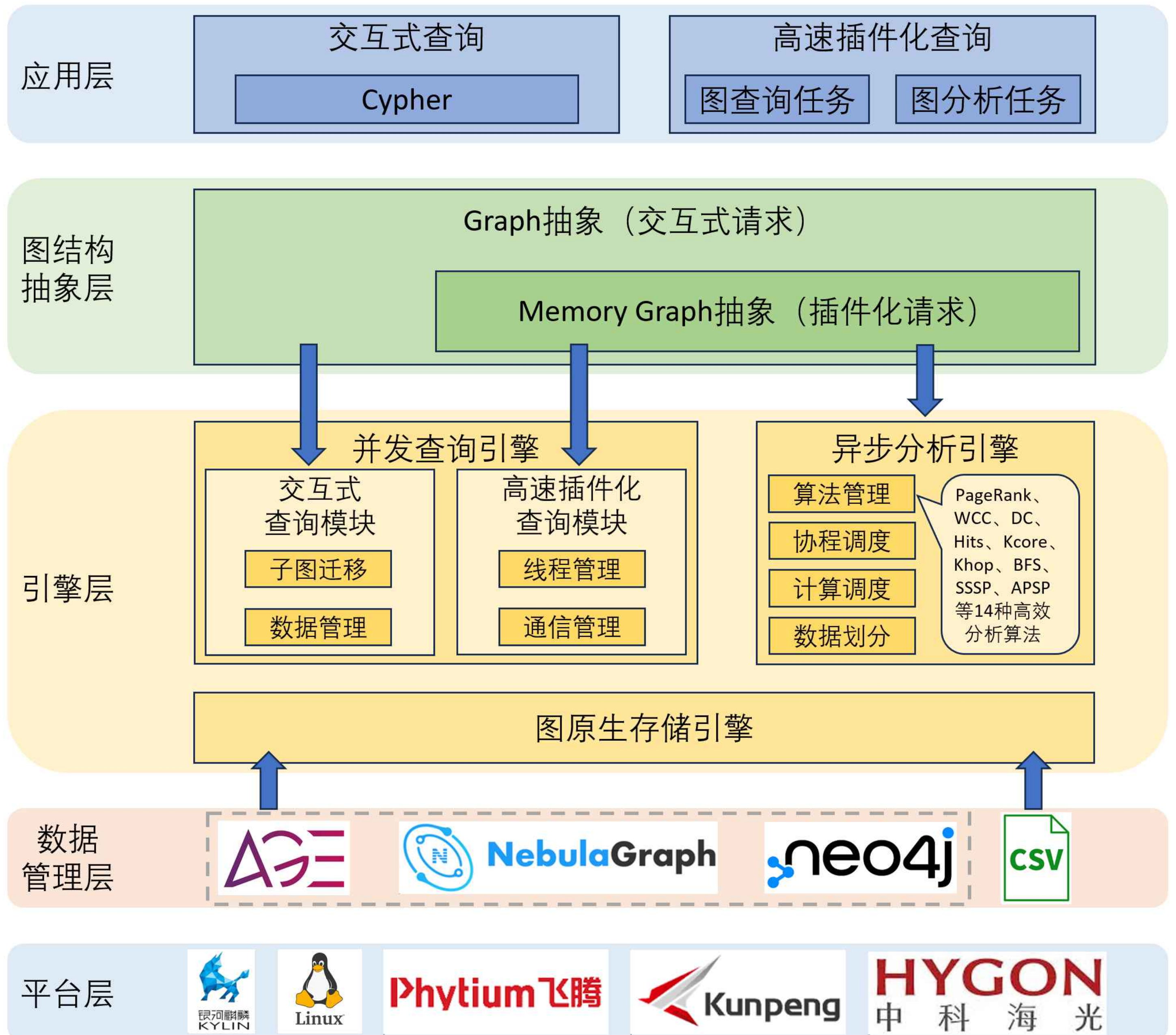
九源智能推理图数据库用户手册

九源智能推理图数据库是一个基于国产化硬件平台设计的、面向智能推理场景的高性能图数据库系统。本系统能够支持各类丰富的计算与查询任务，支持不同的应用使用方式，从而满足智能推理场景复杂的应用需求。本系统功能全面，既支持 Cypher 查询语言，又支持插件化的图分析与图查询任务，并且在性能上远超国内外主流图数据库产品。在图数据库领域权威的 LDBC 基准测试中，性能超过目前榜单世界第一的 AtlasGraph 50%；在图分析任务场景下，目前支持的14种高速插件分析算法性能均超过国际主流产品 Neo4j。

