Intent-Based Access Control: Using LLMs to Intelligently Manage Access Control

Pranav Subramaniam[1], Sanjay Krishnan[1]
[1] University of Chicago
{psubramaniam,skr}@uchicago.edu

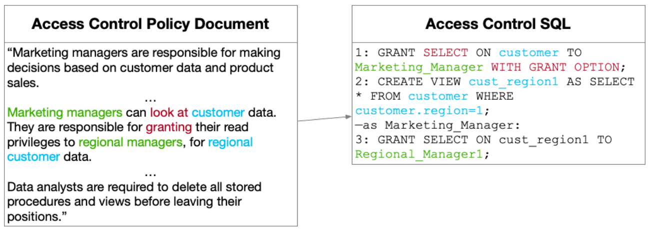
ABSTRACT

In every enterprise database, administrators must define an access control policy that specifies which users have access to which assets. Access control straddles two worlds: policy (organization-level principles that define who "should" have access) and process (database-level primitives that actually implement the policy). Assessing and enforcing process compliance with a policy is a manual and ad-hoc task. This paper introduces a new paradigm for access control called Intent-Based Access Control for Databases (IBAC-DB). In IBAC-DB, access control policies are expressed more precisely using a novel format, the *natural language access control matrix* (NLACM). Database access control primitives are synthesized automatically from these NLACMs. These primitives can be used to generate new DB configurations and/or evaluate existing ones. This paper presents a reference architecture for an IBAC-DB interface, an initial implementation for PostgreSQL (which we call LLM4AC), and initial benchmarks that evaluate the accuracy and scope of such a system. We find that our chosen implementation, LLM4AC, vastly outperforms other baselines, achieving near-perfect F1 scores on our initial benchmarks.

1 INTRODUCTION

Over the last several years, there has been a proliferation of easy-to-use software tools for data science. Organizations of all sizes and from all industries are applying data-driven models to make decisions. This trend, which is often called the democratization of data science [14], is as worrying as it is exciting. On one hand, some of humanity's most difficult scientific challenges are being addressed with data science from particle physics to drug discovery. On the other hand, there are numerous cautionary tales of data misuse, data breaches, data quality issues, and improper data sharing [17, 21–23, 27]. Data governance, the process of managing the availability, usability, integrity, and security of the data in enterprise systems, has consequently emerged as a key discipline in enterprise data management. Access control is one of the most important technical challenges in data governance, where a system must ensure that only authorized users can access particular data assets.

In any sizeable organization, it is impractical to exhaustively map individual users to data assets. Thus, role-based access control (RBAC) was a key innovation in data governance and is widely supported by data management platforms where groups of users can be selectively given access [26]. Such forms of access control are well-studied in enterprise SQL databases, where roles, views, and GRANT statements are already used to restrict user access to data [10, 11, 13]. Similarly, many databases additionally support


Access Control Policy Document

"Marketing managers are responsible for making decisions based on customer data and product sales.

...

Marketing managers can look at customer data. They are responsible for granting their read privileges to regional managers, for regional customer data.

...

Data analysts are required to delete all stored procedures and views before leaving their positions."

Access Control SQL

```
1: GRANT SELECT ON customer TO
Marketing_Manager WITH GRANT OPTION;
2: CREATE VIEW cust_region1 AS SELECT
* FROM customer WHERE
customer.region=1;
—as Marketing_Manager:
3: GRANT SELECT ON cust_region1 TO
Regional_Manager1;
```

Figure 1: Access Control Policies (Written in Natural Language Documents) to Process (Written in SQL GRANT statements)

discretionary access control (DAC) where roles can be given delegation privileges. RBAC and DAC help reduce the administrative burden in defining access control rules for a database.

Despite these features, access misconfiguration is still widely described as a major cyber-security risk and a business cost [29]. Access control fundamentally straddles two different worlds: policy and process. *Policy* describes an organization's intentions with respect to access control, data retention, authorization, etc. These are broad principles that govern how an organization chooses to delegate access to data assets. Policies are often constructed with legal or regulatory advice and are specified in documents, reports, and guidelines. *Process*, on the other hand, describes the translation of these policies into actionable code and rules that implement such functionality. Going from policy to process often requires identifying which text in a document corresponds to access control rules in a database, and which database components (schema elements, database roles, etc.) these rules reference.

For example, consider the access control policy document shown in Figure 1. These documents form a critical part of an organization's security infrastructure and are needed for compliance [4]. An access control policy document is typically written in natural language (NL), and often contains statements that are not relevant for database access control (e.g., job descriptions, such as "Marketing managers are responsible for making decisions based on customer data and product sales."). Further, even after one has identified the relevant parts of a policy document (e.g., "Marketing managers can look at customer data. They are responsible for granting their read privileges to regional managers, for regional customer data."), one must identify which permissions are being given on which tables/views for which roles/users, and generate the appropriate SQL role/user creation statements, view creation statements, and GRANT statements. Lastly, once an access control policy has been implemented in a database, one should be able to audit the implementation to determine whether the policy was implemented correctly.

Often, the individuals or groups in charge of developing policy, e.g., security experts, organizational counsel, and insurance providers, do not have the expertise to advise their correct implementation in a database system. As the database ecosystem grows with new tools, each one has its own access control idiosyncrasies and specification languages. Today's IT practice leaves both policy makers and database administrators with a lack of information that would help verify that a given access control implementation matches an organization's intent. As a step towards a solution, this paper studies *directly* specifying access control policies in natural language, and automatically implementing and auditing such policy implementations for correctness. Specifically, we make the following contributions:

- **IBAC-DB**: A new paradigm for database access control called Intent-Based Access Control for Databases (IBAC-DB)[1]. An IBAC-DB system does not just specify access control rules, but can intelligently evaluate these rules against natural language policies to identify potential violations.
- **LLM4AC**: A system built from a reference architecture for an IBAC-DB implementation that provides a novel format for specifying access control policies, automatically implementing them, and comparing implementations to policies for correctness.
- **Differencing Definition and Algorithm**: A novel definition and algorithm for *differencing* access control policies with their implementations to compare policies with their implementations.
- **Differencing Benchmark Design**: a novel benchmark for testing that an IBAC-DB system can correctly determine whether an access control policy matches its implementation.

Specifically, IBAC-DB enables easy identification and extraction of database access control rules from NL using the *natural language access control matrix* (NLACM), a novel abstraction. We implement access control policies by adapting NL2SQL systems to generate access control SQL, a capability which no current NL2SQL systems possess. We compare a policy to its implementation using a novel *differencing* procedure, which uses LLMs to determine which access control rules were implemented that do not comply with the policy. We find that the accuracy of this procedure increases greatly when we prune the candidate matches input to the LLM that do not have matching DB literals first. This allows LLM4AC to achieve near-perfect F1 scores on our benchmarks (e.g., F1 scores of 0.99).

2 BACKGROUND

First, we introduce some notation and formalism to scope the IBAC-DB problem.

2.1 Access Control Definitions

Let us consider access control to a single database. Over this database, there are a set of possible users U. These users can run a set of possible SQL statements S over this database. An access control *scheme* is a set of rules that specifies who should be given access under what conditions. Formally, an access control scheme for a database is a program, that determines whether a particular user is allowed to run the desired SQL statement over the database:

$$\mathcal{P}(user, stmt) : U \times S \mapsto \{allowed, denied\}$$

[1]Inspired by similarly named techniques in network and firewall configurations.

$\mathcal{P}$ describes a literal implementation of access control for a particular database. For example, in an SQL database, $\mathcal{P}$ is defined as a sequence of GRANT statements. However, such programs are not created in a vacuum. An access control *policy* is a set of rules that specifies who should be given access to what resources. For example, an organization's regulatory posture may dictate who has access to data (e.g., only relevant doctors may view electronic health records). Or, an organization's security team may implement a particular minimal privilege strategy (e.g., only senior management can modify customer records). Whatever the reason, these decisions are described at a higher level of abstraction than $\mathcal{P}$.

SOC-2 compliance requires that access control policies be specified in an access control matrix [4]. An access control matrix defines rows that are users (or groups) and columns that are data assets. The cells of this matrix specify the allowed operations of that user on the asset. Similarly, research institutions that work with identifiable information need to submit data management plans written in document form to institutional review boards and funding agencies [5, 6]. In general, access control policies are written in natural language or quasi-natural language (e.g., an access control matrix) by a database non-expert. To correctly implement such policies, one must determine which parts of the policy represent access control rules for the database, and which components of the database those parts refer to. This can require multiple rounds of communication between a policy expert and a database expert (e.g., a database administrator) [1, 2, 7, 16].

