A Declarative, Recursive SQL Framework for Composable Machine Learning Ensembles

Jason Arnold
Knut Stolze
Dan Zollers
Poojan Khanpara
{jarnold,stolze,dzollers,pkhanpara}@ocient.com
Ocient, Inc.
Chicago, IL, USA

Abstract

Ensemble machine learning methods like bagging, boosting, and stacking are established techniques for improving model accuracy. However, applying them to hyperscale datasets stored in data warehouses typically requires exporting data to external machine learning frameworks—a workflow that presents accessibility challenges for SQL-native users. Furthermore, existing in-database ML systems often restrict users to pre-packaged homogeneous ensembles (e.g., Random Forest), which may limit accuracy when the optimal inductive bias is not tree-based.

We present a declarative, SQL-native framework for constructing generic and recursive ensemble models directly within the Ocient Hyperscale Data Warehouse. By extending the `CREATE MLMODEL` SQL statement with three new ensemble types, `BAGGING`, `BOOSTING`, and `STACKING`, users can define multi-level ensemble pipelines in a single SQL statement. The framework supports heterogeneous base learners (mixing any of Ocient's 20+ model types), recursive composition (ensembles of ensembles), and a weighted resampling strategy that enables gradient boosting with any regression model.

We evaluate the utility of this flexibility on a price prediction dataset where a custom heterogeneous ensemble (a 50/50 mix of Linear Regression and Decision Trees) outperforms single-model baselines, reducing error by approximately 41%. Our approach reduces approximately 120 lines of imperative PySpark code to a single 14-line declarative SQL statement, making ensemble techniques accessible without leaving the database.

Keywords

Ensemble Learning, Hyperscale In-Database Machine Learning, Declarative SQL

1 Introduction

Ensemble learning methods, such as bagging [4], boosting [6], and stacking [18], are widely used in machine learning to improve performance by combining the predictions of multiple base models. Implementing these techniques at hyperscale—on datasets spanning trillions of rows—presents practical and architectural challenges. A common paradigm involves extracting datasets from a data warehouse, transferring them across the network (ETL), and loading them into a separate compute environment like Apache Spark and its machine learning library MLlib [10, 20] for model training. This process separates the data from the compute environment. Setting up, tuning, and maintaining a distributed Spark cluster requires engineering resources. Consequently, practitioners often utilize single-node Python solutions (e.g., Scikit-learn [14]), which offer simplicity but face memory limitations and performance degradation during feature engineering on large datasets.

Furthermore, existing systems often utilize fixed ensemble architectures. Pre-packaged models like Random Forest and Gradient Boosted Trees (GBT) are available, but they constrain the base learner to be a Decision Tree. Tree-based models do not always possess the correct inductive bias for every dataset. For example, in datasets with strong linear relationships and high-cardinality features, a linear model may outperform a tree. In such cases, a user restricted to standard GBTs may experience suboptimal accuracy. To maximize accuracy, a system must allow for *heterogeneous* ensembles, such as bagging linear models or stacking trees with neural networks. Constructing such architectures in frameworks like Scikit-learn or Spark MLlib requires custom coding and manual management of data loops.

To address these challenges of scale and flexibility, we introduce a declarative, recursive, SQL-native framework for creating heterogeneous machine learning ensembles within the Ocient Hyperscale Data Warehouse (Ocient). Our contribution is the extension of the `CREATE MLMODEL` SQL statement to support three new ensemble types: `BAGGING`, `BOOSTING`, and `STACKING`.

This framework enables users to use SQL to:

Compose Heterogeneous Ensembles: Mix and match any of Ocient's 20+ base model types (e.g., Decision Trees, Logistic Regression, Support Vector Machines) within a single bagging, boosting, or stacking ensemble.

Build Recursively: Create nested models, such as a stacking ensemble whose base learners are themselves other ensembles, using a single SQL statement.

Reduce Infrastructure Complexity: Perform model training directly within the Ocient Massively Parallel Processing (MPP) engine. This removes the requirement to provision external Spark clusters for scale or accept the memory constraints of Scikit-learn.

By integrating these ML capabilities directly into the database kernel, we simplify the creation of models and leverage in-database parallelism to facilitate training on massive datasets.