整体架构设计

首先介绍九源智能推理图数据库的整体架构设计，下面是该系统的整体架构图：



目前九源智能推理图数据库的架构设计包含了五个层面的研究，分别是：

- 应用层
- 图结构抽象层
- 引擎层
- 数据管理层
- 平台层

应用层

在应用层面，九源智能推理图数据库支持不同的使用方式与丰富的查询类型，用以满足智能推理场景下复杂的用户需求，在保证图数据库通用功能的同时提供给用户定制特殊任务的接口，用以支持不断演进发展的智能推理应用，实现了系统功能的完备性与可扩展性。

从使用方式的层面看，主要支持两种不同使用方式的查询，分别是**交互式查询**和**高速插件化查询**。其中交互式查询是传统的数据库使用方式，本系统支持 Cypher 声明式语言 —— 图数据库领域使用最广泛的查询语言，作为交互式查询的基础。而高速插件化查询是九源智能推理图数据库专为智能推理应用中的高性能场景开发的使用方式，开发者可以通过本系统提供的编程接口，只需要编写几十行代码，就可以利用本系统自研的高性能引擎开发出高效的定制化查询，解决了通用查询语言和通用系统在特定应用场景下性能需求难以满足的痛点。

从查询类型的层面看，主要支持**图查询任务**和**图分析任务**。图查询任务主要涉及在图数据库中检索特定的数据或数据模式，包括从图中找到特定的顶点、边或者点边的特定模式。这类任务通常通过图数据库查询语言（如 Cypher）来实现，侧重于数据模式的检索。而图分析任务更加关注从整个图的结构中挖掘信息，主要通过执行一些复杂的算法来进一步理解图中的模式和关系。常见的图分析任务包括中心性算法、社群发现、路径算法以及连通性算法等。

图结构抽象层

图结构抽象层主要是为了简化九源智能推理图数据库的接口和使用。由于九源智能推理图数据库支持丰富的功能，考虑到用户的学习成本与使用成本，本系统在客户端层面提供了图结构抽象。用户可以直接通过客户端获得一个 **Graph 抽象**，这个 Graph 抽象对应着图数据库底层存储的同名的图。基于这个 Graph 抽象，用户可以直接进行 Schema 操作、数据导入操作以及交互式查询。

为了提供高性能，支撑高速插件化查询，本系统还提出了一个 **MemoryGraph 抽象**，该抽象能够从一个 Graph 抽象中派生，并且可以从中自定义所需的子图。在定义子图之后，对应的子图会被加载到一个自研的分布式图原生存储引擎中，从而支撑后续的高速插件化查询。同一个 MemoryGraph 抽象能够同时支持插件化的图查询任务与图分析任务，并且可以在多个任务之间复用，省去了重复加载子图的开销。

引擎层

为了满足智能推理场景的复杂需求，九源智能推理图数据库中包含了三套自研的高性能引擎，分别是**并发查询引擎**、**异步分析引擎**以及**图原生存储引擎**。

并发查询引擎用于支持图查询任务。图查询任务主要包含两种使用方式，交互式和插件化。引擎中针对这两种使用方式分别实现了对应的模块，交互式查询模块和高速插件化查询模块。在交互式查询模块中，主要是处理 Cypher 查询。根据上层传来的请求参数或 Cypher 查询，去下层的图数据管理系统中查询对应的数据。交互式查询模块可以对接多种图数据管理系统，从而适应多种复杂的应用场景。此外，交互式查询模块还支持子图迁移，用户可以自定义所需的子图，然后通过交互式查询模块将数据从图数据管理系统中抽取到图原生存储引擎中，从而支持高速插件化查询。在高速插件化查询模块中，通过精细的线程管理和通信管理，实现了高效的调度与计算，并且实现了网络数据的批量通信，从而有效地利用了现代多核处理器以及高性能网卡的能力。该模块提供了简洁的用户编程接口，隐藏了底层的实现，使得以少量代码实现高速定制化查询成为了可能。

