Retrieval-Augmented OLAP: Generative AI Architecture for Smart Systems & Equipment

El Mehdi OUAFIQ[1, 2], Rachid SAADANE[1]
Hassania School of Public Works, Data Intelligence Delivery
elmehdiouafiq@gmail.com, rachid.saadane@gmail.com

Abstract

The integration of Large Language Models (LLMs) within a Retrieval-Augmented Generation (RAG) framework, combined with Symbolic Logic (SL), holds significant potential as a reasoning engine for complex agricultural decision-making to ensure reliability, and as a centralized repository of domain-specific agricultural knowledge. However, since the smart farming domain also deals with Data Mediated by the Process (DMP), which is predominantly structured, LLMs often struggle to generate and run SQL queries for a significant portion of these data interactions. In such cases, the RAG framework resorts to fetching structured data directly from the relevant database, which can lead to performance bottlenecks and increased resource consumption, particularly when Big Data characteristics (Volume, Variety, Velocity) are prominent and traditional Online Analytical Processing (OLAP) frameworks prove ineffective. In this paper, we propose a novel Retrieval-Augmented OLAP (RA-OLAP) framework that leverages NoSQL databases to store OLAP cubes as cuboids with pre-calculated measures, employing a dictionary-based encoding strategy. Simultaneously, the same NoSQL database is utilized for vector search operations within the RAG architecture. This hybrid AI framework addresses three critical agricultural challenges: drought classification, crop production prediction, and equipment maintenance. By integrating these components, our approach aims to enhance the scalability, efficiency, and accuracy of data-driven decision-making.

Introduction

In the field of agriculture, data sources can be broadly categorized into three types, as outlined in Table 1:

- Data Mediated by the Process (DMP): This type of data is generated during daily agricultural operations, such as recording productivity metrics, fertilizer applications, and other routine activities.
- Data Generated by Machines (DGM): This includes data collected from IoT devices, smart sensors, and other automated systems designed to monitor and measure agricultural processes.
- Data of Human Origin (DHO): This encompasses data derived from human inputs, such as farmers' experiential knowledge, consumer feedback, and other qualitative insights and human cognitive abilities.

Data	Structure	Ingestion	Data Model
DMP	Predominantly Structured	Mostly Batch Processing	Relational, Dimensional
DGM	Typically Semi-Structured	Mostly Real-Time Processing	NoSQL, Dimensional
DHO	Often Unstructured	Mostly one-time-Load	Flat, Dimensional

Table 1: Data source nature in Smart Farming

A robust and reliable smart decision-making system must be capable of efficiently processing, storing, and retrieving actionable insights from all three types of data sources [1][2].

Historically, the industry has relied on the Online Transaction Processing (OLTP) data model to manage transactional data through a normalization approach. However, OLTP systems are inherently incompatible with advanced data analytics due to their focus on transactional efficiency rather than analytical flexibility. To address this limitation, the OLAP data model was introduced, which transforms normalized data into fact and dimension schemas via the ETL (Extract, Transform, Load) process. While OLAP facilitates multidimensional analysis, it remains highly structured and is not directly compatible with modern Generative AI systems, particularly those based on Large Language Models (LLMs) [3][4].

LLMs, on the other hand, primarily rely on vector databases, which are optimized for semantic search and natural language processing (NLP) but lack the capability to support seamless OLAP processing [34][35]. This becomes particularly problematic when dealing with Key Performance Indicators (KPIs), which often require complex aggregations across multiple dimensions (primarily from DMPs and DHOs) and measures (primarily from DGMs). Additionally, vector databases face significant challenges in updating or removing specific data points, as these operations can disrupt the integrity of the embedded vector representations [4][5].

This disconnect between structured OLAP systems and unstructured LLM-based systems highlights a critical gap in the current technological landscape for smart agriculture. To address these challenges, we propose a novel framework that leverages the Reduce by Key and Flat Map functions within the Apache Spark framework for the creation of OLAP cuboids. This approach utilizes column-oriented distributed storage to store pre-calculated cube measures in a key-value format, ensuring efficient random read operations while simultaneously supporting vector search capabilities. By integrating these techniques, our framework bridges the gap between structured OLAP processing and unstructured data retrieval, enabling seamless interaction between the two paradigms.

The proposed framework combines symbolic reasoning implemented through expert system rules with machine learning models and RAG to retrieve domain-specific agricultural knowledge and generate explainable insights. The symbolic reasoning component enhances decision support by validating predictions against expert-defined rules and known thresholds, ensuring consistency and reliability. Meanwhile, the RAG mechanism dynamically retrieves relevant knowledge to refine predictions and provide context-aware recommendations. As described in Figure 1, the RA-OLAP architecture leverages a structured pipeline where the Front-end handles query processing, the Back-end integrates AI reasoning with OLAP analytics, and the Memory Subsystem optimizes retrieval and caching to ensure effective Spark lazy evaluation and timely decision-making [6].

