AlloyDB Omni Developers Guide

Table of contents

Build with the PostgreSQL community

Because AlloyDB Omni is compatible with PostgreSQL, you can use it with a wide range of open-source PostgreSQL tools. This section covers some of the open-source PostgreSQL tools that work with AlloyDB Omni.

pgAdmin

pgAdmin is an open-source administration and development platform for PostgreSQL. It provides a graphical user interface (GUI) to manage PostgreSQL databases and their objects.

pgBackRest

pgBackRest is an open-source database backup server that you can use to back up and restore your AlloyDB Omni database clusters. For more information about backup and restore with pgBackRest, see "Configure pgBackRest" in the AlloyDB Omni Configuration Guide.

Barman

Barman is another open-source administration tool designed for disaster recovery and management of PostgreSQL servers. For more information, see Set up Barman for AlloyDB Omni.

Patroni

Patroni is an open-source tool that manages and automates high availability (HA) for PostgreSQL clusters. For more information, see "High Availability and DR" in the AlloyDB Omni Configuration Guide.

pg_auto_failover

pg_auto_failover is an open-source PostgreSQL extension that simplifies the process of setting up and managing a highly available AlloyDB Omni environment.

Develop with AlloyDB Omni

This section covers various AlloyDB features and how to use them.

Enable and monitor the columnar engine

The AlloyDB columnar engine accelerates SQL query processing of scans, joins, and aggregates by providing these components:

- A column store that contains table and materialized-view data for selected columns, reorganized into a column-oriented format.
- A columnar query planner and execution engine to support use of the column store in queries.

The columnar engine can be used on the primary instance, a read replica instance, or both. You can also use auto-columnarization to analyze your workload and automatically populate the column store with the columns that provide the best performance gain.

To use the columnar engine with a specific query, all columns in the query fragments, such as joins and scans, must be in the column store.

By default, the columnar engine is set to use 30% of your instance's memory. Depending on your workload, memory usage, and whether you have a read replica configured, you can reduce the columnar engine memory allocation on your primary instance and allocate more memory to the read replica instance.

To view and monitor memory usage by the columnar engine, see "View column store memory usage" in this guide. To modify the memory size used by the column store, see "Configure the size of the column store" in this guide. To find the recommended columnar engine memory size for your instance, see Recommend column store memory size.

Query types that benefit from the columnar engine

The following is a list of operations and their query patterns that benefit most from the columnar engine:

- Table scans
    - It has selective filters, such as `WHERE` clauses.
    - It uses a small number of columns from a larger table or materialized view.
    - It uses expressions such as `LIKE`, `SUBSTR`, or `TRIM`.
- Aggregation functions
    - They only use the following expressions: `SUM`, `MIN`, `MAX`, `AVG`, and `COUNT`.
    - They are at the beginning of the query fragment of a columnar scan.
    - They are ungrouped, or are grouped-by base columns.
- `ORDER-BY`: only if the operator is on the result of a columnar scan.
- `SORT`: only if the operator is on the result of a columnar scan and sorts only on the base columns of the table or the materialized view.
- `LIMIT`: only if the operator is on a result of a columnar scan and is before any `SORT` or `GROUP BY` operators.
- `INNER HASH JOIN`: only if the keys used are columns and no join qualifiers are used.

For more information about which queries work best with the columnar engine, and whether the columnar engine was used by a query and how, see "Verify usage of the columnar engine using `EXPLAIN`" in this guide.

Use the columnar engine

To use the columnar engine in an AlloyDB instance, perform these high-level steps:

1. Enable the columnar engine on the instance. Enabling the engine is a one-time operation and requires a restart.
2. Add columns to the column store. To add columns to the column store, use one of the following methods:
    - Use auto-columnarization, which analyzes your workload and automatically adds columns.

    - Add the columns manually based on your knowledge of the workload on the databases in the instance.
3. You can track what's in the column store using the `g_columnar_relations` view, and, after columns have been added, you can use the `EXPLAIN` statement to verify usage of the columnar engine in SQL queries.

For more information about how to use the columnar engine, see "Configure the columnar engine" in this guide.

What data you can add to the column store

There are some limitations on the data types and data sources that you can use when you add columns to the column store.

Supported data types

The columnar engine supports only columns with the following built-in data types:

- `array`
- `bigint`
- `boolean`
- `bytea`
- `char`
- `date`
- `decimal`
- `double precision`
- `enum`
- `float4`
- `float8`
- `integer`
- `json`
- `jsonb`
- `numeric`
- `real`
- `serial`
- `short`
- `smallint`
- `text`
- `timestamp`
- `uuid`
- `varchar`

Note: The columnar engine ignores attempts to manually add columns that contain unsupported data types to the column store.

Unsupported data sources

The columnar engine doesn't support tables or materialized views with the following attributes as data sources:

- Non-leaf partitioned tables
- Foreign tables
- Tables or views with fewer than 5,000 rows

Columnar engine limitations

- If you're running an analytical query on a column that has an index, then the AlloyDB optimizer might choose to use row-store with an index scan.
- Columns that you add to the column store manually aren't automatically removed. To force-remove manually added columns, use `google_columnar_engine_drop()` on your instance.
- Auto-columnarization might dynamically add and remove columns based on your workload.
- Not all data types are supported by the columnar engine. For more information, see "Supported data types" in this guide.
- Frequent updates to rows invalidate columnar data. To validate a table or a materialized view in the columnar store, you can either reduce the update frequency or schedule more frequent refreshes from row-store to the columnar engine. To check whether your table or view is impacted, you can compare the `invalid_block_count` and `total_block_count` columns in `g_columnar_relations`. If you have frequent or high-volume changes to your table or view, then the `invalid_block_count` will be high.

Configure the columnar engine

This section describes how to enable or disable the columnar engine on an AlloyDB Omni database cluster. It also covers how to configure an appropriate initial size for its column store.

Enable the columnar engine

To use columnar engine on an instance, set the instance's `google_columnar_engine.enabled` flag to on.

To set this flag on an instance, follow these steps:

1. Run the `ALTER SYSTEM` PostgreSQL command:

```
ALTER SYSTEM SET google_columnar_engine.enabled = 'on';
```

2. Optional: Configure the size of the column store before you restart the database server. See "Configure the size of the column store" in this guide.
3. Restart the database server. See "Manage AlloyDB Omni instances" in the AlloyDB Omni Configuration Guide.

Configure the size of the column store

While the columnar engine is enabled on an instance, AlloyDB allocates a portion of the instance's memory to store its columnar data. Dedicating high-speed RAM to your column store helps AlloyDB access the columnar data as rapidly as possible.

You can also set the allocation to a fixed and specific size using the `google_columnar_engine.memory_size_in_mb` flag. To set this flag on an instance, follow these steps:

1. Run the `ALTER SYSTEM` PostgreSQL command:

   ```
   ALTER SYSTEM SET google_columnar_engine.memory_size_in_mb =
   COLUMN_STORE_SIZE;
   ```

   Replace `COLUMN_STORE_SIZE` with the new size of the column store, in megabytes.

2. Restart the database server. See "Manage AlloyDB Omni instances" in the AlloyDB Omni Configuration Guide.

Enable vectorized join

The columnar engine has a vectorized join feature that can improve the performance of joins by applying vectorized processing to qualifying queries.

After you enable vectorized join, the AlloyDB query planner has the option to apply the vectorized join operator instead of the standard PostgreSQL hash join operator, depending on the cost. The vectorized join node is shown in the explain plan.

To enable vectorized join on an instance, set the instance's `google_columnar_engine.enable_vectorized_join` flag to on.

To set this flag on an instance, run the `ALTER SYSTEM` PostgreSQL command:

```
ALTER SYSTEM SET google_columnar_engine.enable_vectorized_join = 'on';
```

AlloyDB allocates one thread to the vectorized join feature by default. You can increase the number of threads available to this feature by setting the `google_columnar_engine.vectorized_join_threads` flag to a larger value.

Manually refresh your columnar engine

By default, the columnar engine is set to automatically refresh the column store in the background when enabled. You might need to manually refresh the columnar store in certain situations, such as if auto-refresh doesn't refresh a relation with a high number of invalid blocks.

To manually refresh the column engine, run the following SQL query:

```
SELECT google_columnar_engine_refresh('TABLE_NAME');
```

Replace `TABLE_NAME` with the name of the table or the materialized view that you want to manually refresh.

Disable the columnar engine

To disable the columbar engine on an instance, set the `google_columnar_engine.enabled` flag to `off`.

To set this flag on an instance, follow these steps:

1. Run the `ALTER SYSTEM` PostgreSQL command:
   ```
   ALTER SYSTEM SET google_columnar_engine.enabled = 'off';
   ```
2. Restart the database server. See "Manage AlloyDB Omni instances" in the AlloyDB Omni Configuration Guide.

