



SQL (Structured Query Language)

Module 1 (Basics of SQL)

Contents

Week 3	
2.1 SQL Query Processing	2
2.2 CTE in SQL	5
2.3 SQL Trigger Student Database	8
2.4 Book Management Database	12
2.5 Introduction to NoSQL	14

2.1 Query Processing

Translations on high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result.

- It is a step wise process that can be used at the physical level of the file system, query optimization and actual execution of the query to get the result.
- It requires the basic concepts of relational algebra and file structure.
- It refers to the range of activities that are involved in extracting data from the database.
- It includes translation of queries in high-level database languages into expressions that can be implemented at the physical level of the file system.
- In query processing, we will actually understand how these queries are processed and how they are optimized.

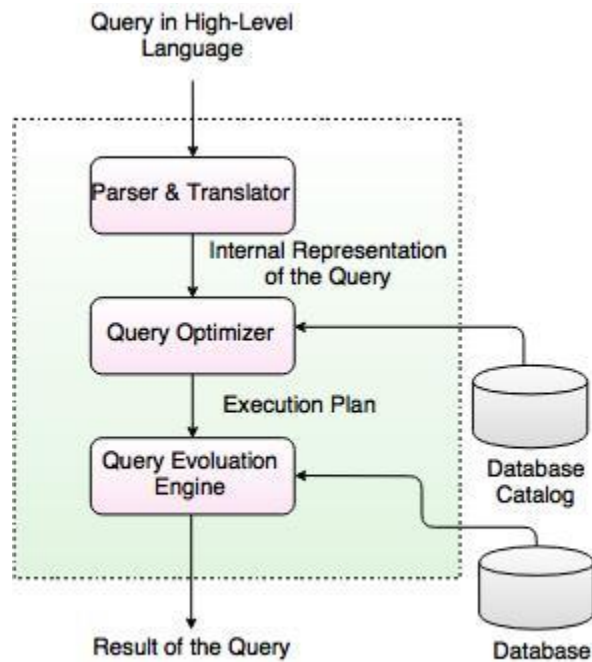
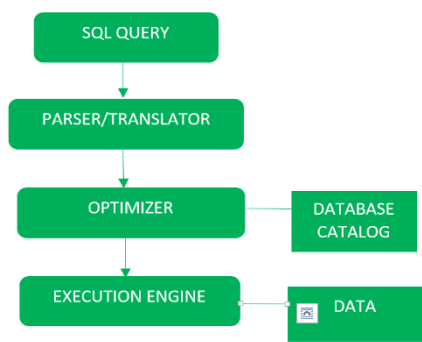


Fig. Query Processing

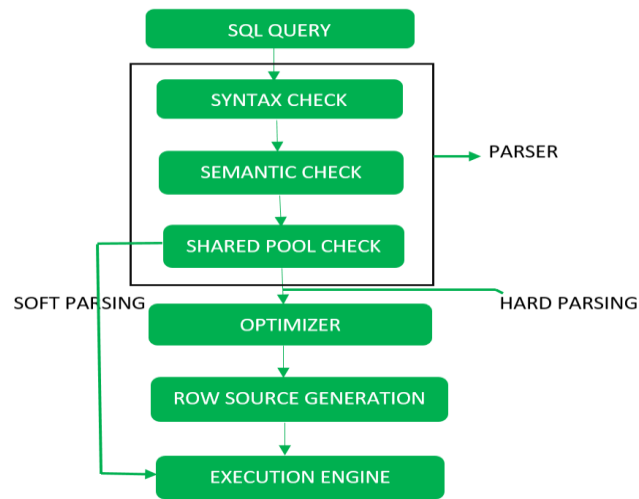
In the above diagram,

- The first step is to transform the query into a standard form.
- A query is translated into SQL and into a relational algebraic expression. During this process, Parser checks the syntax and verifies the relations and the attributes which are used in the query.
- The second step is Query Optimizer. In this, it transforms the query into equivalent expressions that are more efficient to execute.
- The third step is Query evaluation. It executes the above query execution plan and returns the result.

