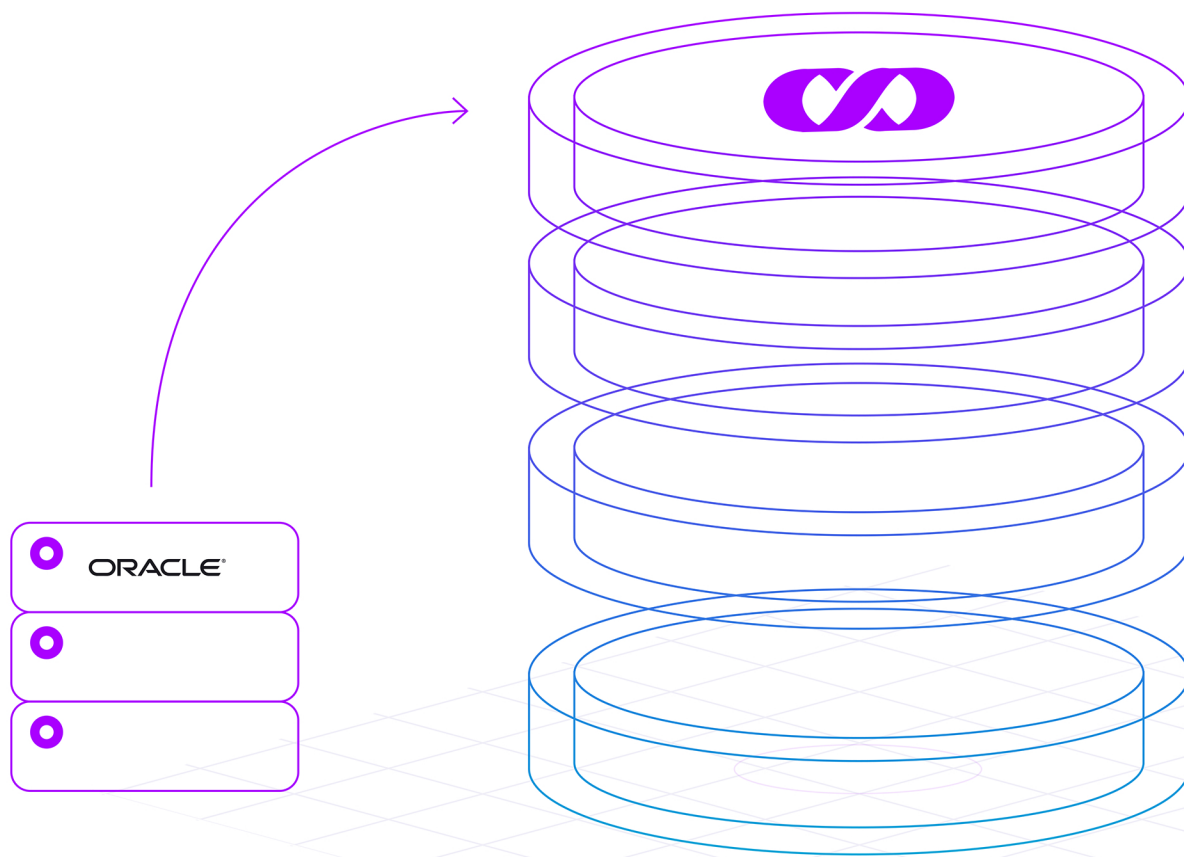


Oracle to MemSQL Migration Overview

Eric Hanson, Principal Product Manager

Krishna Manoharan, Director of Solution Architecture

David Anderson, Principal Sales Engineer



Abstract

This paper describes how you can migrate applications running on an Oracle database to MemSQL. MemSQL is a powerful, scalable, modern RDBMS that can run on-premises and in the cloud. It supports high-performance operational analytics, transaction processing, and traditional analytical applications.

Moving all but the largest and most complex applications from Oracle to MemSQL is feasible because of MemSQL's high performance and broad feature set. MemSQL features that are important to Oracle developers include ANSI SQL, a broad range of data types, primary and secondary indexes, unique constraints, rowstore and columnstore storage structures, transactions, non-blocking concurrency control, stored procedures, user-defined functions, and more.

Performance features that can give dramatic price/performance gains over Oracle, and provide a superior end-user experience, include scale-out, compilation of queries to machine code, in-memory rowstore tables, and vectorized query execution for columnstores using single-instruction, multiple-data (SIMD) support.

