Database Systems

Nguyễn Văn Diêu

HO CHI MINH CITY UNIVERSITY OF TRANSPORT

2025

Kiến thức - Kỹ năng - Sáng tạo - Hội nhập

Sứ mệnh - Tầm nhìn

Triết lý Giáo dục - Giá trị cốt lõi

Outline I

Outline II

Outline III

Outline IV

Outline V

Outline VI

Outline VII

Learning Objective

- Define Database and Database Management System (DBMS).
- Study structure of Relational model in Database.
- Query Data by Relational Algebra language and SQL.
- Analyses Data Integrity Constraint.

Student Requirements

To succeed in this course, students are expected to:

1. **Attendance**: Maintain regular attendance in both lectures and lab sessions
2. **Participation**: Actively engage in class discussions and group activities
3. **Assignments**: Complete all homework assignments and projects by their deadlines
4. **Lab Work**: Practice database concepts through hands-on exercises
5. **Reading**: Review assigned materials before each class session

Databases Are Everywhere

- Database: Collection of related data.
- Changes in the organization = changes in the database.
- Examples:
    - Personnel records
    - Students management
    - Banking
    - Airline reservations

Operations with Databases

- Design
  Define structure and types of data.
- Construction
  Create data structures of DB, populate DB with data.
- Manipulation of Data
  - Insert, delete, update
  - Query: “Which department pays the highest salary?”
  - Create reports:
    “List monthly salaries of employees, organized by department, with average salary and total sum of salaries for each dept”

Database Management System (DBMS)

DBMS is a software package designed to store and manage databases.

Data Model

Data model is a tool for describing data. The description generally consists of three parts:

- **Structure of the data**. Familiar with programming languages such as C: arrays, structures, objects.

  It helps the DBMS work with data.
- **Operations on the data**.
  - Operations that retrieve information: Set of queries.
  - Operations that change the database: Set of modifications.
- **Constraints on the data**. The way to describe limitations on what the data can be.

Important Data Models

Today, there are three data models of importance for database systems:

- **Relational data model**. Including object-relational extensions
- **Semi-structured data model**. Including XML and related standards.
- **NoSQL data model**. Four types of NoSQL databases are Document-oriented, Key-Value Pairs, Column oriented and Graph.

Relational Data Model

Relational model gives a single way to represent data: as a two-dimensional table called a relation.

e.g.

ST_ID	ST_NAME	CLASS_ID
S01	Nguyen Van Nam	L01
S02	Nguyen Van Nam	L01
S03	Tran Quoc Tuan	L01
S04	Le Van Cuong	L02
S05	Nguyen Van Cuong	L02

Attributes

- The columns of a relation are named by attributes.
- An attribute describes the meaning of entries in the column below.
- Each attribute belong to one **data type**, so that have **Domain**.
- **Domain**: Set of possible **atomic** values, including **Null** value.
- in **Domain** not permitted record structure, set, list, array, or any other type that can broken into smaller components.

Schemes

- Relation scheme is the set of attributes. It is a tructure of data.
- e.g. **STUDENT(ST_ID, ST_NAME, CLASS_ID)**
- The attributes in a relation scheme are a set, not a list.
- A database consists of one or more relation schemes.

  That is called a relational database scheme or just database scheme.
- In general: $R(A_1, A_2, ..., A_n)$

Tuples

- The rows of a relation are called tuples.

  e.g. *(S01, Nguyen Van Nam, L01)*

- **t** is a tuple. **t.Attribute** is a value of attribute in tuple **t**.

  e.g. $t =$ *(S01, Nguyen Van Nam, L01)*

  $t.(ST\_ID) =$ *'S01'*

Relation Instances

- Tuples in relation change over time.
- A set of tuples for a given relation in time is an instance of that relation.
- Denote: **r(R)**
- e.g. **STUDENT(ST_ID, ST_NAME, CLASS_ID)**

  in time we see we have one *relation instance*

ST_ID	**ST_NAME**	**CLASS_ID**
S01	Nguyen Van Nam	L01
S02	Nguyen Van Nam	L01
S03	Tran Quoc Tuan	L01
S04	Le Van Cuong	L02
S05	Nguyen Van Cuong	L02

Keys

- K is a key of relation if K is a subset of R such that for any distinct tuples t_1 and t_2 in $r(R)$ then $t_1(K) \neq t_2(K)$

  and **No Proper** subset K' of K shares this property.
- X is a **superkey** of R if X contains a key of R.
- R maybe more than one key.
- Each key denote one underline.
- e.g. **STUDENT** have one key **ST_ID**.

Relational Algebra Language

The operations of the traditional relational algebra fall into four broad classes:

1. Set operations: union, intersection, and difference. Applied to relations.
2. Remove parts of a relation: Selection and Projection.
3. Combine the tuples of two relations: Cartesian product, Join.
4. Renaming: Changes the relation scheme.
5. Group.
6. ...

Conditions

Let $\mathcal{R}$, $\mathcal{S}$ with conditions:

1. Identical sets of attributes, domains for each attribute must be the same.
2. Attributes must be ordered to the same for both relations.

Set Operators

Relation $\mathcal{R}$ and $\mathcal{S}$: set of tuples.

1. $\mathcal{R} \cup \mathcal{S}$: **Union** of $\mathcal{R}$ and $\mathcal{S}$.

   $\mathcal{R} \cup \mathcal{S} = \{t \mid t \in r(\mathcal{R}) \vee t \in s(\mathcal{S})\}$

2. $\mathcal{R} \cap \mathcal{S}$: **Intersection** of $\mathcal{R}$ and $\mathcal{S}$.

   $\mathcal{R} \cap \mathcal{S} = \{t \mid t \in r(\mathcal{R}) \wedge t \in s(\mathcal{S})\}$

3. $\mathcal{R} - \mathcal{S}$: **Difference** of $\mathcal{R}$ and $\mathcal{S}$.

   $\mathcal{R} - \mathcal{S} = \{t \mid t \in r(\mathcal{R}) \wedge t \notin s(\mathcal{S})\}$

Selection σ

Let relation $\mathcal{R}$ and C is a conditional logic expression

- C made from logic operator: $=, \neq, <, \leqslant, >, \geqslant, \wedge, \vee, \mathit{not}$
- Selection operator applied to $\mathcal{R}$, produces a new relation with a subset of tuples of $r(\mathcal{R})$
- The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of $\mathcal{R}$
- Notation: $\sigma_C(\mathcal{R})$
- $\sigma_C(\mathcal{R}) = \{t \mid t \in r(\mathcal{R})\ ,\ t\ \mathit{satisfy}\ C\}$

e.g. σ

Suppose $\mathcal{R}(ABC)$ with instance $r(\mathcal{R})$:

A	B	C
1	3	1
2	3	2
3	4	3

$\sigma_{B=3 \wedge C>1}(\mathcal{R})$

A	B	C
2	3	2

Projection π

Let relation $\mathcal{R}$ and $X \subseteq \mathcal{R}$. Projection of $\mathcal{R}$ on to X:

- Notation: $\pi_X(()\mathcal{R})$
- $\pi_X(()\mathcal{R})$ only have X attributes
- $\pi_X(()\mathcal{R}) = \{t.X \mid t \in r(\mathcal{R})\}$
- Remove duplicate tuples in $\pi_X(()\mathcal{R})$

If $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_m$, then

$$\pi_{X_1}(()\pi_{X_2}(()\cdots(\pi_{X_m}(()\mathcal{R}))\cdots)) = \pi_{X_1}(()\mathcal{R})$$

e.g. π

Suppose $\mathcal{R}(ABC)$ with instance $r(\mathcal{R})$:

A	**B**	**C**
1	3	1
2	3	2
3	4	3

$\pi_B(()\mathcal{R})$

B
3
4

Cartesian product $\times$

Seeing relation $\mathcal{R}$ and $\mathcal{S}$ to be a set of tuples.

