SQL – Interact with databases

*Select a Contact on your phone, name from email address book – using DB.

*Google Search, Login, ATM (SQL- rook from 1970s)

*Initially SQL: Language for generating, manupulating & retrieving data from relational DB. [This decade Hadoop, Spark, NoSQL – gained popularity].

Why SQL

* whether use – Relational DB / not – In Datascience, Business intelligence – data analysis we need SQL with Python/R.

Database

*collection of stored data – organized fashion [regardless of data]

*container – (file / files) to store organized data.

* Caution: misuse: Database software.
  Correct: ↳ DBMS – Database Management System

* DataBase: Containers created & manipulated via DBMS

Tables: *file: table [A structured list of data of specific types]

* 'only one type' – eg: order / list of customers (not both)

* Doing so – hard to retrieve

* Every table has a name to identify! (unique: DB name & table name)

* can't reuse names – in the same database [of tables]

Schema: table layout info, DB info & properties.

eg: how data is stored?, what data, how broken up – schema used to describe specific tables with in a DB, as well as entire DB. (& Relationship b/w tables in them – if any)

Columns & data types:

* Column has a particular piece of info

* Single field in a table – All tables have 1 (or) more columns

eg: Customers number, address, ZIP, Customer name! – seperate columns

* possible to sort, filter – breaking up – we can do specific req.

eg: convenient – address stored with house number & street no together.
(we usually don't sort Address – So fine!) – That case: Split!

* each Column has a datatype — numeric
                                — specific datatype

Datatype: A type of allowed data of a column! (Restrict)

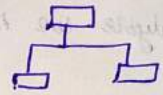
Say: age (only numerical) – prevent alphabetic chars

Datatypes & their names – one of the primary sources of incompatibility; many adv datatypes – not supported!

DB

* more data - time Consuming - Index by first/last name.
* Less accurate - no maintainace (people move on)!
* Computerized - efficient.

Today: handle petabytes of data, accessed by clusters of servers



Non relational DB:

① * Hierarchial system (tree - one/more)
* locate: traverse the tree (customer's)
* each tree node has 0/1 parent - single-parent hierarchy

② * Network hierarchy - expose set of records & links
* Find Customer record - follow the link
* Traverse through chain of account
* Follow the link.

Note: Network hierarchy: accessed from multiple places.
e multi-parent hierarchy

③ Relational model:
* model for Large Shared Data Banks
* Represent as tables.
* Redundant data: links different tables

1	George	Blake
2	Sue	Smith

→ Customer

Cust Id fname lname

Microsoft SQL: allow up to 1024 columns/table

Primary key: (Two/more Column - Compound key)
* Easily uniquely identifiable - eg: CustomerId
* primary key column: never be allowed to change once value-assigned.
* Using foreign keys: access other tables!

Caution: make sure one place - data stored

Say: person changes name.
↳ Reflect at one but not at another.

Refining (ensure - info in only one place) → normalization.

SQL: * Running SQL - embedded Queryhome

* official SQL Standard!
* SQL: Structured Query Language - access & manipulate DB
* ANSI (1986), ISO(1987)

Why?:
execute queries against a DB, retrieve data, insert records, update, delete new records, create new DB, create stored procedure, create views, set permission on table.

SQL - Standard - but (different versions available)

ANSI Std: Support

SELECT
UPDATE
DELETE
INSERT
WHERE

Use SQL in website

* Requires: RDBMS database prog (MS Access, MySQL, SQL server)
* Server side Scripting language (PHP, ASP)
* SQL - get data (query language)
* HTML/CSS - style the page

---

* maintains DBMS - handled structure data - table form
* RDBMS - Relational DBMS [basis for SQL & modern - MS SQL Server, IBM DB2, Oracle, MySQL, Microsoft access.

Why SQL: * Insert, delete, update data in relational DB
* Describe structured data
* Create, drop & manipulate the DB & its tables.
* Creating view, stored procedure & functions in RDB.
* Allows - define data & modify them - set permission.

Components of SQL:

SQL query
↓
SQL Language Processor ← parser + optimizing
↓
DBMS Engine ← File manager + Transaction Manager.
↓
Physical Database

SQL commands:

CREATE - DB, table, table view & other objects of DB
UPDATE - change stored data
DELETE - Erase saved data - Single/multiple tuples
SELECT - Access single/multiple rows from 1/more tables (Also - use with where clause).

DROP - deleting entire table, table view

INSERT - Insert data / records into DB tables [single/multiple rows]

* Schema - predefined, fixed & static
* Scalable - Vertically
* Follow ACID model, Complex queries - easy!
* Not - best in storing hierarchial data
* Require object - relational mapping.

eg: Sqlite, MS-SQL, PostgreSql & MySql

Advantages:

* No programming
* High speed query processing
* Standardized language
* portable
* Interactive
* more than one data view

Disadv

1. cost
2. Interface - Complex
3. partial DB control (Access control)

Primary Keys: Employee table - employee ID | Books - ISBN
Order table - Order ID

"A column - whose values uniquely identify every row in a table"

Without primary key: No way to access a record. [future data manipulation is possible & manageable.

Requirement:

* Unique - no two rows (have same value)
* NULL - not allowed
* Never be modified / updated
* Never reused - If a row deleted - its value can't be reused in future (ID)

Usually : Single columns

Sometimes : multiple columns → All columns should follow above!

Also other type of key: Foreign key.

SQL - (sequel) - Structured query language. - deliberate - does only one things - Few key words - simple, efficient.

Learning SQL - Cope up with vendor specific

* descriptive English - powerful
* Complex & Sophisticated DB operation.

* MySQL
* Microsoft SQL Server Express.

Common Commands

SELECT	ALTER TABLE	INSERT INTO	COMMIT	USE
UPDATE	DROP TABLE	TRUNCATE TABLE	ROLLBACK	
DELETE	CREATE DATABASE	DESCRIBE	CREATE INDEX	
CREATE TABLE	DROP DATABASE	DISTINCT	DROP INDEX	

SQL datatypes

String, Numeric, Date & Time — main classification.

MySQL:

CHAR, VARCHAR, BINARY, VARBINARY, TEXT, TINYTEXT, MEDIUMTEXT, LONGTEXT, ENUM(val1, ...), SET, BLOB ⟶ String

BIT, INT, INTEGER, FLOAT, DOUBLE, DECIMAL, DEC, BOOL ⟶ Numeric

DATE, DATETIME, TIMESTAMP, TIME, YEAR ⟶ DateTime

SQL Server:

char, varchar, text, nchar, nvarchar, ntext, binary, varbinary, image
bit, tinybit, smallint, int, bigint, float, real, money
datetime, datetime2, date, time, timestamp

Operators:

** exponentiation
+, − Identity & negation
*, / mul & div
+, −, || add, sub, string concat
=, !=, <, >, <=, >=, IS NULL, LIKE, ⟶ Comparison
BETWEEN, IN
NOT - logical negation operator
&& / AND — Conjunction
OR — Inclusion.

SET Salary = 20 − 3*5 WHERE Emp_ID = 5

Arithmetic, Comparison, Logical, Set, Bitwise, Unary
(+, −, *, /, %) (= to >=) → &, (|)

AND	BETWEEN	ANY	Union
ALL	IN	LIKE	Union ALL
OR	NOT		Intersect
			Minus

SQL ALL: (Always used with

* SELECT
* HAVING and
* WHERE

SELECT Column_NAME 1, ... Column_Name n FROM
Table_Name WHERE Column Comparison_operator
ALL (SELECT Column FROM tablename2)

SQL AND
SQL OR
SQL BETWEEN

Retrieving Data

SELECT → Retrieve Info [Reserved] → Data not Sorted / Filtered (as of now)

① what you want to Select
② whose you want to select

Retrieve individual Column:

SELECT Prod_name FROM Products;
↓ Column ↓ Table name

get all Columns

SELECT * FROM Customers; (* – Asterisk / wildcard character)

Termination: ;

SQL Statements → 'Not case' sensitive

'SELECT' Same as 'Select'

Convention: uppercase – SQL Keywords
Lowercase – Column & Table names] Read & debug.

Note: names of tables, Col, Values – May be case sensitive
(Reason: Vendor)

White Space

* Ignored !
* use long lines – broke up to many lines → Nothing disappears.

```
SELECT prod_name
FROM Products
```

```
SELECT prod_name FROM Products
```

(or) Read, debug: use!

```
SELECT
prod_name
FROM
Products.
```

Multiple Columns: 'Same SELECT statement' – with Commas

```
SELECT prod_id, prod_name, prod_price FROM Products;
```

Presentation of data: Raw, unformatted & based on DBMS

Data Formatting: presentation concern!

often: Need formatting!

* SELECT * FROM Products; → returns all columns in the table.

Usually returned to an app: which formats!

Note: (*) – don't use unless absolutely necessary.
Retrieving unnecessary data – slows performance.

One advantage: don't explicitly specify names – possible to retrieve columns whose names are unknown.

Retrieve distinct Rows

* Just return not repeating values [single time].

* Say: a customer may purchase many items. we need Customer ID to know Customer count – only once.

Say: 3 unique values

```
SELECT DISTINCT vend_id FROM Products
```

↓

'only distinct values'

Caution:

```
SELECT DISTINCT vend_id, prod_price FROM Products;
```

DISTINCT Keyword applies to all columns – not just one it precedes.

Limiting Results

SELECT Statements return all matched Rows

what if I need: only 1st row / set number of rows.

But: MicroSoft Sql Server: Top Keyword – limit top 5 entries.