异步分析引擎用于支持图分析任务。这类任务的特点是计算量大、通信量大以及算法复杂性高，因此针对每个图分析任务单独设计高效的算法实现非常困难。本引擎中通过设计通用的协程调度、计算调度以及数据划分机制，将图分析任务中的多个核心步骤统一抽象并优化，并提供了简洁的用户编程接口。用户只需要几十行代码就可以编写出复杂的图分析算法，并且有效地利用现有的计算资源去进行分布式计算，从而满足易用性与性能需求。

图原生存储引擎是一个子图管理引擎，可以通过自定义子图映射的方式，从其他数据管理系统中获取用户所需的子图并存储到本引擎中。本引擎通过内存映射的方式，将子图映射到内存中，从而供并发查询引擎中的高速插件化查询模块和异步分析引擎使用。图原生存储引擎充分利用分布式扩展技术与高速内存，从底层为高速插件化查询提供存储基础。

数据管理层

数据管理层主要负责管理多个不同的图数据来源。这些来源包括现有的图数据库系统以及CSV格式存储的图文件，数据存储层可以实现统一的数据管理服务，并兼容不同的系统与图格式。

基于现有的图数据库系统进行数据管理，可以兼容现有的应用与服务，从而实现业务对九源智能推理图数据库的无缝切换。对CSV文件的支持主要是因为在智能推理应用的只读场景下，能够直接对CSV格式的图文件进行加载和查询，因此可以省略将CSV格式的文件导入其他系统的步骤。

平台层

九源智能推理图数据库是一款支持国产化硬件平台的系统软件，整个图数据库软件、数据管理层依赖的其他系统以及依赖的第三方库都对国产通用处理器和国产通用操作系统进行了适配。目前已经支持**银河麒麟**操作系统，以及**鲲鹏**、**飞腾**、海光三款国产处理器。

图数据库使用方式

支持 **Java** 和 **Python** 编写的客户端，用来操作九源智能推理图数据库，便于应用方接入。（后续也可以根据需求支持其他语言）。以下示例中使用的接口为初版设计，具体发布后的产品接口可能会存在轻微差别，但整体思路相同，具体以版本发布后的接口为准。

快速上手

以社交网络图为例，包含 Person 和 Knows 标签，来描述九源智能推理图数据库的使用方法。

Schema操作

以下是连接图数据库、创建图、创建 schema、导入数据的具体使用方式。

```
# 指定网络地址连接图数据库
JiuyuanGraphClient gClient = JiuyuanGraph.newClient("127.0.0.1:9901");

# 创建 social_network 图，如果存在，就获取当前图
Graph graph= gClient.create_graph("social_network", if_not_exists = true);

# 创建点标签 Person
Result ret = graph.create_label("Person");

# 创建边标签 Knows
Result ret = graph.create_edge("Knows");

# 从 csv 文件导入 Person 数据
Result ret = graph.load_vertices("/data/vertex/person.csv", "Person");

# 从 csv 文件导入 Knows 数据
Result ret = graph.load_edges("/data/edge/knows.csv", "Knows");
```

交互式查询

以下是使用交互式查询的方式，查询 id 为 1 的 Person 点的所有一度邻居。九源智能推理图数据库的查询语言目前兼容 AGE 系统支持的查询语言，主要为在 SQL 中包含 Cypher 语言来进行查询，Cypher 查询兼容 OpenCypher 的标准。

参考链接：https://age.apache.org/age-manual/master/clauses/match.html

```
DataSet result = graph.interactive("MATCH(a:Person {id: "1"})-[b:Knows]->
```

```
(c:Person) RETURN c;");
```

