

[170713] 인공지능 하계 워크샵

# Automatic DBMS Tuning 개발 관련 이슈 정리

엑셈-포스텍 R&D 센터  
이도엽

# 하계 프로젝트 진행 소개

□ Senior-New > 1:1 Matching 하여 직접적인 업무 도움 제공 >> 팀장은 전체적 관리 체계

□ 프로젝트 진행

- New 인턴

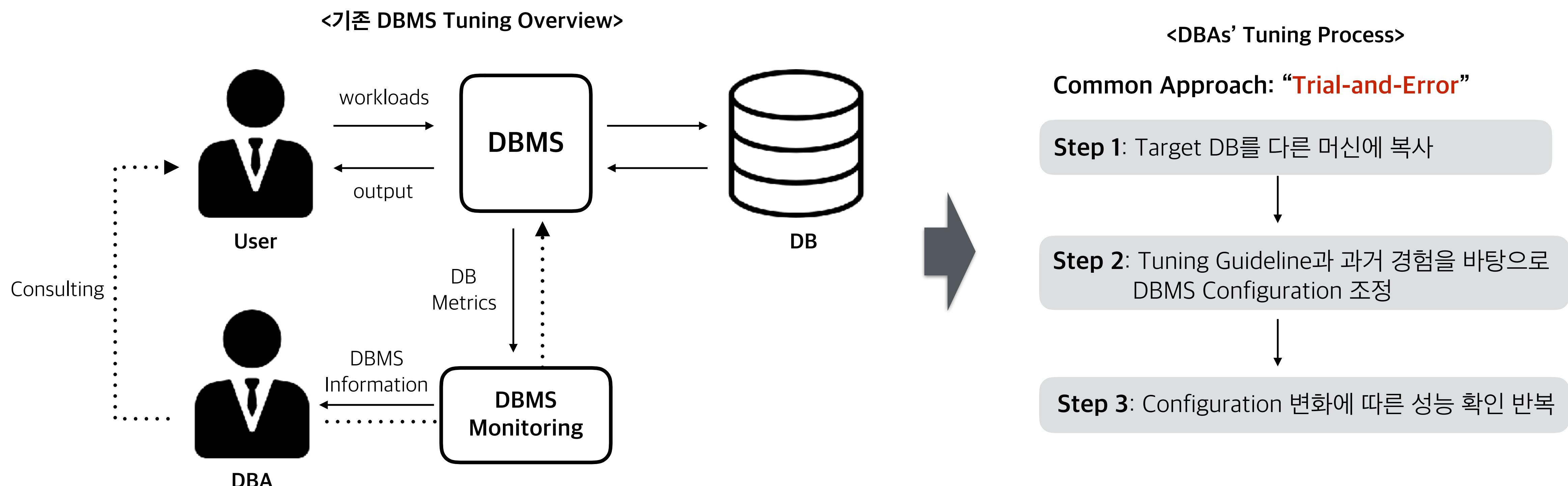
- 1) Machine Learning/Deep Learning Theories and Implementation Study
- 2) 교육자료 Renewal 및 사내 세미나 진행
- 3) 경과에 따라 1개월 후 프로젝트 Support

- 팀장 & Senior: 본격적인 프로젝트 진행

- 1) Automatic DBMS Configuration Tuning (Otter Tune 사례 중심 연구 시작 및 개발)
- 2) Anomaly Detection for Time Series (Unsup. Approach, 팀장 진행)
- 3) 명화공업 과제 ML 기술 분류 지원

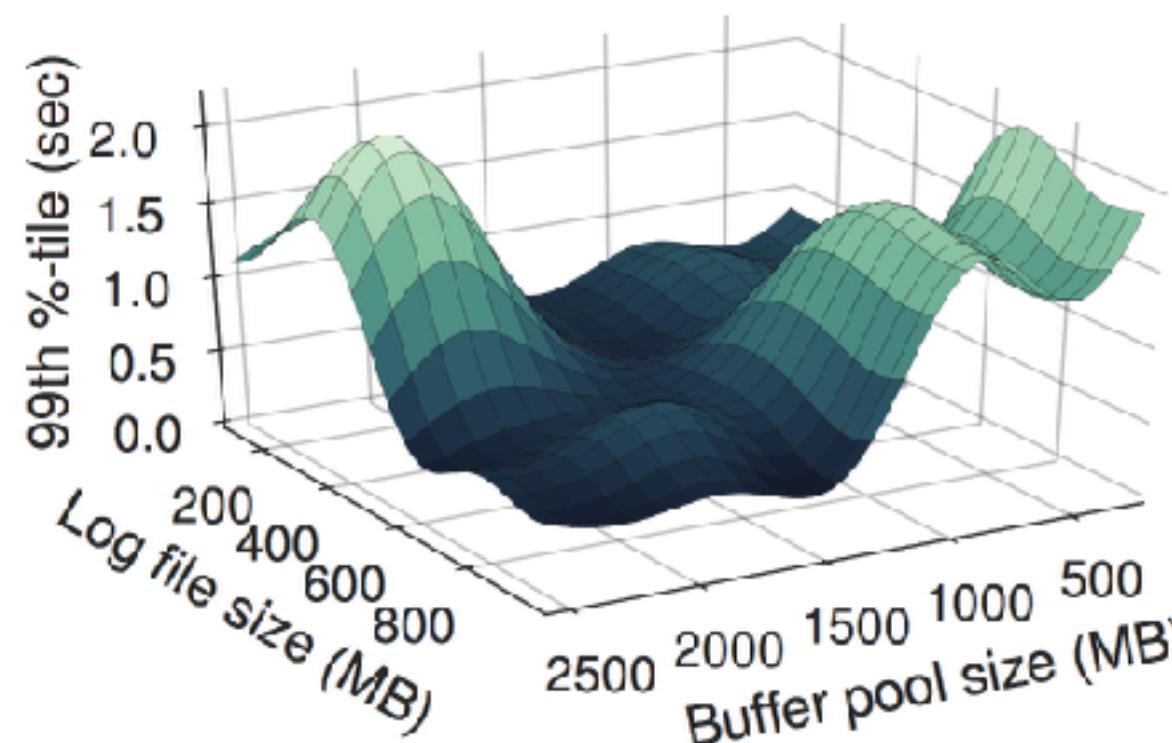
# Background

- DBMS ( DataBase Management System)는 데이터베이스(DB)를 관리하는 프로그램으로 **DB의 종합적인 성능을 결정**
- **Big data applications**의 발달과 함께 **DBMS 최적화**의 중요성은 시간에 따라 더욱 중요한 이슈로 자리잡고 있음
- **DBA ( DataBase Administrator)**는 DBMS의 높은 성능을 위해, 관리 소프트웨어를 사용하고 발생한 상황에 따라 **DB configuration**을 조정

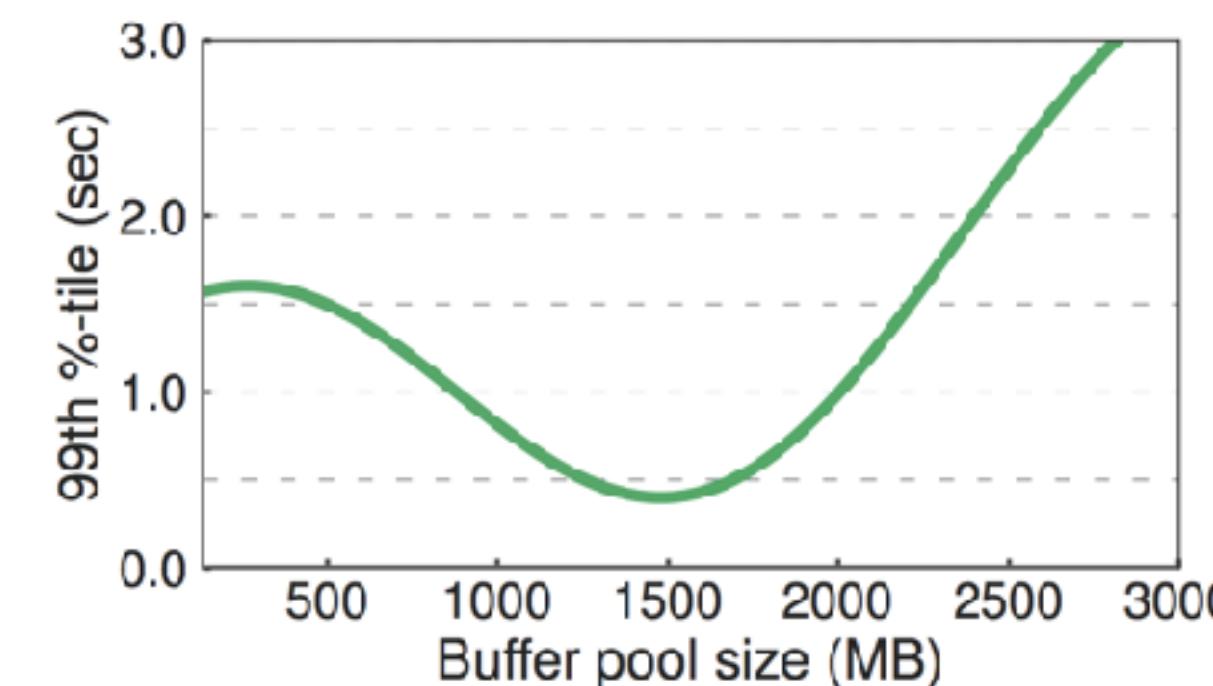


# Motivations of Research

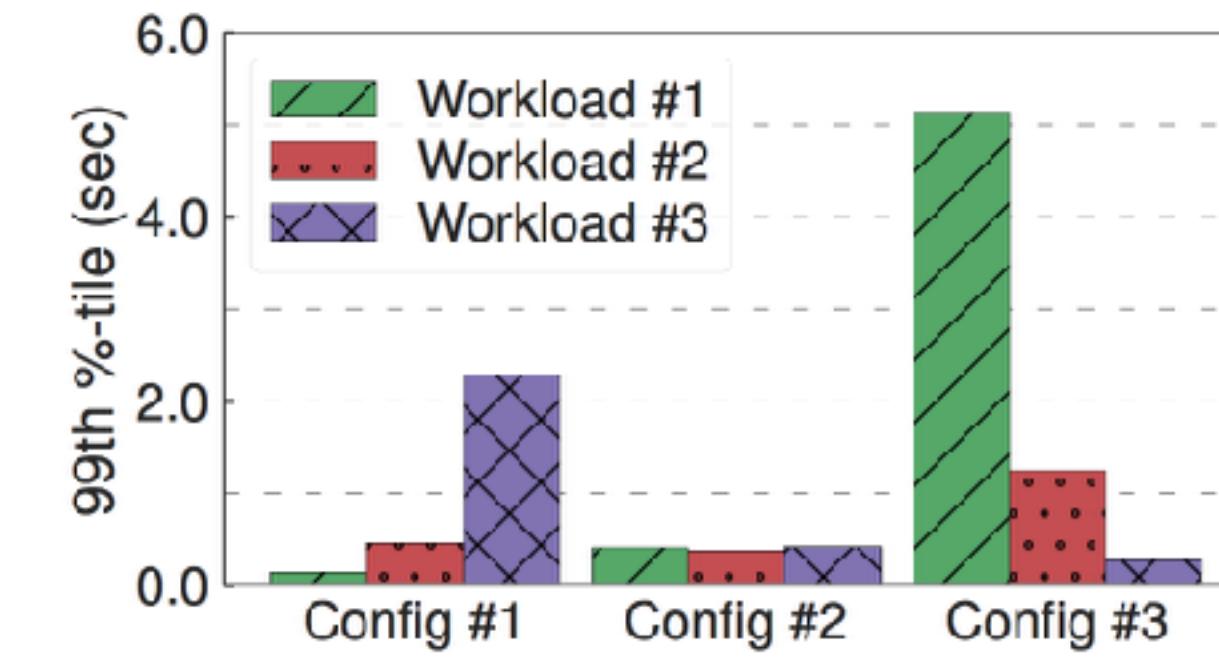
- ▣ 기존 “Trial-and-Error” 방식은 Big Data 기술 발달로 인한 환경 변화에 따라 효율이 낮고 비용이 높은 단점을 가지게 됨
- ▣ 기존 Tuning 방식의 낮은 효율과 높은 비용의 원인은 DBMS Tuning의 아래 5가지 특성에 기인함
  - 1) Dependencies: 튜닝의 대상이 되는 노브(Knob)들의 영향이 독립적이지 않음
  - 2) Continuous Settings: 몇몇 노브들의 값은 연속적인 (continuous) 값을 가지고 있음
  - 3) Non-resusable Configuration: 특정 Configuration이 모든 상황에서 최적의 설정이 아님
  - 4) Tuning Complexity: DBMS들은 항상 새로운 노브를 추가함 (노브의 수가 항상 증가함)
  - 5) Non-transferability of Past Experience: 과거 튜닝을 통해 얻은 DBA의 지식이 축적되어 사용되거나, 다른 DBA에게 전달이 힘듦



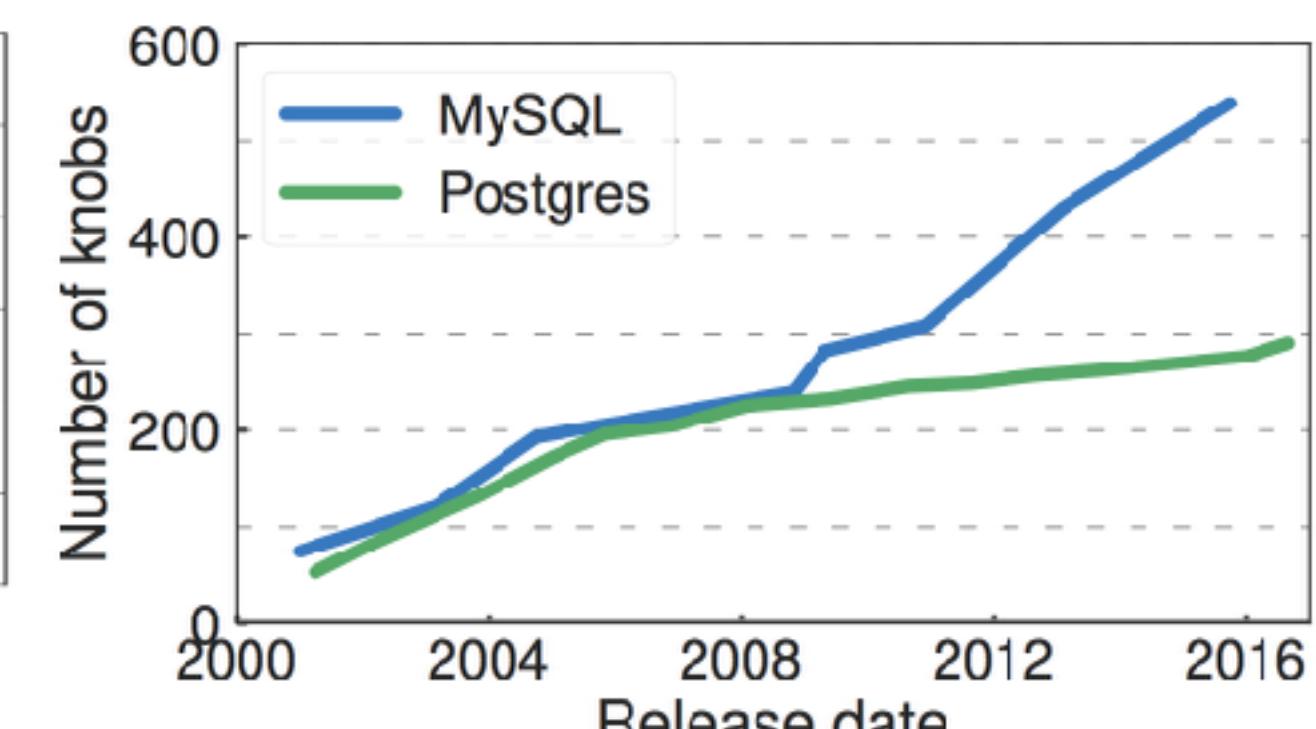
(a) Dependencies



(b) Continuous Settings



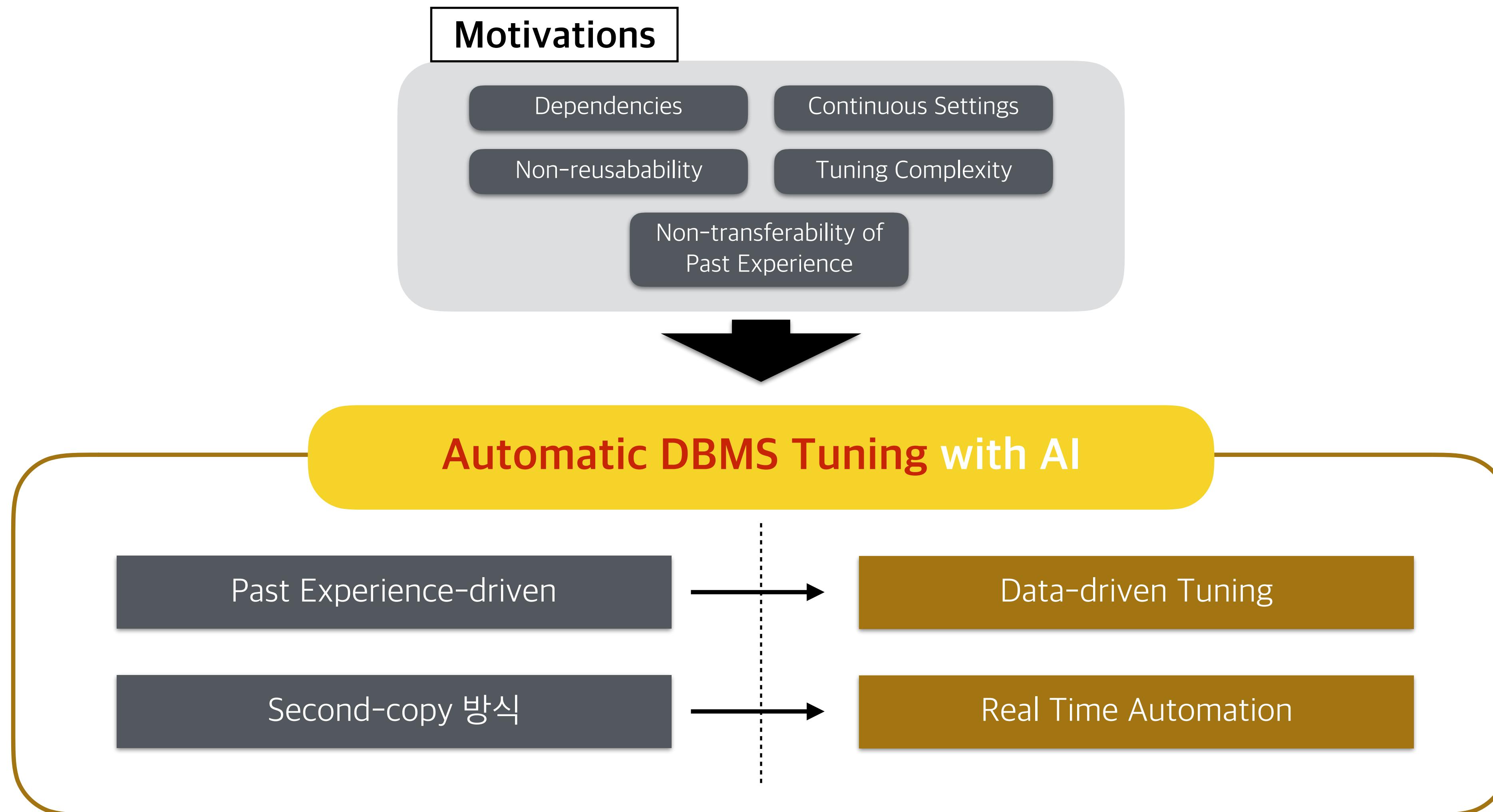
(c) Non-Reusable Configurations



(d) Tuning Complexity

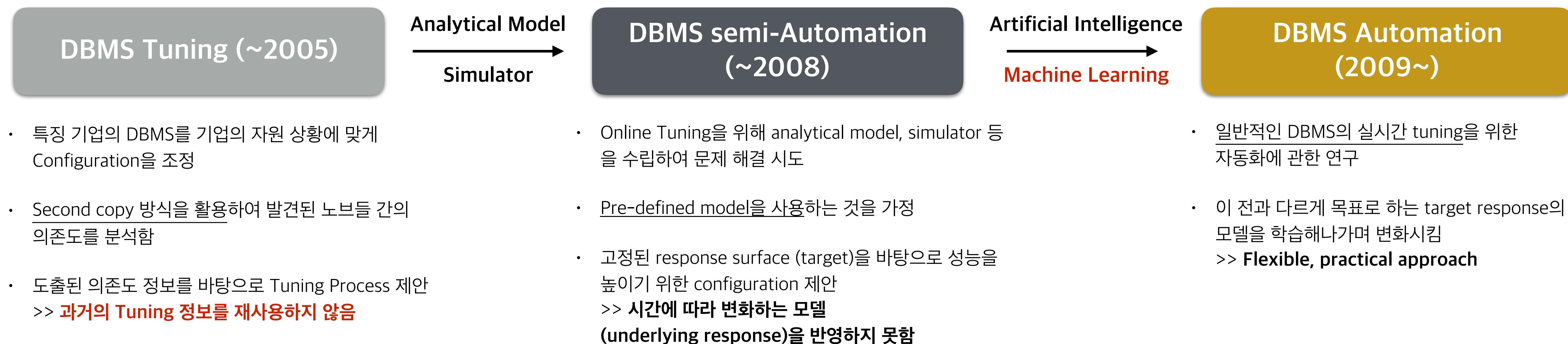
# Objectives of the Study

## ▣ 기존 DBMS Tuning의 단점과 Big Data 기술 환경 변화를 고려한 새로운 DBMS Tuning 방식 필요



# Preceding Researches

- (~2005) 과거 DBMS Configuration Tuning에 관한 선행 연구는 일반적인 목적으로 자동화하기 보다는, 특정 회사의 상황에 맞게 DBMS를 자동화하는 second copy 방식으로 한정적임 (Dias et al., 2005, Kumar et al., 2003, Narayanan et al. 2005)
- (~2008) 실시간 Tuning에 관한 연구는 pre-defined model을 사용하여 시간에 따른 모델의 변화를 반영하지 못함 (Chaudhuri et al., 1997, 2007, Danna et al., 2003)
- (2009~) 신러닝을 적용하여 실시간 tuning을 지원하며, 시간에 따라 알맞은 모델을 지속적으로 학습하는 방향으로 연구 진행



