Redefine database architecture

1st Single-node DB
2nd Cluster DB
3rd Cloud DB
4th AI-native DB

1970s 1990s 2010s 2020s
Relation & SQL Large scale Internet scale Heterogenous SQL High performance High availability Scale-out Autonomous
An intelligent era calls for a more intelligent database
AI-Native Database

AI4DB

- Manual → Automatic
- Self-optimization
- Self-configuration
- Self-monitoring
- Self-healing
- Self-security
- Self-design

DB4AI

- AI → as easy as DB
- Declarative AI
- AI optimization
- Data governance
- Data provenance
- Model management
AI-Native Database

- Knob tuning
- Workload modeling
- Learned index optimizer
- Self assembling AI optimization
- Self design

- Index advisor
- Scheduling
- Declarative AI
- AI&DB Fusion
Level 1: AI-advised DB

- Database advisor for making database more intelligent
  - Database Configuration
    - Knob tuning
    - Workload management
    - Automatic Upgrade
  - Database Optimization
    - Index advisor
    - View advisor
    - Partition advisor
AI for Knob Tuning

- Automatic Tuning is important and challenging
  - Tunable options control nearly all aspects of runtime operations.
  - The number of knobs in a DBMS is huge and the relationships are complex.
CDBTune

<table>
<thead>
<tr>
<th>RL</th>
<th>CDBTune</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>The tuning system</td>
</tr>
<tr>
<td>Env</td>
<td>DB instance</td>
</tr>
<tr>
<td>State</td>
<td>Internal metrics</td>
</tr>
<tr>
<td>Reward</td>
<td>Performance change</td>
</tr>
<tr>
<td>Action</td>
<td>Knob configuration</td>
</tr>
<tr>
<td>Policy</td>
<td>Deep neural network</td>
</tr>
</tbody>
</table>

**Rewards**
- Throughput
- Latency
- SLAs

**States**
- Xact_commit/rollback
- Blk_reads/hit
- Tuple_fetched
- conflicts

**Environment**
- CDB

**Policy**
- CDBTune

**Agent**

**Action**
- Knobs

**Effective Cache Size**
- checkpoint_timeout
- io_concurrency
CDBTune

- using deep reinforcement learning (DRL), an end-to-end automatic CDB (Cloud Database) tuning system
  - deep deterministic policy gradient method (DDPG)
  - try-and-error strategy
- Characteristics:
  - end-to-end learning
  - using a limited number of samples
  - high-dimensional continuous knobs recommendation
  - reducing the possibility of Local Optimum
  - good adaptability
  - accelerates the convergence speed
CDBTune: Working Mechanism

**Offline Training**
- Step 1: builds a training model
- Step 2: trains the training model
  - Training Data
  - Training Model
  - Training Data Generation

**Online Tuning**
- Step 3: utilizes the model to recommend knob settings for an online tuning request
- Step 4: updates the training model by taking the tuning request as training data

Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. SIGMOD 2019
CDBTune

send request to the server through the local interface

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data

Client

send request to the server through the local interface

Controller

Training Request

Knob Settings

CDB

Tuning System

Workload Generator

Metrics Collector

Deep Reinforcement Learning

State

Reward

Environment

Agent

Action

Configurations

Metrics

Recommender

Memory Pool

Cloud

interacts information among the client, CDB and CDBTune

conducts stress testing

collects and processes related metrics

outputs the knob configurations

store processed data
Reinforcement Learning

- **Reinforcement Learning**
  - Method: DDPG
  - Goal: learn the best policy

- **Six key elements in RL**
  - **Agent**
    - receives reward and state, updates the policy
  - **Environment**
    - Environment is the tuning target, specifically an instance of CDB
  - **State**
    - the current state of the agent, i.e., the 63 metrics
    - describe the state at time $t$ as $s_t$
  - **Reward**
    - a scalar described as $r_t$
  - **Action**
    - described as $a_t$, corresponds to a knob tuning operation
  - **Policy**
    - described as $\mu(s_t)$
    - a mapping from state to action
About DDPG

- a policy-based method, combination of DQN and actor-critic
- learn the policy with high dimensional states and actions

