Demonstrating Semantic SQL Queries over Relational Data using the AI-Powered Database

José Luis Neves
IBM Systems
Poughkeepsie, NY 12601
jneves@us.ibm.com

Rajesh Bordawekar
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
bordaw@us.ibm.com

Elpida Tzortzatos
IBM Systems
Poughkeepsie, NY 12601
elpida@us.ibm.com

ABSTRACT

This paper demonstrates key capabilities of AI-Powered Database, a novel relational database system which uses an unsupervised neural network model to facilitate semantic queries over relational data. Our neural network model, called `db2Vec`, applies vector embedding techniques 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 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.

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 are unable to extract and exploit semantic relationships that are collectively generated by tokens in a database relation.

This paper discusses AI-Powered Database [5, 7, 8], a novel relational database system, which uses an unsupervised neural network based approach from Natural Language Processing, called *vector embedding*, to extract *latent* knowledge from a database table. The generated vector 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 AI-powered 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 AI-Powered 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 AI-powered database prototype.

The rest of the paper is organized as follows: In Section 2, we first summarize key design aspects of AI-powered 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 vector embedding model. All the examples are real SQL queries developed to generate the results presented later in the paper. Section 4 outlines key features of the AI-powered database system being demonstrated: data preprocessing 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. The data set used for demonstration purposes is a publicly available dataset, namely all the financial transactions recorded by the State of Virginia during the fiscal year 2015-16 [17]. Furthermore, we used the CI-queries to identify school spending per county and about how much the state spent on average on students per county. Finally, Ap-

1st International Workshop on Applied AI for Database Systems and Applications (AIDB'19), August 26, 2019, Los Angeles, California, CA, USA.

pendix A describes Python-based interfaces for AI-Powered Database.

2. BACKGROUND AND SYSTEM ARCHITECTURE

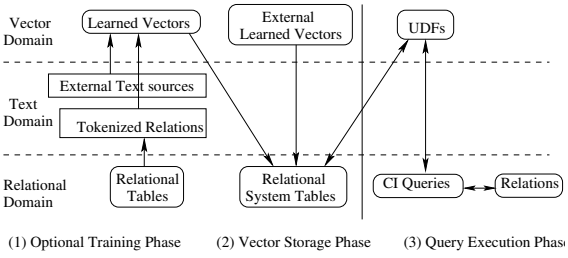


Figure 1: End-to-end execution flow of a AI-powered relational database

Figure 1 presents the three key phases in the execution flow of a AI-powered database. The `Training Phase` takes place when the database is used to train the model. It is only executed when a new model is created or needs to be updated. 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* vector embedding technique to generate meaning vectors from the input text corpus. Every unique token from the input corpus is associated with a meaning vector. The `Vector Storage Phase` manages which vector models are used during the query phase which could include the use of models trained from user specified tables or models trained from external sources. For externally trained models, the meaning of the tokens embedded in each vector are learned from external sources and not from specific user selected tables. Note that the tokens must be the same as the ones used in the query The `Query Phase` is where the user issues SQL statements to extract information from one or more databases and in the process he/she uses the trained model within specific SQL statements. This can be seen as an inference step where the Neural Network model (vectors) is used in SQL through the use of User Defined Functions (UDFs) to drive a SQL query with the learned information contained in the model. Details of the phases are explained in depth in the next sections.

2.1 Data Preparation

The data preparation stage takes a relational table with different SQL types as input and returns an unstructured but meaningful text corpus consisting of a set of sentences. This transformation allows us to generate a uniform semantic representation of different SQL types. This process of *textification* requires two stages: data pre-processing and text conversion (Figure 2).

The textification phase processes each relational row separately and converts data of different SQL data types to text. In some scenarios, one may want to build a model that also captures relational column names. For such cases, the pre-processing stage first processes the column names before processing the corresponding data.