# Introduction to OtterTune

- 최근 Automatic DBMS Tuning과 관련한 연구가 SIGMOD' 17과 AWS AI Blog를 통해 공개됨
- CMU (Carnegie Mellon University)의 Database Group을 중심으로 진행된 DBMS Tuning 자동화 연구는 현재 Apache License 2.0와 함께 Github에 공개되었음
- 머신러닝 기반의 모델을 통해 다양한 DBMS의 Configuration을 효과적으로 튜닝하는 OtterTune 공개

## Automatic Database Management System Tuning Through Large-scale Machine Learning

Dana Van Aken  
Carnegie Mellon University  
dvanaken@cs.cmu.edu

Andrew Pavlo  
Carnegie Mellon University  
pavlo@cs.cmu.edu

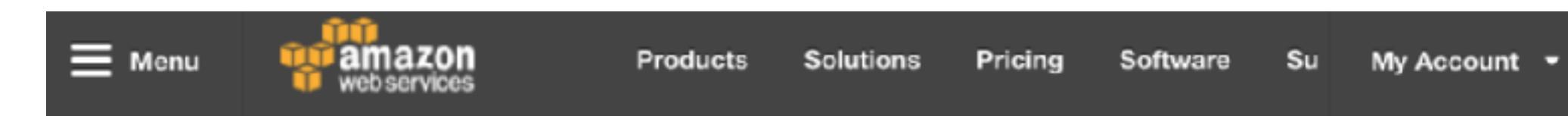
Geoffrey J. Gordon  
Carnegie Mellon University  
ggordon@cs.cmu.edu

Bohan Zhang  
Peking University  
bohan@pku.edu.cn

**Title:** Automatic Database management System Tuning  
Through Large-scale Machine Learning

**Authors:** Dana et al. (CMU)

**Publication:** SIGMOD'17 (May 14-19, 2017)



AWS AI Blog

## Tuning Your DBMS Automatically with Machine Learning

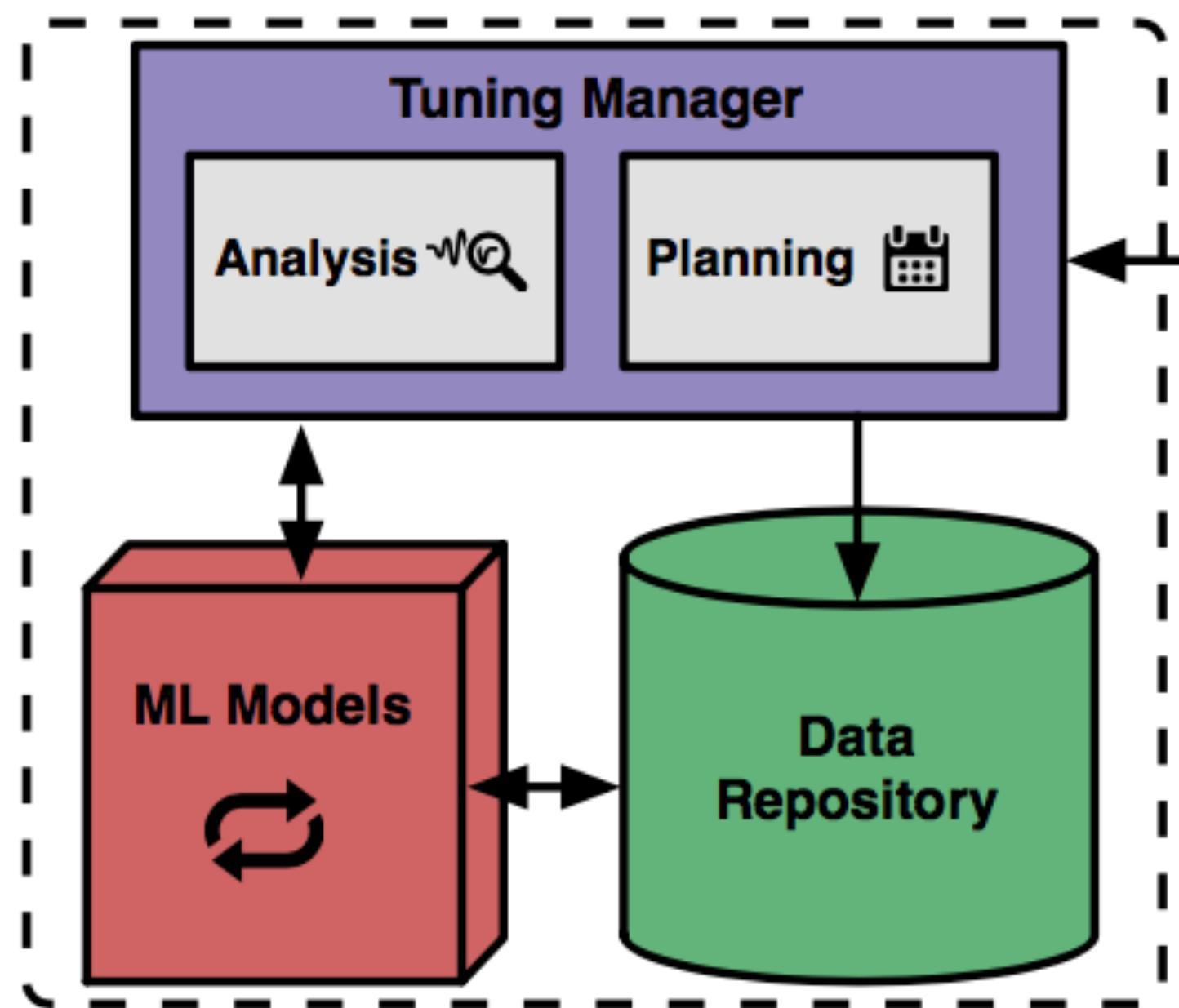
by Dana Van Aken, Geoff Gordon, and Andy Pavlo | on 02 JUN 2017 | Permalink | Comments

Commit	Description	Date
dvanaken committed on GitHub Update README.md	cleanup: removed unused code and organized into directories	2 months ago
analysis	Rename __init__.py to __init__.py	28 days ago
common	Initial commit	3 months ago
.gitignore	Initial commit	3 months ago
LICENSE	Initial commit	28 days ago
README.md	Update README.md	28 days ago
__init__.py	lots of refactoring and added experiment class	a year ago

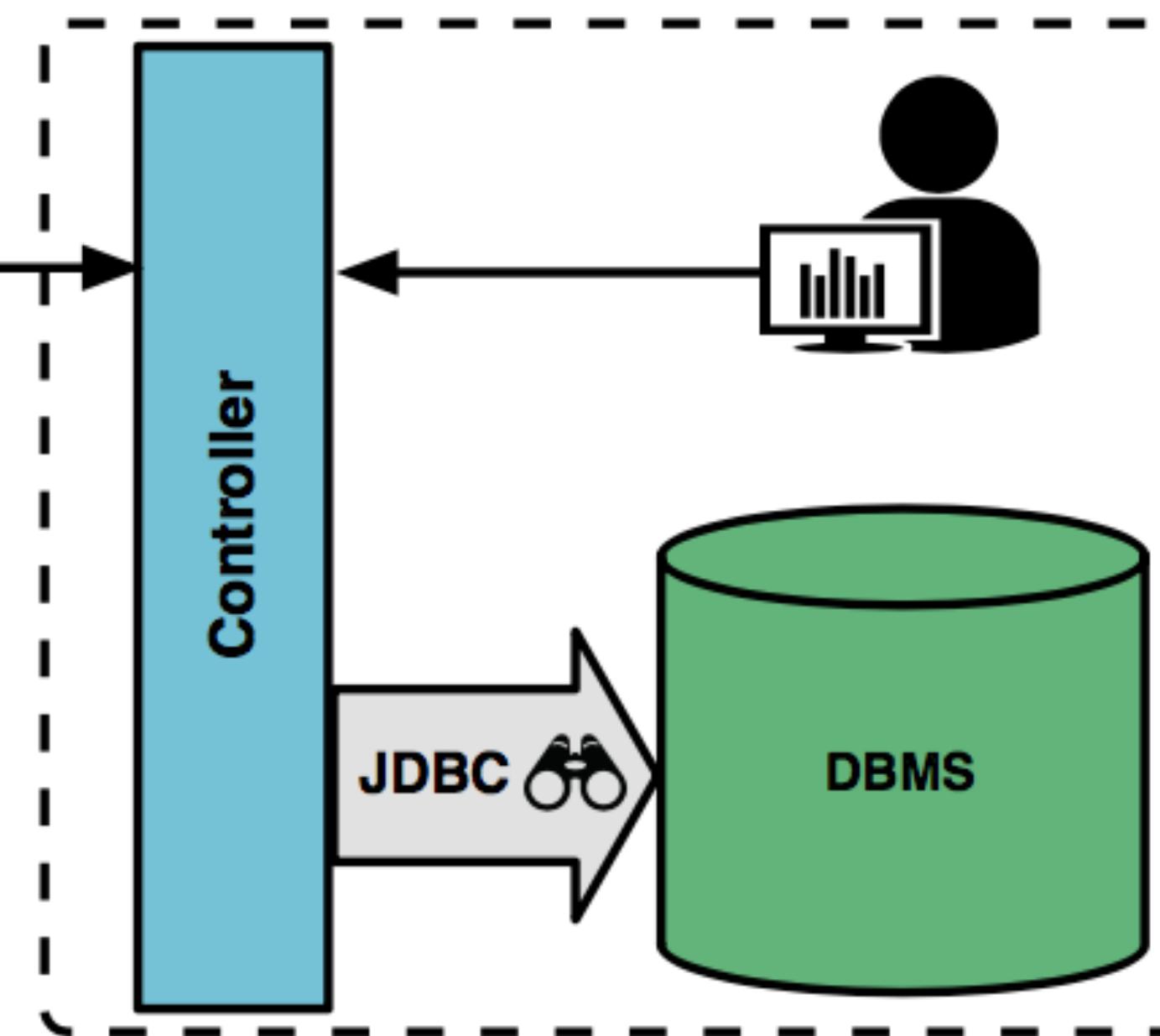
# OtterTune Architecture

- OtterTune 시스템의 구조는 1) Client-side Controller, 2) OtterTune Tuning Manager로 구분됨

## OtterTune Tuning Manager



## Client-side Controller



### 1) Client-side Controller

- DBMS  
: MySQL, Postgres, Vector 등의 일반적인 DBMS
- JDBC  
: DBMS로부터 runtime information을 수집하기 위한 표준 API
- Controller  
: 새로운 configuration을 설치하고, 성능을 측정

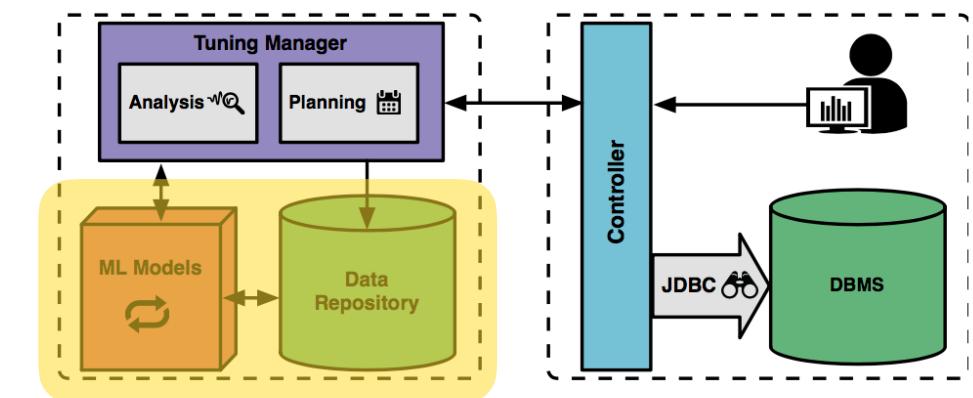
### 2) OtterTune Tuning Manager

- Data Repository  
: 모델 학습을 위해 수집된 데이터를 실시간으로 저장
- ML Model  
: Data Repository에 저장된 데이터를 바탕으로, 모델 업데이트
- Tuning Manager  
: 수집된 DB Metric에 맞는 최적 Configuration을 ML Model을 통해 도출

# ML Model of OtterTune

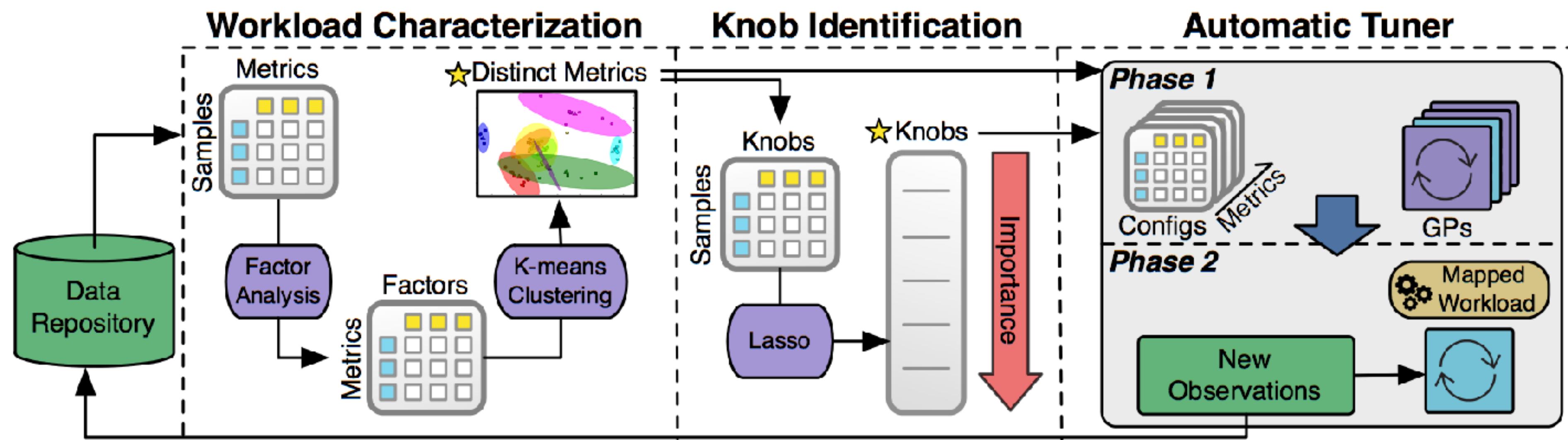
- OtterTune의 ML Model은 3단계의 프로세스를 통해 최적의 DBMS Configuration를 계산함

P1) Workload Characterization: 수 많은 DB Metric 중 Workload를 구분하기 위해 사용할 대표 Metric 선정



P2) Knob Identification: Target Metric(throughput, latency...)을 최적화에 사용할 노브(Knob)들의 중요도 도출

P3) Automatic Tuner: 과거 Workload 중 현재 상황과 가장 잘 맞는 workload를 찾고, 이를 바탕으로 최적의 configuration 예측

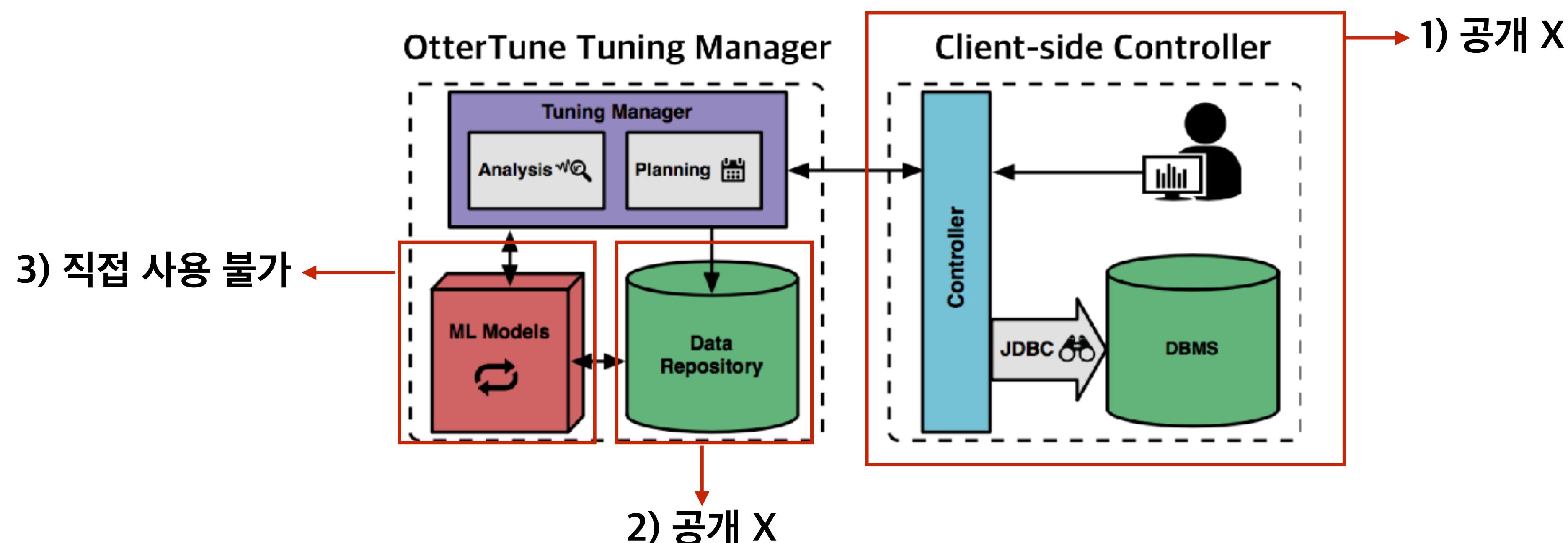


# OtterTune 공개 내용 분석

## ▣ OtterTune 구현 내용은 Github으로 접근가능하도록 공개되어 있지만, 실행에 대한 다수의 문제 존재

- 1) Client-side Controller 환경은 전혀 공개되어 있지 않음
- 2) Training Data와 학습된 Inference Model 공개되어 있지 않음
- 3) Tuning Manager 부분의 경우 일부 자체 제작한 라이브러리들이 공개되어 있지 않아, 알고리즘 자체 실행 어려 존재

<OtterTune Architecture>



- ▣ 현재 해결해야할 현안을 앞에서 언급된 세 가지 문제로 나누고, 센터 역량으로 실행 가능한 부분을 우선적으로 진행하고 있음

## Automatic DBMS Tuning 진행에 관한 R&D 센터 현안 목록

1) DBMS 사용 환경 구축을 통해  
Workload 실행 및 Configuration 조절

2) Config 설정과 Workload 실행에 따른  
데이터 저장 및 관리 환경 (Metric, Config)

3) 보유한 데이터를 바탕으로  
새로운 Config을 제안하는 AI 모델 개발

R&D 센터 차원 독자적 진행 어려움 존재

- 인력 문제
- 기술적 문제

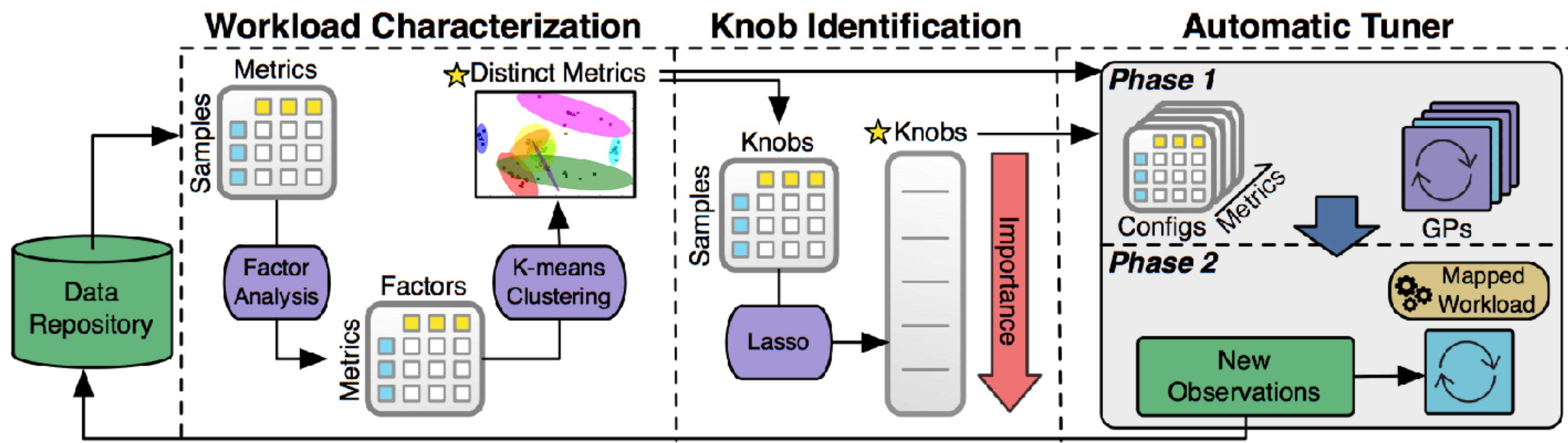
알고리즘 차원에서 기본 모델 구현 완료  
(Ottertune Model)