DB2 : SELECT Prod_name FROM Products FETCH FIRST 5 ROWS ONLY;

ORACLE: SELECT Prod_name FROM Products WHERE ROWNUM <= 5;

MySQL, MaxDB, Postgre SQL, SQLite

SELECT Prod_name FROM Products LIMIT 5;

Conclusion: W3School uses MySQL (maybe not)

Count no. of distinct elements:

21 ← SELECT COUNT (DISTINCT Country) FROM Customers;
(o/p)

First five rows

SELECT Prod_name FROM Products LIMIT 5;

Any 5 rows (say 5 to 10)

SELECT Prod_name FROM Products LIMIT 5 OFFSET 5;
↓
starting from 5

LIMIT — where to stop

OFFSET — where to start.

Say only 9 rows (No 9th row)

Returns only 4 rows.

Note: ROW 0 → starts from 0 (not) 1

Short hand version

SELECT ProductName FROM Products LIMIT 5, 10;

Offset → 5

10 elem ← limit

Note: This shorthand: reversed!

Same as

SELECT ProductName FROM Products LIMIT 10 OFFSET 5;

Keep in Mind: Not all syntaxes are portable!

Based on DBMS'

Comments

* Need to be embedded in SQL Scripts [create.sql, popuate.sql]
* Another use: Comment out - Debug!

Inline Comment:

```
SELECT Prod_name  -- this is a comment
FROM Products;
```

-- (Common)
(less common)

```
#This is a Comment
SELECT Prod_name
FROM Products
```

Multiline Comment

```
/*  _____
_____ */
```

MySQL

-- Inline
/* ..... */ multiline

3. Sorting retrieved data

* Without Sorting - Random order (Initial order - may be);
* DBMS reuses reclaimed storage space.

Relational DB design theory: Sequence of retrieved data - not assumable until ordering is done explicitly!

Key word: ORDER BY _____ column_name

eg:

```
SELECT Product Name
FROM Products
ORDER BY ProductName;
```

Sort using ProductName

Sorting non selected columns:

Sometimes - we need to sort the retrieved data based on the non retrieved data

```
SELECT ProductName FROM Products ORDER BY CategoryID;
```

↓

Sort by ID - but display ProductName!

Caution:

SELECT → may have multiple columns

ORDER BY → must have last column of the SELECT

↓

MySQL working fine: don't worry!

Sorting by multiple Columns

*eg: Employee list: Sorted by First & last name

useful: when multiple employees: with the same last name.

```
SELECT Prod_id, Prod_price, Prod_name
FROM Products
ORDER BY Prod_price, Prod_name;
```

↳ Sort by price (Same price): Sort by name!

Note: Sorted by Prod_name – when collision (same price) happens.
when all prices are unique: won't be sorted by Prod_name.

Sorting by Column Position

```
SELECT Prod_id, prod_price, prod_name
FROM Products
ORDER BY 2,3;
```

2 – prod_price
3 – prod_name

↳ Initially '2' – If Collision – Sort by '3'

use: Avoids typing,
Risk: what if – wrong! (So use name – possible)!

Note: Can't use this technique, – when SELECT list doesn't have that column (Non-retrieved data)

'Sorting by non-Select Columns' – Not possible,

Specify SORT direction

descending order:

```
SELECT Prod_id, Prod_price, Prod_name
FROM Products
ORDER BY Prod_price DESC;
```

Multiple Columns – most expensive first

* Ascending Sort by Name

```
SELECT Prod_id, price, name
FROM Products
ORDER BY Prod_price DESC, Prod_name;
```

DESC → descending
Ascending →

Note: DESC only applies to preceding row – not to all,

For multiple Sort – descending order: must use DESC at each Column name!

DESC / DESCENDING — can be used
ASC / ASCENDING — default - no need to specify!

* Note: 'a', 'A' → Same? → Depends upon DBMS
* most DBMS → Dictionary sort order 'A' same as 'a'
* But most good DBMS enables — admin to change this behaviour (if req)

Filtering

* Retrieve Subset — specific search criteria

'WHERE' → specified after table name

```
SELECT Prod_name, Prod_price
FROM Products
WHERE Prod_price = 3.49;
```

→ Simple equality test.

How many zeros:

* 3.49, 3.490, 3.4900 → DBMS specific [default data type behaviour]
* mathematically identical!

SQL (vs) Application filtering:

case: Data sometimes filtered at client app level, not in DBMS - So in this case SELECT — Just gives required info (whole column)
By looping: app filters data!

'This practice - strongly discouraged'

why: DBMS optimized to perform filtering quickly & effectively - improve app performance - scaling! [waste of network: prevented]

why? Data (whole) — more bandwidth — wasted!

Caution

* ORDER BY must come after WHERE [∵ sort retrieved data]
* First retrieve then sort!

Where clause operators

=, < > → Non equality
!= Non equality
< | >
<= | >=

! < Not less than
! > Not greater than

BETWEEN — B/w two specified values
IS NULL — Is a NULL value?

Not all operators are supported by all DBMS!

MySql → =, >, <, <=, >=, <>, BETWEEN, LIKE, IN
- LIKE → Search for a pattern
- IN → specify multiple possible values for a column
- <> interchangeably → != , ISNULL

```
WHERE Prod_price < 10;
WHERE vend_id <> 'DLL01';
```
→ Single quote delimit a String

Checking for a range of values:

```
SWHERE Prod_Price BETWEEN 5 AND 10;
```
(5 = low, 10 = high)

Checking for No Value:

* Empty or an empty String or Just space
* Designer can specify whether any columns can have no value
* when a column has no value - Said to contain a NULL value.

NULL

Can't do: `If = NULL`

Instead: IS NULL Clause.
↓
Part of a query - allows to filter / customize.

```
SELECT Prod_Name
FROM Products
WHERE Prod_Price ISNULL;
```
→ return prod_Name with NULL Prod_Price

* 0 → doesn't mean NULL
* No NULL values → empty list returned!

DBMS specific operators

* Many DBMS - extend the std. set of operators, — Advanced filtering.

NULL and Nonmatches

* Search for Something (particular value) ≠ Not found - NULL (not returned)!
* NULL not returned - as we expect!
* Rows with NULL in the filter column - not returned when filtering for matches / when filtering for non matches.

Advanced Filtering — Combine WHERE clause

* Multiple WHERE clauses — AND, OR [logical]

```
WHERE Condition1 AND Condition2 AND Condition3 ..... ;
WHERE Condition1 OR Condition2 OR Condition3 ..... ;
WHERE NOT Condition;
```

eg: Country = 'Germany' AND City = 'Berlin';

Also Combine Logical operators

ORDER BY Country ASC, CustomerName DESC

↳ Any collision Sort by DESC based on CustomerName.

---

* WHERE Vend_Id = 'DLL01' AND Prod_Price <= 4;
* WHERE Vend_Id = 'DLL01' OR Vend_Id = 'BRS01';

order of evaluation

WHERE Vend_Id = 'DLL01' OR Vend_Id = 'BRS01' AND Prod_Price >= 10;

First evaluated!

* AND — high precedence than OR

Note: use brackets — parenthesis (if necessary)

* For changing precedence — group using ( ) — No downside — removes ambiguity.

IN Operator

* Range of conditions — any of which can be matched.

WHERE Vend_Id IN ('DLL01', 'BRS01') ⟶ Same as OR but shorthand.

ORDER BY Prod_name;

Advantages:

* cleaner syntax, easy to read — manage — order of eval easy (manage)
* execute more quickly!
* IN operator can have another SELECT statement — highly dynamic WHERE clauses.

NOT Operator

*Negates

WHERE NOT Vend_Id = 'DLL01'

ORDER BY Prod_name;

* NOT – only useful in Complex Expressions!
* eg: Conjunction with an IN operator – makes it simple to find all rows that don't match a list of criteria.

WILDCARD filtering – LIKE

* Sophisticated filtering of retrieved data.
* All previous operators: values used in filtering: known!

eg: Search for all products – having the text – 'bean bag' – within the product name?

'wild card Searching'

* Create patterns – Compare against the data 'Search pattern'

Search pattern: Made up of literal text, wildcard characters / any combination of them.

---

* wildcards – actually characters of special meaning
* To use wildcards – must use 'LIKE' clause. within SQL WHERE clause
* LIKE – Tells DBMS – Search pattern – Compare using wildcard match rather than straight equality match!

Predicate:

* when is an operator – not an operator – predicate
* LIKE – predicate – not an operator
* End result same – Just aware – documentation.

Note: Wildcard Searching – only used with text fields (Strings)

% Sign – wild card:

* % – means match any no. of occurances of any char.
* eg: Start with Fish

```
SELECT prod_id, name FROM Products
WHERE Prod_name LIKE 'Fish%';
```

% means – Irrespective of further char
Accept any char after the word 'Fish'

Case sensitivity: may be / may not be

Note: multiple wild cards can be used!

WHERE prod_name LIKE '%bean bag%'

Anything Anything

WHERE prod_name LIKE 'F%y' — start with F, end with y

Partial email address

WHERE email LIKE 'b%@forta.com'

→ start with b
→ end with @forta.com

% → zero/one/more characters — neglected until @forta.com.

watch for trailing zeros

* pad contents with space — Some DBMS (say: column expects 50 char)

F bean bag toy → 17 char

* 33 spaces appended — no real impact on data — But 'LIKE' affected!

∴ It ends with space not with 'y' → even though!

* Solution: append % → 'F%y%' → But not so efficient!

Better Solution: trim spaces! — then use

NULL:

% → Can't match NULL

WHERE prod_name LIKE '%' → Not matched!

Underscore (_) wildcard:

* (_) matches just a single character (not multiple)
* (_) not supported by DB2.

WHERE prod_name LIKE '__ inch teddy bear';

12
18] anything!

* 8 inch → Can't satisfy.
* why: (_)(_) 2 underscores — Requires 2 char
* So 8" → taken (not matching)!

WHERE name LIKE '% inch teddy bear';

↓

Any char before inch!

