```yaml
id: 1726898206-QKMY
aliases:
  - DBMS U2
tags: []
Author: vortex
```

DBMS U2

Relational Algebra

- Operations take 2 input relations to give a new output relation.
- This algebra is hence *closed*.

Operators

1. σ : Select
   - Selects tuples from a relation.
   - $\sigma_p(R)$: p is the selection predicate
   - Select operation is commutative so order of selects don't matter.
2. Π : Project
   - Returns a relation with some columns removed.
   - Vertical partitioning
   - Duplicate Columns are removed as it is a *set* of attributes.
   - It is NOT commutative and hence order of Project operations matter.
3. ρ : Rename
   - Renames relations without affecting values.
   - $\rho_{oldname1,oldname2->newname1,newname2}$

> There is also an Assignment operator "$\leftarrow$" to simplify the expression by assigning to temporary relation variables.

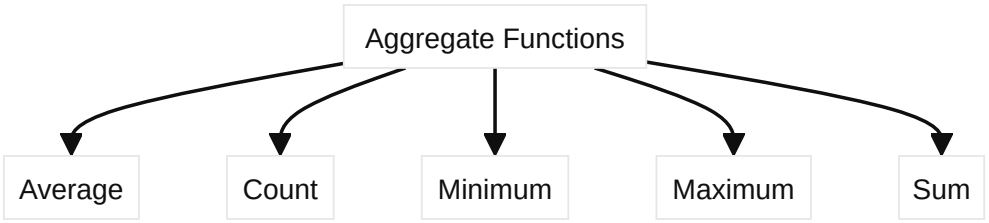
4. X : Cartesian Product
   - Combines 2 relations in every possible way.
   - Combined relation will have all combinations of pairs.
   - By itself it doesn't make much significance.
   - We can combine it with select operation to select only the rows relavent to us.
   - $\sigma_\theta(r_1 X r_2)$
5. $\bowtie$: Natural Join
   - Combines the Select+Cartesian product operation from above.
   - $\sigma_\theta(r_1 X r_2) \equiv r_1 \bowtie_\theta r_2$
6. $\bigcup$: Union
   - Combines 2 relations

- Requirements:
  1. Same Arity(number of attributes)
  2. Type compatibility
- Commutative and Associative
- Removes Duplicates

7. $\bigcap$: Intersection
   - Finds tuples that are in both the relations.
   - Requirements are the same as Union.
   - Commutative and associative
8. $-$: Set-difference
   - Finds tuples that are in one but not the other.
   - *Not* Commutative

Aggregate Functions

- Takes a collection of values and returns *one* value.

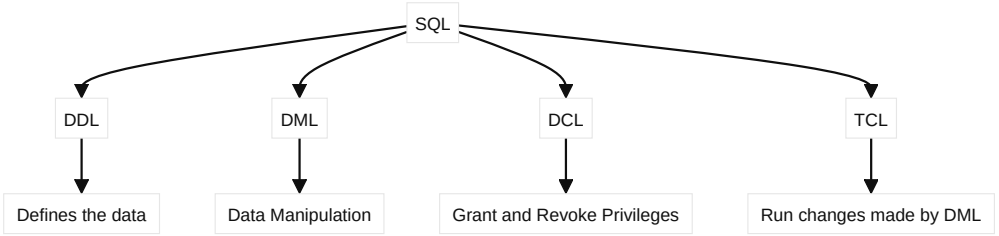


- Such aggregation is not possible using just relational algebra.

For Example:
- $\mathscr{F}_{MAXSalary}(EMPLOYEE)$: Returns max salary value in the `EMPLOYEE` relation
- $_{DNo}\mathscr{F}_{MAXSalary}(EMPLOYEE)$: Grouping by `DNo`

SQL

- Structured Query Language



Datatypes

1. Numerics
    - `BIT(size)` : A type of BIT(M) enables storage of M-bit values. M can range from 1 to 64.

Table 13.1 Required Storage and Range for Integer Types Supported by MySQL

Type	Storage (Bytes)	Minimum Value Signed	Minimum Value Unsigned	Maximum Value Signed	Maximum Value Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	-8388608	0	8388607	16777215
INT	4	-2147483648	0	2147483647	4294967295
BIGINT	8	-2^{63}	0	$2^{63}-1$	$2^{64}-1$