# Tuning Manager AI Model Design

□ OtterTune에서 제안한 AI Tuning Manager 모델 파이프라인은 아래와 같음

□ Input과 Output 설정은 정확하고 나타낼 수 있음

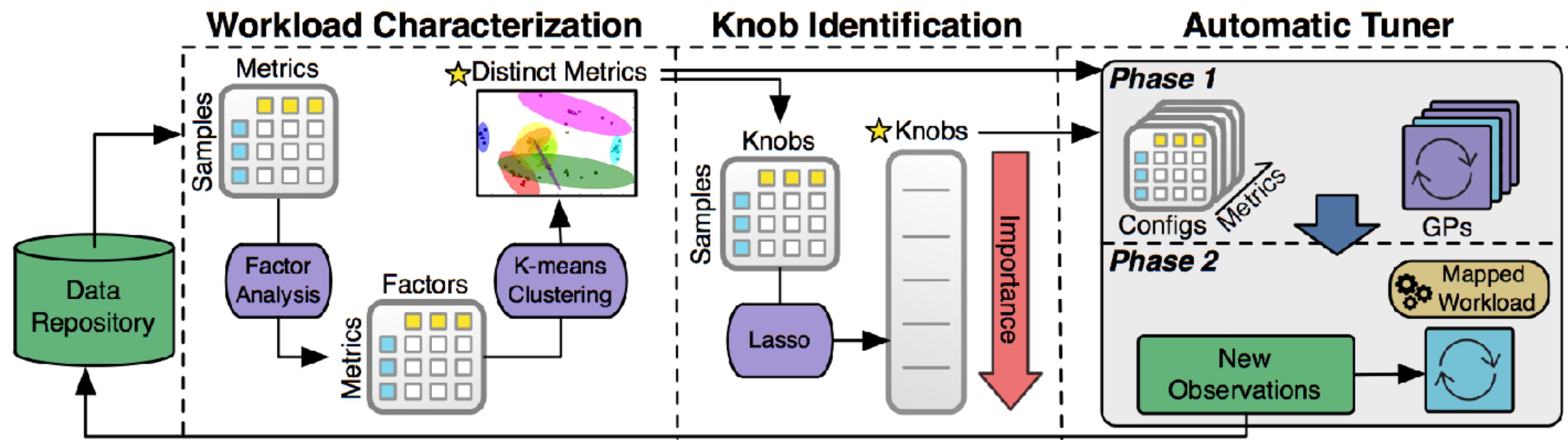
- *Input*: (Workload) x (Metrics) x (Configuration)
- *Output*: (Workload) x (Configuration)
- *Objective*: Improvement of Target Metric



# Tuning Manager AI Model Design

## □ AI DBMS Tuning 모델이 포함하여야할 특징을 OtterTune으로부터 정리

- Metric의 차원이 매우 높으며, 이를 효과적으로 관리할 수 있어야함
- 모든 Knob를 동시에 조절할 수 없으며, Knob 조절의 개수와 순서가 중요함
- 모델은 Online Learning을 지원해야함
- Input에서 Output 도출과정까지 인력의 관여와 의사결정 사항을 최소화해야함 (**Automation**)



## Issues

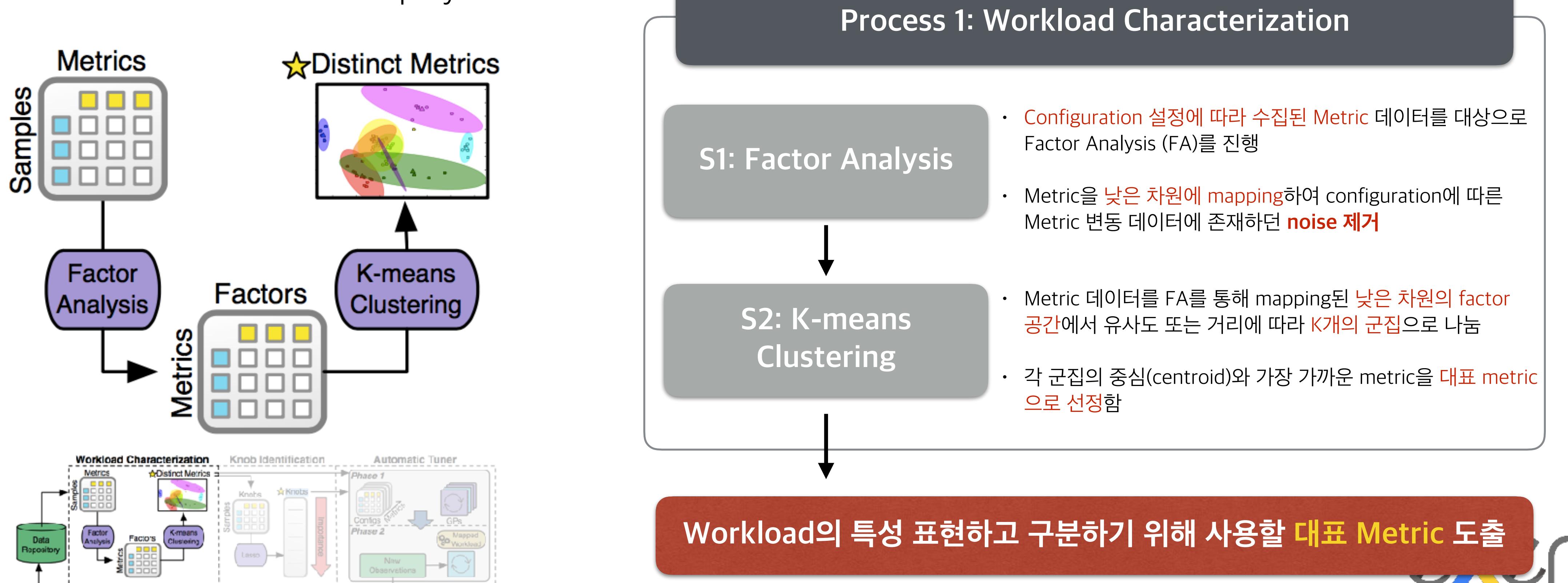
- ❑ 알고리즘 설계 전반적으로 성능의 효율성에 집중되어 있음 >> 효과는 검증하지 않고 타당성 검증 위주
- ❑ **Objective Function Setting** >> 단순 target metric 최대화로는 현실적으로 사용할 수 없는 config. 도출됨  
(실제적인 사용을 위해 고려해야할 여러 constraints를 반영하여 objective model 수립해야함. 사용자 지정 가능해야할까?)
- ❑ P1-P2-P3 모델 파이프라인 사이의 결과의 연결은 구분되어 있음
  - P1 —> P3 결과 영향 존재, 그러나 P3 —> P1 영향
  - P3의 결과로 도출되는 suggested configuration에 따른 오차의 정보가 역으로 P2, P1으로 전달되지는 않음
- ❑ 다량의 많은 종류의 데이터 자체를 이용하는 모델을 사용할 필요가 있음: Deep Learning Model 달리지는 않음

# Workload Characterization

# Overview of Workload Characterization (P1)

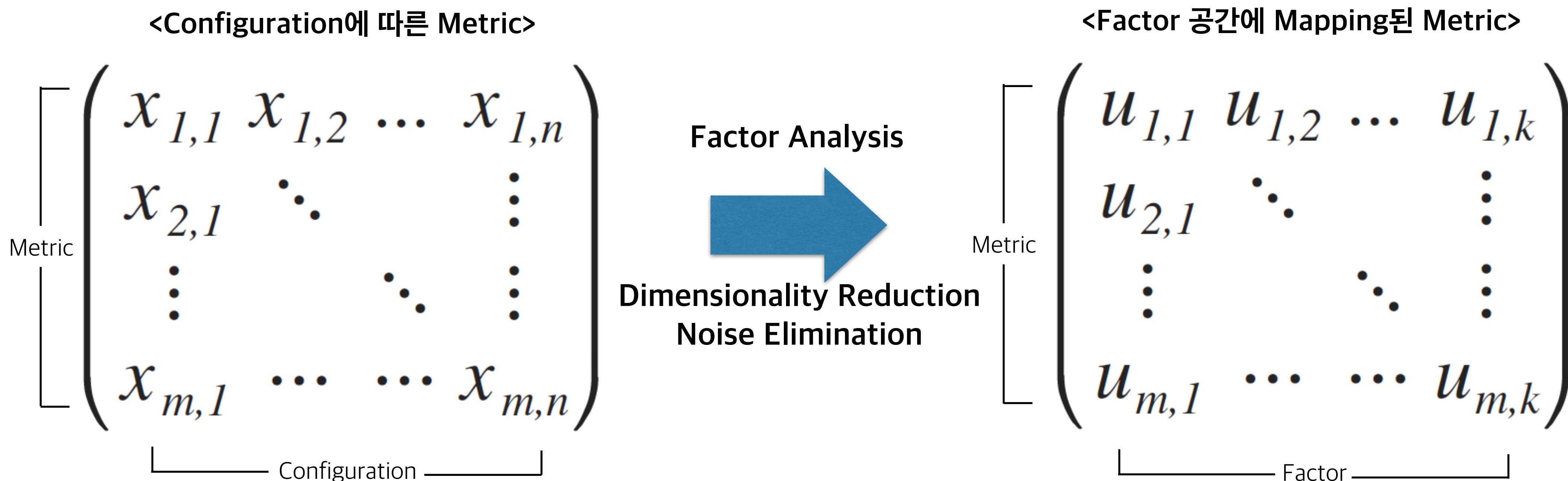
- ❑ Workload의 특성을 파악하기 위해 Workload를 수치화된 특성으로 표현하는 것이 필요함
  - Configuration에 직접적으로 영향을 받는 Internal Metric을 Workload의 수치적 특성으로 사용
- ❑ DBMS Metric의 종류는 매우 많아 모든 Metric으로 Workload를 정의하게 되면 모델의 차원이 높아 계산이 비효율적임
- ❑ Metric의 상관관계와 중복도를 분석하여, 대표적인 Metric을 선정하는 차원 축소 과정이 필요함

<Workload Characterization Deployment>



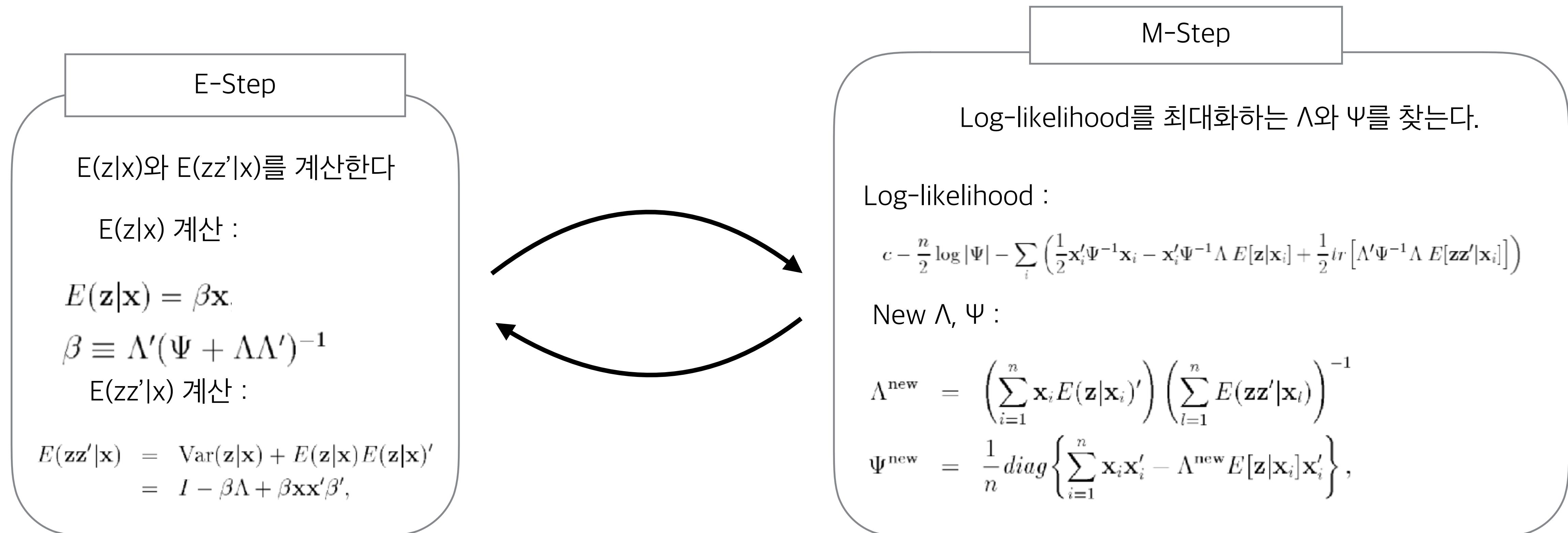
# P1. Data Description for Workload Characterization

- DBMS에 주어지는 Workload를 특성을 파악하고, 구분하기 위하여 **DBMS Internal Metric**을 사용함
  - Read/Write per Transaction 등의 논리적 지표로 workload를 설명할 경우, Knob의 변화에 따른 Impact가 직접적으로 반영되지 않음
  - DBMS의 Runtime Behavior를 나타내며 Knob의 직접적 영향을 받는 Internal Metric을 사용하는 것이 바람직함
- Metric의 종류가 무수히 많고 **중복(redundancy)**가 심하므로 전체 Metric을 대표하는 **소수의 Metric**을 선정할 필요가 있음
- Configuration 변화에 따른 Metric 변화의 특성을 파악하며, 데이터 중복으로 인한 **noise 제거**를 위해 FA 사용



## P1.S1. Factor Analysis

- 수집된 Metric 데이터의 노이즈를 줄이고 Clustering을 Robust하게 만들기 위한 차원축소 과정
- 데이터를 가장 잘 설명하는 Factor의 선형조합을 찾는 과정임
  - $x = \Lambda z + u$ ; ( $x$  : 데이터,  $\Lambda$  : 적재 행렬,  $z$  : Factor,  $u$  : 오차)



E-Step과 M-step을 반복하여 데이터를 가장 잘 설명하는 Factor를 찾을 수 있음

## P1.S2. K-means Clustering

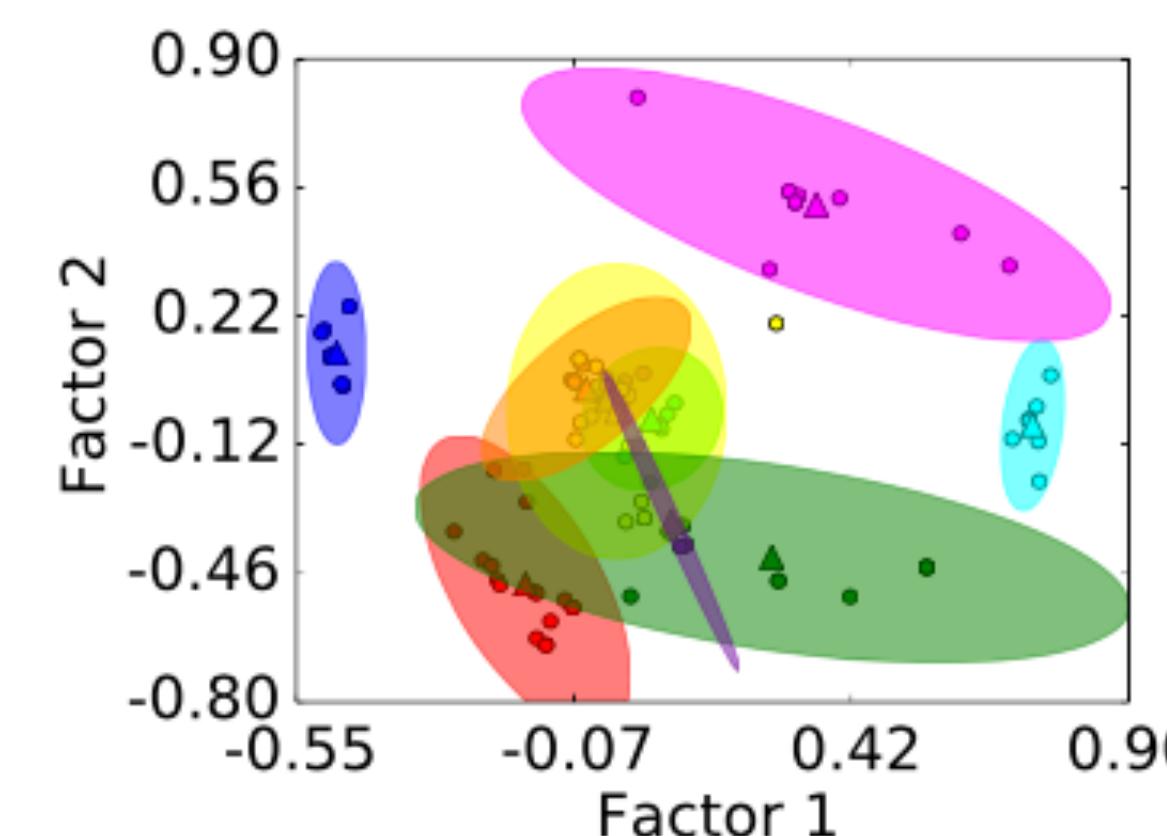
### ❑ K개의 대표 DB Metric을 선정하기 위하여 K-means Clustering 알고리즘을 통한 군집화 진행

- Unlabeled data를 유사한 특성을 갖는 K개의 집단으로 구분하기 위해 사용하는 알고리즘
- 군집의 중심점(centroid)과 각 데이터 포인트의 군집에 대한 할당 결과를 반복적으로 업데이트함
- 각 데이터 포인트는 가장 가까운 중심점에 해당하는 군집에 할당됨

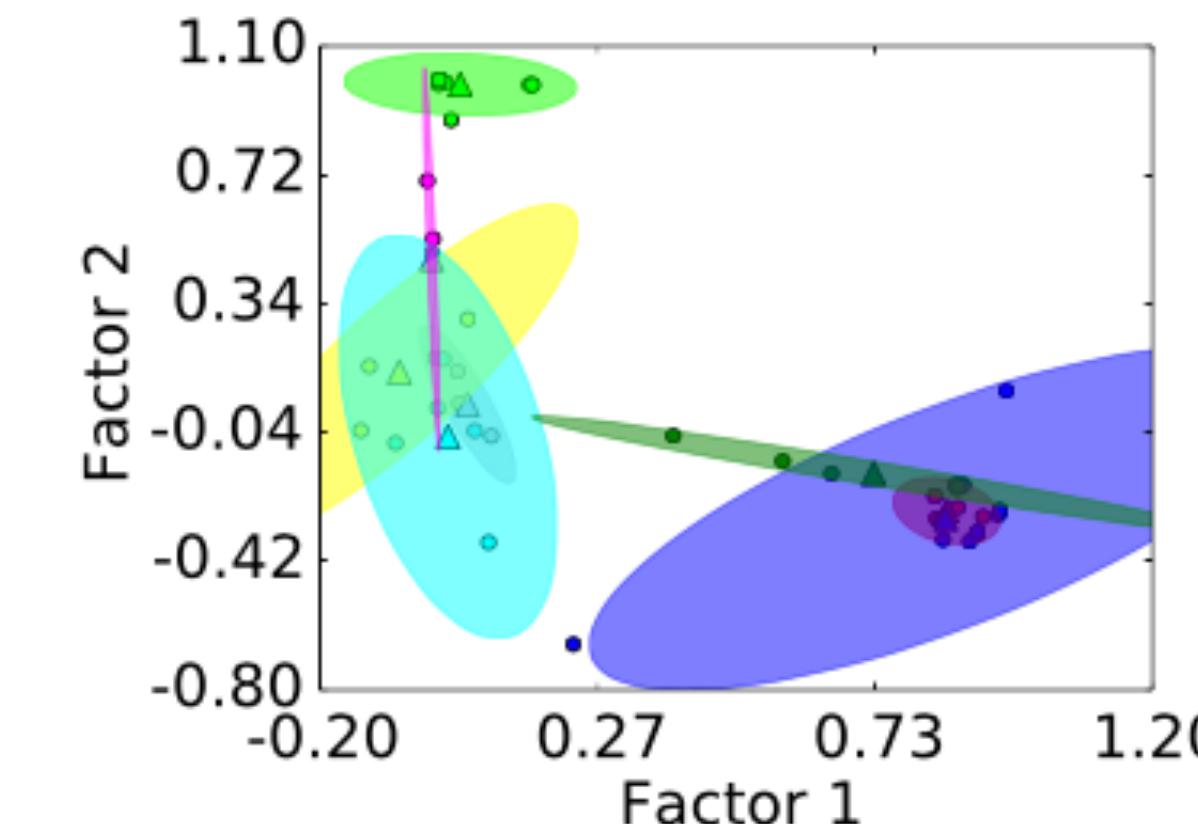
### ❑ P1.S1. Factor Analysis를 통해 얻어진 Factor 공간에서 Metric 데이터의 군집화 진행