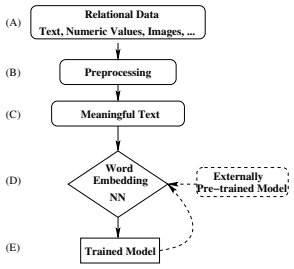


Figure 2: Multiple stages in creating a vector embedding model from relational tables

For SQL variables of `VARCHAR` type, preprocessing involves one or more of the following actions: (1) prepend the column attribute string to a SQL variable, (2) creating a single *concept* token from a group of `VARCHAR` tokens, e.g., `JPMorgan Chase` is represented as `JPMorgan_Chase`, (3) creating a single *token* for semantically similar sequences of `VARCHAR` tokens, e.g., two sequences of tokens, `bank of america` and `BANK OF AMERICA`, can be represented by a single compound token `BANK_OF_AMERICA`, and (4) Using an external mapping or domain-specific ontologies to create a common representative token for a group of different input tokens. In addition to text tokens, our current implementation supports numeric values and images (we assume that the database being queried contains a `VARCHAR` column storing links to the images).

In addition to text tokens, our current implementation supports numeric values and images (we assume that the database being queried contains a `VARCHAR` column storing links to the images). For numeric values, we use three different approaches to generate equivalent text representations: (1) creating a string version of the numerical value, e.g., value `100.0` for the column name `AMOUNT` can be represented by either `AMOUNT_100.0` or `‘‘100.0’’`, (2) User-managed categorization: a user can specify rules to define ranges for the numeric values and use them to generate string tokens for the numeric values. For example, consider values for a column name, `Cocoa Contents`. The value `80%`, can be replaced by the string token `choc_dark`, while the value `35%`, can be replaced by the string token `choc_med`, etc., and (3) user-directed clustering: an user can choose values of one or more numerical columns and cluster them using traditional clustering algorithms such as K-Means. Each numeric value is then replaced by a string representing the cluster in which that value lies.

For image data, we use approaches similar to ones used for numerical values. The first approach represents an image by its string token, e.g., a string representing the image path or a unique identifier. The second approach uses pre-existing classifers to cluster images into groups and then uses the cluster information as the string representation of the image. For example, one can use a domain-specific deep neural network (DNN) based classifier to cluster input images into classes [11] and then use the corresponding class information to create the string identifiers for the images. The final approach applies of-the-shelf image to tag generators, e.g., IBM Watson Visual Recognition System (VRS) [10], to extract image features and uses them as string identifiers for an image. For example, a Lion image can be represented by the following string features, `Animal`, `Mammal`, `Carnivore`,

`BigCat`, `Yellow`, etc.

Once text, numeric values and images are replaced by their text representations, a relational table can be viewed as unstructured meaningful text corpus to be used for building a vector embedding model. For `Null` values of these types, we replace them by the string `column_name_Null`. The methods outlined here can be applied to other data types such as SQL `Date` and spatial data types such as lattitude and longitude.

2.2 Model Training

Traditional word embedding approaches such as Word2Vec (W2V) [14] or GloVe [15] build vector embedding models from natural language text corpus using appropriate language models (e.g., for English). As these approaches fail to address various subtletiles with the relational data model (e.g., supporting primary keys, different data types, or `NULL` values), we have developed a novel vector embedding approach for relational tables, `db2Vec`. `db2Vec` operates on a text corpus generated by one or more relational tables. In this 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.

The `db2Vec` implementation varies from the traditional NLP approaches in a number of ways:

- A sentence generated from a relational row is generally not in any natural language such as English.[1] Therefore, W2V's assumption that within a sentence, the influence of any word on a nearby word decreases as the word distances increases, is not applicable. In our implementation, every token in a sentence has equal influence on all other tokens in that sentence, *irrespective of their positions*; i.e., we view a sentence generated from a relational table as *a bag of words*, rather than *an ordered sequence*.

- Unlike words in a natural language text, entities derived from relational tables are typed (type defined by the corresponding column attribute). `db2Vec` captures the type information for building the meaning vectors, e.g., if an entity appears in two different relational columns, `db2Vec`, treats the two instances as separate entities and build two different meaning vectors.

- For the traditional natural language usecases, the word embedding models generate limited-sized vocabularies (often defined by the language model). In our case, since the string entities are generated from source values of different types (e.g., unique string identifier to represent individual table rows), the vocabulary size can be very large [4].

