缓存知识，而不只是提示：动态 Datalog 作为大语言模型智能体的行动语言

Bruno Rucy Carneiro Alves de Lima
rucy@lucidarium.systems
Lucidarium Systems 爱沙尼亚塔尔图 塔尔图大学计算机科学研究所 爱沙尼亚塔尔图

Victor Henrique Cabral Pinheiro
victor.pinheiro@ut.ee 塔尔图大学计算机科学研究所 爱沙尼亚塔尔图

摘要

一个回答有关知识图谱的问题的大语言模型智能体会针对每个问题编写一个程序，而其工具会从头开始对每个程序进行求值。我们为该智能体提供一个 *Datalog 交互式环境*：它运行在动态 Datalog 引擎之上，在其中，规则可以像事实一样插入和撤回，同时派生事实保持最新。提示缓存会复用模型对会话共享内容的处理，而 Datalog 交互式环境则会复用对这些内容的求值。在三个知识图谱上的问题链中，使用两种模型时，智能体会复用一个交互式环境保留的关系，该关系在 180 个案例中的 179 个中得以保留，并且生成的输出词元更少：SQL 和 Python 智能体多生成 23% 至 82% 的输出词元。使用 SQL 表或 Python 命名空间时，Claude Haiku 5.5 几乎不会为后续问题存储任何内容；在包含 1,300 万个事实的问题链中，连续七个问题需要其耗时中位数分别为 48 秒和 56 秒，而使用交互式环境时仅需 31 秒。Claude Opus 5.5 通过在 SQL 中存储表来达到与交互式环境相当的效果；当数据发生变化时，它会存储视图，而每次读取时都会重新对视图进行求值。交互式环境保留的关系读取方式类似于表，同时又能像视图一样保持最新。相较于 SQL 和 Python，Datalog 是一种小众语言；启用思考功能后，模型使用它回答问题的准确度同样高。

CCS 概念

• **计算方法 → 知识表示与推理**；自然语言处理；• **信息系统 →** 查询语言；演绎数据库。

关键词

大语言模型智能体、Datalog、知识图谱、动态 Datalog、增量计算、工具调用、提示缓存

1 引言

大语言模型智能体越来越多地通过编写代码来执行操作。在 CodeAct 中，[20], 智能体生成 Python 代码，由读取-求值-打印循环（REPL）执行；REPL 会在多轮交互中保留变量和定义，因此后续代码可以基于先前的结果运行。递归语言模型为模型提供了这样一个 REPL，其中以变量形式保留长上下文，并将对子模型的调用作为函数提供[23], ；而程序化工具调用则将允许执行这些操作的工具公开为函数，由生成的代码调用[2]。

在一个会话中，这类智能体的提示会不断增长。系统提示保持不变，而每一轮都会附加智能体的代码及其生成的结果。提示缓存正是利用了这一点。模型对提示的处理过程及其注意力状态可以被存储，并由包含相同声明文本片段的后续请求复用 [9]。商业 API 会存储对提示前缀的处理结果；后续请求复用这些结果时，收费至多为输入价格的十分之一 [3]。

对智能体代码的求值并没有这种复用。设想一个智能体通过编写查询来回答有关知识图谱的问题。许多问题需要递归（“哪些组织直接或间接隶属于这所大学？”），或需要依据本体进行推理（“哪些雇员在这个部门工作？”，而数据中只记录了正教授和讲师）。在一次会话中，智能体会针对每个问题编写一个程序，并在结果看起来不对时修订程序；查询引擎则会在整个图上从头求值每个版本，不论程序有多小，也不论它与之前的程序有多少共用之处。物化视图会存储程序的结果，并在数据发生变化时更新这些结果，从而消除本体所蕴含内容的这项成本 [11] [15]。问题所带来的规则及其每次修订，都会由智能体使用的引擎从头求值。智能体可以将派生关系存储起来，以供后续问题使用，存成表或变量都可以；但副本需要由智能体自行管理，而且会随着数据变化而过时。

Datalog 能简洁地表达这类问题 [1]。Datalog 程序由一组规则构成，每条规则都从其他事实推导出事实。递归是一种普通规则，本体是一组只需编写一次的规则，而一个问题则会添加几条规则和一个查询。在一个会话中，智能体的 Datalog 程序会像其提示一样不断扩展。

动态 Datalog 引擎可以在运行时插入或撤回规则及事实，并在每次更新后报告程序推导出的事实发生了哪些变化 [6]。提示缓存会复用会话共同使用的提示处理过程，而动态 Datalog 引擎会复用会话共同使用的程序求值过程。我们将二者结合起来。Datalog 交互式环境是一个会话，其中保存着知识图谱、其规则以及智能体编写的每条规则，并将智能体的每一步都作为一次更新来应用：可以是新问题的规则，也可以是对这些规则的修订。图谱本体所蕴含的事实会在每个会话中计算一次；每一步都会作为对先前推导出的事实的更新进行求值；智能体定义的关系会作为具名规则保留下来，用于回答之后关于任何实体的问题，并在数据变化时保持最新，而无需智能体管理任何状态。

据我们所知，先前没有研究将动态 Datalog 引擎作为工具交给大语言模型代理，也没有研究将智能体程序的求值与提示缓存联系起来。我们的评估得出五项发现。**智能体会利用会话中保留的内容**。在两个模型中，

在适用的 180 个案例中，有 179 个案例复用了 REPL 保留的关系；在数据不断变化的 66 个案例中，有 64 个案例复用了该关系，而且它们生成的输出词元更少：SQL 和 Python 代理生成的输出词元多出 23% 至 82%。**小模型无法从 SQL 或 Python 中获得这种复用**。使用持久化 SQL 数据库或 Python 命名空间时，它在 288 个案例中仅有 2 个案例存储了关系；在包含 1300 万条事实的图上，七个问题组成的链耗时中位数为 48 秒和 56 秒，而使用 REPL 时为 31 秒。**大模型会在 SQL 中选择使用表还是视图**。数据固定时，它会存储表，此时其问题链耗时与使用 REPL 时相当。数据变化时，它会存储视图，而视图每次读取时都会重新求值。REPL 保留的关系，读取起来像表一样，而保持最新的方式则像视图一样。**图越大，节省的时间越多**。在全部三张图上，基于已保留关系提出的问题，图越大，节省的时间越多，最多可从 4.69 秒降至 0.63 秒。数据变更耗时约 1 毫秒；对于包含 1300 万条事实的数据，修订一个保留大型关系的程序中的一条规则，作为更新只需 2 毫秒，而从头计算则需 36 秒。**Datalog 是一种小众语言，而模型使用它时的准确度同样高**。启用思考功能后，模型使用 Datalog 作答的准确度与使用我们拿来比较的语言时相同；一个拥有 17 亿参数的开放模型，仅使用 Datalog 就能回答递归问题。