DDPG design

- Policy function: \( a_t = \mu(s_t | \theta^\mu) \)
  - \( \theta^\mu \): mapping the state \( s_t \) to the value of action \( a_t \)
- Critic function: \( Q(s_t, a_t | \theta^Q) \)
  - represent the value (score) with specific action \( a_t \) and state \( s_t \)
  - \( \theta^Q \) is learnable parameters
- Inheriting from Bellman Equation and DQN: 
  \[ Q^\mu(s, a) = E_{r, s_{t+1} \sim E}[r(s_t, a_t) + y Q^\mu(s_{t+1}, \mu(s_{t+1}))] \]
  - policy \( \mu(s) \) is deterministic, \( s_{t+1} \) is the next state, \( r_t = r(s_t, a_t) \) is the reward function, and \( y \) is a discount factor
- Minimize the training objective: 
  \[ \min L(\theta^Q) = E[(Q(s, a | \theta^Q) - y)^2] \]
  - where \( y = r(s_t, a_t) + y Q^\mu(s_{t+1}, \mu(s_{t+1}) | \theta^Q) \)
Reward Function

About Reward Function
- feedback information between the agent and environment
- guides the agent to learn by telling what behavior is right or wrong

The design of the reward function
- $r$, $T$ and $L$ denote reward, throughput and latency
- 1. At time $t$, calculate the rate of performance change $\Delta$ from time $t-1$ and the initial time to time $t$ respectively.
- 2. Reward function: use $r$ to denote the sum of rewards of throughput and latency:

$$r = c_T \cdot r_T + c_L \cdot r_L$$

- $r_T$: the reward of throughput
- $r_L$: the reward of latency
- $r$: the sum of rewards of throughput and latency
- $c_T$ and $c_L$ are different coefficients
Table 3: Higher throughput (T) and lower latency (L) of CDBTune than BestCon

We observe that the best tuning candidates, especially gains a remarkable improvement unilaterally in data. Compared with BestCon in RL instead of massive high-quality DBA’s experience tuning, OtterTune performs inferior to the DBA in most cases. This is because we use the try-and-error samples as the result. Besides, OtterTune performs inferior to the DBA in more effective and our algorithm obtains the state-of-the-art method on different workloads, which improve the CPU percentages shown in Figure 9. The tuning performance improvement percentage is shown in Table 3 compared with BestCon.

Figure 8: Performance by increasing number of knobs

5.2.3 Performance improvement. We also evaluate our WO and RW (99%-tile Latency)

Results

<table>
<thead>
<tr>
<th>Workload</th>
<th>BestConfig</th>
<th></th>
<th>DBA</th>
<th></th>
<th>OtterTune</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>L</td>
<td>T</td>
<td>L</td>
<td>T</td>
<td>L</td>
</tr>
<tr>
<td>RW</td>
<td>↑ 68.28%</td>
<td>↓ 51.65%</td>
<td>↑ 4.48%</td>
<td>↓ 8.91%</td>
<td>↑ 29.80%</td>
<td>↓ 35.51%</td>
</tr>
<tr>
<td>RO</td>
<td>↑ 42.15%</td>
<td>↓ 43.95%</td>
<td>↑ 4.73%</td>
<td>↓ 11.66%</td>
<td>↑ 44.66%</td>
<td>↓ 23.63%</td>
</tr>
<tr>
<td>WO</td>
<td>↑ 128.66%</td>
<td>↓ 61.35%</td>
<td>↑ 46.57%</td>
<td>↓ 43.33%</td>
<td>↑ 91.25%</td>
<td>↓ 59.27%</td>
</tr>
</tbody>
</table>
Results on Postgres

Figure 14: Performance comparison for TPC-C workload using instance CDB-D among CDBTune, Postgres default, CDB default, BestConfig, DBA and OtterTune (on Postgres).
Results on MySQL

![Graphs showing performance metrics for TCP-C](image)

(a) TCP-C (Throughput)

(b) TCP-C (99%-tile Latency)

Figure 18: Performance on TPC-C for local MySQL.
Results on MongoDB

Figure 13: Performance comparison for YCSB workload using instance CDB-E among CDBTune, MongoDB default, CDB default, BestConfig, DBA and OtterTune (on MongoDB).
QTune: Query-Aware Tuning

- Query-aware tuning
- Encoding queries
- Encoding cost
- Double state tuning

Guoliang Li, Xuanhe Zhou. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. VLDB 2019
QTune

3.1 Query Information

A SQL query includes query type (e.g., insert, delete, select, update), tables, attributes, operations (e.g., selection, join, groupby). Query type is important as different query types have different query cost (e.g, OLTP and OLAP have different effect on the database), and thus we need to capture the query type information in the vector. Tables involved in a query are also important, because the data volumes and structures of tables will significantly affect the database performance. Based on the table information, our tuning model decides whether the current system configuration can provide high performance; if not, our tuning system can tune the corresponding knobs. For example, if the buffer is not large enough, we can increase the buffer.

