# Business Analytics and Data Driven Decision Making

## Introduction to Datawarehouse Modelling & Basics of SQL

Abid Hussain,
Associate Professor
Center for Business Data Analytics
Department of Digitalization, Copenhagen Business School, Copenhagen, Denmark.

Contact: ah.digi@cbs.dk

CBS COPENHAGEN BUSINESS SCHOOL HANDELSHØJSKOLEN

EFMD EQUIS ACCREDITED · AACSB ACCREDITED · ASSOCIATION OF MBAS ACCREDITED · CEMS · PIM Partnership in International Management

# AGENDA

# Agenda

- Dataware house Models
- Introduction to Structured Query Language (SQL)
  - Select Clause
  - Aggregation
  - Group By
  - String Functions
- Join Relations and Types of Joins

# Learning Objective

- Understand the structure of relations/tables suitable for (Visual) analysis
- Understand use of SQL for reporting
- **Strong understanding of GROUP BY concept for aggregations**
- **I don’t expect that you will remember syntax !**

# Literature

- This lecture will use examples and illustrations for the following books.

Silberschatz, A., Korth, H., Sudarshan, S. Database System Concepts. 7th Edition. McGraw-Hill Education. ISBN13: 9780078022159

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# DATAWARE HOUSE / REPORTING SCHEMAS FOR DATA

# Operational vs Analytical Information

- Operational Information
  - Transactional database storing day to day operations
  - Result of individual transactions i.e. Sales, Purchases, Logs, Conversations
- Analytical Information
  - To support data analysis tasks
  - Transformed from operational databases

# Operational vs Analytical Information

- Data Makeup Differences.
  - Summarized data
  - Granularity of time span
- Technical Differences.
  - Redundancy allowed.
  - Less frequent updates and access.
- Functional Differences.
  - Consumed by decision makers.
  - Organized and structured around subject of analysis

**MEMBERS DATABASE**

MEMBER

| MemberID | MemberName | MemberGender | MLevelID | DateMembershipPaid | ValidUntil |
|---|---|---|---|---|---|
| 111 | Joe | M | A | 1/1/2020 | 1/1/2021 |
| 222 | Sue | F | B | 1/1/2020 | 1/1/2021 |
| 333 | Pam | F | A | 1/2/2020 | 1/2/2021 |
| ... | ... | ... | ... | ... | ... |

MEMBERSHIP LEVEL

| MLevelID | MLevelType | MLevelFee | MLevelDescription |
|---|---|---|---|
| A | Gold | $100 | Includes the Pool Usage |
| B | Basic | $50 | No Pool Usage |

**NONMEMBERS DATABASE**

NONMEMBER DAILY VISIT

| DVisitTID | DVisitLevelID | DVisitDate | DVisitorGender |
|---|---|---|---|
| 11xx22 | YP | 1/1/2020 | M |
| 11xx23 | NP | 1/2/2020 | M |
| 11xx24 | YP | 1/2/2020 | F |
| ... | ... | ... | ... |

NONMEMBER VISIT LEVEL

| DVisitLevelID | DVisitLevelFee | DVisitLevelType |
|---|---|---|
| YP | $15 | With Pool Usage |
| NP | $10 | Without Pool Usage |