Troubleshoot the columnar engine

Fix an insufficient shared memory error

If you run AlloyDB Omni without enough shared memory for the columnar engine to use, then you might see the following error:

```
Insufficient shared memory for generating the columnar formats.
```

You can address this issue by specifying the amount of shared memory that's available to the AlloyDB Omni container. The process varies depending on your host operating system.

Linux

Increase the size of your host machine's `/dev/shm` partition using a technique such as editing your `/etc/fstab` file.

To do this, you must have installed AlloyDB Omni with your `/dev/shm` directory mounted on the container, as shown in Customize your AlloyDB Omni installation.

macOS

Install a new AlloyDB Omni container, specifying a larger shared-memory value for the `--shm-size` flag.

Fix columns not getting populated

If columns don't populate in the columnar engine, then one of the following situations might be true:

- The columns that you want to add include an unsupported data type.
- The requirements of the columnar engine aren't being met.

To find the cause of this issue, try the following:

- Confirm that the tables or materialized views in our query are in the columnar engine.
- Verify the usage of the columnar engine using the `EXPLAIN` statement.

Monitor the columnar engine

This section describes how to monitor utilization of the columnar engine.

Verify usage of the columnar engine using EXPLAIN

You can verify the usage of the columnar engine by using the `EXPLAIN` statement to observe the new columnar operators that appear in a query's generated query plan.

```
EXPLAIN (ANALYZE,COSTS OFF,BUFFERS,TIMING OFF,SUMMARY OFF)
  SELECT l_returnflag, l_linestatus, l_quantity, l_extendedprice,
         l_discount, l_tax
    FROM lineitem
   WHERE l_shipdate <= date '1992-08-06'
;
                                 QUERY PLAN
-----------------------------------------------------------------------------
---
 Append (actual rows=3941797 loops=1)
   Buffers: shared hit=9
   ->  Custom Scan (columnar scan) on lineitem (actual rows=3941797
loops=1)
         Filter: (l_shipdate <= '1992-08-06'::date)
         Rows Removed by Columnar Filter: 56054083
         Columnar cache search mode: columnar filter only
         Buffers: shared hit=9
   ->  Seq Scan on lineitem (never executed)
  Filter: (l_shipdate <= '1992-08-06'::date)
```

- `Custom Scan (columnar scan)` indicates that columnar-engine scanning is being included in the query plan.
- `Rows Removed by Columnar Filter` lists the number of rows filtered out by the columnar vectorized execution.
- `Columnar cache search mode` can be `columnar filter only`, `native`, or `row store scan`. The planner chooses the search mode automatically based on costing and pushdown evaluation capability.

When the planner chooses the `native` mode, it pushes down some of the columnar operators to the scan:

- `Rows Aggregated by Columnar Scan` lists the number of rows that are aggregated.
- `Rows Sorted by Columnar Scan` lists the number of rows that are sorted.
- `Rows Limited by Columnar Scan` lists the limited number of rows that were scanned.

With joins, columnar scan operators can also use the `late materialization` mode.

```
EXPLAIN (ANALYZE,COSTS OFF,BUFFERS,TIMING OFF,SUMMARY OFF)
  SELECT l_shipmode, o_orderpriority
    FROM orders, lineitem
   WHERE o_orderkey = l_orderkey
         AND l_shipmode in ('AIR', 'FOB')
         AND l_receiptdate >= date '1995-01-01'
;
                                 QUERY PLAN
--------------------------------------------------------------------------
---
 Hash Join (actual rows=9865288 loops=1)
   Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
   Buffers: temp read=127738 written=127738
   ->  Append (actual rows=9865288 loops=1)
         ->  Custom Scan (columnar scan) on lineitem (actual rows=9865288
loops=1)
               Filter: ((l_shipmode = ANY ('{AIR,FOB}'::bpchar[])) AND
               (l_receiptdate >= '1995-01-01'::date))
               Rows Removed by Columnar Filter: 50130592
               Columnar cache search mode: native
         ->  Index Scan using idx_lineitem_orderkey_fkidx on lineitem
             (never executed)
               Filter: ((l_shipmode = ANY ('{AIR,FOB}'::bpchar[])) AND
               (l_receiptdate >= '1995-01-01'::date))
   ->  Hash (actual rows=15000000 loops=1)
         Buckets: 1048576  Batches: 32  Memory Usage: 37006kB
         Buffers: temp written=83357
         ->  Append (actual rows=15000000 loops=1)
               ->  Custom Scan (columnar scan) on orders (actual
rows=15000000
                   loops=1)
                     Rows Removed by Columnar Filter: 0
                     Columnar projection mode: late materialization
                     Columnar cache search mode: native
               ->  Seq Scan on orders (never executed)
```

Columnar projection mode can be `late materialization`. Columnar operators choose this mode automatically when the planner optimizes the projection by deferring the materialization of some column values.

View information about tables with columns in the column store

You can view information about the tables or the materialized views with columns in the column store by querying the `g_columnar_relations` view.

```
SELECT * FROM g_columnar_relations;

┌─[ RECORD 1 ]────────┬───────────────────┐
│ relation_name       │ tbl_parallel_test │
│ schema_name         │ public            │
│ database_name       │ advisor           │
│ status              │ Usable            │
│ size                │ 581431259         │
│ columnar_unit_count │ 3                 │
│ invalid_block_count │ 0                 │
│ total_block_count   │ 8337              │
├─[ RECORD 2 ]────────┼───────────────────┤
│ relation_name       │ lineitem          │
│ schema_name         │ public            │
```

```
| database_name       | advisor              |
| status              | Usable               |
| size                | 423224944            |
| columnar_unit_count | 29                   |
| invalid_block_count | 0                    |
| total_block_count   | 115662               |
├─[ RECORD 3 ]────────┼──────────────────────┤
```

View information about the columns in the column store

You can view information about the columns in the column store by querying the `g_columnar_columns` view, including those columns' size and the last access time.

```
SELECT database_name, schema_name, relation_name, column_name,
size_in_bytes, last_accessed_time FROM g_columnar_columns;
```

View columnar engine execution statistics for recent queries

You can view columnar engine execution statistics for recent queries using the `g_columnar_stat_statements` view. This view adds columnar engine statistics to the `pg_stat_statements` view provided by the `pg_stat_statements` extension. To use this view, you must first enable the `pg_stat_statements` extension.

1. Enable the `pg_stat_statements` extension:

   ```
   CREATE EXTENSION pg_stat_statements;
   ```

2. Make the queries whose statistics you want to view. You can do this manually, or you can let enough time pass so that your applications make these queries with `pg_stat_statements` enabled.
3. Query the `g_columnar_stat_statements` and `pg_stat_statements` views. Note the following query retrieves all the columnar execution statistics including those that were collected before the extension `pg_stat_statements` was created. The null value of `userid` indicates that the statistics were collected before the extension `pg_stat_statements` was created.

   ```
   SELECT *
   FROM pg_stat_statements(TRUE) AS pg_stats
        FULL JOIN g_columnar_stat_statements AS g_stats
        ON pg_stats.userid = g_stats.user_id AND
   ```

```
        pg_stats.dbid = g_stats.db_id AND
        pg_stats.queryid = g_stats.query_id
WHERE columnar_unit_read > 0;

┌─[ RECORD 1 ]──────────┬───────────────────────────────┐
│ userid                │ 10                            │
│ dbid                  │ 33004                         │
│ queryid               │ 6779068104316758833           │
│ query                 │ SELECT  l_returnflag,        ↵│
│                       │         l_linestatus,        ↵│
│                       │         l_quantity,          ↵│
│                       │         l_extendedprice,     ↵│
│                       │         l_discount,          ↵│
│                       │         l_tax                ↵│
│                       │FROM  lineitem                ↵│
│                       │WHERE  l_shipdate <= date $1│
│ calls                 │ 1                             │
│ total_time            │ 299.969983                    │
│ min_time              │ 299.969983                    │
│ max_time              │ 299.969983                    │
│ mean_time             │ 299.969983                    │
│ stddev_time           │ 0                             │
│ rows                  │ 392164                        │
│ shared_blks_hit       │ 0                             │
│ shared_blks_read      │ 0                             │
│ shared_blks_dirtied   │ 0                             │
│ shared_blks_written   │ 0                             │
│ local_blks_hit        │ 0                             │
│ local_blks_read       │ 0                             │
│ local_blks_dirtied    │ 0                             │
│ local_blks_written    │ 0                             │
│ temp_blks_read        │ 0                             │
│ temp_blks_written     │ 0                             │
│ blk_read_time         │ 0                             │
│ blk_write_time        │ 0                             │
│ user_id               │ 10                            │
│ db_id                 │ 33004                         │
│ query_id              │ 6779068104316758833           │
│ columnar_unit_read    │ 29                            │
│ page_read             │ 115662                        │
│ rows_filtered        │ 0                            │
│ columnar_scan_time   │ 0                            │
└──────────────────────┴──────────────────────────────┘
```

View column store memory usage

To see the amount of unused RAM available to the columnar engine, you can query the `google_columnar_engine_memory_available()` function. The resulting integer shows the available memory in megabytes (MB).