- *Cartesian product* (cross-product, just product) of two sets $\mathcal{R}$ and $\mathcal{S}$ is the set of pairs of (element of $\mathcal{R}$) and (element of $\mathcal{S}$).
- Denote: $\mathcal{R} \times \mathcal{S}$
- Relation $\mathcal{R} \times \mathcal{S}$ have all attributes of $\mathcal{R}$ and $\mathcal{S}$
- $\mathcal{R} \times \mathcal{S} = \{(t_1, t_2) \mid t_1 \in r(\mathcal{R}), t_2 \in s(\mathcal{S})\}$

e.g. $\times$

Let $\mathcal{R}(AB)$, $\mathcal{S}(CD)$; $\mathcal{R} \times \mathcal{S}$

$\mathcal{R}$

A	B
1	2
3	4

$\mathcal{S}$

C	D
5	6
7	8

$\mathcal{R} \times \mathcal{S}$

A	B	C	D
1	2	5	6
1	2	7	8
3	4	5	6
3	4	7	8

Natural Join $\bowtie$

Let relation $\mathcal{R}$ and $\mathcal{S}$ with $X = \mathcal{R} \cap \mathcal{S}$
Natural Join of $\mathcal{R}$ and $\mathcal{S}$ which we pair only those tuples from $r(\mathcal{R})$ and $s(\mathcal{S})$ that agree in X attribute.

- X is called attribute common.
- Notation: $\mathcal{R} \bowtie \mathcal{S}$
- $\mathcal{R} \bowtie \mathcal{S} = \{(t_1, t_2) \mid t_1 \in r(\mathcal{R}), t_2 \in s(\mathcal{S}), t_1.X = t_2.X\}$

$L = \mathcal{R} \cup (\mathcal{S} - \mathcal{R})$

$$\mathcal{R} \bowtie \mathcal{S} = \pi_L(()\sigma_{\mathcal{R}.X = \mathcal{S}.X}(\mathcal{R} \times \mathcal{S}))$$

If $X = \emptyset$ then *Natural Join* become *Cartesian Product*

e.g. $\bowtie$

Let $\mathcal{R}(ABC)$, $\mathcal{S}(CD)$; $\mathcal{R} \bowtie \mathcal{S}$

$\mathcal{R}$

A	B	C
1	1	2
5	5	3
7	6	4
1	5	7

$\mathcal{S}$

C	D
2	6
5	8
2	4
4	8

$\mathcal{R} \bowtie \mathcal{S}$

A	B	C	D
1	1	2	6
1	1	2	4
7	6	4	8

Theta-join $\bowtie_\theta$

Theta-join of relations $\mathcal{R}$ and $\mathcal{S}$ based on *condition* θ:

The result of this operation is constructed as follows:

1. Take the product of $\mathcal{R}$ and $\mathcal{S}$: $\mathcal{R} \times \mathcal{S}$
2. Select from $\mathcal{R} \times \mathcal{S}$ so that satisfy the condition θ

$$\mathcal{R} \bowtie_\theta \mathcal{S} = \sigma_\theta(\mathcal{R} \times \mathcal{S})$$

e.g. $\bowtie_\theta$

Let $\mathcal{R}(ABC)$, $\mathcal{S}(DE)$; $\mathcal{R} \bowtie_{C>D} \mathcal{S}$

$\mathcal{R}$

A	B	C
1	1	2
5	5	3

$\mathcal{S}$

D	E
2	6
5	8

$\mathcal{R} \bowtie_{C>D} \mathcal{S}$

A	B	C	D	E
5	5	3	2	6

Left Semi-join $\ltimes$

Let $\mathcal{R}$ and $\mathcal{S}$. The Left Semi-join is similar to the natural join.

The result is the set of all tuples in $r(\mathcal{R})$ for which there is a tuple in $s(\mathcal{S})$ that is equal on their common attribute.

The difference from a natural join is that other attributes of $\mathcal{S}$ do not appear.

- Notation: $\mathcal{R} \ltimes \mathcal{S}$
- $\mathcal{R} \ltimes \mathcal{S} = \{t \mid t \in r(\mathcal{R}) \wedge \exists s \in (\mathcal{R} \bowtie \mathcal{S})\}$, or
- $\mathcal{R} \ltimes \mathcal{S} = \pi_{\mathcal{R}}(()\mathcal{R} \bowtie \mathcal{S})$

e.g. $\ltimes$

Let $\mathcal{R}(ABC)$, $\mathcal{S}(CD)$; $\mathcal{R} \ltimes \mathcal{S}$

$\mathcal{R}$

A	B	C
1	1	2
5	5	3

$\mathcal{S}$

C	D
2	6
5	8

$\mathcal{R} \ltimes \mathcal{S}$

A	B	C
1	1	2

Right Semi-join $\rtimes$

Let $\mathcal{R}$ and $\mathcal{S}$. The Right Semi-join is similar to the natural join.

The result is the set of all tuples in $s(\mathcal{S})$ for which there is a tuple in $r(\mathcal{R})$ that is equal on their common attribute.

The difference from a natural join is that other attributes of $\mathcal{R}$ do not appear.

- Notation: $\mathcal{R} \rtimes \mathcal{S}$
- $\mathcal{R} \rtimes \mathcal{S} = \{t \mid t \in s(\mathcal{S}) \land \exists r \in (\mathcal{R} \bowtie \mathcal{S})\}$, or
- $\mathcal{R} \rtimes \mathcal{S} = \pi_{\mathcal{S}}(()\mathcal{R} \bowtie \mathcal{S})$

e.g. $\rtimes$

Let $\mathcal{R}(ABC)$, $\mathcal{S}(CD)$; $\mathcal{R} \rtimes \mathcal{S}$

$\mathcal{R}$

A	B	C
1	1	2
5	5	3

$\mathcal{S}$

C	D
2	6
5	8

$\mathcal{R} \rtimes \mathcal{S}$

C	D
2	6

Anti-join $\triangleright$

Let $\mathcal{R}$ and $\mathcal{S}$. The Anti-join is similar to the semijoin, but the result of an antijoin is only those tuples in $\mathcal{R}$ for which there is no tuple in $\mathcal{S}$ that is equal on their common attribute names.

- Notation: $\mathcal{R} \triangleright \mathcal{S}$
- $\mathcal{R} \triangleright \mathcal{S} = \{t \mid t \in r(\mathcal{R}) \land \neg\exists s \in (\mathcal{R} \bowtie \mathcal{S})\}$, or
- $\mathcal{R} \triangleright \mathcal{S} = \{t \mid t \in r(\mathcal{R})$, there is no tuple $s(\mathcal{S})$ that satisfies $(\mathcal{R} \bowtie \mathcal{S})\}$, or
- $\mathcal{R} \triangleright \mathcal{S} = \mathcal{R} - \mathcal{R} \ltimes \mathcal{S}$

e.g. $\triangleright$

Let $\mathcal{R}(ABC)$, $\mathcal{S}(CD)$; $\mathcal{R} \triangleright \mathcal{S}$

$\mathcal{R}$

A	B	C
1	1	2
5	5	3

$\mathcal{S}$

C	D
2	6
5	8

$\mathcal{R} \triangleright \mathcal{S}$

A	B	C
5	5	3

Complex Queries

- *Complex queries* refer to data in many relations.
- In relational algebra, like all algebras, allows us to *form expressions* by applying operations to the result of other operations.
- Possible to represent expressions as *expression trees*, so that machine program can do it easely.

e.g.