Unlike % ⟶ (_) only matches one character (Not zero)!

'No more / No Less'

Brackets ([]) wildcard:

* Supported by Microsoft SQL Server
* Not by MySQL, ORACLE, DB2, SQLite.

Specify set of char — match char in specified position — location of wildcard.

WHERE Cust_Contact LIKE '[JM]%'

↳ Starts with J (or) M.

[^JM] ⟶ except J (or) M.

Same as

WHERE NOT Cust_Contact LIKE '[JM]%'

Tips: Powerful — price: longer search time

* Don't overuse
* Not to use at the beg of search pattern — unless very necessary. (Slowest to process.
* misplaced wildcards: Wrong answer!

Calculated fields

* Formatting — say: display name of Company with location
* City, State, Zip — retrieved as one
* mixed upper / lower — need: uppercase.
* Invoice: needs quantity, price — expanded price (quant * price)
* Total, average / other calc based on table data.

Retrieve — Convert — calculate — Reformat data directly from DB.

Calculated fields

* Calculated fields — doesn't already exists in DB tables
* Rather — on the fly within a SQL SELECT statement.

Field - Column

* Note: DB knows which one actual fields & calculated fields.

* Client perspective: Calculated fields data - returned in the same way from any other column.

Client (vs) Server Formatting

* Formatting can be done in client app

* DB : efficient & faster.

Concatenating Fields: Join to form single long value.

Join two columns (+, ||, Concat (MySQL))

Note: W3 schools : ||

```
SELECT vend_name || '(' || vend_country || ')'
FROM Vendors
ORDER BY vend_name;
```

Note: W3Schools : No space padding

o/p: Single column.

Trim spaces: RTRIM

```
SELECT RTRIM(vend_name) || '(' || RTRIM(vend_country) || ')'
FROM Vendors
ORDER BY vend_name;
```

RTRIM:

* Trims all spaces from the right of a value.

* Individual columns - Trimmed properly.

RTRIM - Right side
LTRIM - Left side
TRIM - Both sides

Using Aliases

* Name the new field (concatenated) - for using it further - must!

"Column aliasing" - Alternate name for a field or value.

Keyword 'AS'

```
SELECT RTRIM (vend_name) || '(' || RTRIM (vend_country) + ')'
AS vend_title
FROM vendors ORDER BY vend_name;
```

Use of AS – not necessary – But good practice (in many DBMS)

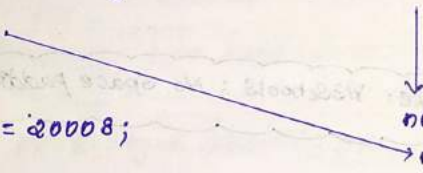
Uses of Aliasing:

* Renaming Column – Say original name – ambiguous / misread.
* Single word – no need for single quotes – Strongly discouraged!
* This approach creates problems!

> Aliases: also known as derived columns
> But mean the same thing

Math Calculations:

```
SELECT Prod_id, quantity, item_price, quantity * item_price
AS expanded_price
FROM OrderItems
WHERE Order_num = 20008;
```



*, -, +, /

Curdate() → returns current date and time [MySQL, MariaDB]

'Not working'

```
SELECT Curdate();
```

Data manipulation functions

* SQL functions – highly problematic.
* DBMS specific – very few (portable)

> 'Function name / Syntax – varies'

Common:

Extract part of a String: SUBSTR() → DB2, Oracle, PostgreSQL, SQLite
SUBSTRING() → MySQL, SQL Server, MariaDB

Datatype Conversion: CAST() → DB2, Postgre, SQL Server
CONVERT() → Maria, MySQL, SQL Server

Current date → CURRENT_DATE → DB2, PostgreSQL
CURDATE() → MariaDB, MySQL
GETDATE() → SQL Server, GETDATE() → SQL Server
DATE() → SQLite.

* Approach: write your own (App works harder - which DBMS does efficiently)
* Make sure - comment well - :: others devs know which implementation

most SQL: trimming, padding, converting values to upper/lowercase.
Returning absolute value, algebraic calculation.
date, time functions
formatting functions - user friendly o/p s. [eg: data, time-format]

we can use some functions not only with SELECT, but with WHERE

Text manipulation functions:
RTRIM(), UPPER, LOWER()

UPPER (Shippers Name)
SELECT LOWER (Shippers_Name) FROM Shippers;

LEFT() → returns chars from left of string
LENGTH() → length of string
LOWER()
LTRIM()
RIGHT() → returns chars from right of string
RTRIM()
SUBSTR() | SUBSTRING()
SOUNDEX() → Returns String's SOUNDEX value
UPPER()

SOUNDEX() → Algo - converts text into an alpha numeric pattern describing phonetic rep of that text

Not supported by postgre SQL ←

* Takes into account: Similar sounding chars, syllable.
enable: Compare by sounds - Not a SQL concept!

SQLite: compile time option enabled - else - won't work!

eg: Michelle Goreen
Michael Goreen
] search won't work!

```
SELECT Cust_name, Cust_Contact
FROM Customers
WHERE SOUNDEX(Cust_Contact) = SOUNDEX('Michael Goreen');
```

Sound alike - matched

Date & time manipulation

* DBMS - uses special Variables - Date & time (For optimized search, sort..)
* Inconsistent, least portable.

Sql server:

```
SELECT order_num
FROM Orders
WHERE DATEPART (yy, order_date) = 2020;
```

Not working: DATEPART, DATE_PART, EXTRACT, BETWEEN to_date (...) AND to_date (....)

to_date() → oracle

Result: SqlLite

```
Strftime ('%Y', OrderDate) = '1997'
```

Numeric manipulation functions — most uniform & consistent

ABS(), COS(), EXP(), PI(), SIN(), SQRT(), TAN()

Summarize data

* Summarize - without retrieving it all
* Summarize - Analysing and reporting purposes

Say: no. of rows / with some condition
Sum of rows
min, max, avg

→ else waste of time & bandwidth

Five aggregate functions:

* enumerated - pretty consistent

Aggregate func: Calculate & return a value.

AVG(), COUNT(), MAX, MIN(), SUM()

```
SELECT AVG (Prod_Price) AS avg_price
FROM Products;
WHERE Vend_Id = 'DLL01'
```

→ Retrieve avg of the data belong to DLL01

Caution: AVG() → takes specific column name as I/P.
For multiple columns → use multiple AVG()

Note: NULL - ignored by AVG()

COUNT() – Counts number of rows.

COUNT(*) → All rows, in a table → Can be NULL

COUNT(column) → Ignores NULL.

```
SELECT COUNT(*) AS num_Cust
FROM Customers;   -- 5
```

```
SELECT COUNT(Cust_email) AS num_Cust
FROM Customers   -- Ans: 3
```

MAX() → Spec Column.

```
SELECT MAX(Prod_Price) AS MaxPrice FROM Products;
```

* MAX() → used with highest numeric or date values (DBMS specific) text also!, Ignores: NULL

MIN() → opposite of MAX()

Sorted text MIN() → returns 1st one string
MAX() → last string

SUM() → Total

```
SELECT SUM(quantity) AS Items_ordered FROM OrderItems
WHERE Order_Num = 20005;   -- 200
```

Say: Total price (quantity * indiv price)

```
SELECT SUM(Item_price * quantity) AS total_price FROM Orders
WHERE Order_Num = 20005;
```

All aggregate functions: used with multiple Columns

Distinct values:

1. perform calculations on all rows, ALL default keyword.
2. only unique values – specify – Distinct.

```
SELECT AVG(DISTINCT Prod_Price) AS Avg_Price FROM Products
WHERE Vend_Id = 'DLL01';
```

↳ 'multiple lines with same lower price' – excluding them – Avg will be higher.

caution: No DISTINCT with COUNT(*)

why? multiple columns – not possible – vague!
must not used with a calculation/expression!] use with column name alone.

DISTINCT with MIN(), MAX()

* No value: why: min value will be always min value (even though multiple copies) – waste of time!
* Additional Aggregate Arguments: TOP, POP PERCENT – Refer DBMS.

Combine Aggregate functions

```
SELECT COUNT(*) AS num_items,          ⟶ 9
       MIN(prod_price) AS price_min,   ⟶ 3.4900
       MAX(prod_price) AS price_max,   ⟶ 11.9900
       AVG(prod_price) AS price_avg    ⟶ 6.823333
FROM Products;
```

Try not to use column names [Sometimes - error]

Grouping data

No. of products offered by each vendor

Products offered by vendors - who offer a single product - who offer more than 10 products.

'groups' → divide data into logical sets so can perform aggregate calc on each group.

Keyword: GROUP BY

```
SELECT vend_id, COUNT(*) AS num_Prods
FROM Products
GROUP BY vend_id;
```

GROUP BY → Instructs Sort data, group it by vend_id

∴ Actual 3 products vend_id BRS01 ┐ O/P → BRS01 3
4 " vend_id DLL01 ┘ DLL01 4
2 " " FNG01 FNG01 2

Note: GROUP BY vend_id → 3 groups → 3 vend_id
For each group Aggregate data type COUNT - Calculated

Adv: No need to Calculate specifically.

Advantages:

* Nest groups - more Control over grouping data
* All the columns specified - evaluated together → [GROUP BY clause]
* Every column in GROUP BY → must be retrieved by SELECT
  Not ALIASES can be used!
* Most SQL Implementations - don't allow GROUP BY Columns with Variable length datatypes (text / memo fields)
* Aside from Aggregate Calculation statements - every column in SELECT must be present in the GROUP BY clause.
* NULL value (Row value of a Column) → NULL returned as group - All NULL groups - grouped together.

