  |  | **JoinKind** | **kind = InnerJoin** |
|  | **)** |  |  |

Left join two RDDs. Inner join returns pairs of matches records in outer and inner table. Outer join also returns records from outer table for which there are matching in inner table.

**Parameters:**

|  |  |
| --- | --- |
| **with** | inner join table |
| **estimation** | estimation for number of joined records |
| **kind** | join kind (inner/outer join) |
|  |  |

template<class T >

template<class K , class V , void(\*)(Pair< K, V > &out, T const &in) map\_f, void(\*)(V &dst, V const &src) reduce\_f>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**<** [**Pair**](structPair.html)**< K, V > > \*** [**RDD**](classRDD.html)**< T >::**[**mapReduce**](classRDD.html#a3d2043580527b7ad328fb308dc2a3e10) | **(** | **size\_t** | **estimation** | **)** | **[inline]** |

Perfrom map-reduce

**Parameters:**

|  |  |
| --- | --- |
| **estimation** | esimation for number of pairs |

**Returns:**

[**RDD**](classRDD.html) with <key,value> pairs

template<class T>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **virtual bool** [**RDD**](classRDD.html)**< T >::**[**next**](classRDD.html#a65f297602cd6d892c3337c911e5159b4) | **(** | **T &** | **record** | **)** | **[pure virtual]** |

Main [**RDD**](classRDD.html) method for iterating thoough records

**Parameters:**

|  |  |
| --- | --- |
| **record** | [out] placeholder for the next record |

**Returns:**

true if there is next record, false otherwise

template<class T >

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **void** [**RDD**](classRDD.html)**< T >::**[**output**](classRDD.html#af22b90c74197a95da0df3cef9aed8c74) | **(** | **FILE \*** | **out** | **)** |  |

Print [**RDD**](classRDD.html) records to the stream

**Parameters:**

|  |  |
| --- | --- |
| **out** | output stream |

template<class T >

template<class P , void(\*)(P &out, T const &in) projection>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< P > \*** [**RDD**](classRDD.html)**< T >::**[**project**](classRDD.html#ad2782c2622ec73e48213db8c580c84f8) | **(** |  | **)** | **[inline]** |

Map records of input [**RDD**](classRDD.html)

**Returns:**

projection of the input [**RDD**](classRDD.html).

template<class T >

template<class S , void(\*)(S &state, T const &in) accumulate, void(\*)(S &state, S const &partial) combine>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< S > \*** [**RDD**](classRDD.html)**< T >::**[**reduce**](classRDD.html#ad25b878bec772659ec98455b2e2bb159) | **(** | **S const &** | **initState** | **)** | **[inline]** |

Perform aggregation of input [**RDD**](classRDD.html)

**Parameters:**

|  |  |
| --- | --- |
| **initState** | initial aggregate value |

**Returns:**

[**RDD**](classRDD.html) with aggregated value

template<class T >

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< T > \*** [**RDD**](classRDD.html)**< T >::**[**replicate**](classRDD.html#aa71deda44b16ca8f2c8716faf37f8a86) | **(** |  | **)** | **[inline, virtual]** |

Replicate data between all nodes. Broadcast local [**RDD**](classRDD.html) data to all nodes and gather data from all nodes. As a result all nodes get the same replicas of input data

**Returns:**

replicated [**RDD**](classRDD.html), combining data from all nodes

template<class T>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **T** [**RDD**](classRDD.html)**< T >::**[**result**](classRDD.html#ab32aecb3128d721bb14a79fadedc4bfb) | **(** | **T const &** | **defaultValue** | **)** | **[inline]** |

Get single record from input [**RDD**](classRDD.html) or substitute it with default value of [**RDD**](classRDD.html) is empty. This method is usful for obtaining aggregation result

**Returns:**

Single record from input [**RDD**](classRDD.html) or substitute it with default value of [**RDD**](classRDD.html) is empty.

template<class T >

template<class I , class K , void(\*)(K &key, T const &outer) outerKey, void(\*)(K &key, I const &inner) innerKey>

|  |  |  |  |
| --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< T > \*** [**RDD**](classRDD.html)**< T >::**[**semijoin**](classRDD.html#a16bdade91a5b31c986aec51f576f47ed) | **(** | [**RDD**](classRDD.html)**< I > \*** | **with,** |
|  |  | **size\_t** | **estimation,** |
|  |  | **JoinKind** | **kind = InnerJoin** |
|  | **)** |  |  |

Left simijoin two RDDs. Semijoin find matched records in both tables but returns only records from outer table. Antijoin returns only this records of outer table for which there are no matching in inner table.

