分区环境搭建过程:

分区使用了分区表的这一个概念，对于一个大的字表我们可以通过主键或者其它键进行Hash分桶来分区处理，本实验采用主键等分的处理，如果想使用其它分区方式请参考这个网站https://docs-cn.greenplum.org/v5/admin_guide/ddl/ddl-partition.html
具体实验中的处理如下：(见 rl_index_selection/index_selection_evaluation/selection/table_generator.py)

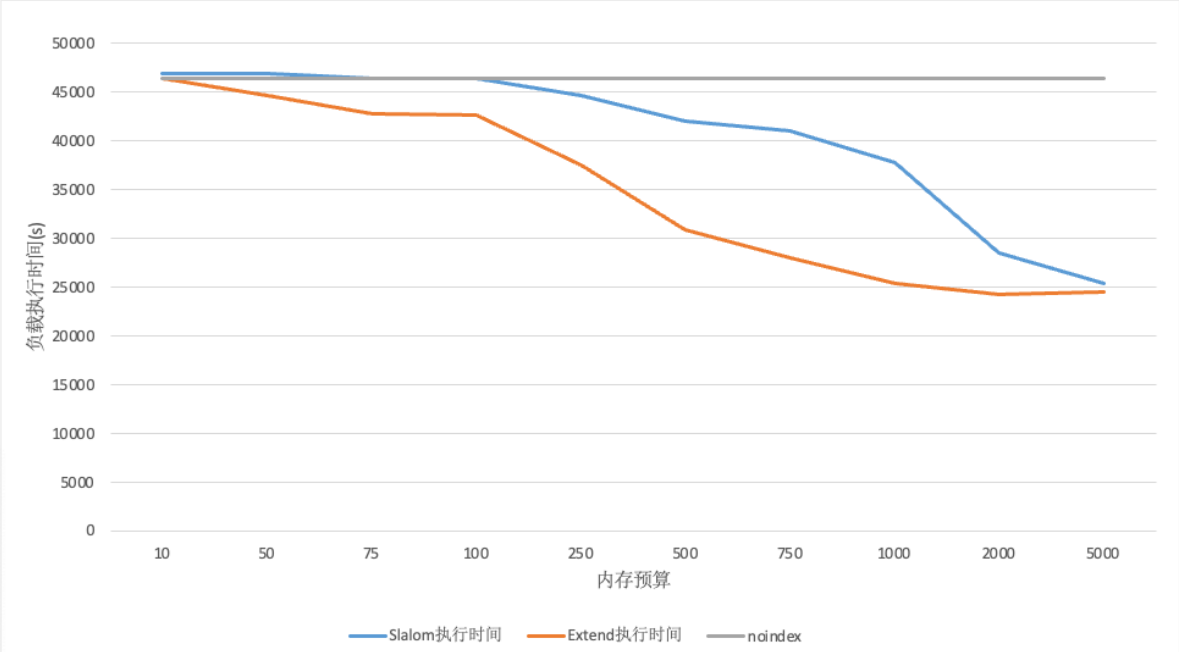
```
def create_partitions(self,partition_num=16):
        logging.info("Creating partition tables")
        tables = {
            'customer': 'c_custkey',
            'lineitem': 'l_orderkey',
            'nation': 'n_nationkey',
            'orders': 'o_orderkey',
            'part': 'p_partkey',
            'partsupp': 'ps_partkey',
            'region': 'r_regionkey',
            'supplier': 's_suppkey'
        }
        # create partition tables
        for key, value in tables.items():
            # partitioning tables
            statements = []
            statements.append(f"ALTER TABLE {key} RENAME TO
{key}_bak;")
            statement = f"create table {key} (LIKE {key}_bak) WITH
(appendonly=true, orientation=column) DISTRIBUTED BY ({value})
PARTITION BY RANGE({value})("
            s_max_value = f"select max({value}) from {key}"
            s_min_value = f"select min({value}) from {key}"
            max_value = self.db_connector.exec_fetch(s_max_value)
            min_value = self.db_connector.exec_fetch(s_min_value)
            value_range = list(np.linspace(min_value, max_value,
partition_num+1))
            value_range = [math.ceil(v) for v in value_range]
```

```
            # deal with the range cannot be divided into
partition_num fields
            value_range = list(set(value_range))
            while len(value_range) < partition_num+1:
                value_range.append(value_range[-1]+1)
            value_range.sort()
            for i in range(len(value_range) - 1):
                if i+1 == len(value_range)-1:
                    op = 'inclusive'
                else:
                    op = 'exclusive,'
                statement += f"PARTITION p{i} start
({value_range[i]})inclusive end ({value_range[i+1]}) {op}"
                # statement = f"create table {key}_{i} as select *
from {key} where {value} >= {value_range[i]} and {value} {op}
{value_range[i+1]}"
                # self.db_connector.exec_only(s)
            statement += ')'
            statements.append(statement)
            statements.append(f"INSERT INTO {key} SELECT * FROM
{key}_bak;")
            for s in statements:
                self.db_connector.exec_only(s)
        self.db_connector.commit()
```