GROUP BY → must come after WHERE
Before ORDER BY clause

* Microsoft SQL Server: Support optional ALL clause within GROUP BY
* This clause → return all groups → even those → no matching rows → In this case aggregate would return NULL.

Filtering groups

* In addition to group data → we can filter the grouped data.
* List of all customers → Atleast 2 orders.

WHERE → doesn't filter specific groups → Just rows

→ Keyword: Having

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id HAVING COUNT(*) >= 2
```

WHERE filters → Before grouping
HAVING filters → After data is grouped.

* 'Rows → eliminated by WHERE' → won't be included (group) * So we need 'Having' clause.

Both WHERE and HAVING

* Any customer → placed 2/more orders in the past 12 months
  1) past 12 months → customer → WHERE
  2) 2/more orders → HAVING

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE Prod_price >= 4 ——→ (1) Product price greater than equal to 4
GROUP BY vend_id
HAVING COUNT(*) >= 2; ——→ (2) Atleast 2 products by the customer from Products ≥ 4 price.
```

* Most DBMS Consider HAVING and WHERE as same → if no GROUP BY is specified.

Note: GROUP BY → use HAVING
Row-level filtering → use WHERE

Grouping & Sorting

* ORDER BY – Sorts – even not selected Rows/Columns ; Aggregate func – never req

* GROUP BY:
  * group rows – only selected Columns – may be used & every selected Column expression must be used. – may/may not be sorted (group)
  * Required if using Columns (multiple) with aggregate functions.

---

*Sometimes: we need different kind of Sorting – not (Sort by groups)
* So specify ORDER BY – whenever you use GROUP BY

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3;
ORDER BY items, order_num.
```

I want Sort by items not by order_num

Without Sort

order num	items
20006	3
20007	5
20008	5
20009	3

With Sort

order num	items
20006	3
20009	3
20007	5
20008	5

Clause Ordering

SELECT	– Columns/exp – returned	– Yes (required)
FROM	– Retrieve from table	– Use only when data got from table
WHERE	– Row level filtering	
GROUP BY	– Group specs	– Only if calc. Aggregates by groups
HAVING	– group level filtering	– No (Use when necessary)
ORDER BY	– o/p Sort order	– No

---

11. Sub queries

Query: eg: SELECT – SQL statement

> SQL Allows: Subqueries – queries embedded in to other queries

Note: we use: Relational tables

Orders → table 1 → Orders (Cust ID, order num, order date) → No customer info
Orders → table 2 → OrderItems (order items)

↓ stored by

Customers table

Say: 1. Retrieve the order numbers of all orders - Containing Item 'RGAN'

2. " all customers - who have orders listed in the order numbers returned in the previous step.

3. Retrieve the customer info - for all the customer IDs - (2) Step

---

* Orders - order num, Orderdate, Cust_Id
* All Customers of the previous table.
* get Customer info from Customers (cust Id, name, city, address, State, zip, country, Contact, email)

3 queries - depends on each other's

---

* Using the return statement returned by one Select Statement - populate the WHERE clause of the next SELECT statement.

1) SELECT order_num FROM OrderItem WHERE Prod_Id = 'RGANO1'

o/p:

order_num
20007
20008

2) SELECT Cust_Id FROM Orders WHERE order_num IN (20007, 20008);

cust_Id
1000000004
1000000005

Combine 1 and 2

```
SELECT Cust_Id
FROM Orders
WHERE order_num IN (SELECT order_num FROM OrderItems
                    WHERE Prod_Id = 'RGANO1');
```

Subqueries: Always processed starting with in the innermost SELECT statement & working outward.

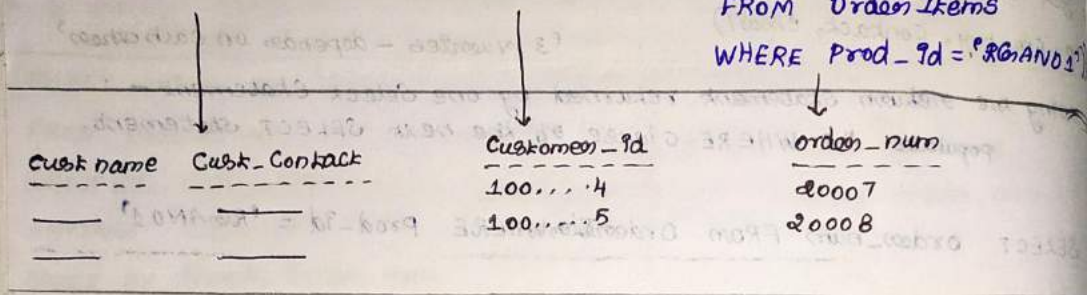
Formatting SQL:

* Read & Debug - Complex! - Breaking up over multiplines
* Intendation, colour coding! (highlight Syntax)

3) SELECT Cust_name, Cust_Contact FROM Customers
WHERE Cust_Id IN (100... 4, 10... 5);

Combining

```
SELECT Cust_name, Cust_Contact
FROM Customers
WHERE Cust_Id IN (SELECT cust_Id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE Prod_Id = 'RGAN01'));
```



Cust name	Cust_Contact
____	____
____	____

Customer_Id
100....4
100....5

order_num
20007
20008

Caution: Subquery SELECT can retrieve only one SELECT

Using Subqueries: Not an efficient way

Efficient way : Joining tables

Use Subqueries as calculated field

Case: Total number of orders placed by every customer_Id in Customers table

① Customer list from Customers table

② For each customer retrieved – Count no. of orders associated (Orders)

```
SELECT COUNT(*) AS orders
FROM Orders
WHERE Cust_Id = 1000...1;
```

For each customer

```
SELECT Cust_name, Cust_State, (SELECT COUNT(*)
                               FROM Orders
                               WHERE Orders.Cust_Id =
                                     Customers.Cust_Id) AS orders
FROM Customers
ORDER BY Cust_name;
```

Both lists!

(So Compare each element with every other element)

```
SELECT cust_name, cust_state, (SELECT COUNT(*)
                               FROM Orders
                               WHERE Orders.cust_id =
                                     Customers.cust_id) AS orders
From Customers
ORDER BY cust_name;
```

cust_name, cust_state → From Customers

Selects Counts

* Customers.cust_id has all customers
* Orders.cust_id has purchased customers!

→ get count, display it w.r.t customer details.

Orders.cust_id = Customers.cust_id

Remove ambiguity.

'Take care of Ambiguous column names'

Sometimes: Common names b/w table (fields) – multiple tables

more than only one table – use syntax.

'JOINS – Best way'

Joining tables – JOINS

* Join tables – on the fly with data retrieval queries – important
* Relational tables
  - → Product list – description, price, vendor info. Inconsistent Update – tough
  - → No need for multiple times mentioning – vendor details – Redundancy
  - → New table: (multiple table) – Related using common values

    Say: vendor info, Product info

* Primary key: Unique: anything – vendor id (In vendor info table)
  - * using vendor Id – (from product info) – Refers vendor info

Use: Vendor info never repeated – Space saving

multiple – chance of error – hard to update

Consistent – non repeating data – easy to report, manipulate!

Scale: Handle increasing load – without failing.

Joins

* Benefit with a price — how to retrieve — multiple select statements.
* Join tables — on the fly (Associates correct Rows in each table on the fly)

Interactive DBMS tool

* JOIN - not a physical entity [virtual] - Created by DBMS as needed - persists until query execution. (GUI - some DBMS - define relations interactively)
    * Relational table — make sure — valid data!
    * Invalid vendor ID — No way home! (Allow only valid values)
    * Referential Integrity: DBMS enforces — data Integrity rules

Data Integrity:

Entity Integrity: No duplicate rows

Domain Integrity: Enforces valid columns — restrict type, format, range of value

Referential Integrity - Rows can't be deleted (if used by other records)

User-defined Integrity - Specific rules that don't fall under the

Database normalization:

* Eliminate redundant data - Store same data (avoided)
* Ensure data dependencies make sense. (Logically stored)

guidelines:

* First Normal Form (1NF)
* Second Normal form (2NF)
* Third Normal form (3NF)

1NF: Organized database

* Define data items rec — they became columns in a table - (locate field
* place related data items in a data
* No duplication
* Ensure - primary key.

2NF: All rules of 1NF - no partial dependencies

Primary key: Customer name, customer product

Say: Same name, same product — ?

Cust_Id, Order_Id → primary key

Assuming: Same customer — hardly orders same product.

Partial dependency : ORDER_ID → unique

↳ CUST_NAME , CUST_ID

Partial dependency

* No link b/w Customer_id & what he purchases
* order detail, purchase date dependent on order ID
* CUST ID, order detail → Independent.

---

* Customer, Order, Customer Order
  - Customer: Id, name
  - Order: id, detail
  - Customer Order: cus id, order id → Datetime

3NF: 3rd normal form, All non primary keys – are dependant on primary key.

Cust_ID → Independent of cust id, name, DOB, Street, City, State, ZIP, email_ID

(primary key)

* Street, zip, City, State → Address (dependency)
  Separate table

Customers table:

Customers → ID, Name, DOB, ZIP, EMAIL_ID, primarykey (Cust_ID)

JOIN

*Specify all the tables to be included and how they are related to each other.

SELECT vend_name, prod_name, Prod... Price ——— two tables

FROM Vendors, Products

WHERE Vendors.vend_id = Products.ven:_id; → when product vend_id in vend_id

display: when vendor id in product id → match vendid with product vendid

Vendors.vend_id → 'Remove ambiguous'

Importance of WHERE:

All the tables – 2 tables
How related to each other?

WHERE

* why WHERE clause – to set the join relationship
* when tables are joined in a SELECT – relationship constructed on the fly.
* 'Nothing in DBMS' – to join the tables!

Insert Into

Create new records in a table

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value 1, value 2, value 3, ...)
```

Note: If we add values to all column — no need for Column

```
INSERT INTO table_name
VALUES (value 1, ....)
```
→ All column values

Insert only in Specific Columns

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```
→ CustomerID (auto-increment field)

Sql Null Values

Optional field — value (not) mentioned — NULL (default from zero or a field having space) — NULL - One left blank during record creation.