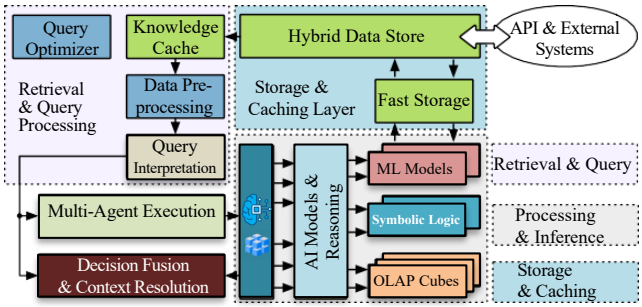


Figure 1: RA-OLAP Computational Workflow Architecture: Integrating RAG, Symbolic Logic, and OLAP for Hybrid AI Reasoning

Additionally, our framework incorporates breakdown detection model to predict maintenance needs for agricultural equipment using data from IoT sensors. This proactive approach minimizes downtime and optimizes resource utilization, further enhancing the efficiency of agricultural operations.

Experimental results demonstrate the effectiveness of our RA-OLAP framework in improving the precision and reliability of agricultural data analysis. The framework provides actionable, interpretable insights for farmers, empowering them to make informed decisions. By combining the strengths of symbolic reasoning, machine learning, and modern retrieval systems with OLAP, this work lays the foundation for future research in the agricultural sector. It opens new avenues for developing hybrid AI systems that integrate structured and unstructured data processing, ultimately advancing the field of smart agriculture.

Related Works

LLMs with Structured Data and OLAP Queries

LLMs like GPT-3 have demonstrated proficiency in NLP tasks; however, their effectiveness diminishes when managing structured data and executing OLAP queries. For instance, Chen et al. (2021) observed that LLMs struggle with tasks requiring precise data manipulation, such as generating SQL queries for complex databases, leading to inaccuracies in data retrieval. Similarly, Shi et al. (2022) highlighted that while RAG systems enhance LLMs by incorporating external knowledge, they often falter in scenarios demanding intricate data aggregations typical of OLAP operations. These limitations underscore the necessity for integrating more structured data handling capabilities into LLMs and RAG frameworks.

To mitigate these challenges, researchers like Zhang et al. (2023) proposed a system combining LLMs with traditional database management techniques to improve the accuracy of structured data queries. Despite these advancements, the integration remains imperfect, with issues in scalability and real-time data processing still prevalent [4].

OLAP Limitations in Distributed Systems

Traditional OLAP systems are designed for interactive analysis of multidimensional data but encounter scalability issues in distributed (big data) environments. Kim and Park (2020) demonstrated that as data volumes expand, maintaining performance and ensuring real-time analytics become increasingly challenging due to the inherent limitations of centralized OLAP architectures. Li et al. (2021) further noted that distributed OLAP architectures, while addressing some scalability concerns, introduce complexities related to data consistency and system latency [7][8].

Moreover, García-Molina et al. (2022) investigated the application of parallel processing techniques in OLAP systems to enhance performance [9]. Their findings indicate that although parallelism can alleviate some bottlenecks, it

also leads to increased system complexity and synchronization issues in heterogeneous computing environments.

Symbolic Logic with LLMs and RAG Systems

The integration of Symbolic Logic (SL) with neural approaches has led to the emergence of neuro-symbolic AI, aiming to combine the strengths of both paradigms. Besold et al. (2017) discussed the potential of neuro-symbolic systems to perform logical reasoning and handle structured data more effectively. Yang et al. (2020) proposed a framework that integrates SL with LLMs to enhance explainability and reasoning capabilities, addressing some limitations of purely neural models [10][11].

However, challenges persist in seamlessly combining these approaches. Sarker et al. (2021) highlighted difficulties in aligning the probabilistic nature of neural networks with the deterministic characteristics of symbolic logic systems, which can lead to inconsistencies and reduced system robustness [12].

Methodology

In this section, we address the challenge of developing OLAP cubes in a distributed system while ensuring real-time processing capabilities. Additionally, we explore leveraging the cuboids-based model for structured data within a RAG framework and integrating vector search to enhance support for LLMs.

Background and Motivation

RAG systems enhance LLM reasoning by integrating retrieval-based augmentation. RAG can be implemented using either distributed systems (e.g., Spark, Databricks, Ray) based on fundamental concepts of mapping, shuffling and compute results or serverless architectures (e.g., AWS Lambda, Azure Functions), each with distinct advantages. In distributed RAG, document embeddings $D = \{d_i \in R^d\}_{i=1}^{N}$ are stored across multiple nodes, with query embeddings $q \in R^d$ retrieved using cosine similarity:

$$\text{sim}(q, d_i) = \frac{q \cdot d_i}{\| q \| \| d_i \|}$$

The dataset is sharded across worker nodes W_j, reducing query latency to $O(\log N)$ via Approximate Nearest Neighbor (ANN) search or locality-sensitive hashing (LSH), making it optimal for high-throughput, multi-terabyte RAG systems. The Parallelized retrieval is formulated as follows: $D = \bigcup_{i=1}^{k} D_j$,where $D_j \cap D_{j'} = \varnothing, \forall j \neq j'$ where k is the number of partitions. LLM inference is distributed using tensor parallelism, reducing processing time per token:

$$t_{\text{LLM}} = \frac{T}{k}, \text{where } T \text{ is sequence length}$$

However, it requires cluster management, leading to higher infrastructure costs [4][16][17].