Unset

```
SELECT google_columnar_engine_memory_available();
```

Store, index, and query vector embeddings with `pgvector`

AlloyDB includes optimizations that let it work especially well with the `pgvector` extension. You can create indexes on vector-storing columns that can significantly speed up certain queries.

Query and index embeddings using `pgvector`

The `pgvector` PostgreSQL extension lets you use vector-specific operators and functions when you store, index, and query text embeddings in your database. AlloyDB has its own optimizations for working with `pgvector`, letting you create indexes that can significantly speed up certain queries that involve embeddings.

Create an optimized nearest-neighbor index

Stock `pgvector` supports approximate nearest-neighbor searching through indexing. AlloyDB adds to this support with a scalar quantization feature that you can specify when you create an index. When enabled, scalar quantization can significantly speed up queries that have larger dimensional vectors, and lets you store vectors with up to 8,000 dimensions.

Note: As an alternative to using the `ivf` index type described by this section, a Google-developed index type for nearest-neighbor searching is available for AlloyDB Omni as a Technology Preview. For more information, see "Preview the ScaNN algorithm" in this guide.

To enable scalar quantization on a `pgvector`-based index, specify `ivf` as the index method, and SQ8 as the quantizer:

Unset

```
CREATE INDEX ON TABLE
  USING ivf (EMBEDDING_COLUMN DISTANCE_FUNCTION)
  WITH (lists = LIST_COUNT, quantizer = 'SQ8');
```

Replace the following:

- `TABLE`: the table to add the index to
- `EMBEDDING_COLUMN`: a column that stores vector data
- `DISTANCE_FUNCTION`: the distance function to use with this index. Choose one of the following:
    - **L2 distance**: `vector_l2_ops`
    - **Inner product**: `vector_ip_ops`
    - **Cosine distance**: `vector_cosine_ops`
- `LIST_COUNT`: the number of lists to use with this index

To create this index on an embedding column that uses the `real[]` data type instead of `vector`, cast the column into the vector data type:

Unset

```
CREATE INDEX ON TABLE
  USING ivf ((CAST(EMBEDDING_COLUMN AS vector(DIMENSIONS)))'}}
DISTANCE_FUNCTION)
  WITH (lists = LIST_COUNT, quantizer = 'SQ8');
```

Replace `DIMENSIONS` with the dimensional width of the embedding column.

The next section demonstrates an example of this kind of index.

Make a nearest-neighbor query with given text

After you have stored and indexed embeddings in your database, the full range of `pgvector` query functionality becomes available to you.

To find the nearest semantic neighbors to a given piece of text, you can use the `embedding()` function to translate the text into a vector. In the same query, you apply this vector to the `pgvector` nearest-neighbor operator, `<->`, to find the database rows with the most semantically similar embeddings.

By default, the operator `<->` calculates the L2 distance between vectors. If you build the index using cosine distance and try to use the operator `<->`, then the index isn't used. The following example is meant for using an L2 distance function. Build your index using the same function.

Because `embedding()` returns a real array, you must explicitly cast the `embedding()` call to vector in order to use these values with `pgvector` operators.

Unset

```
  SELECT RESULT_COLUMNS FROM TABLE
    ORDER BY EMBEDDING_COLUMN
    <-> embedding('MODEL_IDVERSION_TAG', 'TEXT')::vector
    LIMIT ROW_COUNT
```

Replace the following:

- `RESULT_COLUMNS`: the columns to display from semantically similar rows.
- `TABLE`: the table containing the embedding to compare the text to.
- `EMBEDDING_COLUMN`: the column containing the stored embeddings.
- `MODEL_ID`: the ID of the model to query. We recommend that you specify the model version. You must use the same version that you used to build data for the embedding column. If you're using the Vertex AI Model Garden, then specify a `textembedding-gecko` or `textembedding-gecko-multilingual` version here. These are the cloud-based models that AlloyDB can use for text embeddings. For more information, see Text embeddings.
- Optional: `VERSION_TAG`: the version tag of the model to query. Prepend the tag with `@`. If you are using one of the `textembedding-gecko` models with Vertex AI, then specify one of the version tags listed in Model versions. Google strongly recommends specifying the version tag. Not doing so risks unexpected results. For more information, see "Use model version tags to avoid errors" in this guide.
- `TEXT`: the text you want to find the nearest stored semantic neighbors of.
- `ROW_COUNT`: the number of rows to return. Specify 1 if you want only the single best match.

To run this query with a stored embedding column that uses the `real[]` data type instead of `vector`, cast the column into the `vector` data type as well:

Unset

```
  SELECT RESULT_COLUMNS::vector FROM TABLE
    ORDER BY EMBEDDING_COLUMN
    <-> embedding('MODEL_IDVERSION_TAG', 'TEXT')::vector
    LIMIT ROW_COUNT
```

Preview the ScaNN algorithm

AlloyDB Omni version 15.5.1 and later includes a Technology Preview of `alloydb_scann`, a PostgreSQL extension developed by Google that implements a highly efficient nearest-neighbor index powered by [the ScaNN algorithm](#).

You can experiment with the `scann` index type as an alternative to index types provided by the `pgvector` extension, including the `ivf` index type whose use is described in "Create an optimized nearest-neighbor index" in this guide. The `scann` index type is compatible with columns using the `pgvector`-provided `vector` data type.

Because the `scann` index type is available as a Preview, we don't recommend applying it to production workloads.

To enable use of the `scann` index type, run the following `CREATE EXTENSION` DDL queries on a database running in an AlloyDB Omni cluster:

Unset

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE EXTENSION IF NOT EXISTS alloydb_scann;
```

To apply an index using the ScaNN algorithm to a column containing stored vector embeddings, run the following DDL query:

Unset

```
CREATE INDEX ON TABLE
  USING scann (EMBEDDING_COLUMN DISTANCE_FUNCTION)
  WITH (num_leaves=LEAVES_COUNT, quantizer='sq8');
```

Replace the following:

- `TABLE`: the table to add the index to.
- `EMBEDDING_COLUMN`: a column that stores `vector` data.
- `DISTANCE_FUNCTION`: the distance function to use with this index. Choose one of the following:
  - **L2 distance**: `l2`
  - **Dot product**: `dot_product`
  - **Cosine distance**: `cosine`
- `LEAVES_COUNT`: the number of partitions to apply to this index. For information on finding an optimal value for this parameter, see "Tune a `scann` index" in this guide.

To create this index on an embedding column that uses the `real[]` data type instead of `vector`, cast the column into the `vector` data type:

Unset