Table of Contents

Abstract	2
Introduction	5
DDL	6
Data Types	6
User-Defined Types	8
Semi-Structured Data: JSON and XML	8
Spatial Data	8
Character Sets and Collations	8
Defaults	8
Computed Columns	8
Physical Database and Index Design	9
Tracking Column Usage	10
Queries and DML	11
Expression language and built-in functions	11
Update DML	11
Outer Join Syntax	12
Recursive Query	12
Query Execution	12
Functions and Stored Procedures	13
Constraints and Triggers	14
Sequences	14
Partitioning	14
Views and Synonyms	16

Data Loading	16
Application Changes	17
Developing on MemSQL	17
Conclusion	19
References	19

Introduction

This paper describes the process of migrating applications running on an Oracle database to MemSQL, a fast, powerful, scalable, modern relational database management system (RDBMS). MemSQL has a number of advantages that make it an attractive alternative to Oracle, including:

- a scale-out architecture that runs on industry-standard hardware
- broad ANSI SQL support
- ACID transaction support
- programmability with stored procedures and functions
- fast query processing via compilation, in-memory optimizations, columnstore support, and vectorization
- lock-free structures for non-blocking multi-version concurrency control
- support for fast ingest, high concurrency, and a mix of transactional and analytical operations
- simpler to install, manage, and maintain
- ability to run on-premises and in the cloud
- competitive pricing

For an introduction to MemSQL, please see *MemSQL Overview* [[Mem18](#)]. For a more in-depth discussion of MemSQL, refer to our documentation [[Mem19a](#)].

The rest of this paper describes how you can migrate Oracle applications to MemSQL in a straightforward way. In addition, experienced Oracle developers may also find this paper useful to help them understand how concepts they have used with Oracle can be applied in MemSQL. Topics covered include DDL, queries and DML, functions and stored procedures, constraints and triggers, sequences, partitioning, views and synonyms, and application changes.

DDL

Tables are created in MemSQL with the standard [CREATE TABLE](#) statement. MemSQL supports a rich collection of column data types including numbers, strings, binary, blobs, date, time, geospatial, and semi-structured (JSON).

Data Types

Oracle types can be easily mapped to MemSQL types [\[Mem19n\]](#), as illustrated in the following table.

Oracle Type	MemSQL Type	Comments
VARCHAR2(size [BYTE CHAR])	VARCHAR	
NVARCHAR2(size)	VARCHAR	
NUMBER [(p [, s])]	DECIMAL [(p [, s])]	
FLOAT [(p)]	DOUBLE [(p)]	
LONG	TEXT	
DATE	DATE	
BINARY_FLOAT	FLOAT	
BINARY_DOUBLE	DOUBLE	
TIMESTAMP [(fractional_seconds_precision)]	DATETIME [(fractional_seconds_precision)]	Fractional seconds must be 0 or 6 in MemSQL

TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE	DATETIME [(fractional_seconds_precision)]	Timezone information in MemSQL is handled by the application with DATETIME and built-in functions for time zone offsets including CONVERT_TZ
INTERVAL	-	INTERVAL types can't be stored in MemSQL but INTERVAL expressions can be combined with date/time types with DATE_ADD, DATE_SUB
RAW(size)	VARBINARY	
LONG RAW	LONGBLOB	
CHAR [(size [BYTE CHAR])]	CHAR(N)	
NCHAR[(size)]	CHAR(N)	
CLOB	TEXT	
NCLOB	TEXT	
BLOB	LONGBLOB	
BFILE	-	Can use file name in VARCHAR field plus file system to store files in MemSQL

User-Defined Types

MemSQL does not support user-defined datatypes. For such columns, it is recommended to review the data type and convert them as needed manually. User-defined data types in Oracle that are simply used as synonyms for standard types can be expanded to their actual type. More complex user-defined types can be re-implemented as JSON, text, or binary fields, with user-defined functions used as accessors to extract information from them or perform necessary operations on them.

Semi-Structured Data: JSON and XML

MemSQL supports a built-in JSON data type [\[Mem19c\]](#) that can be used for semi-structured data. Oracle character large object (CLOB) fields holding JSON data can be converted to the JSON type when migrated to MemSQL. MemSQL does not directly support XML, but for XML data that is interpreted completely in the client application, the data can simply be stored in a TEXT field in MemSQL. For XML data that needs to be interpreted inside the database, it may be appropriate to convert it to JSON format when moving it to MemSQL.

Spatial Data

Oracle developers that have used spatial data types such as SDOAGGRTYPE and SDO_GEOMETRY can make use of the built-in GEOGRAPHY and GEOGRAPHY_POINT types [\[Mem19d\]](#) in MemSQL to achieve similar results. MemSQL spatial indexes can be used instead of the R-tree indexes supported by Oracle.

Character Sets and Collations

MemSQL supports the UTF-8 character set, which can handle all major languages. Case-sensitive and case-insensitive collations are supported. Collation can be set at the cluster level or at the connection level [\[Mem19m\]](#).

Defaults

As in Oracle, MemSQL column values can be set to a predefined value using the DEFAULT syntax. The MemSQL TIMESTAMP type also allows default timestamps to be placed on new rows, and a fresh timestamp to be placed on a row each time it is updated.