Let $\mathcal{R}(ABCDE)$

A	B	C	D	E
1	2	5	6	2
1	2	7	8	3
3	4	5	6	4

Query: Find **A** when **B** $>$ **3** and **A** when **E** $<$ **4**
Answer: $\pi_A(\sigma_{B>3}(\mathcal{R}) \cup \sigma_{E<4}(\mathcal{R}))$

A
1
3

Another answer: $\pi_A(\sigma_{B>3 \,\vee\, E<4}(\mathcal{R}))$

Rename ρ

- To control the names of the attributes or relation scheme, using *rename* operator. All data (tuples) not change in this operator.

- Notation: $\rho_{\mathcal{S}(A_1,A_2,\ldots,A_n)}(\mathcal{R})$

rename $\mathcal{R}$ to $\mathcal{S}$. n attributes of $\mathcal{R}$ rename to $A_1, A_2, \ldots, A_n$

- eg. Let $\mathcal{R}(ABC)$.

$\rho_{\mathcal{S}}(\mathcal{R})$ give us the relation $\mathcal{S}(ABC)$

$\rho_{ADE}(S)$ give us the relation $\mathcal{S}(ADE)$

Assignment $:=$

- Definition: Assign an relation expression to relation.

- Notation: $:=$

- eg. Suppose

$$\text{StudentMark(StudentID, SubjectID, Mark)}$$

$$\mathcal{R}(T, U, M) := \sigma_{SubjectID='S01' \ \wedge \ Mark>7}(StudentMark)$$
$$\mathcal{S}(T, U, M) := \sigma_{SubjectID='S02' \ \wedge \ Mark>9}(StudentMark)$$
$$Answer(StudentID) := \pi_{T}(\mathcal{R} \cap \mathcal{S})$$

Div $\div$

Divide operator has a rather complex definition, but it does have some applications in natural situations.

Let $\mathcal{R}$ and $\mathcal{S}$, with $\mathcal{S} \subseteq \mathcal{R}$

- Let $\mathcal{R}' = \mathcal{R} - \mathcal{S}$
- $\mathcal{R}$ *divided by* $\mathcal{S}$, written $\mathcal{R} \div \mathcal{S}$
- $r'(\mathcal{R}') = \{t \mid \textit{for every tuple } t_s \in s(\mathcal{S})$
  $\textit{there is a } t_r \in r(\mathcal{R}) \textit{ with } t_r(\mathcal{R}') = t \textit{ and } t_r(\mathcal{S}) = t_s\}$

e.g. $\div$

Suppose $\mathcal{R}(AB)$ and $\mathcal{S}(B)$, with $\mathcal{S} \subseteq \mathcal{R}$

$\mathcal{R}' = \mathcal{R} - \mathcal{S}$

$\mathcal{R}'(A) = \mathcal{R}(AB) \div \mathcal{S}(B)$

$\mathcal{R}'$

A
5
7

$\mathcal{R}$

A	B
7	1
7	2
5	1
5	2
8	1

$\mathcal{S}$

B
1
2

Outer Joins

We know:

- *Null* value is the value unknown or does not exist.
- All comparisons involving null are false by definition.

So that:

- Extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.

Left Outer Joins $⟕$

Let $\mathcal{R}$ and $\mathcal{S}$. *Left outer join* of that two relations is

- Denote: $\mathcal{R} ⟕ \mathcal{S}$
- The result is the set of:
    - Tuples in $\mathcal{R}$ and $\mathcal{S}$ that are equal on their common attribute names, and
    - Addition to tuples in $\mathcal{R}$ that have no matching tuples in $\mathcal{S}$.
- Let $(\perp, ..., \perp)$: Singleton relation with attributes that are unique to the $\mathcal{S}$ (are not attributes of $\mathcal{R}$):

  $(\mathcal{R} \bowtie \mathcal{S}) \cup ((\mathcal{R} - \pi_{\mathcal{R}}(\mathcal{R} \bowtie \mathcal{S})) \times \{(\perp, ..., \perp)\})$

e.g. $⟕$

Suppose $\mathcal{R}(AB)$ and $\mathcal{S}(BC)$.

A	B
1	2
2	4
3	6

B	C
2	7
3	8
5	9

$\mathcal{R} ⟕ \mathcal{S}$

A	B	C
1	2	7
2	4	$\perp$
3	6	$\perp$

Right Outer Joins $\rtimes\!\!\!\sqsupset$

Let $\mathcal{R}$ and $\mathcal{S}$. *Right outer join* of that two relations is

- Denote: $\mathcal{R} \bowtie\!\!\!\sqsupset \mathcal{S}$
- The result is the set of:
  - Tuples in $\mathcal{R}$ and $\mathcal{S}$ that are equal on their common attribute names, and
  - Addition to tuples in $\mathcal{S}$ that have no matching tuples in $\mathcal{R}$.
- Let $(\perp, ..., \perp)$: Singleton relation with attributes that are unique to the $\mathcal{R}$ (are not attributes of $\mathcal{S}$):

  $(\mathcal{R} \bowtie \mathcal{S}) \cup (\{(\perp, ..., \perp)\} \times (\mathcal{S} - \pi_{\mathcal{S}}(\mathcal{R} \bowtie \mathcal{S})))$

e.g. $⟖$

Suppose $\mathcal{R}(AB)$ and $\mathcal{S}(BC)$.

A	**B**
1	2
2	4
3	6

B	**C**
2	7
3	8
5	9

$\mathcal{R} ⟖ \mathcal{S}$

A	**B**	**C**
1	2	7
$\perp$	3	8
$\perp$	5	9

Full Outer Joins $⟗$

Let $\mathcal{R}$ and $\mathcal{S}$. *Full outer join* of that two relations is

- Denote: $\mathcal{R} ⟗ \mathcal{S}$
- The result is the set of:
  - Tuples in $\mathcal{R}$ and $\mathcal{S}$ that are equal on their common attribute names, and
  - Addition to tuples in $\mathcal{R}$ that have no matching tuples in $\mathcal{S}$ and to tuples in $\mathcal{S}$ that have no matching tuples in $\mathcal{R}$.
- $\mathcal{R} ⟗ \mathcal{S} = (\mathcal{R} ⟕ \mathcal{S}) \cup (\mathcal{R} ⟖ \mathcal{S})$

e.g. $⟗$

Suppose $\mathcal{R}(AB)$ and $\mathcal{S}(BC)$.

A	**B**
1	2
2	4
3	6

B	**C**
2	7
3	8
5	9

$\mathcal{R} ⟗ \mathcal{S}$

A	**B**	**C**
1	2	7
2	4	$\perp$
3	6	$\perp$
$\perp$	3	8
$\perp$	5	9

Duplicate Elimination Operators δ

Let $\mathcal{R}$

- Denode: $\delta(\mathcal{R})$
- Return the set consisting of *one copy of every tuple that appears one or more times* in relation $\mathcal{R}$.

e.g. $\mathcal{R}(ABC)$; $\delta(\mathcal{R})$

$\mathcal{R}$

A	B	C
1	2	3
4	5	6
1	2	3

$\delta(\mathcal{R})$

A	B	C
1	2	3
4	5	6

Aggregate functions

Aggregate function takes a collection of values and returns a single value as a result.

- $sum(\cdot)$: Sum of values.
- $avg(\cdot)$: Average value.
- $min(\cdot)$: Minimum value.
- $max(\cdot)$: Maximum value.
- $count(\cdot)$: Number of values.

