网址

- https://nebula-graph.com.cn/
- https://nebula-graph.com.cn/posts/

命令

- cmake

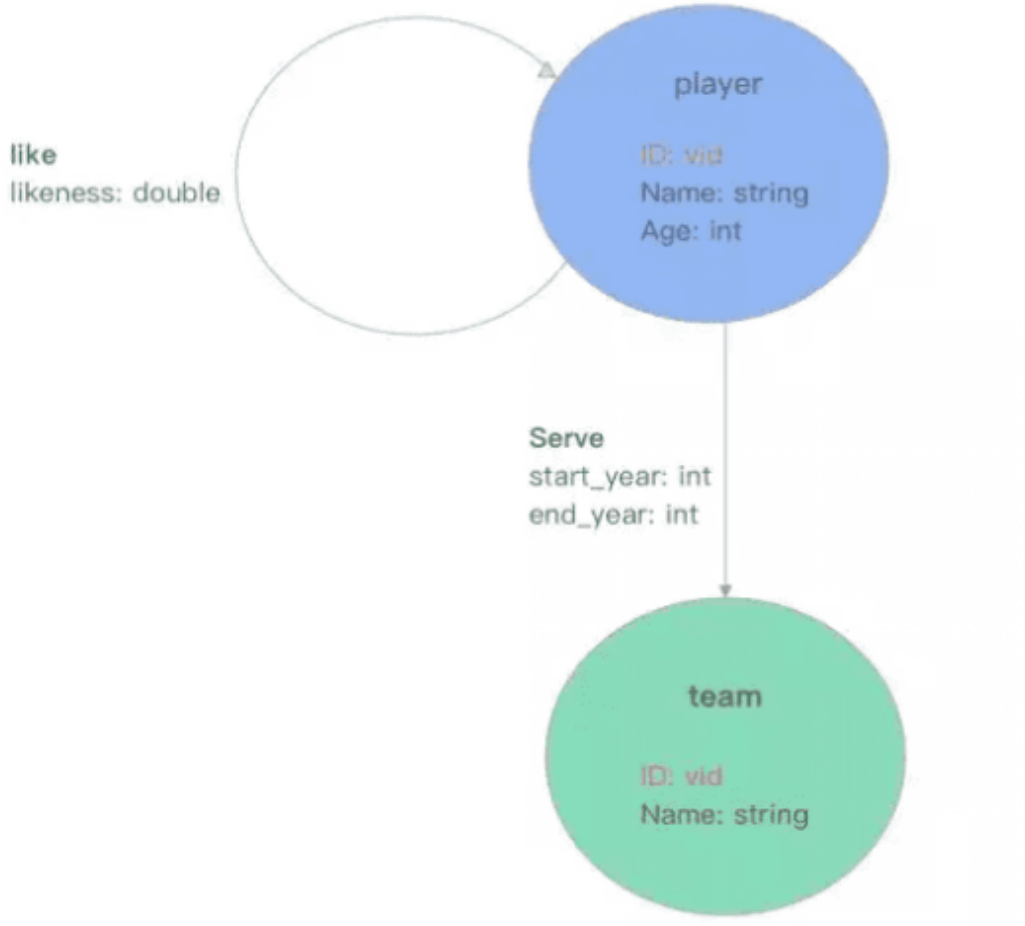
  cmake -DENABLE_BUILD_STORAGE=on -DENABLE_TESTING=OFF -DCMAKE_BUILD_TYPE=Debug -DNEBULA_COMMON_REPO_TAG=v2.0.0 -DNEBULA_STORAGE_REPO_TAG=v2.0.0 .

- make -j8
- sudo make install
- sudo /usr/local/nebula/scripts/nebula.service start all
- sudo /usr/local/nebula/scripts/nebula.service stop all

数据模型和系统架构设计

有向属性图

-

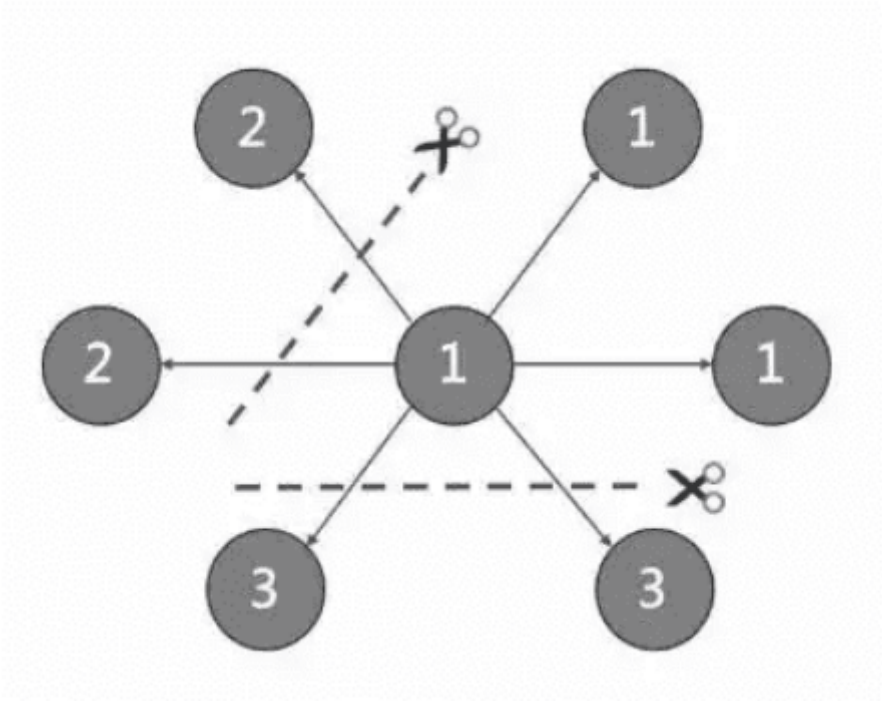


- 顶点 Vertex
  - 顶点由标签 `tag` 和对应 `tag` 的属性组构成， `tag` 代表顶点的类型，属性组代表 `tag` 拥有的一种或多种属性。一个顶点必须至少有一种类型，即标签，也可以有多种类型。每种标签有一组相对应的属性，我们称之为 `schema` 。

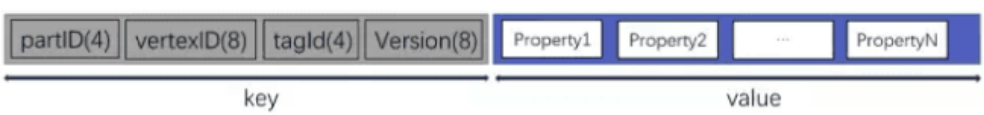
    - 如上图所示，有两种 `tag` 顶点：player 和 team。player 的 `schema` 有三种属性 `ID`（vid），`Name`（sting）和 `Age`（int）；team 的 `schema` 有两种属性 `ID`（vid）和 `Name`（string）。
    - Nebula Graph 是一种强 schema 的数据库，属性的名称和数据类型都是在数据写入前确定的。
- 插入点https://docs.nebula-graph.com.cn/2.0/3.ngql-guide/12.vertex-statements/1.insert-vertex/
    - CREATE TAG
    - INSERT VERTEX
    - 示例
        - ```
          # 插入不包含属性的点。
          nebula> CREATE TAG t1();
          nebula> INSERT VERTEX t1() VALUE "10":();
          ```
        - ```
          nebula> CREATE TAG t2 (name string, age int);
          nebula> INSERT VERTEX t2 (name, age) VALUES "11":("n1", 12);

          # 创建失败，因为"a13"不是int类型。
          nebula> INSERT VERTEX t2 (name, age) VALUES "12":("n1", "a13");

          # 一次插入2个点。
          nebula> INSERT VERTEX t2 (name, age) VALUES "13":("n3", 12), "14":
          ("n4", 8);
          ```
        - ```
          nebula> CREATE TAG t3(p1 int);
          nebula> CREATE TAG t4(p2 string);