Note that we do not featurize the attributes (i.e., columns) and operations (i.e., selection conditions) due to three reasons. First, the query cost will capture the operation information and cost, and we do not need to maintain duplicated information. Second, operations are too specific and adding specific operations into the vectors will reduce the generalization ability. Third, the attributes and operations will be frequently updated and it requires to redesign the model for the updates. We will compare with the method that also considers attributes and operations in Section 6.1.2.

In summary, for query information, we maintain a 4 + |T| dimensional vector, where |T| is the number of tables in the database. The first four features capture the query types, e.g., insert, select, update, delete. For an insert/select/update/delete query, the corresponding value is 1; 0 otherwise. Each last |T| feature denotes a table. If the query contains the table, the corresponding value is 1; 0 otherwise. For example, Figure 3 shows a query vector. There are 8 tables. The first 12 features are used for query information. It is a selection query and uses tbl1, tbl2 and tbl3, so the first four values are 1 and the other 8 values are 0.

3.2 Cost Information

The cost information captures the cost of processing the query. However, a query usually has many possible physical plans and each plan has different query cost. So it is not realistic to directly parse the query statement to extract query cost. Instead, we utilize the query plan generated by the query optimizer, which has a cost estimation for each operation. Figure 3 shows an example query plan, where each node has a cost estimation. As each database has a fixed number of operations, e.g., scan, hash join, aggregate, we use the cost on these operations to capture the cost information. For example, in PostgreSQL there are 38 operations. Note that an operation may appear in different nodes of the tree plan, and the cost of the same operation should be summed up as the corresponding cost value in the query cost vector. For example, in Figure 3, the value of hash join equals to the sum of the costs in the two Hash Join nodes. After gaining the query cost, we normalize the cost by subtracting mean and dividing the std deviation.

In summary, for cost information, we maintain a |P| dimensional vector, where P is the set of operations in database, and |P| is the number of operations.

3.3 Character Encoding

We concatenate the query vector and cost vector to generate an overall vector of a query. For example, Figure 3
QTune

(a) Sysbench (RW)

(d) Sysbench (RW)
View Advisor

- Equivalent subquery detector
- Subquery cost/benefit estimator
- View selector

In this section, we present our system framework. As shown in Figure 3, the system consists of four main components:

1. **SQL Engine**
   - Processes batch queries and generates query results.

2. **View Matcher**
   - Compares subqueries for equivalence.
   - Rewrites SQL statements based on matched subqueries.

3. **View Optimizer**
   - Selects and materializes views.
   - Optimizes view selection and materialization.

4. **View Advisor**
   - Guides the system through online and offline processes.

The system is designed to efficiently optimize and materialize views for query processing. It leverages techniques of reinforcement learning to address the ILP problem and estimate subquery computation cost.

Given a set of queries, the system first collects queries from the meta database. It then parses and compiles the queries into logical expressions. The Materialized View Selector and Subquery Cost Estimator work together to select the most cost-effective subqueries.

Finally, the SQL engine generates views for the selected subqueries, materializes them into the database, and compiles the queries into logical expression plans. This process efficiently improves query latency in analytics clusters by selecting and materializing views for selected subqueries.
View Advisor: Framework

1. MV Candidate Generation
2. MV Estimation Model - encoder-reducer
3. MV Selection - DRL
4. MV-aware Query Rewriting
View Advisor: Equivalent Subquery

- Expensive to verify equivalence
- Extract SPJG segments
- Evaluate SPJG segments
View Advisor: Cost Estimator

1. Serializing and Encoding
2. Encoder-Reducer Model
   - encoder – cost without view
   - reducer – cost with view
3. Attention Mechanism
4. Fine-tuning
View Advisor: Subquery Selector

- Learn the benefit of materializing a view of a subquery

![Diagram of neural network with layers and connections]

1. State $s_t$, reward $r_t$
2. Action $a_t$
3. State $s_{t+1}$, reward $r_{t+1}$

Environment

ILP Solver

- Four common layers
- Advantage layer
- Value layer
- Output layer

To evaluate the performance of our system. Firstly, we illustrated two workloads, which were collected from real SQL query jobs in Alibaba and the two workloads were respectively named as WK1 and WK2. For each workload, we regarded the function \( \text{IterLabel} \) as a baseline method and compared it with some popular machine learning methods, such as LR, GDBT, Wide-Deep, and ProBased-Label. We used three experimental datasets.