In contrast, serverless RAG offloads retrieval to managed vector databases (e.g., Pinecone, OpenSearch), introducing additional network latency: $t_{\text{retrieval}} = t_{\text{query}} + t_{\text{API}}$ where t_{query} is the embedding similarity lookup time, and t_{API} includes network overhead. Unlike distributed RAG, where batch queries reduce overhead, serverless RAG performs individual queries, which increases cumulative latency for multiple requests [18][19][20].

In the LLM generation step, both approaches use an autoregressive model f (e.g., GPT-4, Llama 2), where the probability of generating a token sequence $y_{1:T}$ given the query and retrieved context C is computed as: $P(y_{1:T} \mid q, C) = \prod_{t=1}^{T} P\left(y_t \mid y_{1:t-1}, q, C; \theta\right)$ where θ represents model parameters. In distributed RAG, this process is parallelized across GPU instances using tensor parallelism, reducing inference latency $O(T) \rightarrow O(T/k)$ with k GPU shards. In contrast, serverless RAG relies on external API inference, adding additional latency $t_{\text{API-infer}}$ per request: $t_{\text{total}} = t_{\text{retrieval}} + t_{\text{API-infer}}$ which can be costly due to per-query API pricing in cloud services like OpenAI API, AWS Bedrock, and Azure OpenAI.

Thus, distributed RAG is optimized for large-scale retrieval and parallelized inference with reduced latency $O(\log N)$ for retrieval and $O(T/k)$ for LLM generation. Serverless architectures eliminate infrastructure management and scale efficiently for low-volume workloads but suffer from higher per-query cost and increased latency due to API dependencies, and is best suited for on-demand inference rather than batch processing. A hybrid approach optimizes cost and efficiency by precomputing embeddings in a distributed system and storing them in a managed vector database for real-time serverless retrieval. This minimizes: $t_{\text{retrieval}} = O(\log N), t_{\text{inference}} = O(T/k)$ providing an optimal balance for enterprise-scale applications.

Problem Formulation

OLAP in Distributed Systems. Most distributed systems, especially those designed for large-scale data processing follow a structured execution model comprising three key phases: mapping (data partitioning and parallel task distribution), shuffling (data redistribution across nodes based on keys), and aggregation (final computation and result merging). This model is fundamental in distributed frameworks such as MapReduce (Dean and Ghemawat 2004), Apache Spark (Zaharia et al. 2010), and distributed OLAP engines (Gon-zalez et al. 2014), enabling efficient, fault-tolerant analytics across multiple nodes [21][22][23].

However, OLAP workloads in distributed environments face efficiency constraints due to inter-node coordination overhead. Given an analytical query Q operating on a dataset D distributed across N nodes, the execution time can be approximated as:

$$T(Q) = T_m + T_s + T_a$$

where:

- T_m is the mapping phase time, modeled as:

$$T_m = \max_{i \in N} f(D_i)$$

  where $f(D_i)$ is the computation time for each partition D_i.
- T_s is the shuffling phase time, which is proportional to the data size $| D |$ and the network bandwidth B:

$$T_s \approx \frac{| D |}{B}$$

- T_a is the aggregation phase time, dependent on the function complexity g and the number of nodes N:

$$T_a = O\big(g(N)\big)$$

This decomposition shows that shuffling introduces significant overhead, especially for OLAP queries involving group-by, join, and roll-up operations, which require extensive data redistribution across nodes (Chen et al. 2012). Moreover, in highly distributed OLAP settings, the query response time follows:

$$T_{OLAP} \approx O\left(\frac{| D |}{B} + g(N)\right)$$

Indicating that as the dataset grows, network transfer (shuffling) and aggregation complexity become dominant bottlenecks (Abadi et al. 2009). Figure 2 highlights an example of the fundamental concept of parallel computing in distributed systems, revealing potential inefficiencies that can arise when applied to OLAP workloads. To mitigate these inefficiencies, hybrid architectures integrating precomputed OLAP cubes, adaptive caching, and query optimization techniques (Stonebraker et al. 2013) can minimize redundant data movement and reduce coordination costs, leading to more scalable OLAP workloads in distributed environments [24][25][26].

However, this proposition does not seamlessly transfer to NoSQL systems like Cassandra, which use a wide-column model optimized for high-ingestion rates rather than analytical queries. A key limitation is the shuffling and aggregation overhead inherent in distributed NoSQL architectures, where cross-node data movement significantly impacts query performance. Given an aggregation query Q over a distributed dataset D, the shuffle cost can be approximated as O(n * p), where n is the number of partitions involved, and p is the average partitions scanned per query. Unlike columnar OLAP engines that minimize I/O through compressed columnar scans, Cassandra executes queries via partition-range scans, leading to higher data retrieval costs when performing multi-dimensional roll-ups. Addressing these inefficiencies, an OLAP cuboid model in Cassandra could precompute hierarchical aggregations, thereby reducing shuffle operations and enabling sublinear query execution for structured analytical workloads [27][28][29].