Computed Columns

In addition, MemSQL supports persisted computed columns [\[Mem19i\]](#), which can be formed using an expression to compute their value from the values of other columns. Unlike Oracle,

MemSQL does not support non-persisted computed columns. Persisted computed columns or views can be used as a workaround.

Physical Database and Index Design

MemSQL supports a broad range of physical database designs, which can accommodate virtually all application needs. There are two major table types:

- **Rowstore:** suitable for transactional applications and hybrid transactional/analytical applications [[Mem19p](#)]
- **Columnstore:** suitable for fact tables in data warehouses and data marts, and large tables in primarily analytical applications with some operational characteristics, such as IoT applications, telemetry systems, ad tech applications, and operational data stores [[Mem19q](#)]

Rowstore tables support multiple ordered indexes, primary keys, unique indexes and unique constraints, and hash indexes. Ordered indexes are specialized in-memory skip lists, which allow compiled code to look up rows with very few instructions (about 10X less) compared to an Oracle B-tree. Rows can be accessed extremely fast via any index. Rowstore searches via an index tend to have very consistent (low variance) response time, an important benefit of MemSQL rowstore compared to traditional B-tree access methods.

Columnstore tables are highly compressed, typically by a factor of 5X to 10X, and store data in million-row chunks called segments. Within a segment, columns are stored in separate files or extents on disk. Each column of a segment has metadata to show the minimum and maximum values in the segment, to support fast range elimination for equality and greater-than, less-than, and BETWEEN filters. Columnstores support a single sort key for fast range elimination during query processing. Single row lookup performance for columnstore tables is further improved in the MemSQL 7.0 release using hash indexes and sub-segment access. These allow seeking into a columnstore.

The only difference in CREATE table syntax between a rowstore and columnstore table is the presence of the keywords USING CLUSTERED COLUMNSTORE to define the sort key.

A benefit of MemSQL columnstores compared to the dynamically-built, in-memory columnstores in Oracle, known as IMDB, is that they are persistent and don't have to reside in memory. When the system restarts, MemSQL columnstores are still there in their entirety; no CPU time needs to be spent to reconstruct them. Also, columnstores in MemSQL are available standard on all versions, not just on selected hardware platforms or software versions, as is the case in Oracle EHCC, which is a hybrid of rowstore and columnstore. Oracle EHCC is, to our knowledge, only

available on Exadata or when using ZFS/FS1 storage, not in the mainstream software product. This provides a total cost of ownership (TCO) advantage when comparing MemSQL to Oracle.

Operational applications from Oracle typically will use rowstores when migrated to MemSQL, and use MemSQL memory-optimized, lock-free ordered indexes instead of B-tree indexes. Fact tables from Oracle data warehouses and marts should be migrated to MemSQL as columnstores, regardless of the table structure in the original Oracle system. Fact tables in Oracle applications are often partitioned, but there is no need to explicitly partition such tables in MemSQL. See the section on partitioning later in this document for more details.

MemSQL supports *reference tables* [Mem19e], which are replicated to each leaf node in the distributed MemSQL architecture. These are ideal for dimension tables. Dimension tables being migrated from Oracle to MemSQL should typically be implemented using reference tables. This allows star-join queries common in data warehouses and marts to be executed with less data movement. Very large dimension tables, bigger than a few million rows, can be implemented as standard distributed tables in MemSQL to avoid storing a large volume of data multiple times.

Because MemSQL is truly distributed, an important new capability called *sharding* is available to use during physical database design. A table definition can specify an optional *shard key*, which is a key used to hash partition the table. Typically you will shard a table on a primary key or other natural key. The shard key can contain one column or several columns.

For fast joins of very large tables, you may wish to shard each table on the join column or columns. This will enable what is known as a *co-located join* between the two tables, where the tables can be joined without shuffling one or both of them first. Sharding in MemSQL is somewhat different than hash partitioning in Oracle. For more information, see the section on partitioning later in this document.

Tracking Column Usage

Oracle users may be familiar with running queries similar to the following against **sys.col_usage\$** and related tables to understand how queries in their workload are using columns in database tables:

```
select r.name as r_owner, o.name as r_table , c.name as r_column,
equality_preds, equijoin_preds, nonequijoin_preds, range_preds,
like_preds, null_preds, timestamp
from sys.col_usage$ u, sys.obj$ o, sys.col$ c, sys.user$ r
where o.obj# = u.obj# and c.obj# = u.obj# and c.col# = u.intcol#
and o.owner# = r.user# and r.name in ('ADD_SCHEMAS_HERE')
order by o.name
```

MemSQL supports built-in management views **mv_aggregated_column_usage** and **mv_query_column_usage** [\[Han18b\]](#) that can be queried to find similar information about how columns are used. This can enable the developer or DBA to choose appropriate columns for indexing and sharding.

