

# 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; Online analytical processing 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,

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

KDD'18 Deep Learning Day, August 2018, London, UK

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

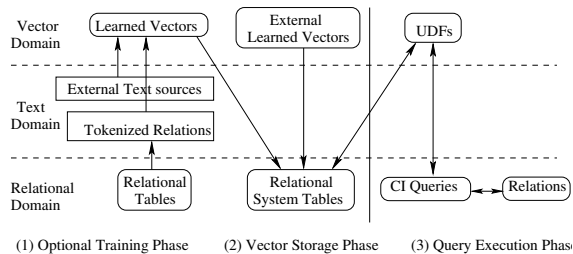
traditional SQL queries rely mainly on value-based predicates to detect patterns. For example, the aggregate of all transactions within a timeframe or the sorting of transaction amounts by decreasing order of value. The meaningful relationship and interpretation between the data of multiple columns is 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 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



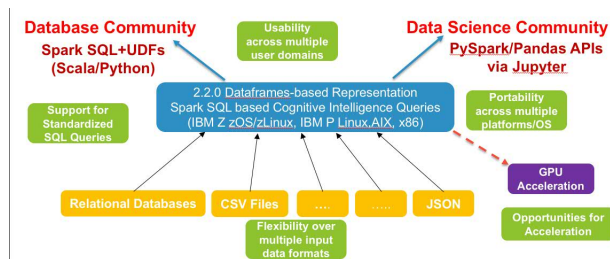
**Figure 1: End-to-end execution flow of a cognitive relational database**

database is used to train the model. Our training approach is characterized by two unique aspects: (1) Using *meaningful* unstructured text representation of the structured relational data as input to the training process (i.e. irrespective of the associated SQL types, all entries from a relational database are converted to text tokens representing them), and (2) Using the *unsupervised* word embedding technique to generate meaning vectors from the input text corpus. We have modified the classical word embedding approach [9] to address various subtleties with the relational data model (e.g., supporting primary keys). Every unique token from the input corpus is associated with a vector of dimension  $d$ . In our scenario, a text token in a training set can represent either text, numeric, or image data. Thus, the model builds a joint latent representation that integrates information across different *modalities* using *untyped uniform* feature (or meaning) vectors. Alternatively, this phase can also use pre-trained vectors built from an external data repository (e.g., Wikipedia).

Following vector training, the resultant vectors are stored in a relational system table (phase 2). At runtime, the SQL query execution engine uses various user-defined functions (UDFs) that fetch the trained vectors from the system table as needed and answer CI queries (phase 3). These UDFs take typed relational values as input and compute semantic relationships between them using uniformly untyped meaning vectors. This enables the relational database system to seamlessly analyze data of different types (e.g., text, numeric values, and images) using the same SQL CI query. Our current implementation is built on the Apache Spark 2.2.0 infrastructure [2] using Scala. It runs on a variety of processors and operating systems. The system implementation follows the cognitive database execution flow as presented in Figure 1. The system first initializes in-memory Spark Dataframes from external data sources (e.g., relational database or CSV files), loads the associated word embedding model into another Spark Dataframe (which can be created offline from either the database being queried or external knowledge bases such as Wikipedia), and then invokes the Cognitive Intelligence queries using Spark SQL. The SQL queries invoke Scala-based cognitive UDFs to enable computations on the meaning vectors. A Python based implementation (Figure 2) is also provided, thus allowing the system to be used by both the database and data science communities.

The distinguishing aspect of cognitive intelligence queries, contextual semantic comparison between relational variables, is implemented 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 cognitive UDF is to calculate similarity between a pair of tokens by computing the cosine distance between the corresponding vectors. 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.