**Figure 7.1** Application-oriented databases serving the Vitality Health Club Visits and Payments Application

on

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

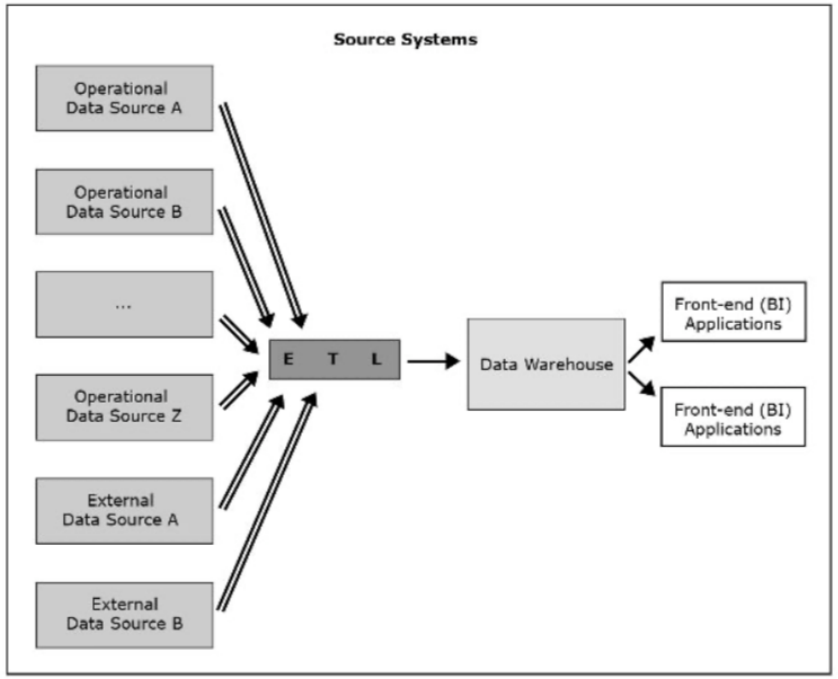
REVENUE

| RevenueRecordID | Date | GeneratedBy | ClientGender | Pool Use Included in Purchase | Amount |
|---|---|---|---|---|---|
| 1000 | 1/1/2020 | Member | M | Yes | $100 |
| 1001 | 1/1/2020 | Member | F | No | $50 |
| 1002 | 1/1/2020 | Nonmember | M | Yes | $15 |
| 1003 | 1/2/2020 | Member | F | Yes | $100 |
| 1004 | 1/2/2020 | Nonmember | M | No | $10 |
| 1005 | 1/2/2020 | Nonmember | F | Yes | $15 |
| ... | | | | ... | ... |

**Figure 7.2** A subject-oriented database for the analysis of the subject revenue

# Components of Analytical Database Systems / Warehouse

- Source Systems. i.e. Operational Databases, external data streams.
- ETL Processes for Data transformation and storage into Datawares.
- Dataware house / Structured schematic of data for analysis is a repository of analytical data.




Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# Modelling of Analytical Databases

- Dimensional Modeling
  - is a data design methodology used for designing subject oriented analytical databases.
  - Consists of two types of tables connected in different structures
    - Dimentional Tables
    - Fact Tables

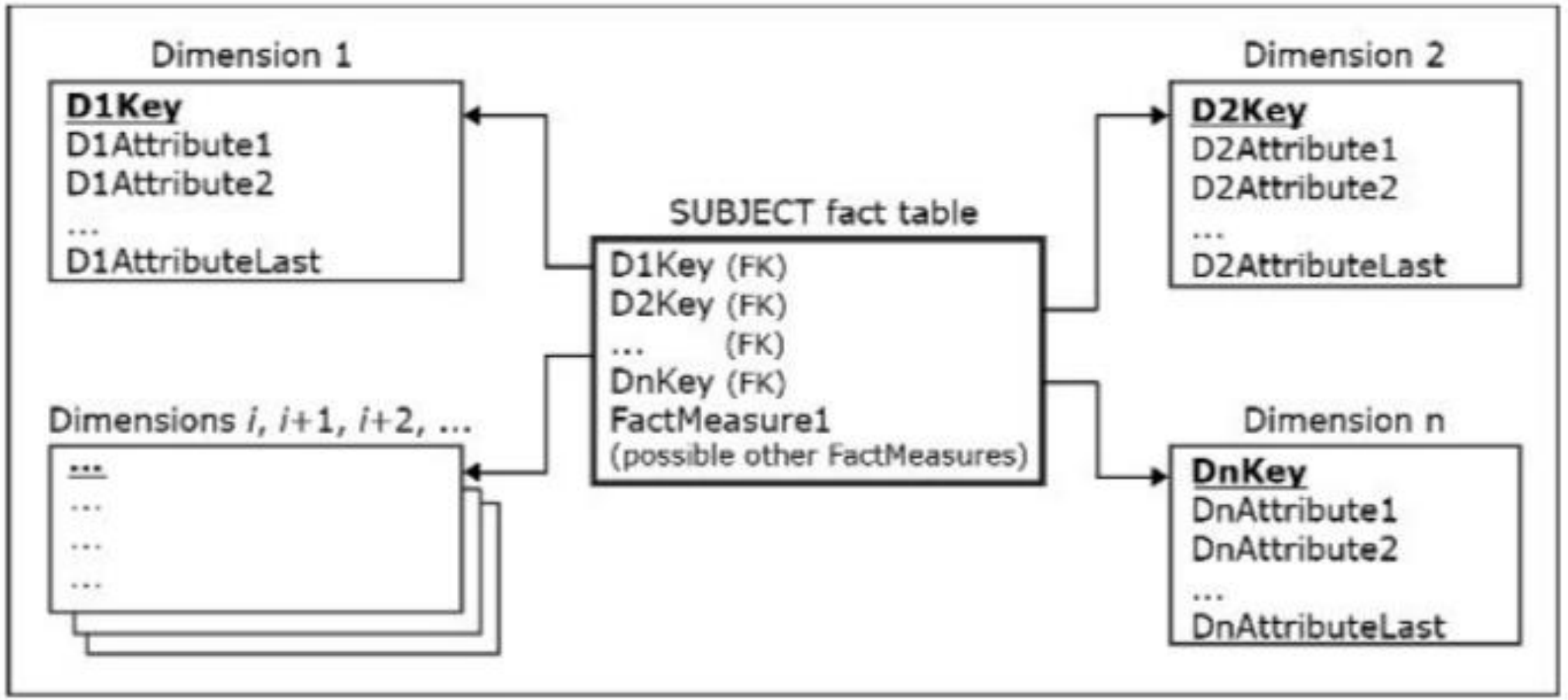
Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# Modelling of Analytical Databases