```
CREATE INDEX ON TABLE
  USING scann (CAST(EMBEDDING_COLUMN AS vector(DIMENSIONS))
DISTANCE_FUNCTION)
  WITH (num_leaves=LEAVES_COUNT, quantizer='sq8');
```

Replace `DIMENSIONS` with the dimensional width of the embedding column.

The samples for building a `scann` index require 50 * `LEAVES_COUNT` * `DIMENSIONS` * 4 bytes of memory. This is because each partition of a `scann` index contains 50 samples, and the index stores its own data in 4-byte floats. Before building a `scann` index, make sure that your database's `maintenance_work_mem` flag is set to a value sufficient for the memory required.

Optionally, you can set the following database flags to tune the behavior of the `alloydb_scann` extension:

`scann.num_leaves_to_search`

To improve the accuracy of queries made using the index, at the cost of query speed, increase this value prior to calling the queries. The maximum meaningful value is the value of `num_leaves` that you specify when creating the index. The default value is `1`.

`scann.pre_reordering_num_neighbors`

To help your query achieve higher recall, set this flag to a value greater than the number of neighbors returned by the query. Increasing this value does carry a performance cost.

To set the value of these flags, use the SET PostgreSQL command. After you create the index, you can run nearest-neighbor search queries that make use of the index by following the instructions in "Make a nearest-neighbor query with given text" in this guide.

Tune a scann index

To achieve both a high query-per-second rate (QPS) and a high recall with your nearest-neighbor queries, you must partition the tree of your `scann` index in a way that is most appropriate to your data and your queries. You do this by adjusting the values of the `num_leaves` index parameter and the `scann.num_leaves_to_search` database flag.

Through iterative tuning, you can find the optimal values of these variables for your workload. To tune your index, you build, test, adjust, and rebuild the index until your test queries reach the right balance of recall and QPS.

The following general recommendations apply when tuning your `scann` index parameters:

- Creating more partitions—also known as leaves—provides better recall. It also increases the size of the tree, and therefore the time required to build the index.
- For partitioning stability, each partition should have at least 100 data points, on average. For this reason, you shouldn't set the value of `num_leaves` to a number greater than the number of indexed rows divided by 100.
- For better performance, a nearest-neighbor query should spend most of its search time in leaf processing. To help ensure this, set `num_leaves` to a multiple of the square root of the indexed table's row count, as described by the following procedure.

To apply these recommendations to help you find the optimal values of `num_leaves` and `num_leaves_to_search` for your dataset, follow these steps:

1. Create the `scann` index with `num_leaves` set to the square root of the indexed table's row count.
2. Run your test queries, increasing the value of `scann.num_of_leaves_to_search`, until you achieve your target recall range–for example, 95%.
3. Take note of the ratio between `scann.num_leaves_to_search` and `num_leaves`.
4. If your QPS is too low when your queries achieve a target recall, then follow these steps:
    a. Recreate the index, increasing the value of `num_leaves` and `scann.num_leaves_to_search` according to the following guidance:
        - Set `num_leaves` to a larger factor of the square root of your row count. For example, if the index has `num_leaves` set to the square root of your row count, try setting it to double the square root. If it is already double, then try setting it to triple the square root.
        - Increase `scann.num_leaves_to_search` as needed in order to maintain its ratio with `num_leaves`, which you noted in an earlier step.
        - Don't set `num_leaves` to a value greater than the row count divided by 100.

    b. Run the test queries again. While doing so, you can experiment with reducing `scann.num_leaves_to_search`, finding a value that increases QPS while keeping your recall high. You can try different values of `scann.num_leaves_to_search` without rebuilding the index.

    c. Repeat this step until both the QPS and the recall range have reached acceptable values.

Use model version tags to avoid errors

Caution: To avoid inconsistent results from the `embedding()` function, always specify a stable embeddings model, including a version tag.

Google strongly recommends that you always use a stable version of your chosen embeddings model. For most models, this means explicitly setting a version tag.

Calling the `embedding()` function without specifying the version tag of the model is syntactically valid, but it is also error-prone.

If you omit the version tag when using a model in the Vertex AI Model Garden, then Vertex AI uses the latest version of the model. This might not be the latest stable version. For more information about available Vertex AI model versions, see [Model versions](#).

A given Vertex AI model version always returns the same `embedding()` response to given text input. If you don't specify model versions in your calls to `embedding()`, then a new published model version can abruptly change the returned vector for a given input, causing errors or other unexpected behavior in your applications.

To avoid these problems, always specify the model version.

Query your database using natural language

This section describes a Technology Preview available with AlloyDB Omni that lets you experiment with querying your database using natural language.

You can use AlloyDB Omni to preview a set of experimental features that allows your database-driven application to more securely execute natural-language queries from your application's users, such as "Where is my package?" or "Who is the top earner in each department?" AlloyDB Omni translates the natural-language input into a SQL query specific to your database, restricting the results only to what the user of your application is allowed to view.

The power and risks of natural-language queries

Large language models, such as [Gemini Pro](#), can enable your application to run database queries based on natural-language queries created by your application's end users. For example, a model with access to your application's database schema can take end-user input like the following and translate it into a SQL query:

Unset

```
What are the cheapest direct flights from Boston to Denver in July?
```

The SQL query might look something like this:

Unset

```
SELECT flight.id, flight.price, carrier.name, [...]
FROM [...]
WHERE [...]
ORDER BY flight.price ASC
LIMIT 10
```

Natural-language queries can provide your application a powerful tool for serving your users. However, this technology also comes with clear security risks that you must consider before you allow end users to run arbitrary queries on your database tables. Even if you have configured your application to connect to your database as a limited-access, read-only database user, an application that invites natural-language queries can be vulnerable to the following:

- Malicious users can submit prompt-injection attacks, trying to manipulate the underlying model to reveal all the data the application has access to.
- The model itself might generate SQL queries broader in scope than is appropriate, revealing sensitive data in response to even well-intentioned user queries.

Sanitize queries with parameterized secure views

To help mitigate the risks described in the previous section, Google has developed parameterized secure views, an experimental feature that you can preview using the techniques described in this section.

Parameterized secure views let you explicitly define the tables and columns that natural-language queries can pull data from, and add additional restrictions on the range of rows available to an individual application user. These restrictions let you tightly control the data that your application's users can see through natural-language queries, no matter how your users phrase these queries.

If you enable this Technology Preview, then you get access to an experimental extension developed by Google called `alloydb_ai_nl`. This extension provides the following features:

- Parameterized secure views, a variant of SQL views for restricting the range of data that a query can access.
- The `google_exec_param_query()` function, which lets you query your parameterized secure views.
- The `google_get_sql_current_schema()` function, which converts natural language queries into SQL queries of tables and views in your current schema.

The following sections describe how to use these features and demonstrate how they can work together.

Before you begin

Install AlloyDB Omni version 15.5.1 or later, including AI model integration. For more information, see "Install AlloyDB Omni with AlloyDB AI" in the AlloyDB Omni Installation Guide.

Set up your database for parameterized secure views

1. Connect to your AlloyDB Omni cluster using `psql`.
2. Edit the contents of your `postgresql.conf` file so that the value of the `shared_preload_libraries` directive includes `alloydb_ai_nl`. The edited directive looks similar to the following:

Unset

```
shared_preload_libraries='g_stats,google_job_scheduler,google_insigh
ts,pg_stat_statements,google_db_advisor,google_columnar_engine,alloy
db_ai_nl'
```

Note: If you are running the Preview of simplified installation method for AlloyDB Omni, then you might need to add the `shared_preload_libraries` directive to your `postgresql.conf` file, with `'alloydb_ai_nl'` as its entire value.

3. Stop AlloyDB Omni.
4. Start AlloyDB Omni.
5. Enable the `alloydb_ai_nl` extension:

Unset

```
CREATE EXTENSION google_ml_integration with version '1.3';

ALTER SYSTEM SET google_ml_integration.enable_model_support=on;
ALTER SYSTEM SET alloydb_ai_nl.enabled=on;
SELECT pg_reload_conf();
CREATE EXTENSION alloydb_ai_nl;
```

6. Register a new language model based on the Gemini Pro API with model endpoint management:

Unset

```
CALL google_ml.create_model(

    model_id => 'MODEL_ID',
    model_request_url =>
'https://us-central1-aiplatform.googleapis.com/v1/projects/PROJECT_ID/locations/us-central1/publishers/google/models/gemini-pro:streamGenerateContent',
    model_provider => 'google',
    model_auth_type => 'alloydb_service_agent_iam');
```

Replace the following:

- `MODEL_ID`: an ID to assign to this model. For more information about model endpoint management, see "Register and call remote AI models in AlloyDB Omni" in this guide.
- `PROJECT_ID`: the ID of your Google Cloud project.

7. Create a new database user. Don't grant it any permissions or roles yet. A subsequent step in this procedure grants the user the permissions that it requires.

   Warning: Don't skip this step. Always use a database user with carefully limited access to execute natural-language queries. Not doing so risks uncontrolled data exposure.

Parameterized secure views

A parameterized secure view works a lot like an ordinary PostgreSQL secure view: a stored `SELECT` statement, essentially. Parameterized secure views additionally allow you to require one or more named parameter values passed to the view when querying it, somewhat like bind variables with ordinary database queries.

For example, imagine running an application whose database tracks shipments of items to customers. A user logged into this application with the ID of `12345` types in the query `Where is my package?`. Using parameterized secure views, you can make sure that the following requirements apply to how AlloyDB for PostgreSQL executes this query:

- The query can read only the database columns that you have explicitly listed in your database's parameterized secure views. In this case, that might be certain columns in your `items`, `users`, and `shipments` tables.
- The query can read only the database rows associated with the user who asked the query. In this case, that might require that returned rows have a data relationship with the users table row whose `id` column value is `12345`.

Create a parameterized secure view

To create a parameterized secure view, use the PostgreSQL `CREATE VIEW` DDL command with the following attributes:

- Create the view with the `security_barrier` option.

- To restrict application users to seeing only the rows they're allowed to see, add required parameters using the `$@PARAMETER_NAME` syntax in the `WHERE` clause. A common case is checking the value of a column using `WHERE COLUMN = $@PARAMETER_NAME`.

The following example parameterized secure view allows access to three columns from a table named users, and limits the results only to rows where `users.id` matches a required parameter:

Unset

```
CREATE VIEW user_psv WITH (security_barrier) AS
SELECT
  username,
  full_name,
  birthday
FROM
  users
WHERE
  users.id = $@user_id;
```

The `SELECT` statements at the core of parameterized secure views can be as complex as the statements allowed by ordinary PostgreSQL views.

After you create a view, you must then grant the user you created earlier permission to run `SELECT` queries on the view:

Unset