2.2 Auditable Access Control

We have to compare two access control schemes $\mathcal{P}$, which is a scheme specified on the database in SQL, and $\mathcal{P}^*$, which is a scheme specified in natural language. Hereafter, we will refer to $\mathcal{P}$ as an access control "implementation" and $\mathcal{P}^*$ as an access control "policy". We fully understand that there is inherent imprecision in natural language and there might not exist an unambiguous $\mathcal{P}^*$; however, we scope our initial exploration of this problem in such a way to avoid such ambiguities. The main goal is to be able to check for compliance, or formally:

Definition 1 (Compliance). *An access control implementation $\mathcal{P}$ is compliant against a policy $\mathcal{P}^*$ if and only if*

$$\forall (u, t) \in U \times T : \mathcal{P}(u, t) \implies \mathcal{P}^*(u, t)$$

This definition states that every allowed SQL statement in $\mathcal{P}$ is also allowed in $\mathcal{P}^*$, or in other words, $\mathcal{P}$ is at least as restrictive $\mathcal{P}^*$. Many solutions have been developed that enable automated implementation and verification of access control policies. However, we will see that these come at the cost of restricting access control rules that can be expressed.

One solution is to adhere to an access control paradigm for specifying access control policies, such as role-based access control (RBAC), discretionary access control (DAC), etc. Adhering to these paradigms alone can greatly simplify defining and implementing access control policies. For example, if one defines a role-based access control policy on a database, one need only implement it by concatenating the SQL describing the roles, tables/views, and privileges into a SQL GRANT statement. However, implementing such policies through a database system alone requires database expertise and explicit enumeration of the complete set of access control rules for all database users.

To reduce the expertise and manual effort needed to implement an access control policy, prior work has focused on allowing access control to be defined via programming languages for access control [3, 20, 34], which can then be translated into database privileges, often with guarantees on the correctness of translation, or automated verification via model-checking.

These programming languages allow automatic and provably correct implementation of access control policy. However, they restrict the access control rules that can be expressed, compared to natural language. For example, ShillDB, a recent contract language for database access control, cannot express access control rules on nested queries [34], but natural language could easily express access control rules on such a query (e.g., Q4, the order priority checking query, from TPC-H).

Based on existing access control solutions, we find that on the one hand, one can use NL to represent policies, but then implementation and auditing for correctness are manual. On the other hand, you can use access control paradigms or languages which automatically and correctly implement access control policies, but then the expression of access control rules is limited.

2.3 Problem Statement

Therefore, in this paper, we solve the problem: *how do we enable access control policies to be expressed in natural language, while still allowing automated, correct implementation and auditing of access control schemes?* We address this problem by making the following contributions:

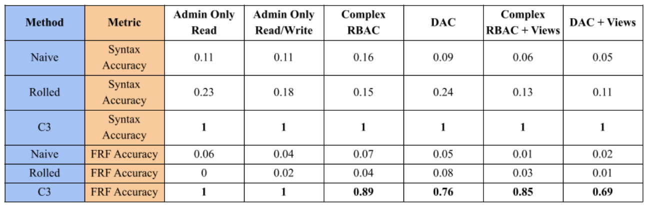
(1) We propose a new paradigm for access control called *Intent-based Access Control for Databases* (IBAC-DB), which uses the *natural language access control matrix* (NLACM) as the input.
(2) We propose LLM4AC, a reference architecture for an IBAC-DB interface.
(3) To allow for policy auditing, we define *differencing*, a procedure for comparing an access control policy and its implementation. Note that we cannot formally verify the correctness of implementations, due to the imprecision of natural language. However, we find that our differencing procedure for auditing policies is mostly correct in practice.

2.4 Existing NL2SQL Capabilities

NL2SQL systems seem helpful in implementing NL access control policies as SQL. Current NL2SQL capabilities focus on converting natural language questions about information in a database to queries (typically SELECT queries) [15, 24, 30, 32]. The state-of-the-art NL2SQL methods develop procedures for prompting LLMs to generate accurate SQL, such as ChatGPT [15] or GPT-4 [19].

However, to our knowledge, current NL2SQL benchmarks do not contain examples that generate access control queries [28, 32, 33], and the database system used for evaluation in most cases is SQLite, which does not support access control.

Further, it is not obvious how to bridge this gap between state-of-the-art NL2SQL methods and the access control use case. For example, enhancing LLM-backed NL2SQL systems by adding naive prompting of ChatGPT can produce an incorrect answer. When given the prompt, *Write the postgresql commands to implement such access control, given the database schema*, two of the queries generated by ChatGPT are "GRANT CREATE ON ALL TABLES IN SCHEMA public TO data_architect;" and "GRANT CREATE ON ALL VIEWS IN SCHEMA public TO data_architect;". The second query is unnecessary, and uses the word "VIEWS" in the grant statement which does not exist.



Method	Metric	Admin Only Read	Admin Only Read/Write	Complex RBAC	DAC	Complex RBAC + Views	DAC + Views
Naive	Syntax Accuracy	0.11	0.11	0.16	0.09	0.06	0.05
Rolled	Syntax Accuracy	0.23	0.18	0.15	0.24	0.13	0.11
C3	Syntax Accuracy	**1**	**1**	**1**	**1**	**1**	**1**
Naive	FRF Accuracy	0.06	0.04	0.07	0.05	0.01	0.02
Rolled	FRF Accuracy	0	0.02	0.04	0.08	0.03	0.01
C3	FRF Accuracy	**1**	**1**	**0.89**	**0.76**	**0.85**	**0.69**

Figure 2: Comparison of NL2SQL Translation by LLM method, with respect to syntax and Forward-Reverse-Forward (FRF) Accuracy. Columns are access control policies.

Simply using NL2SQL methods as-is by inputting access control policy documents can also be inaccurate. Consider a straightforward access control rule: "Grant the user John select access on the customer table with the option of passing down this privilege." C3 [15] translates this NL to the SQL "GRANT SELECT ON customer TO John;". This is incorrect because it ignores the option of passing down the privilege–the query must be suffixed with a "WITH GRANT OPTION".

On the other hand, we find that C3 more accurately generates all other GRANT statements when given a natural language sentence that clearly spells out the role, table/view, and privilege, compared to naive prompting, or prompts engineered using typical prompt engineering techniques (e.g., few-shot learning, chain-of-thought prompting, etc.). In Figure 2, we compare naive prompting of ChatGPT (the "Naive" method), a chain-of-thought prompting solution (the "Rolled" method), and C3, a state-of-the-art NL2SQL system, with respect to accuracy of the generated syntax, and forward-reverse-forward (FRF) accuracy. We systematically perform this comparison on access control policies containing role hierarchies of various complexities, view creation, and discretionary access control. We synthetically generate 100 policy documents of each type, and compute the accuracy based on the syntax/FRF accuracy of the synthesized SQL script. We see that C3's syntax and FRF accuracy far exceeds prompting solutions. This suggests that altering NL2SQL methods for access control may facilitate implementation of access control in databases.

Based on these examples and initial experiments, rather than trying to train NL2SQL systems to 100%, we hypothesize if access control policies can be formatted to clearly reflect a role, privilege, and view, and we can correctly represent access control policies in terms of the database roles, views, and privileges that correspond to the policy, then NL2SQL methods adapted for access control can correctly implement access control policies.

3 ACCESS CONTROL VIA INTENTS

In this section, we explain what an access control policy intent is, and bridge the gap between NL policy and process for RDBMSs via a new access control scheme called *Intent-based access control for* *databases*, or IBAC-DB. Then, we present our system for enabling this paradigm, LLM4AC.

3.1 Specifying Intents

Database access control rules are specified as GRANT statements, each of which consists of the allowed SQL operators for a role/user on a table/view. We refer to NL or SQL that references key elements of access control rules (roles/users, tables/views, or permitted SQL operations) as *intents*. Then, intent-based access control (IBAC-DB) enables the specification of access control policies using intents. IBAC-DB achieves this using a novel abstraction called the *natural language access control matrix*, or NLACM. A NLACM specifies database access control rules as follows: a row represents privileges for a database role/user, a column represents privileges for a table/view, and a cell represents the allowed SQL operators. In this way, each matrix cell represents an access control rule.

Definition of NLACM.: Let D be a database schema, which consists of: (i) table schema definitions, (ii) view definitions, (iii) role and user definitions. Then, a NLACM is a $m \times n$ matrix where each row represents the database privileges of a role/user in the database, and each column represents a table/view. The (i, j)th cell represent the privileges of role i on table/view j.

NLACMs specify access control rules not already implemented in the database. Toward this, they have the following constraints:

- Each role/user appears in only one row of the NLACM.
- Each view appears in only one column of the NLACM.
- (Principle of Failsafe Defaults): There can be roles, views in the DB that do not appear in the NLACM, and we assume they do NOT have any privileges defined.
- Empty cells are permitted, indicating no privilege is assigned.
- Non-empty cells indicate which SQL operators are permitted for a given role/user on a given view.
- Each role and each view can be expressed either as NL or SQL, and the privileges can be a list of SQL operators, or NL.
- Columns are table names expressed in NL, or view definitions that appear in the database.

