東南大學
SOUTHEAST UNIVERSITY

Adda: Towards Efficient in-Database Feature Generation via LLM-based Agents

SIGMOD 2025

Kuan Lu, Zhihui Yang, Sai Wu, Ruichen Xia, Dongxiang Zhang, Gang Chen

浙江大學
ZHEJIANG UNIVERSITY

汇报人：宋青阳

2025年9月18日

Contents

Background

趋势：数据库内机器学习 *In-Database Machine Learning*

将机器学习（ML）直接集成到数据库管理系统（DBMS）中

- *Madlib*
- *Vertica-ML*
- *PostgresML*

传统ML工作流：

传统的机器学习流程通常需要将数据从数据库中提取、转换和加载到一个独立的机器学习平台或库中进行处理。存在以下缺陷：

✓ **数据移动的巨大成本：**

数据首先存储在高性能的数据库中。当分析师需要进行ML分析时必须经过以下过程：将海量数据从数据库中提取出来，传输到另一个独立的计算环境

✓ **数据管理与安全的巨大挑战：**

数据必须离开数据库，失去安全性；管理权限模糊，庞大数据难以管理

数据库内机器学习Pipe Line

```
CREATE MODEL housing_price_predictor
FROM sales_data
TARGET price
WITH ALGORITHM = linear_regression;
```

Contents

- Background
- Challenges
- Design