```
GRANT SELECT ON VIEW_NAME TO NL_DB_USER;
```

Replace the following:

- `VIEW_NAME`: The name of the view that you created in the previous step.
- `NL_DB_USER`: The name of the database user that you have designated to execute natural language queries.

Query a parameterized secure view

Despite their similarity to ordinary PostgreSQL views, you can't query parameterized secure views directly. Instead, you use the `google_exec_param_query()` function provided by the `alloydb_ai_nl` extension. The function has the following syntax:

Unset

```
SELECT * FROM
alloydb_ai_nl.google_exec_param_query(
    query => SQL_QUERY
    param_names => ARRAY [PARAMETER_NAMES],
    param_values => ARRAY [PARAMETER_VALUES]
)
```

Replace the following:

- `SQL_QUERY`: A SQL query whose `FROM` clause refers to one or more parameterized secure views.
- `PARAMETER_NAMES`: A list of parameter names to pass in, as strings.
- `PARAMETER_VALUES`: A list of parameter values to pass in. This list must be of the same size as the `param_names` list. The order of the values matches the order of the names.

The function returns a table of JSON objects. Each row in the table is equivalent to the `row_to_json()` value of the original query result row.

In typical use, the value of the query argument is generated not by your own code, but instead by an AI model that you have integrated your AlloyDB for PostgreSQL database with.

The following example shows how you might query a parameterized secure view in Python, and then display its results. It builds on the `user_psv` example view from the previous section:

Unset

```
# Assume a get_user_id() function that returns the ID of the current
# application user.
app_user_id = get_user_id()

pool = await asyncpg.create_pool(
    host=INSTANCE_IP
    user=NL_DB_USER
    password=NL_DB_PASSWORD
    database=DB_NAME
)

table_name = "user_psv"
```

```
query = f"""
    SELECT
      full_name,
      birthday
    FROM
      {table_name}
"""
params = {
    "user_id": app_user_id
}

param_query = f"""
SELECT * FROM alloydb_ai_nl.google_exec_param_query(
  query => '{query}',
  param_names => $1,
  param_values => $2
);
"""

sql_results = await pool.execute(
    param_query,
    params.keys(),
    params.values()
)

for row in sql_results:
    print(json.loads(row))
```

Execute a natural-language query

Note: The functions described by this section are included in this Technology Preview to let you test and experiment with natural-language processing using parameterized secure views. The functions don't provide context awareness beyond basic schema data or disambiguation capabilities.

Executing a natural-language query using parameterized secure views is a two-step process:

1. As a database user with only `SELECT` access to the appropriate parameterized secure views, convert the natural language query to SQL using a large language model.
2. Use the `google_exec_param_query()` function to process the SQL, binding it to parameter values appropriate to the current user session.

The following sections describe these steps in further detail.

Convert natural language to SQL

To translate natural-language input into SQL, use the `google_get_sql_current_schema()` function that is included with the parameterized secure views Technology Preview:

Unset

```
SELECT alloydb_ai_nl.google_get_sql_current_schema(
  sql_text => 'NL_TEXT',
  model_id => 'MODEL_ID',
  prompt_text => 'HINT_TEXT'
);
```

Replace the following:

- `NL_TEXT`: The natural-language text to turn into a SQL query.
- `MODEL_ID`: The ID of the model that you registered with the model catalog when setting up your database for parameterized secure views.
- `HINT_TEXT`: Additional information about the database schema, expressed in natural language. This lets you give the model additional hints about important aspects of the schema that it might not extract only by analyzing the table, column, and relationship structures. As an example: When joining flights and seats, be sure to join on `flights.id = seats.flight_id`.

Warning: Your application must run `google_get_sql_current_schema()` as the database user that you created specifically to execute natural-language queries. Not doing so risks uncontrolled data exposure.

The output of the function is a string containing a SQL query.

Run the converted SQL using parameters

After converting the natural-language query to SQL, you can call `google_exec_param_query()` as described earlier in this section, passing in any parameters that your parameterized secure views might need.

The function works if you pass it more parameters than it needs with a given query, so you can call it with all of the parameters used by all of the parameterized secure views that your application has values for. The function throws an exception if it tries to run a query requiring an undefined parameter.

An example of executing a natural-language query

This section demonstrates a complete flow from natural-language input to SQL result set. The code samples show the underlying SQL queries and functions that an application runs.

For this example flow, assume the following about your application:

- Your database-driven application tracks product shipments to customers.
- You have registered a Gemini Pro-based model named `my-gemini-model` in the Model Catalog.
- You have defined a parameterized secure view in your database named `shipment_view`.
    - The view selects data from several tables relevant to shipments to customers.
    - The view requires a `user_id` parameter, whose value is the ID of an end user of the application.

The following shows the example flow:

1. An end user whose application user ID is `12345` types `Where is my package?` into your web application.
2. Your application calls `google_get_sql_current_schema()` to translate the input into SQL:

Unset

```
SELECT alloydb_ai_nl.google_get_sql_current_schema(

  sql_text => 'Where is my package?'
  model_id => 'my-gemini-model'
);
```

3. This call returns a string containing a single SQL `SELECT` query. The query is limited only to the parameterized secure views visible to the database user that you created to work with parameterized secure views. The SQL generated from `Where is my package?` might resemble the following:

Unset

```
SELECT current_location, ship_date, ship_eta FROM shipment_view;
```

4. Because shipment_view is a parameterized secure view and not an ordinary PostgreSQL view, your application must use google_exec_param_query() to securely run the query with the user_id parameter that it requires, as shown in the next step.
5. Your application passes the SQL to google_exec_param_query(), along with the parameters that constrain the output. In our example, that is the ID of the application end user who provided the input:

Unset