This paper is structured as follows. Section 2 provides a brief overview of the Ocient architecture, OcientML, and related work. Section 3 details the SQL syntax and semantics. Section 4 discusses key implementation details. We present a comparative analysis in Section 5 and an evaluation in Section 6, before we finally conclude and discuss opportunities for future work in Section 7.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.52

2 Background and Related Work

The generic ensemble framework is built upon two components of the Ocient Hyperscale Data Warehouse: its system architecture and its existing in-database machine learning subsystem, OcientML. We contextualize these capabilities within the broader landscape of in-database machine learning.

2.1 System Architecture

Ocient [17] is a relational, SQL-compliant MPP system designed for Online Analytical Processing (OLAP) workloads on large datasets. Its performance relies on the Compute Adjacent Storage Architecture® (CASA), where high-throughput NVMe drives are co-located with dedicated CPU cores and memory on each *foundation node*. This design minimizes data movement between storage and compute (cf. Figure 1). Ocient's architecture stands in contrast to architectures that separate compute and storage [3, 16, 21].

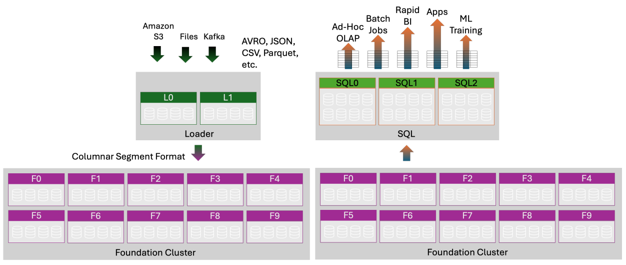

Figure 1: Ocient System Overview

Query processing is distributed across the cluster. An incoming SQL query is parsed and optimized by an *SQL node* into a multi-level distributed execution plan. This plan pushes computation—including filtering, aggregations, and expression evaluation—down to the foundation nodes, increasing data-local processing and parallelism at both the I/O and plan execution levels. This technique, often referred to as *predicate push-down*, is an enabler for in-database model training [19].

2.2 OcientML Foundation

Prior to the work presented in this paper, OcientML provided a library of over 20 in-database machine learning algorithms, including various regression, classification, and clustering models [11]. A key design principle of OcientML is its integration with the SQL language [9], building on the foundations of the relational model [5]. Users create, train, and apply models using DDL and DML constructs.

Model training is initiated via a `CREATE MLMODEL` statement, which takes the result set of a `SELECT` query as its training set. For example:

```
CREATE MLMODEL my_classifier
    TYPE LOGISTIC REGRESSION
    ON ( SELECT feature1, feature2, label
         FROM training_data )
    OPTIONS ( 'metrics' -> 'true' );
```

Listing 1: Creating a simple logistic regression model

The training process is fully distributed: the `SELECT` query is executed across all foundation nodes, and the resulting data is used to train the model in parallel. Once created, the model `my_classifier` becomes a native scalar function that can be used in any SQL query for inference:

```
SELECT customer_id,
       my_classifier(f1, f2) AS predicted_label
FROM new_customer_data;
```

Listing 2: Using a trained model for inference

This "model-as-a-function" paradigm is central to our design. By defining our new ensembles as additional `TYPE` options within the existing `CREATE MLMODEL` framework, they function within the Ocient ecosystem.

2.3 Related Work: In-Database Machine Learning

The concept of moving computation to data is established. Early systems like MADlib [8] introduced analytics functions for PostgreSQL and Greenplum. MADlib relies on imperative function calls (e.g., `madlib.logregr_train()`) rather than unified declarative DDL, requiring users to manage data piping manually to construct ensembles.

Modern cloud data warehouses have increasingly adopted declarative syntax. Google BigQuery ML [7] and Amazon Redshift ML [2] support `CREATE MODEL` statements with various base types (DNNs, XGBoost). However, these implementations are typically monolithic. For example, while BigQuery supports a `BOOSTED_TREE_CLASSIFIER`, it does not support a more generic `BOOSTING` meta-type that can boost arbitrary base learners (e.g., linear regressions). Furthermore, these systems generally lack *composable heterogeneity*: a user cannot declare a stacking ensemble where Level 0 is a mix of BigQuery's Linear models and Trees within a single DDL statement. Achieving such heterogeneity typically requires orchestrating multiple separate training jobs and stitching predictions together via procedural SQL scripts.