### ❑ K개의 중심점과 가장 가까운 K개의 Metric을 대표 Metric으로 선정하고 Workload Characterization에 이용함

<DBMS에 따른 K-means Clustering 결과>



(a) MySQL (v5.6)

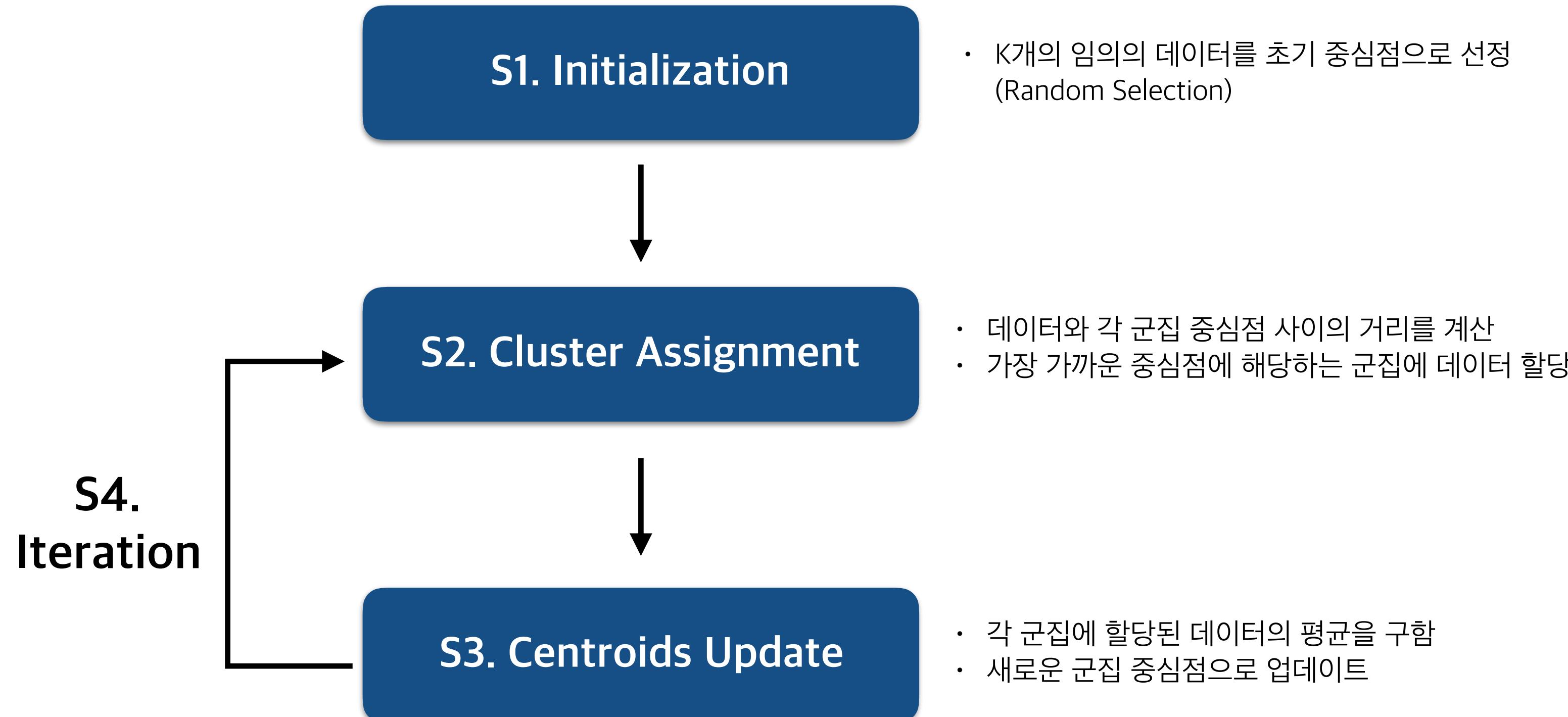


(b) Postgres (v9.3)

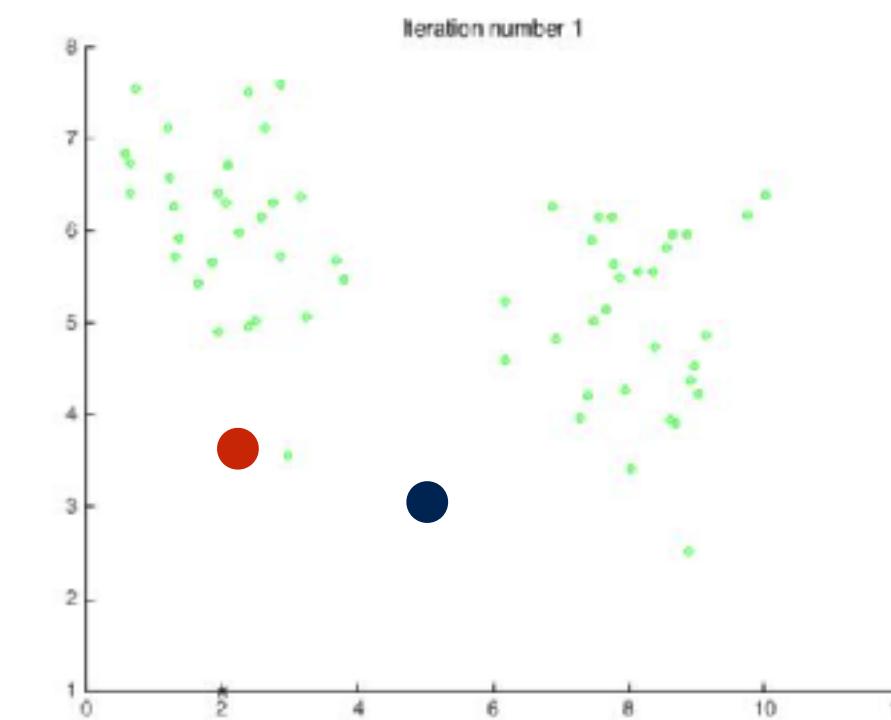
## P1.S2. K-means Clustering in Detail

❑ K-means Clustering Algorithm은 Centroids와 Cluster Assignment의 계산을 반복하여 K 개의 군집 도출

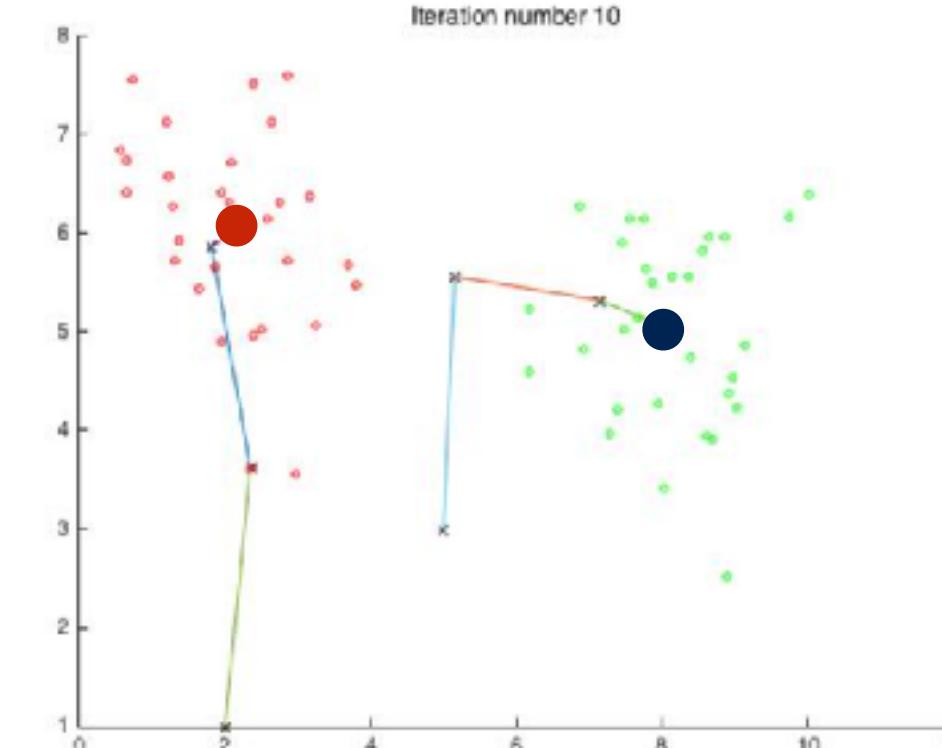
### <K-means Clustering Algorithm>



<K=2, Initialization >



<K=2, Result >



K-means Clustering은 군집 수 K를 사람이 직접 결정해야하는 문제가 있으며, 이는 Automation 관점에서 볼 때 문제가 있음

## P1.S2. K-means Clustering in Detail

- 사람의 직관에 의존하지 않고 데이터를 기반으로 군집 개수를 결정하기 위하여 Simple Heuristic\* 방식 사용
- Simple Heuristic Algorithm for Selecting K

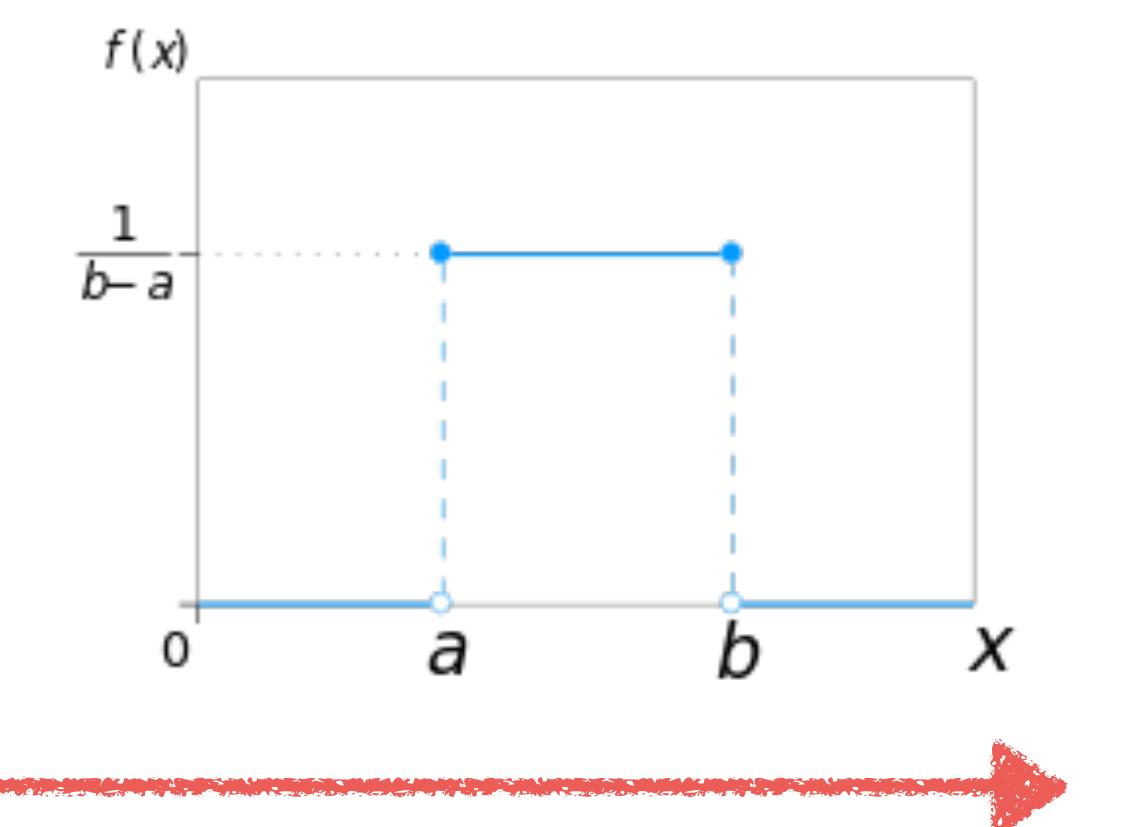
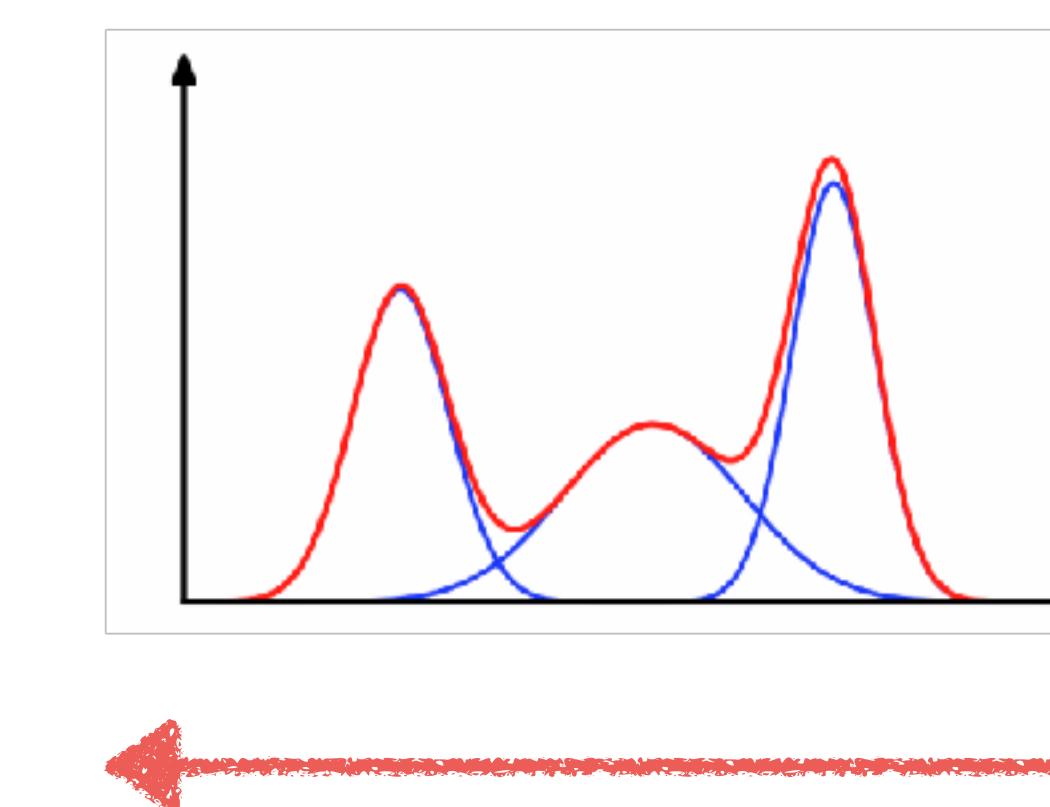
- 1)  $K=1$  부터 늘려가며  $f(K)$ 를 계산함 ( $f(K)$ , 군집 수  $K$ 에 따른 데이터 예상 왜곡 정도에 대한 실제 왜곡 정도 비율)
- 2)  $f(K)$ 값이 가장 작은  $K$ 를 선택 (단,  $f(K)<0.85$ 인  $K$ 가 없다면 1을  $K$ 값으로 사용)
  - 왜곡 정도 비율  $f(K)$ 가 클수록 Uniform Distribution에 가까우며 작을수록 부분적으로 밀집된 Distribution임

$$f(K) = \begin{cases} 1 & \text{if } K = 1 \\ \frac{S_K}{\alpha_K S_{K-1}} & \text{if } S_{K-1} \neq 0, \forall K > 1 \\ 1 & \text{if } S_{K-1} = 0, \forall K > 1 \end{cases} \quad (2)$$

$$\alpha_K = \begin{cases} 1 - \frac{3}{4N_d} & \text{if } K = 2 \text{ and } N_d > 1 \\ \alpha_{K-1} + \frac{1 - \alpha_{K-1}}{6} & \text{if } K > 2 \text{ and } N_d > 1 \end{cases} \quad (3a)$$

$$\alpha_K = \begin{cases} \alpha_{K-1} + \frac{1 - \alpha_{K-1}}{6} & \text{if } K > 2 \text{ and } N_d > 1 \end{cases} \quad (3b)$$

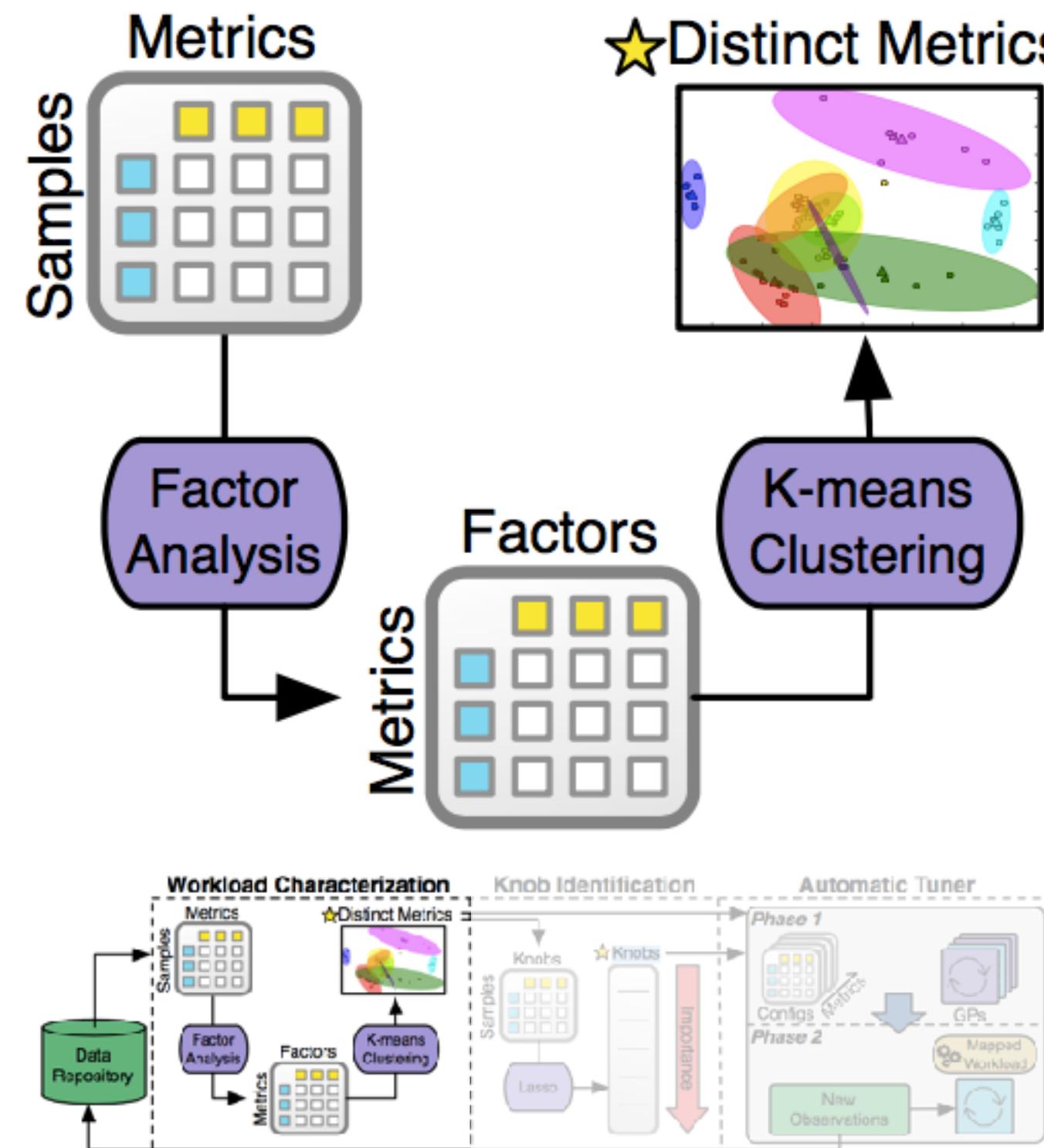
$S_K$  : 군집 수  $K$ 에 대한 왜곡 정도,  
 $\alpha_k$ : 예상 왜곡 비율,  $N_d$  : 데이터의 차원 수



# Review of Workload Characterization (P1)

## □ Workload의 특성을 파악하기 위해 FA와 K-means Clustering을 통한 대표적인 Metric을 선정

<Workload Characterization Deployment>



### Process 1: Workload Characterization

#### S1: Factor Analysis

- Configuration 설정에 따라 수집된 Metric 데이터를 대상으로 Factor Analysis (FA)를 진행
- Metric을 낮은 차원에 mapping하여 configuration에 따른 Metric 변동 데이터에 존재하던 noise 제거
- Metric 데이터를 FA를 통해 mapping된 낮은 차원의 factor 공간에서 유사도 또는 거리에 따라 K개의 군집으로 나눔
- 각 군집의 중심(centroid)와 가장 가까운 metric을 대표 metric으로 선정함