```
SELECT * FROM

alloydb_ai_nl.google_exec_param_query(
    query => 'SELECT current_location, ship_date, ship_eta FROM
shipment_view',
    param_names => ['user_id'],
    param_values => ['12345']
);
```

6. The output is a SQL result set, expressed as JSON data.
7. Your application handles the JSON data as needed.

Database design for natural-language handling

Caution: Sensitive information or categories in your database might cause SQL queries generated from natural-language prompts to reflect biases from the model's training data. Carefully review your schema prior to applying the techniques described in this section.

The google_get_sql_current_schema() function provided with this Technology Preview serves mainly to demonstrate the functionality of parameterized secure views, giving you an early opportunity to experiment with this developing technology. As with any Preview, you shouldn't apply this function to an application in production.

With that in mind, you can apply the advice in this section to improve the quality of google_get_sql_current_schema() output during your experimentation with it.

Design your schema for human comprehension

In general, give your database structures names and comments clear enough to allow a typical human developer to infer the purpose of its tables, columns, and relationships. This clarity can help a large language model generate more accurate SQL queries based on your schema.

Use descriptive names

Prefer descriptive names for tables, columns, and relationships. Avoid abbreviations or acronyms. For example, the model works better with a table named `users` than with one named `u`.

If it's not feasible to rename existing data structures, provide hints to the model using the `prompt_text` argument when calling `google_get_sql_current_schema()`.

Use specific data types

The model can make better inferences about your data if you use more specific data types with your columns. For example, if you use a column exclusively to store true-or-false values, then use a `boolean` data type with `true` and `false` instead of an `integer` with `1` and `0`.

Roll back with caution after enabling the Preview

Caution: If you have enabled this Technology Preview, then rolling back AlloyDB Omni without fully uninstalling the Preview can result in errors.

If you have enabled the parameterized secure views Technology Preview on your database, but then decide to roll back AlloyDB Omni to a version before 15.5.0, then you must take a few manual cleanup steps before downgrading.

If you don't take these steps, then any attempt to query, modify, or drop a parameterized secure view results in a SQL error. This includes queries on your database's view catalog that would otherwise include parameterized secure views in their results, such as `SELECT * FROM pg_views`.

To completely remove this Technology Preview from your database before an AlloyDB Omni rollback, follow these steps:

1. In `psql`, use the `DROP VIEW` command to delete every parameterized secure view in your database.
2. In `psql`, use the `DROP EXTENSION` command to disable the `alloydb_ai_nl` extension on your database.
3. In your `postgresql.conf` file, remove the reference to `alloydb_ai_nl` from the `shared_preload_libraries` directive.

For more information, see Uninstall AlloyDB Omni.

Register and call remote AI models in AlloyDB Omni

This section describes a Preview that lets you experiment with registering AI models and invoking predictions with model endpoint management in AlloyDB Omni. To use AI models in production environments, see Build generative AI applications using AlloyDB AI and Work with vector embeddings.

To register remote models with AlloyDB, see Register and call remote AI models in AlloyDB.

Overview

The *model endpoint management* Preview lets you register and manage your AI models metadata in your database cluster, and then interact with the models using SQL queries. It provides the `google_ml_integration` extension that includes functions to add and register the metadata related to the models, and then use the models to generate vector embeddings or invoke predictions.

Some of the example model types that you can register using model endpoint management are as follows:

- Vertex AI text embedding models
- Embedding models provided by third-party providers, such as Anthropic, Hugging Face, or OpenAI
- Custom-hosted text embedding models
- Generic models with a JSON-based API—for example, the `facebook/bart-large-mnli` model hosted on Hugging Face or the `gemini-pro` model from the Vertex AI Model Garden

How it works

You can use model endpoint management to register models that comply to the following:

- Model input and output supports JSON format.
- Model can be called using the REST protocol.

When you register a model with model endpoint management, it registers each model with a unique model ID that you provided as a reference to the model. You can use this model ID to query models:

- Generate embeddings to translate text prompts to numerical vectors. You can store generated embeddings as vector data when the pgvector extension is enabled in the database. For more information, see Query and index embeddings with pgvector.
- Invoke predictions to call a model using SQL within a transaction.

Your applications can access the model endpoint management using the `google_ml_integration` extension. This extension provides the following functions:

- The `google_ml.create_model()` SQL function, which is used to register the model metadata that is used in the prediction or embedding function.
- The `google_ml.create_sm_secret()` SQL function, which uses secrets in the Google Cloud Secret Manager, where the API keys are stored.

- The `google_ml.embedding()` SQL function, which is a prediction function that generates text embeddings.
- The `google_ml.predict_row()` SQL function that generates predictions when you call generic models that support JSON input and output format.
- Other helper functions that handle generating custom URL, generating HTTP headers, or passing transform functions for your generic models.
- Functions to manage the registered models and secrets.

Key concepts

Before you start using the model endpoint management, understand the concepts required to connect to and use the models.

Model provider

Model provider indicates the supported model hosting providers. The following table shows the model provider value that you must set based on the model provider that you use:

Model provider	**Set in function as…**
Vertex AI	`google`
Hugging Face models	`custom`
Anthropic models	`custom`
Other models	`custom`
OpenAI	`open_ai`

The default model provider is custom.

Based on the provider type, the supported authentication method differs. The Vertex AI models use the AlloyDB service account to authenticate, while other providers can use the Secret Manager to authenticate. For more information, see "Set up authentication" in this guide.

Model type

Model type indicates the type of the AI model. The extension supports text embedding as well as any generic model type. The supported model types that you can set when you register a model are `text-embedding` and `generic`. Setting the model type is optional when you register generic models because `generic` is the default model type.

Text embedding models with built-in support

The model endpoint management provides built-in support for all versions of the `textembedding-gecko` model by Vertex AI and the `text-embedding-ada-002` model by OpenAI. To register these models, use the `google_ml.create_model()` function. AlloyDB automatically sets up default transform functions for these models.

The model type for these models is `text-embedding`.

Other text embedding models

For other text embedding models, you need to create transform functions to handle the input and output formats that the model supports. Optionally, you can use the HTTP header generation function that generates custom headers required by your model.

The model type for these models is `text-embedding`.

Generic models

The model endpoint management also supports registering of all other model types apart from text embedding models. To invoke predictions for generic models, use the `google_ml.predict_row()` function. You can set model metadata, such as a request endpoint and HTTP headers that are specific to your model.

You cannot pass transform functions when you are registering a generic model. Ensure that when you invoke predictions the input to the function is in the JSON format, and that you parse the JSON output to derive the final output.

The model type for these models is `generic`.

Authentication

Auth types indicate the authentication type that you can use to connect to the model endpoint management using the `google_ml_integration` extension. Setting authentication is optional and is required only if you need to authenticate to access your model.

For Vertex AI models, the AlloyDB service account is used for authentication. For other models, the API key or bearer token that is stored as a secret in the Secret Manager can be used with the `google_ml.create_sm_secret()` SQL function.

The following table shows the auth types that you can set:

Authentication method	**Set in function as…**	**Model provider**

AlloyDB service agent	`alloydb_service_agent_iam`	Vertex AI provider
Secret Manager	`secret_manager`	Third-party providers, such as Anthropic, Hugging Face, or OpenAI

Prediction functions

The `google_ml_integration` extension includes the following prediction functions:

`google_ml.embedding()`

Used to call registered text embedding models to generate embeddings. It includes built-in support for the `textembedding-gecko` model by Vertex AI and the `text-embedding-ada-002` model by OpenAI.

For text embedding models without built-in support, the input and output parameters are unique to a model and need to be transformed for the function to call the model. Create a transform input function to transform input of the prediction function to the model specific input, and a transform output function to transform model specific output to the prediction function output.

`google_ml.predict_row()`

Used to call registered generic models, as long as they support JSON-based API, to invoke predictions.

Transform functions

Transform functions modify the input to a format that the model understands, and convert the model response to the format that the prediction function expects. The transform functions are used when registering the `text-embedding` models without built-in support. The signature of the transform functions depends on the prediction function for the model type.

You cannot use transform functions when registering `generic` models.

The following shows the signatures for the prediction function for text embedding models:

Unset

```
// define custom model specific input/output transform functions.
CREATE OR REPLACE FUNCTION input_transform_function(model_id
VARCHAR(100), input_text TEXT) RETURNS JSON;
```

```
CREATE OR REPLACE FUNCTION output_transform_function(model_id
VARCHAR(100), response_json JSON) RETURNS real[];
```

For more information about how to create transform functions, see [Transform functions example](#).

HTTP header generation function

The HTTP header generation function generates the output in JSON key value pairs that are used as HTTP headers. The signature of the prediction function defines the signatures of the header generation function.

The following example shows the signature for the `google_ml.embedding()` prediction function.

Unset

```
CREATE OR REPLACE FUNCTION generate_headers(model_id VARCHAR(100),
input TEXT) RETURNS JSON;
```

For the `google_ml.predict_row()` prediction function, the signature is as follows:

Unset

```
CREATE OR REPLACE FUNCTION generate_headers(model_id VARCHAR(100),
input JSON) RETURNS JSON;
```

For more information about how to create a header generation function, see [Header generation function example](#).

Register a model with model endpoint management

To invoke predictions or generate embeddings using a model, register the model with model endpoint management.

For more information about the `google_ml.create_model()` function, see [model endpoint management reference](#).

Before you register a model with model endpoint management, you must enable the `google_ml_integration` extension and set up authentication based on the model provider, if your model requires authentication.

Make sure that you access your database with the `postgres` default username.

Enable the extension

You must add and enable the `google_ml_integration` extension before you can start using the associated functions. Model endpoint management requires that the `google_ml_integration` version 1.3 extension is installed.

1. Connect to your database using `psql`.
2. Optional: If the `google_ml_integration` extension is already installed, alter it to update the version to `1.3`:

Unset

```
    ALTER EXTENSION google_ml_integration UPDATE TO '1.3'
```

3. Add the `google_ml_integration` version 1.3 extension using `psql`:

Unset

```
  CREATE EXTENSION google_ml_integration VERSION '1.3';
```

4. Optional: Grant permission to a non-super PostgreSQL user to manage model metadata:

Unset

```
  GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA
google_ml TO NON_SUPER_USER;
```

   Replace `NON_SUPER_USER` with the non-super PostgreSQL username.

5. Enable model endpoint management on your database:

Unset

```
  ALTER SYSTEM SET google_ml_integration.enable_model_support=on;

  SELECT pg_reload_conf();
```

Set up authentication

The following sections show how to set up authentication before adding a Vertex AI model or models by other providers.

Set up authentication for Vertex AI

To use the Google Vertex AI models, you must add Vertex AI permissions to the service account that you used while installing AlloyDB Omni. For more information, see Configure your AlloyDB Omni installation to query cloud-based models.

Set up authentication for other model providers

For all models except Vertex AI models, you can store your API keys or bearer tokens in Secret Manager. This step is optional if your model doesn't handle authentication through Secret Manager—for example, if your model uses HTTP headers to pass authentication information or doesn't use authentication at all.

This section explains how to set up authentication if you are using Secret Manager.

To create and use an API key or a bearer token, complete the following steps:

1. Create the secret in Secret Manager. For more information, see Create a secret and access a secret version. The secret name and the secret path is used in the `google_ml.create_sm_secret()` SQL function.
2. Grant permissions to the AlloyDB cluster to access the secret.

Unset

```
gcloud secrets add-iam-policy-binding 'SECRET_ID' \
    --member="serviceAccount:SERVICE_ACCOUNT_ID" \
    --role="roles/secretmanager.secretAccessor"
```

Replace the following:

- `SECRET_ID`: the secret ID in Secret Manager.
- `SERVICE_ACCOUNT_ID`: the ID of the service account that you created in the previous step. Ensure that this is the same account you used during AlloyDB Omni installation. This includes the full `PROJECT_ID` `.iam.gserviceaccount.com` suffix—for example, `my-service@my-project.iam-gserviceaccount.com`.

  You can also grant this role to the service account at the project level. For more information, see Add Identity and Access Management policy binding.

Text embedding models with built-in support

This section shows how to register models that the model endpoint management provides built-in support for.

Vertex AI embedding models

The model endpoint management provides built-in support for all versions of the `text-embedding-gecko` model by Vertex AI. Use the qualified name to set the model version to either `textembedding-gecko@001` or `textembedding-gecko@002`.

Since the `textembedding-gecko` and `textembedding-gecko@001` model endpoint ID is pre-registered with model endpoint management, you can directly use them as the model ID. For these models, the extension automatically sets up default transform functions.

To register the `textembedding-gecko@002` model version, complete the following steps:

1. Set up [AlloyDB Omni to query cloud-based Vertex AI models](#).
2. [Connect to your database using `psql`](#).
3. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
4. Call the create model function to add the `textembedding-gecko@002` model:

Unset