Other approaches, such as Snowpark [15] or Vertica's Python integration [13], focus on bringing procedural code (Python/Java) closer to the data. While this allows for flexibility, it re-introduces the complexity of imperative programming and resource management. Ocient's approach differs by keeping the definition strictly declarative and recursive within the SQL kernel, allowing the database optimizer to handle the execution.

Table 1 summarizes the functional differences between Ocient and other in-database ML systems.

Table 1: Functional Comparison of In-Database Ensembles

Feature	Ocient	BigQuery ML	MADlib
SQL-Native Interface	Yes	Yes	No (UDFs)
Heterogeneous Ensembles	**Yes**	No	Manual
Recursive Composition	**Yes**	No	Manual
Single-Statement Definition	**Yes**	No	No

3 A Declarative Framework for Composable Ensembles

We extend the OcientML framework with three new model types—`BAGGING`, `BOOSTING`, and `STACKING`—that enable the declarative creation of generic and composable ensembles. The framework's parameters are exposed through a structured JSON string in the `OPTIONS` clause, allowing users to define model properties and hierarchies within a single SQL statement.

3.1 Generic Bagging Models

Bagging (Bootstrap Aggregating) [4] is an ensemble technique that trains multiple base models independently on different random sub-samples of the training data and "averages" their predictions to reduce variance. The approach for averaging the predictions depends on the result of the base model. For example, if the base model is a regression model, the predictions are mathematically averaged. If the base model is a classification model, a plurality vote decides. While Random Forest is a popular implementation of bagging using Decision Trees, our framework generalizes this concept.

Figure 2 illustrates a bagging ensemble with 3 models. There can be collections of models, e.g. 10 Decision Trees, 5 Naïve Bayes classifiers, etc. The flow of the training data is depicted in blue, while the flow of the inference data is drawn in red.

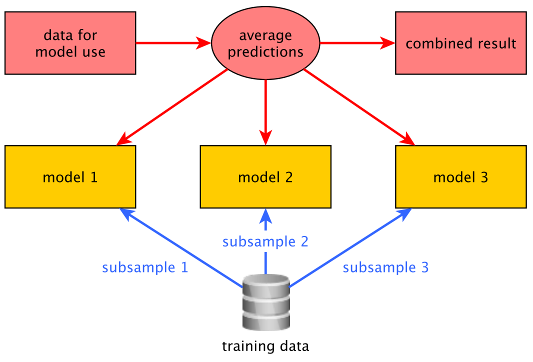

Figure 2: A Bagging Ensemble

A feature of our approach is support for **heterogeneous base models**. Users can create a bagged ensemble from a mix of different model types. The syntax is shown in Listing 3 by means of an example. The `baseModels` option takes a JSON array, where each object specifies a model `type`, the `count` of models to train, and any model-specific `options`.

```
CREATE MLMODEL myMixedEnsemble TYPE BAGGING
    ON ( SELECT f1, f2, f3, target FROM my_data )
    OPTIONS (
        'taskType' -> 'REGRESSION',
        'baseModels' -> '[
            {
                "type": "REGRESSION TREE",
                "count": 10,
                "options": {"maxDepth": 3}
            },
            {
                "type": "MULTIPLE LINEAR REGRESSION",
                "count": 10,
                "options": {}
            }
        ]',
        'bootstrap' -> 'true',
        'fractionPerModel' -> '0.8'
);
```

Listing 3: Creating a heterogeneous bagging regressor (Linear + Tree)

This declarative syntax allows users to create ensembles that are not available in standard libraries. For instance, as shown in Listing 3, one can combine the non-linear capability of tree-based models with the extrapolation capabilities of Linear Regression. Such a specific heterogeneous architecture can outperform standard homogeneous ensembles on complex datasets (cf. Section 6).

3.2 Generic Boosting Models

Boosting [6] sequentially trains a series of weak learners, where each new model focuses on correcting the errors made by its predecessors. We implement a generic Gradient Boosting framework that supports regression and classification tasks.