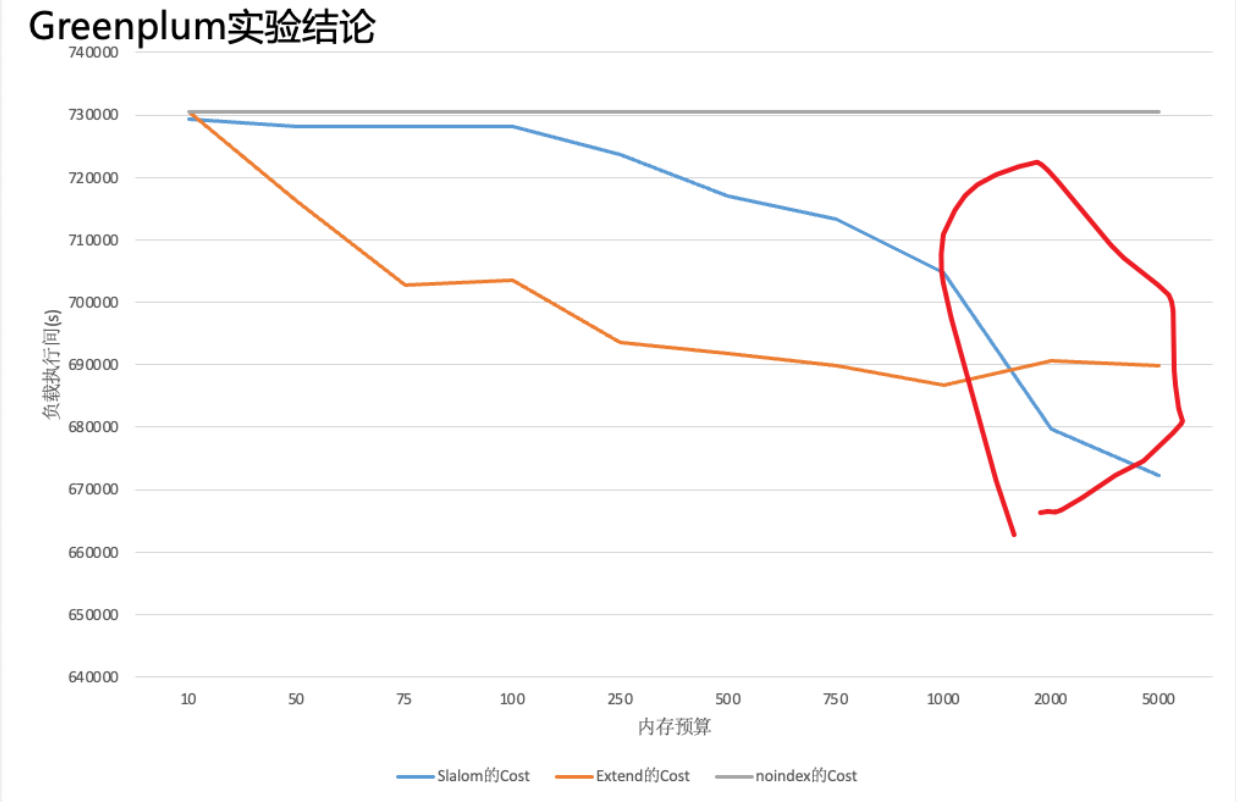
注意：所有对于greenplum数据库、数据表的处理我们都是通过psycopg2库来将sql语句通过python语言写到数据库中的。

