# מסדי נתונים רלציוניים

## מסדי נתונים

מסד נתונים או בסיס נתונים (Database, או בקיצור DB) הוא אמצעי המשמש לאחסון מסודר של נתונים במחשב, לשם אחזורם ועיבודם. הגישה לבסיס הנתונים נעשית באמצעות תוכנה ייעודית - מערכת לניהול בסיס נתונים (DBMS - DataBase Management System).

בסיס הנתונים בנוי לפי מודל לאחסון הנתונים, כמו מנגנונים פנימיים למיון ולחיפוש. ישנם מספר מודלים לבסיסי נתונים: רשתי, היררכי, טבלאי, מונחה עצמים ו-NoSQL. המודלים הללו מבטאים סוגי קשרים שונים בין הנתונים השונים. בקורס זה נלמד את המרכזיים שבהם.

## מסדי נתונים רלציוניים

המודל הנפוץ והבסיסי ביותר הוא המודל הטבלאי (Relational Databases). בסיסי הנתונים הנפוצים שבנויים במודל זה הם: MySQL, SQLite, PostgreSQL (קוד פתוח), Oracle, Microsoft SQL Server (בתשלום). כל בסיסי נתונים אלו הם ב90% זהים ושאר ההבדלים לא כל כך מהותיים.

במודל זה בסיס הנתונים בנוי מטבלאות (relations), כאשר כל טבלה מכילה מידע על ישות מסוימת (לדוגמה, סטודנטים). בכל טבלה העמודות מייצגות תכונות של הישות (לדוגמה, תואר הסטודנט או ממוצע הציונים שלו), והשורות (נקראות גם רשומות) מייצגות מופעים של הישות (סטודנט מסוים). לכל שורה כזאת יש מפתח כלשהו שמזהה באופן ייחודי את המופע של הישות (ת.ז לדוגמה). בהמשך נלמד כיצד מקשרים בין טבלאות.



שליפת מידע ופעולות עדכון ומחיקה בבסיס נתונים טבלאי נעשות באמצעות שפת SQL, המהווה ממשק המאפשר גישה לנתונים מבלי להתייחס לאופן שמירתם בבסיס הנתונים. הפקודות בשפת SQL מבוססות על אלגברה רלציונית, עליה נדבר בהמשך.

## ACID

המונח ACID הוא ראשי תיבות של: Atomicity, Consistency, Isolation, Durability. תרגום המונחים לעברית הוא אַטוֹמִיוּת, עִקְבִיּוּת, בִידוּד ועֲמִידוּת. תכונות אלה הן אבן הפינה של מסדי נתונים ומערכות לניהול תנועות, ובלעדיהן לא ניתן להבטיח את שלמות הנתונים במערכות אלה. אמנם בפועל, תכונות ה-ACID נאכפות במידה רופפת יותר כדי לשפר את ביצועי המערכת.

* **אטומיות (Atomicity)** - כל רצף פקודות מבוצעות בבת אחת, כך שאם המערכת תקרוס באמצע ביצוע הפעולות לא יהיה איבוד מידע. כלומר, כל רצף פקודות מתבצעות במלואן או לא מתבצעות כלל.
* **עקביות (Consistency)** - הבסיס נתונים תמיד נשאר עקבי, אם מנסים לבצע פעולה שתכניס את הבסיס נתונים למצב לא חוקי מבחינתו (השמה של מחרוזת במקום מספר, השמה של מספר שלא זהה לחוק שהוגדר עבור עמודה), אזי בסיס נתונים לא יאפשר ביצוע של אותה הפעולה.
* **בידוד (Isolation)** - בסיס הנתונים מאפשר לבצע פעולות רבות במקביל, כל עוד התוצאה זהה לביצוע הפעולות באופן טורי. המשמעות היא שלא ניתן לגשת לאותו תא משני מקורות במקביל (לדוגמה, משיכת סכום של שני משתמשים שונים מאותו חשבון בנק במקביל).
* **עמידות (Durability)** - פעולה שמתבצעת תמיד נשמרת בבסיס נתונים, אפילו אם היה תקלה באמצע כמו הפסקת חשמל וכו'. פעולות לא הולכות לאיבוד.

# SQL

## הגדרה

SQL (Structured Query Language) היא שפת מחשב הצהרתית לטיפול ועיבוד מידע בבסיסי נתונים טבלאיים, שמתבססת על אלגברה רלציונית. השפה מאפשרת יצירת טבלה, הכנסת נתונים, עדכונם ושליפתם. הכוונה ב"שפת מחשב הצהרתית" שהיא מתארת רק מה לבצע ולא כיצד לבצע, ומתמקדת במהות ולא בטכניקה. מכיוון שכך, ניתן להשתמש באותו משפט SQL בבסיסי נתונים שונים, הפועלים בדרכים שונות לחלוטין.

SQL הינה שפה "לא רגישה", כלומר שאין הבדל בין אותיות גדולות לאותיות קטנות, אמנם המוסכמה היא שאת כל **מילות המפתח** ב-SQL כותבים באותיות גדולות. ישנם מעט מאוד שפות שהן לא רגישות שעדיין בשימוש היום (Fortran, Cobal, Lisp, Basic, Pascal).

## סוגי ערכים

עבור כל עמודה צריך לציין את הטיפוס של הערכים שהעמודה מכילה. ניתן לקרוא על סוגי הטיפוסים ושמותיהם בקישור <https://www.techonthenet.com/mysql/datatypes.php>. בכל תא בטבלה שלא הכנסנו ערך יכיל את הערך הריק NULL.