    - `Numeric` & `Decimal` : use when precision is important and exact value must be stored.

```
salary DECIMAL(5,2)
-- 5 is the precision and 2 is the scale
-- It means there are 5 significant digits and
-- 2 digits after the decimal point.
```

    - `Float/Real(p)` & `Double(p)` : Approx numeric value storage
        - `Float` : 4 bytes
        - `Double` : 8 bytes
        - A precision from 0 to 23 results in a 4-byte single-precision FLOAT column.
        - A precision from 24 to 53 results in an 8-byte double-precision DOUBLE column.
2. Strings
    - `CHAR(n)` & `VARCHAR(n)` :
        - The length of a CHAR column is fixed to the length that you declare when you create the table
        - Length can be betwenn 0-255.
        - Extra space is *right-padded* to fill it to full.
        - On retreival, the right trailing spaces are removed unless `PAD_CHAR_TO_FULL_LENGTH` is enabled.
        - Values in VARCHAR columns are variable-length strings.
        - Length can be 0-65,535.
        - VARCHAR values are not padded when they are stored.
        - Trailing spaces are retained when values are stored and retrieved, in conformance with standard SQL.

Value	CHAR(4)	Storage Required	VARCHAR(4)	Storage Required
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes

- `BIT(n)` & `BIT VARYING(n)`
- `BOOL/BOOLEAN` : basically a `tinyint(1)`

3. Date & Time
   - `DATE`, `DATETIME` & `TIMESTAMP`:
     - DATE : `YYYY-MM-DD`
     - DATETIME : `YYYY-MM-DD hh:mm:ss`
     - TIMESTAMP : `YYYY-MM-DD hh:mm:ss`
     - Both timestamp and datetime can store upto 6-digit precision decimals.
     - Timestamp stores values in UTC and has to convert to local time on retreivals.
4. `NULL` : Absence of data
5. `ENUM` : Choice of possible data.
6. Large OBjects(LOB):
   - Large data formats like images,video,audio can't be efficiently loaded into memory.
   - SQL provides datatypes that are efficiently retrieved for this purpose.
   - BLOB: For raw binaries
     - TINYBLOB
     - BLOB
     - MEDIUMBLOB
     - LONGBLOB
   - CLOB: For Large text blocks
     - TINYTEXT
     - TEXT
     - MEDIUMTEXT
     - LONGTEXT

DDL

1. Create Table

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]
```

2. Alter Table

```
ALTER TABLE table_name ADD Col_name datatype()...;

ALTER TABLE table_name MODIFY (fieldname datatype()...);

ALTER TABLE table_name RENAME COLUMN (Old_fieldname TO New_fieldname...);

ALTER TABLE table_name DROP COLUMN column_name;
```

3. Describe

```
DESC table_name;
SHOW COLUMNS FROM table_name;
DESCRIBE table_name;
show create table table_name; -- shows the create table cmd
```

4. Drop