          # 一次插入两个标签的属性到同一个点。
          nebula> INSERT VERTEX  t3 (p1), t4(p2) VALUES "21": (321, "hello");
          ```
- 边 Edge
    - 边由类型和边属性构成，一个顶点（起点 `src`）指向另一个顶点（终点 `dst`）的**关联关系**。此外，在 Nebula Graph 中我们将边类型称为 `edgetype`，每一条边**只有一种** `edgetype`，每种 `edgetype` 相应定义了这种边上属性的 `schema`。
    - 上面的图例中有两种类型的边，一种为 player 指向 player 的 **like** 关系，属性为 **likeness** (double)；另一种为 player 指向 team 的 **serve** 关系，两个属性分别为 start_year (int) 和 end_year (int)。
    - 插入边https://docs.nebula-graph.com.cn/2.0/3.ngql-guide/13.edge-statements/1.insert-edge/
        - ```
          # 插入不包含属性的边。
          nebula> CREATE EDGE e1();
          nebula> INSERT EDGE e1 () VALUES "10"->"11":();

          # 插入rank为1的边。
          nebula> INSERT EDGE e1 () VALUES "10"->"11"@1:();
          ```

```
nebula> CREATE EDGE e2 (name string, age int);
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", 1);

# 一次插入2条边。
nebula> INSERT EDGE e2 (name, age) VALUES \
        "12"->"13":("n1", 1), "13"->"14":("n2", 2);

# 创建失败，因为"a13"不是int类型。
nebula> INSERT EDGE e2 (name, age) VALUES "11"->"13":("n1", "a13");
```

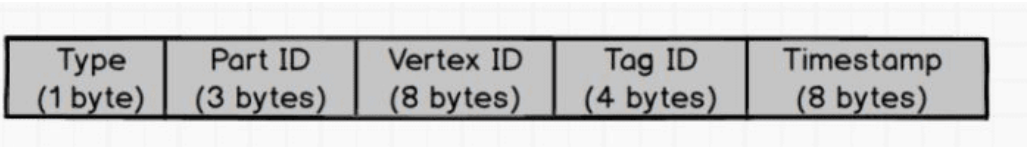
图分割 GraphPartition



数据模型 DataModel

- 重要博客
  - https://nebula-graph.com.cn/posts/nebula-graph-storage-engine-overview/
- 顶点
  - 每个顶点被建模为一个 `key-value`，根据其 vertexID（或简称 vid）哈希散列后，存储到对应的 partition 上
  - 

  - 更新



    - `Type`：1 个字节，用来表示 key 类型，当前的类型有 data, index, system 等
    - `Part ID`：3 个字节，用来表示数据分片 Partition，此字段主要用于 **Partition 重新分布(balance) 时方便根据前缀扫描整个 Partition 数据**

        - `Vertex ID`：8 个字节， 用来表示点的 ID
        - `Tag ID`：4 个字节, 用来表示关联的某个 tag
        - `Timestamp`：8 个字节，对用户不可见，未来实现分布式事务 (MVCC) 时使用
    - 2.0更新

Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	TagID (4 bytes)

        - 如果 VertexID 类型支持 string，则从占用 8 个字节的 int64 改成了固定长度的 `FIXED_STRING`，长度需要用户在 `create space` 时候指定长度。
- 边
    - 一条逻辑意义上的边，在 Nebula Graph 中将会被建模为两个独立的 `key-value` ，分别称为 `out-key` 和 `in-key` 。out-key 与这条边所对应的起点存储在同一个 partition 上，in-key 与这条边所对应的终点存储在同一个 partition 上。
    - 

    - 更新

Type (1 byte)	Part ID (3 bytes)	Vertex ID (8 bytes)	Edge Type (4 bytes)	Rank (8 bytes)	Vertex ID (8 bytes)	Timestamp (8 bytes)

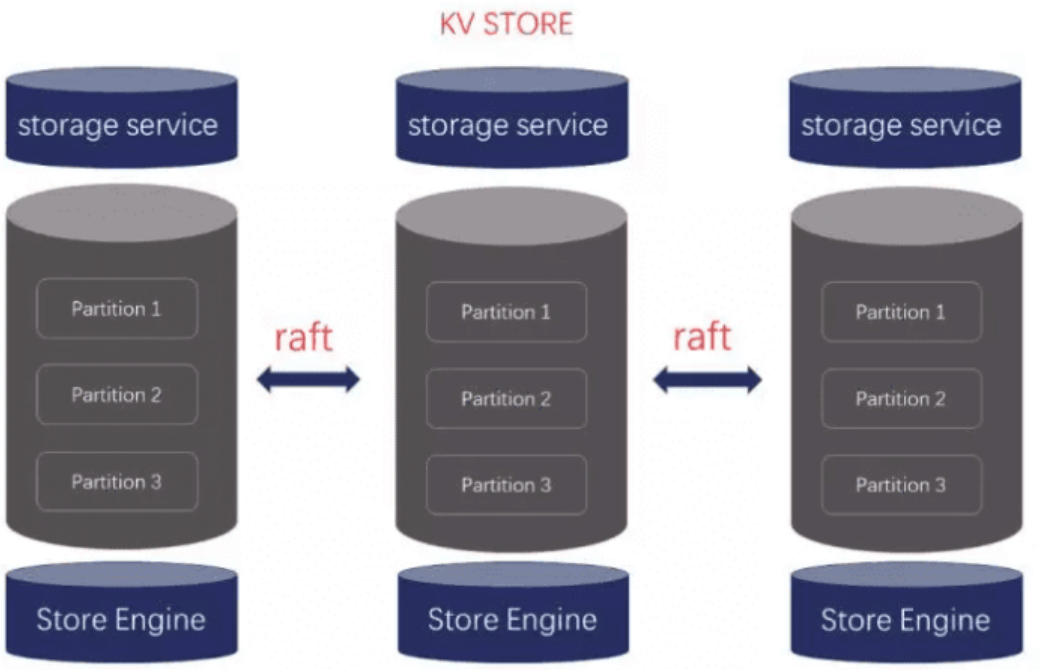
        - `Type`：1 个字节，用来表示 key 的类型，当前的类型有 data, index, system 等。
        - `Part ID`：3 个字节，用来表示数据分片 Partition，此字段主要用于 **Partition 重新分布(balance) 时方便根据前缀扫描整个 Partition 数据**
        - `Vertex ID`：8 个字节， 出边里面用来表示源点的 ID， 入边里面表示目标点的 ID。
        - `Edge Type`：4 个字节, 用来表示这条边的类型，如果大于 0 表示出边，小于 0 表示入边。
        - `Rank`：8 个字节，用来处理同一种类型的边存在多条的情况。用户可以根据自己的需求进行设置，这个字段可*存放交易时间、交易流水号*、或*某个排序权重*
        - `Vertex ID`：8 个字节, 出边里面用来表示目标点的 ID， 入边里面表示源点的 ID。
        - `Timestamp`：8 个字节，对用户不可见，未来实现分布式做事务的时候使用。
    - 针对 Edge Type 的值，若如果大于 0 表示出边，若 Edge Type 的值小于 0表示入边
    - 2.0更新