A specific design decision in our framework is the use of **external weighted resampling**. Instead of requiring base models to support sample weights internally, our boosting execution engine trains each new learner on a temporary dataset created by resampling the original data with replacement. The probability that a sample is selected is proportional to the residual error from the previous stage. This abstraction decouples the boosting logic from the base learner implementation, enabling OcientML regression models to be used as weak learners—including those without native weighting support.

Figure 3 shows the flow for training data and inference data in blue and red, respectively. As with bagging ensembles, collections of models can be used for boosting as well.

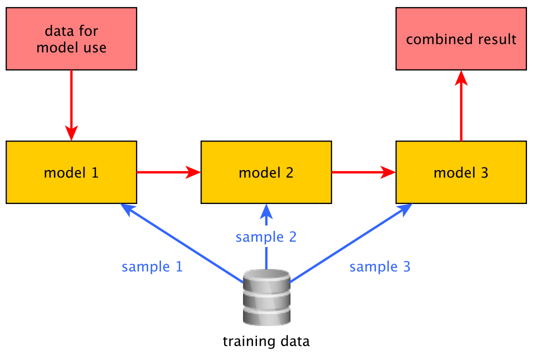

Figure 3: A Boosting Ensemble

Listing 4 shows the creation of a gradient boosting model using shallow regression trees as weak learners to solve a regression problem. The `learningRate` parameter controls the contribution of each weak learner to the final ensemble.

```
CREATE MLMODEL myGradientBooster
    TYPE BOOSTING
    ON ( SELECT f1, f2, f3, target FROM my_data )
    OPTIONS (
        'taskType' -> 'REGRESSION',
        'learningRate' -> '0.1',
        'baseModels' -> '[
            {
                "type": "REGRESSION TREE",
                "count": 100,
                "options": {"maxDepth": 3}
            }
        ]'
    );
```

Listing 4: Creating a gradient boosting regressor

3.3 Generic Stacking and Recursive Composition

Stacking (Stacked Generalization) [18] trains multiple base models (Level 0) and then uses their predictions as features to train a meta-learner model (Level 1). This allows the system to learn how to combine the outputs from different models. Our framework enables **recursive and composable** structures, exemplified by stacking. Figure 4 illustrates this flow for training data in blue. Inference is depicted in red.

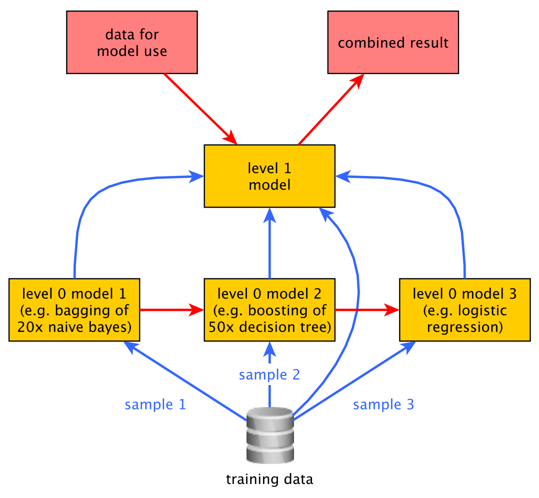
data for model use
combined result
level 1 model
level 0 model 1 (e.g. bagging of 20x naive bayes)
level 0 model 2 (e.g. boosting of 50x decision tree)
level 0 model 3 (e.g. logistic regression)
sample 1
sample 2
sample 3
training data

Figure 4: A Stacking Ensemble

As shown in Listing 5, a user defines the Level 1 meta-learner via `l1ModelType` and its options.[1] The Level 0 base learners are defined in the `l0Models` JSON array. The `type` of an `l0Models` entry is not restricted to a simple algorithm; it can be another ensemble type, such as `BAGGING` or `BOOSTING`.

```
CREATE MLMODEL ultimateEnsemble
    TYPE STACKING
    ON ( SELECT f1, f2, f3, f4, f5, target FROM my_data
        )
    OPTIONS (
        'l1ModelType' -> 'LOGISTIC_REGRESSION',
        'l0Models' -> '[
            {
                "name": "l0_bagged_bayes",
                "type": "BAGGING",
                "options": {
                    "taskType": "CLASSIFICATION",
                    "baseModels":
                        "[{\"type\": \"NAIVE BAYES\",
                           \"count\": 20}]"
                }
            },
            {
                "name": "l0_boosted_trees",
                "type": "BOOSTING",
                "options": {
                    "taskType": "CLASSIFICATION",
                    "baseModels": "
                        [{\"type\": \"DECISION TREE\",
                          \"count\": 50}]"
                }
            }
        ]'
    );
```