**Parameters:**

|  |  |
| --- | --- |
| **with** | inner join table |
| **estimation** | estimation for number of joined records |
| **kind** | join kind (inner/anti join) |

template<class T >

template<int(\*)(T const \*a, T const \*b) compare>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< T > \*** [**RDD**](classRDD.html)**< T >::**[**sort**](classRDD.html#a5585484e67e890d38df8d87a44410bf1) | **(** | **size\_t** | **estimation** | **)** | **[inline]** |

Sort input [**RDD**](classRDD.html)

**Parameters:**

|  |  |
| --- | --- |
| **estimation** | estimation for number of records in [**RDD**](classRDD.html) |

**Returns:**

[**RDD**](classRDD.html) with sorted records

template<class T >

template<int(\*)(T const \*a, T const \*b) compare>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [**RDD**](classRDD.html)**< T > \*** [**RDD**](classRDD.html)**< T >::**[**top**](classRDD.html#a1322d34d2e794c39a8208af37b6571a1) | **(** | **size\_t** | **n** | **)** | **[inline]** |

Find top N records according to provided comparison function

**Parameters:**

|  |  |
| --- | --- |
| **n** | number of returned top records |

**Returns:**

[**RDD**](classRDD.html) with up to N top records

# Flint configurations

## Standalone or Spark extension

Flint can be configured in different ways: its can work standalone, can be integrated with Spark, can work with local files and with distributed file system. Fig. 5 shows this comfigurations.

Node 2

Node 1

Node 0

Local file system

Local file system

Local file system

Fig. 5a. Standalone Flint shared nothing

Node 0

Node 2

Node 1

Distributed file system

Fig. 5b. Standalone Flint with distributed file system

Spark

Worker 1

Worker 0

Worker 2

Worker 1

Worker 0

Worker 2

Worker 1

Worker 0

Flint

Node 1

Node 0

Node 2

Worker 1

Worker 0

Worker 2

Spark

Fig. 5c. Flint with Spark integration

## Data formats recognized by Flint

Flint has it own raw binary format of storing data in the disk. C++ struct (POD – plain old data) are stored in file as it is. So format of data on disk and in memory is the same. No any unpacking is needed to read data from the disk. It is yet another advantage of Flint comparing with Spark where substantial amount of time is spent in the serialization/deserialization code.

But this internal format is not the only one Flint recognizes. It also can work with parquet files: of one the most effiecient data formats for Spark. It can access parquet file sor local file system or from HDFS (using libhdfs which is actually wrapper around Java code).

Fig. 6 illustrates possible Flint used cases:

Fig. 6 Flint input data formats

## Flint integration with Spark

The main idea of integrating Flint in Spark to use speed Flint as accelerator of SQL queries. We try use all capabilities of Spark SQL parser and Catalyst query optimizer, but substitute query executor with Flint (Fig 7):

Fig. 7. Flint integration ni Spark

Consider the following SQL query:

select

l\_returnflag,

l\_linestatus,

sum(l\_quantity) as sum\_qty,

sum(l\_extendedprice) as sum\_base\_price,

sum(l\_extendedprice\*(1-l\_discount)) as sum\_disc\_price,

sum(l\_extendedprice\*(1-l\_discount)\*(1+l\_tax)) as sum\_charge,

avg(l\_quantity) as avg\_qty,

avg(l\_extendedprice) as avg\_price,

avg(l\_discount) as avg\_disc,

count(\*) as count\_order

from

lineitem

where

l\_shipdate <= 19981201

group by

l\_returnflag,l\_linestatus

order by

l\_returnflag,l\_linestatus;