We provide a solution for specifying database access control policies in NL whose implementation is in SQL, but our solution is compatible with any language for implementing database access control. This is because of how database access control is implemented in RDBMSs. Whenever a user issues SQL queries, the DBMS uses specialized algorithms to check the query against GRANT statements [8, 9]. Therefore, access control is carried out according to the *SQL standard* rather than an implementation involving system internals.

3.2 The Benefits of NLACMs

Using NLACMs for access control policies instead of plain NL has three key benefits: (i) NL access control policy documents often contain information *irrelevant* to implementing the policy in a database. The structure of NLACMs guarantee that all NL will be relevant to the database. (ii) NL is often *ambiguous* with respect to the access control rules. The structure of NLACMs alleviate this ambiguity. (iii) NLACMs maintain much of the expressivity of NL

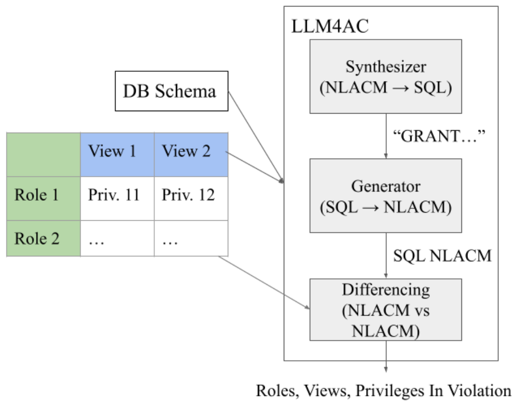


Figure 3: The LLM4AC Architecture

for access control despite their structure. In this section we discuss these benefits of NLACMs in depth.

NL policies often contain irrelevant information to implementing access control policies. For example, an access control policy template [2] has a sentence, *This policy is the property of CompanyName and is intended for internal use only. It should not be reproduced, partially or wholly, in whatever form.*. Another example is, *A database administrator is responsible for the usage, accuracy, efficiency, security, maintenance, administration and development of an organization's computerized database(s), providing support to some or all departments depending on the size of the organization.*. These sentences are clearly not intended to be implemented as GRANT statements on a database. On the other hand, the NLACM's structure lends itself precisely to database access control rules: the (i, j)th cell represents the allowed SQL operators for role/user i on table/view j.

Even when NL describes database access control rules, it can be very ambiguous. For example, one database access control policy says, *SELECT privilege grants a user's access on views and tables should be limited to authorized personnel.*. It is unclear whether this sentence is intended as a general guideline, or a sentence that should result in GRANT statements on the database. Specifically, it is unclear whether to assume that authorized personnel have SELECT privileges on all tables and views in the database. It is also unclear which roles and users are being referenced by "authorized personnel". If this sentence appeared in a NLACM, it would make it clear that this sentence is not a general guideline, and there should be a one-to-one or one-to-many relationship between "authorized personnel" and the database roles/users.

Although NLACMs are a constrained version of NL, they can still be used to express many access control constraints using NL. For example, NLACMs can represent role hierarchies (e.g., a NLACM entry says, "Same as role x"), subsumed views (e.g., a NLACM column name says, "the first 100 rows of View 1"), and dynamic access control constraints (e.g., "read access only from 9am-5pm on weekdays"). Lastly, NLACMs allow IBAC-DB to subsume RBAC (role-based AC) and DAC (discretionary AC), because IBAC-DB intents can be written in NLACMs using plain SQL. In this case, the NLACM becomes a standard ACM representing RBAC and DAC policies.

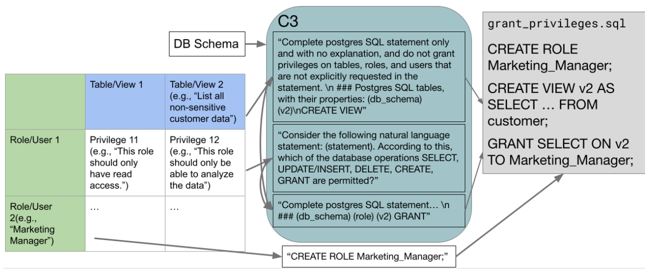

DB Schema
Table/View 1
Table/View 2 (e.g., "List all non-sensitive customer data")
Role/User 1
Privilege 11 (e.g., "This role should only have read access.")
Privilege 12 (e.g., "This role should only be able to analyze the data")
Role/User 2(e.g., "Marketing Manager")
...
...
C3
"Complete postgres SQL statement only and with no explanation, and do not grant privileges on tables, roles, and users that are not explicitly requested in the statement. \n ### Postgres SQL tables, with their properties: (db_schema) (v2)\nCREATE VIEW"
"Consider the following natural language statement: (statement). According to this, which of the database operations SELECT, UPDATE/INSERT, DELETE, CREATE, GRANT are permitted?"
"Complete postgres SQL statement… \n ### (db_schema) (role) (v2) GRANT"
"CREATE ROLE Marketing_Manager;"
grant_privileges.sql
CREATE ROLE Marketing_Manager;
CREATE VIEW v2 AS SELECT … FROM customer;
GRANT SELECT ON v2 TO Marketing_Manager;

Figure 4: LLM4AC Synthesizer Procedure for Policy Implementation

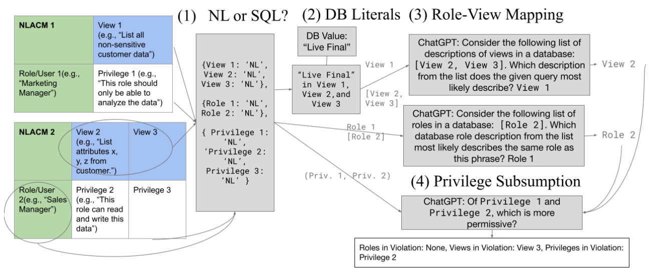

(1) NL or SQL?
(2) DB Literals
(3) Role-View Mapping
NLACM 1
View 1 (e.g., "List all non-sensitive customer data")
Role/User 1(e.g., "Marketing Manager")
Privilege 1 (e.g., "This role should only be able to analyze the data")
NLACM 2
View 2 (e.g., "List attributes x, y, z from customer.")
View 3
Role/User 2(e.g., "Sales Manager")
Privilege 2 (e.g., "This role can read and write this data")
Privilege 3
{View 1: 'NL', View 2: 'NL', View 3: 'NL'}, {Role 1: 'NL', Role 2: 'NL'}, { Privilege 1: 'NL', 'Privilege 2: 'NL', Privilege 3: 'NL' }
DB Value: "Live Final"
"Live Final" in View 1, View 2, and View 3
View 1
[View 2, View 3]
ChatGPT: Consider the following list of descriptions of views in a database: [View 2, View 3]. Which description from the list does the given query most likely describe? View 1
View 2
Role 1
[Role 2]
ChatGPT: Consider the following list of roles in a database: [Role 2]. Which database role description from the list most likely describes the same role as this phrase? Role 1
Role 2
(Priv. 1, Priv. 2)
(4) Privilege Subsumption
ChatGPT: Of Privilege 1 and Privilege 2, which is more permissive?
Roles in Violation: None, Views in Violation: View 3, Privileges in Violation: Privilege 2

Figure 5: LLM4AC Differencer Procedure for Policy Auditing.

3.3 A Reference Architecture for IBAC-DB

We propose a reference architecture and system, *LLM4AC*, for an IBAC-DB interface (see Figure 3 for the full architecture). LLM4AC synthesizes access control policies and compares for the compliance of the implementation as follows: it takes a NLACM expressing an access control policy and a DB schema as input, *synthesizes* database access control rules (GRANT statements), and then assesses the compliance of the implementation by *generating* a NLACM from the DB access control rules and *differencing* the resulting NLACM and the NLACM expressing the policy. We explain each of these steps in detail below and explain the design space for a system built using this reference architecture.

Synthesizer. The synthesizer procedure is shown in Figure 4. LLM4AC takes a NLACM representing a policy and a DB schema as input. Then, we use a LLM-backed NL2SQL system, such as C3[15], which we modify for access control by implementing the novel prompting strategies for generating tables/views, privileges, and GRANT statements from NL in Figure 4. Specifically, we add rules for generating access control SQL and accompanying few-shot examples. We also add chain-of-thought prompting to generate privileges from NL descriptions. The output is a SQL file consisting of GRANT statements, and CREATE ROLE/USER statements.

Generator. The generator procedure simply extracts the list of role/user-privilege-table/view triples from the database (either in the form of a list of GRANT statements issued, or a metadata table containing privileges), and constructs an ACM from these triples.