```
DROP table table_name;
```

5. Rename

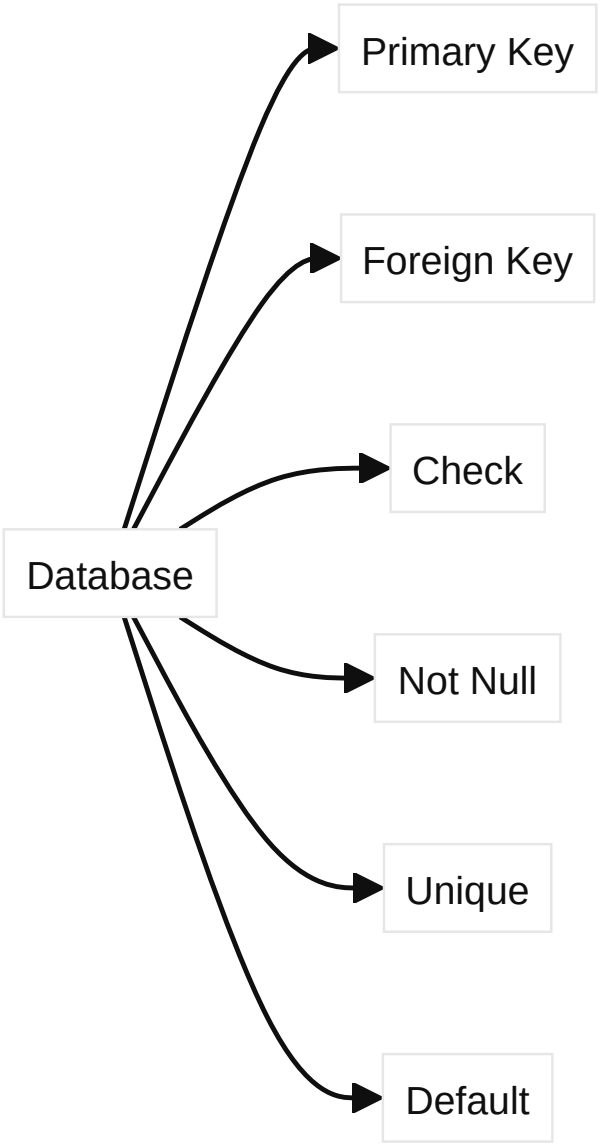
```
RENAME TABLE old new_table;
```

6. Truncate

```
TRUNCATE TABLE table_name -- empties contents
```

Constraints

- Restrictions applied onto the database.



- Basic Constraints in SQL:
  - Key Constraint: Keys can't be duplicated.
  - Entity Integrity Constraint: Primary Key can't be `NULL`.
  - Referential Integrity Constraint: Foreign Key must be a attribute that's already a Primary Key.
- SQL also supports checks,defaults and non nullability

```
-- Setting Primary Keys
CREATE TABLE your_table_name (
    your_column_name INT NOT NULL,
    other_column_name VARCHAR(255),
    PRIMARY KEY (your_column_name)
);
```

```sql
-- Setting a Foreign Key
CREATE TABLE your_table_name (
    your_column_name INT,
    other_column_name VARCHAR(255),
    FOREIGN KEY (your_column_name)
    REFERENCES referenced_table_name(referenced_column_name)
);
```

```sql
-- Setting a Check Constraint
CREATE TABLE your_table_name (
    your_column_name INT,
    other_column_name VARCHAR(255),
    CONSTRAINT constraint_name CHECK (your_condition)
);
```

```sql
CREATE TABLE your_table_name (
    your_column_name INT,
    other_column_name VARCHAR(255),
    CONSTRAINT constraint_name UNIQUE (your_column_name)
);
```

```sql
CREATE TABLE your_table_name (
    your_column_name INT NOT NULL,
    other_column_name VARCHAR(255),
    another_column_name data_type
);
```

```sql
CREATE TABLE your_table_name (
    your_column_name INT DEFAULT your_default_value,
    other_column_name VARCHAR(255),
    another_column_name data_type
);
```

Referential Actions

Delete

```sql
ON DELETE NO ACTION -- default. Delete operation rolls back.
ON DELETE CASCADE -- The rows from the child table are deleted.
ON DELETE SET DEFAULT -- rows in child table are set to their defaults.
ON DELETE SET NULL -- rows in the child table are set to null(foreign key must
be nullable)
```

Update

```
ON UPDATE NO ACTION -- default. Update operation rolls back.
ON UPDATE CASCADE -- The rows from the child table are Update.
ON UPDATE SET DEFAULT -- rows in child table are set to their defaults.
ON UPDATE SET NULL -- rows in the child table are set to null(foreign key must
be nullable)
```

Schema Change Statements

- In MySQL we can change the schema without the need to recompile after every change.

1. DROP

```
-- Drop the schema
DROP SCHEMA db_name [CASCADE|RESTRICT];
-- Drop a table
DROP TABLE table_name [CASCADE|RESTRICT];
```

2. Create table