- Evaluation
- Thinking

Challenges

现有数据库内机器学习的问题（如 *PostgresML*）

Analyst: Build an ML model to predict an individual's survival probability in a pandemic.

Write SQLs for ML analysis task

① SQLs for data preprocessing
② SQLs for feature generation and selection
③ SQLs for model training and prediction

SQL Statements

DBMSs

Query engine　Raw Data

Example Dataset

Sex	Age	SibSP	ParCh	Disease	Bmi	Survived
M	55	0	0	No disease	26.97	0
F	57	3	1	Cancer	28.73	0
M	12	5	2	Obesity	25.34	0
F	29	4	NULL	No disease	28.58	1

Challenges

① *Limited support for AutoFE*
② *Some analytical requirements can not be accurately expressed in SQL.*

① 对自动特征工程（AutoFE）支持有限

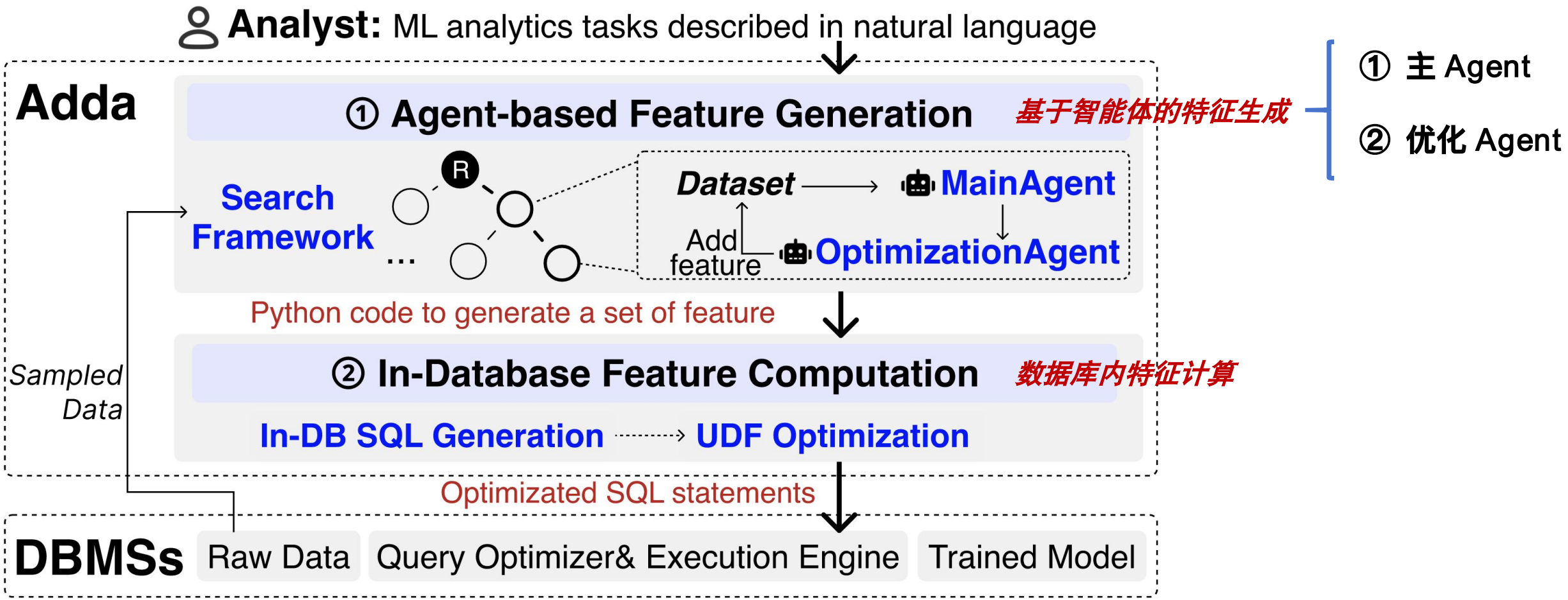
现有框架主要关注的是“模型构建”而非“特征提取”。由于缺乏自动化工具，整个特征工程过程变成了一个高度依赖人工的、繁琐的循环。