Differencer. The differencing procedure is shown in Figure 5. The differencer takes two NLACMs as input and determines if the second NLACM is *in violation* compared to the first. That is, either there are roles or views that are assigned privileges in the second NLACM that are not in the first NLACM, or the same role or view in the second NLACM has more permissive privileges than in the first. The differencer performs the following high-level steps: (a) label each role, view, or privilege in the NLACM as NL or SQL, (b) determine which role-view pairs are shared by both NLACMs by first identifying role-view pairs that share literals from the database, and then giving only these candidate pairs to LLM prompts specialized for comparing NL vs NL and NL vs SQL, (c) for the role-view pairs deemed identical by the LLM, determine whether the privileges are subsumed. The output of differencing will be the role-view pairs and privileges that are in violation.

The Design Space. (1) **DBMS SQL Syntax Rules** Different DBMSs have varying support for access control. For example, SQLite's SQL variant has no constructs for access control at all, while PostgresQL supports table and row-level access control, but must implement column-level access control indirectly. Databricks SQL has SQL keywords specifically for column-level access control. Each access control SQL variant requires different rules and few-shot examples for the synthesizer to be successful. (2) **LLM Choice** Different LLMs are trained to generate different variants of SQL with varying levels of accuracy. Therefore, the LLM must be chosen to align with the desired SQL variant. (3) **NL2SQL System Choice** Similarly to (2), there are many choices for NL2SQL systems. We recommend (and use) LLM-backed systems, as they are more easily adaptable for generating many variants of SQL, using rules/few-shot prompting, instead of more expensive fine-tuning.

3.4 LLM4AC: Our System

In this paper, we build a system based on our reference architecture, *LLM4AC*. LLM4AC uses ChatGPT as the LLM, and PostgresQL as the database. slightly alter C3's existing prompt for translating NL to SQL to also translate view creation, we use chain-of-thought prompting [31] to generate privileges from NL descriptions.

4 LLM4AC, IN DEPTH

4.1 The Synthesizer, In-Depth

Definition 2 (The Synthesizer Problem). *Given a NLACM M and a database D, consisting of table/view definitions, roles, and users, generate the set of GRANT statements over D implementing the access control rule specified in each cell of M.*

Prior Work. As mentioned in Section 2, the closest prior work for synthesizing SQL GRANT statements from NL is NL2SQL systems. There are many state-of-the-art NL2SQL systems. Some of these systems train separate models using grammars to avoid syntax errors [18]. Many of them leverage LLMs (e.g., by fine-tuning LLMs and/or using prompt engineering, including few-shot prompting and describing rules for translation). However, none of these systems are built to generate GRANT or DDL statements.

To bridge this gap, we observe that because many LLM-backed state-of-the-art NL2SQL systems use fine-tuning and prompt engineering, adapting them for access control may simply require adding prompts or examples specific to access control applications.

We enhance one such system which has been observed to be state-of-the-art [32], C3 [15]. C3 is a prompt engineering-based NL2SQL system that uses few-shot examples that include rules to avoid specific syntax errors, and completion prompting to boost performance. We enhance C3 with new access control-specific rules, few-shot examples, and prompts that allow it to accurately translate NL for GRANT statements. We call our variant of C3 *C3-AC*.

We now describe the synthesizer procedure in detail. Given a NLACM and a DB with no roles or privileges,

(1) Extract the roles/user strings from the NLACM and generate a "CREATE ROLE/USER..." statement for each role/user.
(2) Extract the table/view SQL from the NLACM. generate a "CREATE VIEW..." statement for each table/view in the NLACM. We achieve this using the following prompt: *Complete postgres SQL statement only and with no explanation, and do not grant privileges on tables, roles, and users that are not explicitly requested in the statement. CREATE VIEW*. This completion-style prompt is similar to that used by C3, but we have adapted it for access control.
(3) Map the privileges to SQL operators. Once we have mapped the NLACM roles, tables, and views to the DB, we then map the natural language privileges of the resulting Type 1 NLACM (the roles and views are found directly in the database, but the privileges can be expressed in natural language). We use the following prompt: *Consider the following statement: <NL for privileges>. According to this, which of the database operations SELECT, UPDATE/INSERT, DELETE, CREATE, GRANT are permitted for role <r> on table/view <v>?*
(4) Synthesize a SQL script from the CREATE ROLE/USER statements, CREATE VIEW statements, and GRANT statements synthesized at each step.

LLMs are known to have syntax errors when generating SQL [18]. We observe that C3 is usually imprecise in one specific way: when DB literals are used in the NL to be implemented, C3 may generate these literals incorrectly when synthesizing SQL for views (e.g., "Live_Final" becomes "Live Final", "More then 80 centimeters" becomes ">80cm", etc.). When such queries fail to execute, we repair them using the following procedure:

(1) Parse the query for its DB literals (and exclude SQL keywords), column names, and table names.
(2) Use the table and column names to query the specific columns used by the query for values contained in the NL. Call the resulting list of DB literals R.
(3) Embed the literals R using BERT.
(4) For each parsed literal from the SQL, use nearest-neighbor search on R to find the highest-matching literal.
(5) Replace the literal in the SQL query with the top match from R.

4.2 The Differencer, In-Depth

A key problem when implementing access control policies is auditing the implementation for compliance. We cannot verify compliance without a deterministic method for comparing NL and SQL, but we can use LLMs to audit the implementation for compliance. We will show empirically that our policy auditing procedure is mostly accurate. Concretely, we audit policy implementations for compliance by comparing the NLACM containing the NL that expresses policy, and the ACM constructed from privileges implemented on DB. We compare two different policies by comparing two NLACMs that are meant to express policies.

Considering this goal, we define differencing according to the following intuition: given NLACM 1 and NLACM 2, where NLACM 1 represents an access control policy, and NLACM 2 represents its implementation on the database, we want to know whether NLACM 2 matches NLACM 1, and if not, then does NLACM 2 at least grant fewer privileges than NLACM 1. Considering this, we define the difference between NLACM 1 and NLACM 2 as: (i) the roles present in NLACM 2 that are not in NLACM 1, (ii) the views present in NLACM 2 that are not in NLACM 1, (iii) the privileges that are more permissive on the same role-view pair in NLACM 2 than on NLACM 1. By this definition, if there is a difference between NLACM 1 and NLACM 2, we can also conclude that NLACM 2 is in violation with respect to NLACM 1.

Formally, we define the difference between two NLACMs as follows:

Strict Inequality (LLM4AC's default).: Which database roles are in M_2, but not in M_1? Which database tables/views are in M_2, but not M_1? Because we assume the principle of failsafe privileges, any role/view in M_2 but not M_1 implies that M_2 is granting privileges on roles/views that M_1 is not, meaning M_2 is in violation compared to M_1. Lastly, for each role r and view v shared between M_1 and M_2, are the privileges assigned to r on v in M_2 subsumed by those of M_1? If not, then M_2 is in violation.

Full Procedure. We now write out the full procedure, both steps and prompts, below.

Concretely, given NLACMs M_1 and M_2, let R_1 be the roles of M_1 and R_2 be the roles of M_2. Similarly, let V_1 be the views of M_1 and V_2 be the views of M_2. Let R_{1N} be the set of roles expressed in natural language, and R_{1D} be the set of roles expressed in SQL. Similarly, let V_{1N} be the set of views expressed in natural language, and V_{1D} be the set of views expressed in SQL. Assume similar naming conventions for M_2. Then, we compare M_1 and M_2 by choosing a role/view from all roles/views in M_2 each role/view of M_1 (i.e., $\forall r \in R_1$, we match r to a role in R_2).

We use different prompts for the cases: (i) r is NL, and we compare to natural language roles, R_{2N}. (ii) r is NL, and we compare to SQL roles R_{2D}, (iii) r is SQL, and we compare to NL roles R_{2N}. We similarly use different prompts for these cases when dealing with comparing views.

Then, the full procedure is:

(1) Classify the roles/views of each of M_1 and M_2 as NL or SQL by searching for SQL keywords in the text.
(2) Determine which roles and views are the same, and which are different using a prompt for NL roles/views to NL roles/views, or the NL to SQL prompt chosen to lift DB to NL. Store all explanations. Specifically, we use the following prompts:
    (a) Prompt for NL vs SQL views ((i) and (iii)): *Which database table or view from the list <R_{2D}> does this phrase <r> most likely describe? Begin your answer with this table/view.*
    (b) Prompt for NL vs NL views ((ii)): *Which database table or view description from the list most likely describes the same table or view as this phrase? Begin your answer with your chosen description from the list.*
    (c) Prompt for NL vs SQL roles ((i) and (iii)): *Which database role from the list does this phrase most likely describe?*

 (d) Prompt for NL vs NL roles ((ii)): *Which database role description from the list most likely describes the same role as this phrase?*
 (e) SQL vs SQL: simply compare role lists. In the case of views, implement views on DB and determine equality pairwise.