what-if&虚拟索引功能

Greenplum本身支持whatif相关功能，但是其预测结果并不完全符合实际。



推荐的索引效果 = 无索引的负载执行时间 – 算法推荐的索引负载执行时间
Extend相比于Slalom的推荐索引效果提升 = 240%
执行时间提升 = (Slalom的执行时间 – Extend的执行时间) / Slalom的执行时间
Extend相比于Slalom的执行时间提升=30%



下面是优化器的预测结果，上面是真实的结果，可以看到在大体趋势是可以模拟出来的，而且随着元数据存储容量的增加，两者结果都呈现执行时间减少的趋势,但是在元数据存储预算过量的时候优化器的模拟结果出现了偏差。不过其用来应对索引推荐的时候可以探索出哪种更好（如图中真实结果和whatif的大体趋势类似可以体现出来）是可以使用的（✅）

Slalom

具体代码见index_selection_evaluation/selection/algorithms/slalom_algorithm.py,其调用和使用如下：
（index_selection_evaluation/selection/index_selection_evaluation.py），首先需要创建一个algorithm实例，然后将负载作为参数调用calculate_best_indexes函数得到的便是其推荐出的索引组合

```
def _run_algorithm(self, config):
        self.db_connector.drop_indexes()
        self.db_connector.commit()
        self.setup_db_connector(self.database_name,
self.database_system)
        indexes = []
        for partition_id in range(config["parameters"]
["partition_num"]):
            parameters = copy.deepcopy(config["parameters"])
            if config["name"] in BLOCK_AWARED_ALGORITHOM:
                algorithm =
self.create_algorithm_object(config["name"], parameters)
                logging.info(f"Running algorithm {config}")
                indexes +=
algorithm.calculate_best_indexes(self.all_prt_workloads,self.workload
s)
                break
            elif config["name"] in SINGLE_COLUMN_ALGORITHOM:
                algorithm =
self.create_algorithm_object(config["name"], parameters)
                logging.info(f"Running algorithm {config}")
                indexes +=
algorithm.calculate_best_indexes(self.total_workloads)
                break
            else:
```

```
                # mean splite budget for each partitions
                if "budget_MB" in parameters:
                    parameters["budget_MB"] =
parameters["budget_MB"]/config["parameters"]["partition_num"]
                algorithm =
self.create_algorithm_object(config["name"], parameters)
                logging.info(f"Running algorithm {config}")
                indexes +=
algorithm.calculate_best_indexes(self.workloads[partition_id])
```

Extend

承接上文，extend的调用和Slalom算法调用是一致的不过需要注意的是负载的输入：

all_prt_workloads：表示将负载拆成partition_num份，它们组成的一个合集。

```
举例：
对于Sql语句：SELECT * FROM LINEITEM WHERE l_orderkey <100;
all_prt_workloads是：
        SELECT * FROM LINEITEM_1_prt_0 WHERE l_orderkey <100;(对于分区
表的访问)
        SELECT * FROM LINEITEM_1_prt_1 WHERE l_orderkey <100;
        SELECT * FROM LINEITEM_1_prt_2 WHERE l_orderkey <100;
        SELECT * FROM LINEITEM_1_prt_3 WHERE l_orderkey <100;
        SELECT * FROM LINEITEM_1_prt_4 WHERE l_orderkey <100;
        SELECT * FROM LINEITEM_1_prt_5 WHERE l_orderkey <100;
        ....
```

self.workload：是将all_prt_workloads进行分区打包，它是一个list,使用self.workloads[2]'，即表示将'SELECT * FROM LINEITEM_1_prt_2 WHERE l_orderkey <100;'取出来