插件化查询

以下是高速插件化查询的方式。通过 Graph 抽象的 project 方法，抽取一个子图到内存中以支持高速插件化查询。

project 方法用于从 Graph 抽象中抽取特定的子图为 MemoryGraph 抽象，用以支持高速插件化查询的需求。主要的接口如下：

```
MemoryGraph mGraph = graph.project(filters, addresses, partition_num);
DataSet result = mGraph.execute(plugin_name, parameters);
```

其中 filters 中指定抽取的点边标签与对应的过滤条件，具体结构如下：

```
class Filter {
    vlabels: Vector<String>,
    vfilters: Vector<String>,
    elabels: Vector<String>,
    efilters: Vector<String>,
};
```

此外，project 接口中还需要指定插件化执行监听的地址（可以有多个），可以扩展到分布式运行，并且需要指定每个节点上的分片数量。

以下是一个使用插件化查询接口的具体示例：

```
MemoryGraph mGraph = graph.project(Filter {["Person"], ["Person.age > 30"],
["Knows"], [""]}, ["127.0.0.1:9001"], 32);

# 运行插件化的一度邻居查询，并指定起点
DataSet result = mGraph.execute("1-step-neighbors", {"id": "1"});

# 运行插件化的单源最短路，并指定起点
DataSet result = mGraph.execute("SSSP", {"id": "1"});

# 销毁这个子图
Result ret = mGraph.drop();
```

高速插件接口介绍

如果通用的交互式查询无法满足用户需求，也可以通过使用高速插件化查询来定制查询任务的实现。对于图查询任务，需要基于 Rust 语言调用高速插件化查询模块的接口，编写对应的查询实现。对于图分析任务，需要基于 C++ 语言调用异步分析引擎的接口，编写对应的分析任务实现。

为了方便用户的使用，本系统已经支持了多个插件化查询与分析任务，用户可以直接调用使用。如果已支持的插件仍然无法满足需求，那么需要自行编写或联系九源智能推理图数据库开发团队提供支持。

查询任务插件化编写方式

基于 Rust 语言，调用提供的存储接口以及消息传输接口，就可以实现分布式高并发查询任务。

下面以一个简单的一度邻居为例：

```
MATCH (n:Person {id: $personId })-[r:KNOWS]-(friend)
RETURN
    friend.id AS personId,
    friend.firstName AS firstName,
    friend.lastName AS lastName,
    r.creationDate AS friendshipCreationDate
ORDER BY
    friendshipCreationDate DESC,
    toInteger(personId) ASC
```