(3) For the roles or views that are the same, determine whether the privileges of M_2 are in violation of those of M_1 and label them accordingly. Store the explanation. (we prompt ChatGPT as to whether one set of privileges exceeds the other).

LLM4AC At Scale. Using LLM4AC to difference two NLACMs with dimensions $m_1 \times n_1$ and $m_2 \times n_2$ requires $O(m_{min} \times n_{min})$ calls to the LLM for privilege subsumption, in the worst case. This may seem prohibitively expensive for large NLACMs, both in terms of runtime and monetary cost of using commercial LLMs, like ChatGPT.

Firstly, we argue that large NLACMs will not be part of a typical workload for LLM4AC. NLACMs are written by users who want to ensure access control rules are implemented correctly on a small, specific set of roles/users and tables/views. They will likely use differencing to test their policy implementations on a test database containing no roles or users, only base tables.

That said, suppose a user now wants to difference with respect to their production database, which already has many users and view definitions. Then, the NLACM generated from the database may be large, but this does not cause a scalability issue in terms of LLM calls because we ask the LLM to choose from a list of values during role-view mapping instead of checking each pair. If the list of values exceeds the LLM's context window, we find that chunking the list is an accurate strategy. This can also greatly alleviate the scale problem. For example, ChatGPT has a context window length of 8192 tokens, and a list of 10 view definitions, each roughly 50 tokens long, on average, resulting in only 10 calls of 500 tokens each, with 500 more tokens of output. This costs 5500 tokens. This is cheaper compared to 100 pairwise comparisons, each consisting of at least 100 tokens and 1 token of output (a yes/no token) (10001 tokens).

Now, suppose a user wants to create a large NLACM. For example, they want a NLACM that expresses a policy for their whole production database. LLM4AC will not force users to write it themselves. Instead, because NLACMs can be a mix of NL and SQL, users can specify only the key access control rules in NL, and then LLM4AC will perform the role-view mapping step of differencing to determine which roles and views written in the user's NLACM already have privileges in the database. LLM4AC will then drop such overlapping roles and views and union this NLACM with the NLACM generated from the database.

5 EVALUATION

In this section, we propose a benchmark for evaluating IBAC-DB systems (Section 5.1). We use this benchmark to show that our system, LLM4AC, can accurately synthesize and difference access control policies (Sections 5.2 and 5.3), especially compared to other baselines. Finally, we use this benchmark to determine if there are types of access control policies on which LLM4AC is not as accurate (Section 5.4).

For all experiments, the DBMS is PostgreSQL. We use ChatGPT (`gpt-3.5-turbo`) as the LLM, and ChatGPT's APIs for all prompting.

5.1 IBAC-DB Benchmark Design

We design a benchmark to test IBAC-DB systems that implement access control policies in PostgresQL. The goal of the benchmark is to test how accurately an IBAC-DB system synthesizes a SQL implementation of a NLACM policy, and how accurately an IBAC-DB system can compare two NLACMs. We describe the benchmark design for each of IBAC-DB synthesizers and differencers.

5.1.1 The Differencer Benchmark.

Differencer Evaluation Scenarios. It is essential that IBAC-DB system differencers be able to accurately compare a policy to its implementation for compliance, and compare access control policies to ensure they actually match. To this end, we measure differencing with respect to two scenarios: (i) *policy auditing*: compare a NLACM containing NL to its implementation on the DB, which is a NLACM containing SQL. (ii) *policy comparison*: compare two NLACMs containing NL.

Differencer Evaluation Metric. We evaluate differencing with respect to: (i) *role-view mapping accuracy*: how accurately LLM4AC can find the shared roles and views between the two NLACMs, (ii) *privilege subsumption accuracy*: how accurately LLM4AC can determine whether the privileges contained in the second NLACM subsumed those of the first.

We use F1 scores to measure role-view mapping and privilege subsumption accuracies. Specifically, for role-view mapping, a true positive is a role-view pair shared by both NLACMs that was detected by LLM4AC's differencing procedure. A true negative is a role-view pair not shared by both NLACMs that was not detected by LLM4AC's differencing procedure. A false positive is a role-view pair not shared by both NLACMs that LLM4AC's differencing procedure incorrectly determines is shared. A false negative is a shared role-view pair that LLM4AC determines is not shared. Similarly, for privilege subsumption, a true positive is a privilege of the second NLACM that is subsumed by that of the first, and this was detected by LLM4AC's differencing procedure. A true negative is a privilege not subsumed by that of the first that was not detected by LLM4AC's differencing procedure. A false positive is a privilege not subsumed that LLM4AC's differencing procedure incorrectly determines is subsumed. A false negative is a subsumed pair of privileges that LLM4AC determines is not.

Differencer Benchmark Details. To evaluate LLM4AC with respect to policy auditing and comparison, we construct three NLACMs: a *base* NLACM containing NL, a *SQL* NLACM containing the ground truth SQL in the base, and a *perturbed* NLACM, which is the base NLACM with a NL perturbation applied, such that the resulting NL is syntactically different, but semantically equivalent (e.g., all column names are replaced with synonyms). Then, policy auditing is differencing the base NLACM to the SQL NLACM. Policy comparison is the accuracy of differencing the base and perturbed NLACMs.

An IBAC-DB system needs to have high *cross-domain* performance–that is, the differencer should accurately compare policies on databases with a variety of schemas and policy specifications unseen by the model. To test this using our benchmark, we construct these NLACMs from: (i) databases spanning multiple training domains, and (ii) syntactically different but semantically equivalent NL (e.g., column names expressed using carrier phrases instead of their actual names). We gather this data from an existing benchmark, Dr. Spider [12], a NL2SQL benchmark with databases spanning several training domains, and several NL perturbations.

There are only 5 databases in Dr. Spider that have NL-query-perturbation triples for all perturbation types. We create the equivalent base-SQL-perturbed NLACMs for each of these databases. We list the databases and some basic characteristics here:

(1) **orchestra**: 4 tables, 27 columns
(2) **dog_kennels**: 8 tables 57 columns
(3) **employee_hire_evaluation**: 4 tables, 21 columns
(4) **student_transcript_tracking**: 11 tables, 78 columns
(5) **car_1**: 6 tables, 29 columns

NLACM Construction. To construct the necessary NLACM pairs for policy comparison and auditing, we extract the NL describing database views from Dr. Spider. We automatically generate the roles and privileges in the NLACM. To experiment with NL perturbations of roles and views as well, we define and use the following perturbations:

(1) **Role synonyms**: replace all role names with synonyms (e.g., "Nonprofit Organization intern" → "Charitable Organization Administration Intern"
(2) **Role descriptions**: replace all role names with descriptions of the role. (e.g., "A person who works in a charitable organization to gain experience in overseeing operations and programs.")
(3) **Privilege synonyms**: replace permitted SQL operators with synonyms. (e.g., "SELECT, UPDATE, INSERT, GRANT" → "This position holds the authority to CHOOSE, ESTABLISH, BESTOW permissions on this database perspective.")
(4) **Privilege carrier phrases**: replace permitted SQL operators with carrier phrases that imply them. (e.g., "SELECT, UPDATE, INSERT, GRANT" → "This role is authorized to perform actions on this database view.")

The base, SQL, and perturbed NLACMs should be equivalent. That is, the (i,j)th cell of all three NLACMs should describe the same privileges on the same role on the same view. We construct NLACMs that each contain 10 views and 10 roles. We chose this size of NLACM because human users will create NLACMs manually, so it is unlikely that NLACMs will be large.

5.1.2 *The Synthesizer Benchmark.*

Synthesizer Evaluation Scenarios. It is essential that IBAC-DB system synthesizers be able to accurately implement a policy, regardless of the complexity or variety of queries that must be generated. The main difficulty of translating NLACMs to SQL is *translating views*, as view definitions can be very complex, involving multi-way joins, complex filters, etc. To this end, we measure synthesizing with respect to the types of SQL queries that must be generated when given NL descriptions of database views. We use the following taxonomy:

(1) **Simple projections (single SELECT over multiple columns)**
(2) **Complex projections (SELECT over multiple columns, aggregates, over multiple conditions of multiple tables, etc.)**
(3) **Simple whole-table aggregations (single GROUP BY over 1-2 attributes in a single table)**
(4) **Simple joins (join between two or three tables on clearly specified FKs)**
(5) **Multi-way joins (join between more than three tables)**
(6) **Common table expressions (uses the HAVING keyword)**
(7) **Simple conditions (1-3 conditions each using single predicates on an attribute, or an aggregate of an attribute)**
(8) **Complex conditions (conditions can be lengthy, there may be self-defined conditions, self-defined variables in the condition, etc.)**

Synthesizer Evaluation Metric. We input this NLACM to LLM4AC's Synthesizer and test the *execution accuracy* of the result. Specifically, we execute the resulting GRANT statements on one copy of the databases, and execute the ground truth on a separate copy of the databases. PostgreSQL stores the results as a list of privileges in a table. We compare these tables to see if they are equal.