We selected two workloads with different characteristics. Two workloads were formed with a tuple of queries and the number of extracted subqueries, which were respectively called WK1 and WK2. For each workload, we calculated the top-k normalized subqueries by normalizing the maximum number of queries to 1000. We ranked subqueries based on the number of times they appeared among different queries, which were respectively named as \( \text{TopkFrequency} \), \( \text{TopkStorage} \), \( \text{TopkUtility} \), and \( \text{TopkUtility} \). The four methods were compared with \( \text{ProBased-Label} \) and \( \text{TopkUtility} \). The parameters in our system can be divided into two parts. The first part was manually set by users, which were respectively named as the number of queries and the number of extracted subqueries. The second part was adjusted automatically by the system, which was denoted as the final output model. We evaluated the overall performance of our system. Finally, we compared our methods with some popular machine learning methods, such as LR and GDBT.
Level 2: AI-Assisted DB

- Monitor and manage DB
  - Self monitoring
    - Statistics, workload, system
  - Self diagnosis
    - Error detection
  - Self healing
    - Failure recovery
  - Self configuring
    - Workload, upgrade
  - Self optimizing
    - SQL rewriter, online statistics
Self Monitoring

Database System

Instrumentation Information

Database Administrators

Unattended DB Management

Attended DB Management

Online Changeability

Information Store

Repository Manager

Autonomous DB Management Console

Autonomous DB Functionality

Anomaly Manager

Change Manager

Workload Manager

Common Services

Visualization / Rule / ML

OS/HW/Network/App Instrumentation Information
Workload Configuration

- Workload modelling
  - Feature, cost, latency, resources

- Workload scheduling
  - Prioritize workload
  - OLAP & OLTP

- Workload prediction
  - Predicting workloads

![Workload Forecasting Graph](image)
AI-Native Database
Level 3: AI-Enhanced DB

- **AI4DB: Learned DB components**
  - Learned Index
  - Learned Cost estimator
  - Learned Optimizer
  - Learned Statistics

- **DB4AI: Declarative AI**
  - Use SQL for using AI algorithms
  - Lower the burden of using AI
Learned Cost Estimator

- Traditional Cost Estimator
  - Histogram
  - Sketch
  - Empirical functions
  - Failed for correlations between multiple tables

- Learned Cost Estimator
  - Estimation model
  - Tree-structure model
  - Predicate embedding
Learned Cost Estimator

1 Training Data Generator

2 Feature Extractor

3 Tree-structured Estimation
Learned Cost Estimator

Feature Extraction

---

**Execution Plan**

- **Node Type**: "Nested Loop"
  - **Join Filter**: "(mc.movie_id = t.id)"

- **Node Type**: "Hash Join"
  - **Hash**: "(mc.movie_id = mi.movie_id)"

- **Node Type**: "Seq Scan"
  - **Table Name**: "info_type"
  - **Filter**: "info = 'top 250 rank'"

- **Node Type**: "Hash Join"
  - **Hash**: "(mc.movie_id = mi.movie_id)"

- **Node Type**: "Index Scan"
  - **Table Name**: "movie_companies"
  - **Filter**: "production_year > 2010"

---

**One-hot Encoding**

- Operator: one-hot
- Table Name: one-hot
- Column: one-hot

<table>
<thead>
<tr>
<th>Operator</th>
<th>one-hot</th>
<th>Table Name</th>
<th>one-hot</th>
<th>Column</th>
<th>one-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>0001</td>
<td>info_type</td>
<td>000000000001</td>
<td>mc.movie_id</td>
<td>000000000001</td>
</tr>
<tr>
<td>&gt;</td>
<td>0010</td>
<td>movie_companies</td>
<td>000000000100</td>
<td>ct.kind</td>
<td>000000000000</td>
</tr>
<tr>
<td>not like</td>
<td>0100</td>
<td>movie_info_idx</td>
<td>000000001000</td>
<td>mc.movie_id</td>
<td>000000000001</td>
</tr>
<tr>
<td>like</td>
<td>1000</td>
<td>movie_companies</td>
<td>000000000100</td>
<td>mi.movie_id</td>
<td>000000000001</td>
</tr>
</tbody>
</table>