#### S2: K-means Clustering

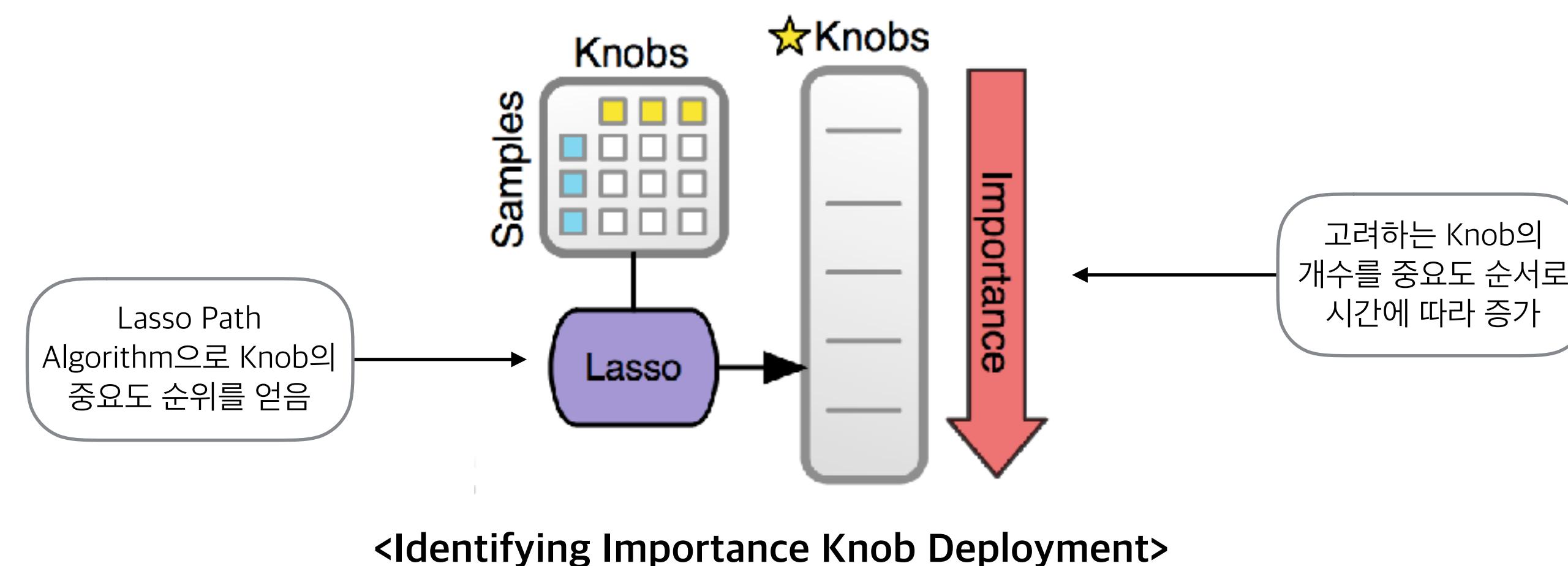
Workload의 특성 표현하고 구분하기 위해 사용할 대표 Metric 도출

Process 1 이 후, 선정된 대표 Metric을 제외한 Metric은 사용하지 않음

# Identifying Important Knob

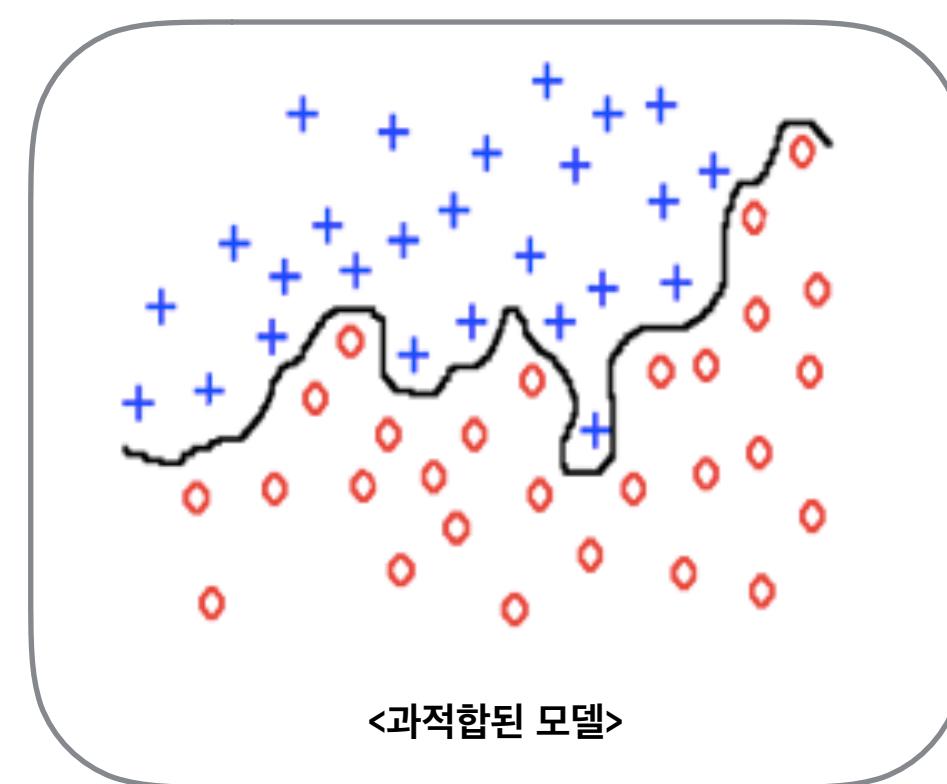
# Overview of Identifying Importance Knob (P2)

- DBMS의 전체 Knob 개수는 약 300~500개지만, 이들 중 **실질적으로 영향을 미치는 Knob**는 소수임
  - 가장 영향이 큰 Knob들만을 선택하여 Configuration 공간의 크기를 줄이고 계산 비용을 낮출 수 있음
  - Lasso Path Algorithm을 이용해 각 Knob의 중요도 순위를 알아낼 수 있음
- Knob들이 서로 **독립적이지 않고 Target Metric과 비선형** 관계를 가질 수 있음
  - Multiplication Term과 Square Term을 Input Feature로 사용하여 모델링



# Lasso Regression

- Regression을 이용해 Knob값으로 Target Metric의 값을 계산/예측
- 트레이닝 데이터에 대한 과적합(Overfitting)이 발생하여 실제 데이터에 대한 예측력이 떨어질 수 있음
- 과적합 모델은 절대값이 큰 가중치를 수반하므로 비용함수에 가중치의 절대값(Lasso Term)을 추가하여 과적합을 방지함
  - 비례상수  $\lambda$ 가 클수록 가중치의 절댓값이 작아짐

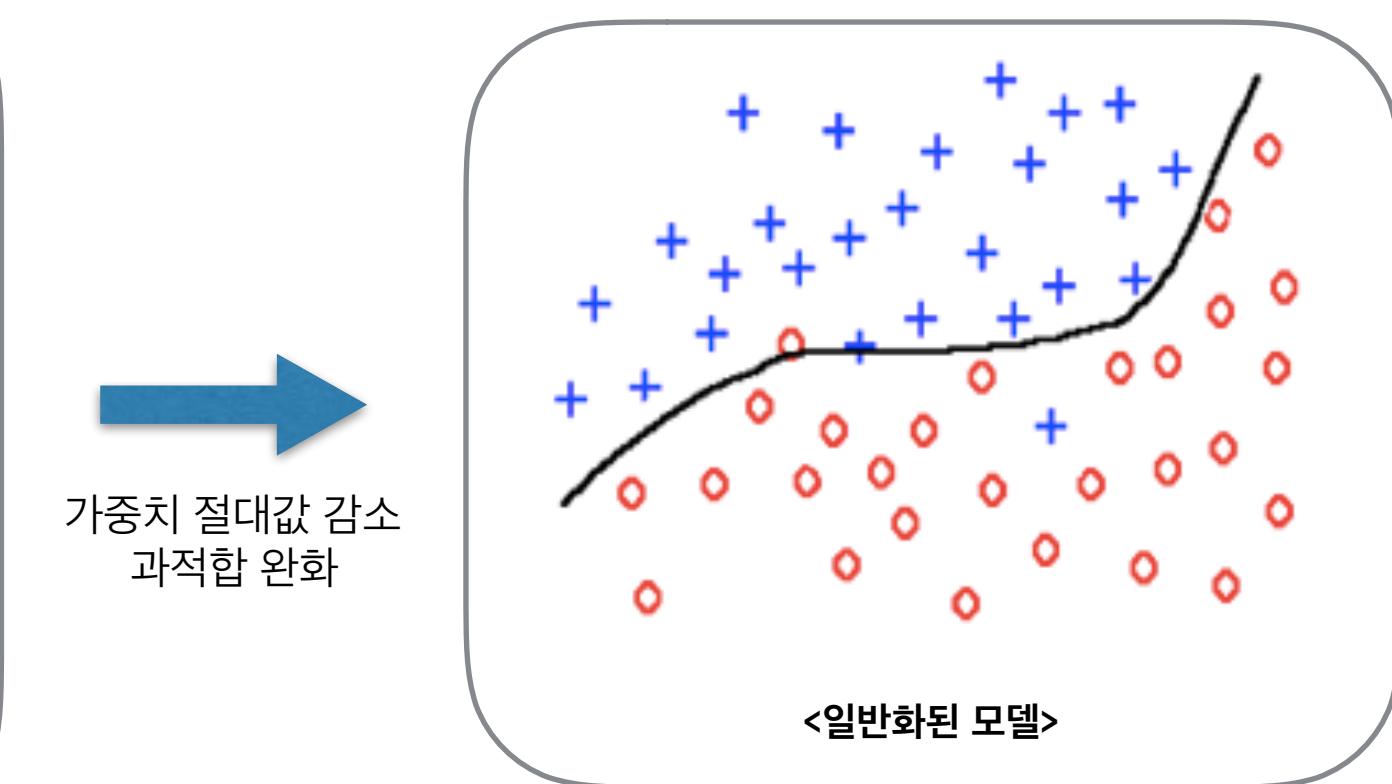


비용함수에  
Lasso Term 추가

$$\begin{aligned}\mathcal{L}(\beta; \lambda) &= \|\mathbf{Y} - \mathbf{X}\beta\|_2^2 + \lambda_1 \|\beta\|_1 \\ &= \sum_{i=1}^n (Y_i - \mathbf{X}_{i*}\beta)^2 + \lambda_1 \sum_{j=1}^p |\beta_j|\end{aligned}$$

↓                      ↓  
Sum of Squared Error Term    Lasso Term

<과적합 방지를 위해 Lasso Term이 추가된 비용함수>



가중치 절대값 감소  
과적합 완화

# Lasso Path Algorithm

□ **λ값이 클 수록 가중치의 절대값이 작아지는 성질을 이용하여 큰 영향을 미치는 Knob를 찾는 것이 목적임**

<λ와 가중치, 변수의 중요도의 관계>

$$\begin{aligned}\mathcal{L}(\beta; \lambda) &= \|\mathbf{Y} - \mathbf{X}\beta\|_2^2 + \lambda \|\beta\|_1 \\ &= \sum_{i=1}^n (Y_i - \mathbf{X}_{i*}\beta)^2 + \lambda \sum_{j=1}^p |\beta_j|\end{aligned}$$

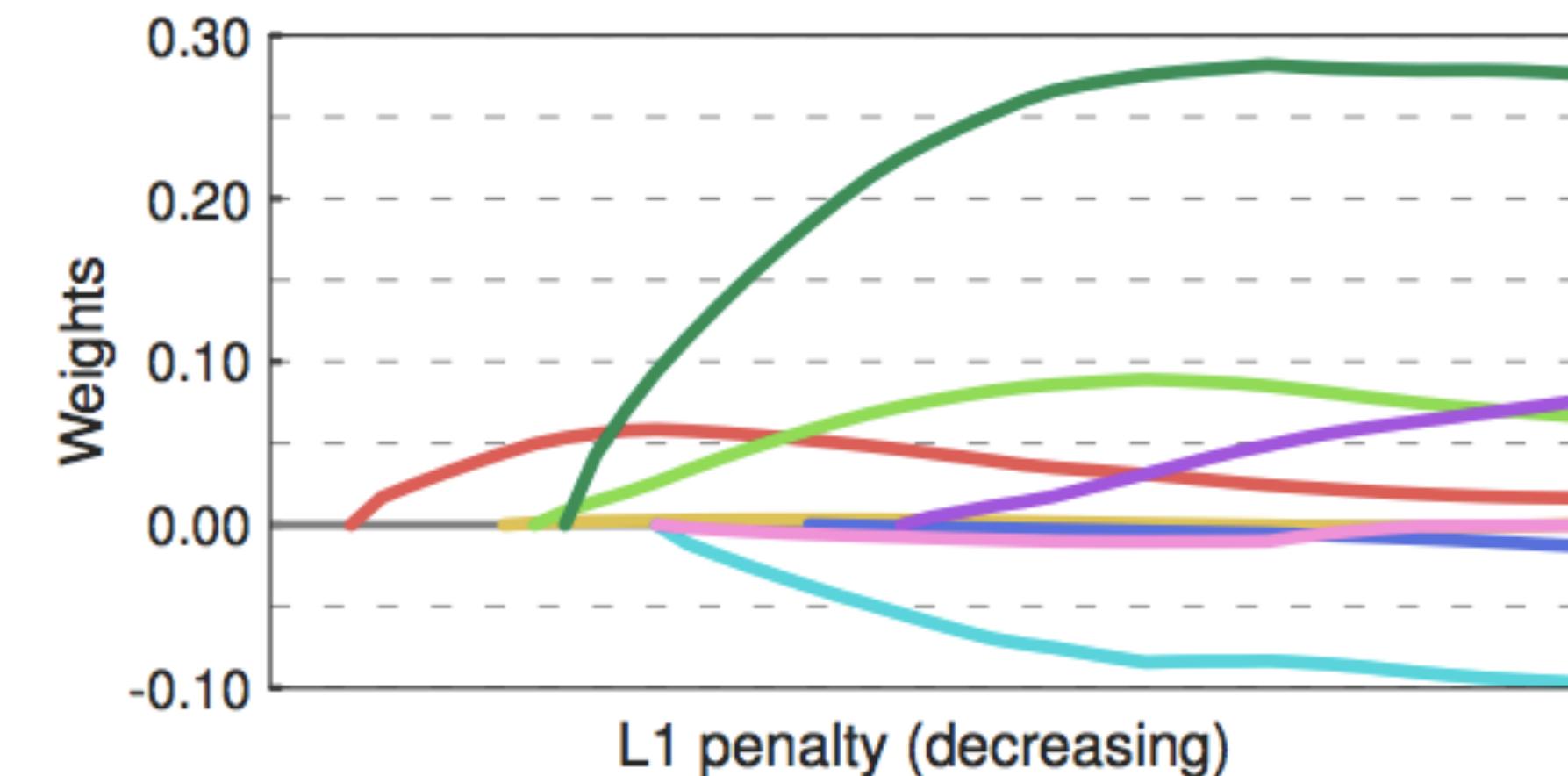
Sum of Squared Error Term

Lasso Term

1. λ가 아주 큰 경우 모든 가중치가 0일 때 비용 함수가 가장 작아짐
2. λ를 감소시키면 Y값을 예측할 때 가장 중요한 역할을 하는 변수의 가중치부터 0을 벗어남
3. 가중치가 먼저 0을 벗어나는 순서로 변수의 영향도 순위를 매김

실험 결과

<λ의 감소에 따른 각 변수의 계수의 변화>



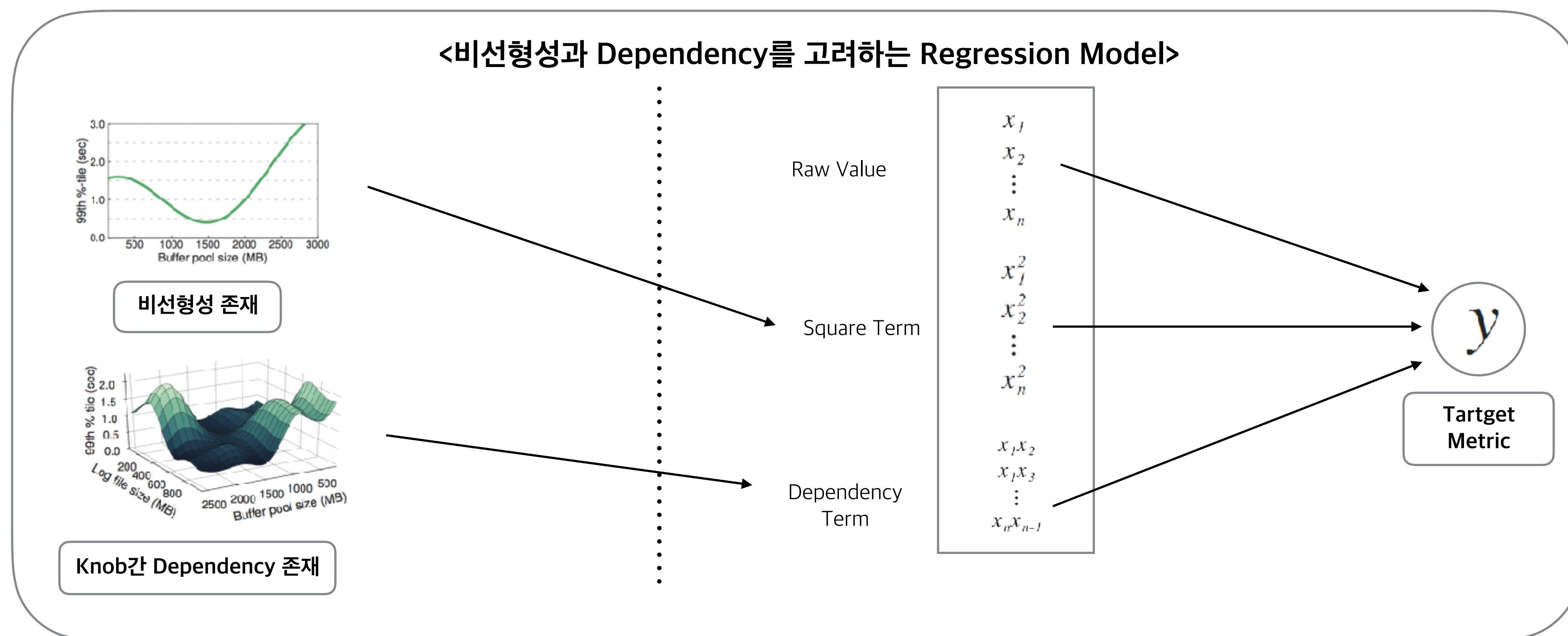
# Nonlinearity and Dependencies

## ❑ Knob값과 Target Metric이 비선형 관계를 가짐

- Knob값의 Polynomial을 Feature로 사용하여 비선형 관계를 모델링 할 수 있음
- Square Term을 새로운 Feature로 사용

## ❑ Knob들이 서로 독립적이지 않음

- 특정 Knob값의 조합은 배제되어야 함 (e.g. 각 Knob의 메모리 할당량의 합이 시스템 메모리의 크기를 초과하는 경우)
- Knob값들의 곱을 새로운 Feature로 사용하여 Dependency를 고려할 수 있음



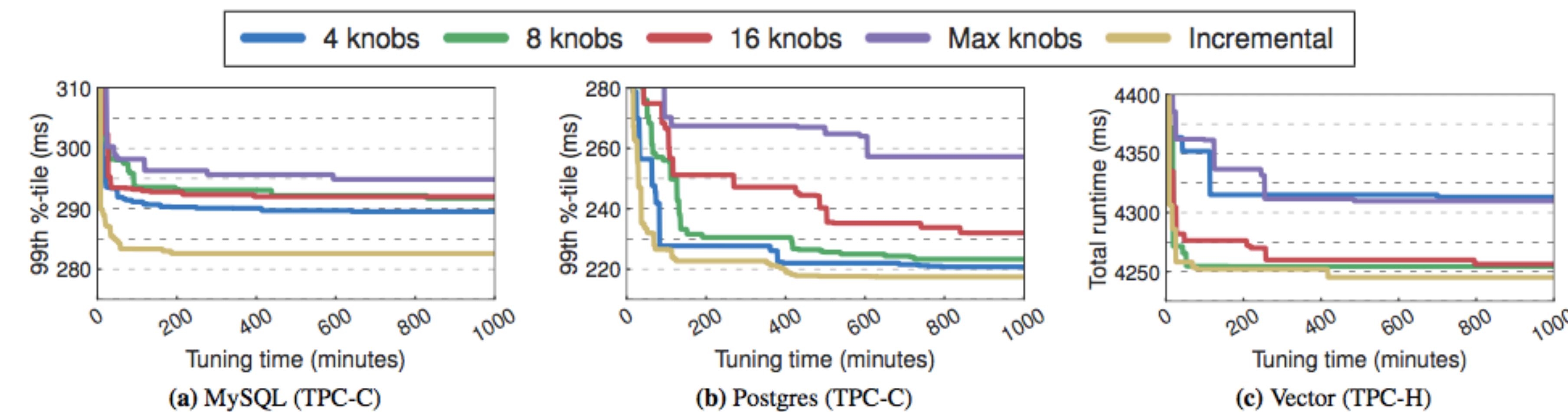
# Incremental Knob Selection

## ❑ Knob개수 결정법에는 크게 두가지 방법이 있음

- Fixed number of knob : 고정된 Knob 개수를 사용하는 방법. Knob 개수가 너무 많으면 모델의 복잡도가 증가하여 투닝 시간이 길어지고 너무 적으면 좋은 성능의 Configuration을 찾을 수 없음
- Incremental Knob Selection : Knob의 개수를 시간에 따라 증가시키는 방법. 중요한 Knob에 대한 Tuning을 우선적으로 수행하고 이후 Fine Tuning을 진행할 수 있음