Listing 5: A nested stacking ensemble

This single SQL statement defines a model consisting of a logistic regression meta-learner that combines predictions from a 20-model bagged Naïve Bayes ensemble and a 50-model gradient-boosted tree ensemble. The database manages the workflow: parallel training of the 20 Naïve Bayes models, sequential training of the 50 Decision Trees, generating the L1 training set by running inference with the L0 models, and finally training the L1 logistic regression model. This declarative method for composing complex ML pipelines within SQL represents a simplification over traditional approaches.

[1]The listing has been reformatted to accommodate the short line width.

4 Implementation and Architecture

The declarative SQL interface for ensembles is supported by an execution framework that leverages Ocient's distributed architecture. The implementation is centered around a recursive execution model, a storage-aware persistence strategy, and a workload-managed approach to system stability.

4.1 The Recursive Dispatch Pattern

To handle the nested structure of an ensemble definition, we implemented a recursive, type-dispatching execution process. When a `CREATE MLMODEL` statement for an ensemble is parsed, a primary "runner" process is instantiated based on the top-level model type (e.g., `stackingRunner_t`).

This primary runner is responsible for parsing its own options and managing the creation of its constituent models. If a base model is itself an ensemble, the primary runner instantiates the appropriate sub-runner (e.g., a `baggingRunner_t`) and delegates the creation task to it, passing along the relevant section of the JSON options.

This design ensures modularity. While implemented here within Ocient's MPP architecture, the core abstraction—treating a model training task as a recursive function call—allows the system to support complex, nested model hierarchies without hard-coding specific permutations.

4.2 Model Persistence and Storage Scopes

Every model, including sub-models, is stored in the OcientML system catalogs. The final ensemble model (e.g., a `STACKING` model) does not contain copies of its base models. Instead, its artifact is a structure that stores references to its child models, allowing the ensemble to adopt model and parameter updates as the solution evolves. When the ensemble is called for inference, it invokes its child models via these internal references.

To ensure atomicity during ensemble creation, we leverage Ocient's native **Storage Scopes**. Every table and segment created during a `CREATE MLMODEL` statement is associated with a specific storage scope. These scopes provide transactional-like semantics without the overhead of a traditional Write-Ahead Logging (WAL). Data is written to new segments on disk, and their metadata is tracked via a Raft-based consensus layer [12]. If the training process completes successfully, the storage scope is made visible cluster-wide. If an error occurs (such as an OOM or disk failure), the entire storage scope and all child model artifacts are discarded, ensuring that failed multi-stage ensembles do not leave partial models in the catalog.

4.3 Implementation of Stacking and Boosting

The implementations for `STACKING` and `BOOSTING` require distinct solutions for data flow and model compatibility.

Stacking and Data Abundance: For `STACKING`, the L1 training set must be generated by running inference with the L0 models. While small-scale frameworks often rely on K-Fold cross-validation to maximize limited data, Ocient is designed for hyperscale environments where data is abundant. We utilize a "Holdout" strategy, automatically partitioning the dataset to ensure the L1 meta-learner is trained on predictions from L0 models that have not seen those specific rows. In environments with billions of rows, this strategy provides a statistically representative sample for the meta-learner while avoiding the computational overhead of retraining L0 models K times.

Boosting via No-Log Resampling: For BOOSTING, we implement external weighting via resampling. For each iteration, the boostingRunner_t generates a training set for the new weak learner by creating a temporary table containing a bootstrapped sample. To prevent metadata contention during high-iteration boosting, these temporary tables utilize the same no-log "Storage Scope" mechanism described above. By operating without a WAL and using large-segment I/O, Ocient can create and tear down these iteration-specific datasets with reduced overhead relative to the training time.