② SQL 语言的表达能力有限

自动特征工程本身就是一个复杂的、迭代的过程。它包含迭代式的树搜索、特征计算和验证，需要循环、分支、状态管理等控制结构。SQL 是一种声明式语言，天生就不适合描述这种复杂的算法流程。

→ **Adda**

Design—Overview



Contents

- Background
- Challenges
- **Design**
  - Agent-based Feature Generation
  - In-Database Feature Computation
- Evaluation
- Thinking

Design—Agent-based Feature Generation

自动化特征工程（AutoFE）数学定义

Example：构建一个ML模型来预测个体在流行病中的生存概率

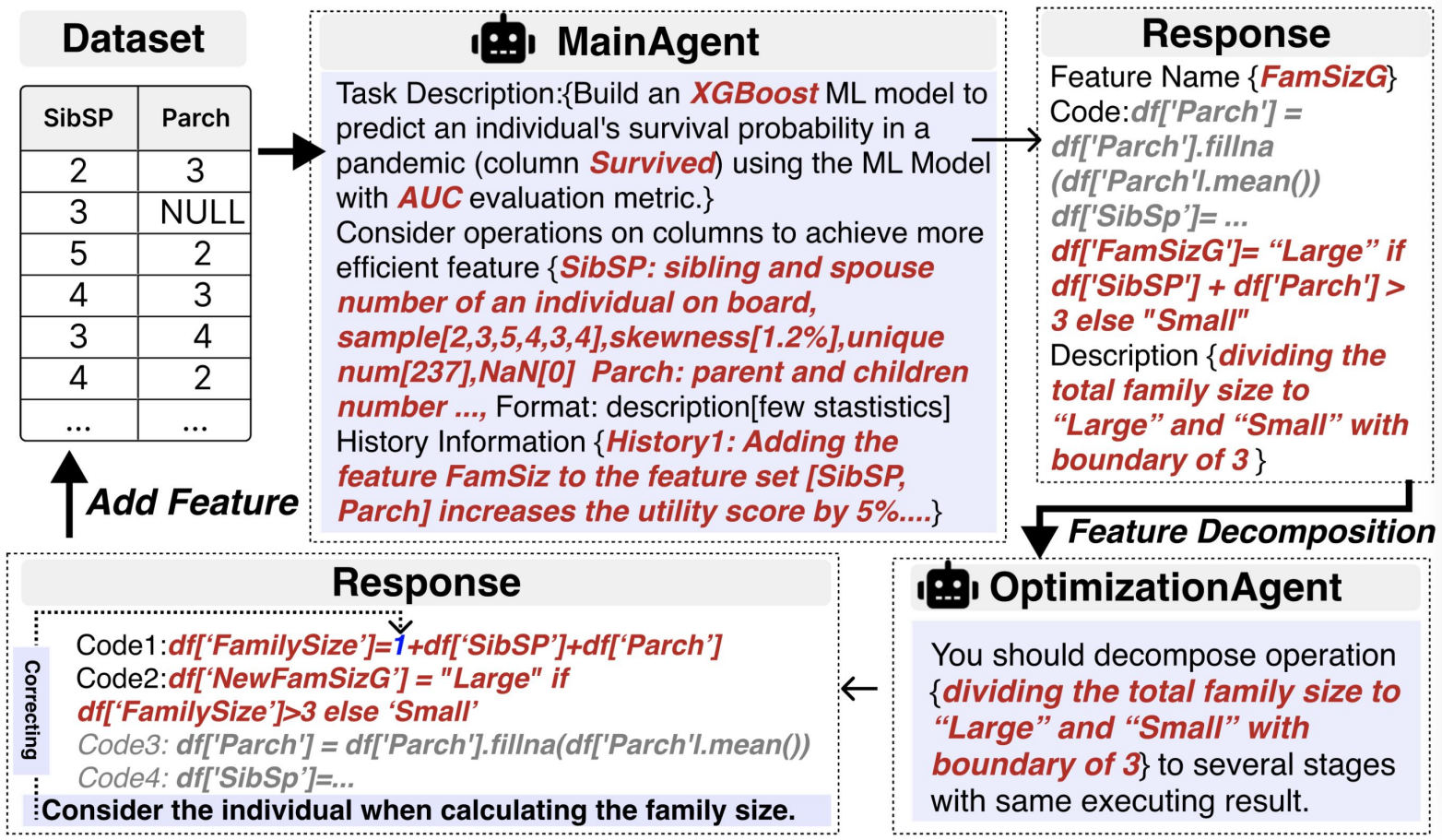
$$\arg\max_{\hat{F}\subseteq\mathbb{F}} \mathcal{L}(L(\hat{F}, Y))$$

目的

让模型 L 在使用这个最优特征集 $\widehat{F}$ 时，表现得最好

- F: 数据集中已有的列（特征），比如“年龄”、“性别”等。
- Y: 我们想要预测的目标列，比如“是否幸存”。
- L: 我们选择使用的机器学习算法，比如 XGBoost。
- $\widehat{F}$: 这是我们要寻找的东西：一个经过优化的、新的特征集合（包含原始特征和由 Adda 创造出的新特征）
- $\mathcal{L}$：代表评估该模型好坏的指标（AUC or F1 score）