Queries and DML

MemSQL supports broad SQL compatibility, including support for SQL-92, SQL-99 OLAP extensions, and some SQL-2003 extensions. As an example of how comprehensive the SQL coverage is in MemSQL, it can run all 99 queries in the complex TPC-DS benchmark [\[She19\]](#).

MemSQL includes support for all types of inner and outer joins, subqueries, EXISTS/NOT EXISTS, UNION, UNION ALL, common table expressions (CTEs), a large set of aggregate functions, CUBE, ROLLUP, grouping sets, PIVOT, window functions, and more. The large majority of Oracle queries can thus be migrated to MemSQL without modification.

Expression language and built-in functions

MemSQL supports a sophisticated expression sublanguage, including all the built-in functions and operators available in MySQL 5.5, as well as the Oracle-style date and time handling functions [TO_DATE\(\)](#), [TO_TIMESTAMP\(\)](#), and [TO_CHAR\(\)](#), which support format masks as used in Oracle. These functions are widely used in Oracle, and their presence in MemSQL is very useful to reduce porting costs.

The [NVL\(\)](#) function, to conditionally define a specified value for an expression if it is null, is supported in MemSQL for Oracle compatibility. In addition, the [concatenation operator symbol](#) `||` that is commonly used in Oracle is supported in MemSQL. Using `||` for concatenation is enabled in MemSQL through setting the SQL_MODE variable to include PIPES_AS_CONCAT.

Update DML

MemSQL supports INSERT, INSERT...ON DUPLICATE KEY UPDATE (a.k.a. UPSERT), DELETE, and UPDATE DML statements. This includes both single-table and multi-table (searched) variants. These allow almost all Oracle DML statements to be ported in a straightforward way. Oracle MERGE statements can be ported to MemSQL using INSERT...ON DUPLICATE KEY or multiple statements combined to achieve the desired goal, typically encapsulated in a stored procedure.

To enforce uniqueness, a unique index or primary key constraint can be used for rowstore tables. For columnstores, uniqueness needs to be guaranteed by the application in some way if

it is required, say with a high-resolution timestamp column, or with a serial number or a composite natural key provided by application logic.

Outer Join Syntax

Oracle supports a special syntax (+) for left and right outer joins which is not supported in MemSQL. Queries that use (+) will need to be modified to use the SQL standard LEFT or RIGHT outer join notation when moved to MemSQL.

Recursive Query

MemSQL does not directly support recursive query processing. Although CTEs are supported, recursive CTEs are not. Also, Oracle applications sometimes use CONNECT BY to traverse graphs or graph-like structures recursively. In MemSQL, recursive traversal is possible using a temporary table and a query in a loop to expand one level through a graph or tree in each trip through the loop. MemSQL documentation gives an example of this technique [\[Mem19I\]](#).

Query Execution

MemSQL query execution technology tends to be superior overall to Oracle query execution technology, sometimes by a performance factor of up to 10x or more. Hence, query execution performance is not typically a concern when migrating applications from Oracle to MemSQL; rather, it is a motivating factor to move applications to MemSQL to get lower TCO, a better user experience, and to enable applications that were not feasible before. Like Oracle, MemSQL parameterizes queries, compiles them, and stores them in a plan cache. On subsequent executions, MemSQL takes a query plan from the cache and runs it so it need not be compiled again.

Unlike Oracle, MemSQL compilation translates a query all the way to machine code. This, combined with in-memory row store storage structures designed with code generation in mind, allows query processing rates on the order of 20 million rows per second per core against an in-memory skip list row store table. That is about 10X faster than Oracle's per-core processing rate against a B-tree in many cases.

For columnstore tables, MemSQL uses a high-performance vectorized query execution engine that can operate on blocks of 4K rows at a time, very efficiently. This vectorized execution engine also makes use of single-instruction, multiple-data (SIMD) instructions available on Intel and compatible processors that support the AVX-2 instruction set. Processing rates on columnstore tables are often over 100 million rows per second per core, and sometimes as much as 2 billion rows per second per core when using SIMD and operations on encoded (compressed) data [\[Mem19f, Han18\]](#).

Functions and Stored Procedures

MemSQL provides a built-in database programming language, MemSQL Procedural SQL (MPSQL). MPSQL [\[MPSQL19\]](#) supports stored procedures (SPs), user-defined scalar functions (UDFs), user-defined aggregates, and table-valued functions. Many of the programming constructs and much of the syntax of MPSQL is modelled after Oracle PL/SQL.

SPs and UDFs in MPSQL can take zero or more arguments and can return values. In addition, an SP can return multiple rowsets if desired. Control flow constructs in SPs and UDFs include:

- BEGIN-END blocks
- Conditional control
 - IF ... THEN ... END IF
 - IF ... THEN ... ELSE ... END IF
 - IF ... THEN ... ELSIF ... THEN ... END IF
- Iterative control
 - LOOP ... END LOOP
 - EXIT and EXIT WHEN
 - WHILE ... LOOP ... END LOOP
 - FOR ... LOOP ... END LOOP
 - Loop Labels
 - CONTINUE and CONTINUE WHEN
- Exception handling
- CALL stored procedure or function
- ECHO command to run stored procedure and output returned rowset to client

MPSQL does not have direct support for cursors, but does support executing a query using the COLLECT() function [\[Mem19o\]](#) to produce an array of values. SPs can iterate through this array result forwards or backwards, equivalent to a read-only cursor that can move in any direction.

