Demonstrating AI-enabled SQL Queries over Relational Data using a Cognitive Database

José Luis Neves  
IBM Systems  
Poughkeepsie, NY  
jneves@us.ibm.com

Rajesh Bordawekar  
IBM T. J. Watson Research Center  
Yorktown Heights, NY  
bordaw@us.ibm.com

ABSTRACT
This paper demonstrates key capabilities of Cognitive Database, a novel AI-enabled relational database system which uses an unsupervised neural network model to facilitate semantic queries over relational data. The neural network model, called word embedding, operates on an unstructured view of the database and builds a vector model that captures latent semantic context of database entities of different types. The vector model is then seamlessly integrated into the SQL infrastructure and exposed to the users via a new class of SQL-based analytics queries known as cognitive intelligence (CI) queries. The cognitive capabilities enable complex queries over multi-modal data such as semantic matching, inductive reasoning queries such as analogies, and predictive queries using entities not present in a database. We plan to demonstrate the end-to-end execution flow of the cognitive database using a Spark based prototype. Furthermore, we demonstrate the use of CI queries using a publicly available enterprise financial dataset (with text and numeric values). A Jupyter Notebook python based implementation will also be presented.

CCS CONCEPTS  
• Information systems → Structured Query Language; Online analytical processing; SQL; Multi-query engines;

KEYWORDS  
Artificial Intelligence, Relational Databases, SQL, Word Embedding

1 INTRODUCTION
Relational Databases store information based on a user defined schema that describes the data types, keys and functional dependencies. Knowing the schema allows someone to extract relevant information. For example, given a table with a column containing financial transactions described by individual amounts one can easily calculate the total amount. Likewise, if there is a date associated with the transaction, one can report the financial data by month, quarter, or year. Database languages like SQL allow a user to make these and more complex queries. However, the semantic relationships represented by the data is mostly left to the user interpretation as queries are executed and data is re-organized. Further, traditional SQL queries rely mainly on value-based predicates to detect patterns. For example, the aggregate of all transactions within a timeframe or the sorting of transaction amounts by decreasing order of value. The meaningful relationship and interpretation between the data of multiple columns is left to the user during the writing of the SQL query. Thus, the traditional SQL queries lack a holistic view of the underlying relations, and thus are unable to extract and exploit semantic relationships that are collectively generated by tokens in a database relation.

This paper discusses Cognitive Database [3, 5], a novel relational database system, which uses an unsupervised neural network based approach from Natural Language Processing, called word embedding, to extract latent knowledge from a database table. The generated word-embedding model captures inter- and intra-column semantic relationships between database tokens of different types. For each database token, the model includes a vector that encodes contextual semantic relationships. The cognitive database seamlessly integrates the model into the existing SQL query processing infrastructure and uses it to enable a new class of SQL-based analytics queries called Cognitive Intelligence (CI) queries. CI queries use the model vectors to enable complex queries such as semantic matching, inductive reasoning queries such as analogies or semantic clustering, and predictive queries using entities not present in a database. In this paper, we demonstrate unique capabilities of Cognitive Databases using an use-case where SQL-based CI queries, in conjunction with traditional SQL queries, are used to analyze a multi-modal relational database containing text and numeric values. We evaluate this use-case using a Spark-based cognitive database prototype.

The rest of the paper is organized as follows: In Section 2, we first summarize key design aspects of cognitive database and then discuss architecture of the Spark-based prototype. Section 3 describes in detail the different types of Cognitive CI queries and how they work with UDFs and the word embedding model. Section 4 outlines key features being demonstrated: data pre-processing to build the word-embedding model, the word-embedding model, design of the Spark-based CI queries over multi-modal data, different examples of CI queries including analysis explaining the results obtained for each type of CI query, and the implementation and Python-based interfaces for the CI queries.

