An introduction to LianjaSQL 11

LianjaSQL 11, is a lightweight relational database management system (RDBMS) developed to be plug-compatible with the core T-SQL functionality of Microsoft SQL Server (MSSQL). It's designed to store, manage, and retrieve data efficiently using a structured format based on tables, rows, and columns, following the relational model.

It can be used in a traditional Client/Server configuration or embedded in Python applications using the **lianjasql** python package which can be installed using *pip*.

Key Features of LianjaSQL

1. **Relational Database**:

- Organizes data into tables with defined relationships (e.g., primary keys, foreign keys).
- Supports T-SQL (Transact Structured Query Language) for querying and manipulation of data.
- Has built-in support for AI powered Natural Language Queries using the PROMPT command.
- Supports triggers.
- Supports Stored Procedures.
- Integrate business rules.
- Can be embedded in Python.
- Cloud enabled.

2. **LianjaSQL custom T-SQL implementation:**

- LianjaSQL incorporates a custom implementation of Transact-SQL (T-SQL), Microsoft's extension of SQL, which adds procedural programming, error handling, and advanced features.

4. **Native built-in JSON support:**

- Supports a native JSON data type.

- Query JSON encoded data using standard T-SQL.
- Perform CRUD operations in the cloud from any client programming language e.g. Python or Node.js.

3. **LianjaSQL AI Powered Database Management Console:**

- Create and manage LianjaSQL databases using AI to generate T-SQL to create databases and their tables.
- Develop and test AI generated T-SQL procedures.
- Query LianjaSQL databases and any other third party databases (using Virtual Tables) with NLQ (Natural Language Queries).
- Work with data in grids, forms and charts.
- Email formatted results to others.
- Manage AI conversations; Save them, Edit them, Delete them, add conditions to prompts for AI. Playback conversations at any time.
- Build hierarchical custom suggestions menus for quick access to data.
- Leverage Business Intelligence and Data Analytics using the built-in AI Assistant.
- Build and integrate AI Data agents in Python.

4. **Integration**:

- Apart from its native database, LianjaSQL can work seamlessly with MSSQL, MySQL, PostgreSQL and others using its unique *Virtual Table* functionality.
- Works seamlessly with Python, Node.js and .NET.

5. **Cloud Database Support:**

- Available on-premises or as a managed service in AWS at **Lianjacloud.com**, a cloud-based version with similar functionality.
- Work with any database using the LianjaSQL CRUD OData compatible REST API from any programming language e.g Python, Node.js and .NET.

What It's Used For

- **Data Storage**: Managing structured data for applications (e.g., customer records, sales data).
- **Web and Enterprise Apps**: Backend for systems like CRMs, ERPs, or e-commerce platforms.
- **Data Warehousing**: Storing and querying large datasets for Business Intelligence and data analytics.
- **Transactional Systems**: Handling high-volume transactions with ACID compliance (Atomicity, Consistency, Isolation, Durability).
- **A drop-in replacement for MSSQL** *without the huge licensing cost*s. It may not have all the MSSQL *yet*, but most applications should just run *out-of-the-box*.

LianjaSQL T-SQL

T-SQL, or **Transact-SQL**, is Microsoft's proprietary extension of SQL (Structured Query Language) used primarily in Microsoft SQL Server (MSSQL). It's a programming language designed for managing and manipulating relational databases, building on the standard SQL with additional features for procedural programming, error handling, and advanced data processing.

If you know SQL, T-SQL is a natural next step—it's just SQL with extra tools. It's widely used by database administrators, developers, and data analysts working in Microsoft ecosystems. Now with LianjaSQL you can now use T-SQL with any database. T-SQL is a widely used language amongst database administrators, developers, and data analysts working in Microsoft ecosystems. It is a natural progression for those familiar with SQL, as it is simply SQL with additional features.

LianjaSQL incorporates a *custom implementation* of **Transact-SQL** (T-SQL), Microsoft's extension of SQL.

Key Characteristics of T-SQL.

SQL Foundation

- T-SQL includes all the standard SQL commands like `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `CREATE` for querying and managing data.
- It adheres to ANSI SQL standards but extends them significantly.