上述 Cypher 查询可以用以下的插件程序来高效实现。其中的核心接口在于：

- Memo的使用
    - 可以在每个分片所在的内存区域中缓存信息
    - self.memo().resume，汇总消息，确定查询任务是否结束
- 存储接口的使用
    - self.get_vertex_idx，获取当前顶点在本模块中的局部编号
    - self.with_edges，获取指定顶点的邻边信息
- 通信接口的使用
    - self.spawn_to，跳转到对应顶点所在的分片去执行查询任务

```
pub struct Result {
    person_id: u64,
    person_first_name: String,
    person_last_name: String,
    friendship_creation_date: i64,
}

impl PartialOrd for Result {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(other))
    }
}

impl Ord for Result {
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {
        match self
            .friendship_creation_date
            .cmp(&other.friendship_creation_date)
        {
            std::cmp::Ordering::Equal => {}
            ord => return ord.reverse(),
        }
        self.person_id.cmp(&other.person_id)
    }
}

struct Memo {
    results: Vec<Result>,
    responder: Responder,
}

impl Memo {
    pub fn new(responder: Responder) -> Self {
        Self {
            results: Vec::new(),
            responder,
        }
    }
}

impl Extend<Option<Result>> for Memo {
    fn extend<T: IntoIterator<Item = Option<Result>>>(&mut self, iter: T) {
        self.results
            .extend(iter.into_iter().filter_map(|elem| elem));
    }
}
```

```
impl Drop for Memo {
    fn drop(&mut self) {
        self.results.sort_unstable();
        self.responder
            .ok(serde_json::to_string(&self.results).unwrap());
    }
}

#[procedure]
fn one_step(personId: u64) {
    let query_uniq_id = self.alloc_uniq_id();
    let responder = self.responder.take();
    self.memo()
        .put(query_uniq_id, Weighted::new(Memo::new(responder)));

    let ts = self.get_read_ts();
    let start = self.get_vertex_idx(Person::TYPE_ID, personId).unwrap();
    Core(&self, ts, start, query_uniq_id);
    self.memo()
        .resume::<Weighted<Memo>, _>(query_uniq_id, (Default::default(),
None));
}

fn Core(cx: &Context, ts: Timestamp, start: usize, query_uniq_id: UniqId) {
    let func = |edges: EdgesIter| {
        edges
            .map(|e| {
                let knows = unsafe { knows::convert(e.data) };
                (*e.dest_partition, *e.dest_idx, knows.creationDate)
            })
            .collect::<Vec<_>>()
    };
    let out_edges = cx.with_edges(ts, Person::TYPE_ID, start,
knows::SRC_EDGELIST_IDX, func);
    let in_edges = cx.with_edges(ts, Person::TYPE_ID, start,
knows::DST_EDGELIST_IDX, func);

    let memo: Rc<Weighted<Memo>> = cx.memo().get(query_uniq_id).unwrap();
    for (dest_partition, dest_id, creation_date) in
        out_edges.into_iter().chain(in_edges.into_iter())
    {
        cx.spawn_to(
            FindFriends(
                ts,
                creation_date,
                dest_id,
                cx.local_id(),
```

```
                    query_uniq_id,
                    memo.weight.clone(),
                ),
                dest_partition,
            );
        }
    }

    #[remote]
    fn FindFriends(
        ts: Timestamp,
        creation_date: i64,
        id: usize,
        master_partition: usize,
        query_uniq_id: UniqId,
        weight: Weight,
    ) {
        let person: Vertex<Person> = self.get_vertex(ts, id).unwrap();
        self.spawn_to(
            FillResult(
                Some(Result {
                    person_id: person.id,
                    person_first_name: self.get_string(&person.firstName),
                    person_last_name: self.get_string(&person.lastName),
                    friendship_creation_date: creation_date,
                }),
                query_uniq_id,
                weight,
            ),
            master_partition,
        );
    }

    #[remote]
    fn FillResult(res: Option<Result>, query_uniq_id: UniqId, weight: Weight) {
        self.memo()
            .resume::<Weighted<Memo>, _>(query_uniq_id, (weight, res));
    }
```

分析任务插件化编写方式

基于 C++ 语言，需要将算法表示为迭代处理的方式，主要编写四个函数，分别是：

- init_func
  - 初始化所有顶点状态的函数
- signal_func

- 每一轮中处理顶点状态生成消息的函数
- slot_func
  - 每一轮中汇总更新消息的函数
- update_func
  - 将汇总后的结果作用到顶点状态的函数