本文做出了三项贡献。

（1）Datalog 交互式环境是一种用于智能体工具的设计，其中规则和事实属于更新，答案是对派生事实的变更，而派生事实会在会话中的各个问题之间持续存在。（2）针对三个知识图谱中的 23 个问题构建了一项基准测试：通过测量每个问题的参考程序所需的内容，为每个问题标注类别；同时，智能体测试框架使用 Datalog、SQL、SPARQL 和 Python 分别提出每个问题，并且每种语言都同时基于基础事实和全部派生事实进行查询。（3）一项求值测量了智能体是否会复用会话所保留的内容、与智能体使用的 SQL 和 Python 工具相比，这种复用节省了什么、一次修订需要付出多少代价，以及大语言模型编写 Datalog 的准确性。

2 相关工作

大语言模型服务中的复用。服务系统会复用模型在单个提示前缀之外的处理结果。SGLang 的 RadixAttention 会缓存语言模型程序调用所共享的前缀的注意力状态 [24], , 而 GPTCache 等语义缓存则根据查询的嵌入向量存储模型的响应，并将已存储的响应返回给相似查询 [8]。Datalog 交互式环境会复用智能体程序的求值结果，而保留的规则能够回答之后关于任意实体的问题。

保留其计算结果的智能体。 Voyager 存储着一个不断扩大的可执行代码技能库，后续任务可以从中检索 [19], , 而智能体工作流记忆则从过去的任务中诱导出可复用的工作流 [21]。Datalog 交互式环境将智能体的关系保留为规则，引擎会随着数据变化持续更新这些规则所派生的事实。

编写逻辑程序的大语言模型。 Faithful Chain-of-Thought 将问题转化为 Python、Datalog 或规划语言中的推理链，由确定性求解器执行，从而使该链成为答案的忠实解释 [13], 以及 Logic-LM 将问题转化为逻辑程序和其他求解器输入，并根据求解器的错误消息进行修订 [16]。Datalog 交互式环境会在一次会话中保留求解器的状态，因此，一个问题对应的程序会建立在先前问题的程序之上，而每次修订都会作为更新应用。

增量式动态 Datalog。 DRed 和计数 [11], , 以及递归计数和正向/反向/正向维护 [15], , 会在给定事实被插入和删除时维护程序所派生的事实；当规则被插入和删除时，DRed 也会进行维护 [11], ；而 IncA 则会在被分析程序发生变化时，维护固定程序分析所派生的事实 [18]。DDlog 将固定程序编译为构建于差分数据流之上的增量数据流 [17] [14], , 而 Soufflé 将固定程序编译为并行 C ++ [12]。DBSP 会从查询本身推导出该查询的增量版本，包括递归 [4]。早期工作评估了差分数据流作为 Datalog 后端在频繁更新场景下的表现 [5] , 并将增量计算应用于分布式本体推理，其中本体本身会发生变化 [7]。[6] 的引擎会将规则在运行时发生变化的动态 Datalog 程序，作为高阶 DBSP 程序进行求值，其中规则本身就是输入。

范围。REPL 使用现有动态 Datalog 引擎的扩展实现 [6], , 而我们的贡献在于设计了一种将动态 Datalog 暴露给智能体的方式，并对其进行测量。我们将 REPL 与智能体所使用的 SQL 和 Python 工具进行比较：按照智能体使用这些工具的方式，针对问题所需的递归关系和派生关系进行比较，且不使用其他推理器。

3 背景

3.1 Datalog 示例

Datalog 程序由事实和规则组成。事实描述常量之间的关系，例如 parentof(ann, bob)。规则说明某个事实由其他事实推导而来。以下两条规则

```
ancestorof(X, Y) :- parentof(X, Y).
```

ancestorof(X, Z) :- parentof(X, Y), ancestorof(Y, Z)。可读作：Y 的每位父母都是 Y 的祖先，而 Z 的祖先的每位父母都是 Z 的祖先。以大写字母开头的名称是变量。第二条规则在 :- 两侧都提到了 ancestorof，因此它是递归的。由事实给出的谓词（如 parentof）是基谓词，而由规则定义的谓词（如 ancestorof）是派生谓词。引擎通过将规则应用于事实来执行程序，直到不再产生新事实。以这种方式获得的、除给定事实之外的事实，是程序的派生事实；所有事实合在一起则构成程序的闭包。问题是对基谓词和派生谓词的查询，也可能带有其自身的规则。“谁是 cid 的祖先？”这一问题对应的查询是 ?- ancestorof(X, cid)。而“哪些女性是 cid 的祖先？”则变为

```
answer(X) :- ancestorof(X, cid), female(X). ?-
answer(X).
```

我们的求值所用的生成的家族图，从这两个问题开始提出一系列问题。

带有本体的知识图谱形式相同。在 LUBM 大学基准测试中，[10], 数据记录的事实如下

```
问题 1（共 5 个）： person115 的祖先是谁？ ancestorof(X, Y) :-
parentof(X, Y). ancestorof(X, Y) :- parentof(X, Z),
ancestorof(Z, Y). answer(X) :- ancestorof(X, ".../person115"). ?-
answer(X). 34 行……变更（按谓词列出的新增和删除事实）： -
ancestorof/2： +2824 -0 …… - answer/1： +34 -0 ……库：空。问
题 2（共 5 个）： 哪些女性是 person115 的祖先？ 库：
ancestorof/2（2 条规则）。answer(X) :- female(X),
ancestorof(X, ".../person115"). ?- answer(X). 17 行……变更（按
谓词列出的新增和删除事实）： - answer/1： +17 -0 ……库：
ancestorof/2（2 条规则）。
```

```
问题 5 个中的第 2 个，SQL 会话： WITH
RECURSIVE anc(a, b) AS ( SELECT subject, object
FROM parentof UNION SELECT p.subject, anc.b
FROM parentof p JOIN anc ON p.object = anc.a )
SELECT DISTINCT anc.a AS entity FROM anc J
OIN female f ON f.entity = anc.a WHERE anc.b =
'.../person115'
```

列表 1：左侧：会话中的两个问题，该会话的 REPL 保留了智能体的谓词，以及智能体的程序和工具的回复。右侧：SQL 智能体针对第二个问题的程序，其中再次陈述了递归。答案行、示例事实和标识符前缀被省略。

例如哪些人员在哪些部门工作，而基准测试的 Univ-Bench 本体规定，例如，每位正教授都是教授，并且雇员是为某个组织工作的人员。每条此类陈述都会成为一条规则，

```
教授(X) :- 全职教授(X). 雇员(X) :- 人员(X), 任职
于(X, Y), 组织(Y)。
```