Procedural Programming

Unlike pure SQL, which is declarative (you specify *what* you want, not *how* to get it), T-SQL adds procedural constructs like:

- Variables (`DECLARE @variable INT`)
- Loops (`WHILE`)
- Conditional statements (`IF...ELSE`)
- Stored procedures and functions

Enhanced Features

- **Error Handling**: Using `TRY...CATCH` blocks.
- **Transactions**: Explicit control with `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`.
- **Built-in Functions**: Extra utilities like `CHECKSUM`, `GETDATE()`, and string manipulation functions (`LEFT`, `RIGHT`, `SUBSTRING`).
- **Triggers**: Code that runs in response to data changes (e.g., `AFTER INSERT`).

Example of a T-SQL stored procedure

Here's a simple T-SQL script (a .sql file) that declares a variable, loops, and inserts data into a table:

```sql
SQL
DECLARE @Counter INT = 1;
WHILE @Counter <= 5
BEGIN
    INSERT INTO Employees (ID, Name)
    VALUES (@Counter, 'Employee' + CAST(@Counter AS VARCHAR(2)));
    SET @Counter = @Counter + 1;
END;
SELECT * FROM Employees;
```

- This creates 5 rows in an `Employees` table with names like "Employee1", "Employee2", etc.

DECLARE variables

Example: Basic syntax to declare a variable

Variable names begin with an @

```sql
DECLARE @VariableName DataType;
```

Example: Declare, Set, and Use Variables

```sql
DECLARE @FirstName VARCHAR(50);
DECLARE @Age INT;
DECLARE @Today DATE;

-- Assign values
SET @FirstName = 'John';
SET @Age = 30;
SET @Today = GETDATE();

-- Use in a SELECT
SELECT @FirstName AS Name, @Age AS Age, @Today AS Today;
```

Declare and Assign in One Step (SQL Server 2012+)

```sql
DECLARE @FullName VARCHAR(100) = 'Jane Doe';
DECLARE @HireDate DATE = '2024-06-01';

SELECT @FullName AS Name, @HireDate AS HireDate;
```

Use Variables in a Query

```sql
DECLARE @MinSalary INT = 50000;

SELECT FirstName, LastName, Salary
```

```
FROM Employees
WHERE Salary >= @MinSalary;
```

IF statements

Basic IF Statement

```
SQL
DECLARE @Age INT = 25;

IF @Age >= 18
    PRINT 'You are an adult.';
```

IF...ELSE Statement

```
SQL
DECLARE @Salary INT = 40000;

IF @Salary > 50000
    PRINT 'High salary';
ELSE
    PRINT 'Normal salary';
```

IF...ELSE IF...ELSE Chain

```
SQL
DECLARE @Score INT = 85;

IF @Score >= 90
    PRINT 'Grade: A';
ELSE IF @Score >= 80
    PRINT 'Grade: B';
ELSE IF @Score >= 70
    PRINT 'Grade: C';
ELSE
    PRINT 'Grade: F';
```

IF with BEGIN...END (for multiple statements)

```sql
DECLARE @Stock INT = 5;

IF @Stock < 10
BEGIN
    PRINT 'Low stock warning';
    -- You can add other logic like sending notifications
END
ELSE
BEGIN
    PRINT 'Stock level is sufficient';
END
```

Using IF in Real SQL Logic

```sql
DECLARE @DeptID INT = 3;

IF EXISTS (SELECT 1 FROM Departments WHERE DepartmentID = @DeptID)
    PRINT 'Department exists';
ELSE
    PRINT 'Department not found';
```

WHILE statements

Basic WHILE Loop

```sql
DECLARE @Counter INT = 1;

WHILE @Counter <= 5
BEGIN
    PRINT 'Counter is: ' + CAST(@Counter AS VARCHAR);
    SET @Counter = @Counter + 1;
END
```

WHILE with BREAK and CONTINUE

```sql
DECLARE @Num INT = 0;

WHILE @Num < 10
BEGIN
    SET @Num = @Num + 1;

    IF @Num = 5
        CONTINUE;  -- Skip this iteration

    IF @Num = 8
        BREAK;     -- Exit the loop

    PRINT 'Number: ' + CAST(@Num AS VARCHAR);
END
```