Test for null: IS NULL
IS NOT NULL

Update

```
UPDATE table_name
SET column 1 = value 1, column 2 = value 2, ....
WHERE condition;
```

eg:

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City = 'Frankfurt'
WHERE CustomerID = 1;
```

Multiple Records

```
UPDATE Customers
SET ContactName = 'Juan'
WHERE City = 'Mexico';
```

Caution: Omit City → All Rows updated!

DELETE

```
DELETE FROM table_name WHERE condition;
```

'without WHERE – all records will be deleted' ← without where

```
DELETE FROM Customers
WHERE CustomerName = 'Alfreds Futterkiste';
```

Select Top

```
SELECT TOP 3 * FROM Customers;
```
→ Top 3 all Columns — SQL like

```
SELECT Column_name
FROM table_name
WHERE Condition
LIMIT number;
```

MIN(), MAX()

```
SELECT MIN(column_name) FROM table_name
WHERE condition;
```

COUNT(), AVG(), SUM()

LIKE

Specified pattern (%, _)

```
SELECT col1, col2 ....
FROM table_name
WHERE columnN LIKE pattern;
```

a% → start with a
%a → end
%or% → or in any position
'_r%' → r in second position
'a_%' → a in 1 pos, atleast 2 chars
'a%o' → start with a, end with o

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a%'
```

SQL wildcards

Not all DBMS supports all – use (MySQL) – %, _ enough.

[ ], [^] ✓ → Not SQL(My) , [a-f]

```
SELECT Column_name
FROM table_name
WHERE Column_name IN (value1, value2);
```
NOT IN

```
WHERE Column_name BETWEEN
_ AND _ ;
```
NOT BETWEEN

SQL Aliases: – give a temporary name for table / Column. – Readable

'Life time: upto the duration of that query'

'AS'

```
SELECT column_name AS alias_name
FROM table_name;
```

```
SELECT column_name
FROM table_name AS alias_name;
```

Multiple Alias

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person];
```

Not encouraged ← double quotes / braces!

Concatenate - || (or) +

```
SELECT CustomerName, Address + ',' + PostalCode + ',' + City + ',' + Country As Address
```

MySql: `CONCAT(—, ',', ',', —) AS Address`

Alias for tables

```
SELECT O.OrderID, O.OrderDate, C.CustomerName
FROM Customers AS C, Orders as O
WHERE C.CustomerName = 'Around' AND C.CustomerID = O.CustomerID;
```

Product (order Id).

↳ Customers who order

* Name 'Around' and must ordered

Sql Joins - Contd

```
SELECT Vend_name, Prod_name, Prod_Price
FROM Vendors, Products
WHERE Vendors.Vend_ID = Products.Vend_ID;
```

Joining Columns from 2 tables

what happens without WHERE

Vend_name | Prod_name | Prod_Price

what happens when Joining 2 tables

'pairing every row with every other row.' of second table

what I need

* merge only corresponding rows
* say Vendor_Id → match Vendor_Id of 1st table with Vendor_Id rows of 2nd table!

→ `WHERE Vendors.Vend_Id = Products.Vend_Id;`

Cartesian product (Cross Join)

* The results returned by a table relationship - without a join condt (WHERE) - no. of rows retrieved is equal to

> No. of rows in table 1 x no. of rows in table 2.

'Mostly - we don't want this?

'Don't forget the WHERE clause' — 'No filter' - gives all combinations

for JOIN — No Keywords — do it by myself.

* Inner Join : Left JOIN RIGHT JOIN FULL OUTER JOIN

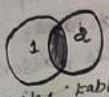


table 1 table 2

INNER JOIN

INNER JOIN: — So far using — equiJoin / Inner Join — we can specify explicitly.

```
SELECT Vend_name, Prod_name, Prod_Price
FROM Vendors
Inner JOIN Products ON Vendors.vend_id = Products.Vend_Id;
```

Difference: FROM clause — we specify relationship — using INNER JOIN

'ON' clause used instead 'WHERE' clause

Condition passed to 'ON' same as 'WHERE'

```
SELECT Column_name FROM table 1
INNER JOIN table 2
ON table 1.column_name = table2.column_name;
```

```
SELECT Orders.OrderID , Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

'Inner Join — selects all rows — from both tables as long as match b/w the columns. No match: won't shown

Note: INNER JOIN Continued with FROM Orders!

Three tables

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

Right Syntax: INNER JOIN preferred, But SQL purists — use Other way.

Use — whichever comfortable.

multiple tables

```
SELECT Prod_name, Vend_name, Prod_price, Quantity
FROM OrderItems, Products, Vendors
```

WHERE Products.vend_id = Vendors.vend_id
AND OrderItems.prod_id = Products.prod_id
AND order_num = 20007;

Products, Vendors, OrderItems
Same vendor id
Same prod_id (Corresponding)

Vendors: id, name, state, add, city, country, zip
Products: prod_id, vend_id, prod_name, prod_price, descrip
OrderItems: order_num, order_item, prod_id, quantity, item_price

* Performance: Resource intensive – avoid joining tables unnecessarily – more (joins) – more performance – degrading!
* while SQL has no constraint – JOIN ing tables (many DBMs has) – but try to avoid (use when abs necessary)

Without Subqueries – using JOIN

SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
AND OrderItems.order_num = Orders.order_num
AND prod_id = 'RGAN01';

↓ 2 Joins!
↳ 'Same as 3 queries (Sub)
check for 'RGAN01'
Connect tables

* More than 1 way to perform SQL operation – rarely right/wrong way
* choose which one is best!

This case: cust_id both in Customers & Orders
Always not the case!

Advanced Joins

Using table Aliases:

SELECT RTRIM(vend_name) + '(' + RTRIM(vend_country) + ')'
AS vend_title
FROM Vendors
ORDER BY vend_name

enables aliasing tables

1. Shortening syntax
2. multiple use eg the same table within a single SELECT statement

SELECT cust_name, cust_contact
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
AND OI.order_num = O.order_num
AND prod_id = 'RGAN01';

'No need for AS'
Oracle: Customers C

Left Join

* All records from table 1, and the matching records from table 2.
* No match: 0 records from table 2.

SELECT column_name
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;

multiple tables (2 – eg)

SELECT Customers.cust_name, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID.
ORDER BY Customers.CustomerName;

Right Join

* All from table 2 and matching from table 1.
* No matching: 0 records from table 1.

SELECT colname
FROM table
RIGHT JOIN table2
ON table1.col = table2.col

FULL OUTER JOIN / FULL JOIN – all records – when there is a match b/w two tables

SELECT col_name
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.col_name
WHERE condition

SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customer.CustomerName;

Self Join – table joined with itself

SELECT column_name
FROM table1 T1, table1 T2
WHERE condition

T1, T2 – aliases eg table1

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City
```

→ Same town diff persons!

Union – Union (two tables – same no. of columns)

```
SELECT Column_name(s) FROM table1
UNION
SELECT column_name FROM table2
```

By default: distinct values
Allow duplicate: UNION ALL

```
SELECT City FROM Customers
UNION                          → each city listed once (distinct)
SELECT City FROM Suppliers
ORDER BY City;

SELECT City, Country FROM Customers
WHERE Country = 'Germany'
UNION                          → discrete (default)
SELECT City, Country from Suppliers
WHERE Country = 'Germany'
ORDER BY City;
```

Another eg

```
SELECT 'Customer' AS Type, ContactName, City, Country
FROM Customers
UNION
SELECT 'Supplier', ContactNumber, City, Country
FROM Suppliers;
```

→ 4 columns

o/p

Type	ContactName	City	Country
Customer	—	—	—
Supplier	—	—	—

* Here we have given String in a SELECT statement – makes all row as that String!

GROUP BY

* Summarize rows – find no. of (each customers) in each Country.

* GROUP BY → often used with Aggregate Functions

```
SELECT Column_name
FROM table_name
WHERE condition
GROUP BY column_name
ORDER BY column_name
```

```
SELECT COUNT(CustomerID, Country)
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS
    NumberOfOrders FROM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

Having – Since WHERE can't be used with aggregate func

```
SELECT colname
FROM table
WHERE condition
GROUPBY colname
HAVING condition
ORDER BY colname
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
ORDER BY COUNT(CustomerID) DESC
```

INNER JOIN

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM ORDERS
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

EXISTS – test existence of any record

(subquery) – True/false

```
SELECT colname
FROM table
WHERE EXISTS
(SELECT colname FROM table WHERE condition)
```

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.
              SupplierID = Suppliers.SupplierID AND Price < 20);
```

'List of suppliers with price less than 20'

ANY and ALL

* ALL – boolean result (TRUE - All of the subquery value meet condition)
  Used with SELECT, WHERE & HAVING

* SELECT ALL Colname
  FROM table
  WHERE Condition;

SELECT Col FROM table ALL
(SELECT Col FROM table WHERE Condition)

Operator: =, <>, !=, >, >=, <, <=

ProductName - Any records in orderDetails - quantity = 10.

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
    (SELECT ProductId FROM
     OrderDetails
     WHERE Quantity = 10);
```

'ALL' – when all subqueries true !