- Dimensional Modeling
  - is a data design methodology used for designing subject oriented analytical databases.
  - Consists of two types of tables connected in different structures
    - Dimentional Tables
    - Fact Tables

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# Modelling of Analytical Databases




Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# Analysis Data Models Schema Types

- Star Schema
  - One fact table for subject of analysis. i.e. Sales
  - Multiple Dimension table according to required analysis
- Snowflake Schema
  - One fact table like Star Schema
  - Normalized (Redundancy removed) expanded dimension table(s)
- Fact Constellation Schema
  - Multiple Fact tables to support analysis of multiple subjects
  - Dimensions are re-used

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

REGION

| RegionID | RegionName |
|---|---|
| C | Chicagoland |
| T | Tristate |

INCLUDES

| ProductID | TID | Quantity |
|---|---|---|
| 1X1 | T111 | 1 |
| 2X2 | T222 | 1 |
| 3X3 | T333 | 5 |
| 1X1 | T333 | 1 |
| 4X4 | T444 | 1 |
| 2X2 | T444 | 2 |
| 4X4 | T555 | 4 |
| 5X5 | T555 | 2 |
| 6X6 | T555 | 1 |

VENDOR

| VendorID | VendorName |
|---|---|
| PG | Pacific Gear |
| MK | Mountain King |

STORE

| StoreID | StoreZip | RegionID |
|---|---|---|
| S1 | 60600 | C |
| S2 | 60605 | C |
| S3 | 35400 | T |

CATEGORY

| CategoryID | CategoryName |
|---|---|
| CP | Camping |
| FW | Footwear |

SALESTRANSACTION

| TID | CustomerID | StoreID | TDate |
|---|---|---|---|
| T111 | 1-2-333 | S1 | 1-Jan-2020 |
| T222 | 2-3-444 | S2 | 1-Jan-2020 |
| T333 | 1-2-333 | S3 | 2-Jan-2020 |
| T444 | 3-4-555 | S3 | 2-Jan-2020 |
| T555 | 2-3-444 | S3 | 2-Jan-2020 |

CUSTOMER

| CustomerID | CustomerName | CustomerZip |
|---|---|---|
| 1-2-333 | Tina | 60137 |
| 2-3-444 | Tony | 60611 |
| 3-4-555 | Pam | 35401 |

PRODUCT

| ProductID | ProductName | ProductPrice | VendorID | CategoryID |
|---|---|---|---|---|
| 1X1 | Zzz Bag | $100 | PG | CP |
| 2X2 | Easy Boot | $70 | MK | FW |
| 3X3 | Cosy Sock | $15 | MK | FW |
| 4X4 | Dura Boot | $90 | PG | FW |
| 5X5 | Tiny Tent | $150 | MK | CP |
| 6X6 | Biggy Tent | $250 | MK | CP |