Loop Through Table Rows Example

```sql
DECLARE @ID INT = 1;

WHILE @ID <= (SELECT MAX(EmployeeID) FROM Employees)
BEGIN
    SELECT FirstName, LastName
    FROM Employees
    WHERE EmployeeID = @ID;

    SET @ID = @ID + 1;
END
```

TRY CATCH statements

Basic TRY...CATCH Syntax

```sql
BEGIN TRY
    -- Code that might cause an error
END TRY
BEGIN CATCH
    -- Error handling code
END CATCH
```

Example: Divide by Zero Error Handling

```sql
DECLARE @Numerator INT = 10;
DECLARE @Denominator INT = 0;
DECLARE @Result FLOAT;

BEGIN TRY
    SET @Result = @Numerator / @Denominator;
    PRINT 'Result: ' + CAST(@Result AS VARCHAR);
END TRY
BEGIN CATCH
    PRINT 'An error occurred: ' + ERROR_MESSAGE();
END CATCH
```

Example: Handling Table Insert Errors

```sql
BEGIN TRY
    INSERT INTO Employees (EmployeeID, FirstName)
    VALUES (1, 'Alice');  -- Assume ID 1 already exists
END TRY
BEGIN CATCH
    PRINT 'Insert failed: ' + ERROR_MESSAGE();
END CATCH
```

TRANSACTIONS

Transactions ensure that a group of SQL operations are treated as a single unit — they **either all succeed** or **all fail**, maintaining **data integrity**.

Basic Transaction Structure

```sql
BEGIN TRANSACTION;

-- Your SQL operations here

COMMIT;  -- If everything is successful
-- or
ROLLBACK;  -- If something goes wrong
```

Example: Transfer Money Between Accounts

```sql
BEGIN TRY
    BEGIN TRANSACTION;

    -- Subtract from Account A
    UPDATE Accounts
    SET Balance = Balance - 100
    WHERE AccountID = 1;

    -- Add to Account B
    UPDATE Accounts
    SET Balance = Balance + 100
    WHERE AccountID = 2;

    COMMIT;
    PRINT 'Transfer successful.';
END TRY
BEGIN CATCH
    ROLLBACK;
    PRINT 'Error occurred: ' + ERROR_MESSAGE();
END CATCH
```

How T-SQL Differs from Standard SQL

- **Standard SQL**: Portable across database systems (e.g., MySQL, PostgreSQL), but limited to basic querying and schema management.

- **T-SQL**: Specific to MSSQL (and LianjaSQL), with richer programming capabilities.

Common Uses

- Writing stored procedures and triggers for business logic.
- Automating database maintenance tasks.
- Complex queries with joins, subqueries.
- Works well with AI (OpenAI in particular) as it *understands* T-SQL syntax. The Lianja AI Assistant uses T-SQL extensively.

LianjaSQL T-SQL COMMAND SUMMARY

Command	Description	Implemented
Data Definition Language		
CREATE DATABASE <*dbname*>		yes
ALTER DATABASE <*dbname*> METADATA "<*metadata*>"		yes
DROP DATABASE <*dbname*>		yes
BACKUP DATABASE <*dbname*>		yes
RESTORE DATABASE <*dbname*>		yes
CREATE TABLE <*tablename*>		yes
ALTER TABLE <*tablename*>		yes
DROP TABLE <*tablename*>		yes
CREATE INDEX		yes
DROP INDEX		yes
TRUNCATE TABLE <*tablename*>		yes
Data Control Language		
GRANT		yes
REVOKE		yes

Transaction Control Language		
BEGIN TRANSACTION		yes
COMMIT		yes
ROLLBACK		yes
SAVEPOINT		no
Data Manipulation Language		
USE *<dbname>;* *OPEN DATABASE <dbname>*	Switch the context to a specific database. Note that the USE statement is compiled in .sql stored procedures to OPEN DATABASE.	yes
GO	Does nothing. Commands are executed one after the other.	yes
SELECT	Many extensions to SAVE AS JSON, XML, EXCEL, CSV etc.	yes
PROMPT <prompt to AI>	Ask AI any questions related to the currently selected database (from the USE statement) and then execute the SQL SELECT generated by AI.	yes
INSERT		yes
UPDATE		yes
DELETE		yes
MERGE		No
EXEC *<storedprocedure.sql>*		yes
EXPLAIN		yes
LOCK		yes