- For relational data, we provide special consideration to primary keys. First, the traditional word embedding approaches discard less frequent words from computations. In our implementation, by default, every token, irrespective of its frequency, is assigned a vector. For an unique primary key (with singular occurrence), its meaning vector represents the meaning of the entire row.

- In some cases, one may want to build a model in which values of particular columns are given higher weightage for their contributions towards meanings of neighborhood words. Our implementation enables users to specify different weight ranges or importance (High, medium, or low) for different columns during model training.

- The `db2Vec` training algorithm provides special treatment for the entities corresponding to the SQL `NULL` (or equivalent) values. The `NULL` values are processed such that they do not contribute to the meanings of neighboring non-null entities; thus eliminating false similarites.

- The `db2Vec` implementation is designed to enable *incremental* training, i.e. the training system takes as input a pre-trained model and a new set of generated sentences, and returns an updated model. This capability is critical as a database can be updated regularly and one can not rebuild the model from scratch every time. The pre-trained model can be built from the database being queried, or from an external source such as text corpus, graph, or a database table [5].

- `db2Vec` supports building vector embedding models from multiple inter-related database tables linked via primary key-foreign key relationships. Forming a training corpus from multiple tables is non-trivial, and `db2Vec` supports different implementation options such as integrating multiple models built from different tables', building models from fewer un-normalized tables, etc. [5].

2.3 Query Execution

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 [3] is built on the Apache Spark 2.4.0 infrastructure [2] using Scala. It runs on a variety of processors and operating systems. The system implementation follows the AI-powered 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 3) 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,

[1]Currently, we assume that database tokens are specified using the English language.

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 $cos(v_1, v_2) = cos(v_1, v_2) = \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|}$. 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.

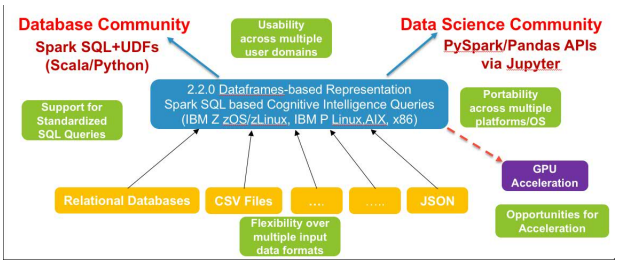


Figure 3: Spark Implementation of a AI-powered relational database

Our current implementation supports four classes of CI SQL queries: similarity based classification, inductive reasoning, prediction, and cognitive OLAP [5]. 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 fiscal year of 2015/2016. 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

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 4 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 4: Example of a SQL CI similarity query: find similar state customers based on expense transaction patterns

The query shown in Figure 4 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 5.

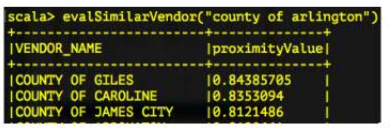


Figure 5: Most similar vendors to *County of Arlington*

```
SELECT VENDOR_NAME, similarityUDF(VENDOR_NAME, 'STEM') AS
similarity
FROM Expenses
WHERE similarityUDF(VENDOR_NAME, 'STEM') > 0.3
ORDER BY similarity DESC
```

Figure 6: Example of a prediction query: find state customers that purchased items affected related to STEM areas

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 cities 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.