Synthesizer Benchmark Details. To gather these queries of varying types and complexity, and their NL to be translated into SQL, we use a mix of the TPC-H queries and queries from several Spider databases.

- **TPC-H**: 8 tables. We select queries from the set given in the benchmark, and we extract the NL from the business questions describing each query, available in the TPC-H specification.
- **Spider** [32]: contains 166 separate databases, each containing between 3 and 14 tables. We choose queries from three databases from Spider: `department_management`, `culture_company`, and `bike_1`. The NL is the ground truth input to a NL2SQL system whose ground truth output is the query.

We gather up to 5 queries for each category above from the TPC-H queries and Spider queries, resulting in 33 queries altogether. We modify each query to create a view from its result. Then, we randomly generate 10 roles, and privileges on each view to create one NLACM with dimensions 10 rows and 33 columns.

5.2 LLM4AC Differencing: Macrobenchmark

In this section, we use our IBAC-DB benchmark to evaluate the accuracy of LLM4AC's differencer.

Evaluation Baselines. We compare LLM4AC's prompting differencing procedure to the following baselines for role-view mapping:

(1) **Plain LLM**: LLM4AC prunes out incorrect answers by only including candidates that share DB literals. But is this DB literals comparison actually necessary? To explore this, we include a plain use of the LLM where answers are not pruned out. Everything else is kept the same.
(2) **Sentence Similarity**: Embed the input role/view and the list of roles/views using SentenceBERT [25]. Then, return the sentence with the least cosine similarity.
(3) **Token Similarity**: Embed the tokens of the input role/view and the tokens of each role/view in the list using BERT. For each role/view in the list, average the token embeddings and choose the role/view whose cosine similarity has the least distance to the average embedding of the input role/view.

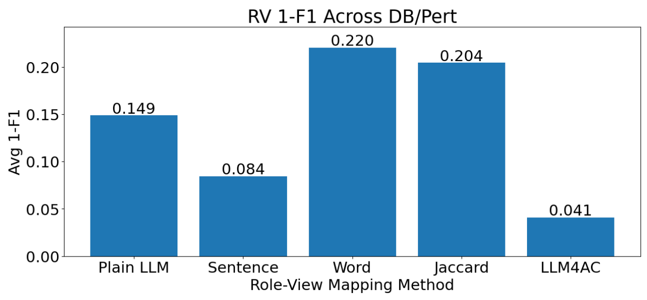


Figure 6: NL vs NL Role-View Mapping F1 Errors across all DBs and Perturbations. Lower is better.

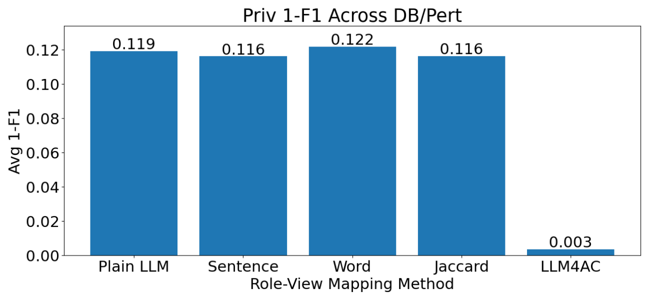


Figure 7: NL vs NL Privilege Subsumption F1 Errors across all DBs and Perturbations. Lower is better.

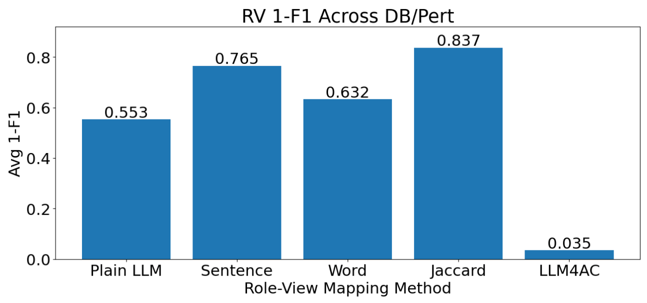


Figure 8: NL vs SQL Role-View Mapping F1 Errors across all DBs and Perturbations. Lower is better.

(4) **Jaccard Similarity**: Choose the role/view from the list whose tokens have the highest Jaccard similarity with the tokens of the input.

For all baselines except Jaccard, we adjust them for comparing NL and SQL by generating a NL description of the SQL query, and comparing the NL description to a list of NL roles/views of the other NLACM.

We do not include baselines for privilege subsumption as it is an ill-defined learning problem and, as we will see, prompting performs sufficiently well on it.

Policy Comparison.: Figure 7 and Figure 6 shows that, across databases and perturbations, role-view mapping appears to be more difficult than privilege subsumption. Further, LLM4AC or sentence embeddings outperform other role-view mapping methods.

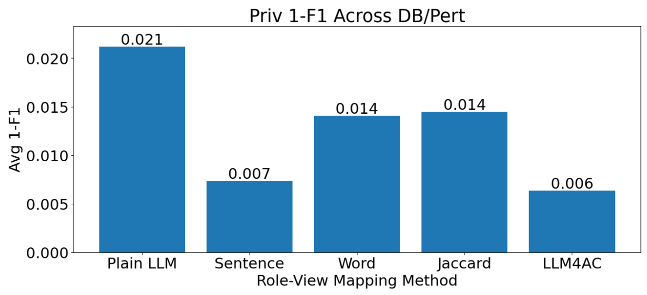


Figure 9: NL vs SQL Privilege Subsumption F1 Errors across all DBs and Perturbations. Lower is better.

Implementation Comparison.: Figure 9 and Figure 8 show that overall, LLM4AC outperforms the baselines. For other methods, the F1 errors are all higher compared to those of policy comparison. The F1 error for Plain LLM is higher than for other methods, which initially suggests that using LLMs is not preferable. However, this is because all the errors are still due to mistakes that could have been fixed by comparing DB literals. In this case, this would simply be a matter of using SQL parsing and then string matching. LLM4AC incorporates this observation, leading to its high performance. The other methods have high errors, likely due to the noise introduced when translating the SQL to NL. For example, the view ChatGPT explains the view definition `CREATE VIEW query2view0 SELECT Model FROM CAR_NAMES GROUP BY Model ORDER BY count(*) DESC LIMIT 1` using NL: *This SQL query creates a view called "query2view0" that selects the "Model" column from a table called "CAR_NAMES". It then groups the results by "Model", counts the number of occurrences of each model, and orders the results in descending order based on the count. Finally, it limits the output to only show the model with the highest count. Essentially, this query finds the most common car model in the "CAR_NAMES" table.*. The baselines must somehow match this long description to the NL question: *How many car models are produced by each maker ? Only list the count and the maker full name .*

Overall, we observe that our baselines perform quite well at policy comparison. Even Jaccard similarity has a F1 of 0.7 on the Dr. Spider perturbations. This is because many of these perturbations do not change many tokens in the original text, only replacing certain phrases (e.g., column names or DB values) with synonyms, phrases, etc. This also helps explain why word embeddings can be even worse than Jaccard similarity. This is because by naively averaging embeddings over words, it is no longer possible to match on DB literals, which would enable more accurate matching.

5.3 LLM4AC Synthesizer: Macrobenchmark

In this section, we test the performance of LLM4AC's synthesizer.

We see that LLM4AC's synthesizer is mostly accurate, failing at multi-way joins and nested queries. We observe these errors for the following reasons: (1) C3 can occasionally choose the wrong tables for a multi-way join (2) C3 can "forget" that a nested query is needed.

Query Type	Accuracy
Single column Projection	50 / 50
Multiple column Projection	50 / 50
Single Whole-table Aggregation	30 / 30
Single join	50 / 50
Multi-way join	10 / 20
Common Table Expression	40 / 40
Nested Queries	0 / 10
Single WHERE clause condition	50 / 50
Multiple WHERE clause conditions	30 / 30
Total Accuracy	310 / 330

Table 1: LLM4AC Synthesizer Execution Accuracy. The only errors were nested queries and multi-way joins.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
multitype	0.837	0.895	0.538	0.563	0.940
column_attribute	0.809	0.863	0.745	0.776	0.953
others	0.922	0.933	0.846	0.808	0.937
value_synonym	0.866	0.896	0.818	0.781	0.936
keyword_carrier	0.862	0.915	0.508	0.678	0.966
column_value	0.727	0.857	0.612	0.694	0.985
column_synonym	0.872	0.928	0.680	0.784	0.937
keyword_synonym	0.848	0.952	0.765	0.733	0.881
column_carrier	0.800	0.909	0.625	0.531	0.990
role_syn	0.608	0.842	0.842	0.000	0.980
roledesc_replace	0.675	0.842	0.842	0.000	0.984
priv_syn	0.952	0.952	0.952	0.952	0.998
priv_inf	0.952	0.952	0.952	0.952	0.986

Figure 10: NL vs NL Role-View Mapping F1s by Perturbation. Darker is better, and shading is row-wise.