Group $\mathcal{G}$

Let $\mathcal{U}$ is any relational-algebra expression
$g_1, g_2, \ldots, g_n$: Attributes on which to group (can be empty)
Each f_i is an aggregate function
Each A_i is an attribute name

- Denote: ${}_{g_1,g_2,\ldots,g_n}\mathcal{G}_{f_1(A_1),f_2(A_2),\ldots,f_m(A_m)}(\mathcal{U})$
- Partition the tuples of $\mathcal{U}$ into groups $g_1, g_2, \ldots, g_n$.
- For each group:
  - The grouping attributes' values for that group and
  - Can use aggregate functions f_i for value in group.

e.g. $\mathcal{G}$

Suppose $\mathcal{R}(ABC)$

A	B	C
1	4	3
1	3	3
4	5	6

$\mathcal{G}_{sum(A),\ min(C),\ count(C)}(\mathcal{R})$

SumA	MinB	CountC
6	3	3

e.g. $\mathcal{G}$

Suppose $\mathcal{R}(ABC)$

A	B	C
1	4	3
1	3	3
4	5	6
4	6	7
4	8	9

$_{A}\mathcal{G}_{min(B),\ max(B),\ Count(A)}(\mathcal{R})$

A	MinB	MaxB	CountA
1	3	4	2
4	5	8	3

Sorting Operator τ

Let $\mathcal{R}$, with $L \subseteq \mathcal{R}$
Sorting data in $r(\mathcal{R})$ over L attributes.
Denote: $\tau_L(\mathcal{R})$

Suppose $\mathcal{R}(ABC)$, $\tau_{A,B}(\mathcal{R})$

$\mathcal{R}$

A	B	C
4	2	3
4	5	6
1	3	3
1	2	5

$\tau_{A,B}(\mathcal{R})$

A	B	C
1	2	5
1	3	3
4	2	3
4	5	6

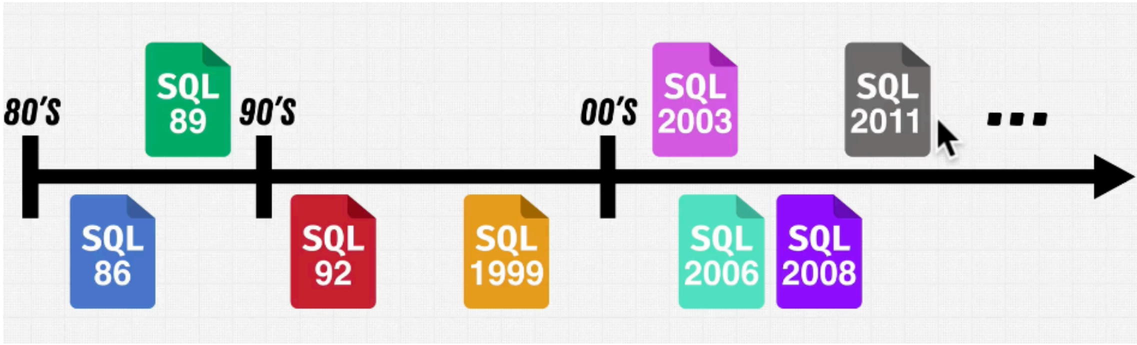
SQL

∗ SQL (pronounced “sequel”) - ”Structured Query Language” is the principal language used to describe and manipulate relational databases.

∗ There are two aspects to SQL:

1. The Data-Definition sublanguage for declaring database schemas (DDL).
2. The Data-Manipulation sublanguage for querying databases and for modifying the database (DML).

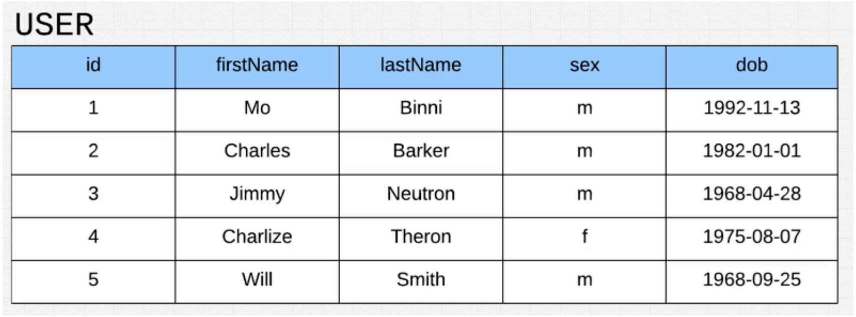
SQL Standards



Relations in SQL

* SQL makes a distinction between three kinds of relations:
  1. Table: Relation that exists in the database and that can be modified by changing its tuples, as well as queried.
  2. View: Relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed.
  3. Temporary table: Constructed by the SQL language processor when it performs its job of executing queries and data modifications. These relations are then thrown away and not stored.

What is the Table



USER

id	firstName	lastName	sex	dob
1	Mo	Binni	m	1992-11-13
2	Charles	Barker	m	1982-01-01
3	Jimmy	Neutron	m	1968-04-28
4	Charlize	Theron	f	1975-08-07
5	Will	Smith	m	1968-09-25

Create Table

Create table in current database:

```
CREATE TABLE tableName(
   column1 datatype,
   .....
   columnN datatype,
   PRIMARY KEY(one or more columns)
);
```

e.g.

```
CREATE TABLE Company(
   ID Int Primary Key,
   Name Text   Not Null,
   Age Int   Not Null,
   Address Char(50),
   Salary Real
);
```

Drop, Alter Table

Drop table in current database:

```
DROP TABLE TableName;
```

∗ Alter table statement is used to add, delete, or modify columns in an existing table.
∗ Alter table statement is also used to add and drop various constraints on an existing table.

Alter Table - ADD Column:

```
ALTER TABLE tableName
ADD columnName datatype;
```

Alter Table - DROP Column:

```
ALTER TABLE tableName
DROP COLUMN columnName;
```

e.g.

- CREATE TABLE contacts
  (id INTEGER PRIMARY KEY,
  name TEXT NOT NULL COLLATE NOCASE,
  phone TEXT NOT NULL DEFAULT 'Unknown',
  UNIQUE (name,phone)
  );

- DROP TABLE Shippers;
- ALTER TABLE Customers ADD Email varchar(255);
- ALTER TABLE Customers DROP COLUMN Email;
- ALTER TABLE Persons ALTER COLUMN DateOfBirth year;

Insert

- **INSERT a single row into a table:**

```
INSERT INTO tableName (column1, column2 ,..)
VALUES
   (value1, value2 ,...);
```

- **INSERT multiple rows into a table:**

```
INSERT INTO tableName (column1, column2 ,..)
VALUES
   (value1, value2 ,...),
   (value1, value2 ,...),
   ...
   (value1, value2 ,...);
```

e.g.

- INSERT INTO Students (name)
  VALUES
  ('Bud Powell');
- INSERT INTO Students (name)
  VALUES
  ("Buddy Rich"),
  ("Candido"),
  ("Charlie Byrd");

Update

The search condition in the WHERE has the following form:

Left Expression **Comparicon Operator** *Right Expression*

```
UPDATE tableName
SET column1 = newValue1,
    column2 = newValue2
WHERE
    SearchCondition
```

e.g.

```
WHERE column1 = 100;
WHERE column2 IN (1,2,3);
WHERE column3 LIKE 'An%';
WHERE column4 BETWEEN 10 AND 20;
```

Comparison Operators

Operator	Meaning
$=$	Equal to
$<>$ or $!=$	Not equal to
$<$	Less than
$>$	Greater than
$<=$	Less than or equal to
$>=$	Greater than or equal to

Logical Operators

Notice that 1 means TRUE, and 0 means FALSE.