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 6). For this scenario, two completely different datasets are used. The first dataset is the transactions dataset that describes all financial expenditures of the state of Virginia. The second dataset is a national dataset that describes the use of STEM (Science, Technology, Engineering, Mathematics) related products, such as books, laboratory data, experiments, etc. across all counties in United States. Consider a scenario where one wants to know how much the state of Virginia counties spend in STEM related activities and how does it compare to other counties in US. This information can be further analyzed with the number of students per county actively engaged in studying STEM related areas at all levels of education (High School, Undergraduate and Graduate levels). This example assumes that we have built word embedding models from two different sources, one for the State of Virginia Expenditures and one for the tracking of STEM spending across the United States organized by products, area and purchaser in terms of county, school district, university, etc. Note that each STEM subject is now associated with the county and/or vendor across United States including the counties and school districts of the State of Virginia. Therefore, querying the STEM model one finds the spending in STEM across the US and can constraint it to the State of Virginia. With such information one can correlate the STEM spending with county and school district budgets as well as the spending per student. As Figure 6 shows, the `similarityUDF()` UDF is used to identify those transactions that contain items similar to *STEM*. 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., *STEM*). 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 historical Department of Education data can be used to determine what actions need to be taken to improve spending on STEM areas as well as where and which levels need a re-adjustment of funding to continuously improve or keep spending in such as areas. Notice that similar models can be applied to different subjects, for example if one wants to do a study on nutrition and county/school district spending and relate it to state/national spending to improve nutrition particularly for k-12 student population.

3.3 Cognitive OLAP Queries

Figure 7 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 [9].

We can carry cognitive OLAP a step further. Let's say we want to find the Vendors that have a similar average expense. First we perform a feature engineering step to find

```
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 7: Example of a cognitive OLAP (aggregation) query

```
SELECT CLUSTER_NUM, AVG(AMOUNT) as avg_value
FROM Expenses
WHERE proximityCust_NameUDF(VENDOR_NAME,'$Vendor_Y') >
$proximity
GROUP BY CLUSTER_NUM ORDER BY avg_value DESC
```

Figure 8: Example of a cognitive OLAP query with clustering

a clustering profile for the vectors of all Vendors in k clusters using for example *k-means* clustering algorithm. We add the cluster number to each Vendor in the original Expenses table. Using the query illustrated in 8 we can identify the clusters of vendors that on average have a similar transaction amount with the state and list them in descending order by the highest to the lowest average.

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 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* [16, 18]. 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 [16]. 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 $V_R = (V_Q + V_Y - V_x)$ [16]. Recently, several solutions have been proposed to solve this formulation of the analogy query [12, 13, 14]. We have implemented the 3COSMUL approach [12] which uses both the absolute distance and direction for identifying the vector V_W as

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

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.

```
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 9: Example of an analogy query

Figure 9 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 13), 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 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, $\{X, Y, Z, ..\}$, the semantic clustering process identifies a set of entities, $\{W, ..\}$, 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 10 presents a CI query which uses a semantic clustering UDF, `semclustUDF()`, to identify verndors 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 $S_w = \{W_1, W_2, .., W_i\}$ 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).

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

Figure 10: Example of a semantic clustering query

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.

4. EXPERIENCING THE AI-POWERED DATABASE

For the evaluation purpose, we will be using a Linux-based Spark Scala implementation of the AI-powered database system [3]. This system is a x86 based system with 56 cores and 512G of physical memory. We plan to demonstrate both the end-to-end features of the Spark based AI-Powered database implementation as well as novel capabilities of CI queries. The rest of the section provides a glimpse of the AI-DB system by illustrating a real database usecase.