---

**Sample Datasets**

<table>
<thead>
<tr>
<th>movie_companies.note</th>
<th>movie_companies.production_year</th>
<th>company_type.kind</th>
<th>distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(as Metro-Goldwyn-Mayer Pictures)%</td>
<td>(2006)(USA)(TV)</td>
<td>1995</td>
<td>distributors</td>
</tr>
<tr>
<td>(co-production)</td>
<td>(2006)(worldwide)(TV)</td>
<td>2013</td>
<td>special effects companies</td>
</tr>
<tr>
<td>(co-production)</td>
<td>(2011)(UK)(TV)</td>
<td>1966</td>
<td>miscellaneous companies</td>
</tr>
</tbody>
</table>

---

**Dictionary**

<table>
<thead>
<tr>
<th>Token</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>top 250 rank</td>
<td>0.14,0.43,...,0.92</td>
</tr>
<tr>
<td>production companies</td>
<td>0.51,0.22,...,0.11</td>
</tr>
<tr>
<td>(as Metro-Goldwyn-Mayer Pictures)</td>
<td>0.91,0.35,...,0.25</td>
</tr>
<tr>
<td>(co-production)</td>
<td>0.37,0.11,...,0.02</td>
</tr>
<tr>
<td>(presents)</td>
<td>0.13,0.41,...,0.76</td>
</tr>
</tbody>
</table>

---

**Sample Bitmap**

<table>
<thead>
<tr>
<th>id</th>
<th>Operation</th>
<th>MetaData</th>
<th>Predicate</th>
<th>Sample Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001</td>
<td>padding</td>
<td>000000000001</td>
<td>padding</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>padding</td>
<td>000000000001</td>
<td>padding</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>padding</td>
<td>000000000001</td>
<td>padding</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0001</td>
<td>000001000001</td>
<td>padding</td>
</tr>
<tr>
<td>5</td>
<td>0100</td>
<td>0010</td>
<td>000000001000</td>
<td>padding</td>
</tr>
<tr>
<td>6</td>
<td>0010</td>
<td>padding</td>
<td>000001000000</td>
<td>padding</td>
</tr>
<tr>
<td>7</td>
<td>0100</td>
<td>0100</td>
<td>000100000000</td>
<td>1111</td>
</tr>
<tr>
<td>8</td>
<td>0100</td>
<td>1000</td>
<td>010000000000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1000</td>
<td>1000</td>
<td>010000000000</td>
<td>0100</td>
</tr>
</tbody>
</table>
Learned Cost Estimator

Tree-structured Model
Learned Cost Estimator

String Embedding

<table>
<thead>
<tr>
<th>Rules</th>
<th>&quot;Din&quot; → &quot;Din&quot;</th>
<th>&quot;Dinos&quot; → &quot;Din&quot;</th>
<th>&quot;Dinos&quot; → &quot;Din&quot;</th>
<th>&quot;Dinos in&quot; → &quot;Din&quot;</th>
<th>&quot;Dinos in &quot; → &quot;Din&quot;</th>
<th>&quot;Dinos in Kas&quot; → &quot;Din&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Prefix, P₁(“Din”), 3)</td>
<td>(Prefix, P₅(“D”)P₅Pₛ, 3)</td>
<td>(Prefix, P₀PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅(“D”)P₅PₛPₛ, 3)</td>
<td>(Prefix, P₆PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅(“D”)P₅PₛPₛPₛ, 3)</td>
</tr>
<tr>
<td></td>
<td>(Prefix, P₅(“in”), 3)</td>
<td>(Prefix, P₅PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅PₛPₛPₛ, 3)</td>
<td>(Prefix, P₅PₛPₛPₛPₛPₛ, 3)</td>
</tr>
</tbody>
</table>

Prefix Trie

Like ‘Din%’
Learned Cost Estimator

<table>
<thead>
<tr>
<th>Method</th>
<th>Median</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGCost</td>
<td>4.90</td>
<td>80.8</td>
<td>104</td>
<td>3577</td>
<td>4920</td>
<td>105</td>
</tr>
<tr>
<td>TLSTMMHashMCost</td>
<td>4.47</td>
<td>53.6</td>
<td>149</td>
<td>239</td>
<td>478</td>
<td>24.1</td>
</tr>
<tr>
<td>TLSTMEmbNRMCost</td>
<td>4.12</td>
<td>18.1</td>
<td>44.1</td>
<td>105</td>
<td>166</td>
<td>10.3</td>
</tr>
<tr>
<td>TLSTMEmbRMCost</td>
<td>4.28</td>
<td>13.3</td>
<td>22.5</td>
<td>104</td>
<td>126</td>
<td>8.6</td>
</tr>
<tr>
<td>TPValEmbRMCost</td>
<td>4.07</td>
<td>11.6</td>
<td>17.5</td>
<td>63.1</td>
<td>67.3</td>
<td>7.06</td>
</tr>
</tbody>
</table>