而我们对 LUBM 本体的翻译包含 99 条规则，它们会推导出数据中从未明确陈述的事实。

3.2 动态 Datalog

在插入和删除基础事实时，保持派生事实与时俱进的做法称为增量视图维护 [11] [15], ，而 DDlog 和 Soufflé 等编译型引擎会在编译时固定程序 [17] [12]。动态 Datalog 引擎以与处理事实相同的方式，将规则视为输入。更新会插入或撤回规则和事实；每次更新后，引擎都会报告派生事实的变化：现在能够推出的事实，以及不再能够推出的事实。引擎根据更新之间保留的状态计算这些变化，而不是从头重新评估程序 [6]。我们通过其 Python 绑定 PyDYRE 使用 materialized-view。它是用 Rust 编写的动态 Datalog 引擎 [6] 的扩展实现，版本对应于 github.com/brurucy/materialized-view 上的提交 b34efb0。它会评估该项目中的高阶 DBSP 程序，我们将其称为引擎。

4 面向智能体的 Datalog 交互式环境

4.1 会话、更新与答案

会话保存知识图谱的基础事实、本体规则以及智能体编写的规则，而动态 Datalog 引擎会持续更新它们所产生的派生事实。智能体使用一个工具，该工具接收一段 Datalog 源代码，并返回对其查询的答案，以及这段代码所产生变更的摘要。

一次工具调用，逐步完成。 REPL 分四步处理一次调用。

(1) 它将片段解析为规则、事实和一个查询，并根据图的模式、会话保留的谓词以及引擎的限制进行检查：原子最多包含三个项，规则最多包含五个变量。如果片段未通过检查，就会被拒绝并显示一条消息，会话则保持原样。(2) 它与为当前问题记录的程序求集合差，并将差集作为一次更新应用到引擎中；这会撤回片段中删去的规则和事实，并插入片段中新增的规则和事实。(3) 它从派生事实中读取查询的答案。(4) 它返回答案行和更新摘要：对于事实发生变化的每个谓词，列出新增和移除的事实数量，并最多提供三个示例；最多列出十个谓词，变化最大的排在最前。

列表 1 展示了一个智能体的两次调用，分别对应上一节提出的两个问题，问题均针对生成的家族图中的 person115。

Python REPL 的状态是按特定顺序执行语句后形成的命名空间，而 Datalog REPL 的状态则是当前规则和事实所产生的派生事实集合。修订就是差异。如果智能体一开始在答案规则中漏掉 female(X)，并在看到结果后再添加，那么第二步会撤回旧的答案规则并插入新规则；两个版本共有的 ancestorof 事实则不会重新推导。因此，智能体像使用 SQL 或 SPARQL 那样编写完整程序，而引擎只处理程序之间的差异。规则以句点结尾，撤回规则即可将其准确撤销，因此 REPL 也可以在模型生成每条规则后立即应用它。

问题的结束。问题以三种模式之一共享同一个会话，并且在所有模式下，知识图谱及其规则都只加载一次。在隔离模式下，问题结束时，REPL 会撤销该问题的规则，因此每个问题只能看到知识图谱和自身的规则。在库模式下，智能体会在程序中使用 `.share`指令标记后续问题可以使用的谓词。问题结束时，REPL 会保留以下规则：

表 1：知识图谱上的智能体会话期间发生的变化，以及哪类系统将每种变化作为更新进行处理。“否”表示系统从头重新进行评估。

	图事实	问题规则	修订	为下一个问题保留
查询引擎，每次从头开始查询	no	no	no	无
物化视图，增量维护 [15] [17]	yes	no	no	本体所蕴含的内容
Python REPL（CodeAct、递归语言模型）	no	no	no	代码存储的值
Datalog 交互式环境	**yes**	**yes**	**yes**	**所有派生事实**

表 2：知识图 谱及其问题，按每个问题自身的程序向图的谓词中添加的内容分类 s.

图	基础事实	带规则的事实	规则	问题	查找	Join	递归
LUBM-3	336,130	462,719	99	12	3	8	1
Follows	921	21,458	3	4	1	3	0
Family	453	8,567	9	7	2	3	2

共享谓词的规则，以及它们所读取的谓词的规则；并撤销该问题的其他规则。在全部保留模式下，REPL 会保留除 answer 及读取 answer 的谓词之外所有谓词的规则，无需进行标记。列表 1 是一个全部保留会话：第一个问题将两条 ancestorof 规则留在会话中，第二个问题的程序则是一条读取这些规则的规则。每个结果和每个新问题都会列出保留的谓词。REPL 会统计每条规则的持有者，因此，某个问题再次声明的规则只会插入一次，并且只有在最后一个持有者将其释放时，才会从引擎中移除。

4.2 如何对更新进行求值

引擎有两个输入：事实和规则。每个输入都是一组变更，其中插入的权重为 $+1$，撤回的权重为 -1 [6]。规则以数据的形式输入。规则体中的每个原子对应一条记录，记录规则的部分匹配（即截至目前已匹配变量的绑定）接下来要与哪个原子匹配，以及届时该原子的哪些列已由常量或先前的原子绑定。还有一条记录用于记下规则体匹配后要推导出的头部。求值是一个由四次连接组成的循环。每个事实都按其谓词下规则体原子中已绑定的列建立索引；一条部分匹配与其规则中的下一个原子配对；与其在已绑定列上相符的索引事实会扩展该部分匹配；通过最后一个规则体原子的匹配则会将其头部作为新事实输出。这个过程持续进行，直到不再产生新事实。每次连接都是一个具有增量语义的 DBSP 运算符，它根据输入的变化及其保留的状态计算输出的变化。因此，更新以变更的形式进入循环，引擎则报告派生事实的变化，REPL 的摘要会按谓词统计这些变化。在列表 1 中，第二个问题的规则以一次变更的形式输入，最终产生 17 条新的答案事实；摘要则显示 ancestorof 没有变化。

REPL 使用的方言是正 Datalog，也就是 [6] 所评估的片段。

4.3 会话会改变什么，又会缓存什么

在一个会话中，有四项内容会发生变化：图中的事实、每个新问题的规则、这些规则的修订，以及可供下一个问题使用的工作。表 1 列出了智能体当前使用的各类系统会将其中哪些内容作为更新来处理。

提示缓存与 Datalog 交互式环境在同一会话中结合：描述图的模式和规则的提示会缓存在模型侧，而图的派生事实会缓存在求值侧。提示缓存会复用模型对先前请求中共同前缀的处理，而从第一个发生变化的词元开始，后续内容都需要重新处理。交互式环境会以任意顺序复用会话中已有每条规则的派生事实，而发生变更时，只会更新变更规则所产生的后果。

5 求值

我们提出四个问题。RQ1：智能体是否会复用会话中保留的内容，包括跨问题复用以及数据发生变化时的复用？RQ2：与智能体使用的 SQL 和 Python 工具相比，这种复用能节省什么？RQ3：修订的代价是多少？RQ4：大语言模型编写小众语言 Datalog 时，准确度是否与编写 SQL、SPARQL 和 Python 时一样高?