'ANY' – when any one subquery true !

```
SELECT ALL ProductName
FROM Products          → All product names
WHERE TRUE;
```

ProductName if all the records in the 'order' table has quantity = 10

Return: FALSE

'Nothing returned'

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL(
    SELECT ProductID FROM OrderDetails
    WHERE Quantity = 10);
```

True → execute Select
FALSE → Don't

SELECT INTO → Copy columns into a new table

```
SELECT *
INTO newtable [IN externaldB]
FROM oldtable
WHERE Condt
```

```
SELECT Col1, Col2, ...
INTO newtable [In ExternaldB]
FROM oldtable
WHERE Condt
```

Backup

```
SELECT * INTO CustomersBackup          → SamedB
FROM Customers;

SELECT * INTO CustomersBackup IN 'Backup.mdb'   → otherdB
FROM Customers;
```

only Germany

```
SELECT *INTO Germanies
FROM Customers
WHERE Country = 'Germany';
```

```
SELECT Customers.CustomerName, Orders.OrderId
INTO CustomersBackup
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID =
                    Orders.CustomerID
```

create new, empty schema of another: (No data – returned)

```
SELECT * INTO newtable
FROM oldtable
WHERE FALSE;
```

Insert INTO SELECT Statement

Copies data from one table & inserts it into another.

'Existing records: unaffected'

```
INSERT INTO Customers (name, contact, Address, City, zip, Country)
SELECT SupplierName, ContactName, Address, City, zip, Country FROM
                                                        Suppliers
```

'copies suppliers to Customers'

only Germany

```
WHERE Country = 'Germany';
```

Format

```
INSERT INTO table2 (Col1, 2, ...)
SELECT Col1, Col2, ....
FROM table1
WHERE Condt
```

All Columns

```
INSERT INTO table2
SELECT * FROM table1
WHERE Condt
```

CASE – Returns a value – when first statement is true

```
CASE
    WHEN Cond1 THEN result1
    WHEN cond2 THEN result2
    WHEN Cond N THEN resultN
    ELSE result.
END;
```

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
```

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'Good'
    WHEN Quantity = 30 THEN 'Exactly'
    ELSE 'Less'
END AS QuantityText
FROM OrderDetails;
```

OrderID	Quantity	QuantityText
—	—	Good
—	—	Exactly
—	—	Less

IFNULL(), ISNULL, COALESCE(), NVL()
↓ MySql ↓ Oracle

Stored Procedure

* prepared SQL code – Save – reuse over & over again.
* Store as prepared (Stored procedure) – then just call to execute it.

Provide parameters, so stored procedure can act – based on parameter values.

```
CREATE PROCEDURE Procedure_name
AS
SQL_Statement
Go;
```

```
EXEC procedure_name;
```

eg:

```
CREATE PROCEDURE SELECTALL_Customers
AS
SELECT * FROM Customers
Go;
```

```
Exec SelectAll Customers
```

one Argument

```
CREATE PROCEDURE SelectAllCust @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
Go;
```

```
EXEC SelectAll Customers @city = 'London';
```

multiple parameters

```
CREATE PROCEDURE SELECTALL CUSTOMERS @City nvarchar(30),
                                     @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
Go;
```

```
Exec SelectAll Customers @City = 'London', @PostalCode = 'WA1 1DP';
```

```
-- Comments
/* */ multiple line Comments
```

operators:

+, -, *, /, %. → Arithmetic

&, |, ^ → Bitwise

=, >, <, <=, >=, <> → Comparison

+=, -=, *=, /=, %=, &=, ^-=, |*= → Bitwise exclusive equals; Bitwise OR equals

ALL, AND, ANY, BETWEEN, EXISTS, IN, LIKE, NOT, OR, SOME → Logical

Create dB

```
CREATE DATABASE testdB;
```

Delete – DROP dB

```
DROP DATABASE testdB;
```

Backup dB

```
BACKUP DATABASE databasename
TO DISK = 'D:\backups\testDB.bak';
WITH DIFFERENTIAL;
```

'Backup to DB' ← BACKUP DATABASE ... TO DISK ... → path

Use this when only need to update – what changed' ← WITH DIFFERENTIAL;

CREATE table

```
CREATE TABLE table_name (Col1 datatype, Col2 datatype,
                         Col3 datatype, ....);
```

eg: varchar, integer, date etc...

varchar(30); max 30 chars

Use Another table.

```
CREATE TABLE new_table_name AS
SELECT Col1, Col2....
FROM existing_table_name
WHERE .... ;
```

```
CREATE TABLE TestTABLE AS
SELECT CustomerName, Contact
FROM Customers;
```

Delete – DROP table

```
DROP TABLE Shippers;
```

Not table. Data alone ←

```
or)
TRUNCATE TABLE table_name;
```

Add Column

```
ALTER TABLE table_name
ADD column_name datatype;
```

Alter table

```
ALTER TABLE Customers
ADD Email Varchar(255);
```

Delete Column

```
ALTER TABLE Customers
DROP COLUMN Email;
```

Modify

Change data type:

ALTER TABLE – ALTER/MODIFY Column

MySQL:

```
ALTER TABLE table_name
MODIFY COLUMN column_name data_type;
```

Create Constraints

while creating – CREATE TABLE or ALTER TABLE

```
CREATE TABLE table_name (
    col1 datatype constraint,
    col2 "  "  ",
    col3 "  "  " ....
    ....
);
```

- NOT NULL → No NULL value
- UNIQUE → No repitation
- PRIMARY KEY → Nonull, no rep
- FOREIGN KEY → Prevent actions – destroy links b/w tables
- CHECK – Specific Condition
- DEFAULT – Set a default value (Say 0 – when value not mentioned)
- CREATE INDEX
  - ↳ Used to create & retrieve data quickly.

NOT NULL

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

UNIQUE Constraint

MySQL:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int, UNIQUE (ID)
);
```

↳ Unique ID

ID – Constraint – (Single Column)

multiple columns

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID, LastName)
);
```

↳ multiple columns

Alter table:

```
ALTER TABLE Persons
ADD UNIQUE (ID)
```

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE
    (ID, LastName);
```

DROP

"MySQL"

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

Oracle/MS access/SQL Server

```
"CONSTRAINT"
```

MySQL

```
CREATE TABLE Persons (
    ID int NOT NULL
    :
    PRIMARY KEY (ID)
);
```

Primary key

Allow naming – multiple columns

```
CREATE TABLE Persons (
    ID int NOT NULL,
    :
    CONSTRAINT PK_Person PRIMARY
        KEY (ID, LastName)
);
```

Alter TABLE

```
ALTER TABLE Person
ADD PRIMARY KEY (ID);
```

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person
    PRIMARY KEY (ID, LastName);
```

DROP – MySQL

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

* Foreign key: prevents – actions that would destroy links b/w tables
* FOREIGN KEY – Collection of fields/field – Refers to primary key of another table

MYSQL : CREATE TABLE Orders (
:
FOREIGN KEY (PersonID) REFERENCES Persons (PersonID)
);

Multiple:

CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
REFERENCES Persons (PersonID)
);

Alter table

ALTER TABLE Orders	ALTER TABLE Orders
ADD FOREIGN KEY (PersonID)	ADD CONSTRAINT FK_PersonOrder
RERERENCES Persons (PersonID)	FOREIGN KEY (PersonID) REFERENCES Persons (PersonID);

DROP - MySQL

ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;

Check

* limit – range that can be placed in a column (Certain values)

mySQL :

CHECK (Age >= 18)	CONSTRAINT CHK_Person CHECK (Age >= 18
);	AND City = 'Sandnes')

Alter table

ALTER TABLE Persons	ALTER TABLE Persons
ADD CHECK (Age >= 18);	ADD CONSTRAINT CHK_PersonAge CHECK (Age >= 18
	AND City = 'Sandnes');

DROP - mySQL

ALTER TABLE Persons
DROP CHECK CHK_PersonAge;

Default

City Varchar (255) DEFAULT 'Sandnes'
);
OrderDate date DEFAULT GETDATE()
);

Alter table – mySQL	DROP
ALTER TABLE Persons	ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';	ALTER City DROP DEFAULT;

Index – affects performance (use when absolutes)

* updating a table with indexes – slow
* Search – fast

UNIQUE Index Syntax – no duplicates

CREATE INDEX Index_name	CREATE UNIQUE INDEX Index_Name
ON table_name (Col1, Col2, ..);	ON table_name (Col1, Col2, ..);

eg:

CREATE INDEX Pdx_lastname	CREATE INDEX Pdx_Pname
ON Persons (LastName);	ON Persons (Lastname, Firstname);

Indexes – Not shown to users

DROP:

ALTER TABLE table_name;
DROP INDEX index_name;

Auto Increment – unique num – generated automatic

CREATE TABLE Persons (
PersonId int NOT NULL AUTO_INCREMENT,
:
);

ALTER TABLE Persons Auto_Increment = 100;

default: Increment is 1

INSERT INTO Persons
(FirstName, LastName)
VALUES ('Lars', 'Monsen');

Dates

MySQL – DATE – format	YYYY-MM-DD	
DATETIME	YYYY-MM-DD	HH:MI:SS
TIMESTAMP	YYYY-MM-DD	HH:MI:SS
YEAR	YYYY (or) YY	

* Two dates – easily compared – if no time involved.