## יצירת טבלה CREATE TABLE

יצירת טבלה נעשית בדר"כ באמצעות ה-GUI אמנם אפשר גם באמצעות הפקודה:

CREATE TABLE table\_name (col1 DATATYPE, col2 DATATYPE, …)

צריך לציין את טיפוס הנתונים בכל עמודה. דוגמה ליצירת טבלת pet עם 5 עמודות:

CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20), species VARCHAR(20), sex CHAR(1), birth DATE)

### מפתחות

בטבלה אנו מגדירים עמודות מיוחדות שישמשו כמפתחות. יש כמה סוגים של מפתחות:

* **PRIMARY KEY** - עמודה או מספר עמודות שמגדירות את הגישה לטבלה, כמו id או courseId. לכל מופע (שורה) חייב להיות ערך **ייחודי** בעמודה זו, כלומר לא יכול להיות NULL. כאשר מגדירים טבלה דרך ה-Workbench חייבים לבחור לפחות עמודה אחת שתהיה מסוג מפתח זה.
* **UNIQUE KEY** - כמו PRIMARY KEY אלא שלא לכל שורה חייב להיות ערך כזה, אלא יכולים להיות NULL. לדוגמה, מספר פספורט, לא לכולם יש מספר פספורט אך למי שכן יש הוא ייחודי.

כדי להגדיר שעמודה היא PRIMARY KEY או UNIQUE KEY יש לרשום מיד לאחר סוג העמודה את סוג המפתח הרצוי, col DATATYPE PRIMARY/UNIQUE KEY. דרך נוספת היא לאחר שהצהרנו על העמודה col לרשום פסיק ולאחריו: PRIMARY/UNIQUE KEY(col).

* **INDEX או KEY** - הם עמודות שהמידע בהן עוזר לנו למצוא את המופע הרצוי, כמו lastName. בדרך כלל נשתמש בהם בפקודות WHERE או JOIN. הגדרת עמודה בתור INDEX גורמת למסד נתונים לארגן את הטבלה כך ששליפה באמצעות עמודה זו תהיה הרבה יותר מהירה. כדי להגדיר INDEX לאחר שהצהרנו על העמודה נרשום פסיק ולאחריו INDEX(col). ניתן גם להשתמש במילה השמורה KEY.
* **Foreign Key** - כדי לקשר בין הטבלה החדשה שאנו יוצרים לטבלה קיימת אחרת, נוכל להגדיר עמודה אחת או יותר המהוות מפתח זר. המשמעות היא שעמודות אלו יכולות לקבל רק את הערכים שהם PRIMARY KEY בטבלה השנייה. באמצעות מפתחות זרים נוכל לשלוף שתי טבלאות אלו, בהמשך נלמד כיצד, בצורה נוחה יותר.

CREATE TABLE pet (petId INT **PRIMARY KEY**, name VARCHAR(20) **UNIQUE KEY**, owner INT NOT NULL, species VARCHAR(20), sex CHAR(1), birth DATE, **INDEX(owner)**)

### אילוצים

מלבד סוג הנתונים בכל עמודה, נוכל להגדיר אילוצים נוספים למידע שמכילה עמודה כלשהי באמצעות המילה CHECK. MySQL לא תומך באפשרות זו. ניתן להכניס לתוך ה-CHECK מגבלה של אפשרויות לבחור מהם, או לוודא שהערכים מתאימים לחישובים מעמודות אחרות.

CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20), species VARCHAR(20), sex CHAR(1), birth DATE, **CHECK**(sex IN('m', 'f')))

בכל עמודה אפשר להגדיר שהיא לא יכולה להכיל ערכי NULL. נרצה לעשות זאת כאשר העמודה מכילה מידע חשוב על המופעים (שורות) ותמיד נרצה שיהיה ערך כלשהו בעמודה זו. נוכל להגדיר זאת על ידי המילים NOT NULL לאחר הגדרת הטיפוס נתונים של אותה עמודה.

CREATE TABLE pet (name VARCHAR(20), **owner VARCHAR(20) NOT NULL**, species VARCHAR(20), sex CHAR(1), birth DATE)

## שינוי עמודות בטבלה ALTER TABLE

לאחר שיצרנו טבלה נוכל לשנות את מה שהגדרנו בו באמצעות פקודת ALTER TABLE table\_name. ניתן להוסיף עמודה, למחוק עמודה, לשנות את טיפוס הנתונים, להגדיר מפתחות, להגדיר NOT NULL, וכו'. צריך לשים לב שאם בטבלה כבר קיימות שורות, אזי כל שינוי בטבלה עלול להשפיע על השורות הקיימות וליצור שגיאות.

ALTER TABLE pet ADD death DATE // add column

ALTER TABLE pet DROP death // delete column

## מחיקת טבלה DROP TABLE

כדי למחוק טבלה עם כל התכונות והמופעים שלה, נשתמש בפקודה: DROP TABLE table\_name. הפקודה DELETE משמשת למחיקת שורה כמו שנלמד בהמשך.

\* כל השינויים של יצירת שינוי ומחיקת טבלה ניתן לבצע, וגם הרבה יותר נוח, דרך ה-Workbench.

## שליפה SELECT

אנו שולפים מידע מטבלה באמצעות פקודת SELECT. השליפה היא של עמודה או עמודות מתוך טבלה. סדר השורות בכל שליפה הוא אקראי, בהמשך נלמד כיצד לסדר את השורות לפי פרמטר כלשהו.

* לשליפת עמודה: SELECT col FROM table.
* לשליפת שתי עמודות ויותר: SELECT col1, col2, … FROM table.
* לשלוף את כל העמודות בטבלה: SELECT \* FROM table.
* לשליפת עמודה ולבצע עליה פעולות אריתמטיות: SELECT col\*2 FROM table. בתצוגה של השליפה שם העמודה יהיה השם המלא של הפעולה (col\*2 בדוגמה). נוכל לשנות את שם העמודה בתצוגה באמצעות המילה השמורה AS ולאחריה השם הרצוי: SELECT col\*2 FROM table AS new\_name.
* להורדת שורות כפולות בשליפה: SELECT DISTINCT col1, col2, … FROM table.