In addition to comprehensive control flow, SPs support embedded SQL statements with inline variable and parameter substitution. Dynamic SQL, including the Oracle-style EXECUTE IMMEDIATE, is also supported.

Constraints and Triggers

MemSQL supports the following types of constraints:

- UNIQUE (enforced)
- UNIQUE ... UNENFORCED [RELY | NORELY]
- PRIMARY KEY

Regular UNIQUE constraints are enforced by the system using an index. UNENFORCED ones can be declared, which, as expected, are not enforced by the system; the application is responsible for making sure they are maintained. If the RELY option is used, the query optimizer can take advantage of these constraints to avoid certain operations, such as DISTINCT. All UNIQUE constraints also tell the optimizer that the column is unique for statistical purposes.

PRIMARY KEY constraints are simply a syntactic variation of UNIQUE constraints.

The equivalent of CHECK constraints and referential integrity constraints are not supported in MemSQL. These constraints need to be maintained with application logic. Similarly, MemSQL does not support triggers, so trigger-style logic needs to be accomplished by the application, potentially with the aid of stored procedures in the database - e.g., after an update, the application can call a MemSQL stored procedure with the key of the updated record or records, and the stored procedure can carry out logic that would have been accomplished in Oracle with an AFTER UPDATE trigger.

Sequences

MemSQL supports declaring a column, which must also be a key, as AUTO_INCREMENT. The system then gives this column a new, unique value for each additional row, similar to the way a SEQUENCE is used in Oracle. If users desire the same behavior as SEQUENCES, they can implement the equivalent with a small stored procedure and a table to allocate ranges of new values.

Partitioning

Oracle application developers may be used to using range/list partitioning to enable data lifecycle management, particularly a "sliding window scenario" for a large historical table. The benefit of this is to enable fast bulk removal of old data by switching out a full partition of data into a separate table with a metadata-only operation, then truncating it.

MemSQL can implement this same sliding window scenario with simple DELETE statements because it can delete data extremely fast. So, while MemSQL doesn't support a range partitioning feature per se, it can handle the primary use case that motivated the feature in other products, including Oracle. There are several reasons that DELETE operations are so fast in MemSQL, including:

1. The data is hash-partitioned by default, typically with one partition per core, and each partition has its own log file, so the log tail is naturally partitioned, and log writes are parallelized.
2. Old copies of a record don't have to be written to the log for undo/redo logging. MemSQL logging is redo-only. Old record versions stay in the database data structures, as a natural byproduct of multi-version concurrency control, enabling fast transaction undo for deletes. To delete a record, only its ID is written to the log, not the full record contents.
3. Deletes to columnstores just require updates to a delete bitmap to mark records as invalid, not removal of the records. Records are removed asynchronously by a background merger process.
4. Updates to the in-memory row store and the in-memory row store segment of the columnstore (containing recent data) happen very fast because in-memory structures are being changed, not disk-based structures. Less CPU is thus needed to do these updates, and no I/O other than minimal log I/O (which is parallelized as discussed in item 1 above).

DELETE is almost always preferred for removing data in MemSQL compared with TRUNCATE. TRUNCATE requires a global lock and invalidates plans and statistics. A DELETE is usually quite fast, and it retains existing plans and statistics. TRUNCATE is still useful for bulk removal of all data from very large tables when keeping plans and statistics around is not a concern.

Another reason Oracle developers use range partitioning is to enable queries with range filters on the partitioning column to run faster, via *partition elimination* -- a technique that removes partitions from consideration if they don't lie in the range of the query filter. MemSQL columnstores have a sort key which can enable range elimination (also called *segment elimination*) easily; it's not necessary to have a partitioning feature to benefit from range elimination in MemSQL. Columns used for range partitioning (including subpartitions) are natural fits for the sort key in MemSQL columnstores.

Oracle developers also sometimes specify hash partitioning as a way to subdivide data into manageable-sized units for parallel query processing. This is not necessary in MemSQL, which uses hash partitioning by default, with the standard level of parallelism being one thread per hash partition [Mem19g]. To specify which column or columns to use to define hash partitioning, specify a *shard key* [Mem19h] in MemSQL. The MemSQL hash partition granularity defaults to

one partition per core, which is appropriate for most use cases. MemSQL further allows CPU usage for individual queries to be limited with the resource governor feature [\[Mem19z\]](#).