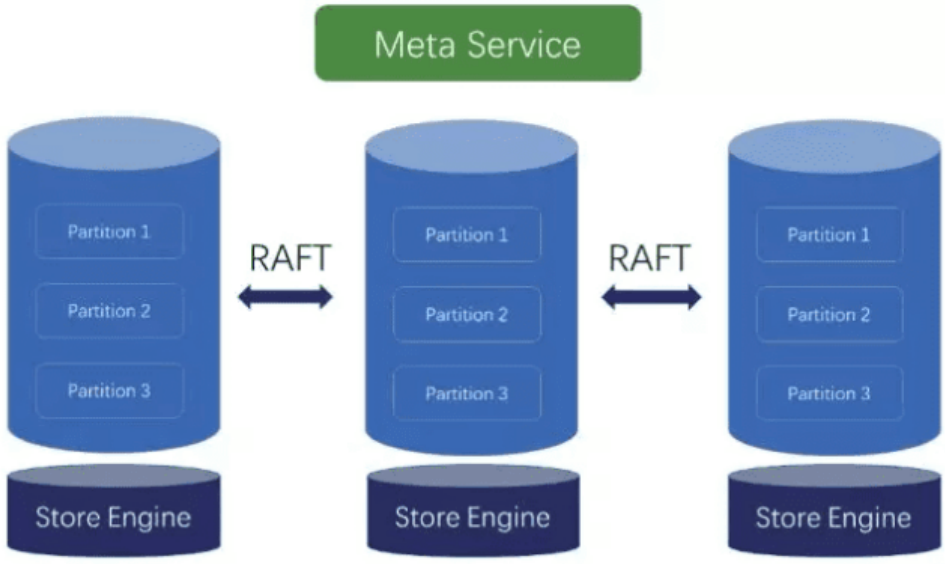
Type (1 byte)	PartID (3 bytes)	VertexID (n bytes)	Edge Type (4 bytes)	Rank (8 bytes)	VertexID (n bytes)	PlaceHolder (1 byte)

- KV存储
    - 对于点或边的属性信息，有对应的一组 kv pairs，Nebula 将它们编码后存在对应的 value 里。由于 Nebula 使用强类型 schema，所以在解码之前，需要先去 Meta Service 中取具体的 schema 信息。
- 分片
    - 通过对 Vertex ID 取模，同一个点的所有*出边*，*入边*以及这个点上所有关联的 *Tag 信息*都会被分到同一个 Partition，这种方式大大地提升了查询效率。

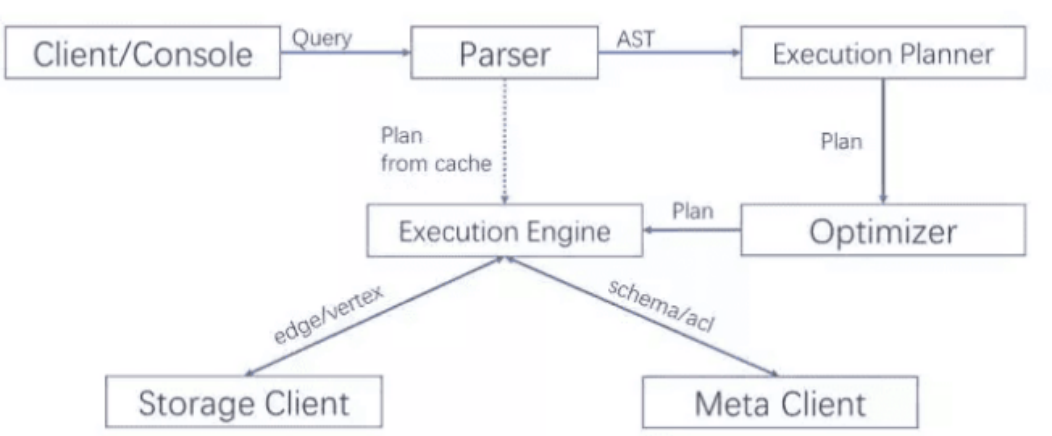
系统架构 Architecture

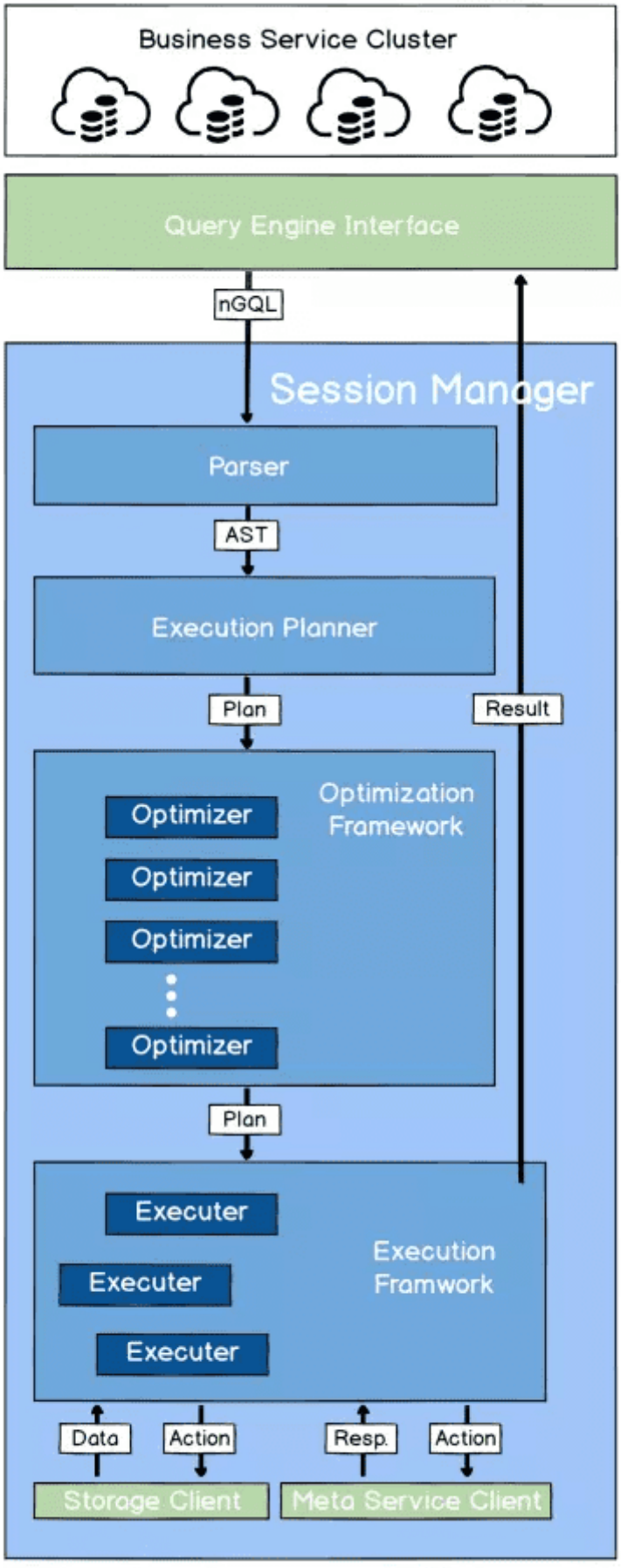
- 存储层 Storage
    - 重要博客

        - https://nebula-graph.com.cn/posts/nebula-graph-storage-engine-overview/
    - nebula-storaged
    - Storage Service 负责 Graph 数据存储。图数据被切分成很多的分片 Partition，相同 ID 的 Partition 组成一个 Raft Group，实现多副本一致性。Nebula Graph 默认的存储引擎是 RocksDB 的 Key-Value 存储。
    - 

    - Storage Service 共有三层，最底层是 Store Engine，它是一个单机版 local store engine，提供了对本地数据的 `get` / `put` / `scan` / `delete` 操作，相关的接口放在 KVStore / KVEngine.h 文件里面
- 元数据服务层 Metaservice
    - nebula-storaged
    - Meta Service 是整个集群的元数据管理中心，采用 Raft 协议保证高可用。主要提供两个功能:
        1. 管理各种元信息，比如 Schema
        2. 指挥存储扩容和数据迁移
    - 