**Figure 2: Spark Implementation of a cognitive relational database**

Our current implementation supports four types of CI SQL queries: similarity based classification, inductive reasoning, prediction, 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 relational variables based on similarity or dissimilarity between the input variables. Each relational variable can be either set or sequence of tokens. In case of sequences, computation of the final similarity value takes the *ordering* of tokens into account. The similarity value is then used to classify and group related data. The inductive reasoning queries exploit latent semantic information in the database to reason from *part* to *whole*, or from *particular* to *general*. We support five different types of inductive reasoning queries: analogies, semantic clustering, analogy sequences, clustered analogies, and odd-man-out. Given an item from an external data corpus (which is not present in a database), the predictive CI query can identify items from the database that are similar or dissimilar to the external item by using the externally trained model. Finally, cognitive OLAP allows SQL aggregation functions such as  $MAX()$ ,  $MIN()$  or  $AVG()$  over a set that is identified by contextual similarity computations on the relational variables.

### 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 purposes 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.

```
SELECT VENDOR_NAME, proximityCust_NameUDF(VENDOR_NAME,
'$aVendor') 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.

```
scala> evalSimilarVendor("county of arlington")
+-----+-----+
|VENDOR_NAME|proximityValue|
+-----+-----+
|COUNTY OF GILES|0.84385705|
|COUNTY OF CAROLINE|0.8353094|
|COUNTY OF JAMES CITY|0.8121486|
+-----+-----+
```

**Figure 4: Most similar vendors to County of Arlington**

Section 4 presents in detail analysis techniques that show why the similarity results for *County of Arlington* are actually other counties and not some other random Vendor. Note that the SQL command to get such results did not filter any data before or after the SELECT statement with respect to the 190950 unique Vendors that had one or more transactions with the state of Virginia.

### 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

---

```

SELECT X.custID, similarityUDF(X.Items, 'listeria') AS
similarity
FROM sales X
WHERE similarityUDF(X.Items, 'listeria') > 0.3
ORDER BY similarity DESC

```

---

**Figure 5: Example of a prediction query: find customers that have purchased items affected by a listeria recall**

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 `similarityUDF()` 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 a token **not present** in the database (e.g., *listeria*). This capability can be applied to different scenarios in which recent, updatable information, can be used to query historical data. For example, a model built using FDA recall notices could be used to identify those customers who have purchased medicines similar to the recalled medicines.

### 3.3 Cognitive OLAP Queries

Figure 6 presents a simple example of using semantic 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

---

```

SELECT VENDOR_NAME, AGY_AGENCY_NAME, SUM(AMOUNT) as
max_value
FROM Expenses INNER JOIN Agencies ON Expenses.AGY_AGENCY_KEY
= Agencies.AGY_AGENCY_KEY WHERE
proximityCust_NameUDF(VENDOR_NAME, '$Vendor_Y') > $proximity
GROUP BY VENDOR_NAME, AGY_AGENCY_NAME ORDER BY max_value DESC

```

---

**Figure 6: Example of a cognitive OLAP (aggregation) query**

choices can be generated by a similarity or inductive reasoning queries.

### 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 transferring *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

$$\operatorname{argmax}_{W \in C} \frac{\cos(V_W, V_Q)\cos(V_W, V_Y)}{\cos(V_W, V_X) + \epsilon} \quad (1)$$

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

---

```

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) > $proximity AND QUARTER == '$aVendor3'
GROUP BY VENDOR_NAME, AGY_AGENCY_NAME, QUARTER ORDER BY
max_value DESC
    
```

---

**Figure 7: Example of an analogy query**

---

```

SELECT VENDOR_NAME, semClustUDF('$vendorA', '$vendorB',
'$vendorC', VENDOR_NAME)
AS proximityValue FROM Expenses
HAVING proximityValue > $proximity
ORDER BY proximityValue DESC
    