Views and Synonyms

MemSQL supports views, so CREATE VIEW statements can often be ported from Oracle to MemSQL without changes. MemSQL does not support synonyms for tables, but views can be used instead. For example, you can define `t2` as a synonym for table `t(a int, b int)` as follows:

```
CREATE VIEW t2 as
SELECT a, b
FROM t;
```

MemSQL uses a definer security model for views, as does Oracle, so the security model for views in MemSQL aligns with that from Oracle.

MemSQL does not support materialized views. The dominant use case for materialized views is to use them to hold summary aggregates, to speed up repetitive aggregate queries. The downside of materialized views is that they can be expensive to maintain, they can limit the rate of concurrent updates, and materialized view query rewrite is doesn't always take effect as and when desired. MemSQL's philosophy is to use high-performance parallel query and compilation, along with columnstores and vectorization, to ensure that processing queries against the raw, stored data is so fast that precomputed aggregates are not needed. This simplifies the developer's job by avoiding a challenging physical database design tradeoff.

Of course, there are some situations where the performance gains of using precomputed aggregates are so high that it is worth the cost and complexity of maintaining them. In this case, with MemSQL, you can create summary tables and query them directly [\[Ada06\]](#), rather than relying on materialized query rewrite in the optimizer.

Data Loading

MemSQL supports a comprehensive loading capability with the LOAD DATA command [\[Mem19j\]](#). This supports flexible definition of field separators and row terminators. LOAD DATA is partially parallelized in MemSQL - lines from input files are distributed to MemSQL leaves for final loading. Load performance is a strong point of MemSQL, which can be configured to load billions of rows per hour. In MemSQL 7.0 and later, trailing null columns can be ignored if desired with LOAD DATA.

Oracle LOAD DATA commands can be converted to LOAD DATA commands in MemSQL, typically with only limited changes. Rows with errors in them in MemSQL can be skipped during loading and then output later using SHOW LOAD ERRORS. The errors can be placed into an output file so the lines can be corrected and re-run through LOAD DATA.

Another popular loading feature in MemSQL is the PIPELINES capability [\[Mem19k\]](#), a streaming loader that can load from file folders, Kafka queues, and S3 buckets. It can simplify applications by removing the need for the developer to write an event-driven fetch-execute loop that monitors a queue or folder for new data. PIPELINES allow distributed, parallel scale-out for loading, enabling very fast ingest rates.

PIPELINES also support transforms which can filter or change the structure of incoming data, either with external scripts or executables, or inside the database using stored procedures. There is no direct analogue in Oracle for MemSQL Pipelines, but Oracle developers may want to consider them instead of LOAD DATA for new application development.

Application Changes

Most applications that utilize ODBC/JDBC connectivity will work natively with either the MySQL or MariaDB client software. You can download the various clients and connectors, including libraries for Python, C, C++, .NET, and Perl, from our Client Downloads page [\[Mem19r\]](#).

Note: When using the MariaDB JDBC client, be sure to include the database name in your connection string or it will fail to connect. Here's an example:

```
"jdbc:mariadb://localhost:3306/databasename", "username", "password");
```

See the MemSQL documentation for examples of how to connect as well as how to implement concurrent-multi-insert statements using typical methods (Python, Bash, Java, C, C#, NodeJS) [\[Mem19s\]](#).