5.1 知识图谱与问题

我们使用了三个知识图谱（表 2）。LUBM-3 由 LUBM 基准测试中编号 0 至 2 的大学组成，[10] 并采用 Univ- Bench 本体。Follows 是一个生成的网络，由 12 个社群组成，每个社群有 25 个账户；每个社群内部都有环，社群之间则由单向链接相连。Family 是一个生成的家谱，包含六代共 159 人，其图见列表 1。这三个图中的问题需要用到连接、本体推理和递归。

每个问题都有一个以 Datalog 编写的参考程序，而每个问题标签都是对其参考程序的一项测量指标。问题为图的谓词增加的内容包括：当其程序没有自己的规则时，是查找；当其程序有规则但没有递归时，是连接；当其某条规则具有递归性时，则是递归。

构建问题集时，如果移除某个问题自身规则中的任意一个条件，或移除其自身的递归规则后，答案仍然不变，该问题就会被剔除。每个标准答案都会计算两次：一次通过 Datalog 参考程序计算，另一次通过 SPARQL 参考查询在添加了所有派生事实的图上计算，两者必须一致。对于全部 41 个类和属性，LUBM-3 的派生事实均与独立 OWL 2 RL 推理器得出的结果一致；生成图的派生事实也与每个派生谓词的纯 Python 实现结果一致。

5.2 智能体测试框架与条件

八种条件由四种语言与两类数据组合而成。在标记为 M 的条件下，程序在包含所有派生事实的图上运行；Datalog 交互式环境加载图的规则后便是如此，因此智能体只需编写问题新增的内容：Datalog、SQL、SPARQL 1.1 查询及辅助规则，或一个返回答案行的 Python 函数。在标记为 B 的条件下，程序仅在基础事实上运行，智能体还需编写图的推理逻辑。Datalog 在 Datalog 交互式环境中运行，SQL 在 DuckDB 2.0.0 中运行，SPARQL 在 Oxigraph（pyoxigraph 0.5.13）中运行，Python 则在无法访问网络的沙盒中运行。

每种条件都会收到一份系统提示，其中各部分相同且顺序一致；图的规则也都以相同的英文句子表述，因此不会有任何语言以自身语法获得这些规则。在交错进行推理和工具调用的智能体循环中，[22], 模型编写程序，通过该条件的工具最多运行三次，读取每次结果，然后提交最终程序。在 Datalog 条件下，交互式环境会将每次运行作为更新应用于前一次运行的程序，而智能体对此并不知情。评分器会重新运行提交的程序，并将其答案行与标准答案按集合进行比较。我们报告精确匹配率，以及答案行的 F1 分数。模型为 Claude Haiku 5.5，启用高强度自适应思考；每个问题运行三次，每次运行称为一个轮次，测试框架为 inspect-ai 0.3.285。独立的问题还在另外两种设置下运行：同一模型关闭思考功能，以及在本地以单轮方式运行的 LFM3-1.7B（一款拥有 17 亿参数的开放模型）。

5.3 会话

独立样本测量准确率，却没有留下任何内容供后续问题复用。在 Datalog-B 条件下，一个会话会让一个智能体在同一次对话中使用同一个 REPL，依次针对一张图提出多个问题；因此，智能体会编写每个问题所需的全部规则，包括该图的规则。

这些独立的问题除了图中的规则外，几乎没有共享的规则，因此我们还针对同一张图提出相互递进的问题链：LUBM-3 上有七个，Follows 上有四个，Family 上有五个，其中前两个问题见列表 1。每个问题都是自包含的，16 个问题中有 12 个还提供了基于基础事实的参考程序，该程序定义了一个谓词，其规则与较早问题中的规则完全相同，例如 Follows 的可达关系，先是针对 account30 提问，之后又针对 account130 提问。当某个问题的程序读取一个保留的谓词，却不定义该谓词时，它会按名称复用较早问题中的谓词；当它用相同的规则陈述一个保留的谓词时，则会按其规则复用该谓词并使用相同的规则，从而形成空更新。否则，它会重述较早问题的规则，由引擎再次推导这些规则，或者重新定义所需内容。

我们将 Datalog 会话与另外两种会话进行比较，在后两种会话中，智能体计算出的内容会得到保留：一种是 Python REPL，其命名空间会在不同问题之间持久存在，如 CodeAct [20], 中的做法；另一种是 DuckDB 数据库，其表和视图会持久保留。视图是一个命名查询，数据库每次读取时都会重新执行该查询，而 DuckDB 没有会持续更新的视图。在这些问题链中，提示词和 Datalog 提示词一样，说明为某个问题计算出的关系通常之后还会用到；结果则会列出先前问题存储的内容。每条问题链都在五种会话中使用 Claude Haiku 5.5 总共提问 12 次；在隔离、全部保留、SQL 和 Python 会话中，则使用同一系列中规模更大的模型 Claude Opus 5.5，在相同的思考设置下提问 3 次。一个会话中的一个问题就是一个案例，因此，可供复用的 12 个问题为 Haiku 的每种会话提供 144 个案例，为 Opus 提供 36 个案例。

5.4 测量与统计

重放。为测量引擎在更大图上的耗时，我们按照各个样本运行智能体程序的顺序，在 LUBM-10、LUBM-30 和 LUBM-100 上重放这些智能体的 LUBM-3 程序。这些数据集分别包含同一生成器生成的 0 到 9、0 到 29 及 0 到 99 号大学，使用相同的 99 条规则和 130 万、400 万及 1340 万条基础事实。以各个规模计算出的标准答案为基准进行评分后，除 LUBM-100 上的合著问题（即通过合著链相连的作者）外，在 LUBM-3 上正确的每个 Datalog 程序在这里也都正确，而每个错误的程序仍然错误。本文报告的所有 LUBM 耗时，均为在 LUBM-100 上进行上述重放所得，使用引擎的 8 个工作线程，并在配备 18 核和 48 GB 内存的 Apple M5 Max 上运行。Follows 和 Family 上的会话，则在各自的图以及由同一生成器生成的更大图上重放：Follows 的社群数量分别扩大 10 倍、30 倍和 100 倍，Family 则包含 100 个和 1000 个家谱。除非另行注明运行次数，否则耗时均指单次运行。会话中某个问题的引擎时间，是该问题各次运行以及结束该问题的更新所耗费的时间。依赖模型的耗时通过再次发送请求进行测量：经由流式 API 发送请求，工具输入在生成时同步流式传输，并省略思考文本。同一模型在会话中的所有类型调用均在同一分钟内发送，各类型调用并行进行。本文的每次重放中，更新与从头求值所得的答案都相同。