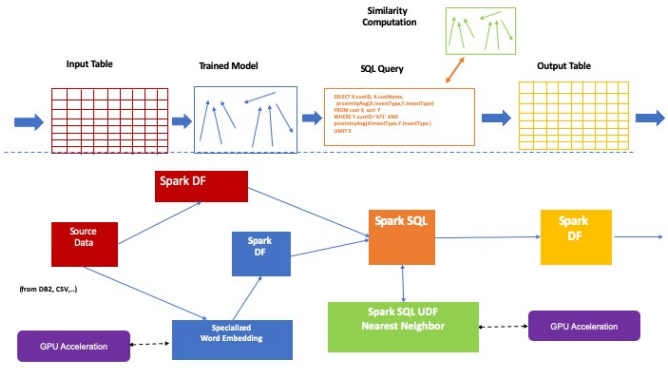


Figure 11: Flow of AI-Powered database from Input to SQL query to output

Before describing the details, the flow of the system is shown in Figure 11. The top part of the figure shows the flow from a conceptual standpoint starting from a table as an input to the training process. The model is fed into the Query system where cognitive SQL queries and traditional SQL queries are used to analyze the data and generate unique outputs. The bottom part of the figure illustrates the process from a Spark data structure standpoint. The source table is pre-processed to generate the input to the training engine and is read by the query engine as a Spark dataframe. Likewise, the output of the training engine is a vector model also read into the system as a dataframe. Both dataframes are input to the Spark SQL engine where the SQL statements are executed. The AI component is processed in the SQL engine by the use of UDFs used inside the SQL statements. The outcome of the query is a new dataframe containing the results of the type of query performed. Since both tasks, the model training and processing of UDFs using the model, are computer intensive operations they can benefit from acceleration. Multi-threading is already exploited during model training and multi-thread/distributed is exploited within the Spark engine at query time. Distributed implementations and the use of GPU acceleration for vector embedding model training [4] and nearest neighbor calculations [6] are currently being implemented and will be made available soon.

4.1 Evaluation Setup

To illustrate the AI-DB execution, we use a publicly available expenditure dataset from the State of Virginia[17]. It is a fairly large dataset spanning across 15 years of data. Since the data is not uniform across all years (more information was added as years went by and previous years data were never updated for consistency). Since the number of transactions per year is large we only used the state-wide expenditure for 2015/2016 fiscal year (at least 5.3 Million records) listing details of every transaction such as vendor name, corresponding state agency, which government fund was used etc, by 190951 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. Other important information related to this evaluation is the census and school population for the State of Virginia as summarized in Figure 12. This information is not present in the state's expenditures data but can be found in the United States Census Bureau Data website (www.census.gov). Furthermore, county population and k-12 county student population is based on web searches for the respective population types. These may vary depending on which year the estimates were done with respect to the official census of 2010.

2016 Estimates (www.census.gov)	
Population	8411808
Percentage < 18yrs	1867421 (22.2%)
High School graduation or higher	88.6%
BS degree or higher	36.9%
Civil labor force (16+)	64.5%
Poverty	11%
Counties + cities	95 + 38 = 133

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

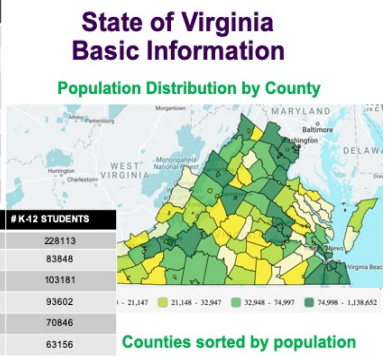


Figure 12: 2016 Census Information for the State of Virginia

The reason why we look at the data from an educational standpoint is because an analysis of the state expenditures shows that the highest state expense is in education when taking into account the cumulative expenses in both k-12 and higher education. In the fiscal year of 2015/16 15.7B dollars where spent in education out of the 46.4B dollars of

state expenditures. As such it is interesting to use CI queries to see if they can help identify counties and school districts with the most expense per student. Note that conclusions made about if the average expense per student and/or area is not the goal of this evaluation. Our objective is to demonstrate that CI SQL queries when combined with traditional SQL are an effective tool to retrieve such information.

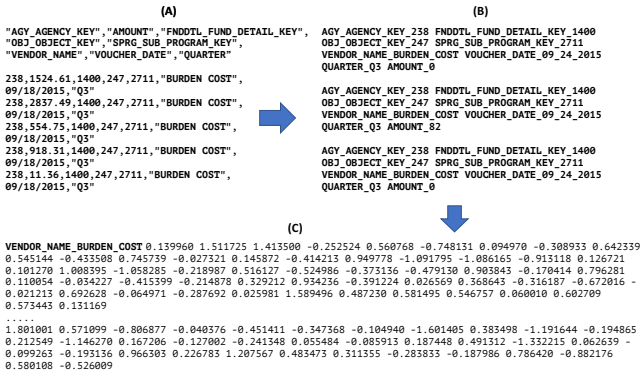


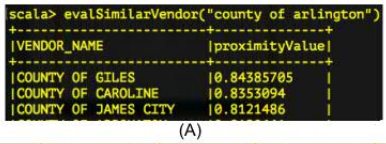
Figure 13: Pre-processing the Virginia Expenditure Dataset

Figure 13(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 13(B)). This *textification* process is implemented using Python scripts that converts a relational row into a sentence. A Spark version embedded into Spark pipelines is also available for textification and can be invoked from the Spark shell or from within a Jupyter Notebook. 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 vector embedding model (`db2Vec`). The `db2Vec` generates a d dimensional feature (meaning) vector for each unique string token in the vocabulary. For example, Figure 13(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 evaluation will go over various pre-processing stages in detail to explain the text conversion and model building scripts.