## תנאי WHERE

אם אנחנו לא רוצים לשלוף את כל השורות נוכל להשתמש בפקודת WHERE ולאחריה תנאי כלשהו שמקיימים רק השורות שאנו מעוניינים לשלוף: SELECT col FROM table WHERE condition יש חמישה סוגי תנאים שמותר להשתמש בהם:

1. **אופרטורים השוואתיים** על עמודות בטבלה - >, <, =, <=, >=, <>. האופרטור <> פירושו "שונה".

SELECT id FROM students WHERE degree < 2

1. **אופרטורים לוגיים** - AND, OR, NOT, כדי לחבר בין תנאים שונים.
2. **BETWEEN** - שולף את כל השורות שבעמודה כלשהי נמצאות בין שני ערכים **כולל**. למשל:

SELECT \* FROM grades WHERE grade BETWEEN 80 AND 90

1. **IN** - שולף את כל השורות שהערכים בסוגריים **מופיעים** בהם. ניתן גם לשימוש הפוך NOT IN. למשל:

SELECT \* FROM grades WHERE studentId IN (111,444)

1. **LIKE** - שולף את כל השורות שבהן בעמודה כלשהי מופיע ערך מסוים. נוסיף לערך זה גרשיים ' ' בהתחלה ובסוף. ניתן להוסיף % בתחילת הערך, באמצע או בסופו, שמשמעותו היא שיכולים להיות תווים נוספים במקום שיש % ולא משנה לנו מהם. למשל:

SELECT \* FROM students WHERE firstName LIKE 'Chay%'

### שאילתות מקוננות

התוצאה של שליפה היא גם כן טבלה (relation), לכן נוכל לבצע שליפה מתוך שליפה, הנקראת שאילתה מקוננת (nested queries). השאילתה המקוננת יכולה להופיע בתוך התנאי WHERE או ב-FROM ואפילו ב-SELECT, ונכתוב אותה בסוגריים. באמצעות שאילתה מקוננת נוכל לצמצם את הטבלה ממנה אנו שולפים, וכן נוכל להצליב מידע מתוך טבלאות שונות. לדוגמה:

SELECT lecturer FROM courses WHERE id NOT IN (SELECT courseId FROM grades)

## תא ריק NULL

תא ריק בטבלה יכיל את הערך NULL.

* נוכל לשלוף את כל התאים בעמודה שהם NULL באמצעות פקודה בתנאי IS NULL, או ההיפך באמצעות פקודת IS NOT NULL.
* לא ניתן לתשאל על ערך NULL, כלומר אם נשלוף מעמודה את כל הערכים השונים מ-1 והשווים ל-1 לדוגמה, השליפה תכלול את כל הערכים חוץ מאלו שהם NULL.

### COALESCE

כדי שלא נקבל בשליפה את הערך NULL, נוכל להשתמש בפקודה COALESCE(col1, col2, col3 …) ולאחריה סוגריים עם מספר כלשהו של שמות עמודות. פקודה זו אומרת שקודם ינסו לשלוף את הערך מהעמודה הראשונה בסוגריים, אך אם ערך זה NULL יעברו לעמודה הבאה ברשימה, וכן הלאה עד סוף הרשימה. אם גם בעמודה האחרונה ברשימה הערך הוא NULL, אזי השליפה גם כן תחזיר NULL. במילים אחרות, יוחזר הערך הראשון בעמודה שאינו NULL, אלא אם כן כולם NULL ואז יוחזר NULL.

נוכל להוסיף בסוף הסוגריים ערך כלשהו בגרשיים, לדוגמה 'unknown', שאם בכל העמודות ברשימה יש NULL אזי יוחזר ערך זה ולא NULL.

SELECT id, COALESCE(lastName, firstName, 'unknown') FROM students.

## שליפה לפי סדר ORDER BY

עד כה בכל פעם ששלפנו מידע מהטבלה הסדר לא היה חשוב, אמנם באמצעות פקודת ORDER BY ניתן לשלוף בסדר רצוי כלשהו.

* הסידור מתבצע לפי עמודה כלשהי שאנו בוחרים, כאשר ברירת המחדל בעמודה של מספרים היא סידור מהקטן לגבוה, ובעמודה של מילים סידור לפי סדר האותיות מהראשון לאחרון. לדוגמה:

SELECT id, firstName FROM students ORDER BY lastName

* ניתן להוסיף עוד עמודות לסדר לפיהן, שיהיו שימושיות רק כאשר יש ערכים זהים בסידור של העמודה הראשונה שציינו (סדר פנימי).
* ניתן לשנות את הסדר ולציין סדר עולה מקטן לגבוה (כמו הברירת מחדל) באמצעות פקודת ASC, או סדר יורד מהגבוה לנמוך באמצעות פקודת DESC. לדוגמה:

SELECT gender, age, lastName FROM students ORDER BY gender ASC, age DESC

## הגבלת שליפת שורות LIMIT

אם נרצה להגביל את מספר השורות שאנו שולפים נוכל לעשות זאת באמצעות פקודות LIMIT. בדרך כלל נשתמש כאשר אין לנו תנאי מספק כדי למצוא את השורות שאנו רוצים.