```

---

**Figure 8: Example of a semantic clustering query**

vectors for the input variables, and using the 3COSMUL approach, returns the analogy score between a vector corresponding to the input token and the computed response vector. Those rows, whose variables (e.g., VENDOR\_NAME) have analogy score greater than a specified bound (0.5), are selected. To facilitate the analysis the results are filtered by the quarter of interest (Q3) and sorted in descending order by the total sum of expenditures and also listing the agency such Vendor(s) had the most transaction with. Since analogy operation is implemented using *untyped* vectors, analogyUDF() UDF can be used to capture relationships between variables of different types, e.g., images and text.

- Semantic Clustering:** Given a set of input entities,  $\{X, Y, Z, \dots\}$ , the semantic clustering process identifies a set of entities,  $\{W, \dots\}$ , that share the most dominant trait with the input data. The semantic clustering operation has a wide set of applications, including customer segmentation, recommendation, etc. Figure 8 presents a CI query which uses a semantic clustering UDF, semClustUDF(), to identify vendors that have the most common attributes with the input set of vendors, e.g., vendorA, vendorB, and vendorC. For solving a semantic clustering query of the form,  $(X, Y, Z ::?)$ , one needs to find a set of tokens  $S_w = \{W_1, W_2, \dots, W_i\}$  whose meaning vectors  $V_{w_i}$  are most similar to the *centroid* of vectors  $V_X, V_Y$ , and  $V_Z$  (the centroid vectors captures the dominant features of the input entities).

Another intriguing extension involves using contextual similarities to choose *members* of the schema dimension hierarchy for aggregation operations like ROLLUP or CUBE. For example, instead of aggregating over all quarters for all years, one can use only those quarters that are semantically similar to a specified quarter.

### 3.5 Cognitive Extensions to the Relational Data Model

There are powerful extensions to SQL that are enabled by word vectors. For this we need the ability to refer to constituent tokens

(extracted during textification) in columns of rows, in whole rows and in whole relations. The extension is via a declaration, in the FROM clause, of the form Token e1 that states that variable e1 refers to a token. To *locate* a token we use, in the WHERE clause, predicates of the form contains(E, e1) where E can be a column in a row (e.g., EMP.Address), a whole row (e.g., EMP.\*) or a whole relation (e.g., EMP). With this extension we can easily express queries such as asking for an employee whose Address contains a token which is very close to a token in a row in the DEPT relation (Figure 9). Furthermore, we can also extend SQL with relational variables, say of the form \$R and column variables, say X, whose names are not specified at query writing time; they are bound at runtime. We can then use these variables in queries, in conjunction with Token variables. This enables database querying *without explicit schema knowledge* which is useful for exploring a database. Interestingly, the notation \$R.X is basically syntactic sugar. A software translation tool can substitute for \$R.X an actual table name and an actual column. Then, perform the query for each such substitution and return the *union* of the results [4].

---

```

SELECT EMP.Name, EMP.Salary, DEPT.Name
FROM EMP, DEPT, Token e1, e2
WHERE contains(EMP.Address, e1) AND
contains(DEPT.*, e2) AND
cosineDistance(e1, e2) > 0.75
    
```

---

**Figure 9: Example of an SQL query with entities**

Lastly, one may wonder how numeric bounds on UDFs (e.g., cosineDistance(e1, e2) > 0.75 in Figure 9) are determined. The short answer is that these bounds are application dependent, much like hyperparameters in machine learning. One learns these by exploring the underlying database and running queries. In the future, we envision a tool that can guide users to select an appropriate numeric bound for a particular CI query.

## 4 DEMONSTRATION OVERVIEW

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 Scala (via command-line) or Python-based (via Jupyter Notebook) interfaces of the cognitive database system and modify the CI queries to experience their capabilities. The rest of the section provides a glimpse of the demo by illustrating a real database usecase.

### 4.1 Experiencing the Cognitive Database

To illustrate the demonstration flow, we use a publically available expenditure dataset from the State of Virginia[12]. It is a fairly large dataset (at least 5.3 Million records), that describes state-wide

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.

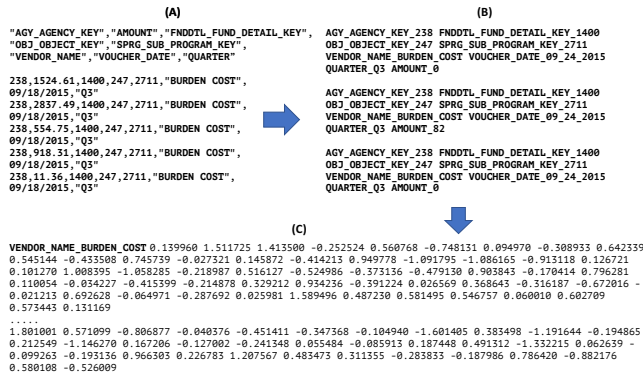


Figure 10: Pre-processing the Virginia Expenditure Dataset

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(), which first converts the input relational variables into corresponding text tokens, fetches the

corresponding vectors from the loaded model, and computes a similarity value by performing nearest neighbor computations on those vectors. Figure 11(A) presents the results of this query for the vendor COUNTY OF ARLINGTON as already shown in Figure 4. In addition Figure 11(B) explains why the Vendors listed are similar based on their transaction history. Since the similarity is based on the commonality of data, the table details how common each Vendor is to the County of Arlington. For example, the County of Giles had a total of 78 transactions with the state. They dealt with three state agencies of which two are the same agencies the County of Arlington dealt with. Of the 78 transactions, 74 were with common agencies for both vendors. Similar analysis is performed for the other fields describing a transaction (e.g., Funds, Objectives, Programs, etc). For simplicity not all the fields characterizing a transaction are included in the table. Missing are the date of transaction, which quarter it happened and if it is a county type transaction or not.