In summary, sample bitmap, tree-structured model, and embedding only on string values in datasets. These representation with stronger generalization ability. For strings in the workload and the method with rule can be trained is difficult, because the tree structure with more strings in the workload and the method with rule can be trained is difficult. Multitask learning can improve the quality of estimation 20% on validation workload, because the tree structure with more strings in the workload and the method with rule can be trained is difficult. For strings in the workload and the method with rule can be trained is difficult. Multitask learning can improve the quality of estimation 20% on validation workload, because the tree structure with more strings in the workload and the method with rule can be trained is difficult. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest. In Figure 8, cause string embedding can capture coexistence relation and this can compare the effects of different predicate embedding representations doesn't have much effect, similar time no matter what predicate is.), we only report the validation queries and it converge speed is the slowest.
Learned Optimizer

- It is expensive to get the optimal plan
- Estimation is not accurate
  - Cost-based method
  - Rule-based method

<table>
<thead>
<tr>
<th>#Relations (n)</th>
<th>#Processing Trees</th>
<th>#Solutions (#Trees · n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>1,680</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>30,240</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
<td>665,280</td>
</tr>
<tr>
<td>8</td>
<td>429</td>
<td>17,297,280</td>
</tr>
<tr>
<td>9</td>
<td>1,430</td>
<td>518,918,400</td>
</tr>
<tr>
<td>10</td>
<td>4,862</td>
<td>17,643,225,600</td>
</tr>
<tr>
<td>11</td>
<td>16,796</td>
<td>670,442,572,800</td>
</tr>
<tr>
<td>12</td>
<td>58,786</td>
<td>28,158,588,057,600</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Worst In Workload

- Left-Deep
- Learning

Sub-Optimality

- Index-Mostly
- Hybrid Hash
- Hash Reuse
Learned Optimizer

- Query feature encoding
- Reward of a join

Q1: Select * From T1, T2, T3, T4 Where T1.a = T2.a and T2.b = T3.b and T1.c = T3.c and T3.d = T4.d and T1.e = T4.e and T2.f = T4.f and T1.h > 50 and T1.h < 100

### A. Query encoding for input SQL

- $R^{c}[T_{1,a}]$
- $R^{c}[T_{2,a}]$
- $R^{c}[T_{2,b}]$
- $R^{c}[T_{3,b}]$
- $R^{c}[T_{1,c}]$
- $R^{c}[T_{1,d}]$
- $R^{c}[T_{1,e}]$
- $R^{c}[T_{2,f}]$
- $R^{c}[T_{3,g}]$
- $R^{c}[T_{4,h}]$

### B. Table and column representation

- Concatenate

### C. Join tree and join state representation
Learned Optimizer

Q1: Select * From T1, T2, T3, T4 Where T1.a = T2.a and T2.b = T3.b and T1.c = T3.c and T3.d = T4.d and T1.e = T4.e and T2.f = T4.f and T1.h > 50 and T1.h < 100

Fig. 2. CTJOIN continuously selects one join as an action until all tables are joined.

Fig. 3. Framework of CTJOIN database. This step can be considered as fine-tuning. We use latency as the standard for the optimizer selection in this step.

B. CTJOIN framework

1) DRL optimizer:
   - State maintains the state information for the join process. Specifically, as shown in figure 2, the state in join process is a join forest. A join forest is composed of several join trees. Termination state is one join tree all tables are joined together. It will be converted to query plan and passed it to the DBMS for execution. Non-terminating state is the state during the join process. It will be passed to the planner to get the action that should be performed now, and state itself is updated according to the selected action. The State can be described as a join tree and initialized using the tables in given query, Eg. initial state \( s = \{T_1, T_2, T_3, T_4\} \) for query accessing \( T_1, T_2, T_3, T_4 \).
   - Planner corresponds to the agent in the RL, is the core part of the entire system. For a given state, each candidate join condition can be considered an action. Action Space = \{ \((T_1.a = T_2.a, joinTree), \ldots\) \} contains all possible actions and corresponding join tree. For join trees in the action space, we will use TreeLSTM to represent them and get the corresponding estimated long-term reward. Action selection gets the result of each join tree from TreeLSTM and selects the action corresponding to the optimal join tree.

   \[ \text{action} = \arg \min_{\text{action}} \text{TreeLSTM}(\text{joinTree}(\text{action})) \]

   - Memory pool records the status of the plan generated by CTJOIN and the feedback from the DBMS. DRL is a neural network-based approach that requires training data to train neural networks. Common practices use replay memory [15] to record the status and systems feedback. We use a memory pool here and sample training data from it to train the TreeLSTM.