SELECT \* FROM students LIMIT 2

בדרך כלל לא נרצה סתם שורות אלא את השורות לפי סדר כלשהו ולכן בדר"כ נשלב עם ORDER BY:

SELECT \* FROM students ORDER BY firstName LIMIT 2

נוכל גם לקחת מספר שורות לא בהכרח משורה 0 אלא מהשורה החמישית והלאה:

SELECT \* FROM students ORDER BY firstName LIMIT 2, 5

## פונקציות מצטברות - Aggregate Functions

לעיתים אנו לא מעוניינים בערכים ספציפיים אלא על איזשהו חישוב על כל הערכים בשליפה. לשם כך נשתמש בפונקציות אגרגטיביות (מצטברות), המבצעות חישוב על כל הערכים בעמודה. אם ננסה לשלוף ביחד עם אחת הפונקציות עמודה נוספת כלשהי, יבחר רק הערך הראשון מעמודה זו. הפונקציות הן:

* SELECT **COUNT**(\*) FROM table - מחשבת את מספר השורות בטבלה גם אם השורה כולה NULL. לעומת זאת COUNT(col) מחשבת כמה ערכים יש בעמודה ללא NULL.
* SELECT **AVG**(col) FROM table - מחשבת ממוצע באותה עמודה.
* SELECT **SUM**(col) FROM table - מחשבת סכום באותה עמודה.
* SELECT **MAX**(col) FROM table - מחשבת ערך מקסימלי באותה עמודה.
* SELECT **MIN**(col) FROM table - מחשבת ערך מינימלי באותה עמודה.

ניתן לקרוא עוד <https://www.w3schools.com/sql/sql_functions.asp>.

## סידור קבוצות GROUP BY

בסעיף קודם אמרנו שאם נשלוף עמודה כלשהי ואז פונקציה, אזי יבחר רק הערך הראשון מהעמודה. אמנם באמצעות פקודת GROUP BY על עמודה col, אנו בעצם מפעילים את הפונקציה על כל קבוצה זהה מ-col. לדוגמה, אם נרצה לדעת מה הממוצע ציונים לכל קורס בנפרד נשתמש בפקודה:

SELECT courseId, AVG(grades) FROM grades GROUP BY courseId

אם היינו משתמשים בפקודה בלי GROUP BY, היינו מקבלים רק את הערך הראשון מ-courseId ואת ממוצע כל הציונים, בלי חלוקה לקורסים. אמנם בפקודה זו נקבל את כל הקורסים עם ID שונה.

### HAVING

ניתן להוסיף תנאי על ה-GROUP BY כך שלא תתבצע הפונקציה על כל קבוצה בעמודה, באמצעות הפקודה HAVING שלאחריה יבוא תנאי כלשהו. לדוגמה אם נרצה לדעת מהו הציון הממוצע לכל הקורסים שבהם הציון הממוצע מתחת ל-70 נשתמש בפקודה:

SELECT courseId, AVG(grade) FROM grades GROUP BY courseId HAVING AVG(grade)<70

ההבדל בין HAVING ל-WHERE הוא ש-WHERE מטרתו לסנן שורות מתוך השליפה, ואילו HAVING מטרתו לסנן קבוצות מתוך ה-GROUP BY. לכן WHERE יבוא לפני GROUP BY ואילו HAVING לאחריו. ה-WHERE לא יכול להכיל פונקציות אגרגטיביות.

## סדר מימוש הפעולות

כמו בסדר פעולות חשבון במתמטיקה, כאשר יש משפט ארוך ב-SQL יש סדר פעולות כיצד לבצע אותו:

1. קודם מבצעים את השליפות הפנימיות שבתוך הסוגריים.
2. מסתכלים על החלק שלאחר ה-FROM כדי לדעת מאיזו טבלה אנו שולפים.
3. מחשבים את התנאי שלאחר ה-WHERE כדי לדעת באיזה שורות אנו מעוניינים.
4. לאחר מכן על ה-GROUP BY כדי לדעת מהי החלוקה לקבוצות.
5. ואז על ה-HAVING כדי לדעת באיזה קבוצות אנו מעוניינים.
6. ולבסוף על ה-SELECT לבחירת העמודות הרצויות.

SELECT courseId, AVG(grade) FROM (SELECT \* FROM grades) WHERE passed > 0 GROUP BY courseId HAVING AVG(grade) < 70;

## איחוד שליפות UNION

נוכל לאחד שתי שליפות מטבלה אחת על ידי פקודת UNION. התוצאה של איחוד הוא טבלה חלקית כלשהי. נרשום כל שליפה בסוגריים וביניהם את הפקודה UNION:

(SELECT col1, col2, … FROM table WHERE condition1)

UNION

(SELECT col1, col2, … FROM table WHERE condition2)

כדי להשתמש בפקודה זו צריך שמספר העמודות וסוגן יהיה זהה, אך לא בהכרח עם שמות זהים. כשמאחדים בין שתי שליפות שיש בהם שורות זהות אזי האיחוד יכיל רק שורה אחת מתוכן, כלומר לא יהיו כפילויות. אמנם אם רוצים את הכפילויות נשתמש בפקודת UNIONALL.

## צירוף טבלאות JOIN

כאשר נרצה לשלוף שתי טבלאות ולא אחת, הפקודה: SELECT \* FROM table1, table2, תיתן לנו את כל האפשרויות לשלב כל שתי שורות בשתי הטבלאות (כמו מכפלה קרטזית). כך נקבל טבלה עם המון ערכים שלא מתואמים וחוזרים על עצמם. נוכל גם להשתמש בתנאי WHERE כדי לצמצם את האפשרויות, וכך לקבל טבלה יותר רציונאלית. אולם הדרך הנכונה והמקובלת לצרף שתי טבלאות היא באמצעות פקודת JOIN, ולאחריה תנאי שנרשום לאחר המילה השמורה ON. פקודת JOIN על שתי טבלאות בלבד שקולה לשליפה של שתי טבלאות כמו שהראנו לעיל, וכן ON שקול ל-WHERE, אך כמו שאמרנו כדי שמטרת הקוד תהיה ברורה מקובל להשתמש ב-JOIN ו-ON.