Operator	Meaning
All	Returns 1 if all expressions are 1.
AND	Returns 1 if both expressions are 1, and 0 if one of the expressions is 0.
ANY	Returns 1 if any one of a set of comparisons is 1.
BETWEEN	Returns 1 if a value is within a range.
EXISTS	Returns 1 if a subquery contains any rows.
IN	Returns 1 if a value is in a list of values.
LIKE	Returns 1 if a value matches a pattern.
NOT	Reverses the value of other operators such as NOT EXISTS, NOT IN, NOT BETWEEN, etc.
OR	returns true if either expression is 1.

e.g.

```
UPDATE employees
SET lastname = 'Smith'
WHERE
   employeeid = 3;
```

```
UPDATE employees
SET city = 'Toronto',
    state = 'ON',
    postalcode = 'M5P 2N7'
WHERE
   employeeid = 4;
```

Delete

```
DELETE FROM tableName
WHERE searchCondition

e.g.
DELETE FROM employees
WHERE
   name LIKE '%Santana%';
```

Database Scheme

Suppose *Order Management* Database scheme:

1. **Categories(CategoryID, CategoryName, Description)**
2. **Products(ProductID, ProductName, UnitPrice, CategoryID)**
3. **Customers(CustomerID, CustomerName, Address, Phone, Email, Country)**
4. **Orders(OrderID, OrderDate, RequiredDate, CustomerID)**
5. **OrderDetails(OrderID, ProductID, UnitPrice, Quantity, Discount)**

Products and Joins in SQL

e.g. Suppose we want to know the name of the customer in which he order with OrderID = 'D01'. To answer this question we need the following two relations:

**Customers(<u>CustomerID</u>, CustomerName, Address,
Phone, Fax, Country)**
Orders(<u>OrderID</u>, OrderDate, RequiredDate, CustomerID)

```
SELECT B.CustomerID, CustomerName
FROM Orders AS A, Customers AS B
WHERE A.CustomerID = B.CustomerID
        AND OrderID = 'D01';
```

Same as the relational algebra expression:

$\pi_{CustomerID,\ CustomerName}\left(\left(\sigma_{OrderID\ =\ 'D01'}\left(Orders \bowtie Customers\right)\right)\right)$

Union $\cup$

e.g. Suppose we wanted the ID and name of all product it is belong to Category = 'C01' or 'C02'.

```
( SELECT ProductID, ProductName
  FROM Products
  WHERE CategoryID = 'C01' )
    UNION
( SELECT ProductID, ProductName
  FROM Products
  WHERE CategoryID = 'C02' );
```

Same as the relational algebra expression:

$$\pi_{ProductID,\ ProductName}(\sigma_{CategoryID\ =\ 'C01'}(Products)) \cup$$
$$\pi_{ProductID,\ ProductName}(\sigma_{CategoryID\ =\ 'C02'}(Products))$$

Intersection ∩

e.g. Suppose we wanted the ID and name of product so that order with customer ID = 'T01' and 'T02'.

```
( SELECT C.ProductID, ProductName
  FROM Orders A, OrderDetails B, Products C
  WHERE CustomerID = 'T01'
          AND A.OderID = B.OrderID
          AND B.ProductID = C.ProductID )
    INTERSECT
( SELECT C.ProductID, ProductName
  FROM Orders A, OrderDetails B, Products C
  WHERE CustomerID = 'T02'
          AND A.OderID = B.OrderID
          AND B.ProductID = C.ProductID );
```

Difference –

e.g.
Suppose we wanted the ID and name of product that never order.

```
( SELECT ProductID, ProductName
  FROM Products

    EXCEPT

( SELECT B.ProductID, ProductName
  FROM OrderDetails A, Products B
  WHERE A.ProductID = B.ProductID );
```

Subqueries

* Subquery: A query that is part of another query.
* Subqueries can have subqueries, and so on, down as many levels as we wish.
* There are a number of other ways that subqueries can be used:

1. Subqueries can return a *single constant*, and this constant can be *compared* with another value in a WHERE clause.
2. Subqueries can return *relations* that can be used in various ways in WHERE clauses.
3. Subqueries can appear in FROM clauses.

e.g.

Finding information about customer that order in OderID = 'D01'.

```
SELECT *
FROM Customers
WHERE CustomerID = ( SELECT CustomerID
                     FROM Orders
                     WHERE OrderID = 'D01' );
```

Conditions Involving Relations

∗ There are a number of SQL operators that we can apply to a relation $\mathcal{R}$ and produce a boolean result.

∗ However, the relation $\mathcal{R}$ must be expressed as a subquery and s is a scalar value. In this situation, the subquery $\mathcal{R}$ is required to produce a *one column relation*.

∗ Here are the definitions of the operators:

1. EXISTS $\mathcal{R}$ is true if and only if $\mathcal{R}$ is not empty.
2. s IN $\mathcal{R}$ is true if and only if s is equal to one of the values in $\mathcal{R}$. Likewise, s NOT IN $\mathcal{R}$ is true if and only if s is equal to no value in R.

Conditions Involving Relations

3. $s > \text{ALL } \mathcal{R}$ is true if and only if s is greater than every value in unary relation $\mathcal{R}$. Similarly, the $>$ operator could be replaced by any of the other five comparison operators, with the analogous meaning: s stands in the stated relationship to every tuple in $\mathcal{R}$. For instance, $s <> \text{ALL } \mathcal{R}$ is the same as s NOT IN $\mathcal{R}$.
4. $s > \text{ANY } \mathcal{R}$ is true if and only if s is greater than at least one value in unary relation $\mathcal{R}$. Similarly, any of the other five comparisons could be used in place of $>$, with the meaning that s stands in the stated relationship to at least one tuple of $\mathcal{R}$. For instance, $s = \text{ANY } \mathcal{R}$ is the same as s IN $\mathcal{R}$.

Conditions Involving Relations

The EXISTS, ALL, and ANY operators can be negated by putting NOT in front of the entire expression.

- NOT EXISTS $\mathcal{R}$ is true if and only if R is empty.
- NOT $s >=$ ALL $\mathcal{R}$ is true if and only if s is not the maximum value in $\mathcal{R}$.
- NOT $s >$ ANY $\mathcal{R}$ is true if and only if s is the minimum value in $\mathcal{R}$.

Conditions Involving Tuples

$\ast$ If a tuple t has the same number of components as a relation $\mathcal{R}$, then it makes sense to compare t and R in expressions.
$\ast$ Examples are t IN $\mathcal{R}$ or t $<>$ ANY $\mathcal{R}$.

e.g. Support *Warehouse Management* database scheme:

1. **Categories(<u>CategoryID</u>, CategoryName, Description)**
2. **Products(<u>ProductID</u>, ProductName, UnitPrice, CategoryID)**
3. **Warehouses(<u>WarehouseID</u>, Name, Address, CategoryID)**
4. **InStocks(<u>WarehouseID, ProductID</u>, Quantity)**

e.g.