2) DBMS: CTJOIN generate the join plan for given query and then passes the join plan to the real DBMS. The DBMS consists of two components, an Estimator and an Executor.
   - Estimator can give the cost of the plan using the static. Estimator uses statistics to estimate the cost without executing.
   - Executor Getting the latency of each join condition is difficult. In order to reduce the difficulty of implementation and make the system easier to migrate to other systems, we adopt a similar approach to ReJoin [12], DQ [10]. We set the feedback(reward) of each step of the intermediate state to 0, and the feedback of the termination state to the cost(latency) of the entire plan.
Learned Optimizer

Query q:
Select *
From T1,T2,T3,T4
Where T1.h > 30
and T1.h < 50
and T1.a = T2.a
and T2.b = T3.b
and T1.c = T4.c

(A) Query Representation for input query

R(q)

R(T1.a) R(T2.a) R(T2)

numerical values and other values (e.g., string). Note that in
We consider two types of columns:
Column Representation.

Next we first discuss the representation
R
B. Representation of Columns and Tables

changes and multi-aliases, which will be further discussed in
for learning the query has good properties to handle schema
R
c
matrix where each cell is 0 or 1.

hidden layers in neural network; and
R
need to be learned using training samples;
W
0
putting all rows, from
0 otherwise. This matrix is then flattened into a vector
i
is a join relation between the

One improved method from this matrix based representation

Let
be the number of tables in a database, and each table

v
vector
0
concatenate
0
into the vector, i.e.,

(T1,T1) (T1,T2) ... (T4,T3) (T4,T4)

(T1,T1) (T1,T2) ... (T4,T3) (T4,T4)

R(q)

R(Join)

ChildSum

N-ary

R(T1)

R(T2)

R(T3)

R(T4)

R(T1)

R(T2)

R(T3)

R(T4)

R(T1)

R(T2)

R(T3)

R(T4)

R(T1)

R(T2)

R(T3)

R(T4)
Learned Optimizer

TABLE II
MEAN RELEVANT COST TO DYNAMIC PROGRAMMING

<table>
<thead>
<tr>
<th>MRC</th>
<th>benchmark</th>
<th>JOB</th>
<th>TPC-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS</td>
<td>1.01</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>ReJoin</td>
<td>1.75</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>QP100</td>
<td>7.81</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>QP1000</td>
<td>1.62</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>DQ</td>
<td>2.34 (1.31)</td>
<td>1.01</td>
<td></td>
</tr>
</tbody>
</table>

TABLE III
EXPONENTIAL MEAN LOG RELEVANT LATENCY (GMRL) TO DP

<table>
<thead>
<tr>
<th>GMRL</th>
<th>benchmark</th>
<th>JOB</th>
<th>TPC-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS</td>
<td>0.67</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>ReJoin</td>
<td>1.14</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>QP100</td>
<td>NA</td>
<td>NA</td>
<td>1.03</td>
</tr>
<tr>
<td>QP1000</td>
<td>1.90</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>DQ</td>
<td>1.23</td>
<td>0.99</td>
<td></td>
</tr>
</tbody>
</table>
Level 3: AI-Enhanced DB

- AI4DB: Learned DB components
  - Learned Index
  - Learned Cost estimator
  - Learned Optimizer
  - Learned Statistics

- DB4AI: Declarative AI
  - Use SQL for using AI algorithms
  - Lower the burden of using AI
DB4AI

- Declarative AI
  - AI to SQL
  - SQL completeness
  - SQL advisor

- AI optimizations
  - Cost estimation
  - Auto parameter
  - Auto model
  - Parallel computing

- Data Governance
  - Data discovery
  - Data cleaning and fusion
Al-Native Database
Level 4: AI-Assembled DB

Self-Assembling

– Each component has multiple options
  • Optimizer
    – CBO, RBO, Learned
  – Assemble the components as a database
    • Reinforcement learning (RL)
  – From single path to multiple paths
    • Like map navigator
  – Scheduling on diversified hardware
    • Learned tensor model on AI hardware
    • Traditional (cost) model on general hardware
Level 4: AI-Assembled DB