## ❑ Knob의 개수를 시간에 따라 증가시킬 때 Recommendation의 성능이 가장 좋음

<Fixed number, incremental knob selection 성능 비교>



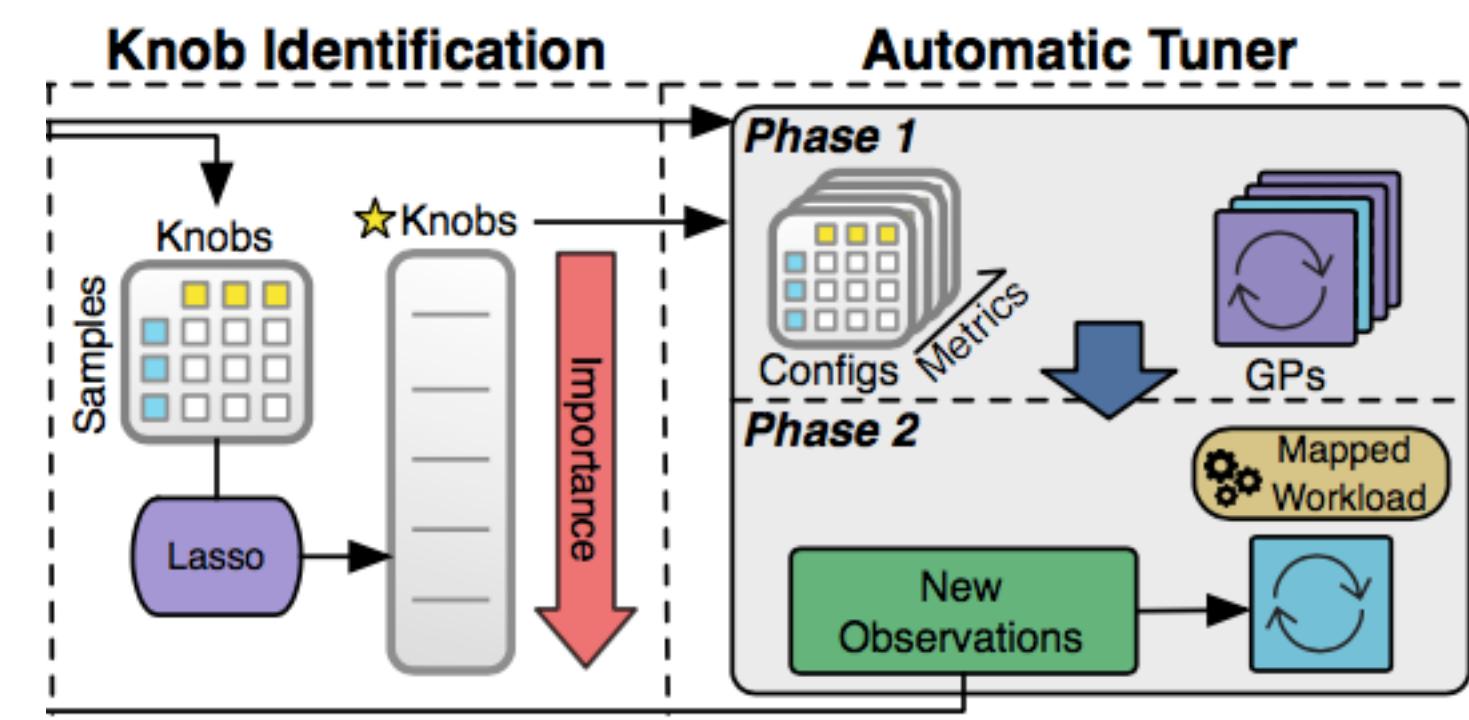
# Summary

## □ 수백개의 Knob중 영향력이 큰 Knob들만을 선택적으로 고려하여 모델의 Complexity를 줄임

- Lasso Path Algorithm을 사용하여 Knob의 중요도 순위를 매김
- 고려하는 Knob의 개수를 시간에 따라 늘릴 때 성능이 가장 좋음
- Square Term과 Multiplication Term을 Input Feature로 추가하여 비선형성과 Knob간 의존성을 반영함

## □ Process 3와의 연계

- 이 과정에서 추려진 적은 개수의 Knob에 대해 GP regression을 수행하고, 그 결과를 Knob의 중요도에 따라 DBMS에 설치함
- GP regression의 계산비용이 줄어들고 성능이 향상됨

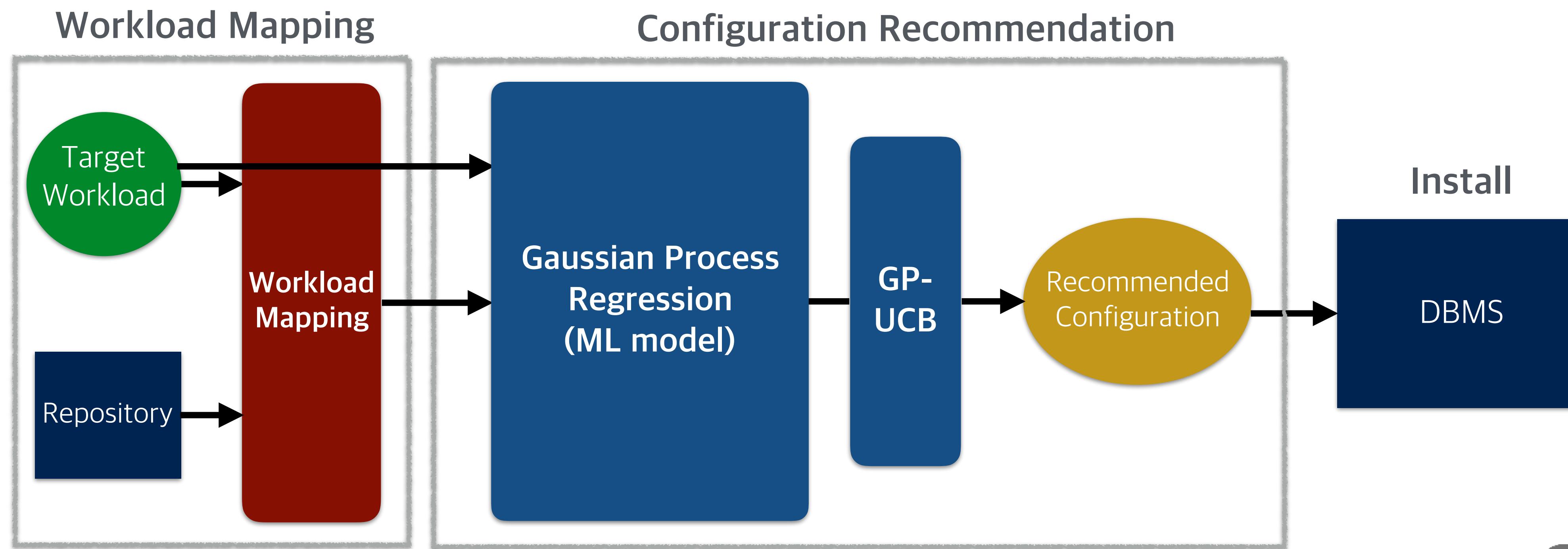


<Knob Identification과정에서 선택된 Knob가 Automatic Tuner에서 사용됨>

# Automated Tuning

# Overview

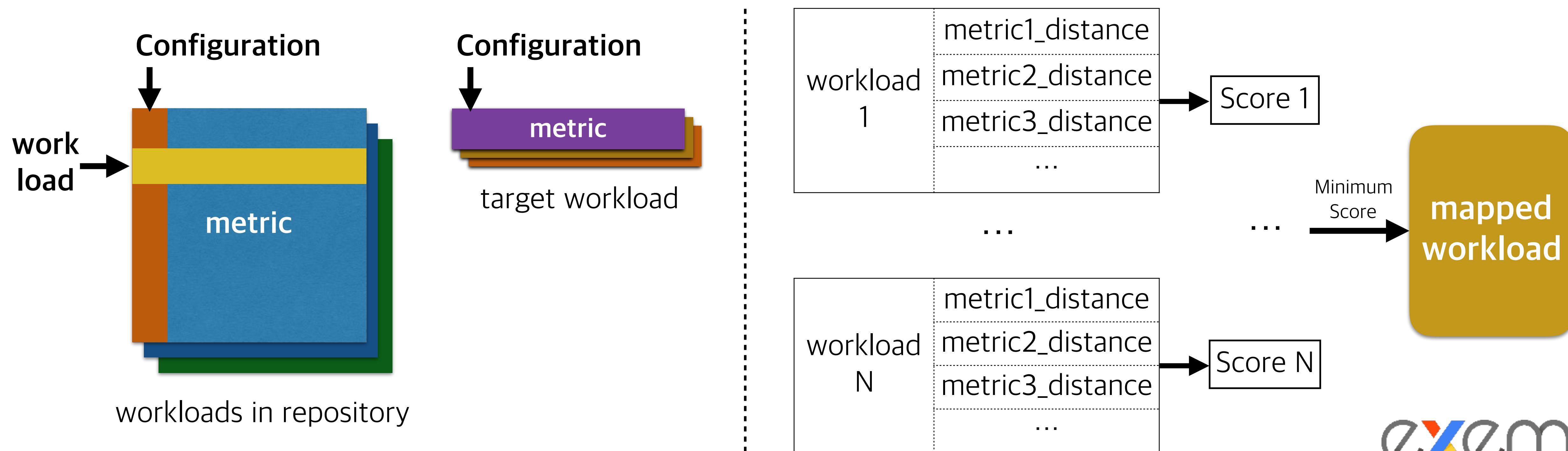
- Automated tuning에서는 previous tuning session을 거친 데이터를 사용함.
- 유클리디안 거리를 이용해 repository에서 target workload와 knob setting에 비슷한 반응을 보이는 workload를 찾음.
- 유사한 workload와 target workload를 Gaussian Process Regression(GPR)모델로 학습하고 target metric을 높이는 configuration을 예측함.



# Workload Mapping

# Workload mapping

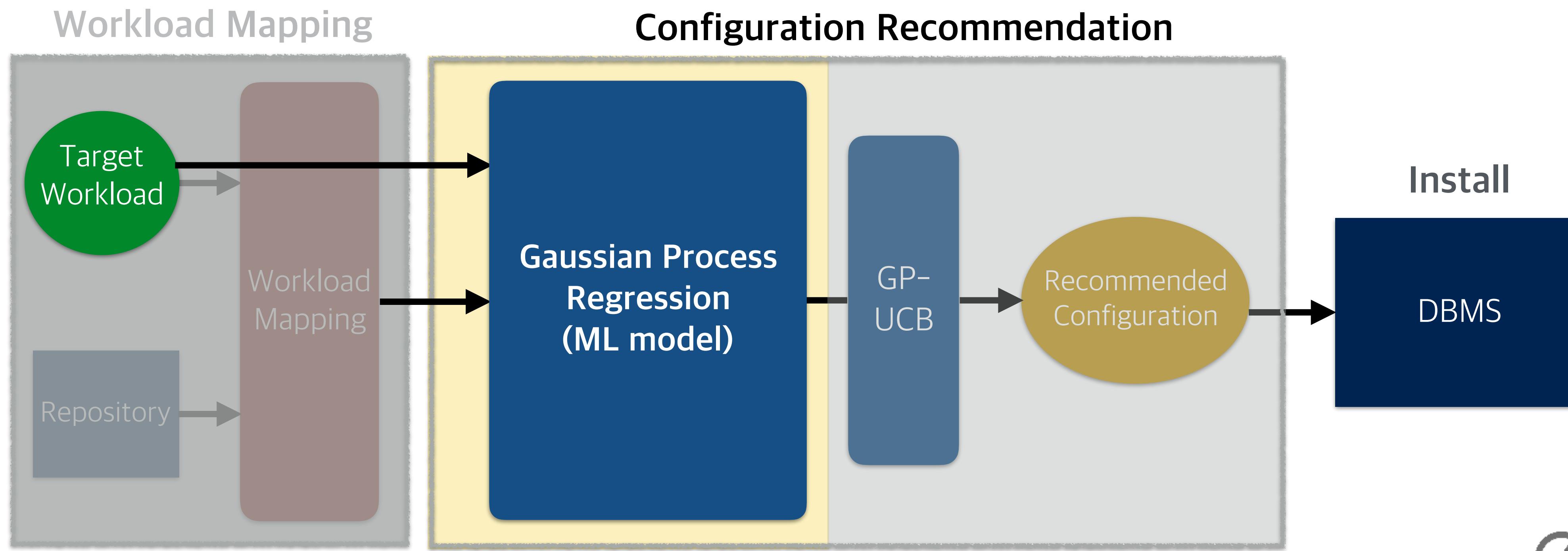
- 다른 knob setting에도 비슷하게 반응하는 workload를 이용하여 머신러닝 모델을 학습하면 metric값 예측 정확도가 높음.
  - 유클리디안 거리 계산을 통해 타겟 workload와 가장 유사한 workload를 찾을 수 있음.
    - workload마다 metric행렬에 대해 target workload와의 거리를 계산하고, 거리들의 평균을 workload의 score로 함.
    - score가 가장 낮은 workload를 ML model에 사용함.
    - workload가 특정 metric 행렬에 의해 결정되지 않도록 metric별 유클리디안 거리 스케일을 조정한 뒤 scoring함.



# Configuration Recommendation

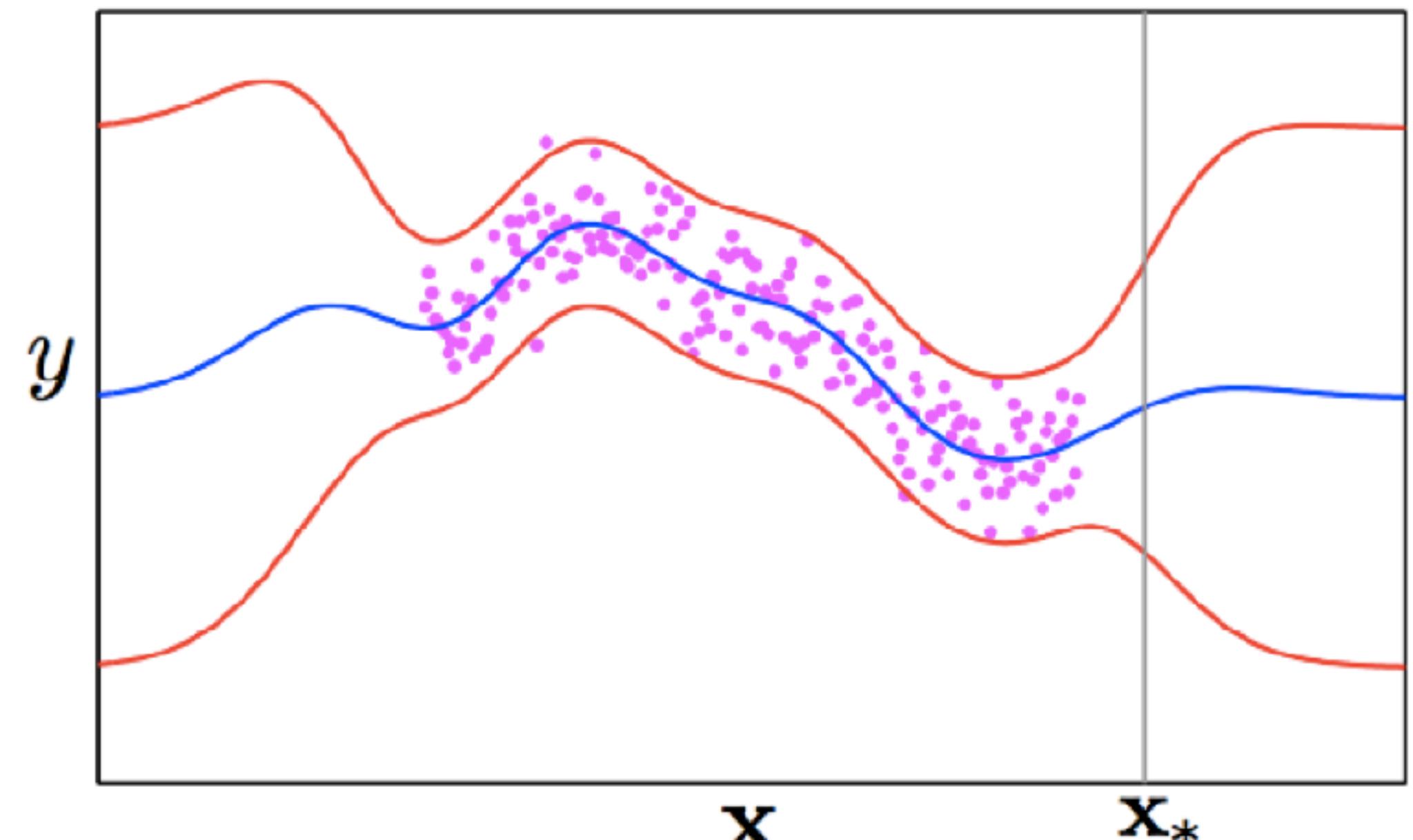
# Configuration Recommendation

- target metric을 높이는 다음 configuration을 예측하기 위해 유사한 workload와 target workload를 Gaussian Process Regression(GPR)모델로 학습함.
- exploration-exploitation 균형을 고려하는 Gaussian Process Upper Confidence Bound(GP-UCB)식을 통해 configuration을 예측함.
- 추천 Configuration을 DBMS에 install하고 다음 observation period를 진행하여 예측 값의 결과를 GPR 모델에 반영 함.



# Gaussian Process

- Gaussian Process (GP)는 모든 부분 집합이 결합 가우시안 분포를 가지는 랜덤 변수의 집합.
- GP는 input space  $X$ 를  $\mathbb{R}$ 로 매핑시키는 함수  $f$  들에 대한 distribution으로  $x$ 를 index로 갖는 랜덤변수  $f(x)$ 의 집합이며 정의역이 무한 차원일 때 사용함.  $x \in D, f: D \rightarrow \mathbb{R}$ 
  - 정의역의 차원이 유한할 때 함수의 확률적 분포는 multivariate Gaussian distribution으로 나타낼 수 있음. 유한 개의 데이터를 선형 결합하여 무한 차원으로 확장할 수 있음.
- 평균 함수와 공분산 함수로 모델을 나타낼 수 있음.



Gaussian Process Regression

Blue line : 평균 함수  
Red line : 공분산 범위  
 $x^*$  : 새로운 데이터 포인트

# Gaussian Process Regression

- Gaussian Process Regression(GPR)은 GP를 따르는 비선형 회귀 모델이며 주어진 데이터 범위에서 벗어난 구간의 예측 값 추정에 유리함.
  - GP를 따르기 때문에 평균 함수와 공분산 함수를 이용해 모델을 정의할 수 있음.
  - 평균 함수는  $x$ 에 대한 예측 값  $y$ 를 의미하고 공분산 함수는 예측의 uncertainty를 의미함.
  - 공분산 함수는 커널 함수로 나타내고 학습 데이터의 종류에 따라 커널 함수를 정의할 수 있음.
- 기존 공분산 행렬에 새로운 데이터에 대한 공분산 행렬을 덧붙여 모델의 공분산 행렬을 나타낼 수 있음. 즉, 새로운 데이터를 재학습 과정 없이 모델에 반영할 수 있음.