**Figure 8.3** Data records for the ZAGI Retail Company Sales Department Database shown in Figure 8.2

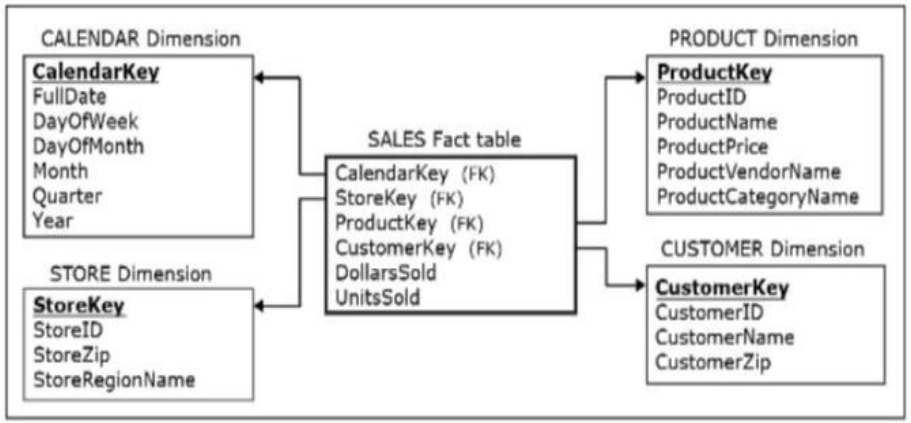
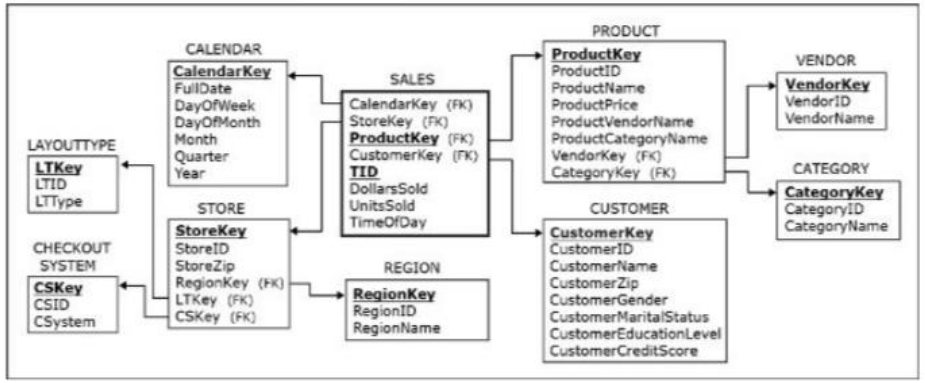



**Figure 8.4** A dimensional model for an analytical database for the ZAGI Retail Company, whose subject of analysis is sales



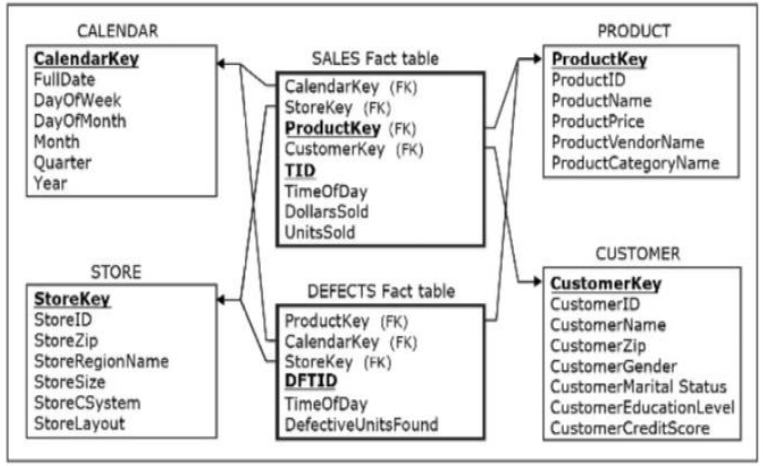




**Figure 8.19** An expanded dimensional model with two subjects

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

REGION

| RegionID | RegionName |
|---|---|
| C | Chicagoland |
| T | Tristate |

INCLUDES

| ProductID | TID | Quantity |
|---|---|---|
| 1X1 | T111 | 1 |
| 2X2 | T222 | 1 |
| 3X3 | T333 | 5 |
| 1X1 | T333 | 1 |
| 4X4 | T444 | 1 |
| 2X2 | T444 | 2 |
| 4X4 | T555 | 4 |
| 5X5 | T555 | 2 |
| 6X6 | T555 | 1 |