It is translated to the following C++ code:

namespace Q1

{

struct Projection

{

double sum\_qty;

double sum\_base\_price;

double sum\_disc\_price;

double sum\_charge;

double avg\_qty;

double avg\_price;

double avg\_disc;

size\_t count\_order;

char l\_returnflag;

char l\_linestatus;

friend void print(Projection const& p, FILE\* out) {

fprintf(out, "%c, %c, %f, %f, %f, %f, %f, %f, %f, %lu",

p.l\_returnflag, p.l\_linestatus, p.sum\_qty, p.sum\_base\_price, p.sum\_disc\_price, p.sum\_charge, p.avg\_qty, p.avg\_price, p.avg\_disc, p.count\_order);

}

};

struct GroupBy

{

char l\_returnflag;

char l\_linestatus;

bool operator == (GroupBy const& other) const

{

return l\_returnflag == other.l\_returnflag && l\_linestatus == other.l\_linestatus;

}

friend size\_t hashCode(GroupBy const& gby) {

return (gby.l\_returnflag << 8) ^ gby.l\_linestatus;

}

};

struct Aggregate

{

double sum\_qty;

double sum\_base\_price;

double sum\_disc\_price;

double sum\_charge;

double sum\_disc;

size\_t count\_order;

};

bool predicate(Lineitem const& lineitem)

{

return lineitem.l\_shipdate <= 19981201;

}

void map(Pair<GroupBy,Aggregate>& pair, Lineitem const& lineitem)

{

pair.key.l\_returnflag = lineitem.l\_returnflag;

pair.key.l\_linestatus = lineitem.l\_linestatus;

pair.value.sum\_qty = lineitem.l\_quantity;

pair.value.sum\_base\_price = lineitem.l\_extendedprice;

pair.value.sum\_disc\_price = lineitem.l\_extendedprice\*(1-lineitem.l\_discount);

pair.value.sum\_charge = lineitem.l\_extendedprice\*(1-lineitem.l\_discount)\*(1+lineitem.l\_tax);

pair.value.sum\_disc = lineitem.l\_discount;

pair.value.count\_order = 1;

}

void reduce(Aggregate& dst, Aggregate const& src)

{

dst.sum\_qty += src.sum\_qty;

dst.sum\_base\_price += src.sum\_base\_price;

dst.sum\_disc\_price += src.sum\_disc\_price;

dst.sum\_charge += src.sum\_charge;

dst.sum\_disc += src.sum\_disc;

dst.count\_order += src.count\_order;

}

void projection(Projection& out, Pair<GroupBy,Aggregate> const& in)

{

out.l\_returnflag = in.key.l\_returnflag;

out.l\_linestatus = in.key.l\_linestatus;

out.sum\_qty = in.value.sum\_qty;

out.sum\_base\_price = in.value.sum\_base\_price;

out.sum\_disc\_price = in.value.sum\_disc\_price;

out.sum\_charge = in.value.sum\_charge;

out.avg\_qty = in.value.sum\_qty / in.value.count\_order;

out.avg\_price = in.value.sum\_base\_price / in.value.count\_order;

out.avg\_disc = in.value.sum\_disc / in.value.count\_order;

out.count\_order = in.value.count\_order;

}

int compare(Projection const\* a, Projection const\* b)

{

int diff = a->l\_returnflag - b->l\_returnflag;

return diff != 0 ? diff : a->l\_linestatus - b->l\_linestatus;

}

RDD<Projection>\* query(SparkRDD<Projection>\* input)

{

return

input->

filter<predicate>()->

mapReduce<GroupBy,Aggregate,map,reduce>(10000)->

project<Projection, projection>()->

sort<compare>(100);

}

class Iterator : KrapsIterator

{

Projection2 row;

RDD<Projection2>\* result;

public:

virtual void\* next(){

return result->next(row) ? &row : NULL;

}

virtual ~Iterator() {

delete result;

}

Iterator(JNIEnv\* env) {

result = query(new SparkRDD<Projection1>(env, 0));

}

};

}

// Function to be obtained with dlsym

extern "C" {

void\* getQ1Iterator(JNIEnv\* env) {

return new Q1::Iterator(env);

}

# Flint performance evoluation