Design—Agent-based Feature Generation

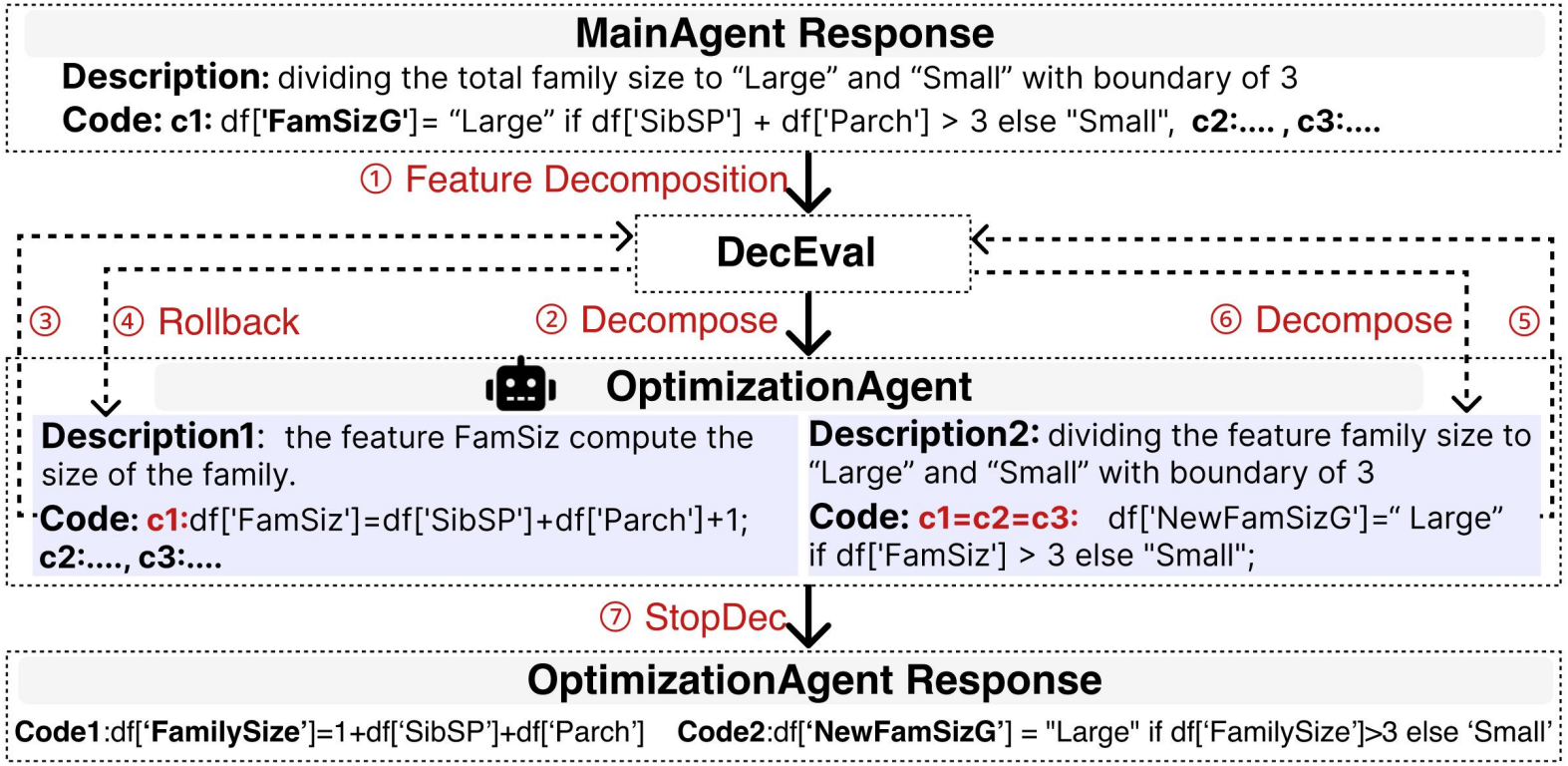


MainAgent

作用：分析 input，最终会输出一个新的〝特征包〞

- **Step1: 构建上下文 *Prompt***
  - ① **静态信息**：由agent自动分析，让LLM对任务有细致的了解
  - ② **动态信息** (via RAG)：查询优先队列Q，其中存储了历史生成的特征，目的是找出最有潜力的下一个探索目标 (U值=AUC+访问次数)
- **Step2: 生成 *Response*（高层概念+Code）**
  - ① **生成高层信息**：将Step1中的Prompt传入LLM，输出了结构化JSON的〝高层概念〞
  - ② **生成可执行代码**：将第一步的原始提示和第二步生成的高层概念（特征名、描述、方法）再次整合成新Prompt，传给LLM生成可执行的Python代码
- **Step3: 本地验证**
  抽样检查代码是否无报错、新特征的值是否合法
- **Step4: *Output* 特征包**
  {Feature Name、Code、Description}输出传给优化Agent