4.2 Examples of CI Queries

Once the model is built, the user program can load the vectors and use them in the SQL queries. Figure 4 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 14(A) presents the results of this query for the vendor `County of Arlington` as already shown in Figure 5. In addition Figure 14(B) explains why the transactions of top counties with the state are similar to the transactions of the `County of Arlington` with the state. Each transaction is described by several fields, i.e., `VENDOR_NAME`, `AGENCY`, `FUNDS`, `OBJECTIVES`, etc. The more in common are the values of the respective fields the more common the counties are and the higher they will rank in similarity. The Table in Figure 14(B) can be explained as follows, the `County of Arlington` had a total of 87 transactions with the state, dealing with 2 agencies, 2 funds, 2 objectives and 15 state programs (Other fields are omitted for simplicity). Similarly, the `County of Giles` had a total of 78 transactions with the state, dealing with 3 agencies, 4 funds, 3 objectives, and 15 programs. Referring to the Agencies row in the table, the next two numbers are the common agencies and the number of transactions with common agencies. As such, `County of Giles` had transactions 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 counties. 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. Note the last two items are engineered features added to extend the analysis of the data.

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



(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 14: 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 fil-

```
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 15: Example of a CI dissimilar query

tered 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 16, where it shows the counties most dissimilar to the `County of Arlington`. We know they are the most dissimilar by performing the same analysis we performed for the similar case, illustrated in Figure 14. As it was described for Figure 14(B), the table in Figure 16(B) also compares how many of the fields are common with the fields of `County of Arlington` and how many transactions are common for such fields. As illustrated in Figure 16(B) the three counties have almost nothing in common with the `County of Arlington` except for `County of Fairfax-SWMP` where of the nine objectives the 193 are grouped on, one is common with `County of Arlington` for 22 of the 193 transactions. However, even for the other fields there is no communality. Considering `Agencies` for example, the `County of Fairfax-SWMP` dealt with 11 state agencies with zero in common with the agencies dealt by the `County of Arlington` and zero transactions in common.

```
scala> evalDissimilarVendor("county of arlington")
+-----------------------------------+-------------+
|VENDOR_NAME                        |proximityValue|
+-----------------------------------+-------------+
|COUNTY OF GREENSVILLE VIRGINIA     |0.36435106   |
|COUNTY OF FAIRFAX-SWMP             |0.36581263   |
|COUNTY OF BUCKS                    |0.36723676   |
```

(A)

	County of Arlington	County of Greensville 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 16: Most dissimilar vendors to COUNTY OF ARLINGTON

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 in Figure 17, where the table is ordered in descending order by population and it includes the K-12 student population.

Figure 7 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 18. 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 17. It does not follow the sorted order from a school population standpoint.

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

```
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|134,376,167.12|
|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 18: 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 17 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 19 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 (SOQ)	186,116,668.86
VA BEACH CITY PUBLIC SCHOOL	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	135,872,302.60
CHESTERFIELD COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	132,798,416.37
LOUDOUN COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	121,667,011.02
HENRICO COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	102,053,637.31
CHESAPEAKE CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	88,741,581.47
NORFOLK CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	70,875,202.43
NEWPORT NEWS CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	69,233,452.31
STAFFORD COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	61,893,535.57
SPOTSYLVANIA COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	53,476,060.96
HAMPTON CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	48,704,198.98
RICHMOND CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	48,097,018.54
HANOVER COUNTY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	37,352,642.31
PORTSMOUTH CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	36,060,408.40
ROANOKE COUNTY TREASURER	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	36,005,193.59
ROANOKE CITY PUBLIC SCHOOLS	STANDARDS OF QUALITY FOR PUBLIC EDUCATION (SOQ)	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 19: OLAP CI query with focus on Programs instead of Agencies