Finding information about product, that it can be store in warehouse 'W01'.

```
SELECT *
FROM Products
WHERE (ProductID, CategoryID) IN
            ( SELECT ProductID, B.CategoryID
              FROM Warehouses A, Products B
              WHERE WarehouseID = 'W01' AND
                   A.CategoryID = B.CategoryID
            );
```

Correlated Subqueries

∗ The simplest subqueries can be evaluated *once and for all*, and the result used in a higher-level query.
∗ *Nested subqueries* requires the subquery to be evaluated many times. This type is called a *correlated subquery*.

e.g. Finding information about warehouse and product that instock with quantity > 30.

```
SELECT A.∗, B.∗
FROM Warehouses A, Products B
WHERE A.CategoryID = B.CategoryID
          AND 30 < ( SELECT Quantity
                      FROM Instocks
                      WHERE WarehouseID = A.WarehouseID
                         AND ProductID = B.ProductID )
ORDER BY WarehouseID, ProductID;
```

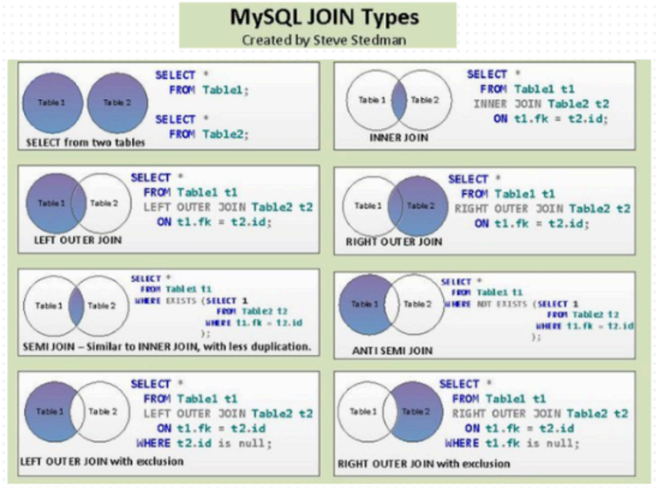
Subqueries in FROM Clauses

* Subqueries is as relations in a FROM clause.
* In a FROM list, use a parenthesized subquery and give it a alias.

e.g. Finding information about product that store in warehouse with WarehouseID = 'W01'.

```
SELECT A.*
FROM Products A, ( SELECT ProductID
                   FROM Instocks
                   WHERE WarehouseID = 'W01'
                 ) B
WHERE A.ProductID = B.ProductID;
```

SQL Join Expressions



MySQL JOIN Types

Created by Steve Stedman

SELECT from two tables

```
SELECT *
  FROM Table1;

SELECT *
  FROM Table2;
```

INNER JOIN

```
SELECT *
  FROM Table1 t1
 INNER JOIN Table2 t2
    ON t1.fk = t2.id;
```

LEFT OUTER JOIN

```
SELECT *
  FROM Table1 t1
  LEFT OUTER JOIN Table2 t2
    ON t1.fk = t2.id;
```

RIGHT OUTER JOIN

```
SELECT *
  FROM Table1 t1
 RIGHT OUTER JOIN Table2 t2
    ON t1.fk = t2.id;
```

SEMI JOIN – Similar to INNER JOIN, with less duplication.

```
SELECT *
  FROM Table1 t1
 WHERE EXISTS (SELECT 1
                 FROM Table2 t2
                WHERE t1.fk = t2.id
              );
```

ANTI SEMI JOIN

```
SELECT *
  FROM Table1 t1
 WHERE NOT EXISTS (SELECT 1
                     FROM Table2 t2
                    WHERE t1.fk = t2.id
                  );
```

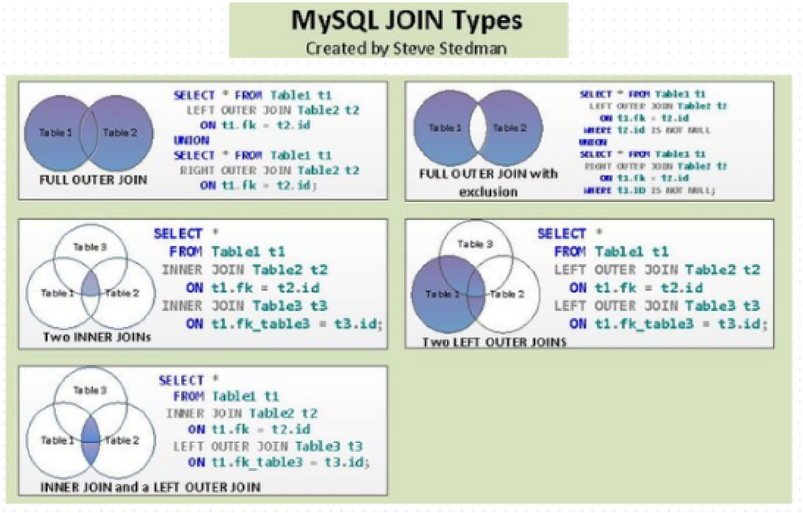
LEFT OUTER JOIN with exclusion

```
SELECT *
  FROM Table1 t1
  LEFT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
 WHERE t2.id is null;
```

RIGHT OUTER JOIN with exclusion

```
SELECT *
  FROM Table1 t1
 RIGHT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
 WHERE t1.fk is null;
```

SQL Join Expressions



MySQL JOIN Types

Created by Steve Stedman

FULL OUTER JOIN

```
SELECT * FROM Table1 t1
  LEFT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
UNION
SELECT * FROM Table1 t1
  RIGHT OUTER JOIN Table2 t2
    ON t1.fk = t2.id;
```

FULL OUTER JOIN with exclusion

```
SELECT * FROM Table1 t1
  LEFT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
 WHERE t2.id IS NOT NULL
UNION
SELECT * FROM Table1 t1
  RIGHT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
 WHERE t1.ID IS NOT NULL;
```

Two INNER JOINs

```
SELECT *
  FROM Table1 t1
 INNER JOIN Table2 t2
    ON t1.fk = t2.id
 INNER JOIN Table3 t3
    ON t1.fk_table3 = t3.id;
```

Two LEFT OUTER JOINS

```
SELECT *
  FROM Table1 t1
  LEFT OUTER JOIN Table2 t2
    ON t1.fk = t2.id
  LEFT OUTER JOIN Table3 t3
    ON t1.fk_table3 = t3.id;
```

INNER JOIN and a LEFT OUTER JOIN

```
SELECT *
  FROM Table1 t1
 INNER JOIN Table2 t2
    ON t1.fk = t2.id
  LEFT OUTER JOIN Table3 t3
    ON t1.fk_table3 = t3.id;
```

SQL Join Expressions

Let $\mathcal{R}_1$, $\mathcal{R}_2$ are relations. $\mathcal{P}$ is a condition expression.

1. Cross Join: $\mathcal{R}_1 \times \mathcal{R}_2$. FROM $\mathcal{R}_1$, $\mathcal{R}_2$
2. Inner Join:
   - Join: FROM $\mathcal{R}_1$ [Inner] Join $\mathcal{R}_2$ On $\mathcal{P}$
   - Theta Join: FROM $\mathcal{R}_1$ Join $\mathcal{R}_2$ On $\mathcal{P}$
   - Natural Join: FROM $\mathcal{R}_1$ Natural Join $\mathcal{R}_2$
     FROM $\mathcal{R}_1$ Join $\mathcal{R}_2$ On $\mathcal{R}_1.X = \mathcal{R}_2.X$

SQL Join Expressions

Outer Join:

- Left Outer Join:
  FROM $\mathcal{R}_1$ Left Outer Join $\mathcal{R}_2$ On $\mathcal{P}$
  FROM $\mathcal{R}_1$ Natural Left Outer Join $\mathcal{R}_2$
- Right Outer Join:
  FROM $\mathcal{R}_1$ Right Outer Join $\mathcal{R}_2$ On $\mathcal{P}$
  FROM $\mathcal{R}_1$ Natural Right Outer Join $\mathcal{R}_2$
- Full Outer Join:
  FROM $\mathcal{R}_1$ Full Join $\mathcal{R}_2$ On $\mathcal{P}$
  FROM $\mathcal{R}_1$ Natural Full Join $\mathcal{R}_2$

e.g.