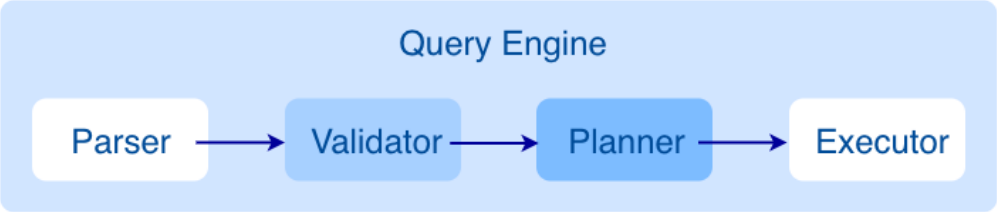
- 计算层 Query Engine & Query Language(nGQL)
    - nebula-storaged

- 
Client/Console
Query
Parser
AST
Execution Planner
Plan
from cache
Plan
Execution Engine
Plan
Optimizer
edge/vertex
schema/acl
Storage Client
Meta Client
-



- 2.0更新
  - 重要博客https://nebula-graph.com.cn/posts/nebula-query-engine-introduction/

    - 在 1.0 版本中 Query、Storage 和 Meta 模块代码不作区分放在同一个代码仓中，而 Nebula Graph 2.0 开始在架构上先解耦成三个代码仓：nebula-graph、nebula-common 和 nebula-storage，其中 nebula-common 中主要是表达式的定义、函数定义和一些公共接口、nebula-graph 主要负责 Query 模块、nebula-storage 主要负责 Storage 和 Meta 模块。
    - 主要分为 4 个子模块



        - Parser：词法语法解析模块
        - Validator：语句校验模块
        - Planner：执行计划和优化器模块
        - Executor：执行算子模块
    -
- 客户端 API & Console

安装部署

- https://nebula-graph.com.cn/posts/nebula-graph-database-deployment/
- https://docs.nebula-graph.com.cn/2.0/4.deployment-and-installation/1.resource-preparations/

一些其他博客

- 数据库和图数据库https://nebula-graph.com.cn/posts/graph-database-knowledge-volume-1/

开发

插入点的过程

- src/service/QueryEngine.cpp创建QueryInstance

```
68  void QueryEngine::execute(RequestContextPtr rctx) {
69      auto ectx = std::make_unique<QueryContext>(std::move(rctx),
70                                                 schemaManager_.get(),
71                                                 indexManager_.get(),
72                                                 storage_.get(),
73                                                 metaClient_.get(),
74                                                 charsetInfo_);
75      auto* instance = new QueryInstance(std::move(ectx), optimizer_.get());
76      instance->execute();
77  }
```

- src/service/QueryInstance.cpp创建scheduler并传入qctx

```
29  QueryInstance::QueryInstance(std::unique_ptr<QueryContext> qctx, Optimizer *optimizer) {
30      qctx_ = std::move(qctx);
31      optimizer_ = DCHECK_NOTNULL(optimizer);
32      scheduler_ = std::make_unique<Scheduler>(qctx_.get());
33  }
```

- src/service/QueryInstance.cpp运行后会调用validator

```
59  Status QueryInstance::validateAndOptimize() {
60      auto *rctx = qctx()->rctx();
61      VLOG(1) << "Parsing query: " << rctx->query();
62      auto result = GQLParser(qctx()).parse(rctx->query());
63      NG_RETURN_IF_ERROR(result);
64      sentence_ = std::move(result).value();
65
66      NG_RETURN_IF_ERROR(Validator::validate(sentence_.get(), qctx()));
67      NG_RETURN_IF_ERROR(findBestPlan());
68
69      return Status::OK();
70  }
```

```
35  void QueryInstance::execute() {
36      Status status = validateAndOptimize();
37      if (!status.ok()) {
38          onError(std::move(status));
39          return;
40      }
41
42      if (!explainOrContinue()) {
43          onFinish();
44          return;
45      }
46
47      scheduler_->schedule()
48          .then([this](Status s) {
49              if (s.ok()) {
50                  this->onFinish();
51              } else {
52                  this->onError(std::move(s));
53              }
54          })
55          .onError([this](const ExecutionError &e) { onError(e.status()); })
56          .onError([this](const std::exception &e) { onError(Status::Error("%s", e.what())); });
57  }
```

```
59  Status QueryInstance::validateAndOptimize() {
60      auto *rctx = qctx()->rctx();
61      VLOG(1) << "Parsing query: " << rctx->query();
62      auto result = GQLParser(qctx()).parse(rctx->query());
63      NG_RETURN_IF_ERROR(result);
64      sentence_ = std::move(result).value();
65
66      NG_RETURN_IF_ERROR(Validator::validate(sentence_.get(), qctx()));
67      NG_RETURN_IF_ERROR(findBestPlan());
68
69      return Status::OK();
70  }
```

- 接下来就是validator
  - src/validator/Validator.cpp利用makevalidator来找到每一个语句的validator

```cpp
Status Validator::validate(Sentence* sentence, QueryContext* qctx) {
    DCHECK(sentence != nullptr);
    DCHECK(qctx != nullptr);

    // Check if space chosen from session. if chosen, add it to context.
    auto session = qctx->rctx()->session();
    if (session->space().id > kInvalidSpaceID) {
        auto spaceInfo = session->space();
        qctx->vctx()->switchToSpace(std::move(spaceInfo));
    }

    auto validator = makeValidator(sentence, qctx);
    NG_RETURN_IF_ERROR(validator->validate());

    auto root = validator->root();
    if (!root) {
        return Status::SemanticError("Get null plan from sequential validator");
    }
    qctx->plan()->setRoot(root);
    return Status::OK();
}
```

对每个特殊的validator进行validate

```cpp
auto validator = makeValidator(sentence, qctx);
NG_RETURN_IF_ERROR(validator->validate());
```

makevalidator

```cpp
std::unique_ptr<Validator> Validator::makeValidator(Sentence* sentence, QueryContext* context) {
    auto kind = sentence->kind();
    switch (kind) {
        case Sentence::Kind::kExplain:
            return std::make_unique<ExplainValidator>(sentence, context);
        case Sentence::Kind::kSequential:
            return std::make_unique<SequentialValidator>(sentence, context);
        case Sentence::Kind::kGo:
            return std::make_unique<GoValidator>(sentence, context);
        case Sentence::Kind::kPipe:
            return std::make_unique<PipeValidator>(sentence, context);
        case Sentence::Kind::kAssignment:
            return std::make_unique<AssignmentValidator>(sentence, context);
        case Sentence::Kind::kSet:
            return std::make_unique<SetValidator>(sentence, context);
        case Sentence::Kind::kUse:
            return std::make_unique<UseValidator>(sentence, context);
        case Sentence::Kind::kGetSubgraph:
            return std::make_unique<GetSubgraphValidator>(sentence, context);
        case Sentence::Kind::kLimit:
            return std::make_unique<LimitValidator>(sentence, context);
        case Sentence::Kind::kOrderBy:
            return std::make_unique<OrderByValidator>(sentence, context);
        case Sentence::Kind::kYield:
```

validate

```cpp
Status Validator::validate() {
    if (!vctx_) {
        VLOG(1) << "Validate context was not given.";
        return Status::SemanticError("Validate context was not given.");
    }

    if (!sentence_) {
        VLOG(1) << "Sentence was not given";
        return Status::SemanticError("Sentence was not given");
    }

    if (!noSpaceRequired_ && !spaceChosen()) {
        VLOG(1) << "Space was not chosen.";
        return Status::SemanticError("Space was not chosen.");
    }

    if (!noSpaceRequired_) {
```