2 BACKGROUND AND SYSTEM

ARCHITECTURE
Figure 1 presents the three key phases in the execution flow of a cognitive database. The first, only executed when a new model is created or needs to be updated, training phase takes place when the
We have modified the classical word embedding approach [9] to can support both the traditional value-based as well as the new
mented using user-defined functions (UDFs). Thus, the CI queries
textual semantic comparison between relational variables, is imple-
provide, thus allowing the system to be used by both the database
invoke Scala-based cognitive UDFs to enable computations on the
Cognitive Intelligence queries using Spark SQL. The SQL queries
can be created offline from either the database being queried or
external knowledge bases such as Wikipedia), and then invokes the
CI queries provide approximate answers that reflect a given model.

Our current implementation supports four types of CI SQL
queries: similarity based classification, inductive reasoning, pre-
diction, and cognitive OLAP [3]. These queries can be executed
over databases with multiple datatypes: we currently support text,
numeric, and image data. The similarity queries compare two re-
lational variables based on similarity or dissimilarity between the
input variables. Each relational variable can be either set or se-
lational variables based on similarity or dissimilarity between the
similarity

ees that reflect a given model.

Figure 2: Spark Implementation of a cognitive relational
database

Our current implementation supports four types of CI SQL
queries: similarity based classification, inductive reasoning, pre-
diction, and cognitive OLAP [3]. These queries can be executed
over databases with multiple datatypes: we currently support text,
numeric, and image data. The similarity queries compare two re-
lational variables based on similarity or dissimilarity between the
input variables. Each relational variable can be either set or se-

3 COGNITIVE INTELLIGENCE QUERIES
The basic UDF and its extensions are invoked by the SQL CI queries
to enable semantic operations on relational variables. Each CI query
uses the UDFs to execute nearest neighbor computations using the
vectors from the current word-embedding model. Thus, CI queries
provide approximate answers that reflect a given model. For the pur-
poses of demonstrating the various CI queries a publicly available

Figure 1: End-to-end execution flow of a cognitive relational
database

The distinguishing aspect of cognitive intelligence queries, con-
textual semantic comparison between relational variables, is imple-
imated using user-defined functions (UDFs). Thus, the CI queries
can support both the traditional value-based as well as the new
semantic contextual computations in the same query. Each CI
query uses the UDFs to measure semantic similarity between a pair
of sets (or sequences) of tokens associated with the input
relational parameters. The core computational operation of a cog-
nitive UDF is to calculate similarity between a pair of tokens by
computing the cosine distance between the corresponding vec-
tors. For two vectors $v_1$ and $v_2$, the cosine distance is computed
as $\cos(v_1, v_2) = \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|}$. The cosine distance value
varies from 1.0 (very similar) to -1.0 (very dissimilar). Each CI query
uses the UDFs to execute nearest neighbor computations using the
vectors from the current word-embedding model. Thus, CI queries
provide approximate answers that reflect a given model.

Our current implementation supports four types of CI SQL
queries: similarity based classification, inductive reasoning, pre-
diction, and cognitive OLAP [3]. These queries can be executed
over databases with multiple datatypes: we currently support text,
numeric, and image data. The similarity queries compare two re-
lational variables based on similarity or dissimilarity between the
input variables. Each relational variable can be either set or se-

3 COGNITIVE INTELLIGENCE QUERIES
The basic UDF and its extensions are invoked by the SQL CI queries
to enable semantic operations on relational variables. Each CI query
uses the UDFs to execute nearest neighbor computations using the
vectors from the current word-embedding model. Thus, CI queries
provide approximate answers that reflect a given model. For the pur-
poses of demonstrating the various CI queries a publicly available
dataset is used as presented in Section 4. However, lets briefly describe the data as it helps explain the different types of CI queries detailed in this section. The dataset contains all the expenditure transactions for the state of Virginia for the year of 2016. Each transaction is characterized by several fields, namely `VENDOR_NAME`, `AGENCY`, `FUND_DETAIL`, `OBJECTIVE`, `SUB_PROGRAM`, `AMOUNT`, and `VOUCHER_DATE`. Through feature engineering two more fields were added to the dataset, namely `QUARTER` and `COUNTY`, the purpose explained at a later time. The `VENDOR_NAME` field names the institution the state had an expense with. This institution can be a state, county or local agency, a physical person, a local, state or national business, etc. The CI queries can be broadly classified into four categories as follows:

### 3.1 Similarity Queries

The basic UDF that compares two sets of relational variables can be integrated into an existing SQL query to form a similarity CI query. In a traditional SQL environment, to determine similarity of transaction expenses of a given customer against all the other customers, one would have to determine the terms of comparison, meaning which columns to compare followed by gathering statistics for all the transactions of the given customer. This process would have to be repeated for all the other customers in the table. Finally, each customer would be compared against the given customer and a score calculated that represents similarity. Note that the terms that define similarity would have to be defined prior to all this process, meaning the rules amongst the columns that define similarity. Also note, that the aforementioned process becomes more complex and time consuming as the number of features describing each transaction increases. The alternative is a SQL Similarity CI query as illustrated in Figure 3 that identifies similar transactions to all the transactions by a given `VENDOR_NAME`, in this case the County of Arlington. Assume that Expenses is a table that contains all transaction expenses for the state of Virginia whose Expenses.VENDOR_NAME column contains each individual entity or customer the state had an expense with. To identify which transactions have a similar transaction pattern to a given customer one would use a SQL query with a UDF, in this case `proximityCust_NameUDF()`, that computes similarity score between two sets of vectors, that correspond to the fields describing each `VENDOR_NAME`.

```sql
SELECT VENDOR_NAME, proximityCust_NameUDF(VENDOR_NAME, 'Vendor') AS proximityValue
FROM Expenses
WHERE proximityValue > 0.5
ORDER BY proximityValue DESC
```

**Figure 3: Example of a SQL CI similarity query: find similar state customers based on expense transaction patterns**

The query shown in Figure 3 uses the similarity score to select rows with related Vendors and returns an ordered set of similar Vendors sorted in descending order of their similarity score. The similarity score is computed by calculating the cosine distance between vectors, one being the vector of the customer of interest, County of Arlington, and the other the vector associated with Expenses.VENDOR_NAME for all the VENDOR_NAMEs in the table Expenses. The similarity score is sorted in descending order and the outcome presented in a table as illustrated in Figure 4.

### 3.2 Dissimilar Queries

Dissimilarity is a special case of similarity where the query will first choose rows whose Vendors have lower similarity (e.g., $< 0.3$) to a given Vendor and the results ordered in an ascending form using the SQL ASC keyword. This variation returns Vendors that are highly dissimilar to a given Vendor (i.e., the transaction with the state is with completely different agencies, objectives, funds and/or programs). If the results are ordered in the descending order using the SQL DESC keyword, the CI query will return Vendors that are somewhat dissimilar to a given Vendor.

If one is interested to find the counties that are most dissimilar to a given county, a dissimilar SQL CI query can still be used to reduce the number of dissimilar Vendors and extract from that subset any row which VENDOR_NAME contains the keyword COUNTY OF. However, the local governments within the State of Virginia are composed of Counties and Independent Cities. As such the filter step can be enhanced to include Independent Cities or a new feature can be added to the database that correctly identifies each Vendor transaction if it is a local government transaction or not. Engineering this feature into the dataset is also useful when comparing the transactions/expenditures of the state with its local governments with respect to other characteristics not present in the expenditure database. For example, the correlation of money spent by the state in K-12 and higher education with the amount of students within each county and independent city that finish high school and university.

Similarity and dissimilarity queries can be customized to restrict transactions to a particular time period, e.g., a specific quarter or a month. The query would use vector additions over vectors to compute new vectors (e.g., create a vector for transaction patterns of a Vendor VENDOR_A in quarter Q3 by adding vectors for VENDOR_A and quarter_Q3), and use the modified vectors to find the target customers.

The patterns observed in these queries can be applied to other domains as well, e.g., identifying patients that are taking similar drugs, but with different brand names, or identifying food items
with similar ingredients, or recommending mutual funds with similar investment strategies. As we will see in the next section, the similarity query can be applied to other data types, such as images.