Finding information about customer have never order.

```
SELECT A.∗
FROM Customers A Natural Left Outer Join Orders B
WHERE B.OrderID is Null
ORDER BY CustomerName;
```

Eliminating Duplicates

∗ SQL system does not eliminate duplicates. Thus, the SQL response to a query may list the same tuple several times.

∗ If we do not wish duplicates in the result, then we may follow the keyword SELECT by the keyword DISTINCT.

SELECT DISTINCT X

∗ One might be tempted to place DISTINCT after every SELECT, on the theory that it is harmless.

∗ So the relation must be sorted or partitioned.

Grouping and Aggregation Operators

In algebra: Aggregation functions and Group.

$$_{g_1,g_2,\ldots,g_n}\mathcal{G}_{f_1(A_1),f_2(A_2),\ldots,f_m(A_m)}(\mathcal{U})$$

In SQL:

```
SELECT < One or more Attritbutes in Group clause >
          [, Aggregation Functions]
FROM < Relations >
[WHERE < Tuple-Condition >]
GROUP BY <Attribute want to Group >
[HAVING < Group-Condition >]
```

Aggregation Operators

- Five aggregation operators: Sum(·), Avg(·), Min(·), Max(·), and Count(·).
- Augument is a scalar valued expression, typically a column name, in a SELECT clause.
- Count(*): Counts all the tuples in the relation that agree with condition on WHERE clause.
- Another option of eliminating duplicates from the column before applying the aggregation operator by using the keyword Distinct.
- Count(Distinct A) counts the number of distinct values in column A.
- We could use any of the other operators in place of Count here.

Grouping

∗ To group tuples, using GROUP BY clause, following the WHERE clause.

Notice:

1. ∗ GROUP BY are followed by a list of *grouping attributes.*
2. ∗ *Whatever aggregation operators* are used in the SELECT clause are applied *only within groups.*

e.g. Finding total quantity for each product its order.

```
SELECT ProductID, Sum(Quantity)
FROM OrderDetails
GROUP BY ProductID;
```

Grouping, Aggregation, and Nulls

When tuples have nulls, there are a few rules we must remember:

- The value Null is ignored in any aggregation. It does not contribute to a sum, average, or count of an attribute, nor can it be the minimum or maximum in its column. For example, Count(*) is always a count of the number of tuples in a relation, but Count(A) is the number of tuples with *non-NULL* values for attribute A.
- On the other hand, Null is treated as an ordinary value when forming groups. That is, we can have a group in which one or more of the grouping attributes are assigned the value Null.
- When we perform any aggregation except count over an empty bag of values, the result is Null. The count of an empty bag is 0.

e.g.

Show order total cost for each customer (ID and Name).

```
SELECT C.CustomerID, CustomerName,
       Sum(Quantity × Price) As 'Total Cost'
FROM Orders A, OrderDetails B, Customers C
WHERE A.OrderID = B.OrderID
     And A.CustomerID = C.CustomerID
GROUP BY C.CustomerID, CustomerName;
```

e.g. Show order total cost for each order at year 2018.

```
SELECT A.OrderID, Sum(Quantity × Price) As 'Total Cost'
FROM Orders A, OrderDetails B
WHERE A.OrderID = B.OrderID
     And strftime(%Y,OrderDate) = 2018
GROUP BY A.OrderID;
```

HAVING Clauses

Condition after WHERE restrict tuples prior to grouping.

Condition after HAVING restrict group.

e.g. Suppose we want to print the order total cost for only those cost of one product in its order at least 100.

```
SELECT OrderID, Sum(Quantity × Price) As 'Total Cost'
FROM OrderDetails
GROUP BY OrderID
HAVING Min(Quantity × Price) >= 100;
```

There are several rules we must remember about HAVING clauses:

- An aggregation in a HAVING clause applies only to the tuples of the group being tested.
- Any attribute of relations in the FROM clause may be aggregated in the HAVING clause.

Finding Maximum Group

e.g. Suppose finding the Order in which Total Cost is the maximum.

```
SELECT OrderID, Sum(Quantity × Price) As 'TotalFree'
FROM OrderDetails
GROUP BY OrderID
HAVING TotalFree >= ALL ( SELECT Sum(Quantity × Price)
                          FROM OrderDetails
                          GROUP BY OrderID
                        )
```

But SQLite is not support >= ALL operator, so that we use this algorithm below:

Finding Maximum Group in SQLite

This algorithm find maximum group in SQLite:

```
SELECT OrderID, Sum(Quantity × Price) As 'TotalFree'
FROM OrderDetails
GROUP BY OrderID
HAVING TotalFree =
        ( SELECT Max(TotalFree)
            FROM ( SELECT Sum(Quantity × Price) AS 'TotalFree'
                      FROM OrderDetails
                      GROUP BY OrderID
                    )
        )
```

Integrity Constraints

Parts of Integrity Constraints

Suppose $\mathcal{R}(ABC)$ with $B > C$
There is an *integrity constraint* of scheme R.

Integrity Constraints

- Context: R
- Condition:

```
$\forall t \in \forall r$
    $t.B > t.C$
end.
```

- Influence table:

	Insert	Delete	Update
R	+	-	+(B/C)

Parts of Integrity Constraints

- In influence table, plus($+$) sign if statement (operation) on relation maybe enforce business rule's wrong. Otherwise, minus(-) sign.
- With plus sign we write one *Trigger* to control that operation.
- Trigger is an event-driven action that is run automatically when a specified change operation (INSERT, UPDATE, and DELETE statement) is performed on a specified table.
- Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

Trigger (SQLite)

```
CREATE TRIGGER [IF NOT EXISTS] triggerName
          [BEFORE/AFTER/INSTEAD OF]
          [INSERT/UPDATE/DELETE]
          ON tableName
BEGIN
         [WHEN condition]
         BEGIN
             statements
         END;
END;

DROP TRIGGER [IF EXISTS] triggerName;
```

Trigger (SQLite)

We can access the data of the row being inserted, deleted, or updated using the OLD and NEW references in the form:

- OLD.Attribute
- NEW.Attribute

OLD and NEW references are available depending on the event that causes the trigger to be fired:

Action	Reference
INSERT	NEW is avilable
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

e.g.

Suppose we want to validate email address before insert a new customer into the Customer relation.

```
CREATE TRIGGER validateEmailCustomer
            BEFORE INSERT ON Customer
BEGIN
     SELECT CASE
         WHEN NEW.email NOT LIKE '%_@__%.__%' THEN
             RAISE(ABORT, 'Invalid email address')
     END;
END;
```

e.g.

When insert a tuple, if the email is not valid, the RAISE function aborts the insert and issues an error message 'Invalid email address'.

e.g.

```
INSERT INTO Customer
    VALUES('C01', 'John', '188, ...','0123',
    'John.Nguyen@database.net','vietnam');
```

Because the email is valid, the insert statement executed successfully.

SELECT * FROM Customer

Result:

('C01', 'John', '188, ...','0123', 'John.Nguyen@database.net','vietnam')

SQL Common Integrity Constraints

```
CREATE TABLE tableName (
    att1 datatype columnConstraint,
    att2 datatype columnConstraint,
    ...,
    Constraint tableContraintName < Logic expression >
);
```

- Constraints are the rules enforced on a data columns on table.
- Constraints could be column level or table level.
- Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.
- Following are commonly used constraints available in SQLite.