会调用

```
317
318         NG_RETURN_IF_ERROR(validateImpl());
319
```

- 以insertverticesvalidator为例

```
116         case Sentence::Kind::kInsertVertices:
117             return std::make_unique<InsertVerticesValidator>(sentence, context);
```

```
17  Status InsertVerticesValidator::validateImpl() {
18      spaceId_ = vctx_->whichSpace().id;
19      auto status = Status::OK();
20      do {
21          status = check();
22          if (!status.ok()) {
23              break;
24          }
25          status = prepareVertices();
26          if (!status.ok()) {
27              break;
28          }
29      } while (false);
30      return status;
31  }
32
33  Status InsertVerticesValidator::toPlan() {
34      auto doNode = InsertVertices::make(qctx_,
35                                         nullptr,
36                                         spaceId_,
37                                         std::move(vertices_),
38                                         std::move(tagPropNames_),
39                                         overwritable_);
40      root_ = doNode;
41      tail_ = root_;
42      return Status::OK();
43  }
```

toplan完成从语句到plannode的转化

src/planner/Mutate.h

```
22      static InsertVertices* make(QueryContext* qctx,
23                                  PlanNode* input,
24                                  GraphSpaceID spaceId,
25                                  std::vector<storage::cpp2::NewVertex> vertices,
26                                  std::unordered_map<TagID, std::vector<std::string>> tagPropNames,
27                                  bool overwritable) {
28          return qctx->objPool()->add(new InsertVertices(qctx,
29                                                         input,
30                                                         spaceId,
31                                                         std::move(vertices),
32                                                         std::move(tagPropNames),
33                                                         overwritable));
34      }
```

- src/scheduler/Scheduler.cpp中scheduler创建时传入qctx

```
25  Scheduler::Scheduler(QueryContext *qctx) : qctx_(DCHECK_NOTNULL(qctx)) {}
26
```

- src/scheduler/Scheduler.cpp开始创建executor

```
27  folly::Future<Status> Scheduler::schedule() {
28      auto executor = Executor::create(qctx_->plan()->root(), qctx_);
29      analyze(executor);
30      return doSchedule(executor);
31  }
```

- src/executor/Executor.cpp开始
  - makeExecutor加入执行计划

```
case PlanNode::Kind::kInsertVertices: {
    return pool->add(new InsertVerticesExecutor(node, qctx));
}
case PlanNode::Kind::kStart: {
    return pool->add(new StartExecutor(node, qctx));
}
```

- src/scheduler/Scheduler.cpp
  -
```
folly::Future<Status> Scheduler::schedule() {
    auto executor = Executor::create(qctx_->plan()->root(), qctx_);
    analyze(executor);
    return doSchedule(executor);
}
```
  - doSchedule(executor)

```
return execute(executor)
```

  - execute(executor)

```
return executor->execute().then([executor](Status s) {
        NG_RETURN_IF_ERROR(s);
        return executor->close();
    });
```

    至此会执行executor的execute函数，而executor就是在makeExecutor中add的那些executor，进入查看

- src/executor/mutate/InsertExecutor.h

```
class InsertVerticesExecutor final : public StorageAccessExecutor {
public:
    InsertVerticesExecutor(const PlanNode *node, QueryContext *qctx)
        : StorageAccessExecutor("InsertVerticesExecutor", node, qctx) {}

    folly::Future<Status> execute() override;

private:
    folly::Future<Status> insertVertices();
};
```

- src/executor/mutate/InsertExecutor.cpp

```
    return qctx()->getStorageClient()->addVertices(ivNode->getSpace(),
                                                   ivNode->getVertices(),
                                                   ivNode->getPropNames(),
                                                   ivNode-
>getOverwritable())
        .via(runner())
        .ensure([addVertTime]() {
            VLOG(1) << "Add vertices time: " << addVertTime.elapsedInUSec()
<< "us";
        })
        .then([this]
(storage::StorageRpcResponse<storage::cpp2::ExecResponse> resp) {
            SCOPED_TIMER(&execTime_);
            NG_RETURN_IF_ERROR(handleCompleteness(resp, false));
            return Status::OK();
        });
```

其实就是调用了StorageClient的addVertices

- modules/common/src/common/clients/storage/GraphStorageClient.cpp
  - ```
    GraphStorageClient::addVertices(GraphSpaceID space,
                                    std::vector<cpp2::NewVertex> vertices,
                                    std::unordered_map<TagID, std::vector<std::string>> propNames,
                                    bool overwritable,
                                    folly::EventBase* evb) {
        auto& clusters = status.value();
        std::unordered_map<HostAddr, cpp2::AddVerticesRequest> requests;
        for (auto& c : clusters) {
            auto& host = c.first;
            auto& req = requests[host];
            req.set_space_id(space);
            req.set_overwritable(overwritable);
            req.set_parts(std::move(c.second));
            req.set_prop_names(propNames);
        }
        return collectResponse(
            evb,
            std::move(requests),
            [] (cpp2::GraphStorageServiceAsyncClient* client,
                const cpp2::AddVerticesRequest& r) {
                return client->future_addVertices(r);
            });
    }
    ```
- modules/common/src/common/interface/gen-cpp2/GraphStorageServiceAsyncClient.cpp
  - future_addVertices
  - addVertices
  - addVerticesImpl
  - addVerticesT

继续深入剖析进程间通信过程-继续以存储为例

- modules文件夹
  - modules/common/src/common/interface：定义接口，使用thrift自动生成
    - modules/common/src/common/interface/storage.thrift
      - 里面有AddVerticesRequest
    ```

```
struct AddVerticesRequest {
    1: common.GraphSpaceID                      space_id,
    // partId => vertices
    2: map<common.PartitionID, list<NewVertex>>
        (cpp.template = "std::unordered_map")   parts,
    // A map from TagID -> list of prop_names
    // The order of the property names should match the data order specified
    //   in the NewVertex.NewTag.props
    3: map<common.TagID, list<binary>>
        (cpp.template = "std::unordered_map")   prop_names,
    // If true, it equals an (up)sert operation.
    4: bool                                     overwritable = true,
}
```

- 还有service

```
682
683  service GraphStorageService {
684      GetNeighborsResponse getNeighbors(1: GetNeighborsRequest req)
685
686      // Get vertex or edge properties
687      GetPropResponse getProps(1: GetPropRequest req);
688
689      ExecResponse addVertices(1: AddVerticesRequest req);
690      ExecResponse addEdges(1: AddEdgesRequest req);
691
692      ExecResponse deleteEdges(1: DeleteEdgesRequest req);
693      ExecResponse deleteVertices(1: DeleteVerticesRequest req);
694
695      UpdateResponse updateVertex(1: UpdateVertexRequest req);
696      UpdateResponse updateEdge(1: UpdateEdgeRequest req);
697
698      ScanVertexResponse scanVertex(1: ScanVertexRequest req)
699      ScanEdgeResponse scanEdge(1: ScanEdgeRequest req)
700
701      GetUUIDResp getUUID(1: GetUUIDReq req);
702
703      // Interfaces for edge and vertex index scan
704      LookupIndexResp lookupIndex(1: LookupIndexRequest req);
705
706      GetNeighborsResponse lookupAndTraverse(1: LookupAndTraverseRequest req);
707      ExecResponse addEdgesAtomic(1: AddEdgesRequest req);
708  }
709
```

- gen-xxx的文件夹中为自动生成的代码文件
  - 服务端接口文件：modules/common/src/common/interface/gen-cpp2/GraphStorageService.cpp
  - 客户端接口文件：modules/common/src/common/interface/gen-cpp2/GraphStorageServiceAsyncClient.cpp
- 接下来寻找第一层调用关系
  - 调用客户端的最直接函数位于 modules/common/src/common/clients/storage/GraphStorageClient.cpp
    - 猜测可能是自定义的客户端实现
  - 服务端实现这些接口函数位于 modules/storage/src/storage/GraphStorageServiceHandler.cpp，里面直接调用了各种processor
    - 比如modules/storage/src/storage/mutate/AddVerticesProcessor.cpp
      - doprocess中会直接进行插入KV的底层操作
      -
```
118         if (code != cpp2::ErrorCode::SUCCEEDED) {
119             handleAsync(spaceId_, partId, code);
120         } else {
121             doPut(spaceId_, partId, std::move(data));
122         }
```

- 关于thrift
  - https://zhuanlan.zhihu.com/p/45194118

- **Iface**：**服务端**通过实现 HelloWorldService.Iface 接口，向**客户端**的提供具体的**同步**业务逻辑。
- **AsyncIface**：**服务端**通过实现 HelloWorldService.Iface 接口，向**客户端**的提供具体的**异步**业务逻辑。
- **Client**：**客户端**通过 HelloWorldService.Client 的实例对象，以**同步**的方式**访问服务端**提供的服务方法。
- **AsyncClient**：**客户端**通过 HelloWorldService.AsyncClient 的实例对象，以**异步**的方式**访问服务端**提供的服务方法。

  - https://www.cnblogs.com/shangxiaofei/p/8504932.html
  - https://blog.csdn.net/houjixin/article/details/42778335

尝试添加addRDFVertices通路（自底向上至Executor）

- 首先修改thrift文件，添加addRDFVertices的service以及相应的request结构体，重新make，这一步会生成新的接口文件。
- 生成接口
  - 客户端：modules/common/src/common/interface/gen-cpp2/GraphStorageServiceAsyncClient.cpp