4.4 Resource Governance and Stability

A common concern with recursive execution models is the risk of resource exhaustion or "fork bombs". Ocient addresses this through its integrated **Workload Management (WLM)**.

Ensemble training does not spawn unmanaged threads or processes. Instead, every sub-model training task is scheduled within the user's existing WLM resource group. The WLM acts as a central governor, queuing tasks if the system's concurrency limits or memory quotas are reached. This ensures that a user cannot destabilize the cluster by defining a deeply nested ensemble; rather than enforcing an arbitrary recursion depth limit, the system relies on WLM to serialize the execution of the sub-models based on available resources. Furthermore, if memory pressure increases, Ocient can page intermediate ML structures to NVMe storage.

5 Comparative Analysis: Declarative vs. Imperative

To illustrate the application of a declarative, in-database approach, we contrast the creation of the ultimateEnsemble model from Listing 5 in Ocient with the steps required to build an equivalent model using Apache Spark (PySpark).

5.1 Ocient: A Single Declarative Statement

As previously shown, the nested ensemble is defined and trained with a single SQL command spanning 14 lines of code. The user declares the desired final model structure, and the database handles the multi-stage execution, data movement, parallelism, and model persistence.

5.2 PySpark: A Multi-Stage, Imperative Workflow

Building the same heterogeneous stacking model in PySpark is a complex task. Because Spark MLlib lacks generic "Bagging" or "Stacking" meta-estimators that accept arbitrary base models (it primarily supports specific implementations like RandomForest), the user must implement the ensemble logic manually. An equivalent implementation required approximately **120 lines of imperative code** and involved:

Data Extraction (ETL): Extracting data from the warehouse to object storage (e.g., Parquet), introducing latency.

Manual Bagging Loops: Writing explicit Python loops to bootstrap sample the data 20 times and train individual Naïve Bayes models, as no BaggingClassifier exists for Naïve Bayes in Spark.

Manual Stacking Logic: Generating predictions from the L0 models (the Bagged Naïve Bayes and Boosted Trees) and joining these predictions to create the feature set for the L1 Logistic Regression.

Pipeline Assembly: Creating a custom prediction function to orchestrate the flow of data through the 20+ sub-models during inference.

This comparison highlights the simplification offered by our framework. Ocient abstracts the multi-stage workflow into a single command, reducing code volume (by almost 90%) and eliminating the primary performance bottleneck—ETL and data movement—by performing all operations inside the database engine.

6 Evaluation

We evaluated our framework using the "Price Prediction" dataset (Use Case 5) from the TPCx-AI benchmark [1]. This workload involves predicting the price of retail items based on unstructured text descriptions.

The goal of this evaluation is to demonstrate three key architectural claims:

Necessity of Heterogeneity: That restricting users to homogeneous ensembles (e.g., only Trees) creates an accuracy limit that can be overcome by mixing model types (e.g., Linear + Trees).

Memory Scalability: That Ocient scales beyond the limits of single-node libraries like Scikit-learn.

No "Preprocessing Wall": That at hyperscale (1 TB+), the primary performance bottleneck is data engineering (joins/aggregations) rather than model training, giving Ocient an advantage over separated compute engines like Spark.

6.1 Methodology and Setup

6.1.1 Dataset and Feature Engineering. We utilized one of the use cases (UC5) from TPCx-AI at multiple scale factors, ranging from a 17 MB input table up to a 1 TB input table. Note that because Use Case 5 represents a small fraction of the total benchmark data, a 1 TB input table for this specific use case corresponds to a TPCx-AI Scale Factor of 3.24 exabytes. The 1 TB input tables contained 5.68 billion rows.

The feature engineering process is computationally intensive, requiring the tokenization of product descriptions into 1-grams and 2-grams, joining them against the training set, and computing statistical features. At the 1 TB scale, this process generates large intermediate results: the 1-gram intermediate table contained 148 billion rows (4 TB), and the 2-gram table contained 154 billion rows (5.1 TB).

While this n-gram based approach deviates from the standard methodology for Use Case 5, it was selected to specifically showcase the capabilities of our ensemble framework. This strategy allowed us to demonstrate competitive training times and model quality metrics while rigorously testing the system's ability to handle the massive data volumes associated with heterogeneous ensemble creation.