统计学。 比较以所标明的单位进行配对，且检验均为双侧检验：配对测量采用精确符号检验或 Wilcoxon符号秩检验；两组会话采用精确 Mann–Whitney 检验；配对正确性采用精确 McNemar检验和 Cochran Q 检验；计数采用 Fisher精确检验；每个 Family 内采用 Holm校正。区间为 95% 自助法区间。

5.5 RQ1：智能体复用会话保留的内容

复用。（使用 Claude Haiku 5）在全部保留会话的所有案例里，只要能复用先前的谓词，就都会这样做：98 个按名称复用，46 个通过其规则复用；在库会话中，144 个案例里有 99 个通过智能体共享的谓词进行复用。Python 和 SQL 会话运行并提交了 786 个程序。没有 SQL 查询会为后续问题存储表

表 3：使用两种模型的链式会话。使用 Claude Haiku 5.5 时，全部保留智能体在每个案例和每个图中都会复用先前的谓词，而 Python 和 SQL 智能体在 288 个案例中有 2 个会复用。使用 Claude Opus 5.5 时，全部保留智能体和 SQL 智能体在 36 个案例中有 35 个会复用。当问题中有三行代码与先前问题的代码重复时，Python 或 SQL 问题会被重新表述。

		在能够复用的案例中			按图划分的复用案例				
会话	精确	按名称	按其规则	重新表述	Family	Follows	LUBM-3	规则	输出词元
Claude Haiku 5.5，每种类型每条链包含十二个会话，共有144个可复用的案例									
隔离	192/192	0	0	144	0/48	0/36	0/60	5.0	705
库	191/192	96	3	44	31/48	27/36	41/60	3.2	669
全部保留	192/192	98	46	0	48/48	36/36	60/60	2.8	582
Python	192/192	2		140	2/48	0/36	0/60		849
SQL	192/192	0		114	0/48	0/36	0/60		919
Claude Opus 5.5，每条链包含三种类型的会话各 3 个，36 个可复用案例									
隔离	48/48	0	0	35	0/12	0/9	0/15	5.4	403
全部保留	48/48	35	0	0	12/12	9/9	14/15	3.0	333
Python	48/48	12		24	0/12	3/9	9/15		607
SQL	48/48	35		1	12/12	9/9	14/15		412

格，而有一个 Python 会话定义了一个函数，供后续两个问题调用。在 144 个案例中，这些会话分别有 140 个和 114 个案例重新陈述了先前的代码。按问题配对后，在 12 个可以复用的问题中，全部保留智能体的复用频率均高于隔离、库、Python 和 SQL 智能体（符号检验，$p < 0.001$ 各）。没有会话定义包含其他规则的保留谓词。

更大的模型。使用 Claude Opus 5.5 时，全部保留智能体在 36 个案例中的 35 个复用了先前的谓词，SQL 代理也一样。在各自的九个会话中，SQL 代理会将某个关系（例如每个人的祖先）存储为表或视图，后续查询再读取它。Python 代理在 12 个案例中读取了已存储的名称，分布在九个会话中的四个会话里。

更短的程序。保留谓词后，在 12 个可复用的 Haiku 问题中（符号检验，$p < 0.001$ 各项），所需规则比独立会话更少，输出词元也更少。与 Python 和 SQL 会话相比，全部保留智能体在 16 个问题中的 15 个问题里输出的词元更少（Wilcoxon 符号秩检验，$p = 0.001$ 和 $p < 0.001$）：每个问题 582 个，而 Python 为 849 个、SQL 为 919 个；平均比 Python 少 267 个（95% 区间为 148 至 386），比 SQL 少 337 个（206 至 480）。Opus 全部保留智能体每个问题输出 333 个词元，而 Python 为 607 个、SQL 为 412 个；在 16 个问题中的 16 个和 15 个问题里分别更少（$p < 0.001$ 各项），而 SQL 代理的复用频率与之相同。

数据变更。使用 Haiku 时，我们还让这些链运行三次，并在问题之间插入和撤回基础事实，总共进行了八次变更；此后，若根据已存储的复用内容副本作答，16 个问题中有 11 个会答错。每次变更都通过会话自身的工具执行，下一个问题会列出新增和移除的事实。REPL 保留了谓词的智能体在这些问题的全部 33 个案例中都复用了至少一个谓词；在全部 11 个问题中，其复用频率高于隔离、Python 和 SQL 智能体（符号检验，$p = 0.001$），而五种会话类型的全部 240 个答案都完全正确。在 LUBM-100 上，全部保留会话执行一次变更的耗时中位数为 0.9 毫秒，而重新推导其保留谓词需要 31 毫秒；全部 9 次变更中都是如此（$p = 0.004$）。变更后的问题耗时则分别为 25 毫秒和 197 毫秒。

使用 Opus 时，在四种类型的会话中各进行三次会话，全部 192 个答案都完全正确，全部保留智能体在 33 个案例中的 31 个案例里复用了保留的谓词。SQL 智能体的提示中说明了数据可能变更；在全部九个会话中，它们都定义了视图而没有定义表，并在 33 个案例中的 30 个案例里读取了视图。

5.6 RQ2：复用节省了什么

智能体的链。表 4 将 LUBM-100 上的全部保留会话与 SQL 和 Python 会话进行了比较。使用 Haiku 时，针对合著关系的前两个问题中，全部保留会话的引擎时间中位数分别为 48 和 25 毫秒。独立会话分别耗时 722 和 755 毫秒，其中 335 和 350 毫秒用于重新推导该关系，其余时间则是在问题结束时撤销该关系；库会话则分别耗时 416 和 452 毫秒。SQL 会话耗时 3.8 和 3.3 秒，Python 会话耗时 5.2 和 5.4 秒，而且两者都会在每个问题中重新计算该关系。一个 SQL 会话从问题中提到的单个作者开始递归，其三个合著问题耗时 51 至 58 毫秒。在十二个全部保留会话中，有两个会话里的智能体推导了所有作者的合著链，其余十个则只推导了一位作者的合著链。在这两个会话中的第一个会话里，插入该闭包耗时 18.4 秒；随后两个问题分别耗时 0.18 和 0.24 秒，而重新推导该闭包时分别耗时 31.5 和 30.3 秒，以上数据均为三次运行的中位数。