```
template <typename Protocol_>
void GraphStorageServiceAsyncClient::addRDFVerticesT(Protocol_* prot, bool useSync, apache::thrift::RpcOptio
  struct HeaderAndConnContext {
    HeaderAndConnContext() : header(apache::thrift::transport::THeader::ALLOW_BIG_FRAMES) {}

    apache::thrift::transport::THeader header;
    apache::thrift::Cpp2ConnContext connContext;
  };
  auto headerAndConnContext = std::make_shared<HeaderAndConnContext>();
  std::shared_ptr<apache::thrift::transport::THeader> header(headerAndConnContext, &headerAndConnContext->he
  header->setProtocolId(getChannel()->getProtocolId());
  header->setHeaders(rpcOptions.releaseWriteHeaders());
  headerAndConnContext->connContext.setRequestHeader(header.get());
  std::unique_ptr<apache::thrift::ContextStack> ctx = this->getContextStack(this->getServiceName(), "GraphSt
  GraphStorageService_addRDFVertices_pargs args;
  args.get<0>().value = const_cast< ::nebula::storage::cpp2::AddRDFVerticesRequest*>(&req);
  auto sizer = [&](Protocol_* p) { return args.serializedSizeZC(p); };
  auto writer = [&](Protocol_* p) { args.write(p); };
  apache::thrift::clientSendT<Protocol_>(prot, rpcOptions, std::move(callback), std::move(ctx), header, chan
  headerAndConnContext->connContext.setRequestHeader(nullptr);
}
```

  - 服务端：modules/common/src/common/interface/gen-cpp2/GraphStorageService.cpp

```
55  void GraphStorageServiceSvIf::addRDFVertices( ::nebula::storage::cpp2::ExecResponse& /*_return*/, const  ::ne
56    apache::thrift::detail::si::throw_app_exn_unimplemented("addRDFVertices");
57  }
58
59  folly::Future< ::nebula::storage::cpp2::ExecResponse> GraphStorageServiceSvIf::future_addRDFVertices(const  :
60    return apache::thrift::detail::si::future_returning([&]( ::nebula::storage::cpp2::ExecResponse& _return) {
61  }
62
63  void GraphStorageServiceSvIf::async_eb_addRDFVertices(std::unique_ptr<apache::thrift::HandlerCallback< ::nebu
64    apache::thrift::detail::si::async_eb(this, std::move(callback), [this, req = std::move(req)]() mutable { re
65  }
```

- 首先先对服务端进行添加
  - future_addRDFVertices是暴露的接口，默认会调用同文件的addRDFVertices函数，这个函数会抛出"没有实现接口"的警告
  - 也就是说，我们需要自己继承future_addRDFVertices这个接口，并且实现它，这样在服务端调用这个函数的时候会执行我们的代码。
  - modules/storage/src/storage/GraphStorageServiceHandler.cpp就是上述文件，实现接口
    - GraphStorageServiceHandler会继承GraphStorageServiceSvIf类
    - GraphStorageServiceSvIf类就是thrift自动生成的那个服务类，包含我们要实现的接口
    - 在modules/storage/src/storage/GraphStorageServiceHandler.cpp中实现这个接口
    - 不要忘记添加头文件
    - 不要忘记初始化counter
  - 添加真正的执行代码，也就是相应的processor
    - 先复制一份

- modules/storage/src/storage/CMakeLists.txt这里也有可能要加

```
nebula_add_library(
    graph_storage_service_handler OBJECT
    GraphStorageServiceHandler.cpp
    context/StorageExpressionContext.cpp
    mutate/AddVerticesProcessor.cpp
    mutate/DeleteVerticesProcessor.cpp
    mutate/AddEdgesProcessor.cpp
    mutate/AddEdgesAtomicProcessor.cpp
    mutate/DeleteEdgesProcessor.cpp
    mutate/UpdateVertexProcessor.cpp
    mutate/UpdateEdgeProcessor.cpp
    query/GetNeighborsProcessor.cpp
    query/GetPropProcessor.cpp
    query/ScanVertexProcessor.cpp
    query/ScanEdgeProcessor.cpp
    index/LookupProcessor.cpp
)
```

- 不要忘记添加头文件

GraphStorageServiceHandler.cpp 3 | CMakeLists.txt | AddRDFVerticesProcessor.cpp

modules > storage > src > storage > GraphStorageServiceHandler.cpp > ...

```
/* Copyright (c) 2020 vesoft inc. All rights reserved.
 *
 * This source code is licensed under Apache 2.0 License,
 * attached with Common Clause Condition 1.0, found in the Li
 */

#include "storage/GraphStorageServiceHandler.h"
#include "storage/mutate/AddVerticesProcessor.h"
#include "storage/mutate/AddRDFVerticesProcessor.h"
#include "storage/mutate/AddEdgesProcessor.h"
#include "storage/mutate/AddEdgesAtomicProcessor.h"
```

- 对客户端进行添加
  - modules/common/src/common/interface/gen-cpp2/GraphStorageServiceAsyncClient.cpp是自动生成的客户端接口
    - 最外层暴露接口为

```
folly::Future< ::nebula::storage::cpp2::ExecResponse> GraphStorageServiceAsyncClient::future_addRDFVertices(const  ::nebula::storage::cpp2::AddRDFVerticesRequest& req) {
  ::apache::thrift::RpcOptions rpcOptions;
  return future_addRDFVertices(rpcOptions, req);
}
```

    - 接下来调用

```
folly::Future< ::nebula::storage::cpp2::ExecResponse> GraphStorageServiceAsyncClient::future_addRDFVertices(apache::thrift::RpcOptions& rpcOptions, const  ::nebula::storag
  folly::Promise< ::nebula::storage::cpp2::ExecResponse> _promise;
  auto _future = _promise.getFuture();
  auto callback = std::make_unique<apache::thrift::FutureCallback< ::nebula::storage::cpp2::ExecResponse>>(std::move(_promise), recv_wrapped_addRDFVertices, channel_);
  addRDFVertices(rpcOptions, std::move(callback), req);
  return _future;
}
```

    - 接下来调用

```
void GraphStorageServiceAsyncClient::addRDFVertices(apache::thrift::RpcOptions& rpcOptions, std::unique_ptr<apache::thrift::RequestCallback> callback, const  ::nebula::sto
  addRDFVerticesImpl(false, rpcOptions, std::move(callback), req);
}
```

- 接下来调用