```

SELECT VENDOR_NAME, proximityCust_NameUDF(VENDOR_NAME,
'${aVendor}') AS proximityValue FROM Expenses
WHERE proximityValue > 0.5
ORDER BY proximityValue DESC
    
```

```

scala> evalSimilarVendor("county of arlington")
+-----+-----+
|VENDOR_NAME|proximityValue|
+-----+-----+
|COUNTY OF GILES|0.84385705|
|COUNTY OF CAROLINE|0.8353094|
|COUNTY OF JAMES CITY|0.8121486|
+-----+-----+
    
```

(A)

	County of Arlington	County of Giles	County of Caroline	County of James City
# Transactions	87	78	97	60
AMOUNT	5.3 Million	0.75 Million	0.97 Million	.79 Million
Agencies	2	3 / 2 / 74	3 / 2 / 92	1 / 1 / 60
Funds	2	4 / 2 / 74	4 / 2 / 92	1 / 1 / 60
Objectives	2	3 / 2 / 76	4 / 2 / 92	1 / 1 / 60
Programs	15	14 / 11 / 68	17 / 14 / 86	10 / 10 / 60

(B)

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

```
scala> evalDissimilarVendor("county of arlington")
-----
|VENDOR_NAME|proximityValue|
-----
|COUNTY OF GREENSVILLE VIRGINIA|0.36435186|
|COUNTY OF FAIRFAX-SWMP|0.36581263|
|COUNTY OF BUCKS|0.36723676|
-----
```

(A)

	County of Arlington	County of Greenville Virginia	County of Fairfax-SWMP	County of Bucks
# Transactions	87	34	193	1
AMOUNT	5.3 Million	0.018 Million	4.87 Million	3.75 dollars
Agencies	2	1 / 0 / 0	11 / 0 / 0	1 / 0 / 0
Funds	2	1 / 0 / 0	15 / 0 / 0	1 / 0 / 0
Objectives	2	5 / 0 / 0	9 / 1 / 22	1 / 0 / 0
Programs	15	1 / 0 / 0	19 / 0 / 0	1 / 0 / 0

(B)

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.

COUNTY	POPULATION	# K-12 STUDENTS
FAIRFAX COUNTY	1118683	228113
CITY OF VIRGINIA BEACH	445378	83848
PRINCE WILLIAM COUNTY	430100	103181
LOUDOUN COUNTY	337248	93602
CHESTERFIELD COUNTY	323862	70846
HENRICO COUNTY	315200	63156

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

Figure 6 presents a simple cognitive OLAP query that identifies Vendors and State Agencies that are similar to a given Vendor, in this case the Fairfax County Public Schools. The query identifies the most common Agency amongst all the Vendors that are similar to the target Vendor adds up the transaction value associated with each Vendor and presents the results in descending order as shown in Figure 15. The first observation is the same observed with similarity CI queries where the transactions of the different Vendors have many fields and values in common. The second observation is that the query automatically picks other school districts indicating that these school districts work with the same agencies, funds, objectives and programs as Fairfax County Public Schools for the most part. The third observation is that the sorted order of the

CI query result is very similar to the order the counties are sorted from a population count standpoint as shown in Figure 14. It does not follow the sorted order from a school population standpoint.