VENDOR

| VendorID | VendorName |
|---|---|
| PG | Pacific Gear |
| MK | Mountain King |

STORE

| StoreID | StoreZip | RegionID |
|---|---|---|
| S1 | 60600 | C |
| S2 | 60605 | C |
| S3 | 35400 | T |

CATEGORY

| CategoryID | CategoryName |
|---|---|
| CP | Camping |
| FW | Footwear |

SALESTRANSACTION

| TID | CustomerID | StoreID | TDate |
|---|---|---|---|
| T111 | 1-2-333 | S1 | 1-Jan-2020 |
| T222 | 2-3-444 | S2 | 1-Jan-2020 |
| T333 | 1-2-333 | S3 | 2-Jan-2020 |
| T444 | 3-4-555 | S3 | 2-Jan-2020 |
| T555 | 2-3-444 | S3 | 2-Jan-2020 |

CUSTOMER

| CustomerID | CustomerName | CustomerZip |
|---|---|---|
| 1-2-333 | Tina | 60137 |
| 2-3-444 | Tony | 60611 |
| 3-4-555 | Pam | 35401 |

PRODUCT

| ProductID | ProductName | ProductPrice | VendorID | CategoryID |
|---|---|---|---|---|
| 1X1 | Zzz Bag | $100 | PG | CP |
| 2X2 | Easy Boot | $70 | MK | FW |
| 3X3 | Cosy Sock | $15 | MK | FW |
| 4X4 | Dura Boot | $90 | PG | FW |
| 5X5 | Tiny Tent | $150 | MK | CP |
| 6X6 | Biggy Tent | $250 | MK | CP |

**Figure 8.3** Data records for the ZAGI Retail Company Sales Department Database shown in Figure 8.2

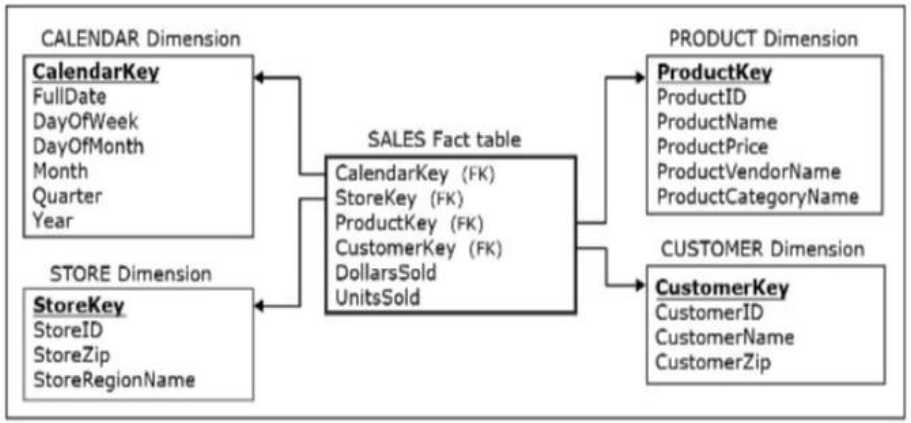
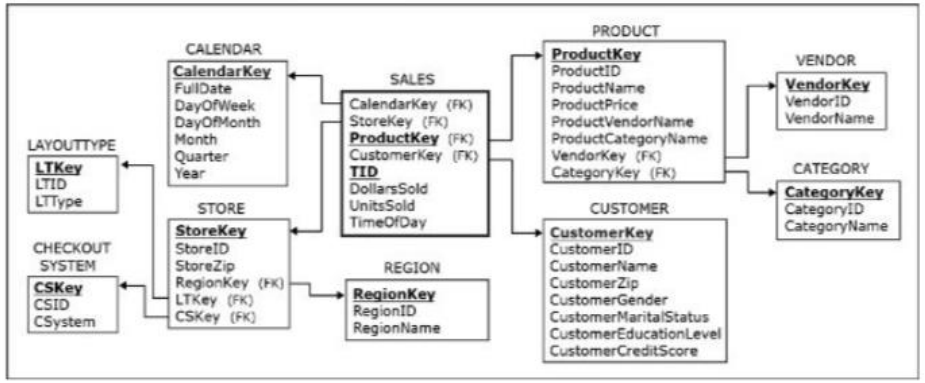