使用 Opus 时，三个 SQL 会话中有两个将合作者关系对存储为表，后续的两个问题耗时 41 至 95 毫秒，而第三个会话则耗时 4.0 至 4.6 秒。全部保留会话耗时 11 至 34 毫秒，而 Python 会话的中位数分别为 7.7 秒和 10.6 秒。当数据发生变化时，Opus 没有任何 SQL 会话存储表，而是每次提问都重新计算合作者关系：三个关于合作者关系的问题分别耗时 3.9 秒、4.5 秒和 3.9 秒；相比之下，全部保留会话分别耗时 106 毫秒、69 毫秒和 38 毫秒，其中一次变更的代价为 0.9 毫秒。因此，REPL 保留的谓词读取起来像表一样，并且能够像视图一样保持最新。

端到端。根据重新发送各智能体的模型调用来测量生成时间，包含七个问题的 LUBM 链中，Haiku 全部保留会话的中位数为 31.2 秒：十个会话用时为 25.4 至 36.7 秒，另有两个会话因推导出

表 4：全部保留模式下的 Datalog 交互式环境，与链中的 SQL 和 Python 会话相比，每个智能体使用各自的程序。时间取自 LUBM-100，按会话计算中位数及范围：问题的引擎或执行时间，以及智能体生成链的时间。使用 Haiku 时，Datalog 智能体回答递归问题并生成链所用的时间更短。使用 Opus 时，SQL 智能体会存储表，其链所用的时间与 Datalog 智能体相当。

测量指标	Datalog 交互式环境	SQL 会话	Python 会话
Claude Haiku 5.5，每个工具十二个会话			
关于共同作者的问题	0.34 秒（0.31 至 18.6）	3.30 秒（0.05 至 4.07）	6.75 秒（5.19 至 7.29）
其上的第一个问题	48 毫秒（22 至 324）	3.83 秒（0.05 至 5.01）	5.24 秒（4.98 至 5.65）
其上的第二个问题	25 毫秒（13 至 237）	3.29 秒（0.06 至 4.18）	5.37 秒（4.95 至 5.87）
七个问题的链条	31.2 秒（25.4 至 51.2）	48.2 秒（37.5 至 57.3）	56.3 秒（51.1 至 63.2）
Claude Opus 5.5，每个工具进行三次会话			
合著问题	0.37 秒（0.13 至 0.38）	0.18 秒（0.16 至 3.98）	9.20 秒（8.12 至 9.72）
其上的第一个问题	32 毫秒（31 至 34）	50 毫秒（41 至 4,647）	7.74 秒（7.57 至 7.93）
其上的第二个问题	27 毫秒（11 至 30）	95 毫秒（53 至 4,038）	10.6 秒（8.59 至 11.6）
七个问题的链条	46.0 秒（45.9 至 47.8）	45.2 秒（44.9 至 58.9）	96.8 秒（83.3 至 99.9）

所有作者的闭包，用时分别为 45.0 秒和 51.2 秒。SQL 会话的中位数为 48.2 秒，Python 会话为 56.3 秒（表 4）。在每种类型的十二个会话中，全部保留链的平均用时比 SQL 链短 15.6 秒（95% 区间为 10.3 至 20.1 秒），比 Python 链短 22.3 秒（17.8 至 26.3 秒）；两项差异均显著（精确 Mann–Whitney 检验，$p < 0.001$ 每项）。按问题计算，全部保留会话在全部 7 个问题上的用时都少于 SQL 会话和 Python 会话（符号检验，$p = 0.016$ 每项）。使用 Opus 时，全部保留链和 SQL 链的中位数分别为 46.0 秒和 45.2 秒，Python 链则为 96.8 秒。

收益从何而来。使用 Haiku，SQL 链与全部保留链之间的 17 秒差距中，约有 8 秒来自生成，因为全部保留智能体写出的词元更少；另有 9 秒来自执行。独立会话完成这条链的中位时间为 36.7 秒，长于全部保留会话（精确 Mann–Whitney 检验，$p = 0.02$）；两者之间的差异来自复用：全部保留会话每个问题的平均耗时少 0.7 秒（95% 区间为 0.1 至 1.5 秒，Wilcoxon 符号秩检验，$p = 0.047$）。与其自身上一次从头评估的程序相比，全部保留模式下每个问题的中位耗时为 3.93 秒，而前者为 4.29 秒；84 个问题中有 26 个耗时减少超过 5%，且全部都是共同作者问题：其中十个会话减少了 6% 至 14%，而保留了闭包的两个会话减少了 32% 至 90%。

图规模。在 LUBM 的四种规模下重放时，Haiku 全部保留问题的运行在保留的谓词之上，耗时中位数分别为 5.0、5.7、8.3 和 22 毫秒；而从头开始使用其谓词时，耗时分别为 12、25、61 和 195 毫秒。在 LUBM-100 的 84 个问题中，有 70 个问题的耗时更短（Wilcoxon符号秩检验，$p < 0.001$）。对于社群数量分别为原来的 10、30 和 100 倍的 Follows，事实数分别为 9、28 和 92 千条，可达关系分别包含 0.7、3.7 和 18.3 百万对；第一个问题之后的问题，在 Haiku 全部保留会话中耗费的引擎时间中位数分别为 55 毫秒、205 毫秒和 0.63 秒，而从头开始则分别为 191 毫秒、908 毫秒和 4.69 秒。对于包含 100 和 1,000 个家谱、事实数分别为 47 和 447 千条的 Family，耗时分别为 22 和 160 毫秒，而从头开始则为 47 和 396 毫秒。在这些规模下的每种情况下，会话耗时都更短：Follows 的 36 个问题全部如此；在规模为 100 倍时，9 个问题中有 9 个如此，此处重放了三个会话；Family 的 48 个问题也全部如此（符号检验，$p \le 0.004$）。在两个图各自的规模下，耗时分别为 14 毫秒对 12 毫秒，以及 7 毫秒对 4 毫秒。

5.7 研究问题3：修订成本

智能体的修订。修订是指智能体针对同一个问题，在先前运行过的程序之后运行的程序。在三种设置中，针对独立问题的智能体共运行了 61 次修订：LUBM 上有 33 次，其中 31 次在 LUBM-100 上重放；Family 和 Follows 上有 28 次。我们按照测试框架的方式应用每次修订：将其作为一次更新，应用于持有智能体先前程序的引擎，并将其与在该条件的持久化数据上从头评估修订后程序进行比较。在 LUBM-100 上，删除规则的 23 次修订作为更新的耗时中位数为 73 毫秒，而从头开始为 37 毫秒；这是因为更新会撤回已删除规则所产生的事实，而从头运行的引擎从未推导出这些事实（Wilcoxon符号秩检验，$p = 0.11$）。有 7 次修订原样重新提交先前程序，这些都是空更新，耗时为 1.2 毫秒，而从头开始为 41 毫秒（$p = 0.03$）。在 Family 和 Follows 上，28 次修订作为更新的耗时中位数为 3.4 毫秒，而从头开始为 1.4 毫秒（$p = 0.002$）。