LianjaSQL T-SQL COMMANDS DEEP DIVE

https://learn.microsoft.com/en-us/sql/t-sql/statements/statements

LianjaSQL T-SQL FUNCTION SUMMARY

Function	Description	Implemented
Date/Time		
GETDATE()		yes
DAY()		yes
MONTH()		yes
YEAR()		yes
GETUTCDATE()		yes
SYSDATETIME()		yes
SYSUTCDATETIME()		yes
DATEADD()		yes
DATEDIFF()		yes
DATEDIFF_BIG()		no
DATENAME()		yes
DATEPART()		yes
DATEFROMPARTS()		yes
DATETIMEFROMPARTS()		yes
DATETIME2FROMPARTS()		no
EOMONTH()		yes
DATETRUNC()		yes
CURRENT_DATE		yes
CURRENT_TIMESTAMP		yes
CURRENT_USER		yes
CURRENT_TIMEZONE()		no
CURRENT_TIMEZONE_ID()		no
DATE_BUCKET()		no

ISDATE()		yes
Bitwise		
LEFT_SHIFT()		yes
RIGHT_SHIFT()		yes
GET_BIT()		yes
SET_BIT()		yes
BIT_COUNT()		yes
CHECKSUM()		yes
BINARY_CHECKSUM()		yes
String		
ASCII()		yes
CHAR()		yes
NCHAR()		no
LEN()		yes
UNICODE()		no
CHARINDEX()		yes
LEFT()		yes
RIGHT()		yes
SUBSTRING()		yes
STUFF()		yes
STR()		yes
REPLACE()		no
REPLICATE()		yes
QUOTENAME()		yes
PATINDEX()		no
REVERSE()		no
STRING_AGG()		no

STRING_ESCAPE()		no
STRING_SPLIT()		no
TRANSLATE()		no
DATALENGTH()		yes
LTRIM()		yes
RTRIM()		yes
TRIM()		yes
SPACE()		yes
UPPER()		yes
LOWER()		yes
COMPRESS()		yes
DECOMPRESS()		yes
CONCAT()		yes
CONCAT_WS()		no
FORMAT()		yes
SOUNDEX()		yes
DIFFERENCE()		yes
BASE64_ENCODE()		yes
BASE64_DECODE()		yes
UNISTR()		no
Fuzzy String Match		
EDIT_DISTANCE()		no
EDIT_DISTANCE_SIMILARITY()		no
JARO_WINKLER_DISTANCE()		no
JARO_WINKLER_SIMILARITY()		no
Logical		
CHOOSE()		yes

GREATEST()		yes
IIF()		yes
LEAST()		yes
Mathematical		
ABS()		yes
ACOS()		yes
ASIN()		yes
ATAN()		yes
ATN2()		yes
CEILING()		yes
COS()		yes
COT()		no
DEGREES()		no
EXP()		yes
LOG()		yes
LOG10()		yes
PI()		yes
POWER()		no
RADIANS()		no
RAND()		yes
ROUND()		yes
SIGN()		yes
SIN()		yes
SQRT()		yes
TAN()		yes
Regular Expressions		
REGEXP_LIKE()		no

REGEXP_REPLACE()		no
REGEXP_SUBSTR()		no
REGEXP_INSTR()		no
REGEXP_COUNT()		no
REGEXP_MATCHES()		no
REGEXP_SPLIT_TO_TABLE()		no
System Info		
HOST_NAME()		yes
HOST_ID()		yes
Type Checking		
ISNUMERIC()		yes
ISJSON()		
Casting/Parsing		
PARSE()		no
TRY_CAST()		no
TRY_CONVERT()		no
TRY_PARSE()		no
CAST()		yes
CONVERT()		yes
JSON		
JSON_ARRAY()		no
JSN_ARRAYAGG()		no
JSON_OBJECT()		no
JSON_MODIFY()		no
JSON_VALUE()		no
JSON_PATH_EXISTS()		no
JSON_QUERY()		no