```
SELECT * FROM Orders WHERE OrderDate = '2008-11-11';
```

If time also : no values returned

View – Virtual table based on Sub query

View: Virtual table – based on the result-set of an SQL statement
A view has rows & col – like table
Fields: From one/more real tables.
Use: Look like all fields are from a single table

```
CREATE VIEW view_name AS
SELECT col1, col2, ....
FROM table name
WHERE condition
```

Shows up to date data
Why: SQL (DB engine) – recreates it every time.

Create:

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

(' ' / [ ])

query

```
SELECT * FROM [Brazil Customers]
```

View having Products price > Avg Price

```
CREATE VIEW [Products above Avg] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

Update – Recreate (∵ each time newly created)

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

DROP

```
DROP VIEW view_name;
```

SQL Injection

* Code Injection technique: destroy our DB
* Common web hacking technique.
* placement of malicious code in SQL statements, via SQL i/p.

webpages: Ask users i/p – gives an SQL statement – unknowingly runs on DB.

eg:

```
txtUserID = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

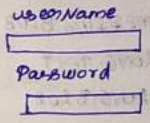
Say: I/p

105 OR 1=1 → TRUE [So o/p all the UserIds]

All userid & password – In hackers hip.

How to prevent?

Login website




```
uName = getRequestString("userName");
uPass = getRequestString("userPassword");
sql = 'SELECT * FROM Users WHERE Name = "' + uName + '" AND
      Pass = "' + uPass + '"';
```

Say: John Doe, myPass → No problem

Say: `" or "" = "`

`" or "" = "`

Now: `SELECT * FROM Users WHERE Name = "" or "" = "" AND Pass = "" or "" = "";`

= → Always TRUE

Based on batched SQL Statements

* group of 2/more SQL statements – seperated by semicolon.

Risk: delete all, Return all.

Protection – use parameters.

* SQL parameters values – added to an SQL query – at execution time (in Controlled manner)
* "@" marker, checks each parameter – correct?, treated as string – not a part of SQL.

SQL hosting

* Web server – access to a DB that uses SQL MySQL, SQL Server, MS Access

Datatypes

String:

- CHAR (size)
- VarCHAR (size)
- BINARY (size)
- VARBINARY (size)
- TINYBLOB
- TINY TEXT
- TEXT (size)
- BLOB (size)
- MEDIUM TEXT
- MEDIUM BLOB
- LONG TEXT
- LONG BLOB
- ENUM (val1, .. val3, ..)
- SET (val1, ...)

Numeric

- BIT (size)
- TINYINT (size)
- BOOL
- BOOLEAN
- SMALLINT (size)
- MEDIUMINT (size)
- INT (size)
- INTEGER (size)
- BIGINT (size)
- FLOAT (size, d)
- FLOAT (p)
- DOUBLE (size, d)
- DOUBLE PRECISION (size, d)
- DECIMAL (size, d)
- DEC (size, d).

Date & Time

- DATE ( )
- DATETIME (fsp)
- TIMESTAMP (fsp)
- TIME (fsp)
- YEAR ( )

Self Joins

```
SELECT  Cust_Id, cust_name, Cust_Contact
FROM    Customers
WHERE   Cust_name = (SELECT Cust_name FROM Customers
                     WHERE  Cust_Contact = 'Jim Jones');
```

↓ List of Customer name – with Contact – Jim Jones

Send mailing – to all Customers Contact – who r for same Company as Jim Jones.

① Approach: which Company – Jim works
② Send mail list to all Contacts customers of that Company!

USING INNER :– Select

```
SELECT  C1.cust_Id, C1. Cust_name, C1. Cust_Contact
FROM    Customers AS C1, Customers AS C2
WHERE
        C1. cust_name = C2. cust_name
        AND  C2. Cust_Contact = 'Jim Jones'
```

→ 2 things used (Self Join) For remove ambiguity.

Self Joins: Instead of Subquery – **Preferred**

Natural Join

* Atleast 2 columns – appears in both tables – Inner Join returns all data
* multiple occurence of same data
* Natural join – eliminates this! ⟶ We need to do it.

* Nothing special: use * for one table, then for other explicitly mention

```
SELECT  C.*, O. order_num, O.order_date,
        OI. prod_Id, OI. Quantity, OI. Item_Price
FROM    Customers AS C, Orders AS O, OrderItems AS OI
WHERE   C.Cust_Id = O. Cust_Id
AND     OI. order_num = O. order_num
AND     Prod_Id = 'RGAN01';
```

Truth: Every Inner Join thus created – actually natural join

We never need Inner Join – not a natural join

Outer Joins

* most joins : relate one table with other – occasionally – include rows – no related rows.

* eg: use Joins to accomplish

No associated rows in related table

- how many orders placed by each customer, including customers yet to place an order.
- list all products with order quantities, including products not ordered by anyone
- Avg Sale sizes – taking into account customers that have not yet ordered an order.

'outer Join'

```
SELECT  Customers.Cust_Id , Orders. order_num
FROM    Customers
LEFT OUTER JOIN Orders ON Customers. Cust_Id = Orders. Cust_Id;
```

* outer joins include rows with no related rows from left/right table.

Difference: (Right, Left) – order

Left can be turned to Right ⟶ Just reverse the order of the tables in the FROM (or) WHERE clause!

'FULL OUTER JOIN' – unrelated from both tables

FULL OUTER JOIN — not Supported by MySQL

Using Joins with Aggregate Function

Aggregate Func: Summarize data.

```
SELECT Customers.cust_id, COUNT(Orders.order_num) AS num_ord
FROM Customers
INNER JOIN Orders ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

Using JOINS and JOIN Conditions:

- ★ type of join: note inner/outer
- ★ check DBMS – syntax, check join condition
- ★ check for cartesian product.
- ★ include multiple table in a joint

"Test – each join seperately" – Troubleshoot

Combining queries [UNION / Compound] queries

2 scenarios:

1) Return similarly structured data from diff tables
2) Multiple queries against a single table – return data as one query

Combined queries & multiple WHERE Conditions:

Any SELECT with multiple WHERE — Combined query.

```
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI');

UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state = 'Fun4 ALL';
```

UNION – Execute both statements – Combine results as a single query.

Complex clauses: using UNION pays off

UNION limit: no std. SQL limit (may be DBMS)

★ Performance issues: most DBMS has internal query optimizer

★ SELECT Statements – Combined – before processed

No diff b/w multiple WHERE / UNION – Best stick with multiple WHERE "AS most query optimizers" – not great.

Rules:

- ★ Easy to use
- ★ Composed of 2/more SELECT statements
- ★ Same no. of columns, exp, aggregate function (Same order)
- ★ Compatible datatypes – must be compatible (Atleast implicit convertible)

UNION Column names

★ multiple columns combined with a UNION – different column names

★ which is actually returned – ?

Say:

```
SELECT Prod_name
UNION
SELECT Productname
```

Answer: first_name used.

Also: can use Alias on 1st name!

Side effect: First set of columns – will be used (naming columns)

Must use first column name

ORDER BY Prod_name — alone — else (error)!

Include / eliminate duplicate Rows

★ UNION → Removes duplicates by default. [behaves like multiple where in a single select]

★ I want duplicates

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4 All'
```

"UNION (vs) WHERE"

UNION ALL → can't be accomplished with WHERE

↳ "for this case alone use UNION along with ALL"

Sorting

```
ORDER BY cust_name, cust_contact;
```

Single ORDER BY clause → DBMS sorts all the combined queries!

Some DBMS (other types of UNION) → EXCEPT (MINUS) – Rows only in 1st, but no 2nd (both)
(Rarely used) – ∴ JOINS (That's why?)

Multiple tables: (without joins)
- ★ use union – do it slowly
- ★ (JOINS – preferred!)

Insert data

Complete row (all columns) – no need for Col names

```
INSERT INTO Customers  ——→ Not recommended.
VALUES (1000..6, 'Toy', '123 Any', 'Newyork', 'NY', '111', 'USA', NULL, NULL);
```

Dependant on order

Partial row

```
INSERT INTO Customers (cust_id, name, add, City, zip, Country, Contact_mail)
VALUES (10006, 'Toy' . . . . . NULL, NULL);
```

Just omit

No need for NULL (even)

Note: Colname – any order
Values – Same order as col-name

Can't Insert Same Record twice! → why? (PRIMARY KEY – No duplicates)
So delete, then do / update

Always use a Column list:

As a rule, NEVER use INSERT without explicitly specifying Column list

Note: Correct no. of values, matching types – error message (else)

Omit Columns:
1) defined as NULL
2) default value specified.

Key: field (NO NULL) – omitted – error (In this case must)
field – omitted – Assigned as NULL (must not be NO NULL)

Insert Retrieved data – INSERT SELECT

```
INSERT INTO Customers (cust_id, cust_Contact, cust_email, cust_name
                       cust_address, City, State, zip, Country)
SELECT                                  FROM  Cust New;
```

Note: INSERT fails (If Primary keys – duplicated)

★ INSERT → single Row → can be inserted
★ multiple Rows → multiple Inserts

Copying from one Table to Another: – 'Not Supported by DBs'

Without INSERT – Another form:

making Copies of tables

```
CREATE TABLE CustCopy AS
SELECT * FROM Customers;
```

Updating and Deleting Data

★ Don't omit WHERE
★ update specific rows, update all rows. → Special privileges in client & server DBMS.

UPDATE:
- ★ Table to be updated
- ★ column names & their new values
- ★ The filter condition that determines which rows should be updated

```
UPDATE Customers
SET cust_email = 'kim@thetoystore.com'
WHERE cust_id = 1000....5;   → else ALL Rows updated!
```

- ★ Using Subqueries – enable update using retrieved columns
- ★ FROM clause

DELETING DATA

```
UPDATE Customers
SET cust_email = NULL
WHERE cust_id = 10005;
```

- ★ Delete specific rows from a table
- ★ Delete all rows from a table.