Another use case provides an illustration of a predictive CI query which uses a model that is externally trained using an unstructured data source or another database (Figure 5). For this scenario, two completely different datasets are used. The first dataset is a sales dataset that describes customer purchasing patterns in terms of products. The second dataset describes the products in terms of potential infections and/or deceases. Consider a scenario of a recall of various fresh fruit types due to possible listeria infection. This example assumes that we have built a word embedding model using the recall notices as an external source. Assume that recall document lists all fruits impacted by the possible listeria infection, e.g., Apples, Peaches, Plums, Nectarines,... The model will create vectors for all these words and the vector for the word listeria will be closer to the vectors of Apples, Peaches, Plums, etc. Now, we can import this model and use it to query the sales database to find out which customers have bought items that may be affected by this recall, as defined by the external source. As Figure 5 shows, the similarity() UDF is used to identify those purchases that contain items similar to listeria, such as Apples. This example demonstrates a very powerful ability of CI queries that enables users to query a database using recall notices as an external source. Assume that recall document lists all fruits impacted by the possible listeria infection, e.g., Apples, Peaches, Plums, etc. Now, we can import this model and use it to query the sales database to find out which customers have bought items that may be affected by this recall, as defined by the external source. As Figure 5 shows, the similarity() UDF is used to identify those purchases that contain items similar to listeria, such as Apples. This example demonstrates a very powerful ability of CI queries that enables users to query a database using recall notices as an external source. Assume that recall document lists all fruits impacted by the possible listeria infection, e.g., Apples, Peaches, Plums, etc. Now, we can import this model and use it to query the sales database to find out which customers have bought items that may be affected by this recall, as defined by the external source.

### 3.3 Cognitive OLAP Queries

Figure 6 presents a simple example of using similarities in the context of a traditional SQL aggregation query. This CI query aims to determine the maximum amount a State Agency paid to in the Expenses table for each Vendor that is similar to a specified Vendor, Vendor_Y. The result is collated using the values of the Vendor, the Agency and ordered by the total expense paid. As illustrated earlier, the UDF proximityCust_NameUDF defined for similarity queries is also used in this scenario. The UDF can use either an externally trained or locally trained model. This query can be easily adapted to support other SQL aggregation functions such as MAX(), MIN(), and AVG(). This query can be further extended to support ROLLUP operations over the aggregated values [6].

We are also exploring integration of cognitive capabilities into additional SQL operators, e.g., IN and BETWEEN. For example, one or both of the value ranges for the BETWEEN operator can be computed using a similarity CI query. For an IN query, the associated set of choices can be generated by a similarity or inductive reasoning query.

#### 3.4 Inductive Reasoning Queries

A unique feature of word-embedding vectors is their capability to answer inductive reasoning queries that enable an individual to reason from part to whole, or from particular to general [11, 13]. Solutions to inductive reasoning queries exploit latent semantic structure in the trained model via algebraic operations on the corresponding vectors. We encapsulate these operations in UDFs to support following five types of inductive reasoning queries: analogies, semantic clustering, and analogy sequences, clustered analogies, and odd-man-out [11]. We discuss key inductive reasoning queries below:

- **Analogies:** Wikipedia defines analogy as a process of trans-ferring information or meaning from one subject to another. A common way of expressing an analogy is to use relationship between a pair of entities, source_1 and target_1, to reason about a possible target entity, target_2, associated with another known source entity, source_2. An example of an analogy query is Lawyer :: Client :: Doctor :: ?, whose answer is Patient. To solve an analogy problem of the form (X : Y :: Q : ?), one needs to find a token W whose meaning vector, V_W, is closest to the ideal response vector V_R, where V_R = (V_Q + V_Y - V_X) [11]. Recently, several solutions have been proposed to solve this formulation of the analogy query [7, 8, 10]. We have implemented the 3COSMUL approach [7] which uses both the absolute distance and direction for identifying the vector V_W as

\[
\text{argmax}_{W \in C} \frac{\cos(V_W, V_Q) \cos(V_W, V_Y)}{\cos(V_W, V_X) + \epsilon}
\]

where \( \epsilon = 0.001 \) is used to avoid the denominator becoming 0. Also, 3COSMUL converts the cosine similarity value of \( c \) to \( \frac{c + 1}{2} \) to ensure that the value being maximized is non-negative.