```
CREATE DATABASE [IF NOT EXISTS] database_name;
```

3. Alter Table

```
-- Add a new column
ALTER TABLE table_name ADD new_column_name column_definition
[ FIRST | AFTER column_name ];

-- Adding multiple columns
ALTER TABLE table_name ADD new_column_name column_definition
[ FIRST | AFTER column_name ],
ADD new_column_name column_definition
[ FIRST | AFTER column_name ];

-- Modify column
ALTER TABLE table_name MODIFY column_name column_definition
[ FIRST | AFTER column_name ];

-- Drop a column
Alter TABLE table_name DROP column_name;
```

```sql
-- Rename using Alter
Alter TABLE table_name CHANGE COLUMN old_name new_name definition;

-- Rename Table
Alter TABLE table_name RENAME TO new_name;

-- Changing Column Defintion
ALTER TABLE table_name MODIFY c CHAR(10);
ALTER TABLE table_name MODIFY j BIGINT NOT NULL DEFAULT 100;
ALTER TABLE table_name ALTER i SET DEFAULT 1000;

-- Add Constraint
ALTER table people add constraint people_gender_fk foreign key (gender)
references gender_tab;

-- Remove Constraint
ALTER TABLE Persons DROP CONSTRAINT PK_Person;

-- Add and remove Defaults
ALTER TABLE DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT;
ALTER TABLE DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT 339;

-- Remove all rows(Delete with no "Where" Clause)
TRUNCATE TABLE table_name;
```

Views

- Virtual tables.

```sql
CREATE VIEW view_name as
Select col1,col2 -- query
from table;
```

- This view is not precomputed.
- The DBMS stores the query and creates the view table on demand.

Advantages

1. Reduces Complexity
2. Increases Security
3. Simplifies queries
4. Columns in views can be renamed without needing to change the database
5. Data Integrity constraints are still met by data inserted via the view
6. Very little storage
7. Logical Data Independence

Disadvantages

1. Insertion will fail if main table has a non-null column that's not in the view.
2. Insertion and Update will fail if columns contain group by statements
3. Cannot change contents if read-only is enabled.
4. Can't create views using temporary tables.
5. Can't pass params to SQL server views.

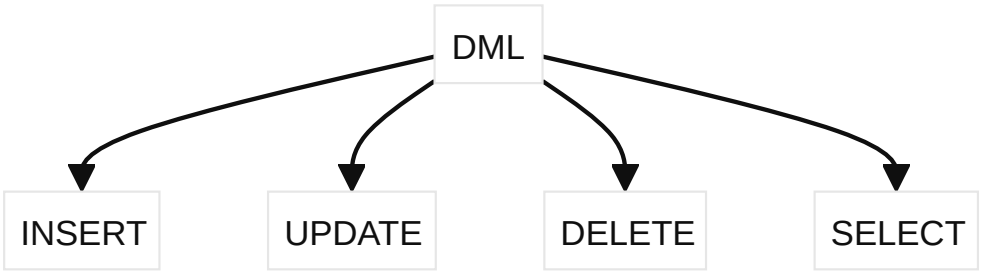
Users and Permissions

```
-- Create new user
Create USER 'Ben'@'localhost' IDENTIFIED BY 'password';

-- Grants
GRANT SELECT
ON lol_db.*
TO 'Ben@localhost';

-- View Privileges
SHOW GRANTS FOR 'Ben@localhost';

-- Give a user every privilege
GRANT ALL ON lol_db.* TO 'Ben@localhost';

-- revoke permissions
REVOKE <privilege>
ON <table/view>
FROM <user>;
```

- Permissions:
    1. Select
    2. Update
    3. Delete
    4. Insert
    5. Create
    6. Alter
    7. Drop
    8. Index
    9. All
    10. Grant

DML

- Changes data in the database.



Insert

```
-- Insert one tuple
INSERT INTO table_name VALUES(...);

-- Specify attributes to insert
INSERT INTO table_name(attr1,attr2...) VALUES(...);