Design—Agent-based Feature Generation



Optimization Agent

作用：优化 *MainAgent* 提出的特征及代码，使之能高质量、可靠且逻辑清晰地被实现

➢ **Step1: 初始评估:判断是否要分解(DecEval)**

对MainAgent传来的特征包进行评估，指标如下：

① 判断是否可靠、自洽

② 判断代码复杂度

若有一条违背，则特征需要分解

➢ **Step2: 分解特征(Feature Decompostion)**

① 告诉LLM，如：将这个任务分解为多个逻辑步骤。

② 优化智能体会调用主智能体，让它为每一个更简单的子描述（d_{j^1} ，d_{j^2} ）分别生成代码（c_{j^1} ，c_{j^2} ）和特征（f_{j^1} ，f_{j^2} ）。

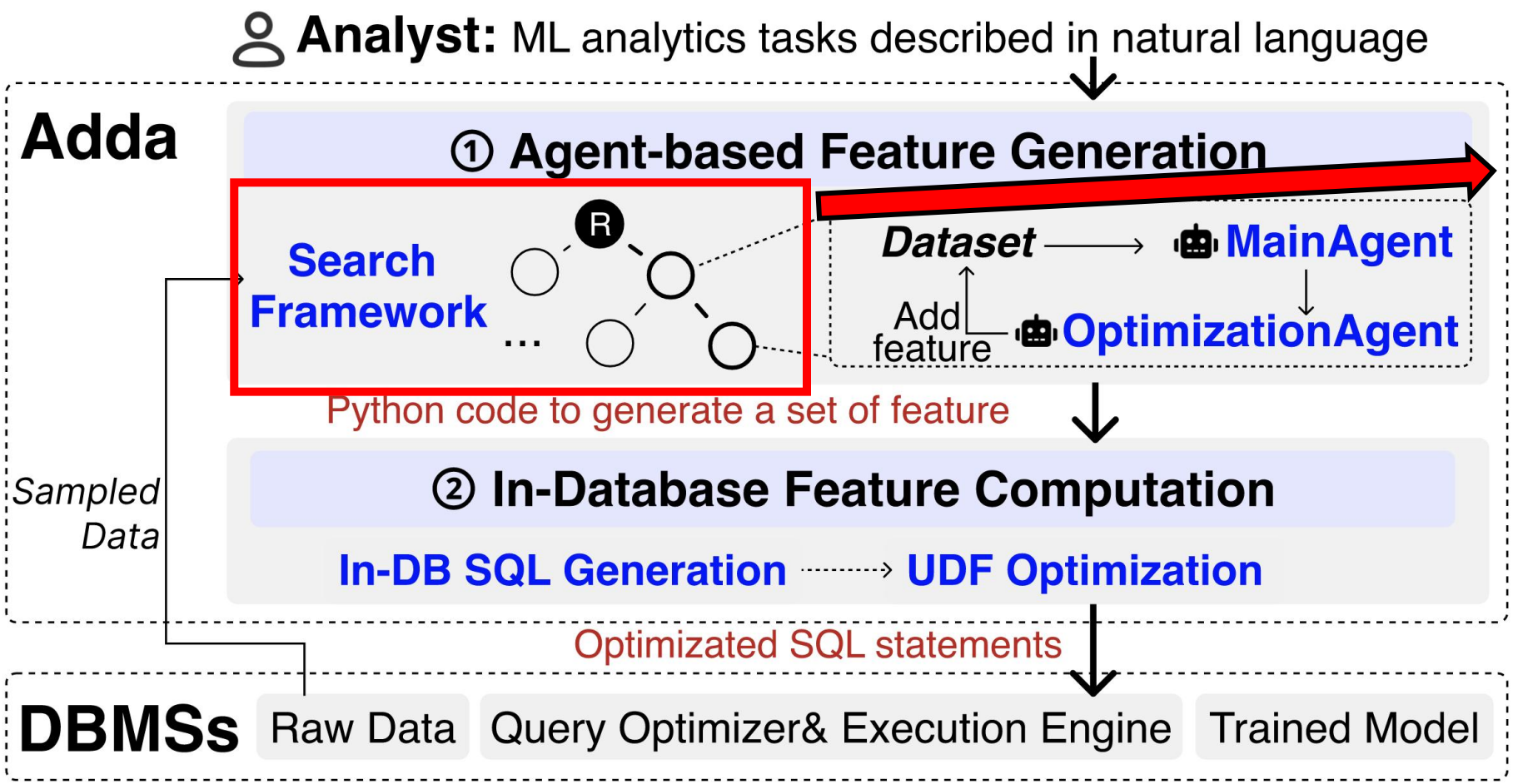
➢ **Step3: 递归检查与回滚(Rollback)**

优化智能体递归每一个新生成的子特征重复执行以上步骤，直到所有特征都足够简单；若过度分解则回滚

➢ **Step4: 分解*验证(StopDec)***

保证分解前后得到的列结果一致

Design—Overview



Search Framework

作用：系统化地记录和管理整个特征探索的过程，并为下一步的探索提供决策依据。

最终目标：通过规定的有限次迭代，构建一个特征树，最终用指标（如AUC）找出最优特征集

迭代实现方式：以综合效用值U为标准的有限队列Q

流程：

① 初始化队列

② 主循环 - 选择与扩展特征
这个循环会重复 s 次
1. 查看队列：系统会审视优先队列 Q 中所有的候选节点。选出最优的当前节点（综合效用值U）
2. 以此节点为基础，通过双Agent生成新特征节点，放回队列Q。

③ 决策出最优特征集——直接使用性能指标 L（例如 AUC 分数）

Contents

- Background
- Challenges
- **Design**
  - Agent-based Feature Generation
  - **In-Database Feature Computation**
- Evaluation
- Thinking

Design—In-Database Feature Computation

Table 2. Operator types in Adda

Operator Type	Operator Description	Example Python Code[1]	SQL Statement
