GA.R.B.A.G.E.

GlobAl Rule-Based
Action Generation & Enumeration

Why we need GARBAGE

Previous self-driving DBMS efforts focus on a **fixed** action space

- Fixed schema, limited actions to consider

The entire action space that a self-driving DBMS could explore is too large

- Similar issue in query optimization

Rule-based action enumeration defines the space **dynamically** based on parameters

- Who writes the rules?
- Who sets the parameters?

The Big Picture

DB State **S**:

- Tables, indexes
- Physical design
- Knobs

Behavior models:

- (**S**, **Query Plan**) ⇒ **Cost**

SQL Actions **A**:

- Change knobs
- Add/drop indexes
- Partitioning, MVs, etc.

Forecasted Workload:

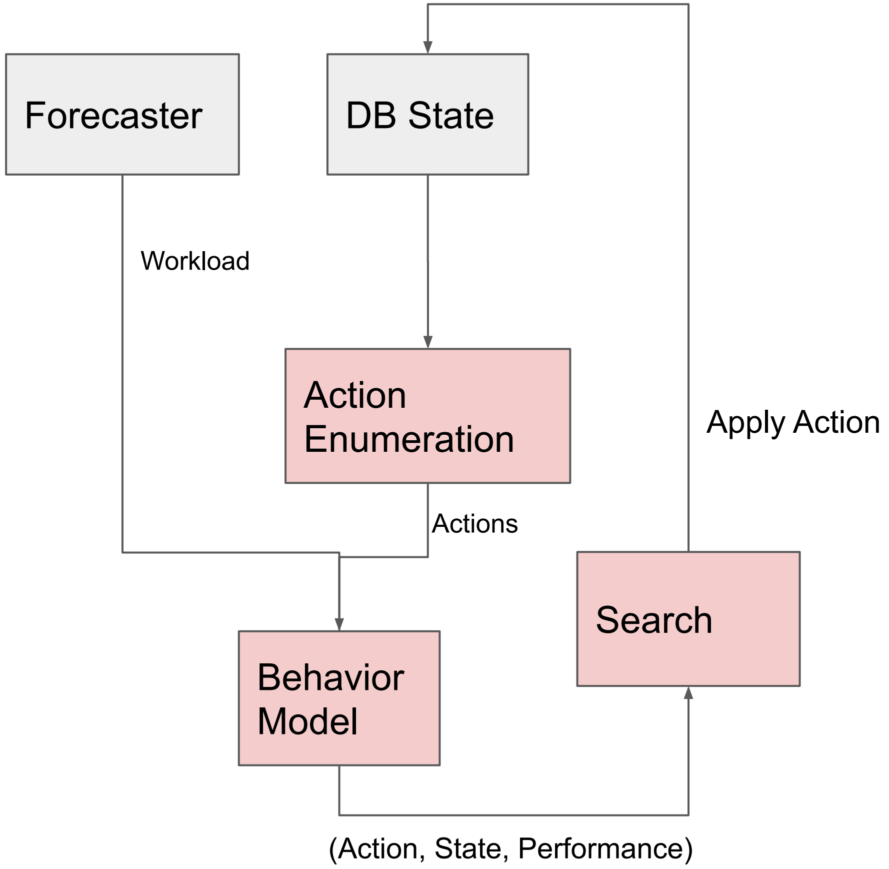
- {(**Query, Weight**), …}

Action Generation:

- **S** ⇒ {**A**, **A**, …}

Search:

- (**S**, {**A**, **A**, …}) ⇒ **A**



Big Questions

How should database state be represented?

- Access DB schema/knobs/physical design via a “DB connector”
- Integrate with control plane team
- Workload representation should not limit any new rules from being developed

How should actions be represented?

- Internally operate on custom objects, expose SQL string externally

How should rules be represented?