-- Another form of insert
INSERT INTO WORKS_ON_INFO ( Emp_name, Proj_name, Hours_per_week )
SELECT E.Lname, P.Pname, W.Hours
FROM PROJECT P, WORKS_ON W, EMPLOYEE E
WHERE P.Pnumber = W.Pno AND W.Essn = E.Ssn;

-- Backup table
INSERT INTO WORKS_ON_INFO_Backup (select * from WORKS_ON_INFO );
```

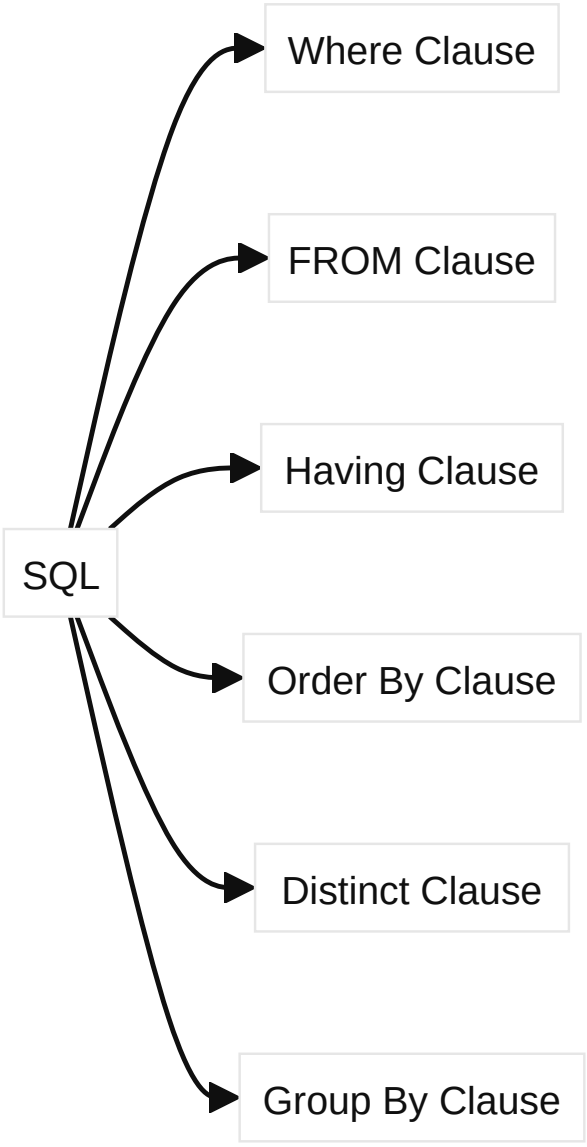
Update

```
-- Syntax
UPDATE table_name SET column_name = value WHERE condition;

-- Example
UPDATE NameList SET Fname='Ben', Lname='Chode' Where Sno=42;
```

Delete

```
-- Syntax
DELETE FROM table_name WHERE condition;

-- Example
DELETE FROM NameList Where Lname='Chode';
```

SQL Clauses



From Clause

- Specifies the table/relation.

> Select clause is basically the projection operation from relational algebra.

```
SELECT * FROM PLACEMENT_DATA;
```

Distinct Clause

- Removes Duplicates.

- Select doesn't remove duplicates by default.

```
SELECT DISTINCT FName, LName from EMPLOYEE;
```

Where Clause

- Conditions that a relation must satisfy.

```
SELECT FName FROM UNIVERSITY WHERE GPA>9.0;
-- Can use <,>,=<,≥,=,≠
-- Can chain conditions using AND/OR/NOT
```

Group-By Clause

- To be used with aggregate functions.

```
select count(ssn),gender from employee group by Gender;
```

Having Clause

- Query returns only when condition to having is true.
- Used with aggregate functions.
- Always used with Group-By.