**Figure 8.4** A dimensional model for an analytical database for the ZAGI Retail Company, whose subject of analysis is sales



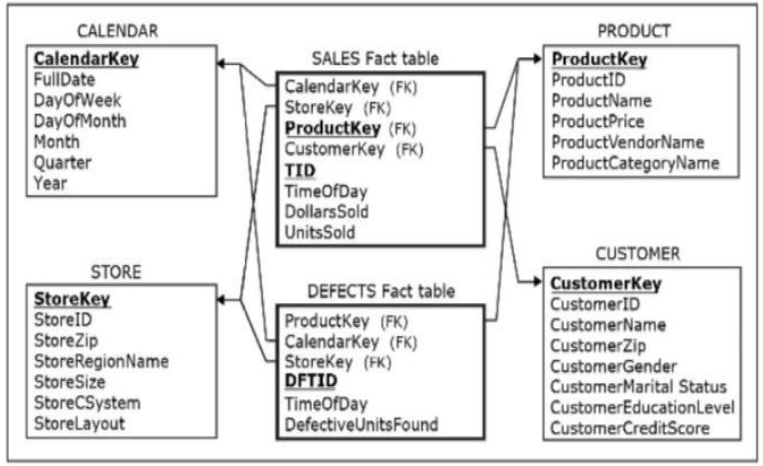




**Figure 8.19** An expanded dimensional model with two subjects

Jukic, N. (2019). Database systems: Introduction to databases and data warehouses.

# SQL QUERIES

# Basic Query Structure

- A typical SQL query has the form:

**select** $A_1, A_2, ..., A_n$

**from** $r_1, r_2, ..., r_m$

**where** $P$

  - $A_i$ represents an attribute
  - $R_i$ represents a relation
  - $P$ is a predicate.
- The result of an SQL query is a relation.

# SELECT Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  **select** *name*

  **from** *instructor*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

# SELECT Clause – Duplicate elimination

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the department names of all instructors, and remove duplicates

**select distinct** *dept_name* **from** *instructor*

# SELECT Clause – literal output

- An asterisk in the select clause denotes “all attributes”

  **select** *

  **from** *instructor*

- An attribute can be a literal with no **from** clause

  **select** ‘437’

  - Results is a table with one column and a single row with value “437”
  - Can give the column a name using: **select** ‘437’ **as** *FOO*

- An attribute can be a literal with **from** clause

  **select** ‘A’

  **from** *instructor*

  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value “A”

# SELECT Clause – Embedded Arithmetic Operations

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
  - The query:

    **select** *ID, name, salary/12*

    **from** *instructor*

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
  - Can rename "*salary/12*" using the **as** clause:

    **select** *ID, name, salary/12* **as** *monthly_salary*

# WHERE Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

  **select** *name*

  **from** *instructor*

  **where** *dept_name = 'Comp. Sci.'*
- Comparison results can be combined using the logical connectives **and, or,** and **not**
  - To find all instructors in Comp. Sci. dept with salary > 80000

    **select** *name.* **from** *instructor* **where** *dept_name = 'Comp. Sci.'* **and** *salary* > 80000

# FROM Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  **select** ∗

  **from** *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

# Cartesian Product

*instructor*

| ID | name | dept_name | salary |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 60000 |

*teaches*

| ID | course_id | sec_id | semester | year |
|---|---|---|---|---|
| 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | FIN-201 | 1 | Spring | 2010 |
| 15151 | MU-199 | 1 | Spring | 2010 |
| 22222 | PHY-101 | 1 | Fall | 2009 |

| Inst.ID | name | dept_name | salary | teaches.ID | course_id | sec_id | semester | year |
|---|---|---|---|---|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 10101 | Srinivasan | Comp. Sci. | 65000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| … | … | … | … | … | … | … | … | … |
| … | … | … | … | … | … | … | … | … |
| 12121 | Wu | Finance | 90000 | 10101 | CS-101 | 1 | Fall | 2009 |
| 12121 | Wu | Finance | 90000 | 10101 | CS-315 | 1 | Spring | 2010 |
| 12121 | Wu | Pinance | 90000 | 10101 | CS-347 | 1 | Fall | 2009 |
| 12121 | Wu | Pinance | 90000 | 12121 | FIN-201 | 1 | Spring | 2010 |
| 12121 | Wu | Finance | 90000 | 15151 | MU-199 | 1 | Spring | 2010 |
| 12121 | Wu | Pinance | 90000 | 22222 | PHY-101 | 1 | Fall | 2009 |
| … | … | … | … | … | … | … | … | … |
| … | … | … | … | … | … | … | … | … |