```
void GraphStorageServiceAsyncClient::addRDFVerticesImpl(bool useSync, apache::thrift::RpcOptions& rpcOptions, std::unique_ptr<apache::thrift::RequestCallback> callback, co
  switch(getChannel()->getProtocolId()) {
    case apache::thrift::protocol::T_BINARY_PROTOCOL:
    {
      apache::thrift::BinaryProtocolWriter writer;
      addRDFVerticesT(&writer, useSync, rpcOptions, std::move(callback), req);
      break;
    }
    case apache::thrift::protocol::T_COMPACT_PROTOCOL:
    {
      apache::thrift::CompactProtocolWriter writer;
      addRDFVerticesT(&writer, useSync, rpcOptions, std::move(callback), req);
      break;
    }
    default:
    {
      apache::thrift::detail::ac::throw_app_exn("Could not find Protocol");
    }
  }
}
```

- 接下来调用

```
template <typename Protocol_>
void GraphStorageServiceAsyncClient::addRDFVerticesT(Protocol_* prot, bool useSync, apache::thrift::RpcOptions& rpcOptions, std::unique_ptr<apache::thrift::RequestCallback
  struct HeaderAndConnContext {
    HeaderAndConnContext() : header(apache::thrift::transport::THeader::ALLOW_BIG_FRAMES) {}

    apache::thrift::transport::THeader header;
    apache::thrift::Cpp2ConnContext connContext;
  };
  auto headerAndConnContext = std::make_shared<HeaderAndConnContext>();
  std::shared_ptr<apache::thrift::transport::THeader> header(headerAndConnContext, &headerAndConnContext->header);
  header->setProtocolId(getChannel()->getProtocolId());
  header->setHeaders(rpcOptions.releaseWriteHeaders());
  headerAndConnContext->connContext.setRequestHeader(header.get());
  std::unique_ptr<apache::thrift::ContextStack> ctx = this->getContextStack(this->getServiceName(), "GraphStorageService.addRDFVertices", &headerAndConnContext->connContex
  GraphStorageService_addRDFVertices_pargs args;
  args.get<0>().value = const_cast< ::nebula::storage::cpp2::AddRDFVerticesRequest*>(&req);
  auto sizer = [&](Protocol_* p) { return args.serializedSizeZC(p); };
  auto writer = [&](Protocol_* p) { args.write(p); };
  apache::thrift::clientSendT<Protocol_>(prot, rpcOptions, std::move(callback), std::move(ctx), header, channel_.get(), "addRDFVertices", writer, sizer, apache::thrift::Rp
  headerAndConnContext->connContext.setRequestHeader(nullptr);
}
```

此时与服务端进行通讯

- modules/common/src/common/clients/storage/GraphStorageClient.h和modules/common/src/common/clients/storage/GraphStorageClient.cpp是自定义的最底层客户端代码，直接调用自动生成的代码接口
  - 添加相应rdf代码（先直接复制）
  - 这里暴露出去的接口是addRDFVertices

```
folly::SemiFuture<StorageRpcResponse<cpp2::ExecResponse>>
GraphStorageClient::addRDFVertices(GraphSpaceID space,
                                   std::vector<cpp2::NewVertex> vertices,
                                   std::unordered_map<TagID, std::vector<std::string>> propNames,
                                   bool overwritable,
                                   folly::EventBase* evb) {
```

- 继续往上走是executor，关于插入点和边的executor位于src/executor/mutate/InsertExecutor.cpp
  - 加入InsertRDFVerticesExecutor

```
folly::Future<Status> InsertRDFVerticesExecutor::execute() {
    return insertRDFVertices();
}

folly::Future<Status> InsertRDFVerticesExecutor::insertRDFVertices() {
    SCOPED_TIMER(&execTime_);

    auto *ivNode = asNode<InsertVertices>(node());
    time::Duration addRDFVertTime;
    return qctx()->getStorageClient()->addRDFVertices(ivNode->getSpace(),
                                                       ivNode->getVertices(),
                                                       ivNode->getPropNames(),
                                                       ivNode->getOverwritable())
        .via(runner())
        .ensure([addRDFVertTime]() {
            VLOG(1) << "Add vertices time: " << addRDFVertTime.elapsedInUSec() << "us";
        })
        .then([this](storage::StorageRpcResponse<storage::cpp2::ExecResponse> resp) {
            SCOPED_TIMER(&execTime_);
            NG_RETURN_IF_ERROR(handleCompleteness(resp, false));
            return Status::OK();
        });
}
```

- 继续往上位于src/executor/Executor.cpp，与暑假工作联系起来。

尝试添加addRDFVertices通路（自顶向下至Executor）

- 现在还是位于客户端的部分，但是我们从顶层向下添加至executor
- 添加insertrdfvertices的语法等
  - src/parser/parser.yy

        - 不赘述
    - src/parser/scanner.lex

```
"VERTEX"                { return TokenType::KW_VERTEX; }
"RDFVERTEX"             { return TokenType::KW_RDFVERTEX; }
```

    - .linters/cpp/checkKeyword.py

```
    'KW_DESC',
    'KW_VERTEX',
    'KW_RDFVERTEX',
    'KW_EDGE',
    'KW_EDGES',
```

- 一些表面的东西，plannode的kind定义
    - src/planner/PlanNode.h

```
        kInsertVertices,
        kInsertRDFVertices,
        kInsertEdges,
```

    - src/executor/Executor.cpp

```
        }
        case PlanNode::Kind::kInsertRDFVertices: {
            return pool->add(new InsertRDFVerticesExecutor(node, qctx));
        }
```

    - src/planner/PlanNode.cpp

```
            return "InsertVertices";
        case Kind::kInsertRDFVertices:
            return "InsertRDFVertices";
        case Kind::kInsertEdges:
```

    - 先进行一次编译
        - 上层向下到添加了rdfinsert的语句，没有更改到sentence
        - 下层向上到达了kinsertrdfvertices，这里就可以直接调用InsertRDFVerticesExecutor，更改完plannode的kind，但是没有添加sentence的kind
        - 中间过度为sentence的定向和validator的修改
- 继续深入
    - 注意注意kind有两个，分别是plannode的kind和sentence的kind
        - src/planner/PlanNode.h

```
23  class PlanNode {
24  public:
25      enum class Kind : uint8_t {
26          kUnknown = 0,
27          kStart,
28          kGetNeighbors,
29          kGetVertices,
30          kGetEdges,
31          kIndexScan,
32          kFilter,
33          kUnion,
34          kUnionAllVersionVar,
35          kIntersect,
36          kMinus,
37          kProject,
38          kUnwind,
39          kSort,
```

- src/parser/Sentence.h

```
26  class Sentence {
27  public:
28      virtual ~Sentence() {}
29      virtual std::string toString() const = 0;
30
31      enum class Kind : uint32_t {
32          kUnknown,
33          kExplain,
34          kSequential,
35          kGo,
36          kSet,
37          kPipe,
38          kUse,
39          kMatch,
40          kAssignment,
41          kCreateTag,
42          kAlterTag,
43          kCreateEdge,
44          kAlterEdge,
45          kDescribeTag,
46          kDescribeEdge,
47          kCreateTagIndex,
48          kCreateEdgeIndex,
```

- 前面讲过了，把二者联系起来的是validator，进行了从sentence kind到plannode kind的转化