1. 要想“extend每次推荐的索引会在所有列上创建”那么在调用extend算法的时候需要使用如下的用法：

```
 for partition_id in range(config["parameters"]
["partition_num"]):
            parameters = copy.deepcopy(config["parameters"])
            if config["name"] in BLOCK_AWARED_ALGORITHOM:
#BLOCK_AWARED_ALGORITHOM是一个字典，我们需要把extend算法放到这个字典
里面
                algorithm =
self.create_algorithm_object(config["name"], parameters)
                logging.info(f"Running algorithm {config}")
                indexes +=
algorithm.calculate_best_indexes(self.all_prt_workloads,self.w
orkloads)
                break
```

2. 要想“使用extend并将预算进行均分的策略”，那么在调用extend算法的时候需要使用如下的用法:

```
for partition_id in range(config["parameters"]
["partition_num"]):
# mean splite budget for each partitions
    if "budget_MB" in parameters:
        parameters["budget_MB"] =
parameters["budget_MB"]/config["parameters"]["partition_num"]
    algorithm = self.create_algorithm_object(config["name"],
parameters)
    indexes +=
algorithm.calculate_best_indexes(self.workloads[partition_id])
```

注意：和强化学习对比的时候需要将这两个算法放到rl_index_selection目录下面的相关文件中，那里是处理TPCH负载的对比环境，具体调用方式类似:
(rl_index_selection/swirl/experiment.py)

```
def _compare_extend(self):
        self.evaluated_workloads = set()
        for model_performances_outer, run_type in
[self.test_model(self.model), self.validate_model(self.model)]:
            for model_performances, _, _ in model_performances_outer:
```

```
                self.comparison_performances[run_type]
["Extend"].append([])
                for model_performance in model_performances:
                    assert (

model_performance["evaluated_workload"].budget ==
model_performance["available_budget"]
                    ), "Budget mismatch!"
                    assert model_performance["evaluated_workload"]
not in self.evaluated_workloads

self.evaluated_workloads.add(model_performance["evaluated_workload"])

                    parameters = {
                        "budget_MB":
model_performance["evaluated_workload"].budget,
                        "max_index_width":
self.config["max_index_width"],
                        "min_cost_improvement": 1.003,
                    }
                    extend_connector =
PostgresDatabaseConnector(self.schema.database_name, autocommit=True)
                    extend_connector.drop_indexes()
                    extend_algorithm =
ExtendAlgorithm(extend_connector, parameters)
                    indexes =
extend_algorithm.calculate_best_indexes(model_performance["evaluated_
workload"])
                    self.comparison_indexes["Extend"] |=
frozenset(indexes)

                    self.comparison_performances[run_type]["Extend"]
[-1].append(extend_algorithm.final_cost_proportion)
```

TPCH负载的处理

对于TPCH负载处理的主要问题是Join，Join的过程中各分区需要全局的一个信息从而保留该分区中能被join的records

1. 修改sql进行模拟，让每个分区都保留有需要被全局join的records,分析见文件中的ppt，下面是对于sql模拟的实例&验证：
   假设只有两个分区

   对于sql语句

```
SELECT c_custkey, o_custkey, o_orderkey, l_orderkey,
l_shipdate
FROM customer, orders, lineitem
WHERE (c_custkey = o_custkey AND l_orderkey = o_orderkey AND
l_shipdate between date '1995-01-01' and date '1995-01-01')
```