此外，还需要调用到存储接口来获取特定顶点的邻边（get_out_degree_by_vid，get_out_edge_by_vid），以及调用 process_vertices 和 process_edges 来进行点和边的处理。其中对点的处理包括算法开始前的初始化、每一轮迭代中生成消息、每一轮结束后将结果作用于顶点状态，对边的处理包括消息聚合、消息传输、消息汇总。用户在使用这些接口时不需要了解内部的具体实现，只需要按照示例提供对应的上述四个函数与参数即可。

```
folly::coro::Task<void> PageRankCore(
    Distributed_Executor<VertexWeight, EdgeWeight, VertexValue>* executor) {
    double d = 0.85;
    // 创建初始化函数，对每个已激活的顶点进行初始化
    auto init_func =
        [](IntID lid,
           Distributed_Executor<VertexWeight, EdgeWeight, VertexValue>*
executor) -> IntID {
        executor->vertex_value_in_[lid] = 1;  // 将本轮迭代中的顶点值设置为1
        executor->vertex_value_out_[lid] = 0; // 下一轮迭代的值设置为0
        // 返回已激活顶点的出边数量（用于统计当前系统中的活跃边数量）
        return executor->graph_handler_->get_out_degree_by_vid(lid);
    };

    // 发送端生成消息的函数
    // 传入一个本地点id与一个消息buffer
    auto signal_func =
        [](IntID lid, std::vector<std::vector<MessageUnitWithIntId<double>>>&
message_buffer,
           Distributed_Executor<VertexWeight, EdgeWeight, VertexValue>*
executor) {
            // 先获取该点的出度
            auto degree = executor->graph_handler_->get_out_degree_by_vid(lid);
            // 获取该点当前的 Rank 值
            double message = executor->vertex_value_in_[lid];
            if (degree > 0) {
                message /= degree; // 当前Rank值除以度数为发往邻居节点的
            }
            // 获取当前节点的所有出边
            auto edges = executor->graph_handler_->get_out_edge_by_vid(lid);
            for (auto& e : edges) {
                // 往每个邻居顶点发送消息
```

```
                executor->emit_with_int_id(e.get_dst(), message,
message_buffer);
            }
        };

    // 接收端处理消息的函数
    // 传入一个本地点id与接收到的消息值
    auto slot_func =
        [&](IntID lid, double message,
            Distributed_Executor<VertexWeight, EdgeWeight, VertexValue>*
executor) -> IntID {
        executor->vertex_value_out_[lid] += message;  // 将消息值增加到
vertex_value_out_ 上
        return 1;
    };

    // 每轮迭代最后更新处于激活状态的顶点Rank值
    auto update_func =
        [d](IntID lid,
            Distributed_Executor<VertexWeight, EdgeWeight, VertexValue>*
executor) -> IntID {
        // vertex_value_in_ = vertex_value_in_ * （1-d） + d * vertex_value_out_
        executor->vertex_value_in_[lid] = 1 - d;
        executor->vertex_value_in_[lid] += d * executor-
>vertex_value_out_[lid];
        executor->vertex_value_out_[lid] = 0; // 将 vertex_value_out_ 重置为 0
        // 返回每个点的出度，用于计算当前全局的活跃边数
        return executor->graph_handler_->get_out_degree_by_vid(lid);
    };

    using InitFuncType = decltype(init_func);
    using SignalFuncType = decltype(signal_func);
    using SlotFuncType = decltype(slot_func);
    using UpdateFuncType = decltype(update_func);

    IntID total_active_edges, active_edges;
    // 先将本轮所有顶点置为激活状态，然后调用 process_vertices去初始化所有顶点
    // 并获取当前节点的活跃边数量
    executor->active_in_->fill();
    active_edges = co_await executor->process_vertices<IntID, InitFuncType>
(std::move(init_func),

executor->active_in_);
    // 设置活跃顶点数量为本节点上所负责的所有顶点数量
    executor->active_vertices_ = executor->owned_vertices_;
    for (int iter = 0; iter < 20; iter++) {
        // 同步目前全局的活跃边数量
```

```
        if (executor->node_num_ == 1) {
            total_active_edges = active_edges;
        } else {
            total_active_edges = co_await executor->my_rpc_-
>co_synchronize(active_edges);
        }

        // 调用process_edges去计算本轮的结果
        executor->active_vertices_ = co_await executor->process_edges<
            double, IntID, SignalFuncType, SlotFuncType>
(std::move(signal_func), std::move(slot_func),
                                   executor->active_in_, total_active_edges);

        // 更新本轮结果并统计本轮结束后的活跃边数量
        active_edges = co_await executor->process_vertices<IntID,
UpdateFuncType>(
            std::move(update_func), executor->active_in_);
    }
    co_return;
}
```