공분산 행렬  $K =$

$$\begin{matrix} K' & K^{*^T} \\ K^* & K^{**} \end{matrix}$$

$x'$  : new data

$K'$  : 기존 공분산 함수.  $[K(x_1, x_1), \dots, K(x_1, x_N), \dots, K(x_N, x_N)]$

$K^*$  :  $[K(x_1, x'), K(x_2, x'), \dots, K(x_N, x')]$

$K^{**}$  :  $[K(x', x')]$

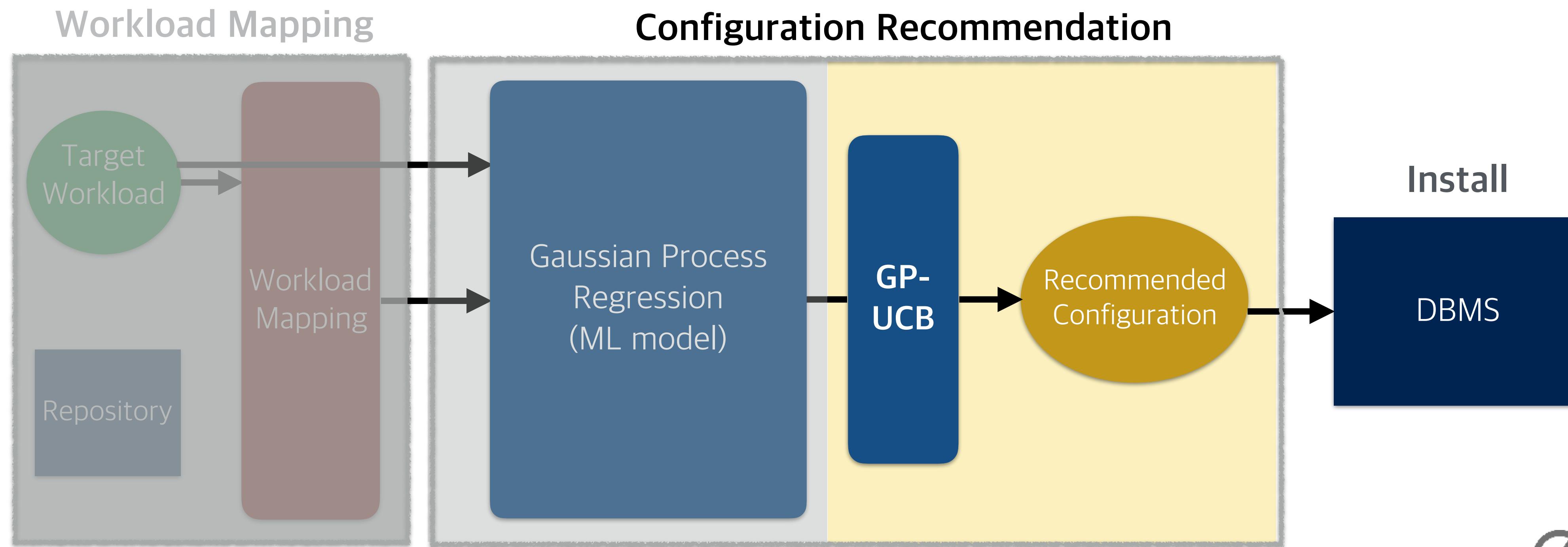
□ OtterTune은 ridge term이 추가된 squared exponential 커널 함수를 따르는 GPR모델을 학습함.

- 독립 변수가 configuration이고 종속 변수는 target metric인 GPR 모델을 정의함.
- OtterTune에서 커널함수는 squared exponential kernel로 정의됨.
- 유사 workload와 target workload의 차이로 인한 noise를 상쇄하기 위해 커널 함수에 ridge term을 더함.

OtterTune's GPR 모델	
랜덤 변수 f의 분포	$f \sim GP(\mu(x), \sigma(x)) = GP(\mu(x), K_T(x, x))$
공분산함수	$\sigma_T^2(x) = K_T(x, x)$
평균함수	$\mu_T(x) = K_T(x)^T(K_T + \sigma_2 I)^{-1}y_T$
	$K_T(x, x') = K(x, x) - K_T(x)^T(K_T + \sigma^2 I)^{-1}K_T(x')$
커널함수 (Squared Exponential)	$K(x, x') = \sigma_f^2 \exp\left[\frac{-(x - x')^2}{2l^2}\right] + (\text{ridge term})$

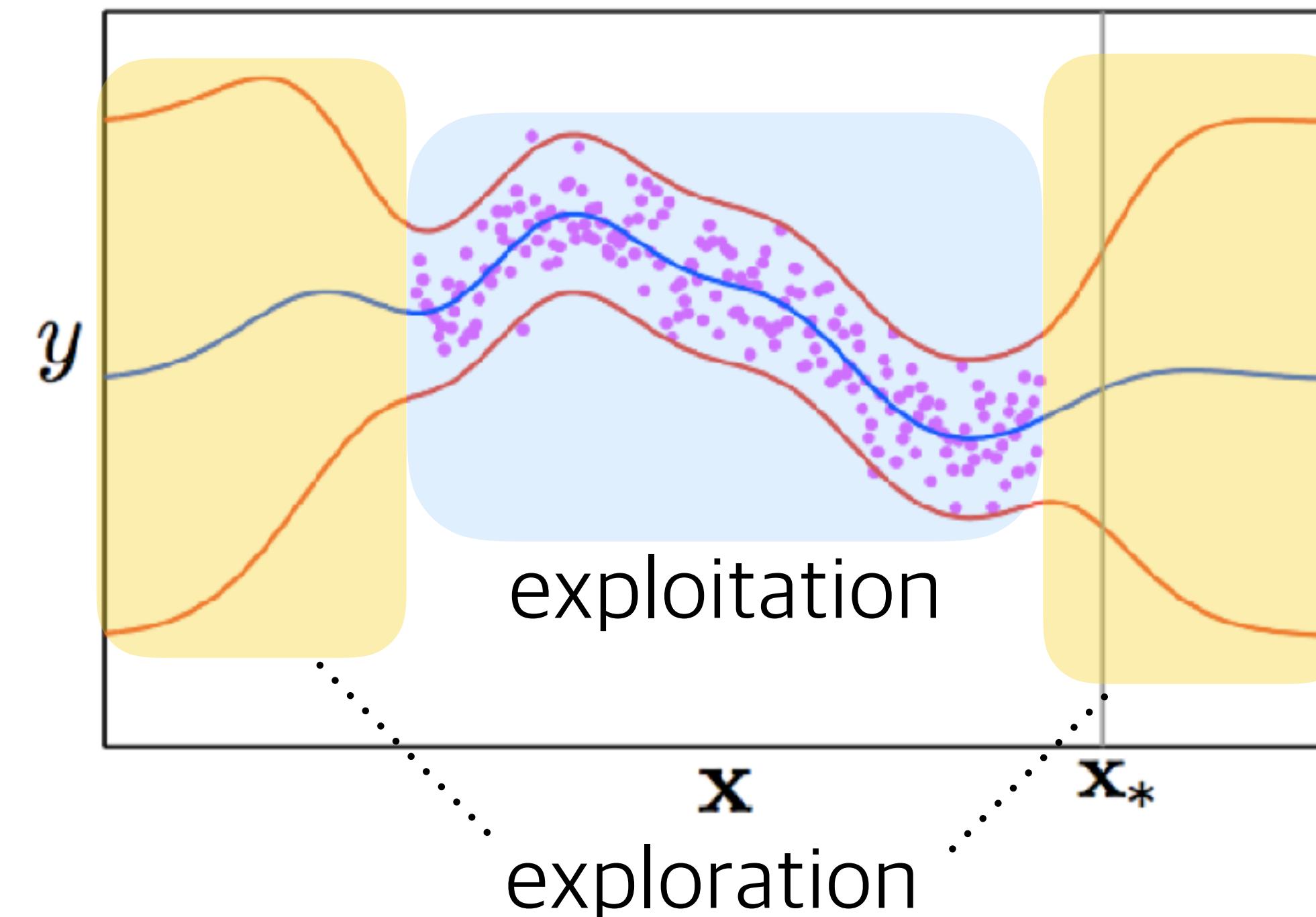
# Configuration Recommendation

- target metric을 높이는 다음 configuration을 예측하기 위해 유사한 workload와 target workload를 Gaussian Process Regression(GPR)모델로 학습함.
- exploration-exploitation 균형을 고려하는 **Gaussian Process Upper Confidence Bound(GP-UCB)**식을 통해 configuration을 예측함.
- 추천 Configuration을 DBMS에 install하고 다음 observation period를 진행하여 예측 값의 결과를 GPR 모델에 반영 함.



# Exploration-Exploitation

- GPR는 탐색 구간에서 예측하는 것(exploitation)과 탐색되지 않은 구간에서 예측하는 것(exploration)으로 나눌 수 있음.
- exploration-exploitation간 균형을 고려한 반복적인 x값(OtterTune's configuration) 추정으로 높은 성능(OtterTune's metric)을 내는 모델로 확장할 수 있음.
  - exploration은 모델의 최댓값을 높일 수 있으나 최솟값도 낮아질 확률이 있음.
  - exploitation은 안정적인 예측 결과를 보장하지만 최소값~최대값 범위에서 벗어날 수 없음.



# Gaussian Process Upper Confidence Bound

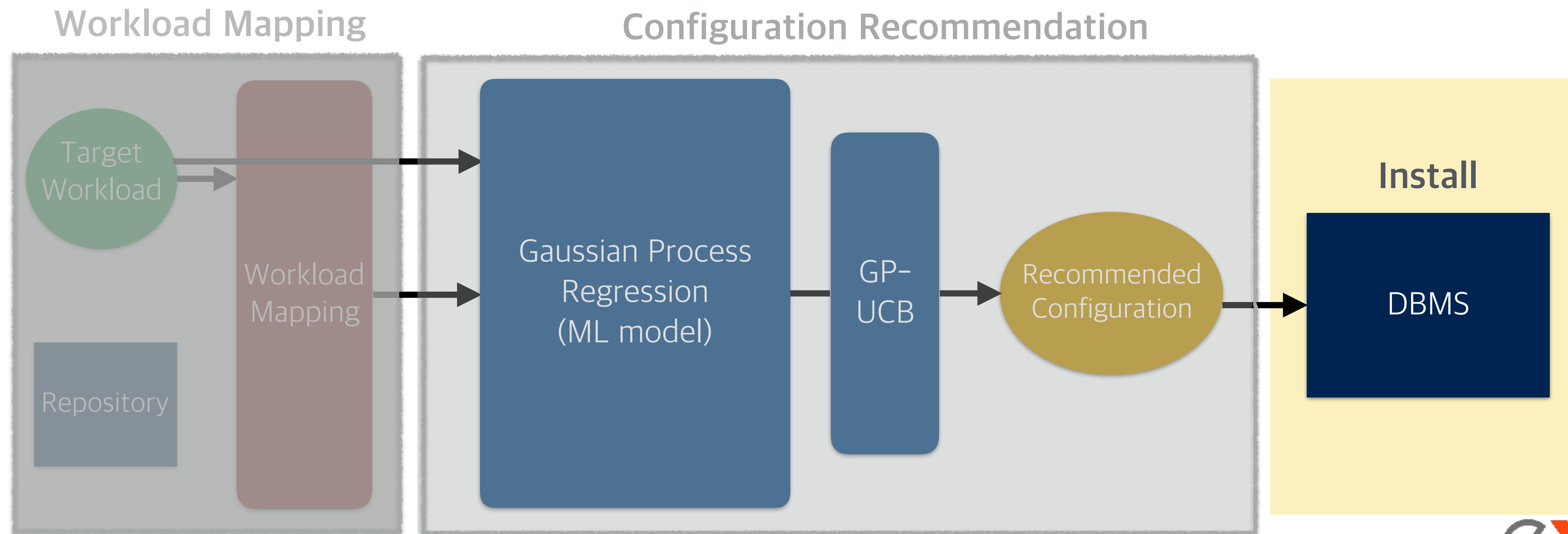
- Gaussian Process Upper Confidence Bound (GP-UCB)은 exploration과 exploitation의 균형을 유지하는 목적함수를 정의하고 이를 최대화하는 새로운 독립변수  $x$ (OtterTune's configuration)를 반복적으로 선택하여 모델을 확장함.
  - exploitation구간에 존재하는  $x$ 는 평균 값이 높고 exploration구간에 존재하는  $x$ 는 공분산 값이 높음.
  - 목적함수는 (평균 함수)와 (공분산 함수)의 합으로 정의하되 confidence parameter인 베타를 곱하여 exploration-exploitation 균형을 유지함.
  - 커널 함수 종류에 따라 confidence parameter 베타 수식이 결정됨.
  - (추후 추가할 부분)베타 값을 어떻게 저 수식으로 가정하는지, UCB의 진정한 원리가 뭔지 @@원리에 confidence의 upper bound를 이용해하는 부분 이해 후 설명 추가할 것.
  - OtterTune에서 베타는 squared exponential 커널에 따라 아래 수식과 같이 결정됨.

$$\text{choose } x_t = \operatorname{argmax}_{x \in D} \mu_{t-1}(x) + \beta_t^{1/2} \sigma_{t-1}(x)$$

$$\beta_t = 2 \log(|D| t^2 \pi^2 / 6\delta) \quad \delta \in (0, 1)$$

# Configuration Recommendation

- target metric을 높이는 다음 configuration을 예측하기 위해 유사한 workload와 target workload를 Gaussian Process Regression(GPR)모델로 학습함.
- exploration-exploitation 균형을 고려하는 Gaussian Process Upper Confidence Bound(GP-UCB)식을 통해 configuration을 예측함.
- 추천 Configuration을 DBMS에 install하고 다음 observation period를 진행하여 예측 값의 결과를 GPR 모델에 반영 함.



# Experiment Result

## Developmental and experimental environment

- ❑ 구현 환경 : Google Tensorflow, python's scikit-learn
- ❑ 사용 DBMS : MySQL (v5.6), Postgres (v9.3), Action Vector (v4.2)
- ❑ 실험 환경 : clouding virtual machine인 Amazon EC2. 두 개의 instances가 존재함.
  - OtterTune's controller : OLTP-Bench framework. m4.large instance (4 vCPUs and 16GB RAM)
  - target DBMS : m3.xlarge instance (4 vCPUS and 15GB RAM)
- ❑ OtterTune's tuning manager and repository는 local server에 존재. (20 cores and 128GB RAM)
- ❑ Observation period : 5분.
- ❑ target metric은 99%-tile latency로 할당.

## Workload Experiment List

### ❑ YCSB (The Yahoo! Cloud Serving Benchmark) : 데이터 매니지먼트 어플리케이션

- simple workload, high scalability requirements.
- 데이터베이스에 10개의 attribute를 가진 1개 테이블이 존재함.
- 18m tuples (~18GB) 데이터베이스 사용.

### ❑ TPC-C : 주문 프로세스 어플리케이션

- 데이터베이스에 9개 테이블과 5개의 transactions 존재
- 200 warehouses (~18GB) 데이터베이스 사용.

### ❑ Wikipedia : 온라인 백과사전

- 데이터베이스에 11개 테이블과 8개 서로 다른 transaction 타입 존재.
- 100k개 기사 (~20GB) 데이터베이스 사용.

### ❑ TPC-H : OLAP 환경에서 시뮬레이트되는 decision support system

- 데이터베이스에 8개 테이블, 3NF schema, 22 queries.
- scale factor of 10 in each experiment (~10GB)

## Training Data Collection

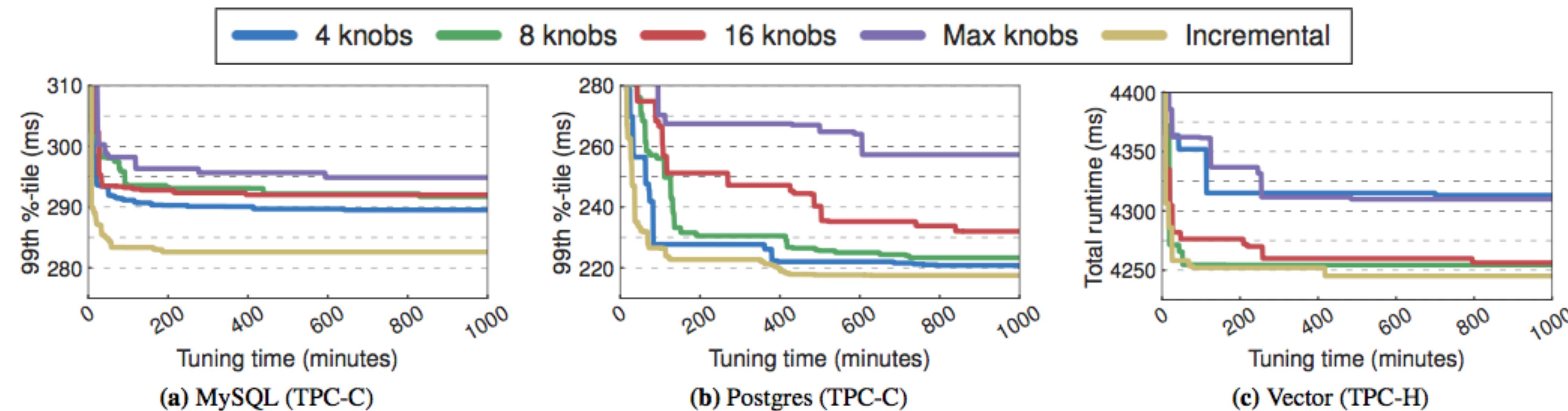
- ❑ 모든 workload를 run하기보다 YCSB와 TPC-H의 순열로 workload를 실행함.
- ❑ YCSB는 여러 workload mixture로 15가지 변형을 만들고, TPC-H는 쿼리를 각 workload를 대표하는 네 그룹으로 나눔.
- ❑ 여러 knob configuration에 대해 실험을 진행해야하므로 각 workload별로 knob parameter들을 랜덤 값으로 설정하고 물리적 용량을 넘는 값에 대해서 따로 조정함. (실제로는 물리적 용량을 넘는 경우 DBMS가 실행될 수 없고 OtterTune도 데이터를 수집할 수 없으므로 이런 경우를 고려하지 않음.)
- ❑ DBMS당 여러 workload와 knob configuration으로 총 30k번 이상의 시도를 함. 여러 번의 시도는 한 번의 observation period로 간주됨.
- ❑ 시스템은 DBMS로부터 외부 metrics(ex. throughput, latency)와 내부 metrics(ex. pages read/written)를 수집함.
- ❑ 이전 실험 결과 데이터에 영향을 받지 않기 위해 각 실험마다 학습 데이터를 load한 뒤 OtterTune's repository를 초기 설정으로 되돌림.

# Experimental setup of Number of Knobs

- ❑ Knob 갯수가 고정되었을 때(Fixed number of knob)와 점점 증가할 때(Incremental number of knob)의 성능을 비교함.
  - 실험 1은 knob 4개, 8개, 16개, max개로 구성.
  - 실험 2는 4개 knob로 시작하여 60분마다 knob 갯수 2개 증가(incremental).
  - top-k개의 knob만 사용.
  - 15시간 tuning session 진행.
- ❑ TPC-C benchmark for the OLTP DBMS (MySQL and Postgres)
  - 설명1 특성을 중심으로
- ❑ TPC-H benchmark for the OLAP DBMS (Vector)
  - 설명2 특성을 중심으로

## Number of Knobs

- ❑ tuning session동안 OtterTune이 생성하는 여러 configuration 이용해 DBMS을 성능 비교한 결과 incremental knob가 우세한 결과를 보임.
  - MySQL (TPC-C) : Incremental number of knob가 성능, 최적 성능 도달 속도 측면에서 우세.
  - Postgres (TPC-C) : Incremental number of knob가 최적 성능으로 수렴하는 속도 측면에서 우세하나 최고 성능은 4 knobs, 8knobs와 크게 차이 없음.
  - Vector (TPC-H) : Incremental number of knob가 최적 성능으로 수렴하는 속도 측면에서 우세하나 최고 성능은 8 knobs, 16knobs와 크게 차이 없음.
- ❑ DBMS 종류에 따라 최고 성능을 내는 knobs의 갯수가 다르지만 Incremental knobs 방법은 DBMS 종류에 관계 없이 빠른 수렴 속도로 최적 성능을 냄.



