Special Topics:
Self-Driving Database Management Systems
P2: Pilot Control Plane

@Kushagra Singh
@Tim Lee
Pilot Control Plane

- Orchestrates the tuning life cycle. Build communication channels between Pilot and other services (DBMS instances, ML components).
- Some examples:
  - Capture and archiving data from PG instances
  - Pipelining ML models training & inference
Architecture

• Control plane
  • Spawn primary worker and exploratory worker
  • Issue commands to workers
  • Collect and archive data
  • Handle tuning life cycle; maintain states
**Architecture**

- **Primary Worker**
  - Capture workload, transfer back to Control Plane
  - Apply recommended actions
Architecture

- Exploratory Worker
  - Start / stop exploratory PG instances
  - Take snapshot from replica (if specify)
  - Collect data (execution cost, catalog, etc.)
Sample Workflow

- Launch Exploratory Worker
  - Completed
- Launch Exploratory Postgres
- Collect Exploratory Catalog
- Collect OU Training Data
- Forecast Workload
- Train OU Models
- Generate Actions
- Recommend Actions
- PRIMARY WORKER
  - Execute commands
  - Metrics and logs
  - Send training data
- EXPLORATORY WORKER
  - Run experiments
  - Send training data
- CONTROL PLANE
  - API interface
  - Data Archive
  - User
  - TUNING REQUEST
  - PRIMARY
    - DB Replication
    - Execute commands
    - Monitor DB state
  - REPLICA
    - Run experiments and collect data
DAG Execution Engine

- An execution engine that can run DAGs consisting of customized commands
- More extensible than fixed workflow per endpoint
Exploratory Data Collector

- A generic interface to collect data on exploratory PG instance
- E.g. training data collection team can implement this base class to collect data with benchbase / sqlsmith

```python
class BaseDataCollector(ABC):
    def __init__(self, postgres_port, data_dir, config):
        
        postgres_port: port on which the exploratory postgres is running
        data_dir: directory where collected data will be stored
        config: any config to be passed to the DataCollector
        
        self.postgres_port = postgres_port
        self.data_dir = data_dir
        self.config = config

    @abstractmethod
    def setup(self):
        pass

    @abstractmethod
    def collect(self):
        pass

    @abstractmethod
    def cleanup(self):
        pass
```
What we completed

- Messaging infrastructure between 3 workers
- DAG Execution Engine
- Commands supported:
  - Launch Primary Worker
  - Launch Exploratory Worker
  - Capture + archive workload
  - Launch exploratory PG instance (can take snapshot from replica)
  - Stop exploratory PG instance
  - An extendible interface to collect training data
Demo
Original Timeline

Mar 17
• Server setup
• Control Plane messaging setup

Mar 24
• Worker (primary + exploratory) spawn setup

Mar 31
• Exploratory PG orchestration

Apr 7
• Workload capture (75%)
• Workload archival

Apr 14
• Training data execution and archival (100%)

Future
• train OU models (125%)
• Forecasting, action recommendation
Future Work

- OU model training
- Workload forecasting model training + inference
- Action generation + picking
- Run 3 workers on different node (Ansible)
- Support concurrent tuning sessions
- Web UI?
- TODO list: https://github.com/cmu-db/noisepage-control/issues/7