```
SELECT VENDOR_NAME, AGY_AGENCY_NAME, SUM(AMOUNT) as
max_value
FROM Expenses INNER JOIN Agencies ON Expenses.AGY_AGENCY_KEY
= Agencies.AGY_AGENCY_KEY WHERE
proximityCust_NameUDF(VENDOR_NAME, '$Vendor_Y') > $proximity
GROUP BY VENDOR_NAME, AGY_AGENCY_NAME ORDER BY max_value DESC
```

VENDOR_NAME	AGY_AGENCY_NAME	max_value
PRINCE WILLIAM COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	224,397,625.51
VA BEACH CITY PUBLIC SCHOOL	Department of Education - Direct Aid to Public Education	157,261,696.96
CHESTERFIELD COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	150,974,715.77
LOUDOUN COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	134,376,169.12
HENRICO COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	123,888,652.60

	Fairfax County	Prince William	VA Beach County
# Transactions	841	319	214
AMOUNT	699M	224M	157M
Agencies	22	15 / 11 / 308	2 / 2 / 214
Funds	26	18 / 14 / 307	4 / 4 / 214
Objectives	16	11 / 6 / 312	2 / 2 / 214
Programs	23	20 / 14 / 303	9 / 8 / 213

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.

```
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') > $proximity
GROUP BY VENDOR_NAME, SPRG_SUB_PROGRAM_NAME ORDER BY
max_value DESC
```

VENDOR_NAME	SPRG_SUB_PROGRAM_NAME	max_value
PRINCE WILLIAM COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	1186,116,668.86
VIA BEACH CITY PUBLIC SCHOOL	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	135,872,302.69
CHESTERFIELD COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	132,798,416.37
LOUDOUN COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	121,667,011.82
HENRICO COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	102,053,637.31
CHESAPEAKE CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	88,741,581.47
NORFOLK CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	70,875,202.43
NEWPORT NEWS CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	69,233,452.31
STAFFORD COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	61,893,535.57
SPOITS/VANLIA COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	53,476,060.96
HAMPTON CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	48,704,190.98
RICHMOND CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	48,097,018.54
HANOVER COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	37,352,642.31
PORTSMOUTH CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	36,060,400.40
ROANKE COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	36,005,192.59
ROANOKE CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (500)	32,430,443.13
RICHMOND CITY PUBLIC SCHOOLS	FEDERAL ASSISTANCE TO LOCAL EDUCATION PROGRAMS	31,586,657.21
PRINCE WILLIAM COUNTY PUBLIC SCHOOLS	FEDERAL ASSISTANCE TO LOCAL EDUCATION PROGRAMS	31,339,512.76

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 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,