- User/Pilot should be able to “control” the size of the action space
- Rules should be configurable and potentially exhaustive

Design Decisions

Workload Representation

- How much should our representation summarize the workload?
  - Keep as much information as possible so any future rules can take advantage of the details
  - Built in methods for accessing some summaries - can change as needed

Knob Rule Representation

- Knob rules should be applicable for many different knobs instead of being used on a per-knob basis

Rule Involvement “Levels”

- Use & toggle pre-built rules
- Altering rule parameters
- Adding new rules

Implementation Overview

Inputs

- Database (connection string)
- Rule Configurations (yaml)
- Workload files (csv)

Outputs

- Currently: actions.sql
- Plan: np_generated_actions table in pilot

Configuration

Action generation is configured with generation rules:

```
- DropIndexGenerator:
    Name: generator1
    Indexes:
      - Name1
      - Name2
- WorkloadIndexGenerator:
    Name: generator2
    WorkloadFile: logs/epinions.csv
    MaxWidth: 5
- TypedIndexGenerator:
    Name: generator3
    Types:
      - "hash"
      - "brin"
    Upstream: generator2
```

```
- ExhaustiveIndexGenerator:
    Name: generator3
    MaxWidth: 10
- NumericalKnobGenerator:
    Name: generator5
    Knob: work_mem
    Type: PCT  # DELTA, ABSOLUTE, POW2
    Min: .1
    Max: 5
    Interval: .1
- CategoricalKnobGenerator:
    Name: generator6
    Knob: enable_seqscan
    Values:
      - "on"
      - "off"
```

These rules can be configured by either a user or the action selection process.

Implementation Overview

State Representation:

- Indexes, Tables, Schemas from DB connector (from control plane in the future)
- Parse logs to generate a workload object for templates / column references

Action Representation:

- Action object -> SQL
    - IndexAction attributes: table, cols, type, index_name
    - KnobAction attributes: name, val, restart_required

Generation Process

- Generator object -> action objects
    - Each generator is configured with associated parameters in a specified configuration
    - Some of generation rules can be composed upon other generators

Goals

75%:

✅ Initial version of state, action, and rule representation
✅ Focus on indexes (creating, dropping, index type)
📍 De-duplication if multiple rules generate the same action
✅ Configurations to turn rules on and off
📍 Schema in the pilot database

100%:

✅ Knob actions and generation rules (deltas, absolute, and categorical/flags)
📍 Consideration of mutually exclusive actions
✅ Additional configurability for rules (parameterization)
📍 Capturing additional aspects of “state”

125%:

📍 More sophisticated index actions (fill factor, clustering, e.g.)
📍 Rule-based lightweight scoring
📍 Action reversal
📍 Generate action representations systematically

Testing

- Correctness Unit Tests
  - Configuration rules sanity check
  - Output of individual rules
  - Validity and effect of action SQLs
  - Deduplication
- Integration
  - Configuration file generation
  - Joint debugging
- Performance
  - Initialization + workload processing time
  - Generation time
  - System metrics affected by running generator

Integration / Project 3

- Deduplication:
  - Different rules may end up generating the same actions, so we will deduplicate and attribute the action to the first generator in the config that produced the action via a Postgres object comment
- Schema acquisition + versioning:
  - We can re-run schema-dependent generation rules in the event of schema changes, with messages sent through the control plane
- DB state improvements:
  - There are salient aspects of the database we can capture and utilize for targeted rules (e.g. size and storage limits), we would want future rules to have access to this information
- Generator relationship:
  - Chaining generators into a pipelined workflow

Trade-offs

- Workload
  - Access methods are based on the needs of the generators we wrote
  - Potential bloat in the future as more rules are created
- DB State
  - As in workload
- Exhaustive vs. Selective
  - Exhaustive rules may generate too many options
  - Most selective configurations may leave too few
- Configurability vs. Intelligence
  - Rules leave significant configurability to a user (e.g. numerical action values, configurable action options) which can potentially be automated or managed intelligently

DEMO

Future Work

- Action reversal:
  - Each generated action can be associated with a corresponding “UNDO” action for reversal
- Lightweight scoring:
  - An ordering of actions may be valuable to the action selection process, and some lightweight scoring may optimize the sequence of actions considered
- “Vectorized” Action Space:
  - When integrating with future action-search/planning components, we could vectorized/one-hot encode the rule parameters
  - This would allow search/planning components to expand/restrict the search base based on e.g. available compute budget.
- Lazy Generation:
  - If we can associate actions with schemas/states in which they are legal, we don’t need to repeatedly output the same action each time it is prompted.