לדוגמה, כדי לשלוף רק את השורות שבהם הערך מעמודת id מטבלת student שווה לערך מעמודת studentId מטבלת grades, נשתמש בפקודה:

SELECT \* FROM students INNER JOIN grades ON students.id = grades.studentId

### סוגי פקודות JOIN

בתנאי ON אנו מפעילים תנאי (בדרך כלל שווה =, אך אולי יש עוד) על עמודות משני הטבלאות, כך שמסך כל האפשרויות לחבר בין שתי שורות משתי הטבלאות ניקח רק את השורות שמקיימות את התנאי. אך מה קורה כאשר באחת הטבלאות יש ערכים באחת העמודות שלא מופיעים בעמודה בטבלה השנייה, הרי לא ניתן להפעיל את התנאי? לכן יש שלושה סוגים של פקודות JOIN לשלושה מקרים אפשריים:

1. **INNER JOIN** - בפקודה זו אם באחת העמודות משתי הטבלאות יש ערך שלא מופיע בעמודה של הטבלה השנייה, אזי שורה זו לא תופיע בכלל בשליפה.
2. **LEFT JOIN** - אם בעמודה של הטבלה השמאלית יש ערך שלא מופיע בעמודה של הטבלה הימנית, אזי שורה זו תופיע בכל זאת בשליפה עם ערכי null בעמודות של הטבלה הימנית. אך אם יש ערך בעמודה של הטבלה הימנית שלא מופיע בעמודה של הטבלה השמאלית, השורה לא תופיע בשליפה.
3. **RIGHT JOIN** - כמו LEFT JOIN רק הפוך. כלומר, אם יש ערך בטבלה הימנית שלא מופיע בטבלה השמאלית אזי שורה זו תופיע בכל זאת בשליפה עם ערכי null, אך ההיפך לא תופיע.
4. **OUTER JOIN** - שילוב של LEFT JOIN ו-RIGHT JOIN. אומרת שאם יש ערך באחת מהעמודות בשתי הטבלאות שלא מופיעה בעמודה בטבלה השנייה, אזי שורה זו בכל זאת תופיע בשליפה עם ערכי null בעמודות של הטבלה השנייה. פקודה זו **לא קיימת ב-MySQL** אך ישנם מסדי נתונים שכן קיימת. אמנם גם ב-MySQL ניתן לבצע זאת באמצעות פקודת UNION כך:

(SELECT \* FROM students LEFT JOIN grades ON students.id = grades.studentId)

UNION

(SELECT \* FROM students RIGHT JOIN grades ON students.id = grades.studentId)

### אפשרויות נוספות

ניתן לשלוף יותר משתי טבלאות באופן הבא:

SELECT \* FROM **students** INNER JOIN **grades** on students.id = grades.studentId INNER JOIN **courses** on grades.courseId = courses.id

עד כה הראנו דוגמאות ששולפות מכל טבלה את כל העמודות, אמנם ניתן גם לבחור איזה עמודות אנו רוצים מכל טבלה. לאחר SELECT נרשום ביחד את העמודות שאנו רוצים משתי העמודות.

SELECT id, name, courseId, grade FROM students LEFT JOIN grades ON students.id = grades.studentId)

## הכנסת שורה INSERT INTO

להכנסת שורה לטבלה נשתמש בפקודה:

INSERET INTO table (co1, co2, ...) VALUES (val1, val2, ...)

כאשר הערכים יכנסו לעמודות בהתאמה. צריך להקפיד שמספר הערכים שווים למספר העמודות וגם הסוגים של כל הערכים מתאימים. מילים צריך לרשום עם גרשיים ' ', וניתן להכניס גם ערך NULL בכל עמודה שנרצה. ניתן גם להכניס בסוגריים רק חלק מהעמודות בטבלה. במקרה זה שאר העמודות בשורה החדשה יקבלו ערך NULL או ערך ברירת מחדל אחר שהגדרנו לטבלה.

ניתן גם להכניס שליפה מטבלה נוספת אל הטבלה הנוכחית:

INSERT INTO table (SELECT co1, co2 FROM table2)

## מחיקת שורה DELETE

כדי למחוק שורות בטבלה נשתמש בפקודה: DELETE FROM table WHERE condition, כל השורות המקיימות את התנאי יימחקו. דוגמה למחיקה מטבלת grades:

DELETE FROM grades WHERE studentId=600 OR courseId=20

## עדכון ערכים UPDATE

כדי לעדכן ערכים בטבלה נשתמש בפקודה: UPDATE table SET col= WHERE condition. table כמובן יכול גם להיות שליפה כלשהי ולא טבלה שלימה. ניתן גם לעדכן מספר עמודות במקביל.

UPDATE grades SET grade=78, passed=1 WHERE studentId=111 AND courseId=20

# SQL - מתקדם

## משתנים

בשפת SQL ניתן להגדיר משתנים שיכילו ערך יחיד. נרצה להגדיר משתנים כדי לפשט את הקוד שיהיה יותר מובן וקצר, וגם כדי לשמור ערכים בודדים משליפה כדי להשתמש בה שוב. כדי להגדיר משתנים נשתמש בסימן `@`. ניתן להגדיר משתנים במשפט נפרד באמצעות המילה השמורה SET, או תוך כדי שליפה באמצעות `:=`. לדוגמה:

SET @passGrade = 60

SELECT @avgGrade := AVG(grade) FROM grades

נוכל לשלוף את הערך בכל משתנה באמצעות: SELECT @avgGrade

## טבלאות זמניות TEMPORARY TABLE

משתנים אינם יכולים לשמור טבלאות אלא רק ערכים בודדים. לכן כאשר נרצה לבצע שליפה מטבלה, ולאחר מכן לשמור את השליפה כדי להשתמש בה בהמשך התוכנית, ניצור טבלה זמנית. הטבלה הזמנית אינה נשמרת במסד הנתונים אלא נמחקת לאחר ריצת התוכנית.