5.4 LLM4AC Differencer: Microbenchmark

We study the variations in the LLM4AC Differencer performance due to specific NL perturbations or databases. We find that, even after stratifying the results across databases and NL perturbations, LLM4AC still outperforms other baselines (Section 5.4.1). We then study LLM4AC's performance in-depth (Section 5.4.2).

5.4.1 LLM4AC vs Baselines, by DB/Perturbation. **Policy Comparison.** When we compute the F1s for each perturbation, we find that LLM4AC vastly outperforms all other methods, followed by sentence embeddings (Figure 10 and Figure 11). We find that this is because naively providing all inputs to prompting generates incorrect answers that are fixed simply by comparing sentences with shared DB literals instead. On the other hand, sentence embeddings have more obscure errors: namely, when the correct answer is a more abstract sentence (e.g., "How many orchestras has a conductor done?") sentence embedding similarity is incorrect. We find the same pattern over databases (Figure 12 and Figure 13).

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
multitype	0.967	0.994	0.984	0.984	0.994
column_attribute	0.947	0.995	0.987	0.989	1.000
others	0.985	0.994	0.992	0.994	1.000
value_synonym	0.957	0.978	0.979	0.978	1.000
keyword_carrier	0.965	0.996	0.983	0.980	1.000
column_value	0.981	0.998	0.993	0.982	1.000
column_synonym	0.963	0.981	0.989	0.987	1.000
keyword_synonym	0.966	0.989	0.991	0.989	0.981
column_carrier	0.946	0.973	0.976	0.990	1.000
role_syn	0.950	0.997	0.957	0.000	1.000
roledesc_replace	0.957	0.997	0.958	0.000	1.000
priv_syn	0.540	0.374	0.382	0.379	1.000
priv_inf	0.428	0.224	0.227	0.236	0.980

Figure 11: NL vs NL Privilege Subsumption F1s by Perturbation. Darker is better, and shading is row-wise.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
orchestra	0.821	0.920	0.806	0.635	0.971
dog_kennels	0.804	0.826	0.667	0.705	0.954
car_1	0.848	0.914	0.747	0.672	0.954
employee_hire_evaluation	0.842	0.930	0.798	0.663	0.956
udent_transcripts_tracking	0.865	0.943	0.795	0.702	0.955

Figure 12: NL vs NL Role-View Mapping F1s by DB. Darker is better, and shading is row-wise.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
orchestra	0.943	0.891	0.847	0.824	0.995
dog_kennels	0.849	0.884	0.859	0.862	0.995
car_1	0.895	0.925	0.908	0.885	0.995
employee_hire_evaluation	0.838	0.899	0.861	0.841	0.995
udent_transcripts_tracking	0.893	0.910	0.892	0.863	0.995

Figure 13: NL vs NL Privilege Subsumption F1s by DB. Darker is better, and shading is row-wise.

Implementation Comparison. Figure 14 and Figure 15 show the results for differencing stratified by perturbation. Based on the F1 scores listed, we do not find a difference between the performance of LLM4AC relative to other role-view mapping methods with respect to perturbation. Similarly, Figure 16 and Figure 17 show the results stratified by database. We do not find a difference between the performance of LLM4AC relative to other role-view mapping methods with respect to database, either.

5.4.2 LLM4AC's Performance, In-Depth. We analyze LLM4AC's performance variations on various databases and perturbations to better characterize its performance. We do not show LLM4AC's privilege subsumption performance, as it performs perfectly with no variation across databases and perturbations.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
multitype	0.373	0.237	0.082	0.088	0.993
column_attribute	0.432	0.189	0.092	0.140	0.953
others	0.486	0.254	0.037	0.094	0.998
value_synonym	0.530	0.287	0.093	0.045	0.981
keyword_carrier	0.372	0.146	0.042	0.211	0.955
column_value	0.298	0.261	0.129	0.224	0.789
column_synonym	0.429	0.241	0.068	0.128	0.991
keyword_synonym	0.376	0.179	0.044	0.127	0.992
column_carrier	0.517	0.185	0.056	0.140	0.947
role_syn	0.459	0.194	0.048	0.057	0.993
roledesc_replace	0.450	0.227	0.018	0.094	0.968
priv_syn	0.463	0.214	0.021	0.057	0.987
priv_inf	0.492	0.236	0.042	0.084	0.994

Figure 14: NL vs SQL Role-View Mapping F1s, by Perturbation. Darker is better, and shading is row-wise.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
multitype	0.958	0.993	0.980	0.963	1.000
column_attribute	0.992	0.980	0.957	1.000	1.000
others	1.000	1.000	1.000	1.000	1.000
value_synonym	0.942	0.978	0.867	1.000	1.000
keyword_carrier	1.000	1.000	1.000	0.974	1.000
column_value	0.941	1.000	1.000	1.000	1.000
column_synonym	0.977	0.986	1.000	1.000	1.000
keyword_synonym	0.996	0.991	0.833	0.965	0.990
column_carrier	0.986	0.991	1.000	0.976	1.000
role_syn	1.000	1.000	0.968	0.974	1.000
roledesc_replace	0.991	0.985	1.000	1.000	1.000
priv_syn	0.997	0.992	1.000	1.000	0.958
priv_inf	0.994	1.000	1.000	1.000	0.969

Figure 15: NL vs SQL Privilege Subsumption F1s by Perturbation. Darker is better, and shading is row-wise.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
orchestra	0.451	0.216	0.063	0.182	0.987
dog_kennels	0.480	0.241	0.048	0.098	0.974
car_1	0.323	0.148	0.051	0.058	0.974
employee_hire_evaluation	0.491	0.185	0.049	0.113	0.976
udent_transcripts_tracking	0.485	0.308	0.053	0.093	0.976

Figure 16: NL vs SQL Role-View Mapping F1s by DB. Darker is better, and shading is row-wise.

Policy Comparison. In Figure 21, we show LLM4AC's NL vs NL F1 scores stratified by perturbation, in order to determine if policy comparison varies by perturbation type. Overall, LLM4AC's performance shows slight variations by perturbation. Similarly, in Figure 20, LLM4AC's performance stratified by database has almost no variation among different databases. That said, the LLM4AC's performance at comparing raw NL to its keyword synonym perturbation is noticeably lower compared to other perturbations. That is, LLM4AC's role-view mapping performance at detecting whether NL is the same as replacing SQL keyword indicators with keyword synonyms is lower compared to other perturbations. This is because ChatGPT hallucinates by assuming the correct answer is not among the answer choices provided (even when there is only one answer choice provided, which is the correct one). This leads to a lower recall, and therefore a lower F1.

	Plain LLM	Sentence	Word	Jaccard	LLM4AC
orchestra	0.983	0.996	0.947	0.991	0.992
dog_kennels	0.983	0.988	0.954	0.993	0.992
car_1	0.993	1.000	0.988	0.968	0.992
employee_hire_evaluation	0.984	0.976	0.949	0.974	0.992
udent_transcripts_tracking	0.991	0.998	1.000	1.000	0.992

Figure 17: NL vs SQL Privilege Subsumption F1s by DB. Darker is better, and shading is row-wise.

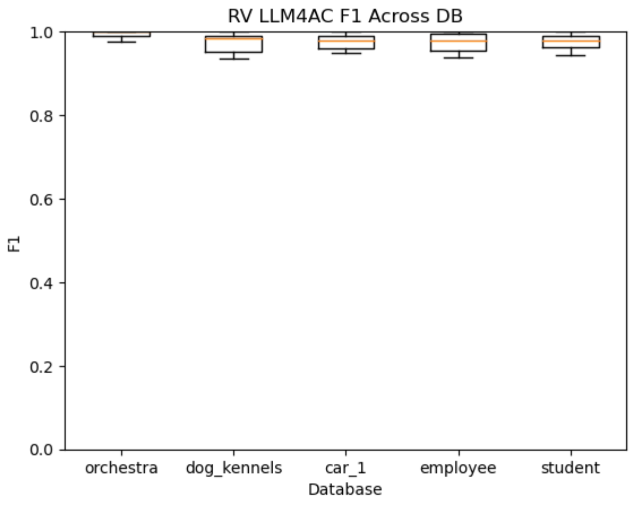


Figure 18: LLM4AC's NL vs SQL Role-View Mapping F1s by DB. y-axis is boxplot of F1s across perturbations.