```
CALL

  google_ml.create_model(
    model_id => 'textembedding-gecko@002',
    model_provider => 'google',
    model_qualified_name => 'textembedding-gecko@002',
    model_type => 'text_embedding',
    model_auth_type => 'alloydb_service_agent_iam');

  The request URL that the function generates refers to the project
associated with the AlloyDB Omni service account. If you want to
refer to another project, then ensure that you specify the
`model_request_url` explicitly.
```

Open AI text embedding model

The model endpoint management provides built-in support for the `text-embedding-ada-002` model by OpenAI. The `google_ml_integration` extension automatically sets up default transform functions and invokes calls to the remote model.

The following example adds the `text-embedding-ada-002` OpenAI model.

1. Connect to your database using `psql`.
2. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
3. Add the OpenAI API key as a secret to the Secret Manager for authentication. See "Set up authentication for other model providers" in this guide.
4. Call the secret stored in the Secret Manager:

Unset

```
CALL

google_ml.create_sm_secret(
  secret_id => 'SECRET_ID',
  secret_path =>
'projects/PROJECT_ID/secrets/SECRET_MANAGER_SECRET_ID/versions/VERSION_NUMBER');
```

Replace the following:

- `SECRET_ID`: the secret ID that you set and is subsequently used when registering a model—for example, `key1`.
- `SECRET_MANAGER_SECRET_ID`: the secret ID set in Secret Manager when you created the secret.
- `PROJECT_ID`: the ID of your Google Cloud project.
- `VERSION_NUMBER`: the version number of the secret ID.

5. Call the create model function to register the `text-embedding-ada-002` model:

Unset

```
CALL

  google_ml.create_model(
    model_id => 'MODEL_ID',
    model_provider => 'open_ai',
    model_type => 'text_embedding',
    model_qualified_name => 'text-embedding-ada-002',
    model_auth_type => 'secret_manager',
    model_auth => 'SECRET_ID');
```

Replace the following:

- `MODEL_ID`: a unique ID for the model that you define. This model ID is referenced for metadata that the model needs to generate embeddings or invoke predictions.
- `SECRET_ID`: the secret ID you used earlier in the `google_ml.create_sm_secret()` procedure.

For more information about generating embeddings, see "Generate vector embeddings with model endpoint management" in this guide.

Other text embedding models

This section shows how to register any custom-hosted text embedding model or text embedding models provided by model hosting providers. Based on your model metadata, you might need to add transform functions, generate HTTP headers, or define endpoints.

Custom-hosted text embedding model

This section shows how to register a custom-hosted model along with creating transform functions, and optionally, custom HTTP headers. AlloyDB Omni supports all custom-hosted models regardless of where they are hosted.

The following example adds the `custom-embedding-model` custom model hosted by Cymbal. The `cymbal_text_input_transform` and `cymbal_text_output_transform` transform functions are used to transform the input and output format of the model to the input and output format of the prediction function.

To register custom-hosted text embedding models, complete the following steps:

1. Connect to your database using `psql`.
2. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
3. Optional: Add the API key as a secret to the Secret Manager for authentication. See "Custom-hosted text embedding model" in this guide.
4. Call the secret stored in the Secret Manager:

Unset

```
CALL

  google_ml.create_sm_secret(
    secret_id => 'SECRET_ID',
    secret_path =>
'projects/project-id/secrets/SECRET_MANAGER_SECRET_ID/versions/VERSIO
N_NUMBER');
```

Replace the following:

- `SECRET_ID`: the secret ID that you set and is subsequently used when registering a model—for example, `key1`.
- `SECRET_MANAGER_SECRET_ID`: the secret ID set in Secret Manager when you created the secret.
- `PROJECT_ID`: the ID of your Google Cloud project.
- `VERSION_NUMBER`: the version number of the secret ID.

  Note: Secret Manager generates an Authorization: Bearer `SECRET_VALUE_FROM_SECRET_MANAGER` header for authentication by default. If this format matches your model's authorization bearer token format, then you don't have to generate auth headers using the header generation function.

5. Create the input and output transform functions based on the following signature for the prediction function for text embedding models. For more information about how to create transform functions, see [Transform functions example](#).

   The following are example transform functions that are specific to the `custom-embedding-model` text embedding model:

Unset

```
-- Input Transform Function corresponding to the custom model

CREATE OR REPLACE FUNCTION cymbal_text_input_transform(model_id
VARCHAR(100), input_text TEXT)
RETURNS JSON
LANGUAGE plpgsql
AS $$
DECLARE
  transformed_input JSON;
  model_qualified_name TEXT;
BEGIN
  SELECT json_build_object('prompt',
json_build_array(input_text))::JSON INTO transformed_input;
  RETURN transformed_input;
END;
$$;
-- Output Transform Function corresponding to the custom model
CREATE OR REPLACE FUNCTION cymbal_text_output_transform(model_id
VARCHAR(100), response_json JSON)
RETURNS REAL[]
LANGUAGE plpgsql
AS $$
DECLARE
  transformed_output REAL[];
BEGIN
  SELECT ARRAY(SELECT json_array_elements_text(response_json->0))
INTO transformed_output;
  RETURN transformed_output;
END;
$$;
```

6. Call the create model function to register the custom embedding model:

Unset

```
CALL

  google_ml.create_model(
    model_id => 'MODEL_ID',
    model_request_url => 'REQUEST_URL',
    model_provider => 'custom',
    model_type => 'text_embedding',
    model_auth_type => 'secret_manager',
    model_auth_id => 'SECRET_ID',
    model_qualified_name => 'MODEL_QUALIFIED_NAME',
    model_in_transform_fn => 'cymbal_text_input_transform',
    model_out_transform_fn => 'cymbal_text_output_transform');
```

Replace the following:

- `MODEL_ID`: required. A unique ID for the model that you define—for example, `custom-embedding-model`. This model ID is referenced for metadata that the model needs to generate embeddings or invoke predictions.
- `REQUEST_URL`: required. The model specific endpoint when adding custom text embedding and generic models—for example, `https://cymbal.com/models/text/embeddings/v1`.
- `MODEL_QUALIFIED_NAME`: required if your model uses a qualified name. The fully qualified name in case the model has multiple versions.
- `SECRET_ID`: the secret ID you used earlier in the `google_ml.create_sm_secret()` procedure.

OpenAI Text Embedding 3 Small and Large models

You can register the OpenAI `text-embedding-3-small` and `text-embedding-3-large` models using the embedding prediction function and transform functions specific to the model. The following example shows how to register the OpenAI `text-embedding-3-small` model.

To register the `text-embedding-3-small` embedding model, do the following:

1. Connect to your database using `psql`.
2. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
3. Add the OpenAI API key as a secret to the Secret Manager for authentication. See "Set up authentication for other model providers" in this guide. If you have already created a secret for any other OpenAI model, then you can reuse the same secret.
4. Call the secret stored in the Secret Manager:

Unset

```
CALL

  google_ml.create_sm_secret(
    secret_id => 'SECRET_ID',_
    secret_path =>
'projects/project-id/secrets/SECRET_MANAGER_SECRET_ID/versions/VERSION_NUMBER');
```

Replace the following:

- `SECRET_ID`: the secret ID that you set and is subsequently used when registering a model.
- `SECRET_MANAGER_SECRET_ID`: the secret ID set in Secret Manager when you created the secret.

   - `PROJECT_ID`: the ID of your Google Cloud project.
   - `VERSION_NUMBER`: the version number of the secret ID.

5. Create the input and output transform functions based on the following signature for the prediction function for text embedding models. For more information about how to create transform functions, see [Transform functions example](). To learn about the input and output formats that OpenAI models expect, see [Embeddings]() in the OpenAI documentation.

   The following are example transform functions for the `text-embedding-ada-002`, `text-embedding-3-small`, and `text-embedding-3-large` OpenAI text embedding models.

Unset

```
-- Input Transform Function corresponding to openai_text_embedding
model family