CREATE TEMPORARY TABLE tempTable (col1 DATATYPE, col2 DATATYPE, …)

INSERT INTO tempTable (SELECT col1, col2 FROM table)

ניתן גם לקצר על ידי פקודה אחת בדרך הבאה:

CREATE TEMPORARY TABLE tempTable AS (SELECT col1, col2 FROM table)

## כינויים ALIASES

ב-SQL יש אפשרות לתת כינויים לעמודות ושליפות שביצענו. נתינת הכינוי תיעשה באמצעות המילה השמורה AS. נשתמש בכינויים לצרכים הבאים:

1. לתת כינוי לעמודה שביצענו עליה פעולות אריתמטיות בשליפה, כך שבתצוגה של השליפה במקום ששם העמודה יהיה השם המלא שם הפעולה (col\*2 לדוגמה), נוכל לבחור מה השם הרצוי.

SELECT col\*2 FROM table **AS new\_name**

1. לתת כינוי לשליפות כדי שנוכל בהמשך הפקודה להשתמש בהם שוב. הרבה פעמים זו תהיה גם הדרך היחידה. לשליפות אין צורך להשתמש ב-AS אלא אפשר ישר לרשום את הכינוי.

SELECT \* FROM students **s** INNER JOIN grades **g** ON **s.id = g.studentId**

SELECT \* FROM (SELECT id, age FROM students) **AS s** INNER JOIN grades **AS g** ON s.id = g.studentId

1. כאשר יש לנו שאילתה מקוננת שבתוכה השתמשנו ב-ALIAS, אזי צריך לתת ALIAS גם לכל השאילתה הפנימית אחרת נקבל שגיאה.

SELECT MAX(av) FROM (SELECT studentId, AVG(grade) **AS av** FROM grades GROUP BY studentId) **AS t**

## Transactions

טרנזקציה היא רצף של שאילתות SQL שמבוצעות כולן כיחידה אחת או לא מבוצעות כלל. באמצעות טרנזקציות אנו יכולים להבטיח שאנו שומרים על עקרונות ה-ACID ובכך להבטיח שאין שגיאות לא צפויות במסד הנתונים. השאילתות אותן ניתן להכניס לטרנזקציה הן כאלו שמשנות את המידע במסד הנתונים כמו: INSERT, UPDATE ו-DELETE.

לדוגמה, עבור רצף של שאילתות SQL שמבצעות העברה בנקאית מחשבון אחד לחשבון אחר, אם תהיה שגיאה באמצע הרצת השאילתות יכול להיווצר מצב שבו נמשך הכסף מחשבון אחד אך לא הועבר לאחר. בעיה נוספת אם באותו זמן ירוץ עוד תהליך שמושך כסף מהחשבון וידרוס את השינוי שעשינו. הפתרון לבעיות אלו הוא להצהיר על רצף שאילתות אלו כטרנזקציה, ובכך יבוצעו כולם יחד או לא יבוצעו בכלל.

SET @transferAmount = 1000;

START TRANSACTION;

SELECT @firstBalance = amount FROM bankBalances WHERE userId = 777;

UPDATE bankBalances SET amount = @firstBalance - @transferAmount WHERE userId = 777;

SELECT @secondBalance = amount FROM bankBalances WHERE userId = 888;

UPDATE bankBalances SET amount = @secondBalance + @transferAmount WHERE userId = 888;

COMMIT;

### פקודות

**START TRANSACTION** - מצהיר על תחילת רצף הפקודות של הטרנזקציה.

**COMMIT** - מבצע את כל הפקודות מתחילת הטרנזקציה כיחידה אחת ומשנה את מסד הנתונים בהתאם.

**ROLLBACK** - אומר לבטל את כל השאילתות שנכתבו מאז ה-COMMIT האחרון ועד ה-ROLLBACK. מופעל בדרך כלל שהייתה שגיאה באחת השאילתות.

**SAVEPOINT** - נקודה המאפשרת ל-ROLLBACK לא לחזור עד ה-COMMIT האחרון אלא רק עד הנקודת שמירה האחרונה.

SAVEPOINT SAVEPOINT\_NAME;

...

ROLLBACK TO SAVEPOINT\_NAME;

**RELEASE SAVEPOINT** - משחררת נקודת שמירה.

RELEASE SAVEPOINT SAVEPOINT\_NAME;

ב-MySQL Workbench יש שני כפתורים רלוונטיים לטרנזקציות: האחד אומר שגם אם יש שגיאה בשאילתה אזי תמשיך לשאילתה הבאה. מומלץ בדר"כ שכפתור זה יהיה כבוי. הכפתור השני אומר לבצע COMMIT אוטומטי לאחר כל שאילתה. מומלץ להשאיר כפתור זה דלוק.