Unary	Conduct arithmetic computation on a feature	df['f_4'] = (df['f_1']/2).astype(int)	SQL: CREATE TABLE T1 AS SELECT (CAST (f_1/2 AS INT)) AS f_4, * FROM SRC;
Numerize	Map a non-numeric feature into an integer feature	df['f_4'] = LabelEncoder.fit_transform(df['f_1'])	SQL: CREATE TABLE T1 AS SELECT (CASE WHEN f_1='op1' THEN 0, CASE WHEN f_1='op2' THEN 1···) AS f_4, * FROM SRC;
Drop	Drop some feature from feature set	df = df.drop(['f_1', 'f_2'])	SQL: CREATE TABLE T1 AS SELECT f_3 FROM SRC;
Multi-element	Generate a new feature by arithmetic computation on several input features	df['f_4'] = df['f_1'] + df['f_2'] / df['f_3']	SQL: CREATE TABLE T1 AS SELECT f_1 + f_2/f_3, * FROM SRC;
Normalize	Scale a feature using MinMaxScaler, StandardScaler or RobustScaler	df['f_4'] = StandardScaler.fit_transform(df['f_1']))	SQL: CREATE TABLE T1 AS SELECT (f_1 - AVG(f_1)) OVER ())/STDDEV(f_1) OVER() AS f_4, * FROM SRC;
FillNaN	fill the NaN v a feature with mean, mode, or a constant		
Discretize	Convert a fea with continuc values into di labeled group		
UDF	Encapsulate a Python code as a UDF		