JSON_ENCODE()		yes
JSON_ENCODE_FILE()		yes
JSON_DECODE()		yes
JSON_DECODE_FILE()		yes

LianjaSQL T-SQL FUNCTION DEEP DIVE

https://learn.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql

LianjaSQL T-SQL DEEP DIVE

SQL SELECT

Here's a comprehensive overview of T-SQL SELECT statements, which are used to query and retrieve data from SQL Server databases.

Basic Syntax

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column;
```

Key Clauses in SELECT

Clause	Purpose
SELECT	Specifies the columns to retrieve
FROM	Specifies the table(s) to query from
WHERE	Filters rows based on conditions
ORDER BY	Sorts the result set

GROUP BY	Aggregates rows by column(s)
HAVING	Filters aggregated results (like WHERE, but after `GROUP BY`)
JOIN INNER JOIN OUTER JOIN LEFT JOIN RIGHT JOIN	Combines rows from multiple tables based on related columns
TOP	Limits the number of rows returned
OFFSET x ROWS FETCH NEXT y ROWS ONLY	Skip first x rows Get next y rows
DISTINCT	Removes duplicate rows

Example Pagination

```SQL
SELECT FirstName, LastName
FROM Employees
ORDER BY EmployeeID
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY;
```

Subquery in the SELECT Clause

```SQL
SELECT
    EmployeeID,
    (SELECT DepartmentName FROM Departments WHERE Departments.DepartmentID =
Employees.DepartmentID) AS DepartmentName
FROM Employees;
```

Subquery in the FROM Clause

```SQL
SELECT
    e.EmployeeID,
    e.FirstName,
    d.DepartmentName
FROM
    Employees e
    INNER JOIN (SELECT DepartmentID, DepartmentName FROM Departments WHERE
IsActive = 1) d
        ON e.DepartmentID = d.DepartmentID;
```

Subquery in the WHERE Clause

```SQL
SELECT
    EmployeeID,
    FirstName
FROM
    Employees
WHERE
    DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName =
'Sales');
```

IN Subquery

```SQL
SELECT
    EmployeeID,
    FirstName
FROM
    Employees
WHERE
    DepartmentID IN (SELECT DepartmentID FROM Departments WHERE Location = 'New
York');
```

EXISTS Subquery

```sql
SELECT
    e.EmployeeID,
    e.FirstName
FROM
    Employees e
WHERE
    EXISTS (
        SELECT 1 FROM Projects p WHERE p.EmployeeID = e.EmployeeID AND p.Status
= 'Active'
    );
```

Correlated Subquery Example

```sql
SELECT
    EmployeeID,
    FirstName,
    Salary
FROM
    Employees e
WHERE
    Salary > (
        SELECT AVG(Salary)
        FROM Employees
        WHERE DepartmentID = e.DepartmentID
    );
```

Common Table Expressions (CTEs)

A CTE (Common Table Expression) in T-SQL is a temporary named result set that you define within a query and can reference immediately afterward. It's similar to a subquery or a derived table, but more readable and reusable, especially for complex queries or recursive operations.

```sql
WITH HighEarners AS (
    SELECT FirstName, LastName, Salary
    FROM Employees
    WHERE Salary > 100000
)
```

```
SELECT * FROM HighEarners;
```

Using Temporary Tables

In T-SQL, temporary tables are used to store intermediate results temporarily during a session or batch. They are useful for breaking down complex queries, improving performance, or storing data for reuse within stored procedures or scripts.

Local Temporary Tables

- Visible only to the current stored procedure.
- Name starts with #.
- Automatically dropped when the stored procedure ends.

```
SQL
CREATE TABLE #TempEmployees (
    EmployeeID INT,
    Name VARCHAR(100)
);

INSERT INTO #TempEmployees VALUES (1, 'Alice'), (2, 'Bob');
SELECT * FROM #TempEmployees;
```

Global Temporary Tables

- Visible to **all stored procedures.**
- Name starts with ##.
- Dropped when the **last session using it** ends.