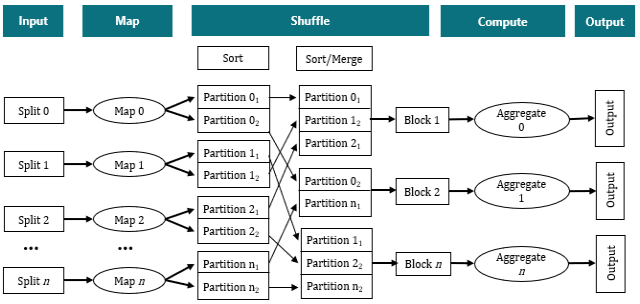


Figure 2: Example of fundamental concept of Parallel computing process in a distributed system

Reduce by Key and Flat Map for Cubes Creation

Cubes creation workflow. We propose utilizing the Apache Spark framework to construct OLAP cubes, ensuring efficient in-memory calculations and massive parallel processing (MPP) [30], while leveraging Cassandra for the storage of cuboids, providing interval-based lookups through row key prefixes. As depicted in Figure 3, the process begins by selecting the relevant dimensions and measures for the OLAP cube from the centralized repository (in our case, a Data Vault-based model within the structure layer) and integrating them into a flat table. The second step involves performing all necessary aggregations as defined in the configuration table, based on the data model, and storing the results in an aggregated table. The final step entails converting the aggregated tables into key-value format, transforming them into Sorted String Tables (SSTables) [31], and loading them into Cassandra via a bulk load process, which, according to our tests, performs eight times faster than traditional methods.

Optimized construction of OLAP aggregate table. The construction of the OLAP aggregate table in Apache Spark follows a hierarchical approach based on the generation of cuboids, reducing dimensionality at each step. As illustrated in the Figure 4, given an initial cuboid with N dimensions, the process is structured as follows:

1. Initial Grouping and Aggregation: The first transformation applies ReduceByKey, where the key K is represented by the N dimensions, and the value V corresponds to the aggregated measure. Formally:

$$K = (d_1, d_2, \ldots, d_N), V = f(m_1, m_2, \ldots, m_k)$$

where d_i are the dimensions, and m_i are the measures being aggregated (sum, count, average, etc.). This operation is performed in-memory due to Spark's lazy evaluation mechanism (Zaharia et al., 2012).

2. Flattening and Parent-Child DataFrame Derivation: After aggregation, a FlatMap operation is applied to decompose the parent DataFrame into multiple child DataFrames with $N-1$ dimensions. If C_N represents the parent cuboid at level N, then its derived cuboids C_{N-1} are given by:

$$C_{N-1} = \bigcup_{i=1}^{N} \pi_{-d_i}(C_N)$$

where π_{-d_i} denotes the projection operation that removes dimension d_i. This step exploits Spark's in-memory processing to avoid unnecessary disk I/O, reducing computational overhead.

3. Execution and Persistence in HDFS: Spark's lazy evaluation ensures that transformations are only computed when an action (e.g., write()) is triggered. Until this moment, intermediate computations remain in-memory. The parent DataFrame is only written to HDFS at the final stage to release memory and enable subsequent processing steps (Armbrust et al., 2015). This iterative process continues until reaching the apex cuboid or 0D cuboid, which corresponds to the fully aggregated measure:

$C_0 = \text{Global Aggregate} = f(V)$

where C_0 represents the total aggregation over all dimensions, meaning the group-by query is empty.

This hierarchical OLAP cuboid generation model in Spark provides a scalable and memory-efficient approach for multidimensional aggregation, addressing traditional shuffle and aggregation bottlenecks in large-scale data processing (Stonebraker et al., 2013).

Encoding based Conversion to Key Value Format

The encoding process aims to leverage hashing and interval-based lookups in Cassandra by efficiently constructing row keys. Each OLAP cube maintains an encoding dictionary, where each dimension in the pre-joined table is assigned an encoded value. This encoding facilitates direct row access during queries, avoiding costly full-table scans.

Encoding Process and Dimension Representation. Given a cube with D dimensions and M measures, each dimension $d_i \in D$ is encoded using a binary representation. The encoding dictionary E maps each dimension value v_j to a unique integer:

$$E: v_j \rightarrow e_j, \text{ where } e_j \in Z^+$$

For example, in the Figure 5, the dictionary assigns:

$$E(\text{Y1}) = 0, E(\text{Y2}) = 1, E(\text{X1}) = 0, E(\text{X2}) = 1$$

Each cuboid $C_k \subseteq D$ (a subset of dimensions used in aggregation) is identified by a bit vector of size D. The presence of a dimension is denoted as 1, while its absence is 0 (i.e., dimensions excluded from the GROUP BY clause). Given two dimensions, the bit vector B for a cuboid C_k is:

$$B(C_k) = (b_1, b_2, \ldots, b_D), b_i \in \{0,1\}$$

For example, if C_k contains both DY and DX, then $B(C_k) = (1,1)$, while if it contains only DY, then $B(C_k) = (1,0)$.

Row Key Generation in Cassandra. The Cassandra row key is constructed by concatenating the bit vector $B(C_k)$ with the encoded dimension values e_j from E. Formally, the row key $R(C_k)$ is:

$$R(C_k) = \text{concat}\big(B(C_k), E(d_1), \text{E}(d_2), \ldots, E(d_m)\big), \text{ where } d_i \in C_k$$

For instance, in the Figure 5:

- The row key 00000000 corresponds to the fully aggregated cube (*, *) with a total sum of 100.
- The row key 00000001+0 represents dimension DY = Y1, summing over all values of DX, with a sum of 80.
- The row key 00000011+01 encodes DY = Y2, DX = X2, storing the sum 20.