SQL Common Integrity Constraints

1. NOT NULL constraint: Ensures that a column cannot have NULL value.
2. DEFAULT constraint: Provides a default value for a column when none is specified.
3. UNIQUE constraint: Ensures that all values in a column are different.
4. PRIMARY Key: Uniquely identifies each row/record in a database table.
5. CHECK constraint: Ensures that all values in a column satisfies certain conditions.

At column level contraint, each column followed by common contraint above.

Table Constraint

∗ At the table level, SQLite supports the UNIQUE, CHECK, and PRIMARY KEY constraints.

∗ The check constraint is very similar, requiring only an expression (CHECK(expression)).

∗ Both the UNIQUE and PRIMARY KEY constraints, may be require a list of columns:

e.g.
UNIQUE (columnName, [...]),

PRIMARY KEY (columnName, [...])).

e.g. NOT NULL Constraint

```
CREATE TABLE Products (
   ProductID        Text      Primary Key,
   ProductName      Text      Not Null,
   UnitPrice        Real      Not Null,
   CategoryID       Text      Not Null
);
```

e.g. DEFAULT Constraint

```
CREATE TABLE Orders (
   OrderID        Text    Primary Key,
   OrderDate      Text    Default  Current_Date,
   RequiredDate   Text    Not Null,
   CustomerID     Text
);
```

When insert data into table, we can miss default column name to get default value:
e.g. suppose now is '2018-10-20'

```
insert into Orders (OrderID, RequiredDate, CustomerID)
Values ('O01', '2018-10-22', 'C01');
```

```
Select * from Orders;
('O01', '2018-10-20', '2018-10-22', 'C01')
```

e.g. UNIQUE Constraint

```
CREATE TABLE Categories (
    CategoryID       Text    Primary Key,
    CategoryName     Text    Not Null Unique,
    Deacription      Text
);
```

CHECK Constraint

∗ *CHECK constraint at the column level:*

```
CREATE TABLE tableName (
    ...,
    columnName dataType CHECK(expression),
    ...
);
```

∗ *CHECK constraint at the table level:*

```
CREATE TABLE tableName (
    ...,
    CHECK(expression)
);
```

e.g. Table Constraint

```
CREATE TABLE Orders (
OrderID        Text    Primary Key,
OrderDate      Text    Default  Current_Date,
RequireDate    Text    Not Null,
CustomerID     Text,
Constraint ValidateRequireDate Check (
        strftime('%d', OrderDate) <=
        strftime('%d', RequiredDate) )
);
```

SQL Foreign Key Constraints

Foreign key: In a relation, attributes (one or many) are called foreign key if they are not a key in this relation, but a primary key in another relation.

Foreign key assert that a value appearing in one relation must also appear in the primary key component(s) of another relation.

e.g.

Class(ClassID, Description)

C01

C02

Student(StudentID, Name, Address, Email, *ClassID*)

C01

C02

C02

~~C03~~

Declaring Foreign Key Constraints

Let $\mathcal{R}_1(\underline{X}Y)$, $\mathcal{R}_2(Z, \overline{X})$

Declaring attributes of relation to be a foreign key, referencing some attribute(s) of a second relation (possibly the same relation) must be have twofold:

- Attribute(s) X of the $\mathcal{R}_1$ relation must be declared UNIQUE or the PRIMARY KEY. Otherwise, $\mathcal{R}_2$ cannot make the foreign key declaration.
- $\pi_X(\mathcal{R}_2) \subseteq \pi_X(\mathcal{R}_1)$

Declaring Foreign Key Constraints

When foreign key constraint is declared, the system has to reject the violating modification.

There are two policy for Update and Delete tuples in foreign key:

1. *Default* policy: Reject Violating Modifications.
2. *Cascade* policy: When changes to the referenced attribute(s) in $\mathcal{R}_1$. There are following change to the foreign key in $\mathcal{R}_2$.
3. *Null* policy: When a modification to the referenced relation ($\mathcal{R}_1$) affects a foreign-key value (in $\mathcal{R}_2$) is changed to NULL.

Usually declare: Update Cascade; Delete Set Null.

e.g.

CREATE TABLE $\mathcal{R}_2$ **(**
  Z datatype,
  X datatype,
  Foreign Key (X) References $\mathcal{R}_1(X)$
    [On Update $<$ Cascade / Set Null $>$]
    [On Delete $<$ Cascade / Set Null $>$] **);**

e.g.

```
Create Table ShipperDetail (
  ShipperID Text, ProductID Text, Quantity int,
  Primary Key (ShipperID, ProductID),
  Foreign Key (ShipperID) References Shippers(ShipperID)
     On Delete Set Null On Update Cascade,
  Foreign Key (ProductID) References Products(ShipperID)
     On Delete Set Null On Update Cascade );
```

Views

- CREATE TABLE statement create table in database, it stores in physical.
- There is another class of SQL relations, called *(virtual) views*, that do not exist physically. Rather, they are defined by an expression much like a query.
- Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify views.
- Views in SQLite *do not allow* to perform any operations like INSERT, UPDATE and DELETE on views.

Views

```
CREATE VIEW viewName AS
SELECT column1, column2, ...
FROM tableName
[ WHERE condition ];
```

```
DROP VIEW viewName;
```

e.g. Create a view for customer with total orders

```
CREATE VIEW CustomerTotalOrder AS
SELECT CustomerID, Sum(UnitPrice * Quantity) AS Total
FROM Orders A, OrderDetail B
WHERE A.OrderID = B.OrderID
GROUP BY CustomerID;
```

Querying Views

A view may be queried exactly as if it were a stored table.

e.g. Finding information about customer with total of orders:

SELECT A.*, Total
FROM Customers A, CustomerTotalOrder B
WHERE A.CustomerID = B.CustomerID

Querying Views

We can replace each view in a FROM clause by a subquery that is identical to the view definition.

```
SELECT A.*, Total
FROM Customers A, ( SELECT CustomerID,
                 Sum(UnitPrice * Quantity) AS Total
                 FROM Orders A, OrderDetail B
                 WHERE A.OrderID = B.OrderID
                 GROUP BY CustomerID ) B
WHERE A.CustomerID = B.CustomerID
```

Renaming Attributes

We can give a view's attributes in new names.

```
CREATE VIEW ProductInstock(Product, TotalQuantity) AS
SELECT ProductID, Sum(Quantity)
FROM Instocks
GROUP BY ProductID
```

Notice: Views can modify in tuples by Insert, Delete and Update by SQL statements. But, it refer some *confuse conditions*. We do not present in here.

What is an index?

- Each row in table has a *rowID* sequence number used to identify the row.
- Therefore, we can consider a table as a list of pairs: *rowID* and *one or more column* we want to sort values; It is called *Index table*.
- Index is an additional data structure that helps speed up querying, join, and grouping operations.
- But, Index *slows down data input*, with the update and the insert statements.
- Indexes can be created or dropped with *no effect on the data.*

Create & Drop Indexes

CREATE INDEX indexName **ON** tableName column;

CREATE INDEX indexName **ON** tableName
        (cloumn1, column2, ...);

CREATE UNIQUE INDEX indexName **ON** tableName column;

DROP INDEX indexName;

e.g.

Categories:

rowID	CategoryID	CategoryName	Description
1	c1	Rice	...
2	c2	Electronic	...
3	c3	Flower	...

```
CREATE INDEX CategoryNameIndex ON
          Categories(CategoryName ASC)
```

Index:

CategoryName	rowID
Electronic	2
Flower	3
Rice	1

When should indexes be created

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When should indexes be avoided

- The table is small.
- The columns are not often used as a condition in the query.
- The column is updated frequently.