To demonstrate an analogy query we use the CI query in Figure 9. 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

```
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
VA BEACH CITY PUBLIC SCHOOL	Department of Education - Direct Aid to Public Education	Q3	63,199,782.35
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	8,919,065.00
ALEXANDRIA CITY PUBLIC SCHOOLS	Department of Education - Direct Aid to Public Education	Q3	7,068,007.94
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	1,074,689.29
FAIRFAX CITY TREASURER	Department of Education - Direct Aid to Public Education	Q3	853,910.50

Figure 20: Example of Analogy CI query comparing Fairfax County in Q1 with Vendors in Q3

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 20. 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.

Table 1 presents preliminary performance results for executing different CI queries using the Spark implementation. We ran the CI queries on a single dual-socket x86 system (56 cores) using the Local Spark mode (Spark 2.4.0), and various driver memory configurations, e.g., 20, 40, 60 GB. The UDFs were developed in Scala and the queries were invoked as a Scala program. We executed several times of queries, the ones which the values are presented in this paper, from the `Spark-shell` running on the system described previously, with the caveat that we were the only users of the system. The results reported in 4.2 are based on repeating each query 5 times and averaging the runtimes by the number of runs.

CI Query	Runtime (sec)		
	20 GB	40 GB	60 GB
Similarity	57	44	35
Dissimilarity	53	45	37
Analogy	55	47	38
OLAP Agencies	54	49	35
OLAP Programs	55	44	36

Table 1: CI Query runtimes from Spark-shell (Single Node, Local Mode, Spark 2.4.0) and different driver memory configurations

We attribute the poor performance of the SQL CI queries to the following factors: (1) Lack of Spark SQL query optimizations leading to UDF being invoked for every row, (2) Cost of individual Spark UDF invocation, (3) Cost of Scala-based nearest neighbor computations, and (4) Limited degree of concurrency as the program was run on a single node (using all available threads on the 2 x86 CPUs).

We are currently exploring a variety of optimization options, specifically, partitioning the computation across multiple Spark nodes in a scalable manner, exploring opportunities for reducing UDF invocation calls, and improving the Scala nearest neighbor performance via using SIMD vector instructions or GPU accelerated kernels, and/or math libraries such as BLAS that efficiently implement dot product and vector/matrix operations.

5. CONCLUSIONS

AI-Powered Database is an innovative relational database system that uses the power of vector embedding models to enable novel AI capabilities in database systems. Our implementation of vector embedding, `db2Vec`, uses unsupervised learning to generate meaning vectors using database-derived text. These vectors capture syntactic as well as semantic characteristics of every database token. The vector embedding model is then exposed to the users via a new class of SQL based queries called *Cognitive Intelligence* (CI) queries. The CI queries enable information retrieval from relational databases by semantic context, not by entity values.

We demonstrated the unique capabilities of the AI-Powered Database on a real enterprise financial dataset. We implemented and evaluated a Spark-based implementation of our system using a variety of cognitive intelligence queries. As demonstrated in our experiments, the CI queries enable application developers to get new semantic insights from relational data using existing SQL infrastructure. The end user is not bothered with subtlies associated with exploiting the neural network models: the only interface to the word embedding model is via SQL. In addition, the new semantic components can be integrated with traditional SQL operations such as grouping or aggregation. We are currently working on applying the AI-Powered Database to a variety of domains (e.g., financial, insurance, and retail), and evaluating approaches for optimizing performance of CI queries.

6. REFERENCES

[1] Jupyter notebook: Open-source web application for creating and sharing documents, 2017.

[2] Apache Foundation. Apache spark: A fast and general engine for large-scale data processing. http://spark.apache.org, 2017. Release 2.2.