Efficient OLAP Query Processing in Cassandra. This row key encoding approach enables direct access to the required records in Cassandra via prefix-based row key scans rather than exhaustive table scans. Specifically:

- Queries on higher-dimensional aggregations (e.g., GROUP BY DY) leverage prefix scans over the row keys.
- The sequence of dimensions in the aggregated table ensures that query predicates maintain the same order, optimizing access patterns.

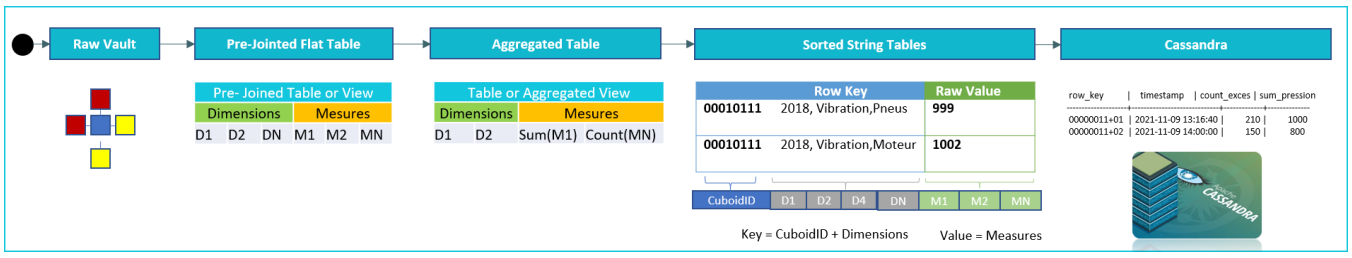


Figure 3: Cubes development methodology: Spark-Based Cuboid Creation and Cassandra for Data Storage

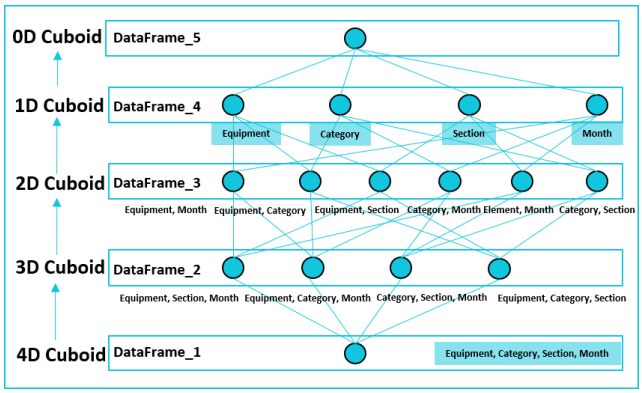


Figure 4: Construction of the Aggregate Table

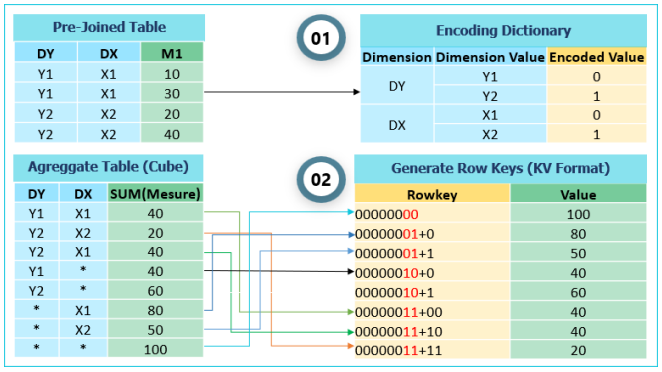


Figure 5: Encoding Flat Tables

Reverse Engineering Retrieval of OLAP Queries

Upon receiving a user query, it undergoes a transformation into an optimized execution plan. OLAP queries predominantly consist of column projections, relational table joins, predicate filters, and aggregation functions. The proposed approach achieves computational efficiency by precomputing aggregations, eliminating the need for runtime joins, filters, or aggregation operations. This optimization is enabled by the pre-materialization of cuboids within the data cube, ensuring that query execution is reduced to direct key-based lookups, thereby minimizing computational complexity and I/O overhead.

Identify Cassandra Cuboids. The first step in query execution (Algorithm 1) involves identifying the target cuboid that corresponds to the dimensions present in the SELECT and GROUP BY clauses of the user query. This is achieved through a reverse engineering process, where the query structure is analyzed to derive the corresponding Cuboid ID using the same encoding steps. Additionally, filtering conditions specified in the WHERE clause are mapped to their corresponding encoded values from a dictionary.