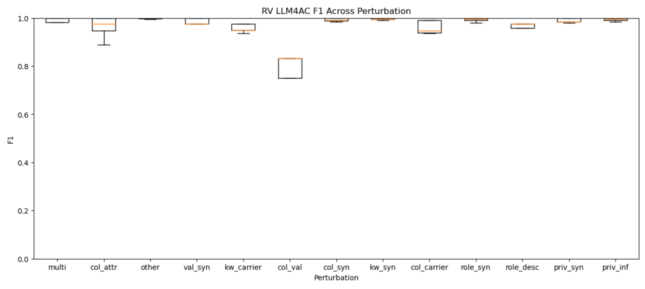


Figure 19: LLM4AC's NL vs SQL Role-View Mapping F1s by Perturbation. y-axis is boxplot of F1s across databases.

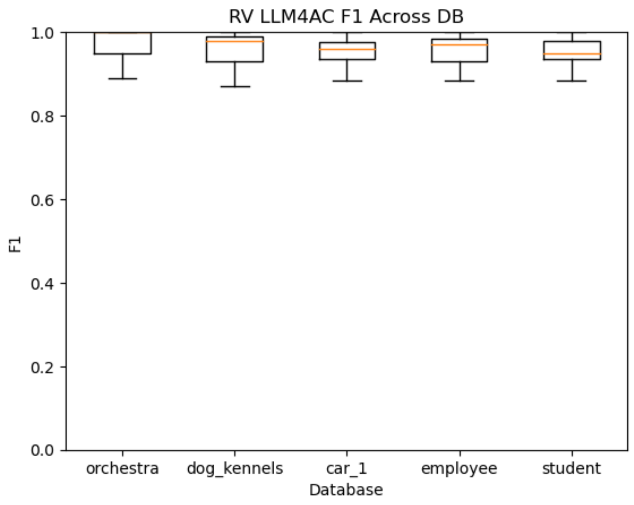


Figure 20: LLM4AC's NL vs NL Role-View Mapping F1s by DB. y-axis is boxplot of F1s across perturbations.

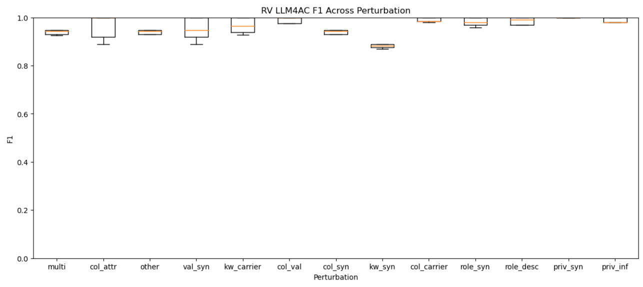


Figure 21: LLM4AC's NL vs NL Role-View Mapping F1s by Perturbation. y-axis is boxplot of F1s across databases.

Implementation Comparison. In Figure 19, we show LLM4AC's NL vs SQL F1 scores stratified by perturbation, in order to determine if implementation comparison varies by perturbation type. Similarly, in Figure 18, LLM4AC's performance stratified by database has almost no variation among different databases. LLM4AC's performance shows little variation across database, and slight variation across perturbation. That said, the LLM4AC's performance at comparing NL whose column indicators have been replaced with column values is noticeably lower compared to other perturbations. This is because ChatGPT hallucinates by assuming the correct answer is not among the answer choices provided when matching SQL to NL whose column indicators have been replaced by values in the column. One way to solve this would be to input the distinct values that can appear in a column as well, as hints to the LLM about which column may have the chosen value. But this would likely require a LLM with a very large context length.

6 CONCLUSION

In this paper, we recognize the problem of automating the policy comparison and auditing of access control policies written in NL. To facilitate this, we propose IBAC-DB, a new access control paradigm. We define a reference architecture for IBAC-DB which can be applied across several different LLMs and DBMSs. Then, we build one possible system, LLM4AC, using this reference architecture. We find that LLM4AC performs well compared to other role-view mapping methods, and that LLM4AC performs reliably: it has low performance variation by database and NL perturbation.

REFERENCES

[1] [n.d.]. Access Control Policy and Implementation Guides. https://csrc.nist.gov/projects/access-control-policy-and-implementation-guides.
[2] [n.d.]. Database Security Policies: Examples and Creation. https://study.com/academy/lesson/database-security-policies-examples-and-creation.html.
[3] [n.d.]. OASIS eXtensible Access Control Markup Language (XACML) TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml. Accessed 12-10-23.
[4] [n.d.]. SOC 2 Compliance. https://www.aicpa-cima.com/topic/audit-assurance/audit-and-assurance-greater-than-soc-2.
[5] 2020. Access Control Policy. https://www.luc.edu/its/aboutus/itspoliciesguidelines/accesscontrolpolicy/.
[6] 2022. IT Access Control and User Access Management Policy. https://www.nwpolytech.ca/about/administration/policies/fetch.php?ID=320.
[7] 2023. What is the Purpose of a Data Access Control Policy? https://satoricyber.com/data-access-control/what-is-the-purpose-of-a-data-access-control-policy/.
[8] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. 1996. A temporal access control mechanism for database systems. *IEEE Transactions on Knowledge and Data Engineering* 8, 1 (1996), 67–80. https://doi.org/10.1109/69.485637
[9] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. 2000. TRBAC: a temporal role-based access control model. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control* (Berlin, Germany) *(RBAC '00)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/344287.344298
[10] Elisa Bertino, Gabriel Ghinita, Ashish Kamra, et al. 2011. Access control for databases: Concepts and systems. *Foundations and Trends® in Databases* 3, 1–2 (2011), 1–148.
[11] Elisa Bertino and Ravi Sandhu. 2005. Database security-concepts, approaches, and challenges. *IEEE Transactions on Dependable and secure computing* 2, 1 (2005), 2–19.
[12] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Dr.Spider: A Diagnostic Evaluation Benchmark towards Text-to-SQL Robustness. arXiv:2301.08881 [cs.CL]
[13] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. 2011. Database access control and privacy: Is there a common ground?. In *CIDR*. Citeseer, 96–103.
[14] Sophie Chou, William Li, and Ramesh Sridharan. 2014. Democratizing data science. In *Proceedings of the KDD 2014 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA*. 24–27.
[15] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. 2023. C3: Zero-shot Text-to-SQL with ChatGPT. arXiv:2307.07306 [cs.CL]
[16] Evren Eryurek, Uri Gilad, Valliappa Lakshmanan, Anita Kibunguchy-Grant, and Jessi Ashdown. 2021. *Data Governance: The Definitive Guide*. O'Reilly Media, Inc.
[17] Bridget A Fahey. 2021. Data federalism. *Harv. L. Rev.* 135 (2021), 1007.
[18] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (feb 2023), 1534–1547. https://doi.org/10.14778/3583140.3583165
[19] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363 [cs.DB]
[20] Yaga Dylan Hu Vincent, Kuhn Richard. [n.d.]. Verification and Test Methods for Access Control Policies/Models. https://csrc.nist.gov/pubs/sp/800/192/final.
[21] Marijn Janssen, Paul Brous, Elsa Estevez, Luis S Barbosa, and Tomasz Janowski. 2020. Data governance: Organizing data for trustworthy Artificial Intelligence. *Government Information Quarterly* 37, 3 (2020), 101493.
[22] Wolfgang Kerber. 2020. From (horizontal and sectoral) data access solutions towards data governance systems. (2020).
[23] Thema Monroe-White, Brandeis Marshall, and Hugo Contreras-Palacios. 2021. Waking up to Marginalization: Public Value Failures in Artificial Intelligence and Data Science. In *Proceedings of 2nd Workshop on Diversity in Artificial Intelligence (AIDBEI) (Proceedings of Machine Learning Research)*, Deepti Lamba and William H. Hsu (Eds.), Vol. 142. PMLR, 7–21. https://proceedings.mlr.press/v142/monroe-white21a.html
[24] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. arXiv:2302.08468 [cs.LG]
[25] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. arXiv:1908.10084 [cs.CL]
[26] RS Sandhu, E Coyne, H Feinstein, and C Youman. [n.d.]. Role-Based Access Control Models, 1996.
[27] Adil Hussain Seh, Mohammad Zarour, Mamdouh Alenezi, Amal Krishna Sarkar, Alka Agrawal, Rajeev Kumar, and Raees Ahmad Khan. 2020. Healthcare data breaches: insights and implications. In *Healthcare*, Vol. 8. MDPI, 133.
[28] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Ozcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: Natural Language Querying for Complex Nested SQL Queries. *Proc. VLDB Endow.* 13, 11 (2020), 2747–2759.
[29] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS*. Citeseer.
[30] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 7567–7578. https://doi.org/10.18653/v1/2020.acl-main.677
[31] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
[32] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425
[33] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.00103 (2017).
[34] Ezra Zigmond, Stephen Chong, Christos Dimoulas, and Scott Moore. 2019. Fine-Grained, Language-Based Access Control for Database-Backed Applications. *The Art, Science, and Engineering of Programming* 4, 2 (Sept. 2019). https://doi.org/10.22152/programming-journal.org/2020/4/3