实验结果

LDBC基准测试

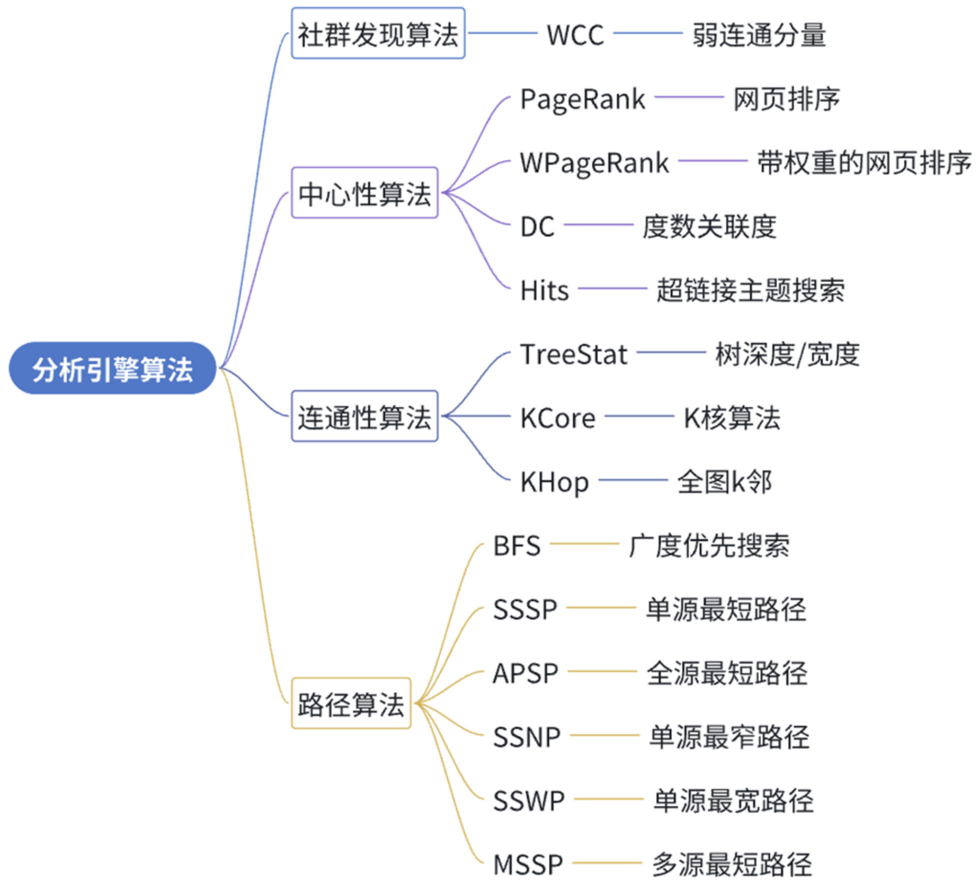
LDBC是图数据库领域权威的基准测试研发与标准制定机构，汇聚了包括Oracle、Intel、蚂蚁集团、AWS、TigerGraph等众多软硬件巨头和全球图数据库领域的专家学者，旨在共同推进图数据库前沿技术的发展。LDBC致力于制定公平、诚信、可对比的方法和机制来衡量各类图数据库系统的功能和性能。

目前LDBC榜单的前三位分别为 StarGraph 的 AtlasGraph，Alibaba DAMO Academy 的 GraphScope，Ant Group 的 TuGraph，具体榜单链接：https://ldbcouncil.org/benchmarks/snb-interactive/。九源智能推理图数据库目前也进行了LDBC基准测试的相关测试（官方认证正在筹备中），选取LDBC SF 100 数据集为例，目前与榜单前三位的性能对比如下：

图数据库系统	九源智能推理图数据库	AtlasGraph	GraphScope	TuGraph
吞吐量 ops/s	72802.94	48764.08	33625.36	16966.26

分析算法性能对比

九源智能推理图数据库目前支持了14种高速分析算法插件，包括：



目前分析算法的性能已经均全面超过国际领先产品 Neo4j。下面是在相同的硬件环境（x86平台，4并发）和数据集（点41652230，边1468365182）下与 Neo4j 进行的分析算法性能对比测试结果（单位：秒）：

分析算法	Neo4j	九源智能推理图数据库
WCC	75	73.6
PageRank	555.5	464
WPageRank	806.8	527.5
DC	30.2	0.2
Hits	6708	979.8
TreeStat	不支持	75.9
KCore	187.9	4

KHop	321.6	31.2
BFS	71.8	41.8
SSSP	187.2	75.6
APSP	449.4	375.5
SSNP	不支持	216.9
SSWP	不支持	213.8
MSSP	不支持	375.3