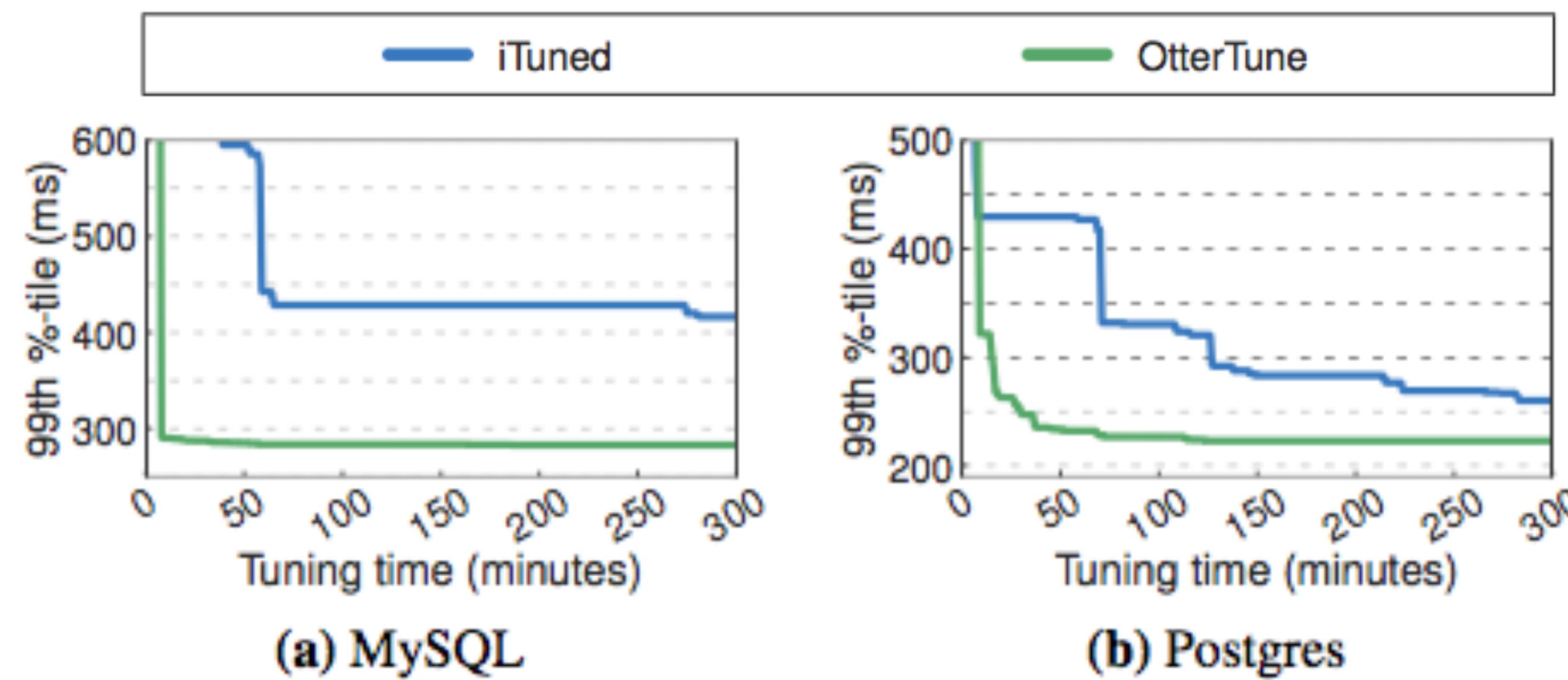
## Experimental setup of Tuning evaluation

- ❑ OtterTune의 tuning session이 DBMS knob configuration을 찾는데 얼마나 영향을 주는지 실험.
- ❑ 다른 tuning tool인 iTuned와 OtterTune을 비교함.
  - OtterTune은 incremental knob 방식으로 tuning session 진행 후 가장 유사한 workload로 GP모델 학습.
  - iTuned는 stochastic sampling을 이용하여 초기 10 DBMS configuration 집합을 생성하여 GP모델 학습.
- ❑ TPC-C, Wikipedia benchmarks for the OLTP DBMS (MySQL and Postres)
- ❑ TPC-H workload for the OLAP DBMS (Vector)

# OtterTune vs iTuned (TPC-C workload)

## TPC-C workload를 이용한 OtterTune과 iTuned 비교 실험은 OtterTune이 우세한 결과를 보임.

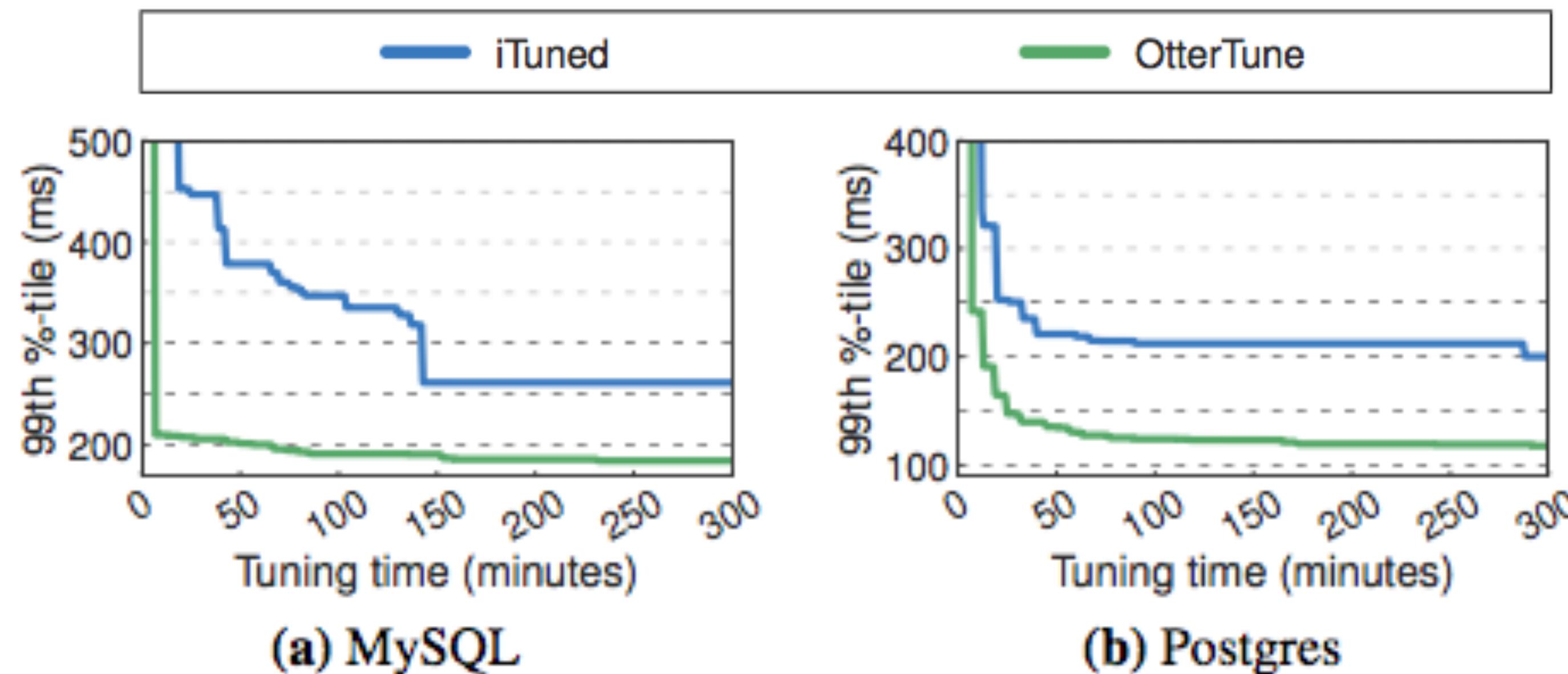
- better configuration을 찾는 시간 : OtterTune 30분 (MySQL), 45분 (Postgres) / iTuned : 60-120분
- MySQL에서 OtterTune이 iTuned보다 85% 낮은 latency를 기록.
- Postgres에서 OtterTune이 iTuned보다 75% 낮은 latency를 기록.
- iTuned는 여러 knob들간 균형을 찾지 못하나 OtterTune은 많은 데이터로 GP를 학습하기 때문에 configuration space에 대한 이해도가 높아서 knob들 간 균형을 찾을 수 있음.



# OtterTune vs iTuned (Wikipedia workload)

■ Wikipedia workload를 이용한 OtterTune과 iTuned 비교 실험은 OtterTune이 우세한 결과를 보임.

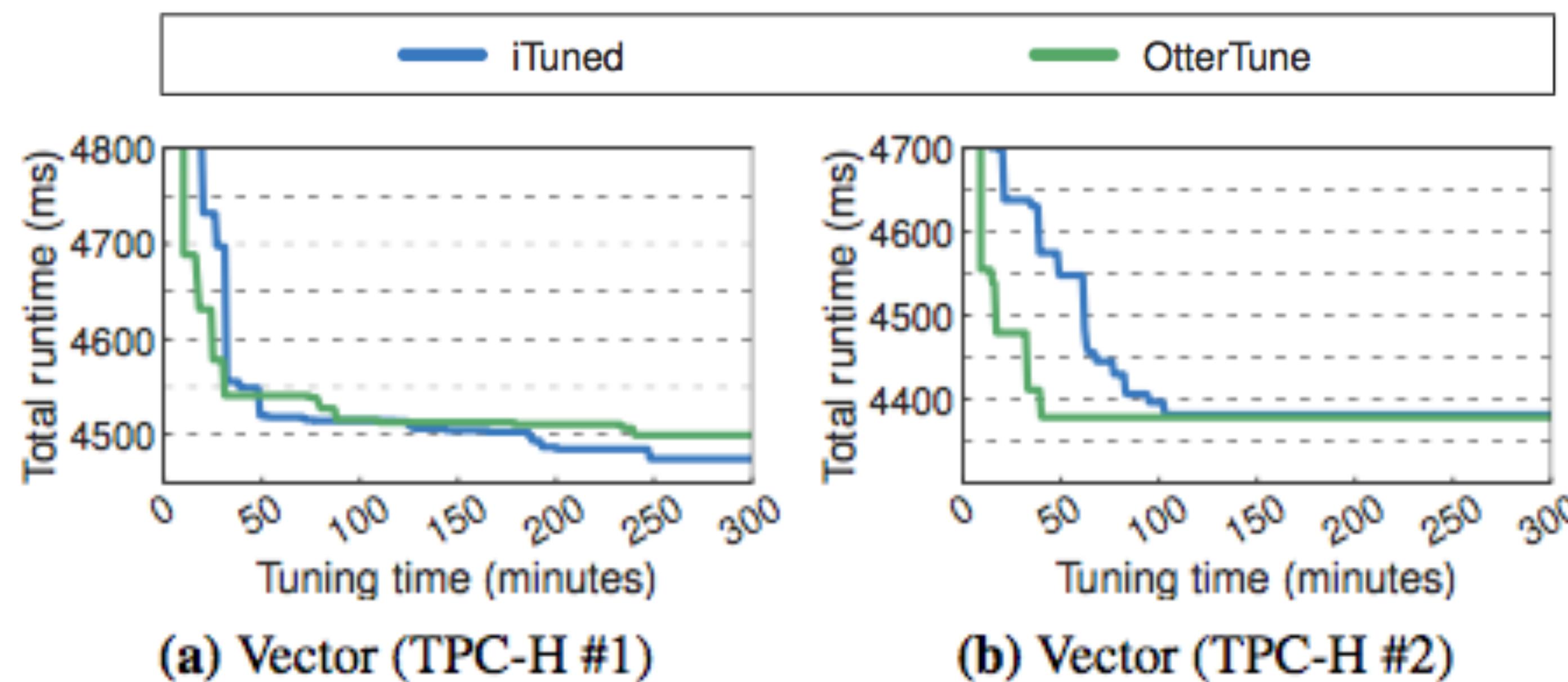
- better configuration을 찾는 시간 : OtterTune 15분 (MySQL), 100분 (Postgres) / iTuned : failed



## OtterTune vs iTuned (TPC-H workload)

□ TPC-H workload를 이용한 OtterTune과 iTuned 비교 실험은 차이가 극명하지 않음.

- Vector는 tuning할 수 있는 knob가 제한적이기 때문에 OLTP workload실험과 다르게 OtterTune과 iTuned의 차이가 극명하지 않음.



## 4 categories of execution time

- Execution time은 OtterTune의 과정에 따라 네 단계로 구분할 수 있음.

- Workload Execution : DBMS가 new metric 데이터를 수집하기 위해 workload를 실행한 시간.
- Prep & Reload Config : OtterTune's controller가 다음 configuration을 install하고 DBMS가 다음 observation period를 준비하는데 걸린 시간.
- Workload Mapping : OtterTune's dynamic mapping하는데 걸린 시간(process #3-1).
- Config Generation : target DBMS에 대하여 GP모델을 이용하여 다음 configuration을 찾는데 걸린 시간(process #3-2).

# Results of execution time

## ❑ workload execution : MySQL & Postgre 5분, Vector 5초

- MySQL과 Postgres는 observation period가 고정되어 있어 observation time이 가장 큰 부분을 차지함.
- Vector는 TPC-H workload를 사용하므로 observation time이 매우 짧음.

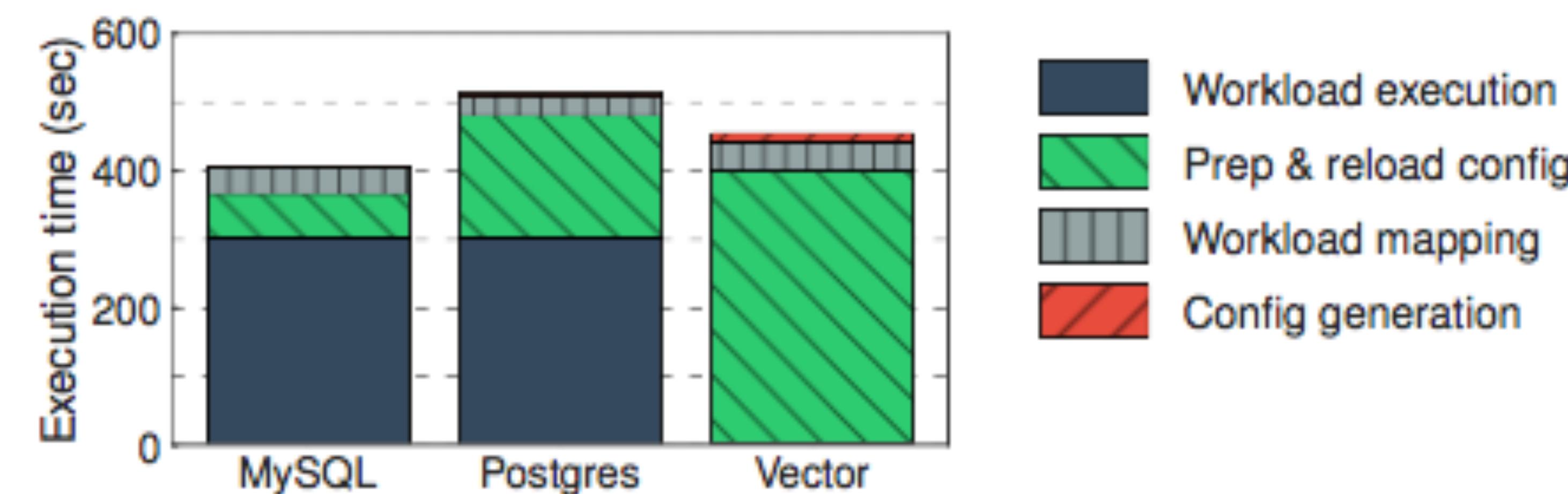
## ❑ Prep & reload config : MySQL 62초, Postgres 3분, Vector 6분 30초

- DBMS restart time으로 측정.

## ❑ workload mapping은 같은 양의 데이터를 사용하므로 소요 시간은 DBMS에 상관 없이 비슷함 (30-40초).

## ❑ configuration recommendation도 같은 양의 데이터를 사용하므로 소요 시간은 DBMS에 상관 없이 비슷함 (5-15초).

처음 두스텝의 시간  
비중이 훨씬 높음



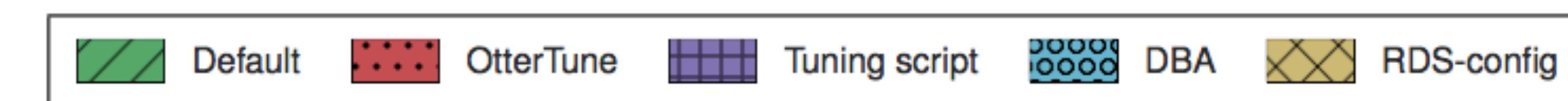
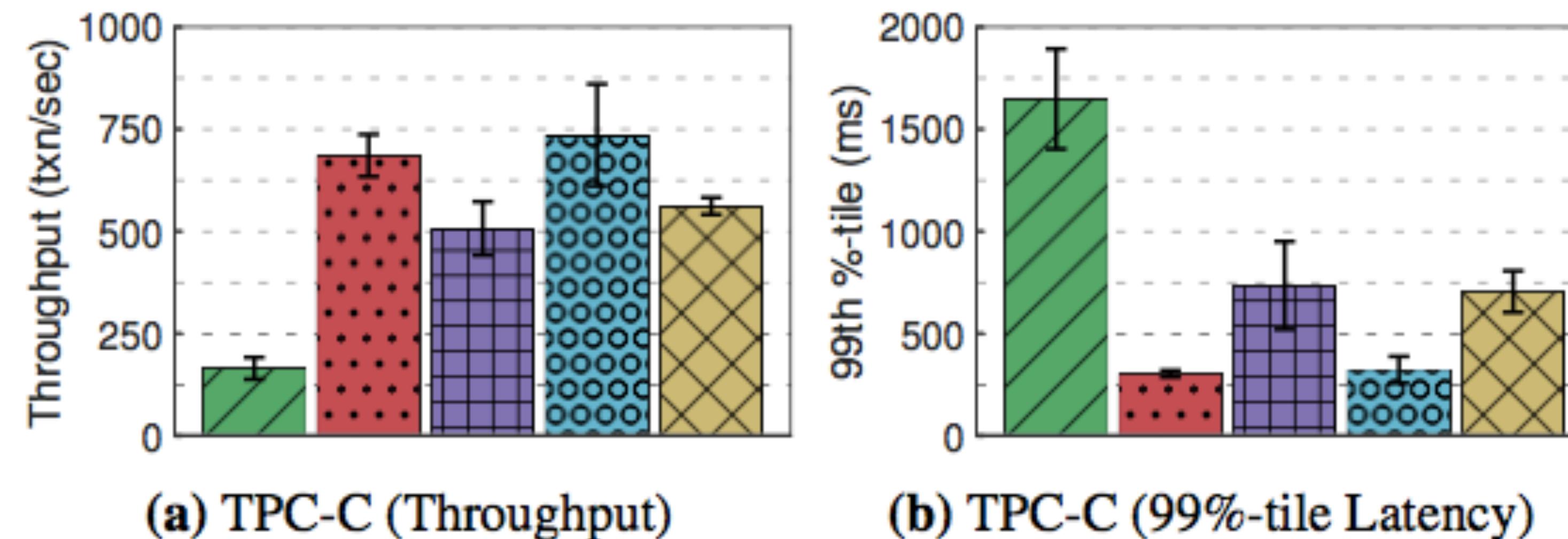
## Comparison Setting

- ❑ MySQL과 Postgres에서 Latency와 Throughput을 비교
- ❑ EC2 환경에서 다양한 튜닝 방법의 성능을 비교
  - OtterTune
  - DBA : DB 전문가
  - RDS-config(DBaaS) : 아마존 클라우드 DB 서비스에서 생성한 Configuration
  - Tuning Script : DBMS 제작사에서 제공하는 튜닝 툴
  - Default Config

# MySQL Result

## □ OtterTune이 DBA와 비슷한 수준의 성능을 보임

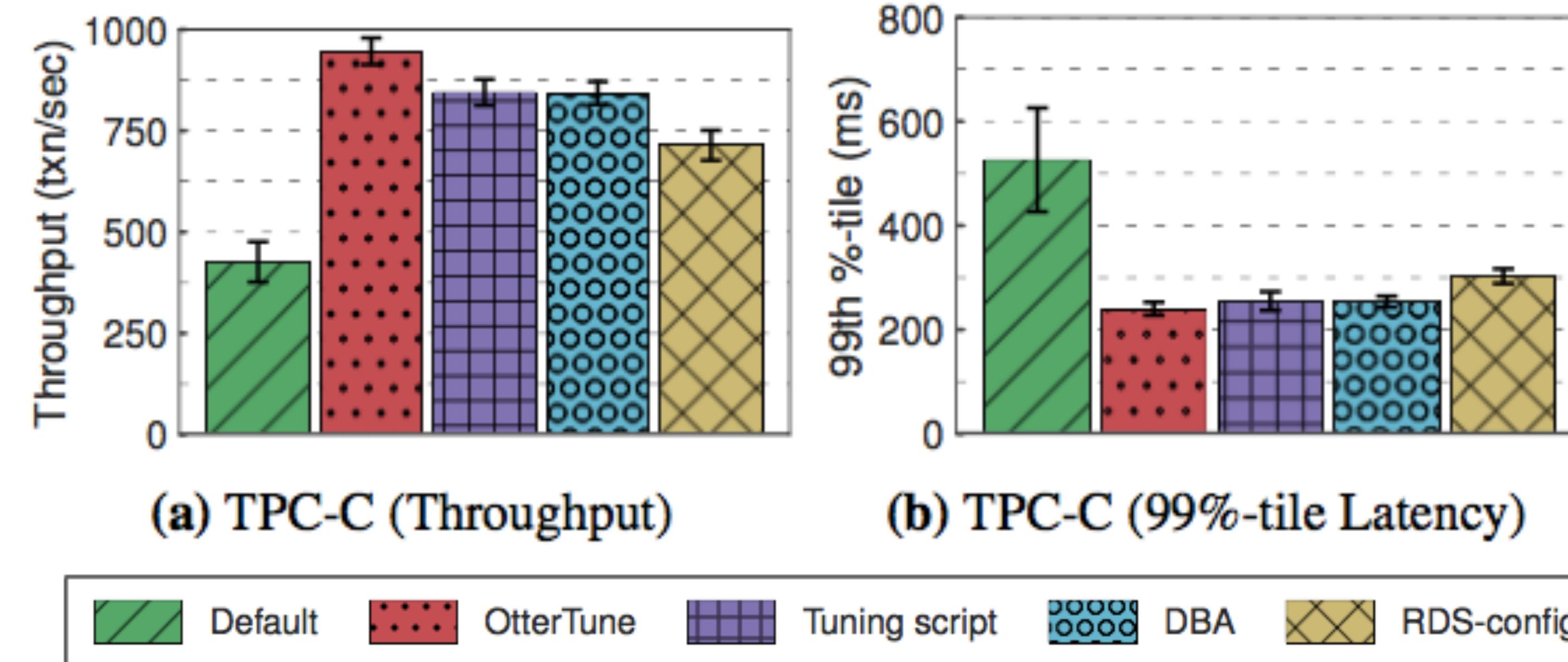
- OtterTune : 60분간 4개의 Knob값 변경
- DBA : 20분간 8개의 Knob 값을 변경함
- RDS-config : 3개의 Knob 값 변경
- Tuning Script(MySQLTuner) : 45분간 5개의 Knob를 변경함, 중요한 노브는 하나만 변경함



# Postgres Result

## □ OtterTune이 DBA보다도 좋은 성능을 보임

- Latency의 경우 OLTP-Bench client와 DBMS간의 통신이 Overhead로 작용하여 성능에 큰 차이 없음
- OtterTune : 60분간 4개의 Knob값 변경
- DBA : 20분간 14개 Knob 수정
- Tuning Script(PGTune) : 30초간 8개 Knob 수정



# Q&A