6.1.2 Hardware and Baselines. The experiments were conducted on an Ocient cluster with 8 worker ("foundation") nodes. Each node is equipped with an AMD EPYC 9654 96-Core Processor, 2 TB of system memory, and twelve 7.68 TB NVMe SSDs.

We compared against two baselines:

Single-Node Scikit-learn: Running on a single node with identical hardware specifications as above.

Distributed Apache Spark: Running on a cluster of 5 worker nodes with identical specifications as above.

6.2 Experimental Results

6.2.1 The Necessity of Generic Ensembles (Inductive Bias). We first tested the accuracy impact of model restrictions. Table 2 compares a single Multiple Linear Regression (MLR) against a single Regression Tree.

Table 2: Inductive Bias Test (17 MB Dataset)

Model Type	RMSLE (Lower is Better)
Single Regression Tree	0.3092
Single Linear Regression (MLR)	**0.2980**

The Linear model outperformed the Tree model, confirming that the dataset possesses strong linear relationships. This validates our architectural choice: a system that restricts users to tree-based ensembles (like standard GBT) forces them to accept suboptimal accuracy. Using our generic framework to create a **Mixed Ensemble** (50/50 Bagging of Linear + Trees) further reduced the error to **0.1799**.

6.2.2 Scalability vs. Single-Node Baselines. We compared the execution time of Ocient versus Scikit-learn across increasing dataset sizes to identify the "Memory Wall".

Table 3: Performance Scalability: Ocient vs. Scikit-learn

	Scikit-learn (s)			Ocient (s)
Dataset	Feat. Eng.	Training	**Total**	**Total**
17 MB	5.5	3.5	9.0	76.9
56 MB	18.0	8.0	26.0	84.5
180 MB	52.0	21.0	73.0	89.4
574 MB	198.0	37.0	235.0	**79.8**
1.8 GB	655.0	144.0	799.0	**100.8**
5.7 GB	3152.0	122.0	3274.0	**125.0**
1 TB	*Out of Memory*			**16,627.3**

As shown in Table 3, Ocient overcomes the overhead of distributed coordination at roughly 500 MB and scales linearly, while Scikit-learn hits a bottleneck during feature engineering. On the 5.7 GB dataset, Python required over 50 minutes merely to tokenize strings, while Ocient completed the entire workflow in roughly 2 minutes. At the 1 TB scale, the single-node approach failed due to memory exhaustion.

6.2.3 Scalability vs. Distributed Baselines. To evaluate performance at the 1 TB scale, we compared Ocient against the distributed Spark cluster.

Table 4: Performance Breakdown at 1 TB Scale

Phase	Spark (5 Nodes)	Ocient (8 Nodes)	Speedup
Preprocessing	14,266 s	**3,296 s**	**4.3×**
Model Training	**12,423 s**	13,331 s	0.9×
Total End-to-End	26,690 s	16,627 s	1.6×[2]

The results in Table 4 illustrate the architectural trade-offs between in-database processing and dedicated ML engines and are discussed below.

[2]Comparison is unadjusted for hardware differences (5 vs 8 nodes).

The Preprocessing Wall: Ocient demonstrates a significant advantage in the feature engineering phase. Generating the 300+ billion rows of intermediate n-gram data took Spark over 4 hours (14,266 s). Ocient completed this phase in under an hour (3,296 s). Even if we linearly scale the Spark performance to match the 8-node Ocient configuration (multiplying by 5/8), the theoretical Spark preprocessing time would be ≈8,900 seconds. Ocient remains **2.7× faster** in this data-intensive phase.

Training Efficiency: Conversely, Spark's MLlib, which utilizes highly optimized C++ backends for tree construction, outperformed Ocient in the pure model training phase.

End-to-End Analysis: When adjusting for hardware parity, a theoretical 8-node Spark cluster could complete the workload in approximately 16,682 seconds. Recent optimizations to the Ocient regression tree implementation yielded a total runtime of 16,627 seconds. This result indicates that the in-database approach has reached performance parity (approximately 0.3% faster) with the theoretical scalability of the dedicated engine. Ultimately, the two systems display inverted strengths: Ocient dominates the data-intensive preprocessing phase, while Spark is faster at the compute-intensive training phase. However, Ocient achieves this performance while eliminating the complexity of managing a separate cluster and external ETL pipelines.