Algorithm 1: Identify Cuboid ID from Query (Cassandra-Based)

procedure IDENTIFYCUBOIDID(dimensions, encoding-dict, df)
  Define *dim.position.map* ← {'DY': 'Y1', 'DX': 'X1',, 'farm': '0', 'farmer': '0'}
  cuboid_id ← concatenate [dim.position.map[dim] for each dim in dimensions]
  cuboid_id ← cuboid_id.zfill(8) ▷ Ensure 8-bit Cuboid ID
  encoded.values ← [ ]
  for each *dim* **in** *dimensions* **do**
    dim_name ← dim.capitalize()
    dim_values ← encoding dict[df][filter where dimension_name = dim_name]
  Append dim_values['encoded.value'].unique().tolist() to *encoded.values*
  return *cuboid_id*, *encoded.values*

Define the Scan Range. The Define Scan Range from Query step (Algorithm 2) is designed to efficiently translate user query conditions into a scan range within Cassandra, specifically targeting row keys generated through encoding. First, the query's filter values are mapped to their encoded values based on the encoding dictionary, ensuring that the exact key range for the cuboid (identified by cuboid ID) is scanned. The start and end of the scan range are defined by appending the encoded dimension values to the cuboid ID. This selective key construction allows for precise and efficient querying of the required data, thus preventing a full table scan. This range scan technique is not only essential for improving query performance in large datasets but also provides substantial efficiency during ingestion, ensuring that the generated keys align with Cassandra's partitioning and clustering keys for optimized storage and retrieval.

Algorithm 2: Define Scan Range from Query

Procedure DEFINESCANRANGEFROMQUERY (filters, cuboid_id, encoding_dict)
  Initialize *encoding.map* ← encoding.dict set as index on ('dimension.name','encoded.value')['encoded.value'].to.dict()
  start, end ← cuboid_id, cuboid_id {Initialize start and end with Cuboid ID}
  for each *dim*, *dim_filters* **in** filters.items() **do**
    dim.encoded ← encoding.map[(dim.capitalize(), str(dim_filters['value']))]
    start ← start + '+' + dim.encoded
    end ← end + '+' + dim.encoded
  end for
  return *start*, *end*

Database Scan. In this step (refer to Algorithm 3), we use interval-based lookups through row key prefixes. NoSQL databases like HBase support range scans and fuzzy row scans, which allow flexible and approximate searches within a dataset [32]. However, in Cassandra, we achieve the same filtering effect by leveraging efficient row key prefix searches within a defined scan range. The adapted algorithm ensures that only relevant data is retrieved by filtering based

on row_key >= scan_start AND row_key <= scan_end, effectively replacing HBase's scan and fuzzy row filtering with Cassandra's partitioned range queries. This enables optimized query execution, minimizes unnecessary data retrieval, and maintains performance efficiency in OLAP workloads.

Algorithm 3: Cassandra Results Retrieval

```
Procedure CASSANDRARESULTS (cassandra.data, scan.start, scan.end)
    Initialize filtered.results ← { }
    for each kv in cassandra.data do
        if kv['row.key'] ≥ scan.start AND kv['row.key'] ≤ scan.end then
            Append kv to filtered.results
        end if
    end for
    return pd.DataFrame(filtered.results) {Convert results to DataFrame}
```

Decoding Outputs. When dealing with data stored in Cassandra, the row key often encodes multiple dimensions, while the row value contains corresponding measures. To facilitate analysis and querying, it is necessary to decode the row key into meaningful dimensions and extract the relevant measures from the row value. This step ensures that the structured information is properly reconstructed from the storage format, allowing for effective data processing and visualization. Thus, the Algorithm 4 decodes the stored Cassandra data by parsing the row key and combining it with extracted measures into a structured dataset.

Algorithm 4: Decode Encoded Cassandra Results

```
Procedure DECODERESULTS (cassandra_results, encoding_dict)
    Initialize decoded_results ← { }
    for each (row_key, row_value) in cassandra_results.items() do
        dimensions ← [encoding_dict[row_key[2:4]], encoding_dict[row_key[4:]]]
        Append dimensions + list(row_value.values()) to decoded_results
    end for
    return decoded_results
```

Experiments

Experimental Setups

Datasets. For drought management, we utilized data from an official government website, which includes crop statistics (such as name, season, year, zone, production, and productivity), drought declarations (with associated year and month), and hub device data from weather stations (containing geography, precipitation, temperature, and atmospheric pressure) [33]. For preventive maintenance, we employed data from phosphate extraction machines provided by Data Intelligence Delivery. This data encompasses measurement trends (e.g., vibration, tension, temperature) captured by sensors, along with machine, section, intervention demand, and order information loaded from an Oracle 11G.

Use Cases. For drought management, we implemented a two-stage classification approach. The first stage employs an Iterative Dichotomiser 3 (ID3) classifier, which uses inputs such as precipitation, pressure, and temperature to classify drought status into categories of low, medium, or high. The second stage utilizes another ID3 classifier, incorporating soil statistics, fertilizer data, and the results from the first classifier to predict crop growth and identify the most suitable crops for each district. For preventive maintenance, we applied an Artificial Neural Network (ANN) to analyze machine behavior, predict the Mean Time Between Failures (MTBF), and generate notifications for timely interventions.

RAG Flow. The preventive maintenance flow consists of three key components: (1) an Artificial Neural Network (ANN) regression model, (2) a Large Language Model (LLM) for interaction, and (3) Symbolic AI for explainable decision-making. The ANN model predicts machine failure risk based on sensor data such as temperature, vibration, voltage, and time since last maintenance, achieving an R^2 score of 0.89.