一条规则发生变化。单规则修订会改变程序中的一条规则，并保留其他规则：定义查询谓词的规则会失去最后一个条件；如果它只有一个条件，则程序中的一个常量会指明相邻实体。我们将此修订应用于四种图规模下智能体的每个 LUBM 程序，分别将其作为更新应用于持有该程序的引擎，以及在持久化数据上从头运行修订后的程序。图1展示了每种规模下155次修订的中位数（LUBM-100下为151次），并列比较143次首次求值的中位数（LUBM-100下为137次）；首次求值是将样本中第一个获接受的程序作为更新插入。

首次求值在两种形式下的成本相同：在 LUBM-100 中，从头开始所需时间与更新所需时间之比的中位数为 1.01（95% 区间为 0.99 至 1.03）。单规则修订会将变更的规则作为更新来处理，并从头开始处理整个程序，因此差距会随着保留规则所推导出的内容增多而扩大。在四种规模下，更新分别在 110、131、135 和 132 次修订中更快，中位倍数分别为 1.1、1.4、1.8 和 2.0（对每种规模都进行了测量的 151 次修订执行 Page趋势检验，$p < 0.001$）。在 LUBM-100 中，该倍数的 95% 区间为 1.7 至 4.0；按问题划分，则介于 1.0 至 154 之间。通用合著程序所保留的规则会推导出每位作者的全部合著链，是其中最极端的情况：其修订作为更新时耗时分别为 1.9、2.2、1.6 和 2.1 毫秒，而从头开始则耗时 0.33、

表5：三种设置下的准确率：启用自适应思考和关闭思考的 Claude Haiku 5.5（每种条件69个样本），以及 LFM3-1.7B（23个样本）。精确匹配、每种数据在四种语言上的测试结果，以及每个样本的平均输出词元数（包括思考过程）。

	精确			每个样本的输出词元数		
条件	思考	不思考	LFM3-1.7B	思考	不思考	LFM3-1.7B
Datalog-M	66/69	62/69	12/23	569	324	12,352
SQL-M	69/69	69/69	12/23	649	365	9,215
SPARQL-M	67/69	66/69	9/23	710	327	8,515
Python-M	68/69	69/69	9/23	700	469	8,147
Datalog-B	65/69	60/69	9/23	948	466	16,498
SQL-B	66/69	63/69	12/23	916	503	10,716
SPARQL-B	67/69	55/69	8/23	1,084	469	8,887
Python-B	67/69	67/69	9/23	923	577	8,534
Cochran Q 检验，M：p	0.29	0.002	0.48			
Cochran Q 检验，B：p	0.72	0.003	0.52			

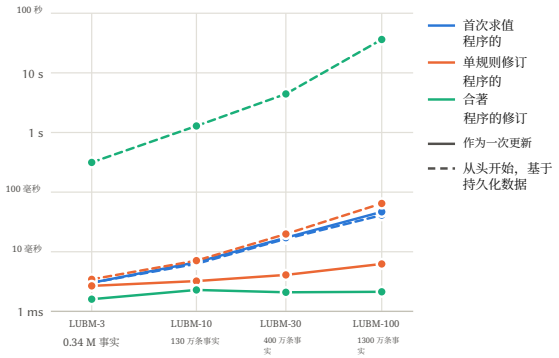


图1：智能体的 LUBM 程序在四种图规模下重放，每个程序取中位数；实线表示作为一次更新，虚线表示基于持久化数据从头运行同一程序。

1.39、6.27 和 36.3 秒；这些数值分别是五次运行的中位数，以及在 LUBM-100 上三次运行的中位数。

5.8 研究问题4：小众语言中的准确率

我们预计，Datalog 在模型训练数据中的出现频率会远低于 SQL 或 Python，但在进行思考时，模型使用 Datalog 的准确率仍与使用其他语言一样高。在两个模型的链中，全部保留、SQL 和 Python 智能体都能准确回答每个问题（表 3）。对于独立问题（表5），在进行思考时，每种条件都能准确回答 65 至 69 个样本，且四种语言在两类数据上均无差异（Cochran Q 检验，$p = 0.29$ 和 0.72）。在相同数据上，通过对 69 个问题与 epoch 对进行精确 McNemar检验，将每种语言与其他语言分别比较后，Datalog 条件均无显著差异（六次比较中为 $p \geq 0.25$ ）：Datalog 条件准确回答的样本数少 1.4 至 4.3 个百分点，95% 区间为 −11.6 至 +4.3 个百分点。不进行思考时，每种条件都能准确回答 55 至 69 个样本，且四种语言存在差异（$p = 0.002$ 和 0.003）。经过 Holm校正后，Datalog 条件与其他语言之间均无差异达到显著水平（$p \geq 0.09$）。LFM3-1.7B 对每种条件下的 23 个样本都能准确回答其中 8 至 12 个，且四种语言之间无差异（p =0.48 和 0.52）。只有在 Datalog 中，它才能回答三个递归问题中的任何一个：6 次尝试中答对 3 次，而其他语言的 18 次尝试中一次也未答对（Fisher精确检验，$p = 0.01$）。

5.9 普适性

RQ1 中的复用在三种图上均成立，无论数据固定还是变化，也无论使用哪种模型（表 3）。在所有三种图上，保留谓词所节省的时间都会随图规模增大而增加，也会随保留规则推导出的内容增多而增加（图1）。在 LUBM-100 上，单规则修订作为更新的速度在三种模型设置下都更快，中位数分别快 2.6、1.9 和 2.9 倍；在两种数据类型下也都更快，分别快 2.0 和 2.1 倍。

6 结论

Datalog 交互式环境为大语言模型代理提供了一种工具，其状态是由它所编写的规则推导出的事实集合，并由动态 Datalog 引擎持续更新。提示缓存会复用模型对会话共享内容的处理，而 Datalog 交互式环境则会复用对这些内容的求值。使用两种模型时，智能体在 180 个案例中的 179 个案例里复用了交互式环境保留的谓词；数据变化时，则在 66 个案例中的 64 个案例里复用。SQL 和 Python 代理生成的输出词元多出 23% 至 82%。使用 SQL 表或 Python 命名空间时，小模型几乎不会为后续问题保存任何内容；在拥有 1300 万条基础事实的数据上，其七问题链的耗时中位数为 48 秒和 56 秒，而使用交互式环境时为 31 秒。大模型通过将表存储在 SQL 中，可以达到相同的耗时；数据变化时，它会存储视图，而每次读取时都要重新求值。交互式环境保留的谓词既像表一样可供读取，又像视图一样保持最新；它节省的时间在所有三种图上都会随图规模增大而增加。与 SQL 和 Python 相比，Datalog 是一种小众语言；经过思考后，模型使用它也能同样准确地作答。