## Stored Procedure

הוא קוד SQL מוכן השמור במסד הנתונים עצמו, כך שניתן יהיה להשתמש בקוד זה שוב ושוב. אם יש שאילתות SQL שאנו משתמשים בו פעמים רבות, נשמור אותם כ-Stored Procedure, ואז פשוט ניתן לקרוא להם מתי שנרצה ולהפעיל אותם. יש אפשרות גם להעביר פרמטרים ל- Stored Procedureכך שיוכל לפעול על סמך ערכי הפרמטרים המועברים. היתרונות בשימוש ב-Stored Procedure הם:

1. מאפשר לנו להשתמש בקוד קיים במקום לכתוב אותו מחדש ובכך חוסך זמן.
2. ניתן להפעיל את הקוד מכל מקום שנרצה בגישה למסד הנתונים, ואף משפות שונות, ובכך חוסך לנו את ההתאמות הנדרשות לכל שפה.
3. חוסך זמן תקשורת, שהרי הקוד מתבצע בבת אחת על המסד נתונים ולא צריך להחזיר כל שאילתה תשובה לשרת.
4. מאפשר לבצע שינויים בקוד השמור בלי לאתחל כל פעם את השרת מחדש.

### ללא פרמטרים

כדי להצהיר על Stored Procedure נרשום CREATE PROCEDURE, ואז את שם ה-Stored Procedure. בפנים, לפני כל השאילתות שנרצה לשמור, נרשום BEGIN ולבסוף נרשום END.

לפני ההצהרה של CREATE נצטרך לשנות את ה-Delimiter שהוא הסימן המפריד בין שאילתות. הסיבה לכך היא ש-SQL צריך לדעת מתי נגמר ה-Stored Procedure, ולשם כך צריך להכניס לו Delimiter לאחר END. אמנם אם נשתמש ב-Delimiter הברירת מחדל שהוא `;` ולא נשנה אותו, כבר לאחר השאילתה הראשונה שיבוא אחריה `;` הוא יחשוב שזהו סוף ה-Stored Procedure. לכן עם שינוי ה-Delimiter נוכל לשלב ביניהם כך שלאחר כל שאילתה נרשום `;`, ולאחר END נרשום את ה-Delimiter שקבענו. ה-Delimiter יכול להיות כל דבר שנרצה, בדוגמה בחרנו `$$`. בסוף התהליך נשנה את ה-Delimiter בחזרה ל-`;`.

DELIMITER $$

CREATE PROCEDURE procedure\_name  
BEGIN  
sql\_statements;

END$$

DELIMITER ;

כדי לקרוא ל-Stored Procedure שיצרנו כדי שיבצע את השליפות ששמנו בתוכו נרשום: CALL *procedure\_name*;

נוכל למחוק Stored Procedure באמצעות הפקודה: DROP PROCEDURE procedure\_name. אין פקודה לשינוי הקוד השמור לאחר שנוצר, לכן צריך למחוק וליצור מחדש.

### עם פרמטרים

כאמור, נוכל גם לשלוח פרמטרים ל-Stored Procedure כך שהשליפה תתבצע בהתאם לפרמטר שהכנסנו. בנוסף ניתן גם להחזיר משתנים מ-Stored Procedure עליהם עשינו חישוב פנימי כלשהו. כדי לעשות זאת בשורת ההצהרה של CREATE PROCEDURE נוסיף בסוגריים את הפרמטרים. כל פרמטר יתחיל ב-IN אם זהו פרמטר שמתקבל כקלט, או ב-OUT אם זהו פרמטר המוחזר. לאחר מכן שם הפרמטר ולבסוף הטיפוס שלו. במקרה של פרמטר מוחזר נצטרך לבצע שאילתה של INSRT INTO כדי להכניס ערך למשתנה. לדוגמה, Stored Procedure שמקבל כקלט id של סטודנט ומחזיר ממוצע וציון מקסימלי.

DELIMITER $$

CREATE PROCEDURE `student\_info`(IN Id INT, OUT avg\_g REAL, OUT max\_g INT)

BEGIN

SELECT AVG(grade) INTO avg\_g FROM grades WHERE studentId = Id;

SELECT MAX(grade) INTO max\_g FROM grades WHERE StudentId = Id;

END$$

DELIMITER ;

כדי לקרוא ל-Stored Procedure, לקבל ממנו את שני הערכים ואז לשלוף אותם נרשום:

CALL student\_avg\_21(222, @avg\_grade, @max\_grade);

SELECT @avg\_grade, @max\_grade;

### תפעול ב-Workbench

אפשר בקלות ליצור, לשנות ולמחוק Stored Procedure דרך ה-Workbench באמצעות מקש ימני על תווית Stored Procedure. אם יוצרים דרך ה-Workbench אין צורך להוסיף DELIMITER בהתחלה וסוף, הם יתווספו אוטומטית. אפשרות השינוי ב-Workbench מתבצעת מאחורי הקלעים דרך מחיקה ויצירה מחדש.

### שגיאות וטרנזקציות ב-Stored Procedure

נוכל להגדיר בתוך הקוד השמור טרנזקציות כמו שלמדנו, ובנוסף להם להגדיר handler שיתמודד עם סוגי שגיאות. בדרך כלל נרצה להפעיל בתוך מיני handler כאלו את הפקודה ROLLBACK והודעה על השגיאה. דוגמה במצגת 3 שקופית 19.

## Triggers

הם כמו Stored Procedure, כלומר קוד השמור במסד הנתונים, אך להבדיל מ-Stored Procedure אנו לא קוראים להם אלא כבר שיוצרים אותם מגדירים להם במדויק מתי הקוד יתחיל לפעול בצורה אוטומטית. המוטיבציה להגדיר טריגרים היא במקרים בהם יש איזו פעולה שנרצה לבצע כל פעם שמבצעים איזושהי פעולה על טבלה.

ההצהרה מאוד דומה לזו של Stored Procedure. יש לשנות את ה-Delimiter, לתת שם לטריגר, להתחיל את השאילתות עם BEGIN ולסיים ב-END. אלא שבנוסף צריך להגדיר מתי הטריגר יופעל: עבור איזה שינוי (הכנסה, שינוי או מחיקה) באיזו טבלה, האם לפני או אחרי השינוי, והאם להפעיל את הטריגר פעם אחת או עבור כל שורה בטבלה בנפרד.