Block Diagram of Query Processing is as:



Detailed Diagram is drawn as:



It is done in the following steps:

- Step-1:**
Parser: During parse call, the database performs the following checks- Syntax check, Semantic check and Shared pool check, after converting the query into relational algebra.
Parser performs the following checks as (refer detailed diagram):
 - Syntax check** – concludes SQL syntactic validity. Example:
SELECT * FORM employee

Here error of wrong spelling of FROM is given by this check.
 - Semantic check** – determines whether the statement is meaningful or not. Example: query contains a tablename which does not exist is checked by this check.
 - Shared Pool check** – Every query possess a hash code during its execution. So, this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.



Hard Parse and Soft Parse –

If there is a fresh query and its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not pass through additional steps. It just passes directly to execution engine (refer detailed diagram). This is known as soft parsing.

Hard Parse includes following steps – Optimizer and Row source generation.

- **Step-2:**
Optimizer: During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. This database never optimizes DDL unless it includes a DML component such as subquery that require optimization.
It is a process in which multiple query execution plan for satisfying a query are examined and most efficient query plan is satisfied for execution.
Database catalog stores the execution plans and then optimizer passes the lowest cost plan for execution.

Row Source Generation –
The Row Source Generation is a software that receives a optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. the iterative plan is the binary program that when executes by the sql engine produces the result set.
- **Step-3:**
Execution Engine: Finally runs the query and display the required result.

2.2 CTE in SQL

A Common Table Expression (CTE) is the result set of a query which exists temporarily and for use only within the context of a larger query. Much like a derived table, the result of a CTE is not stored and exists only for the duration of the query. This article will focus on non-recursive CTEs. Enable users to more easily write and maintain complex queries via increased readability and simplification. This reduction in complexity is achieved by deconstructing ordinarily complex queries into simple blocks to be used, and reused if necessary, in rewriting the query. Performing the same calculation multiple times over across multiple query components

The common table expression (CTE) is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. You can also use a CTE in a CREATE a view, as part of the view's SELECT query. In addition, as of SQL Server 2008, you can add a CTE to the new MERGE statement.

Using the CTE –

We can define CTEs by adding a WITH clause directly before SELECT, INSERT, UPDATE, DELETE, or MERGE statement. The WITH clause can include one or more CTEs separated by commas. The following syntax can be followed:

Syntax:

1

```
[WITH [, ...]]  
::=  
cte_name [(column_name [, ...])]  
AS (cte_query)
```

2

```
WITH  
expression_name_1 AS  
(CTE query definition 1)  
  
[, expression_name_X AS  
(CTE query definition X)  
, etc ]  
  
SELECT expression_A, expression_B, ...  
FROM expression_name_1
```

Creating a Recursive Common Table Expression –

A recursive CTE is one that references itself within that CTE. The recursive CTE is useful when working with hierarchical data as the CTE continues to execute until the query returns the entire hierarchy.

A typical example of hierarchical data is a table that includes a list of employees. For every employee, the table provides a reference to that person's manager. That reference is itself an employee ID within the same table. You can use a recursive CTE to display the hierarchy of employee data.

If a CTE is created incorrectly it can enter an infinite loop. To prevent this, the MAXRECURSION hint can be added in the OPTION clause of the primary SELECT, INSERT, UPDATE, DELETE, or MERGE statement.

A table is created:

```
CREATE TABLE Employees
```

```
(  
    EmployeeID int NOT NULL PRIMARY KEY,  
    FirstName varchar(50) NOT NULL,  
    LastName varchar(50) NOT NULL,  
    ManagerID int NULL  
)
```

```
INSERT INTO Employees VALUES (1, 'Ken', 'Thompson', NULL)
```

```
INSERT INTO Employees VALUES (2, 'Terri', 'Ryan', 1)
```

```
INSERT INTO Employees VALUES (3, 'Robert', 'Durello', 1)
```

```
INSERT INTO Employees VALUES (4, 'Rob', 'Bailey', 2)
```

```
INSERT INTO Employees VALUES (5, 'Kent', 'Erickson', 2)
```


```
INSERT INTO Employees VALUES (6, 'Bill', 'Goldberg', 3)
```

```
INSERT INTO Employees VALUES (7, 'Ryan', 'Miller', 3)
```

```
INSERT INTO Employees VALUES (8, 'Dane', 'Mark', 5)
```

```
INSERT INTO Employees VALUES (9, 'Charles', 'Matthew', 6)
```

```
INSERT INTO Employees VALUES (10, 'Michael', 'Jhonson', 6)
```