致谢

本工作得到爱沙尼亚研究委员会项目（PRG2764）的资助。

参考文献

[1] Serge Abiteboul、Richard Hull 和 Victor Vianu。1995 年。《数据库基础》。Addison-Wesley。[2] Anthropic。2026 年。程序化工具调用。Claude API 文档，https://platfor m.claude.com/docs/en/agents-and-tools/tool-use/programmatic- tool-calling。[3] Anthropic。2026 年。提示缓存。Claude API 文档，https://platform. claude.com/docs/en/build-with-claude/prompt-caching。[4] Mihai Budiu、Tej Chajed、Frank McSherry、Leonid Ryzhyk 和 Val Tannen。2023 年。DBSP：丰富查询语言的自动增量视图维护。《VLDB基金会会刊》16，7（2023），1601–1614。doi:10.14778/3587136.3587137[5] Bruno Rucy Carneiro Alves de Lima、Merlin Kramer 和 Kalmer Apinis。2023 年。论差分数据流在高度动态环境下进行Datalog解释的适用性。载于《2023年第6届人工智能与云计算会议论文集》（AICCC 2023）。ACM，日本京都，218–225。doi:10.1145/3639592.3639622[6] Bruno Rucy Carneiro Alves de Lima、Merlin Kramer、Kalmer Apinis 和 Kristo- pher Micinski。2024 年。将动态 Datalog 程序的增量求值表示为高阶 DBSP 程序。载于《Datalog在学术界和工业界复兴国际研讨会》第5届论文集（Datalog-2.0 2024）（CEUR研讨会论文集，第3801卷）。2–16。https://ceur-ws.org/Vol-3801/paper1.pdf[7] Bruno Rucy Carneiro Alves de Lima、Merlin Kramer 和 Victor Hen- rique Cabral Pinheiro。2024 年。动态 T-boxes 上的分布式增量本体推理。载于《2024年第7届人工智能与云计算会议论文集》（AICCC 2024）。ACM，日本东京，423–428。doi:10.1145/3719384.3719446[8] Bang Fu 和 Di Feng。2023 年。GPTCache：面向大语言模型应用的开源语义缓存，实现更快的答案生成和成本节约。载于《第3届自然语言处理开源软件研讨会论文集》（NLP- OSS 2023）。212–218。doi:10.18653/v1/2023.nlposs-1.24[9] In Gim、Guojun Chen、Seung-seob Lee、Nikhil Sarda、Anurag Khandelwal 和 Lin Zhong。2024 年。提示缓存：用于低延迟推理的模块化注意力复用。载于《机器学习与系统论文集》（MLSys）。arXiv:2311.04934[10] Yuanbo Guo、Zhengxiang Pan 和 Jeff Heflin。2005 年。LUBM：OWL知识库系统的基准测试。《网络语义学期刊》3，2-3（2005），158–182。doi:10.1016/j.websem.2005.06.005[11] Ashish Gupta、Inderpal Singh Mumick 和 V. S. Subrahmanian。1993 年。增量维护视图。载于《ACM SIGMOD数据管理国际会议论文集》。157–166。doi:10.1145/170035.170066[12] Herbert Jordan、Bernhard Scholz 和 Pavle Subotić。2016 年。Soufflé：关于程序分析器的综合。载于《计算机辅助验证国际会议论文集》（CAV）（LNCS，第 *9780* 422 430 10 1007 978 3 319 41540 6_23 卷）。–。[13] doi：。/ - - - - Qing Lyu、Shreya Havaldar、Adam Stein、Li Zhang、Delip Rao、Eric Wong、Mar- ianna Apidianaki 和 Chris Callison-Burch。2023 年。忠实的思维链推理。载于《第13届自然语言处理国际联合会议暨计算语言学协会亚太分会第3届会议论文集》（第1卷：长篇论文）。计算语言学协会，努沙杜瓦，巴厘岛，305–329。doi:10.18653/ v 1 2023 20 / .ijcnlp-main。Frank McSherry、Derek G. Murray、Rebecca Isaacs 和 Michael Isard。2013 年。差分数据流。载于《第6届创新数据系统研究双年会议论文集》（CIDR）。[15] Boris Motik、Yavor Nenov、Robert Piro 和 Ian Horrocks。2019 年。重新审视Datalog物化结果的维护。《人工智能》269（2019），76–136。doi:10.1016/j.artint.2018.12.004[16] Liangming Pan、Alon Albalak、Xinyi Wang 和 William Yang Wang。2023 年。Logic-LM：利用符号求解器增强大语言模型，实现忠实的逻辑推理。载于《计算语言学协会论文集：EMNLP 2023研究成果》。计算语言学协会，新加坡，3806 3824 10 18653 1 2023 248 –。[17] doi：。/v / .findings-emnlp。Leonid Ryzhyk 和 Mihai Budiu。2019。差分 Datalog。载于《Datalog 2.0研讨会论文集》（CEUR研讨会论文集，第2368卷）。56–67。[18] Tamás Szabó、Gábor Bergmann、Sebastian Erdweg 和 Markus Voelter。2018 年。在 Datalog 中对基于格的程序分析进行增量化。《ACM编程语言论文集》2，OOPSLA（2018），139:1–139:29。doi:10.1145/3276509[19] Guanzhi Wang、Yuqi Xie、Yunfan Jiang、Ajay Mandlekar、Chaowei Xiao、Yuke Zhu、Linxi Fan 和 Anima Anandkumar。2024 年。Voyager：具备大语言模型的开放式具身智能体。《机器学习研究汇刊》（2024）。arXiv:2305.16291[20] Xingyao Wang、Yangyi Chen、Lifan Yuan、Yizhe Zhang、Yunzhu Li、Hao Peng 和 Heng Ji。2024 年。可执行代码动作激发更出色的大语言模型智能体。载于第*41*届国际机器学习国际会议论文集（*ICML*）。arXiv:2402.01030[21] Zora Zhiruo Wang、Jiayuan Mao、Daniel Fried 和 Graham Neubig。2025。智能体工作流记忆。载于 第*42*届国际机器学习国际会议论文集（*ICML*） （*PMLR*，第*267*卷）。63897–63911。[22] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik Narasimhan 和 Yuan Cao。2023。ReAct：在语言模型中协同推理与行动。载于国际学习表征会议（*ICLR*）。arXiv:2210.03629[23] Alex L. Zhang、Tim Kraska 和 Omar Khattab。2025。递归语言模型。arXiv:2512.24601 [cs.AI][24] Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Sun、Jeff Huang、Cody Hao Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E. Gonzalez、Clark Barrett 和 Ying Sheng。2024。SGLang：高效执行结构化语言模型程序。载于《神经信息处理系统进展》第*37*卷（*NeurIPS 2024*）。arXiv:2312.07104