[3] R. Bordawekar. Cognitive Database: An Apache Spark-Based AI-Enabled Relational Database System. Spark+AI Summit 2018, June 2018.

[4] R. Bordawekar. GPU Acceleration of Word Embedding Models for Large Datasets. Nvidia Global Technical Conference (GTC), March 2019.

[5] R. Bordawekar, B. Bandyopadhyay, and O. Shmueli. Cognitive database: A step towards endowing relational databases with artificial intelligence capabilities. *CoRR*, abs/1712.07199, December 2017.

[6] R. Bordawekar and P. D'Souza. Optimizing Out-of-core Nearest Neighbor Problems on Multi-GPU Systems using NVLink. Nvidia Global Technical Conference (GTC), March 2017.

[7] R. Bordawekar and O. Shmueli. 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, pages 5:1–5:4, New York, NY, USA, 2017. ACM.

[8] R. Bordawekar and O. Shmueli. Exploiting latent information in relational databases via word embedding and application to degrees of disclosure. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[9] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 73–88, 1997.

[10] IBM Watson. Watson visual recognition service. www.ibm.com/watson/services/visual-recognition/, 2016.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[12] O. Levy and Y. Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning, CoNLL 2014*, pages 171–180, 2014.

[13] T. Linzen. Issues in evaluating semantic spaces using word analogies. *arXiv preprint arXiv:1606.07736*, 2016.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *27th Annual Conference on Neural Information Processing Systems 2013.*, pages 3111–3119, 2013.

[15] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543, 2014.

[16] D. E. Rumelhart and A. A. Abrahamson. A model for analogical reasoning. *Cognitive Psychology*, 5(1):1 – 28, 1973.

[17] State of Virginia. State of virginia 2016 expenditures. http://www.datapoint.apa.virginia.gov/, 2016.

[18] R. J. Sternberg and M. K. Gardner. Unities in inductive reasoning. Technical Report Technical rept. no. 18, 1 Jul-30 Sep 79, Yale University, 1979.

APPENDIX

A. USING PYTHON INTERFACES

In this section, we describe the Python implementation of AI-Powered 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 AI-Powered Database.

During demonstration, 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 AI-Powered 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 AI-Powered Database connection, they can perform CI Queries to expose powerful semantic relationships (Figure 21).

Figure 21: 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 22).

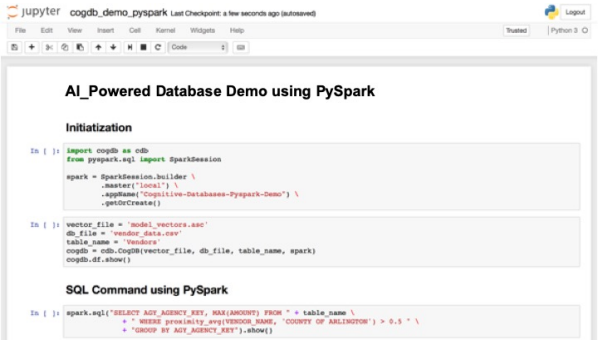


Figure 22: Example CI Queries in Jupyter Notebook using PySpark

B. JUPYTER NOTEBOOKS & SCALA

An alternative approach to Python implementations is to use the Jupyter Notebooks running Scala Language a java based programming language that unifies object-oriented and functional programming. The purpose of this system is to create a more tightly integrated solution to run in Spark. As previously described the use of AI-powered data base requires a training step where the contents of a table is textified prior to being fed into the model generation code. In Section A the pre-processing step is written in python. We migrated this processing step to Scala and merge it around the Spark ML Pipelines architecture. We created 3 new stages: `Excluder` which allows a user to enumerate the columns in a table the user does not want to include in model generation; `Prefixer` a stage to allow the user to define the character for labeling text contents. The convention used is to prefix the column contents with the column name followed by the prefixer character chosen by the designer; `Clusterer` is a stage to be used with columns with numerical values, where the ideal set of clusters is found and the numerics translated into the respective cluster labels. With these extensions, a user can easily create a Scala-based Jupyter Notebook that loads a table, does some data manipulation, such as removing rows with corrupt data, clean-up data to conform with requirements for model generation. Once the table is ready a user can define the ML Pipeline stages and just execute the `fit` and `transform` Spark methods to generate the textified version of the data ready to train the model.