```
select count(ssn), dno from employee group by dno having count(ssn)>2;
```

Order-By Clause

- To be used with Aggregate functions, Having and Group-By
- Orders in ascending or descending

```
select fname, lname, salary from EMPLOYEE order by salary [asc|desc];
```

String Comparison

- SQL allows for string comparison using the `LIKE` operator.
- `%` matches any substring
- `_` matches any char

```
SELECT * from EMPLOYEE WHERE FName LIKE "_AND_";
```

> Incase we need to match "100%" we write "100%" as `\%` is an escape sequence.

SET Operations

- Used to combine 2 or more `SELECT` statements.
- Set Operations
  1. Union
     - Combines Result of 2 Select statements.
     - Provided they have same arity and datatypes.
     - Removes duplicates.

     ```
     SELECT grade FROM student_course
     WHERE course = 'Physics'
     UNION
     SELECT grade FROM student_course
     WHERE course = 'Mathematics';
     ```

  2. Union All
     - Same as `UNION` But doesn't remove duplicates.
     - If first select returns `c1` duplicates and second returns `c2` duplicates of a particular value, the result will have `c1+c2` occurences of that value.

     ```
     SELECT column_name FROM table1
     UNION ALL
     SELECT column_name FROM table2;
     ```

  3. Intersect
     - Returns entries that belong to both results.
     - Removes Duplicate entries.

     ```
     SELECT column_name FROM table1
     INTERSECT
     SELECT column_name FROM table2;
     ```

  4. Intersect All
     - Same as `INTERSECT` without removing duplicates.
     - Gives `min(c1,c2)` occurences of a particular duplicate value.

     ```
     SELECT column_name FROM table1
     INTERSECT ALL
     SELECT column_name FROM table2;
     ```

  5. Except

- Returns entries in result 1 that's not in result 2.
- Removes duplicates.

```
SELECT column_name FROM table1
EXCEPT
SELECT column_name FROM table2;
```

6. Except All
   - Same as `EXCEPT` without removing duplicates.
   - Result has `max(c1-c2,0)` occurences.

```
SELECT column_name FROM table1
EXCEPT ALL
SELECT column_name FROM table2;
```

Null & Unknowns

- Cases where values can be NULL
  1. Unavailable or withheld
  2. Not Applicable
  3. Unknown
- Arithmetic operations on `NULL` always return `NULL`.
- Comparisons with `NULL` are tough, hence any comparison operation on `NULL` returns a new type called `UNKNOWN`
- Can be checked using the `IS NULL`, `IS NOT NULL`, `IS UNKNOWN` or `IS NOT UNKNOWN` checks.

NULL Values with distinct clause

- In a special case comparison can be done with NULL values.
- This is when `DISTINCT` compares tuples.
- The tuples may have nullable attributes.
- `('A',NULL)` and `('A',NULL)` are treated as equal even though one is NULL.

Nested Queries

- A complete Select-FROM-Where block inside another SQL statement.
- Membership is checked using the `IN` or `NOT IN` operators.
- Additional comparisons can be done using the `comp_op[ALL|SOME|ANY]`

```
...
WHERE GPA>ALL(SELECT GPA FROM merit_list);

=SOME & =ANY -- same as IN operator
<>ALL -- Same as NOT IN operator
```

```
-- In Such cases we MUST name the attributes of the subquery or it will throw
an error
SELECT Dno, avg_salary
FROM (SELECT Dno, ROUND(AVG(Salary),2) FROM EMPLOYEE GROUP BY Dno)
AS dept_avg_salary(Dno, avg_salary) -- name subquery
WHERE avg_salary > 32000;
```

Temporary Relations

- The `WITH` clause can be used to define temporary relations.

```
WITH max_work(max_hours) AS(
SELECT MAX(HOURS) from WORKS_ON
)
Select ...
FROM max_work
WHERE HOURS=max_hours;
```

- The `WITH` clause creates something called a Common Table Expression(CTE).
- This is a temporary entity that exists only for the duration of this query.

Joins

```
-- Natural Join
Select * from t1 NATURAL JOIN t2;
-- if there isn't a common column, it performs cartesian product.

-- Inner Join
SELECT * FROM t1 JOIN t2 ON t1.col1=t2.col1;

-- Outer Joins
-- Left Join
SELECT * from t1 LEFT OUTER JOIN t2 ON t1.col1=t2.col1;
-- Right Join
SELECT * from t1 RIGHT OUTER JOIN t2 ON t1.col1=t2.col1;
```

- SQL Doesn't have a default mechanism for a full outer join. Hence we can do a union of the left and right joins

Index

- Allows database to find those tuples in the relation that have a particular value for an attribute.
- Forms the physical schema
- Improves speed of `SELECT` and `WHERE` but reduces that of `INSERT` and `UPDATE`.
- Helps search large DBs properly.