```
54  private:
55      InsertVertices(QueryContext* qctx,
56                     PlanNode* input,
57                     GraphSpaceID spaceId,
58                     std::vector<storage::cpp2::NewVertex> vertices,
59                     std::unordered_map<TagID, std::vector<std::string>> tagPropNames,
60                     bool overwritable)
61          : SingleDependencyNode(qctx, Kind::kInsertVertices, input),
62            spaceId_(spaceId),
63            vertices_(std::move(vertices)),
64            tagPropNames_(std::move(tagPropNames)),
65            overwritable_(overwritable) {}
```

- 一些表面的对象，sentence的kind
  - src/parser/Sentence.h

```
54          kDropEdge,
55          kInsertVertices,
56          kInsertRDFVertices,
57          kUpdateVertex,
```

  - src/service/PermissionCheck.cpp

```
111         case Sentence::Kind::kInsertVertices :
112         case Sentence::Kind::kInsertRDFVertices :
```

  - src/validator/Validator.cpp
    - 先等等
- 所有sentence位于src/parser/MutateSentences.h和src/parser/MutateSentences.cpp中定义
  - 添加InsertRDFVerticesSentence

```
191 class InsertRDFVerticesSentence final : public Sentence {
192 public:
193     InsertRDFVerticesSentence(VertexTagList *tagList,
194                               VertexRowList *rows,
195                               bool overwritable = true) {
196         tagList_.reset(tagList);
197         rows_.reset(rows);
198         overwritable_ = overwritable;
199         kind_ = Kind::kInsertRDFVertices;
200     }
201
202     bool overwritable() const {
203         return overwritable_;
204     }
205
206     auto tagItems() const {
207         return tagList_->tagItems();
208     }
209
210     std::vector<VertexRowItem*> rows() const {
211         return rows_->rows();
212     }
213
214     std::string toString() const override;
215
216 private:
217     bool                                    overwritable_{true};
218     std::unique_ptr<VertexTagList>          tagList_;
219     std::unique_ptr<VertexRowList>          rows_;
220 };
```

  - src/parser/MutateSentences.cpp重写tostring

```cpp
std::string InsertRDFVerticesSentence::toString() const {
    std::string buf;
    buf.reserve(256);
    buf += "INSERT RDFVERTEX ";
    buf += tagList_->toString();
    buf += " VALUES ";
    buf += rows_->toString();
    return buf;
}
```

- validator相关的东西
  - plannode深入定义：

    src/planner/Mutate.h

```cpp
class InsertRDFVertices final : public SingleDependencyNode {
public:
    static InsertRDFVertices* make(QueryContext* qctx,
                                   PlanNode* input,
                                   GraphSpaceID spaceId,
                                   std::vector<storage::cpp2::NewVertex> vertices,
                                   std::unordered_map<TagID, std::vector<std::string>> tagPropNames
                                   bool overwritable) {
        return qctx->objPool()->add(new InsertRDFVertices(qctx,
                                                          input,
                                                          spaceId,
                                                          std::move(vertices),
                                                          std::move(tagPropNames),
                                                          overwritable));
    }
```

```cpp
private:
    InsertRDFVertices(QueryContext* qctx,
                      PlanNode* input,
                      GraphSpaceID spaceId,
                      std::vector<storage::cpp2::NewVertex> vertices,
                      std::unordered_map<TagID, std::vector<std::string>> tagPropNames,
                      bool overwritable)
        : SingleDependencyNode(qctx, Kind::kInsertRDFVertices, input),
          spaceId_(spaceId),
          vertices_(std::move(vertices)),
          tagPropNames_(std::move(tagPropNames)),
          overwritable_(overwritable) {}

private:
    GraphSpaceID spaceId_{-1};
    std::vector<storage::cpp2::NewVertex> vertices_;
    std::unordered_map<TagID, std::vector<std::string>> tagPropNames_;
    bool overwritable_;
};
```

    和src/planner/Mutate.cpp

```cpp
std::unique_ptr<PlanNodeDescription> InsertRDFVertices::explain() const {
    auto desc = SingleDependencyNode::explain();
    addDescription("spaceId", folly::to<std::string>(spaceId_), desc.get());
    addDescription("overwritable", util::toJson(overwritable_), desc.get());

    folly::dynamic tagPropsArr = folly::dynamic::array();
    for (const auto &p : tagPropNames_) {
        folly::dynamic obj = folly::dynamic::object();
        obj.insert("tagId", p.first);
        obj.insert("props", util::toJson(p.second));
        tagPropsArr.push_back(obj);
    }
    addDescription("tagPropNames", folly::toJson(tagPropsArr), desc.get());
    addDescription("vertices", folly::toJson(util::toJson(vertices_)), desc.get());
    return desc;
}

std::unique_ptr<PlanNodeDescription> InsertEdges::explain() const {
```

  - 正式添加validator

    src/validator/Validator.cpp报错很正常，开始定义真正的InsertRDFVerticesValidator

```
118         case Sentence::Kind::kInsertRDFVertices:
119             return std::make_unique<InsertRDFVerticesValidator>(sentence, context);
```

src/validator/MutateValidator.h

```
44  class InsertRDFVerticesValidator final : public Validator {
45  public:
46      InsertRDFVerticesValidator(Sentence* sentence, QueryContext* context)
47          : Validator(sentence, context) {
48      }
49
50  private:
51      Status validateImpl() override;
52
53      Status toPlan() override;
54
55      Status check();
56
57      Status prepareVertices();
58
59  private:
60      using TagSchema = std::shared_ptr<const meta::SchemaProviderIf>;
61      GraphSpaceID                                            spaceId_{-1};
62      std::vector<VertexRowItem*>                             rows_;
63      std::unordered_map<TagID, std::vector<std::string>>     tagPropNames_;
64      std::vector<std::pair<TagID, TagSchema>>                schemas_;
65      uint16_t                                                propSize_{0};
66      bool                                                    overwritable_{false};
67      std::vector<storage::cpp2::NewVertex>                   vertices_;
68  };
```

src/validator/MutateValidator.cpp添加各种实现

```
142 Status InsertRDFVerticesValidator::validateImpl() {
143     spaceId_ = vctx_->whichSpace().id;
144     auto status = Status::OK();
145     do {
146         status = check();
147         if (!status.ok()) {
148             break;
149         }
150         status = prepareVertices();
151         if (!status.ok()) {
152             break;
153         }
154     } while (false);
155     return status;
156 }
157
158 Status InsertRDFVerticesValidator::toPlan() {
159     auto doNode = InsertRDFVertices::make(qctx_,
160                                           nullptr,
161                                           spaceId_,
162                                           std::move(vertices_),
163                                           std::move(tagPropNames_),
164                                           overwritable_);
165     root_ = doNode;
166     tail_ = root_;
167     return Status::OK();
168 }
169
170 Status InsertRDFVerticesValidator::check() {
```

- 连接!

```
319         case PlanNode::Kind::kInsertRDFVertices: {
320             return pool->add(new InsertRDFVerticesExecutor(node, qctx));
321         }
```

最低层KV存储

- modules/storage/src/kvstore/KVEngine.h
- future_addVertices值得注意

- 一个在：modules/storage/src/storage/GraphStorageServiceHandler.cpp
  - ```
    folly::Future<cpp2::ExecResponse>
    GraphStorageServiceHandler::future_addVertices(const cpp2::AddVerticesRequest& req) {
        auto* processor = AddVerticesProcessor::instance(env_, &kAddVerticesCounters, &vertexCache_);
        RETURN_FUTURE(processor);
    }
    ```

    ```
    #define RETURN_FUTURE(processor) \
        auto f = processor->getFuture(); \
        processor->process(req); \
        return f;
    ```

    一个宏定义：执行process

    - modules/storage/src/storage/mutate/AddVerticesProcessor.cpp

      ```
      void AddVerticesProcessor::process(const cpp2::AddVerticesRequest& req) {
      ```

- 另一个在：modules/common/src/common/interface/gen-cpp2/GraphStorageServiceAsyncClient.cpp
-
      ```