6.2.4 Usability and Tuning Overhead. Beyond raw execution time, we observed a significant difference in usability. The Ocient workload ran successfully on the first attempt using default system configurations ("out of the box"). In contrast, the Spark workload failed repeatedly on the 1 TB dataset, requiring six attempts to achieve a successful run.

The failures were primarily due to OOM (Out Of Memory) errors during the feature engineering phase. Success required manual tuning of the number of partitions, the number of executors, the number of cores per executor, the amount of memory per executor, etc. This "human time" overhead is a hidden cost of separated compute architectures that Ocient's managed execution environment eliminates.

7 Conclusion and Future Work

In this paper, we introduced a SQL-native framework for constructing generic and recursive machine learning ensembles. By decoupling the ensemble strategy from the base learner, we addressed a gap in existing in-database ML systems: the inability to adapt the model architecture to the data's inductive bias.

Our evaluation demonstrated that this flexibility is necessary for high accuracy. On a real-world price prediction workload, standard tree-based ensembles failed to capture the underlying linear relationships in the data. By leveraging our framework to construct a heterogeneous ensemble mixing Linear Regression and Decision Trees, we reduced the error by approximately 41% compared to individual base models. Furthermore, we showed that this approach scales beyond multi-gigabyte datasets where traditional Python-based tools fail due to memory constraints.

Future work will focus on automating the model selection process to identify the optimal mix of base learners for a given dataset, thereby lowering the barrier to entry for in-database machine learning. Additional directions include support for incremental ensemble updates—adding or replacing base learners without full retraining.

Trademarks

References

[1] 2024. *TPC Express AI (TPCx-AI) Standard Specification, Revision 2.0.0.* Technical Report TPCx-AI v2.0.0. Transaction Processing Performance Council (TPC), San Francisco, CA. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPCx-AI_v2.0.0.pdf [Online; accessed 2025-11-27].

[2] Amazon Web Services. 2024. Amazon Redshift ML. https://aws.amazon.com/redshift/features/redshift-ml/. [Online; accessed 2025-11-20].

[3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

[4] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140. https://doi.org/10.1007/BF00058655

[5] E. F. Codd. 1983. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 26, 1 (1983), 64–69. https://doi.org/10.1145/357980.358007

[6] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232. https://doi.org/10.1214/aos/1013203451

[7] Google Cloud. 2024. BigQuery ML: Machine Learning in SQL. https://cloud.google.com/bigquery-ml/docs/introduction. [Online; accessed 2025-11-20].

[8] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[9] ISO/IEC. 2023. *ISO/IEC 9075-2, Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation).* Standard. International Organization for Standardization.

[10] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[11] Ocient Inc. 2025. *Ocient Documentation – Version 25.* https://docs.ocient.com/ [Online; accessed 2025-10-27].

[12] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[13] OpenText. 2024. In-Database Machine Learning with Vertica. https://www.vertica.com/python/. [Online; accessed 2025-11-20].

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[15] Snowflake Inc. 2024. Snowpark: The Data Programmability Framework. https://docs.snowflake.com/en/developer-guide/snowpark/index. [Online; accessed 2025-11-20].

[16] Snowflake Inc. 2025. *Snowflake Documentation – Key Concepts & Architecture.* https://docs.snowflake.com/en/user-guide/intro-key-concepts [Online; accessed 2025-10-27].

[17] Knut Stolze, Jason Arnold, and Andrew Park. 2025. Ocient: A Hyperscale Database System for Next-Generation OLAP Workloads. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025) (LNI).* Gesellschaft für Informatik e.V.

[18] David H. Wolpert. 1992. Stacked generalization. *Neural Networks* 5, 2 (1992), 241–259. https://doi.org/10.1016/S0893-6080(05)80023-1

[19] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment* 14, 11 (2021). https://doi.org/10.14778/3476249.3476273

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing.*

[21] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings.* www.cidrdb.org. https://vldb.org/cidrdb/2021/lakehouse-a-new-generation-of-open-platforms-that-unify-data-warehousing-and-advanced-analytics.html