Figure 7 illustrates a CI query that performs an analogy computation on the relational variables using the analogyUDF(). This query aims to find a Vendor from the Expenditures table (Figure 10), whose relationship to the category, Q3, or Third Quarter, is similar to what Fairfax County Public Schools has with the category, Q1, or First Quarter (i.e., if Fairfax County Public Schools is the most prolific public school system in terms of expenditures during the first quarter, find such Vendors who are the most prolific spenders in the third quarter, excluding Fairfax County Public Schools). The analogyUDF() UDF fetches...
There are powerful extensions to SQL that are enabled by word vectors. For this we need the ability to refer to constituent tokens of a specified quarter. For example, we can use the analogyUDF() function to find a set of tokens that share the most dominant trait with the input token and the computed response vector. The similarity score is calculated using the cosineDistance(e1, e2) function, where e1 and e2 are the input and response vectors, respectively. The function can be used to capture relationships between variables of different types, for example, between numeric and textual data.

4.1 Experiencing the Cognitive Database

To illustrate the demonstration flow, we use a publically available cognitive database implementation as well as novel capabilities of CI queries using realistic enterprise-class database workloads. For demonstration purposes, we will be using a Linux-based Spark Scala implementation of the cognitive database system. We plan to demonstrate both the end-to-end features of the Spark based cognitive database implementation as well as novel capabilities of CI queries using realistic enterprise-class database workloads. The audience will be able to step through various stages of the cognitive database life cycle, namely, data pre-processing, training, model management, and querying. Participants will also be able to interact with the database via Python-based (via Jupyter Notebook) interfaces of the cognitive database system and experience their capabilities. The rest of the section provides a glimpse of the demo by illustrating a real database use case.
expenditure for a year (e.g., 2016) listing details of every transaction such as vendor name, corresponding state agency, which government fund was used etc, by 190950 unique customers or Vendors. Note that Vendors can be individuals, and/or private and public institutions, such as county agencies, school districts, banks, businesses, etc. The data was initially organized as separate files, each belonging to a quarter. A single file was created and two other features engineered and added to the dataset, mainly Quarter and County, identifying which quarter the transaction happened and if the transaction is associated with one of the 133 counties and independent cities in the state.

Figure 10: Pre-processing the Virginia Expenditure Dataset

Figure 10(A) illustrates a portion of the CSV file that represents the original database which contains both text and numeric values. The first step in the pre-processing phase is to convert the input database into a meaningful text corpus (Figure 10(B)). This textification process is implemented using Python scripts that converts a relational row into a sentence. Any original text entity is converted to an equivalent token, e.g., a vendor name, BURDEN_COST, is converted to a string token VENDOR_NAME_BURDEN_COST. For numeric values, e.g., amount of 1524.61, the preprocessing stage first clusters the values using the K-Means algorithm, and then replaces the numeric value by a string that represents the associated cluster (e.g., AMOUNT_0 for value 1524.61). The resultant text document is then used as input to build the word embedding model. The word embedding model generates a d dimensional feature (meaning) vector for each unique string token in the vocabulary. For example, Figure 10(C) presents a vector of dimension 300 for the string token VENDOR_NAME_BURDEN_COST, that corresponds to the value BURDEN_COST in the original database. Our demonstration will go over various pre-processing stages in detail to explain the text conversion and model building scripts.

Once the model is built, the user program can load the vectors and use them in the SQL queries. Figure 3 presents an example of a SQL CI similarity query. The goal of the query is to identify vendors that have overall similar transactional behavior to an input vendor over the entire dataset (i.e., transacted with the same agencies with similar amounts etc.) The SQL query uses an UDF, proximityCust_NameUDF(VENDOR_NAME, "$aVendor") AS proximityValue FROM Expenses

Figure 11: Most similar vendors to COUNTY OF ARLINGTON

As described in Section 3.2 a similarity CI query can be easily modified to implement a dissimilarity query by changing the comparison term and ordering in ascending order. Since the original dataset contains all types of transactions and Vendors this query would not be very useful. For example, a single transaction performed by an individual would be very different from the 87 transactions of the County of Arlington. This is expected and would not provide any useful knowledge. However, if the dissimilar results are filtered down to focus on a particular group of transactions, insight can be obtained that leads to other analysis. The outcome of such query is shown in Figure 13.