The LLM (GPT-4) translates natural language queries into optimized OLAP queries for efficient data retrieval from pre-calculated OLAP cubes, e.g.:

- Input in Natural Language Query: "What's the failure risk for Section JFCIII's water pumps this week?"
- Context from OLAP: Aggregated sensor trends, failure probabilities, historical maintenance.
- OLAP Query (linear algebra representation):

$$\gamma\ \text{Section, Machine\_Type, AVG(Failure\_Probability)}(\sigma \text{Section}='\text{JFCIII}' \wedge \text{Machine\_Type} = '\text{Water Pump}'(\text{Machine\_Failure}))$$

- Natural Language Explanation: "3 water pumps in Section JFCIII show a high failure risk due to rising temperature and frequent voltage drops."

The structure data will be retrieved from the OLAP cube using vector search in Cassandra 5.0 [34]. Finally, the Symbolic AI system applies expert-defined rules on ANN predictions and retrieved data, validating predictions and providing clear, rule-based explanations for maintenance decisions. Rules example:

Method	CPU%	Memory	Execution	Concurrent Users	Throughput
Hive Star Schema (1M Records)	<80%	0.95GB	300s	5 users: AVG. 360s	5 users: 0.0139s
				10 users: AVG 420s	10 users: 0.0238s
				20 users: 480s	20 users: 0.0417s
RA-OLAP (1M Records)	50%<	47.68MB	250s	5 users: AVG. 300s	5 users: 0.0167s
				10 users: AVG 350s	10 users: 0.0286s
				20 users: AVG. 400 s	20 users: 0.05s

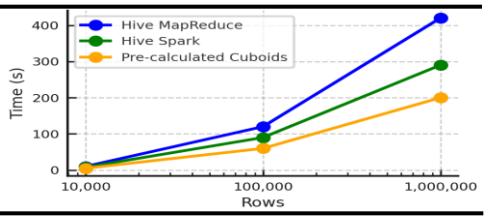


Table 2, Figure 6: Performance Analysis between RA-OLAP and standard Star Schema in Hive (Spark & Hadoop Engines)

- IF Temperature > 80°C AND Vibration > 45Hz THEN Risk = High
- IF Voltage < 200V THEN Risk = Medium
- IF Maintenance History > 100 days THEN Risk = High

Input Example: Temperature: 85°C; Vibration: 50Hz; Voltage: 210V; Maintenance History: 120 days; Failure Probability: 0.87.

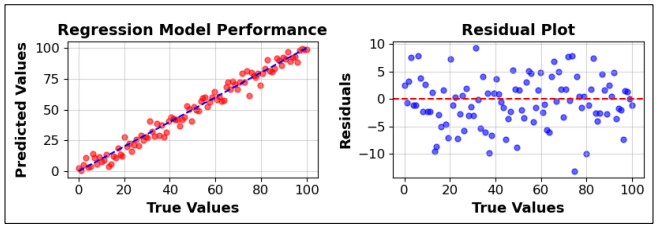
Output Example: "Machine at high risk: Temperature exceeds safe limits, high vibration detected, and long time since last maintenance."

This framework ensures high-performance, interpretable, and context-aware maintenance recommendations.

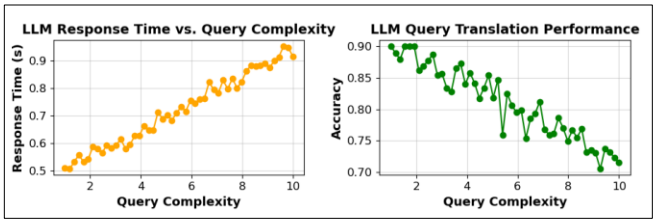
Experimental Results

Evaluation Metrics. To assess the performance of our AI/ML models, we employ metrics such as accuracy, R2R^2R2 Score, Residual analysis, Rule Coverage Distribution, computational efficiency, etc.

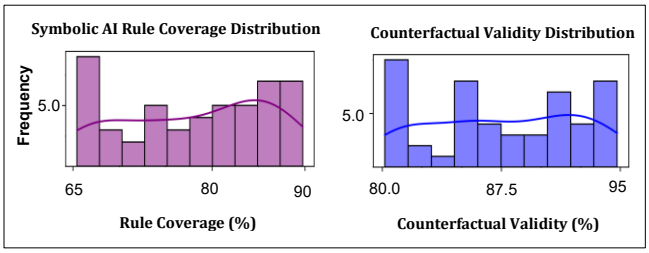
The Figure 7 (below) illustrates the regression model's prediction vs. ground truth scatter plot, which demonstrates a strong correlation between predicted and actual machine failure risks, with minor deviations indicating inherent model uncertainty, and confirms the model's reliability, as residuals are symmetrically distributed around zero, suggesting minimal systematic bias.



The Figure 8 (below) shows that LLM's accuracy is >80% even for complex queries, while response time increases non-linearly, reflecting computational complexity growth.



The Figure 9 (below) indicates that most expert-defined rules apply in 70–90% of cases, ensuring broad generalizability, and it demonstrates that symbolic reasoning aligns with ANN predictions in over 80% of cases, validating its role in enhancing explainability.



RA-OLAP Performance. As the dataset size increases from 10,000 rows to 1,000,000 rows, the execution time grows linearly, with Hive MapReduce exhibiting the highest execution time, followed by Hive Spark, and Pre-calculated Cuboids demonstrating the lowest execution time. Pre-calculated cuboids demonstrated a reduction of ~44% in CPU consumption and ~62.5% in memory usage. Also, the standard approach exhibits linear time complexity (O(n)), with execution time increasing from 2s to 20s as measures grow from 1 to 10. In contrast, the pre-computed approach maintains constant time complexity (O(1)), with execution time consistently at 0.5s, regardless of the number of measures.