After the Employees table is created, following SELECT statement, which is preceded by a WITH clause that includes a CTE named cteReports is created:

WITH

cteReports (EmpID, FirstName, LastName, MgrID, EmpLevel)

AS

(

SELECT EmployeeID, FirstName, LastName, ManagerID, 1

FROM Employees

WHERE ManagerID IS NULL

UNION ALL

SELECT e.EmployeeID, e.FirstName, e.LastName, e.ManagerID,

r.EmpLevel + 1

FROM Employees e

INNER JOIN cteReports r

ON e.ManagerID = r.EmpID

)

SELECT

FirstName + ' ' + LastName AS FullName,

EmpLevel,

(SELECT FirstName + ' ' + LastName FROM Employees

WHERE EmployeeID = cteReports.MgrID) AS Manager

FROM cteReports

ORDER BY EmpLevel, MgrID

Thus CTEs can be a useful tool when you need to generate temporary result sets that can be accessed in a SELECT, INSERT, UPDATE, DELETE, or MERGE statement.

2.3 SQL Trigger | Student Database

A trigger is a special type of stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Database manipulation DML triggers run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. These triggers fire when any valid event fires, whether table rows are affected or not.

Database definition DDL triggers run in response to a variety of data definition language (DDL) events. These events primarily correspond to Transact-SQL CREATE, ALTER, and DROP statements, and certain system stored procedures that perform DDL-like operations.

A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN)

Syntax:

```
1 create trigger [trigger_name]
   [before | after]
   {insert | update | delete}
   on [table_name]
   [for each row]
   [trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.
3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.

Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;
```

Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto_increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
subj2	int(2)	YES		NULL	
subj3	int(2)	YES		NULL	
total	int(3)	YES		NULL	
per	int(3)	YES		NULL	

7 rows in set (0.00 sec)

SQL Trigger to problem statement.

```
create trigger stud_marks
```

```
before INSERT on Student for each row set Student.total = Student.subj1 + Student.subj2 + Student.subj3,  
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
```

Query OK, 1 row affected (0.09 sec)

```
mysql> select * from Student;
```

tid	name	subj1	subj2	subj3	total	per
100	ABCDE	20	20	20	60	36

1 row in set (0.00 sec)

In this way trigger can be creates and executed in the databases.

Creating Triggers

The syntax for creating a trigger is –

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name – Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} – This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} – This specifies the DML operation.
- [OF col_name] – This specifies the column name that will be updated.
- [ON table_name] – This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) – This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.



Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters

Select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

2.4 SQL Trigger | Book Management Database

Given Library Book Management database schema with Student database schema. In these databases, if any student borrows a book from library then the count of that specified book should be decremented. To do so,

Suppose the schema with some data,

```
mysql> select * from book_det;
```

bid	bttitle	copies
1	Java	10
2	C++	5
3	MySQL	10
4	Oracle DBMS	5

4 rows in set (0.00 sec)

```
mysql> select * from book_issue;
```

bid	sid	bttitle
-----	-----	---------

1 row in set (0.00 sec)

To implement such procedure, in which if the system inserts the data into the book_issue database a trigger should automatically invoke and decrements the copies attribute by 1 so that a proper track of book can be maintained.

Trigger for the system –

```
create trigger book_copies_deducts
after INSERT
on book_issue
for each row
update book_det set copies = copies - 1 where bid = new.bid;
```

Above trigger, will be activated whenever an insertion operation performed in a book_issue database, it will update the book_det schema setting copies decrements by 1 of current book id(bid).