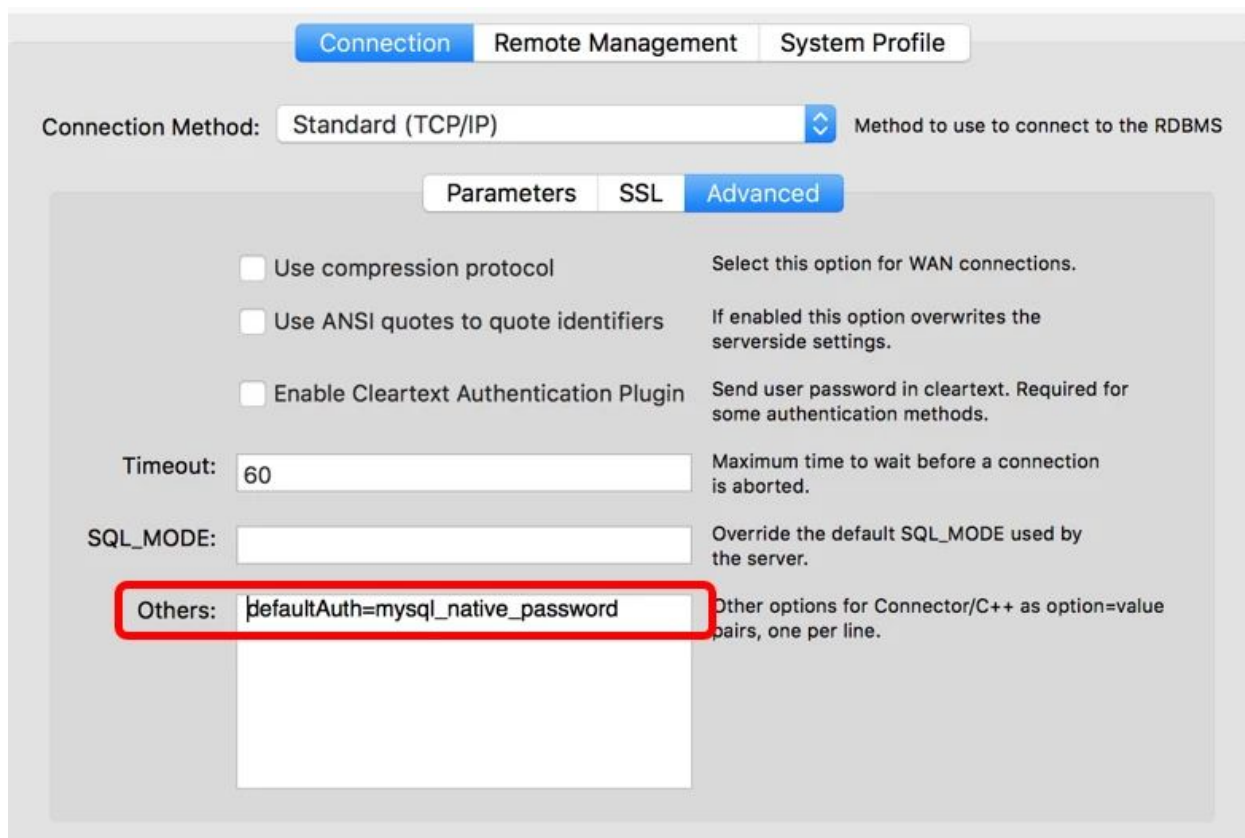
Developing on MemSQL

There are multiple widely used database development and administration applications that are compatible with MemSQL, including our own proprietary browser-based visual user interface, MemSQL Studio [\[Mem19t\]](#). For the most part, if an application is compatible with the MySQL Wire Protocol, it should be able to connect to MemSQL. Our proprietary structures may not be visible in their object explorers; however, the basic functionality should be available.

Some IDEs commonly used with MemSQL include:

- SQL Pro [[Mem19v](#)]
- SQL Workbench [[Mem19w](#)]
- MySQL Workbench [[MySQL19](#)].

If you are using the latest versions of MySQL Workbench, you will need to navigate to Connection > Advanced and enter `defaultAuth=mysql_native_password` into the Others: text box



The screenshot shows the MySQL Workbench Connection dialog box with the 'Advanced' tab selected. The 'Connection Method' is set to 'Standard (TCP/IP)'. Under the 'Advanced' tab, there are several options: 'Use compression protocol', 'Use ANSI quotes to quote identifiers', and 'Enable Cleartext Authentication Plugin'. Below these are fields for 'Timeout' (set to 60) and 'SQL_MODE'. The 'Others:' field is highlighted with a red rectangle and contains the text `defaultAuth=mysql_native_password`. The 'Others:' field is described as 'Other options for Connector/C++ as option=value pairs, one per line.'

Additional information about how to connect to some common tools can be found in the MemSQL documentation [[Mem19x](#)].

Conclusion

Moving applications from Oracle to MemSQL is possible because of MemSQL's support for standard SQL and DDL, stored procedures and functions, Oracle-like expression language and date functions, a generous set of data types, transactions, non-blocking concurrency control, indexes, row store and columnstore tables, and views. MemSQL's true scale-out architecture, in-memory row store, compilation of queries to machine code, and vectorized query execution over columnstore tables give fast and, in many cases, truly stunning performance. This provides a payoff for your end users if you make the switch. Moreover, the price-performance benefit of MemSQL over Oracle can be dramatic.

Perhaps business requirements are forcing you to scale an existing application to the point where it is no longer economically viable to use Oracle in general, or Exadata in particular. Maybe you want to try a modern alternative to Oracle for a less complex application to avoid Oracle lock-in. Or maybe you are seeking an alternative data management platform that's cloud-native, as business demands and new technology push your data and applications to the cloud. Whatever the reason may be that you're exploring alternatives to Oracle, MemSQL is an attractive destination. Once you make the switch with your first application, you can be confident that MemSQL technology can benefit you, your users, and your business well into the future.