# Examples

- Find the names of all instructors who have taught some course and the course_id

  **select** *name, course_id*

  **from** *instructor , teaches*

  **where** *instructor.ID = teaches.ID*

- Find the names of all instructors in the Art department who have taught some course and the course_id

  **select** *name, course_id*

  **from** *instructor , teaches*

  **where** *instructor.ID = teaches.ID* ***and*** *instructor. dept_name =* 'Art'

# String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent ( % ). The % character matches any substring.
  - underscore ( _ ). The _ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.
- **se**l**ect** *name*

  **from** *instructor*

  **where** *name* **like** '%dar%'

# String Operations

- Patterns are case sensitive.
- Pattern matching examples:
  - ‘Intro%’ matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘_ _ _’ matches any string of exactly three characters.
  - ‘_ _ _ %’ matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

# Ordering the Tuples

- List in alphabetic order the names of all instructors

  **select distinct** *name*

  **from** *instructor*

  **order by** *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

# WHERE Clause predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)

  **select** *name*

  **from** *instructor*

  **where** *salary* **between** 90000 **and** 100000

# SET Operations

- Find courses that ran in Fall 2009 or in Spring 2010
  - (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)

    **union**

    (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)
- Find courses that ran in Fall 2009 and in Spring 2010
  - (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)

    **intersect**

    (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)
- Find courses that ran in Fall 2009 but not in Spring 2010
  - (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)

    **except**

    (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

# NULL Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*

  Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.

  Example: Find all instructors whose salary is null

  **select** *name*

  **from** *instructor*

  **where** *salary* **is null**

# AGGREGATE Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
- **avg:** average value

  **min:** minimum value

  **max:** maximum value

  **sum:** sum of values

  **count:** number of values

# AGGREGATE Functions

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)

    **from** *instructor*

    **where** *dept_name*= ’Comp. Sci.’;
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)

    **from** *teaches*

    **where** *semester* = ’Spring’ **and** *year* = 2010;
- Find the number of tuples in the *course* relation
  - **select count** (*)

    **from** *course*;

# AGGREGATE Functions – Group BY

- Find the average salary of instructors in each department
- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

**select** *dept_name*,

**avg** (*salary*) **as** *avg_salary*

**from** *instructor*

**group by** *dept_name*;

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | Finance | 80000 |
| 32343 | El Said | History | 60000 |
| 58583 | Califieri | History | 62000 |
| 15151 | Mozart | Music | 40000 |
| 33456 | Gold | Physics | 87000 |
| 22222 | Einstein | Physics | 95000 |

| *dept_name* | *salary* |
|---|---|
| Biology | 72000 |
| Comp. Sci. | 77333 |
| Elec. Eng. | 80000 |
| Finance | 85000 |
| History | 61000 |
| Music | 40000 |
| Physics | 91000 |

Silberschatz, A., Korth, H., Sudarshan, S. Database System Concepts. 7th Edition. McGraw-Hill Education. ISBN13: 9780078022159

# AGGREGATE Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

  **select** *dept_name*, **avg** (*salary*)
  **from** *instructor*
  **group by** *dept_name*
  **having avg** (*salary*) > 42000;

- **Note**: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

# AGGREGATE and NULL Values

- Total all salaries

  **select sum** (*salary* )

  **from** *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

# SET Membership and Sub Queries

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

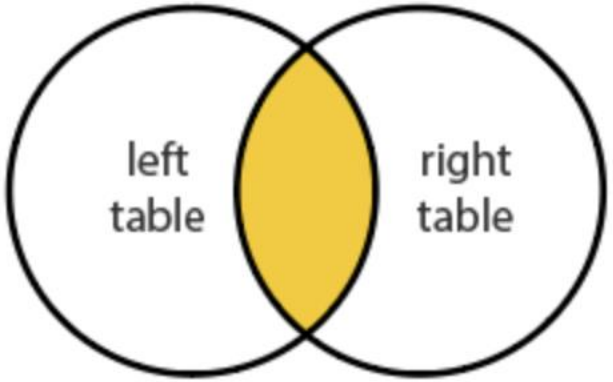
```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2010);
```