Results –

```
mysql> insert into book_issue values(1, 100, "Java");
```

Query OK, 1 row affected (0.09 sec)

```
mysql> select * from book_det;
```

bid	btitle	copies
1	Java	9
2	C++	5
3	MySQL	10
4	Oracle DBMS	5

4 rows in set (0.00 sec)

```
mysql> select * from book_issue;
```

bid	sid	btitle
1	100	Java

1 row in set (0.00 sec)

As above results show that as soon as data is inserted, copies of the book deducts from the book schema in the system.



2.5 Introduction to NoSQL

NoSQL databases (aka "not only SQL") are non-tabular, and store data differently than relational tables. Originally referring to non SQL or non-relational is a database that provides a mechanism for storage and retrieval of data. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.

NoSQL databases are databases that store data in a format other than relational tables. A common misconception is that NoSQL databases or non-relational databases don't store relationship data well. NoSQL databases can store relationship data—they just store it differently than relational databases do. NoSQL data models allow related data to be nested within a single data structure.

A NoSQL database includes simplicity of design, simpler horizontal scaling to clusters of machines and finer control over availability.

Many NoSQL stores compromise consistency in favor of availability, speed and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, lack of standardized interfaces, and huge previous investments in existing relational databases. Most NoSQL stores lack true ACID(Atomicity, Consistency, Isolation, Durability) transactions but a few databases, such as MarkLogic, Aerospike, FairCom c-treeACE, Google Spanner (though technically a NewSQL database), Symas LMDB, and OrientDB have made them central to their designs.

Advantages of NoSQL:

1. **High scalability** –
NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.
2. **High availability** –
Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.



Disadvantages of NoSQL:

NoSQL has the following disadvantages.

1. **Narrow focus** –
NoSQL databases have very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
2. **Open-source** –
NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words two database systems are likely to be unequal.
3. **Management challenge** –
The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
4. **GUI is not available** –
GUI mode tools to access the database is not flexibly available in the market.
5. **Backup** –
Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
6. **Large document size** –
Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

Types of NoSQL database:

Types of NoSQL databases and the name of the databases system that falls in that category are:

1. **Graph databases** database.
Store data in nodes and edges. Nodes typically store information about people, places, and things while edges store information about the relationships between the nodes. Graph databases excel in use cases where you need to traverse relationships to look for patterns such as social networks, fraud detection, and recommendation engines.
2. **Key value store:** Memcached, Redis, Coherence
Are a simpler type of database where each item contains keys and values. A value can typically only be retrieved by referencing its key, so learning how to query for a specific key-value pair is typically simple. Key-value databases are great for use cases where you need to store large amounts of data but you don't need to perform complex queries to retrieve it.
3. **Tabular:** Hbase, Big Table, Accumulo
Store data in tables, rows, and dynamic columns. Wide-column stores provide a lot of flexibility over relational databases because each row is not required to have the same columns. Wide-column stores are great for when you need to store large amounts of data and you can predict what your query patterns will be. Wide-column stores are commonly used for storing Internet of Things data and user profile data. Cassandra and HBase are two of the most popular wide-column stores.
4. **Document based:** MongoDB, CouchDB, Cloudant
store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects, and their structures typically align with objects developers are working with in code.



How NoSQL Databases Work

The NoSQL Case. In the section Types of NoSQL Databases above, there were four types described, and each has its own data model.

Each type of NoSQL database would be designed with a specific customer situation in mind, and there would be technical reasons for how each kind of database would be organized. The simplest type to describe is the document database, in which it would be natural to combine both the basic information and the customer information in one JSON document. In this case, each of the SQL column attributes would be fields and the details of a customer's record would be the data values associated with each field.

For example: Last_name: "Jones", First_name: "Mary", Middle_initial: "S", etc

When should NoSQL be used:

1. When huge amount of data need to be stored and retrieved .
2. The relationship between the data you store is not that important
3. The data changing over time and is not structured.
4. Support of Constraints and Joins is not required at database level
5. The data is growing continuously and you need to scale the database regular to handle the data.