References

- [Ada06], Christopher Adamson, Mastering Data Warehouse Aggregates: Solutions for Star Schema Performance, Wiley, 2006.
- [Han18] Eric Hanson, Shattering the Trillion-Rows-Per-Second Barrier With MemSQL, <https://www.memsql.com/blog/memsql-processing-shatters-trillion-rows-per-second-barrier/>, 2018.
- [Han18b] Eric Hanson, MemSQL 6.7 Performance Blog, <https://www.memsql.com/blog/performance-for-memsql-67/>, 2018.
- [Mem18] MemSQL Overview, https://www.memsql.com/assets/MemSQL_Overview_August_2018.pdf, August, 2018.
- [Mem19a], MemSQL Documentation, <https://docs.memsql.com/>, 2019.
- [Mem19c], JSON Guide, MemSQL, <https://docs.memsql.com/concepts/v6.8/json-guide/>, 2019.
- [Mem19d], Geospatial Guide, MemSQL, <https://www.memsql.com/content/geospatial/>, 2019.

- [Mem19e], Reference Tables, MemSQL, <https://docs.memsql.com/concepts/v6.8/table/#reference-tables>, 2019.
- [Mem19f] Understanding Operations on Encoded Data, MemSQL, <https://docs.memsql.com/concepts/v6.8/understanding-ops-on-encoded-data/>, 2019.
- [Mem19g] Distributed SQL, MemSQL, <https://docs.memsql.com/concepts/v6.8/distributed-sql/>, 2019.
- [Mem19h] Optimizing Table Data Structures, MemSQL, <https://docs.memsql.com/tutorials/v6.8/optimizing-table-data-structures/#shard-keys>, 2019.
- [Mem19i] Persisted Computed Columns, MemSQL, <https://docs.memsql.com/concepts/v6.8/persistent-computed-columns/>, 2019.
- [Mem19j] LOAD DATA, MemSQL, <https://docs.memsql.com/sql-reference/v6.8/load-data/>, 2019.
- [Mem19k] Pipelines Overview, MemSQL, <https://docs.memsql.com/memsql-pipelines/v6.8/pipelines-overview/>, 2019.
- [Mem19l] Recursive Tree Expansion Example, MemSQL, <https://docs.memsql.com/sql-reference/v6.8/create-procedure/#recursive-tree-expansion-example>, 2019.
- [Mem19m], MemSQL Collations, <https://docs.memsql.com/configuration-reference/v6.8/list-of-engine-variables/#collation-connection-collation-database-and-collation-server>, 2019.
- [Mem19n] Data Types, MemSQL, <https://docs.memsql.com/sql-reference/v6.8/datatypes/>, 2019.
- [Mem19o] Collect Function, MemSQL, <https://docs.memsql.com/sql-reference/v6.8/collect/>, 2019.
- [Mem19p] Rowstore conceptual overview, MemSQL, <https://docs.memsql.com/concepts/v6.8/rowstore/>, 2019.
- [Mem19q] Columnstore conceptual overview, MemSQL, <https://docs.memsql.com/concepts/v6.8/columnstore>, 2019.

- [Mem19r] MemSQL Client Downloads, <https://docs.memsql.com/client-downloads/>, 2019.
- [Mem19s] Concurrent Multi-Insert Examples, <https://docs.memsql.com/tutorials/v6.8/concurrent-multi-insert-examples/>, 2019.
- [Mem19t] MemSQL Studio, <https://docs.memsql.com/memsql-studio/latest/memsql-studio-overview/>, 2019.
- [Mem19u] Modernizing Data Platforms with MemSQL, June 2019.
- [Mem19v] Connecting SQL Pro to MemSQL, MemSQL, <https://docs.memsql.com/tutorials/v6.8/how-to-connect-to-memsql/#sequel-pro>, 2019.
- [Mem19w] Connecting SQL Workbench to MemSQL, MemSQL, <https://docs.memsql.com/tutorials/v6.8/how-to-connect-to-memsql/#sql-workbench>, 2019.
- [Mem19x] How to Connect to MemSQL, MemSQL, <https://docs.memsql.com/tutorials/v6.8/how-to-connect-to-memsql/>, 2019.
- [Mem19y] Modernizing Data Platforms with MemSQL, for Oracle environments, MemSQL, http://img04.en25.com/Web/MemSQL/%7B7f61289f-cb33-447e-88b0-95f034823bf9%7D_Modernizing_Data_Platforms_with_MemSQL_-_for_Oracle_Environments.pdf, 2019.
- [Mem19z] [Setting Resource Limits](#), MemSQL, 2019.
- [MySQL19] MySQL Workbench, <https://www.mysql.com/products/workbench/>, 2019.
- [MPSQL19], Procedural Extensions, MemSQL, <https://docs.memsql.com/concepts/v6.8/procedural-extensions/>, 2019.
- [She19], John Sherwood et al., We Spent a Bunch of Money on AWS And All We Got Was a Bunch of Experience and Some Great Benchmark Results, <https://www.memsql.com/blog/memsql-tpc-benchmarks/>, 2019.

Keywords: fast database, scalable SQL, translytics, translytical, HTAP, hybrid transactional and analytical processing, HOAP, hybrid operational-analytic processing, Oracle, Sybase ASE, Microsoft SQL Server, SAP HANA, MongoDB, Elastic, Elasticsearch