For the OLAP and Analogy examples we use another set of Vendors from the Expenditures dataset. In this case we are looking at the money spent by public school districts per quarter to understand how much money the districts spend for a given quarter when it is compared to a given district. Such district is the Fairfax County Public Schools since it is the largest district with 2x to 3x more students than the other closest school districts, as illustrated
val result2_df = spark.sql(s""
SELECT VENDOR_NAME,
proximityCust_NameUDF(VENDOR_NAME,'$aVendor')
AS proximityValue FROM Expenses
HAVING (proximityValue < $proximity AND proximityValue > -1)
ORDER BY proximityValue ASC");
result2_df.filter(result2_df("VENDOR_NAME").contains("COUNTY OF")).show(100,false)

Figure 12: Example of a CI dissimilar query

Figure 13: Most dissimilar vendors to COUNTY OF ARLINGTON

in Figure 14, where the table is ordered in descending order by population and it includes the K-12 student population.

<table>
<thead>
<tr>
<th>COUNTY</th>
<th>POPULATION</th>
<th># K-12 STUDENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIRFAX COUNTY</td>
<td>1119953</td>
<td>228113</td>
</tr>
<tr>
<td>CITY OF VIRGINIA BEACH</td>
<td>445378</td>
<td>83848</td>
</tr>
<tr>
<td>PRINCE WILLIAM COUNTY</td>
<td>430100</td>
<td>103181</td>
</tr>
<tr>
<td>LOUDOUN COUNTY</td>
<td>327248</td>
<td>93602</td>
</tr>
<tr>
<td>CHESTERFIELD COUNTY</td>
<td>323862</td>
<td>70846</td>
</tr>
<tr>
<td>HENRICO COUNTY</td>
<td>315220</td>
<td>63156</td>
</tr>
</tbody>
</table>

Figure 14: Largest Counties in the State of Virginia and corresponding K-12 student population

Figure 15: OLAP CI query results for Fairfax County Public Schools

The immediate benefit of this query is that when combined with data like the one described in Figure 14 it gives an idea how much the Department of Education is spending on each student per county. If there are other State Agencies responsible for handling K-12 education expenses, one can get a very good picture of the total amount per county and per student within the county. With external data, such as successful graduation rate and cost of operation (Buildings, Materials, Teachers, Special Education, etc.) the State is better positioned to determine if the money allocated per county is producing the desired results. Note that the gathering of information per county can be obtained directly with multiple filter and aggregation SQL queries, particularly after the County and Independent City identifier has been engineered into the Expenditure dataset. However, the SQL CI query significantly simplifies the access of such data and without requiring that extra features are added to the dataset. One can easily modify this query and perform the same type of analysis to gather the amount spent per county for a given program or a set of programs. The outcome of such query is shown in Figure 16 and it shows that amongst the top school districts the money spend in K-12 education comes from the same program Standards of Quality for Public Education(SOQ) directly addressing the article in the State Constitution that requires the Board of Education to prescribe standards of quality for the public schools. It is worth mentioning that the first 100 entries obtained with the OLAP CI query are all but one directly related to the public schools and the money invested in education. Furthermore, they show that the money comes from the SOQ program or from federal assistance programs designed to help local education. The entries also show that the budgets for public education are not necessarily paid directly to the school districts. In some cases the county or the county treasurer are involved in the administration of the funds even though they are being used for education. The semantic
relationships between the transactions contain such information as obtained by the query.

\[
\text{SELECT VENDOR\_NAME, SPRG\_SUB\_PROGRAM\_NAME, SUM(AMOUNT) as max_value FROM Expenses INNER JOIN Programs ON Expenses.SPRG\_SUB\_PROGRAM\_KEY = Programs.SPRG\_SUB\_PROGRAM\_KEY WHERE proximityCust\_NameUDF(VENDOR\_NAME, \'$Vendor\_Y\') > } \text{ $proximity} \text{ GROUP BY VENDOR\_NAME, SPRG\_SUB\_PROGRAM\_NAME ORDER BY max_value DESC}
\]