[1] We present some Pytho

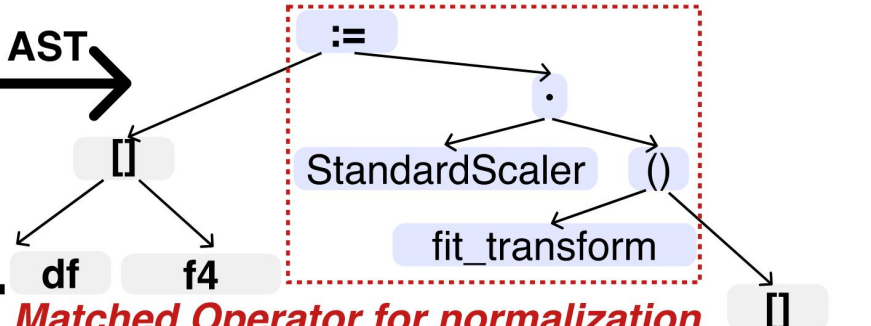
目的：***Python-to-SQL***

- **Step1:** 解析为 Python 抽象语法树（方便机器识别）
- **Step2:** 匹配操作符 & 生成 SQL

Python Code:

df['f4']= StandardScaler.fit_transform(df['f1'])

① Python AST



②

SQL Statement:

CREATE TABLE TB1 AS SELECT (f1 - AVG(f1)) OVER ())/ STDDEV(f1) OVER() AS f4， * FROM SRC

Matched Operator for normalization

CREATE TABLE TB1 AS SELECT (· - AVG(·)) OVER ())/STDDEV(·) OVER() AS · , * FROM SRC

Design—In-Database Feature Computation

UDF Optimization

WHY ?

UDF虽然可以解决复杂特征工程的表示，但是它是完全黑盒的。

因此数据库的调度系统无法知道哪些是有效调包，只能将所有数据全从外存读取。

“打包”和“解包”的过程开销极大

策略一：合并 UDFs (Merge UDFs) - 减少调用次数

核心思想：合并两个连续的复杂特征

ep：**原始步骤**：

```
-- 步骤1: 创建特征 X
CREATE TABLE T1 AS SELECT my_udf_1(a, b) AS X, a, b, c FROM SRC;
-- 步骤2: 基于 X 创建特征 Y
CREATE TABLE T2 AS SELECT my_udf_2(X, c) AS Y, X, c FROM T1;
```

优化策略：合并my_udf_1 和 my_udf_2为new_udf

```
CREATE TABLE T2 AS SELECT my_merged_udf(a, b, c) AS (X, Y), a, b, c FROM SRC;
```

策略二：缩减 UDFs (Shrink UDFs) - 减少打包内容

核心思想：Adda 在创建 UDF 时，做了一个补充信息，记录下了这个 UDF 到底需要哪些输入列。再用成本模型 (Cost Model) 来估算新旧计划哪个更快

ep：SELECT my_udf(a, b, c, d, e, ...) AS Y FROM table;

（其中table中有100项，实际UDF只需要用a、b两列）

现在，Adda 有了两个执行计划可以选择：

计划 A：保持原样，把所有 102 列都传给 UDF。

计划 B:

① 第一步 (UDF): 只把需要的 a 和 b 两列传给 UDF 进行计算，得到结果 Y。

② 第二步 (JOIN): 将第一步的结果 Y，与那些不需要参与计算的 100 列 (c, d, e, ...) 拼接 (JOIN) 回来，形成最终完整的行。

最终使用一个成本模型 (Cost Model) 来估算这两种计划哪个更快

Contents

- Background
- Challenges
- Design
- Evaluation
- Thinking

Evaluation

A. Experimental Setup

➢ *Datasets*

收集了14个公共数据集，包括来自UCI仓库、OpenML和Kaggle的8个分类数据集和6个回归数据集。从每个数据集中随机抽取1000行作为样本数据集

➢ *Tasks and Metrics*

任务： 采用了来自scikit-learn的五个回归和分类模型，包括决策树、随机森林、极端树、LightGBM 和 XGBoost；

指标： AUC 作为**分类**任务的指标，1 – rae（相对绝对误差）作为**回归**任务的指标；端到端延迟、特征工程时间及模型预测时间作为**效率**指标

➢ *Adda 配置*

在 PostgreSQL 上实现；调用 GPT-4o 的 API

Adda 默认参数：样本数据大小为2000；

step（探索深度） = 10，t = 6（每次探索分支数）

➢ *Baselines*

SOTA AutoFE： *AutoFeat*、*CAAFE*、*SmartFeat ...*

SOTA In-Database ML： *Madlib*、*PostgresML*

➢ *Environment*

X86 CPU (Xeon(R) Silver 4114, 20核@ 2.20GHz)、78GB RAM、一个最大传输速率为 12Gb/s 的 2TB 磁盘以及一个带 24GB VRAM 的 Nvidia Quadro RTX 6000 GPU 的服务器

Evaluation

B. Optimization Performance

- **模型性能提升效果**

  对比 AutoFE 方法和数据库内 ML 方法，Adda 生成的特征能够让机器学习模型的预测准确率（AUC）达到最高。

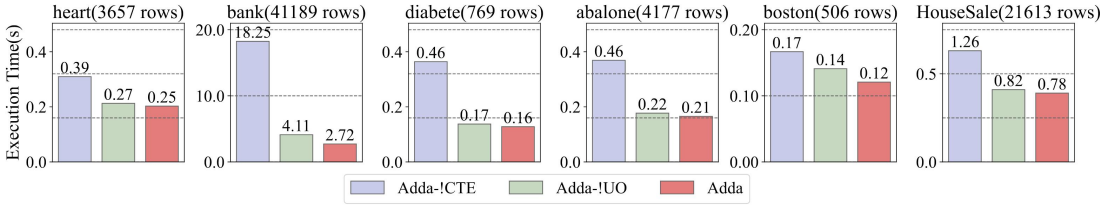
- **运行效率**

  端到端延迟：在小数据集上与其他的基于LLM的方法速度相近；对于近百万行的大型数据集，Adda只需其他系统1/10的时间（得益于抽样验证）

  模型训练时间：Adda的数据库原生 SQL 和优化的 UDF使得模型训练时间显著减少

- **消融实验**

  对RAG、特征分解、UDF优化进行消融实验，说明Adda不是黑盒的，是多部件协同工作



- **真实多客户端服务场景表现**

  与在 Python 中提供模型服务相比，Adda 这种数据库内方法初始吞吐量高出4-6倍，延迟也低得多。随着并发量增加，Adda 的纯SQL架构表现更稳定。此外，Adda 的内存占用远低于 Python，因为它们能复用数据库的内存管理机制。

Contents

- Background
- Challenges
- Design
- Evaluation
- Thinking

Thinking

这篇 paper 的 idea 引起的思考?

- 本文提出了一种“LLM-Agent驱动”的数据库内自动特征工程框架 Adda。该框架通过一个双 Agent 协作系统（主智能体负责特征生成，优化智能体负责特征优化），将自然语言任务，自动转化为高质量的特征及代码。
- 另外，本文提出了“Python - to - SQL”的流程。先将 Python 转化为抽象语法树 AST，然后通过模式匹配和UDF（用户定义函数）优化（合并与缩减），将这些逻辑高效地翻译成数据库可以高效执行的 SQL 语句。

这篇 paper 的思想能否泛化?

- 面对一个复杂的、巨大的、难以暴力搜索的资源管理问题（如异构计算资源管理），都可以借助 Main Agent + LLM 和 Optimization Agent + Cost Model 这样的框架，通过不断迭代和反馈，最终找到一个最优的部署策略。

東南大學
SOUTHEAST UNIVERSITY

Q & A

汇报人：宋青阳

2025年9月18日