   其查询结果如下

c_custkey	o_custkey	o_orderkey	l_orderkey	l_shipdate
40	40	38055	38055	1995-01-01
106	106	29634	29634	1995-01-01
250	250	20263	20263	1995-01-01
1144	1144	26432	26432	1995-01-01
1177	1177	4550	4550	1995-01-01
1217	1217	11520	11520	1995-01-01
1256	1256	8164	8164	1995-01-01
1279	1279	39620	39620	1995-01-01
94	94	50656	50656	1995-01-01
187	187	24165	24165	1995-01-01
404	404	26565	26565	1995-01-01
421	421	49731	49731	1995-01-01
446	446	36416	36416	1995-01-01
452	452	38915	38915	1995-01-01
505	505	646	646	1995-01-01
613	613	37507	37507	1995-01-01
622	622	50277	50277	1995-01-01
664	664	35107	35107	1995-01-01
892	892	15748	15748	1995-01-01
1063	1063	37473	37473	1995-01-01

(20 rows)

   对于分区Block0，我们需要执行一下语句：

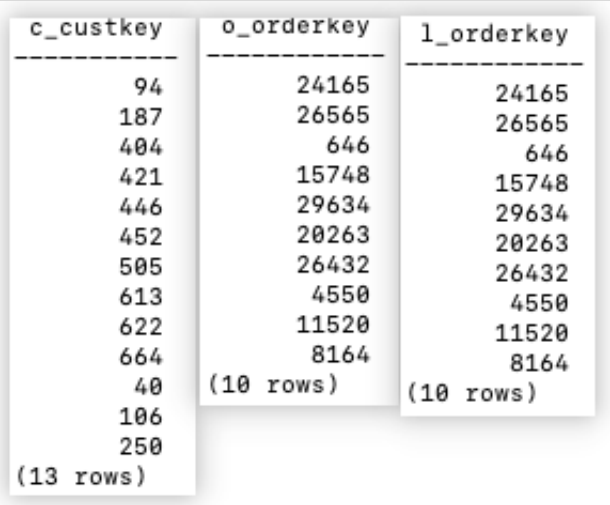
```
# 保留c表中需要join的records
SELECT c_0.c_custkey
FROM orders, lineitem, customer_1_prt_p0 as c_0
WHERE (c_0.c_custkey = orders.o_custkey AND
lineitem.l_orderkey = orders.o_orderkey AND
lineitem.l_shipdate between date '1995-01-01' and date '1995-
01-01');
# 保留o表中需要join的records
SELECT o_0.o_orderkey
FROM lineitem, customer,orders_1_prt_p0 as o_0
WHERE (customer.c_custkey = o_0.o_custkey AND
lineitem.l_orderkey = o_0.o_orderkey AND lineitem.l_shipdate
between date '1995-01-01' and date '1995-01-01');
# 保留l表中需要join的records
SELECT l_0.l_orderkey
FROM orders, customer,lineitem_1_prt_p0 as l_0
WHERE (customer.c_custkey = orders.o_custkey AND
l_0.l_orderkey = orders.o_orderkey AND l_0.l_shipdate between
date '1995-01-01' and date '1995-01-01');
```

对于分区Block1，我们需要执行一下语句:

```
# 保留c表中需要join的records
SELECT c_1.c_custkey
FROM orders, lineitem, customer_1_prt_p1 as c_1
WHERE (c_1.c_custkey = orders.o_custkey AND
lineitem.l_orderkey = orders.o_orderkey AND
lineitem.l_shipdate between date '1995-01-01' and date '1995-
01-01');
# 保留o表中需要join的records
SELECT o_1.o_orderkey
FROM lineitem, customer,orders_1_prt_p1 as o_1
WHERE (customer.c_custkey = o_1.o_custkey AND
lineitem.l_orderkey = o_1.o_orderkey AND lineitem.l_shipdate
between date '1995-01-01' and date '1995-01-01');
# 保留l表中需要join的records
SELECT l_1.l_orderkey
FROM orders, customer,lineitem_1_prt_p1 as l_1
WHERE (customer.c_custkey = orders.o_custkey AND
l_1.l_orderkey = orders.o_orderkey AND l_1.l_shipdate between
date '1995-01-01' and date '1995-01-01');
```

它们的执行结果如下所示:

BLOCK1



c_custkey
94
187
404
421
446
452
505
613
622
664
40
106
250

(13 rows)

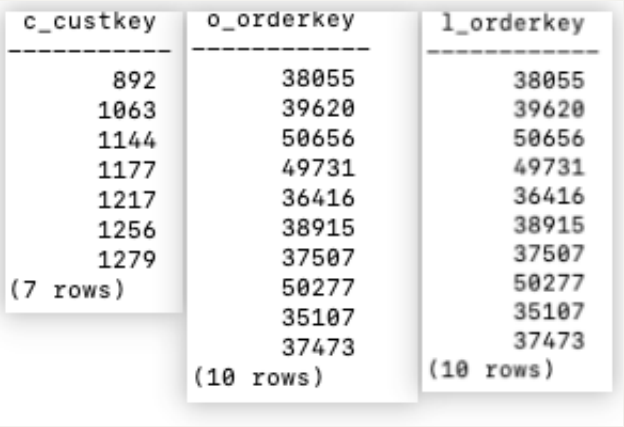
o_orderkey
24165
26565
646
15748
29634
20263
26432
4550
11520
8164

(10 rows)

l_orderkey
24165
26565
646
15748
29634
20263
26432
4550
11520
8164

(10 rows)

BLOCK2



c_custkey
892
1063
1144
1177
1217
1256
1279

(7 rows)

o_orderkey
38055
39620
50656
49731
36416
38915
37507
50277
35107
37473

(10 rows)

l_orderkey
38055
39620
50656
49731
36416
38915
37507
50277
35107
37473

(10 rows)

可以看到两个分区的结果合并起来就是大表join的正确结果。
至于时间问题的验证，采用这种修改sql语句的方式其实是用来模拟子节点Join的操作，它们的时间总和会大于全表的join操作,但是由于每一种索引推荐算法都是使用的同一套负载，所以使用模拟的方式来实现join是一种可以考虑的方案。

2. 除此之外还有一种方案称为“跨库join"这个需要用到数据库中间件，需要做调研，参考文章https://juejin.cn/post/6982460392077262885#heading-11。

action masking

- 这部分的功能主要是将swirl的action masking进行扩展为我们的多动作环境所用，其具体内容在BlockAwaredMultiColumnIndexActionManager类中的四个类函数。
- 如何将这部分从sb2移植到sb3的一些我调研出来的建议：
    a. 首先对于action_mask的使用，可以参考官网文档https://sb3-contrib.readthedocs.io/en/master/modules/ppo_mask.html

```
env = InvalidActionEnvDiscrete(dim=80,
n_invalid_actions=60)
```

```
model = MaskablePPO("MlpPolicy", env, gamma=0.4,
seed=32, verbose=1)
model.learn(5_000)
evaluate_policy(model, env, n_eval_episodes=20,
reward_threshold=90, warn=False)

model.save("ppo_mask")
del model # remove to demonstrate saving and loading

model = MaskablePPO.load("ppo_mask")

obs, _ = env.reset()
while True:
    # Retrieve current action mask
    action_masks = get_action_masks(env)
    action, _states = model.predict(obs,
action_masks=action_masks)
    obs, reward, terminated, truncated, info =
env.step(action)
```

其中这个get_action_masks函数是sb3自带的一个函数，其可以处理环境让其返回一个np.ndarray类型的数据，这里是需要使用sb3的写法来实现一个返回函数。对于我实现的这个版本可供使用的属性是在action_spaces在rl_index_selection/gym_db/envs/db_env_v3.py中的self.valid_total_actions和self.valid_actions变量，前者是一个`[index1,index2,index3,.....indexn]`的onehot编码0代表无效，1代表有效；后者是一个按照块进行分割后的二维list，即

```
[
  [block1_index1,block1_index2,block1_index3,....],
  [block2_index1,block2_index2,block1_index3,....],
  [block2_index1,block2_index2,block1_index3,....],
  ...
]
```