Figure 16: OLAP CI query with focus on Programs instead of Agencies

To demonstrate an analogy query we use the CI query in Figure 7. As described, we are looking for Vendors that are similar to the transactions of Fairfax County Public Schools in a given quarter in terms of transactions with a State Agency and another quarter. In a normal SQL query one would collect all the transactions the Vendor had with any State Agency in a given quarter. Repeat the process for other quarters. Afterwards, one would compare both sets of data to determine the Vendors and State agencies that showed a similar behavior for a given quarter, collect all the transactions associated with the Vendor and State Agency and list the results in descending order by the aggregated amount value. Conversely, one can use a single analogy CI query and get a similar list without the extensive comparisons. Once again, by combining the vectors of vendors and quarters the CI query captures vendors and state agencies describing transactions in the area of county public education, see Figure 17. This demonstrates that applying Equation 1 still results in a vector that contains enough embedded information to extract Vendors and State Agencies. Without any additional filtering the resulting list shows other public school systems dealing with similar State Agencies. Like the previous example this CI query can be easily modified to analyze another field, for example a State Program.

4.2 Using Python Interfaces

In this section, we describe the Python implementation of Cognitive Databases using two different approaches for creating CI queries. One approach uses Pandas, a library used for data analysis and manipulation, and sqlite3, a module that provides access to the lightweight database, SQLite. The other approach uses PySpark, the Spark Python API, in a case where big data processing is required. In both cases, we will use Jupyter Notebook [1], a web-based application for executing and sharing code, to implement the CI queries for interacting with the Cognitive Database. During demonstration,

\[
\text{SELECT VENDOR\_NAME,AGY\_AGENCY\_NAME, QUARTER, SUM(AMOUNT) as max_value FROM Expenses INNER JOIN Agencies ON Expenses.AGY\_AGENCY\_KEY = Agencies.AGY\_AGENCY\_KEY WHERE analogyUDF(\'$aVendor1\', \'$aVendor2\', \'$aVendor3\', VENDOR\_NAME) > } \text{ $proximity AND QUARTER == \'$aVendor3\' GROUP BY VENDOR\_NAME, AGY\_AGENCY\_NAME, QUARTER ORDER BY max_value DESC}
\]

Figure 17: Example of Analogy CI query comparing Fairfax County in Q1 with Vendors in Q3

the audience will be able to interact with the Jupyter notebook and modify the CI queries.

Pandas provides a rich API for data analysis and manipulation. We use Pandas in conjunction with sqlite3 to implement the CI queries. Similarly as in the Scala approach, the Cognitive Database is initialized with the passed in model vectors and data files the user intends to use for analysis. During the initialization process, the data is converted from a Pandas dataframe to a SQLite in-memory database. Then, sqlite3’s create_function method is used to create the cognitive UDFs. From the user’s perspective, using pandas SQL methods and the internal Cognitive Database connection, they can perform CI Queries to expose powerful semantic relationships (Figure 18).

Figure 18: Example CI Queries in Jupyter Notebook using Pandas and sqlite3

The PySpark approach is useful when the user needs to perform big data processing. In the PySpark approach, we call Scala UDFs from PySpark by packaging and building the Scala UDF code into a jar file and specifying the jar in the spark-defaults.conf configuration file. Using the SparkContext’s internal JVM, we are able to access and register Scala UDFs and thus the user can use them in CI queries within Python. We chose this approach instead of the Python API’s support for UDFs because of the overhead when serializing the data between the Python interpreter and the JVM. When creating UDFs directly in Scala, it is easily accessible by the executor JVM and won’t be a big performance hit. The CI queries
in Python using PySpark look similarly as they do in the Scala implementation (Figure 19).

![Figure 19: Example CI Queries in Jupyter Notebook using PySpark](image)

### 4.3 Summary

We plan to demonstrate capabilities of the Spark-based Cognitive Database using both Scala and Python interfaces. We will be demonstrating various types of CI queries over enterprise-class multimodal databases, such as the State of Virginia Expenditure data. The demonstration will allow the participants to interact with the pre-processing tools as well as with different CI queries.

### REFERENCES