```
CREATE INDEX index_name on table_name (attrlist)

-- Show indexes
SHOW INDEXES FROM table_name;

-- Drop an index
DROP INDEX index_name on table_name;
```

- Only the queries that use the attributes in the `attrlist` are sped up.
- Any query with an attribute from the `attrlist` will benefit from better speeds.
- Any new addition to the table required the indexes to be recomputed.
- Indexes require more storage.

Roles & Permissions

1. Aggregates permissions into one single entity.
2. Simplifies User management.
3. Roles can be granted and revoked.
4. Roles can contain other roles which make it easier to manage(hierarchical).
5. Has capacity for dynamic and static roles.
6. Role Based Access Control(RBAC) - fine grained control over who accesses data objects.

```
-- create a role
CREATE ROLE awesome_role;

-- assign role to user
GRANT awesome_role TO 'Ben@localhost';

-- assign permissions to role
GRANT SELECT ON table1 TO awesome_role;

-- show existing grants
SHOW GRANTS FOR 'Ben@localhost';
SHOW GRANTS FOR 'Ben@localhost' USING awesome_role;
```

```
-- Revoke
REVOKE SELECT,INSERT from awesome_role ON table1;

-- Drop roles
DROP ROLE role1,role2;
```

User Defined Functions

```
CREATE FUNCTION schema_name.function_name (parameter_list)
RETURNS data_type [DETERMINISTIC|NONDETERMINISTIC]AS
BEGIN
statements
RETURN value
END

-- drop
DROP FUNCTION function_name;
```

- Functions can be either deterministic(returns same value for a combo of input params) or non-deterministic(different values for same input combo).

Stored Procedures

- Precompiled SQL output stored in the DBMS.
- Essentially a sub-routine.

```
DELIMITER && -- temporary change of delimiter
CREATE PROCEDURE procedure_name [IN | OUT | INOUT] parameter_name datatype
[, parameter
datatype]) ]
BEGIN
Declaration_section
Executable_section
END &&
DELIMITER ; -- set delimiter back

-- IN param
CREATE PROCEDURE CUCK (IN var1 INT)
BEGIN
...
END

-- proc call
CALL CUCK(3)
```

```
-- OUT param
CREATE PROCEDURE CUCK2(OUT out INT)
BEGIN
...
END;

-- Proc call
CALL cuck2(@out);
select @out;

-- INOUT param
CREATE PROCEDURE cuck3 (INOUT out INT)
BEGIN
...
END;

-- Proc Call
SET @out = 3;
Call cuck3(@out);
Select @out;
```

Functions	Stored Procedures
Only Input params	Input and Output params
Can't call a stored proc	Can call a function
Can be called from `SELECT`	Can't be called from `select`, `where`, `having`
No Transactions allowed	Transactions can be done
No exception handling	Use try-except blocks
Must return a value	Not mandatory to return value
Only Select operation allowed	All operations can be performed

Triggers & Recursive Queries

- Recursive queries in SQL are used to describe iterative procedures on hierarchical data.
- Recursion is not a native concept in SQL but is just used to describe the part of the iterative process.

```
WITH RECURSIVE cte_name(col_list_output) AS(
-- Anchor Member(Base case of Recursion)
SELECT..
FROM..
WHERE..
UNION ALL
-- Recursive Case
SELECT..
FROM cte_name -- condition referring back to cte
WHERE..
)
```

- Triggers are Event based actions that can be performed in reaction to an event occuring.
- Requires an **Event** and a **Condition**.
- Triggers execute whenever the given event occurs and given condition is satisfied.

```
CREATE TRIGGER trigger_name
(AFTER | BEFORE) (INSERT | UPDATE | DELETE) ON table_name
FOR EACH ROW
BEGIN
--variable declarations
--trigger code
END;
```