```
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) > $proximity AND QUARTER == '$aVendor3'
GROUP BY VENDOR_NAME, AGY_AGENCY_NAME, QUARTER ORDER BY
max_value DESC
```

VENDOR_NAME	AGY_AGENCY_NAME	QUARTER	max_value
VIA BEACH CITY PUBLIC SCHOOL	Department of Education - Direct Aid to Public Education	Q3	63,199,782.55
LOUDOUN COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	54,973,992.63
AUGUSTA COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	9,894,631.57
MANASSAS CITY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	8,919,865.00
ALEXANDRIA CITY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	7,860,007.94
PRINCE EDWARD COUNTY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	2,674,590.93
FAIRFAX CITY TREASURER	Department of Accounts Transfer Payments	Q3	1,898,671.94
FAIRFAX CITY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	1,874,609.29
FAIRFAX CITY TREASURER	Department of Education - Direct Aid to Public Education	Q3	853,910.50

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).

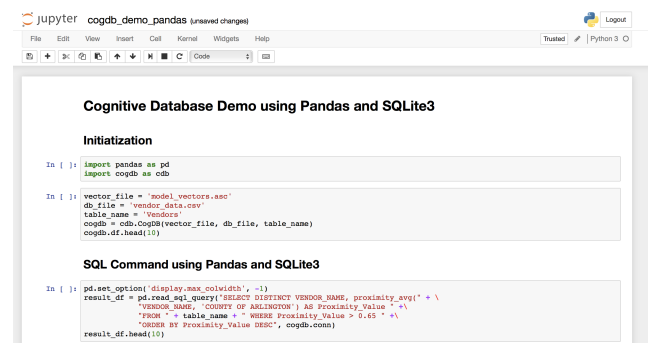


Figure 18: Example CI Queries in Jupyter Notebook using Pandas and sqlite3

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).

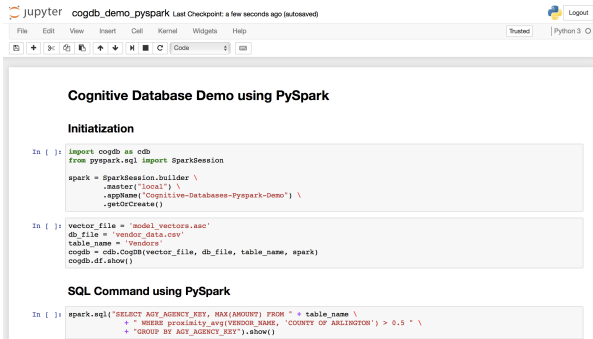


Figure 19: Example CI Queries in Jupyter Notebook using PySpark

### 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 multi-modal 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

- [1] 2017. Jupyter Notebook: Open-source web application for creating and sharing documents. (2017). <http://jupyter.org/>
- [2] Apache Foundation. 2017. Apache Spark: A fast and general engine for large-scale data processing. <http://spark.apache.org.> (2017). Release 2.2.
- [3] Rajesh Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. 2017. Cognitive Database: A Step towards Endowing Relational Databases with Artificial Intelligence Capabilities. *CoRR abs/1712.07199* (December 2017). <http://arxiv.org/abs/1712.07199>
- [4] Rajesh Bordawekar and Oded Shmueli. 2016. Enabling Cognitive Intelligence Queries in Relational Databases using Low-dimensional Word Embeddings. *CoRR abs/1603.07185* (March 2016). <http://arxiv.org/abs/1603.07185>
- [5] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning (DEEM'17)*. ACM, New York, NY, USA, Article 5, 4 pages.
- [6] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. 1997. Range Queries in OLAP Data Cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 73–88.
- [7] Omer Levy and Yoav Goldberg. 2014. Linguistic Regularities in Sparse and Explicit Word Representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning, CoNLL 2014*. 171–180. <http://aclweb.org/anthology/W/W14/W14-1618.pdf>
- [8] Tal Linzen. 2016. Issues in evaluating semantic spaces using word analogies. *arXiv preprint arXiv:1606.07736* (2016).
- [9] Tomas Mikolov. 2013. word2vec: Tool for computing continuous distributed representations of words. (2013). [github.com/tmikolov/word2vec](https://github.com/tmikolov/word2vec).
- [10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119. <http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality>
- [11] David E Rumelhart and Adele A Abrahamson. 1973. A model for analogical reasoning. *Cognitive Psychology* 5, 1 (1973), 1 – 28. DOI:[https://doi.org/10.1016/0010-0285\(73\)90023-6](https://doi.org/10.1016/0010-0285(73)90023-6)
- [12] State of Virginia. 2016. State of Virginia 2016 Expenditures. <http://www.datapoint.apa.virginia.gov/>. (2016).
- [13] Robert J Sternberg and Michael K Gardner. 1979. *Unities in Inductive Reasoning*. Technical Report Technical rept. no. 18, 1 Jul-30 Sep 79. Yale University. <http://www.dtic.mil/docs/citations/ADA079701>