根据所编写算法的需要选择需要的变量进行利用。

访问某一个分区的数据

```
访问父表全部数据：select * from customer
访问第x个分区的数据：select * from customer_1_prt_px
```

这里注意分区的后缀"_1_prt_p1"是自定义的可以去修改，我源码中实现的都是使用该名称(名称修改可以看分区环境搭建过程的代码)

如何使用env代码

这部分我带着锦慧配置和学习过,她应该可以上手很快

1. 配置相关环境（包括各种使用的包及数据库相关内容），
2. env：multdiscrete实现的action_spaces在rl_index_selection/gym_db/envs/db_env_v3.py中注意action_manager使用的是rl_index_selection/swirl/multagent_action_manager.py中的BlockAwaredMultiColumnIndexActionManager类
3. env的使用：可以直接参考swirl中的主函数

```
ParallelEnv = SubprocVecEnv if experiment.config["parallel_environments"] > 1 else DummyVecEnv

training_env = ParallelEnv(
    [experiment.make_env(env_id) for env_id in range(experiment.config["parallel_environments"])]
)
training_env = VecNormalize(
    training_env, norm_obs=True, norm_reward=True, gamma=experiment.config["rl_algorithm"]["gamma"], training=True
)

experiment.model_type = algorithm_class

with open(f"{experiment.experiment_folder_path}/experiment_object.pickle", "wb") as handle:
    pickle.dump(experiment, handle, protocol=pickle.HIGHEST_PROTOCOL)

model = algorithm_class(
    policy=experiment.config["rl_algorithm"]["policy"],
    env=training_env,
    verbose=2,
    seed=experiment.config["random_seed"],
    gamma=experiment.config["rl_algorithm"]["gamma"],
    tensorboard_log="tensor_log",
    policy_kwargs=copy.copy(
        experiment.config["rl_algorithm"]["model_architecture"]
    ),  # This is necessary because SB modifies the passed dict.
    **experiment.config["rl_algorithm"]["args"],
)
logging.warning(f"Creating model with NN architecture: {experiment.config['rl_algorithm']['model_architecture']}")

experiment.set_model(model)
experiment.compare()
```

这里需要注意的是，将算法部分替换成自己的模型。

4. 四条评价标准：

a. 基于gym实现✅
b. What if模拟结果精度能够近似greenplum效果（或相对差异保持一定）✅
c. 训练时间参考swirl （小时级别）✅：由于上面使用的是whatif方法，这部分主要耗费时间在虚拟索引的建立，以及不停地对whatif进行调用
d. 100G数据集，目前测试环境条件有限需要自行测试。

词袋模型的修改

SWIRL源代码的设计其针对于postgresql的执行计划，但是Greenplum和Postgresql的查询计划不一致：

PostgreSQL和Greenplum是两种不同的数据库系统，它们在查询优化和执行上有一些区别，因此它们生成的查询计划也有所不同。

1. PostgreSQL是一个关系型数据库系统，它使用基于成本的查询优化器来生成查询计划。成本估算器使用表、索引和统计信息来计算每个操作的成本，并选择最佳的执行计划。PostgreSQL的执行计划通常是基于行的，这意味着它将以行为单位处理数据。在查询计划中，它通常使用 Seq Scan、Index Scan、Hash Join、Nested Loop Join、Sort、Aggregate等操作符。
2. Greenplum是一个基于PostgreSQL的大数据分布式数据库系统，它使用基于代价的查询优化器来生成查询计划。Greenplum的查询优化器在计算成本时，会考虑数据分布、数据倾斜、网络传输等因素。Greenplum的执行计划通常是基于片段的，这意味着它将以数据分片为单位处理数据。在查询计划中，它通常使用Gather Motion、Redistribute Motion、Broadcast Motion、Hash Join、Merge Join、Sort、Aggregate等操作符。

所以我添加和修改了词袋模型的关键字进行处理，我目前是将用到的关键字都进行处理了，这部分需要根据需求自行挑选保留。

具体代码位置：rl_index_selection/swirl/boo.py