CREATE OR REPLACE FUNCTION openai_text_input_transform(model_id
VARCHAR(100), input_text TEXT)
RETURNS JSON
LANGUAGE plpgsql
AS $$
#variable_conflict use_variable
DECLARE
  transformed_input JSON;
  model_qualified_name TEXT;
BEGIN
  SELECT google_ml.model_qualified_name_of(model_id) INTO
model_qualified_name;
  SELECT json_build_object('input', input_text, 'model',
model_qualified_name)::JSON INTO transformed_input;
  RETURN transformed_input;
END;
$$;

-- Output Transform Function corresponding to openai_text_embedding
model family
CREATE OR REPLACE FUNCTION openai_text_output_transform(model_id
VARCHAR(100), response_json JSON)
RETURNS REAL[]
LANGUAGE plpgsql
```

```
AS $$
DECLARE
  transformed_output REAL[];
BEGIN
  SELECT ARRAY(SELECT
json_array_elements_text(response_json->'data'->0->'embedding'))
INTO transformed_output;
  RETURN transformed_output;
END;
$$;
```

6. Call the create model function to register the `text-embedding-3-small` embedding model:

Unset

```
CALL

  google_ml.create_model(
    model_id => 'MODEL_ID',
    model_provider => 'open_ai',
    model_type => 'text_embedding',
    model_auth_type => 'secret_manager',
    model_auth_id => 'SECRET_ID',
    model_qualified_name => 'text-embedding-3-small',
    model_in_transform_fn => 'openai_text_input_transform',
    model_out_transform_fn => 'openai_text_output_transform');
```

Replace the following:

- `MODEL_ID`: a unique ID for the model that you define—for example `openai-te-3-small`. This model ID is referenced for metadata that the model needs to generate embeddings or invoke predictions.
- `SECRET_ID`: the secret ID you used earlier in the `google_ml.create_sm_secret()` procedure.

For more information, see "Generate vector embeddings with model endpoint management" in this guide.

Generic models

This section shows how to register any generic model that is available on a hosted model provider such as Hugging Face, OpenAI, Vertex AI, or any other provider. This section shows examples to register a generic model hosted on Hugging Face and a generic `gemini-pro` model from Vertex AI Model Garden, which doesn't have built-in support.

You can register any generic model as long as the input and output is in the JSON format. Based on your model metadata, you might need to generate HTTP headers or define endpoints.

Generic model on Hugging Face

The following example adds the `facebook/bart-large-mnli` custom classification model hosted on Hugging Face.

1. Connect to your database using `psql`.
2. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
3. Add the bearer token as a secret to the Secret Manager for authentication. See "Set up authentication for other model providers" in this guide.
4. Call the secret stored in Secret Manager:

Unset

```
CALL

  google_ml.create_sm_secret(
    secret_id => 'SECRET_ID',
    secret_path =>
'projects/project-id/secrets/SECRE_MANAGER_SECRET_ID/versions/VERSION_NUMBER');
```

Replace the following:

- `SECRET_ID`: the secret ID that you set and that's subsequently used when registering a model.
- `SECRET_MANAGER_SECRET_ID`: the secret ID set in Secret Manager when you created the secret.
- `PROJECT_ID`: the ID of your Google Cloud project.
- `VERSION_NUMBER`: the version number of the secret ID.

5. Call the create model function to register the `facebook/bart-large-mnli` model:

Unset

```
CALL
```

```
  google_ml.create_model(
    model_id => 'MODEL_ID',
    model_provider => 'custom',
    model_request_url => 'REQUEST_URL',
    model_qualified_name => 'MODEL_QUALIFIED_NAME',
    model_auth_type => 'secret_manager',
    model_auth_id => 'SECRET_ID');
```

Replace the following:

- `MODEL_ID`: a unique ID for the model that you define—for example, `custom-classification-model`. This model ID is referenced for metadata that the model needs to generate embeddings or invoke predictions.
- `REQUEST_URL`: the model specific endpoint when adding custom text embedding and generic models—for example, `https://api-inference.huggingface.co/models/facebook/bart-large-mnli`.
- `MODEL_QUALIFIED_NAME`: the fully qualified name of the model version—for example, `facebook/bart-large-mnli`.
- `SECRET_ID`: the secret ID you used earlier in the `google_ml.create_sm_secret()` procedure.

Gemini model

Ensure that you set up AlloyDB Omni to query cloud-based Vertex AI models.

The following example adds the `gemini-1.0-pro` model from the Vertex AI Model Garden.

1. Connect to your database using `psql`.
2. Create and enable the `google_ml_integration` extension. See "Enable the extension" in this guide.
3. Call the create model function to register the `gemini-1.0-pro` model:

Unset

```
CALL

  google_ml.create_model(
    model_id => 'MODEL_ID',
    model_request_url =>
'https://us-central1-aiplatform.googleapis.com/v1/projects/PROJECT_I
D/locations/us-central1/publishers/google/models/gemini-1.0-pro:stre
amGenerateContent',
    model_provider => 'google',
    model_auth_type => 'alloydb_service_agent_iam');
```

Replace the following:

- `MODEL_ID`: a unique ID for the model that you define—for example, `gemini-1`. This model ID is referenced for metadata that the model needs to generate embeddings or invoke predictions.
- `PROJECT_ID`: the ID of your Google Cloud project.

For more information, see "Invoke predictions with model endpoint management" in this guide.

Generate vector embeddings with model endpoint management

This section describes a preview that lets you experiment with registering AI models and invoking predictions with model endpoint management. For information about using AI models in production environments, see Build generative AI applications using AlloyDB AI and Work with vector embeddings.

After the models are added and registered with model endpoint management, you can reference them using the model ID to generate embeddings.

Before you begin

Make sure that you have registered your model with model endpoint management. For more information, see "Register a model with model endpoint management" in this guide.

Generate embeddings

Use the `google_ml.embedding()` SQL function to call the registered models with the text embedding model type to generate embeddings.

To call the model and generate embeddings, use the following SQL query:

Unset

```
SELECT
  google_ml.embedding(
    model_id => 'MODEL_ID',
    content => 'CONTENT');
```

Replace the following:

- `MODEL_ID`: the model ID that you defined when registering the model.
- `CONTENT`: the text to translate into a vector embedding.

Examples

Some examples for generating embeddings using registered models are listed in this section.

Text embedding models with in-built support

To generate embeddings for a registered `textembedding-gecko@002` model, run the following statement:

Unset

```
    SELECT
      google_ml.embedding(
        model_id => 'textembedding-gecko@002',
        content => 'AlloyDB is a managed, cloud-hosted SQL database
service');
```

To generate embeddings for a registered `text-embedding-ada-002` model by OpenAI, run the following statement:

Unset

```
    SELECT
      google_ml.embedding(
        model_id => 'text-embedding-ada-002',
        content => 'e-mail spam');
```

Other text embedding models

To generate embeddings for a registered `text-embedding-3-small` or `text-embedding-3-large` models by OpenAI, run the following statement:

Unset

```
  SELECT
    google_ml.embedding(
      model_id => 'text-embedding-3-small',
      content => 'Vector embeddings in AI');
```

Invoke predictions with model endpoint management

This section describes a preview that lets you experiment with registering AI models and invoking predictions with model endpoint management. For using AI models in production environments, see Build generative AI applications using AlloyDB AI.

After the models are added and registered in model endpoint management, you can reference them using the model ID to invoke predictions.

Before you begin

Make sure that you have registered your model with model endpoint management. For more information, see "Register a model with model endpoint management" in this guide.

Invoke predictions for generic models

Use the `google_ml.predict_row()` SQL function to call a registered generic model to invoke predictions. You can use google_ml.predict_row() function with any model type.

Unset

```
SELECT
  google_ml.predict_row(
    model_id => 'MODEL_ID',
    request_body => 'REQUEST_BODY');
```

Replace the following:

- `MODEL_ID`: the model ID you defined when registering the model.
- `REQUEST_BODY`: the parameters to the prediction function, in JSON format.

Examples

Some examples for invoking predictions using registered models are listed in this section.

To generate predictions for a registered `gemini-pro` model, run the following statement:

Unset

```
    SELECT
        json_array_elements(
        google_ml.predict_row(
            model_id => 'gemini-pro',
            request_body => '{
        "contents": [
            {
                "role": "user",
                "parts": [
                    {
                        "text": "For TPCH database schema as
mentioned here
https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.
pdf , generate a SQL query to find all supplier names which are
located in the India nation."
                    }
                ]
            }
        ]
        }'))-> 'candidates' -> 0 -> 'content' -> 'parts' -> 0 ->
'text';
```

To generate predictions for a registered `facebook/bart-large-mnli` model on Hugging Face, run the following statement:

Unset

```
  SELECT
  google_ml.predict_row(
    model_id => 'facebook/bart-large-mnli',
    request_body =>
      '{
       "inputs": "Hi, I recently bought a device from your company
but it is not working as advertised and I would like to get
reimbursed!",
    "parameters": {"candidate_labels": ["refund", "legal", "faq"]}}
      ');
```