```
SQL
CREATE TABLE ##GlobalTemp (
    ID INT,
    Value ARCHAR(50)
);

INSERT INTO ##GlobalTemp VALUES (1, 'Test');
SELECT * FROM ##GlobalTemp;
-- You can explicity delete it
```

```
DROP TABLE ##GlobalTemp;
```

SQL INSERT

Insert a Single Row (All Columns)

```
SQL
INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID)
VALUES (1001, 'Alice', 'Smith', 2);
```

Insert a Single Row (Some Columns, Defaults Used for Others)

```
SQL
INSERT INTO Employees (FirstName, LastName)
VALUES ('Bob', 'Johnson');
```

Insert Multiple Rows

```
SQL
INSERT INTO Departments (DepartmentID, DepartmentName)
VALUES
    (1, 'Sales'),
    (2, 'HR'),
    (3, 'IT');
```

Insert Using a SELECT Statement (from another table or subquery)

```
SQL
INSERT INTO ArchiveEmployees (EmployeeID, FirstName, LastName, DepartmentID)
SELECT EmployeeID, FirstName, LastName, DepartmentID
FROM Employees
WHERE DepartmentID = 3;
```

Insert Using a Scalar Subquery

```sql
INSERT INTO DepartmentStats (DepartmentID, EmployeeCount)
VALUES (
    2,
    (SELECT COUNT(*) FROM Employees WHERE DepartmentID = 2)
);
```

Insert Default Values (for tables with only default-valued columns)

```sql
INSERT INTO SomeDefaultsTable DEFAULT VALUES;
```

SQL UPDATE

Update a Single Row by Primary Key

```sql
UPDATE Employees
SET FirstName = 'Robert', LastName = 'Williams'
WHERE EmployeeID = 1001;
```

Update Multiple Rows Based on a Condition

```sql
UPDATE Employees
SET DepartmentID = 2
WHERE DepartmentID = 5;
```

Update Using Values from Another Table (with JOIN)

```sql
UPDATE e
SET e.DepartmentID = d.NewDepartmentID
FROM Employees e
```

```
JOIN DepartmentChanges d ON e.DepartmentID = d.OldDepartmentID
WHERE d.ChangeDate = '2025-05-01';
```

Update with a Subquery

```
SQL
UPDATE Employees
SET Salary = Salary * 1.10
WHERE DepartmentID = (
    SELECT DepartmentID FROM Departments WHERE DepartmentName = 'Sales'
);
```

Update Multiple Columns

```
SQL
UPDATE Employees
SET FirstName = 'Mary', LastName = 'Lee', DepartmentID = 4
WHERE EmployeeID = 1002;
```

Update Based on Aggregate from Another Table

```
SQL
UPDATE e
SET e.Bonus = (
    SELECT AVG(Bonus) FROM Employees WHERE DepartmentID = e.DepartmentID
)
FROM Employees e;
```

Update Using CASE Expression

```
SQL
UPDATE Employees
SET Salary =
    CASE
```

```
        WHEN DepartmentID = 1 THEN Salary * 1.05
        WHEN DepartmentID = 2 THEN Salary * 1.10
        ELSE Salary
    END;
```

SQL DELETE

Delete a Single Row by Primary Key

```
SQL
DELETE FROM Employees
WHERE EmployeeID = 1001;
```

Delete Multiple Rows Based on a Condition

```
SQL
DELETE FROM Employees
WHERE DepartmentID = 5;
```

Delete All Rows from a Table

```
SQL
DELETE FROM Employees;
```

Delete Using a Subquery

```
SQL
DELETE FROM Employees
WHERE DepartmentID IN (
    SELECT DepartmentID FROM Departments WHERE Location = 'Remote'
);
```

Delete Using JOIN (using a FROM clause)

```SQL
DELETE e
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID
WHERE d.IsActive = 0;
```

Delete Top N Rows (e.g., for cleanup/testing)

```SQL
DELETE TOP (10) FROM Employees
WHERE DepartmentID = 3;
```