# TYPES OF JOINS

# JOIN in Relational Databases

- A SQL JOIN combines records from two Tables
- A JOIN is used to locate related records from tables
- A query may contain zero, one or multiple JOIN operations
- There are following commonly used types of Joins
  - INNER JOIN
  - LEFT JOIN
  - RIGHT JOIN

# INNER JOIN



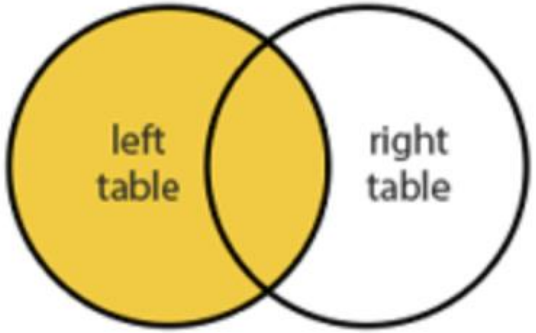

- SELECTS records with the matching values in both tables
  - **Example:**

    **select** * **from** *student* **join** *takes* **on** *student*.*ID* = *takes*.*ID*;

    or

    S**elect** * **from** *student*, *takes* **where** *student*.*ID* = *takes*.*ID*;

# LEFT (OUTER) JOIN

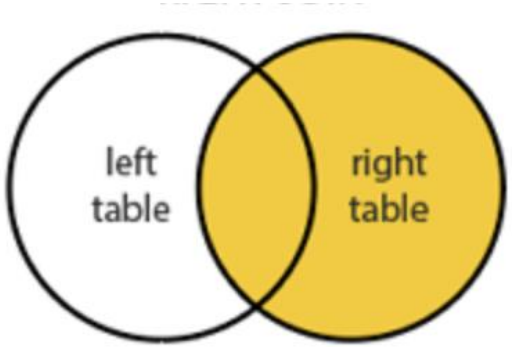



- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation

**Example:** Get a list of student who did not take a course

```
select *
from student s left join takes t
on s.id=t.id where t.id is null;
```

# LEFT (OUTER) JOIN




- The **right outer join** is symmetric to the **left outer join**
- Tuples from the right-hand- side relation that do not match any tuple in the left-hand-side relation are padded with nulls
-

**Example:** Get a list of student who did not take a course

```
select *
from takes t
right join student s on s.id=t.id
 Where s.ID is null
```

# VIEWS

# Views

- In some cases, it is not desirable for all users to see the entire logical model

  (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  **select** *ID*, *name*, *dept_name*

  **from** *instructor*
- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.
- Views are widely used to provide access for Visual Dashboards.

# Views

- A view is defined using the **create view** statement which has the form

  **create view** *v* **as** < query expression >

  - where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.
- A view in some cases and under some conditions be used for updates however Views should best be used for data fetch and Reporting Only and not for insertion or update

# Views

- A view of instructors without their salary

  **create view** *faculty* **as**
  **select** *ID*, *name*, *dept_name*
  **from** *instructor*

- Find all instructors in the Biology department

  **select** *name*
  **from** *faculty*
  **where** *dept_name* = 'Biology'

- Create a view of department salary totals

  **create view** *departments_total_salary*(*dept_name*, *total_salary*) **as**
  **select** *dept_name*, **sum** (*salary*)
  **from** *instructor*
  **group by** *dept_name*;

# Views

- A view of instructors without their salary

  **create view** *faculty* **as**
  **select** *ID*, *name*, *dept_name*
  **from** *instructor*

- Find all instructors in the Biology department

  **select** *name*
  **from** *faculty*
  **where** *dept_name* = 'Biology'

- Create a view of department salary totals

  **create view** *departments_total_salary*(*dept_name*, *total_salary*) **as**
  **select** *dept_name*, **sum** (*salary*)
  **from** *instructor*
  **group by** *dept_name*;

# THANK YOU