Conclusion

We propose a framework dedicated to leveraging pre-calculated OLAP cubes stored in a NoSQL database to enable fast, distributed analytical processing and efficient data retrieval for AI-driven decision support systems. The experimental results unequivocally demonstrate the capability of RA-OLAP in minimizing query execution time and reducing the complexity of structured data retrieval, with advantageous performance in predicting drought status and crop productivity, as well as analyzing equipment behavior and determining breakdown time. The results show superior efficiency in query throughput, execution time, and scalability compared to traditional OLAP approaches, making RA-OLAP a robust solution for distributed AI architectures in agricultural analytics.

References

1. El Mehdi OUAFIQ, A. Data Management and Integration of Low Power Consumption Embedded Devices IoT for Transforming Smart Agriculture into Actionable Knowledge. MDPI Agriculture 2022.
2. Sjaak Wolfert, Big Data in Smart Farming – A review, Agricultural Systems, Volume 153, 2017.
3. El Mehdi. Ouafiq, IoT in Smart Farming Analytics, Big Data Based Architecture, Springer, 2020.
4. Younghun Lee, Learning to Reduce: Optimal Representations of Structured Data in Prompting Large Language Models, 2024.
5. El Mehdi Ouafiq, AI-based modeling and data-driven evaluation for smart farming-oriented big data architecture using IoT with energy harvesting capabilities, Elsevier, Volume 52, 2022.
6. Chen Bai, Towards Automated RISC-V Microarchitecture Design with Reinforcement Learning.
7. Kim, J., & Park, S. (2020). Scalable OLAP query processing in distributed data warehouse systems.
8. Li, H. (2021). Distributed OLAP query processing: A survey. Data Science and Engineering.
9. García-Molina, H., (2022). Database Systems: The Complete Book (3rd ed.). Pearson.
10. Besold, T. R. (2017). Neural-Symbolic Learning and Reasoning: Contributions and Challenges. In Proceedings of the AAAI (Vol. 31, No. 1).
11. Chen X., Gan Y. and Purver M. "Exploring underexplored limitations of cross-domain text-to-sql generalization", 2021.
12. Sarker, M. K., Yang, F., & Yao, Y. (2021). Neuro-symbolic AI: Current trends. 2105.05330.
13. Kamilaris, A. (2018). A review of the use of convolutional neural networks in agriculture.
14. Liakos, K. G. (2018). Machine learning in agriculture: A review. Sensors, 18(8), 2674.
15. Mohanty, S. P. (2020). Using deep learning for image-based plant disease detection.
16. El Mehdi OUAFIQ. Data Lake Conception for Smart Farming: A Data Migration Strategy for Big Data Analytics. Springer.
17. El Mehdi. Ouafiq, Data Architecture and Big Data Analytics in Smart Cities, (KES 2022), Springer.
18. Yao Fu, Serverless LLM: Low-Latency Serverless Inference for Large Language Models. 2024.
19. El Mehdi OUAFIQ, "6G Enabled Smart Environments and Sustainable Cities: an Intelligent Big Data Architecture" 2022, IEEE, 2022, pp. 1-5.
20. Dean, J. MapReduce: Simplified Data Processing on Large Clusters. (OSDI). 2004.
21. Zaharia. Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing. NSDI. 2010.
22. Gonzalez, J. E. GraphX: Graph Processing in a Distributed Dataflow Framework. (OSDI). 2014.
23. Chen, J. The Design and Implementation of a Distributed OLAP System. 2012. VLDB Endowment.
24. Abadi, D. J. Column-Stores vs. Row-Stores: How Different Are They Really? ACM SIGMOD 2009.
25. Stonebraker, M. MapReduce and Parallel DBMSs: Friends or Foes? ACM, 2010.
26. El Mehdi. Ouafiq, Data-Driven Agriculture and Role of Artificial Intelligence in Smart Farming.
27. Armbrust, M. Spark SQL: Relational Data Processing in Spark. SIGMOD. 2015.
28. Stonebraker, M. C-Store: A Column-oriented DBMS. 2013.
29. Zhengyu Yang, Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks, 2018.
30. Avinash Lakshman, Cassandra - A Decentralized Structured Storage System. 2009.
31. Welcome to Apache Kylin: Analytical Data Warehouse for Big Data: https://kylin.apache.org/docs/4.0.4 /overview; 12/03/2025.
32. (1) https://data.gov.in/catalog/district-wise-season-wise-crop-production-statistics; (2) http://krishi.maharashtra.gov.in; (3) www.timeanddate.com
33. Vector search concepts: https://cassandra.apache.org/doc/latest/cassandra/vector-search/concepts.html ; 12/03/2025.
34. Z. Hong, Z. Yuan, Q. Zhang, H. Chen, J. Dong, F. Huang, and X. Huang, "Next-generation database interfaces: A survey of llm-based text-to-sql," arXiv preprint arXiv:2406.08426, 2024.
35. T. Yu, R. Zhang, H. Y. Er, S. Li, E. Xue, B. Pang, X. V. Lin, Y. C. Tan, T. Shi, Z. Li et al., "Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases", 2019.