SQL
Parser
Rewriter
Optimizer
Executor
Storage
Hardware

Parser
Rewriter (Rule)
Rewriter (Learned)
Optimizer (Rule)
Optimizer (Cost)
Optimizer (Learned)
Executor (Row)
Executor (Column)
Storage (Row)
Storage (Column)
Storage (Memory)
Hardware (CPU)
Hardware (ARM)
Hardware (AI)
Hardware (NVM)
Hardware (SSD)
Level 4: AI-Assembled DB

Self-Assembling

- Each component has multiple options
  - Optimizer
    - CBO, RBO, Learned
- Assemble the components as a database
  - Reinforcement learning (RL)
- From single path to multiple paths
  - Like map navigator
- Scheduling on diversified hardware
  - Learned tensor model on AI hardware
  - Traditional (cost) model on general hardware
Al-Native Database

AI-Native Database

AI-Optimized

AI Advised

Level 1
AI engine plug-in
Offline advisor

Level 2
AI engine built-in
Online monitor

Level 3
AI-driven components &
Declarative AI in DB

Level 4
AI-driven self-assembling & Optimal
AI in DB

Level 5
AI-driven design, verification, and
development

AI Assembled

AI Enhanced

AI Designed
Level 5: AI-Designed DB

- **AI-based design**
  - Data structure design
  - Transaction design
  - Storage design
  - Index design
  - Optimizer design

- **AI & DB Co-design**
  - Unified model
  - Unified optimization
# Five Levels of DB4AI

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1     | **AI Advised:**  
  - Offline AI-based knob tuning/statistics recommendation, offline data placement;  
  - Offline workload management, offline optimization; |
| 2     | **AI Assisted:**  
  - Self monitoring/tuning: online knob tuning, monitoring;  
  - Self optimization: query tuning, online index/view advisor;  
  - Self diagnoses, healing, protection; |
| 3     | **AI Enhanced:**  
  - Using AI-based algorithms to enhance the core components of database;  
  - Learned index, learned optimizer, learned storage, query engine customization, etc.;  
  - AI in database, declarative AI, DB-optimized AI; |
| 4     | **AI Assembled:**  
  - Functions decoupled as services. Functions deployed on heterogeneous environments;  
  - AI-based algorithms to choose the best execution paths of different services; |
| 5     | **AI Designed:**  
  - DB designed by AI, hardware and software codesign, automatic evaluation.  
  - AI-assisted semi-formal or formal verification for trustworthy and security; |
## Five Levels of DB4AI

<table>
<thead>
<tr>
<th>Level</th>
<th>Consumability</th>
<th>Description</th>
<th>AI Skill Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AI Models as UDFs</td>
<td>Algorithms available in the underlying DB system as UDFs or Stored Procedures</td>
<td>High</td>
</tr>
<tr>
<td>2</td>
<td>AI Models as Views</td>
<td>Materialized the trained models as ‘views’ which can be utilized by other users. The views will be automatically updated which is triggered by data update or model update</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td>Model-free</td>
<td>No need to specify models. Given a problem, automatically identify the models.</td>
<td>Low</td>
</tr>
<tr>
<td>4</td>
<td>Problem-free</td>
<td>No need to specify problems. Automatically identify the problems and models.</td>
<td>Low</td>
</tr>
<tr>
<td>5</td>
<td>Full-automatic</td>
<td>Automatically discover AI opportunities, including model selection, algorithm selection and data discovery</td>
<td>Very Low</td>
</tr>
</tbody>
</table>
Lessons Learned

- **AI4DB**
  - Database can learn from both internal and external “environments” to achieve high performance
  - AI enhances database, especially
    - Fast, flexible and strong adaptability
    - Make DB more intelligent

- **DB4AI**
  - In-DB AI consumability
  - Make AI easily used in different fields
Challenges and Opportunities

- **Hardware/Software Co-Design**
  - Database chips
  - Tensor model
  - DB evaluation tool
    - Like EDA

- **OLAP 2.0**
  - Multi-model, DB&AI

- **OLTP 2.0**
  - New hardware
  - NVM, RDMA, etc.
  - Programable RDMA

![Diagram showing Agile Hardware Dev. Methodology](image)

- DP Process
  - CRUD/DDL
- DB Process
  - Analytics style
    - Query
- DB Process
  - Logger/Shared Data

![Diagram showing Hardware and Software Architecture](image)
Thanks!

Q&A