```
DELETE FROM Customers
WHERE cust_id = 10006;
```

Foreign keys: Friends – don't delete.

★ DELETE deletes entire rows, even all rows from tables, – delete never deletes the table itself. – 'Table contents', not 'table itself'.

Faster deletion: Use truncation

Guidelines:
- ★ Update / delete – Always use where (unless every row)
- ★ Make sure: primary key.
- ★ Before using where with an update / delete → test with with SELECT.

* use enforced referential Integrity – so enforce (not deletion) –
* too easy to Incorrect WHERE clauses.
* Admin.

creating & manipulating tables

```
CREATE TABLE Product (
    Prod_Id    CHAR(10)      NOT FULL,
    Vend_Id    CHAR(10)      NOT FULL,        → values must
    Prod_name  CHAR(254)     NOT NULL,
    Prod_Price DECIMAL(8,2)  NOT NULL  DEFAULT 1,
    Prod_desc  VARCHAR(1000) NULL
);
```

Updating Tables: ALTER TABLE

Ideally → Tables should never be altered (once they contain data)

* All DBMS allow – Add Columns (add NULL, DEFAULT)
* Many DBMS – don't allow you to remove columns
* Most DBMS – allow renaming
* Restrict changes → Table must exist

```
ALTER TABLE Vendors
ADD Vend_Phone CHAR(20);
```

```
ALTER TABLE Vendors
DROP COLUMN Vend_Phone;
```

* Have backup! (Always)
* Database changes can't be undone!

Relational Rules to prevent accidental deletion:

RENAME → mySQL supports!

Views

* Virtual tables – dynamically retrieve data – Up to date

```
SELECT Cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
AND   OrderItems.order_num = Orders.order_num
AND   prod_id = 'RGAN01'
```

↳ For retrieve same data for another product – modify – where clause

* Lets merge in a virtual table – write queries for products!
* VIEW – Consistent

Use of views:

* Reuse SQL statements
* Simplify Complex SQL statements, Reuse, expose parts of the table instead of table itself.
* Secure data – given access to specific subsets of tables to USERS!
* change data format & rep.

* we can perform SELECT, Sort, Join etc... (Some restrictions also)
* performance issue: views (everytime) retrieve – degrade performance

Rules & restriction:

* Uniquely named – no limit to the no. of views.
* Security access (admin) – views can be nested – built using a query retrieves data from other query. (Nesting – degrade performance)
* Many DBMS prohibit use of ORDER BY in View
* SQLite (view) – Read only!
* Some – Don't allow update / insertion

update email – must reflect

Create VIEWS:

CREATE VIEW / CREATE TABLE → used to create only nonexisting Views / tables

Remove: use DROP

```
CREATE VIEW ProductCustomers AS
SELECT Cust_name, Cust_Contact, prod_id
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
AND   OI.Order_num = O.orders_num
```

Now – use reusable views (general-not tied to any data)

```
SELECT RTRIm (vend_name) + '(' + RTRIM (vend_country) + ')'
AS Vend_title
FROM Vendors ORDER BY Vend_name;
```

```
SELECT * FROM VendorLocations;  ⟶ Access VIEW
```

Views to filter data

```
CREATE VIEW CustomerEmailList AS
SELECT cust_id, cust_name, cust_email
FROM Customers
WHERE cust_email IS NOT NULL;
```

Remove Customers without mail id.

VIEW's WHERE and SELECT's WHERE ⟶ Combined.

With calculating field.

```
SELECT id, quantity, price, quantity * price AS exp_price
FROM OrderItems
WHERE order_num = 20008;
```

Now

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num, prod_id, quantity, item_price, quantity * itemPrice
       AS expanded_price
FROM OrderItems;

SELECT * FROM OrderItemsExpanded
WHERE OrderItems = 20008;
```

Stored procedures

* process an orders – items in stock
* Not in stock – Reserve – so not sold to anyone else – available (reduce restock in site)
* No stock – Don't accept order [Interaction with Vendor]
* Notify – when stock available – Shipped immediately

↓

* many tables, dynamic!

SqLite (don't support) * Stored procedure – Collection of SQL Statements (reusable)

Why?

* Encapsulation – Complexity reduced!
* Consistent (Procedure)
* Debug – Troubleshoot
* manage – modify – Secure – Less work, powerful, Flexible!

```
EXECUTE AddNewProduct ('JT502', 'Towel', 6.49, '...');
```

↓

* Add a new product (4 parameters)
* Automated – generate Ids.
* generate Id, Validate data, parameters, Inserts

  * Optional parameters – available
  * Range Specified.
  * o/p parameter
  * Data retrieved by SELECT

Create:

```
CREATE PROCEDURE MailingListCount (
    ListCount OUT INTEGER
)
IS
v_rows INTEGER;
BEGIN
   SELECT COUNT(*) INTO v_rows
   FROM Customers
   WHERE NOT cust_email IS NULL;
   ListCount := v_rows;
END
```

⟶ Not MySQL (Skip)

Managing transaction process

* Ensure batches of SQL operations – executed properly.

Add order:

* Already a customer ⟶ else Add
* Retrieve customer ID
* Add a row to orders table with cus_ID
* Retrieve new order ID
* Foreach order – 1 row in order table – productID

Transaction ⟶ Block of SQL statements

Roll back – undoing specified SQL statements

Commit – write unsaved SQL statements to DB

Savepoint – a temp placeholder in a transaction set to which you can issue a rollback (as opposed to rolling back an entire transaction). ⟶ we can rollback here.

Transaction → manage, insert, delete, update

Don't Rollback → DROP / CREATE MYSQL

START TRANSACTION

...

Rollback

DELETE FROM ORDERS;

ROLLBACK;

Commit – (implicit)

BEGIN TRANSACTION

DELETE OrderItems WHERE order_num = 12345

DELETE Orders WHERE order_num = 123

COMMIT TRANSACTION

Savepoint

SAVEPOINT delete1; → position to ROLLBACK

ROLLBACK TO delete1;

More Savepoints – the better

Cursors

* SQL retrieve operation work with set of rows – result sets
* SELECT → No way to get specific rows

Cursor – step forward / backward and one / more at a time

Cursor – DB query stored on server – not SELECT statement (Result retrieved from that statement)

* once cursor stored – app can scroll / browse up & down through data as needed.

Cursors:

* Read only
* directional operations – (forward, back, first, last, absolute pos, relative pos) – can be controlled

* flag – some Columns – editable
  others – not!

Scope

Instruct to make a copy (so data don't change using cursors.

* Cursors – used by interactive apps.

1) define Cursor – before use [defines SELECT statement to be used & options of cursor]
2) Once declared – open for use [retrieve data using prev defined SELECT]
3) With Cursor populated with data – individual rows – fetched as needed
4) Done – close cursor.

Create Cursor:

```
DECLARE CustCursor CURSOR
FOR
SELECT * FROM Customers
WHERE Cust_email IS NULL;
```

* OPEN CURSOR CustCursor → Fetch data
* CLOSE CustCursor → close.

Advanced SQL: * Referential Integrity!

Primary Key: * No two rows – same primary key, Every row has an unique key. (No NULL), Never modified, never reused, even – deleted – don't assign to any new rows.

Foreign Key: * Primary key in other table – Referential Integrity –

Multiple orders – Same Customer – multiple (repeat) Cust_Id

Only valid (cust_id) → ID column of Cust_Id from Customer table!

* Foreign Keys: prevents accidental deletion.
* Cascading delete – delete all related data – when a row – deleted from a table

Triggers

* Special Stored procedures – executed automatically when specific DB activity occurs. [INSERT, UPDATE, DELETE] / Combination.
* Triggers – tied to individual tables.

First normal form: (1NF)

* No Composite / multi-value attribute (1 row - 1 record)

2NF: 1st Normal form, No partial dependency.

No Nonprime attribute: dependant on any proper subset of any Candidate Key.

1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

→ Fees dependant on Course

1	C1
2	C2
1	C4
4	C3
4	C1

Table 1

C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

Table 2

Reduces redundancy

AB → C
BC → D

∴ By having AB we can get C, D

∴ AB - candidate key.

3NF:

* To transitive dependancy
* Atleast one of the following holds in every non-trivial func dep

X → Y

X Super key

Y prime attribute (each element of Y - part of Candidate key.

Candidate Key: Single / multiple Key - Uniquely identifies rows of a table

A → B, B → C [Two Function dependancies

A → C [Transitive dep]

{STUD No → STUD Name, STUD No → State, Stud - State → Country

Stud - Country → Stud - Age}

Candidate Key: Student No

A → BC | B → D

CD → E | E → A

Possible [A, E, CD, BC]

from these identify all!

Boyce Normal Form: For every FD, LHS is super key.

X → Y (X super key)

eg: R (A, B, C, D, E) with FD (BC → D, AC → BE, B → E)

highest Normal form

→ None of A, B, C, D, E → enough to determine all attribute

→ (AC) - candidate key.

1st Normal form ✓ (No composite values)

→ No partial dependencies (2NF) ✓

→ Super key X → Y

BC → D (neither BC is Super key, D - prime key)!

---

* No sql - Non Relational DB
* more flexible, Real time web applications

Adv: Highly scalable [add more resources] - mongodB, Cassandra

Disadv - Narrow focus, open source, management challenge, NoGUI, weak backup

Large document size.

Types → Doc based: mongodB, couchdB, Claudant

Types → Tabular: Hbase, BigTable, Accumulo

Types → Key value Store: Coherence, memCached, Redes.

Use: when huge data stored & retrieval

Relationship - not important

Dynamic (changing) non structured data

Support of constraint & Joins - no need at DB level.

Growing data, need for scaling.

MySql ranking func → dense_rank() - No gaps (Consecutive)

→ rank() - gaps: Tie - next rank one greater!

→ percent_rank(): 0 to 1, divide by highest!

→ non Consecutive ranks

next rank = prev rank + count of duplicates

Correlated Subquery evaluated once - for each row processed by the Parent Statement like UPDATE, DELETE, SELECT.

* one way - Read every row in a table - Compare with every other row

* normal nested subquery -

INNER SELECT - runs once

Return to prev subquery.

'Driven by outer subquery'

Correlated: Can USE ANY and ALL operator in a Correlated query

Correlated Subquery: Answers a multipart question where ANSWER depends on value in each row processed by the parent