צריך לבחור **רק ערך אחד** מכל סוגריים מרובעות, אם רוצים לבחור יותר יש לכתוב מספר טריגרים כל אחד עם אפשרות אחרת. במקרה של הוספת שורה חדשה, נתייחס אליה כאל טבלה בפני עצמה שהשם שלה הוא NEW (ראה דוגמה למטה).

DELIMITER $$

CREATE TREAGER *trigger\_name*

[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON *table\_name*

[FOR EACH ROW] //optional

BEGIN

sql\_statements;

END$$

DELIMITER ;

כדי למחוק טריגר נשתמש בפקודה: DROP TRIGGER *trigger\_name*;

לדוגמה, אם יש טבלה של סטודנטים ואנו רוצים שכל פעם שיוכנס סטודנט חדש מיד תופעל שאילתה שתעדכן את הממוצע שלו לפי הנתונים בטבלה הציונים, נגדיר לכך את הטריגר הבא:

DELIMITER $$

CREATE TRIGGER new\_grade\_received

AFTER INSERT ON grades

FOR EACH ROW

BEGIN

UPDATE students SET avg\_grade = (SELECT AVG(grade) FROM grades WHERE studentId=**NEW**.studentId)

where id = **NEW**.studentId;

END$$

DELIMITER ;

## Views

הן סוג של טבלאות וירטואליות שאינן נשמרות במסד הנתונים. טבלאות מסוג views נוצרות באמצעות שליפה מטבלה אחת או יותר שקיימות במסד נתונים. יש בהן שורות ועמודות כפי שיש בטבלה אמיתית במסד הנתונים, ואף ניתן לשלוף ולהציג את המידע באמצעות WHERE ו-JOIN כאילו הנתונים מגיעים מטבלה אחת יחידה. בכל פעם שמבצעים שליפה מ-view, הפקודה המלאה שבאמצעותה יצרנו את ה-view מקוננת לתוך השליפה הנוכחית, וכך נוצרת האשליה שה-view היא טבלה אמיתית.

הסיבות שנרצה להשתמש ב-views:

* לפשט שאילתות מורכבות - במקרה שניגשים למידע שהוא עצמו חישוב מורכב, השאילתה תהיה הרבה יותר פשוטה אם נגדיר את המידע הזה כ-view.
* הגבלת גישה למידע - במקרה שיש טבלה שחלקה מכיל מידע רגיש שלא נרצה שייחשף למשתמשים מסוימים, אך כן נרצה שיוכלו לגשת למידע הלא רגיש, נוכל ליצור view של טבלה זו שיכיל רק את מידע הלא רגיש ונמנע את הגישה לטבלה המקורית.
* עדכון אוטומטי - כיוון שה-view נוצר בכל פעם שקוראים לו, הוא כל הזמן מעודכן במידע של הטבלאות מהן נוצר. זה עובד גם בכיוון ההפוך, כאשר מעדכנים את ה-view גם הטבלאות המקוריות מתעדכנות. ישנם כללים מתי כיצד אפשר לעדכן את ה-view.

כדי ליצור view יש להצהיר על כך באמצעות הפקודה CREATE VIEW, לאחריה ניתן שם ל-view, נרשום AS, ולבסוף את השליפה שה-view תכיל. לדוגמה, כדי ליצור view המכילה את השם והממוצע של תלמיד:

CREATE VIEW avg\_grades\_view AS

SELECT students.firstName, AVG(grade) AS average FROM grades INNER JOIN students

ON grades.studentId = students.id GROUP BY studentId;

לאחר יצירת ה-view פעולות השליפה, עדכון, הכנסת ומחיקת שורה, זהות במדויק לפקודות שלמדנו על טבלאות רגילות. כדי לעדכן את ה-view צריכים להתקיים כל חמשת התנאים הבאים:

1. ה-view יוצרה מטבלה יחידה. אם ה-view נוצרה מטבלאות מרובות, לא ניתן לנו לעדכן את התצוגה.
2. ה-view לא יוצרה באמצעות שאילתות מקוננות או שאילתות מורכבות.
3. הצהרת SELECT המשמשת ליצירת ה-view לא תכלול GROUP BY או ORDER BY.
4. להצהרת ה-SELECT לא צריכה להיות מילת המפתח DISTINCT.
5. ה-view צריכה שכל ערכיה יהיו NOT NULL.

במקרה שקיים view עם שם מסוים, ונרצה ששם זה מעתה יכיל טבלה וירטואלית אחרת, נוכל להשתמש בפקודת CREATE OR REPLACE VIEW כדי לעשות זאת. השימוש בפקודה זו זהה לצורה שבה הגדרנו את CREATE VIEW. כדי למחוק view נשתמש בפקודת DROP VIEW ולאחריה שם ה-view שנרצה למחוק.

# Normalization

## הקדמה

נורמליזציה של מסד נתונים היא שיטה המתארת תהליך לבניית מסד נתונים רלציוני בהתאם לסדרה של צורות נורמליות, המתארות רמות של נרמול. לכל צורת נרמול ישנם כללים כיצד ליישם אותה במסד נתונים. כללים אלו ניתן ליישם או כאשר בונים מסד נתונים חדש או כאשר משפרים מסד נתונים קיים. המטרות של נרמול הן:

1. להפחית נתונים מיותרים.
2. לארגן את מסד הנתונים כך שיחזיק מידע על נושא אחד בלבד.
3. לפשט את פעולות השליפה והעדכון במסד נתונים.

לפני שנתאר את דרגות הנרמול, נגדיר מספר הגדרות שנשתמש בהן בהמשך.

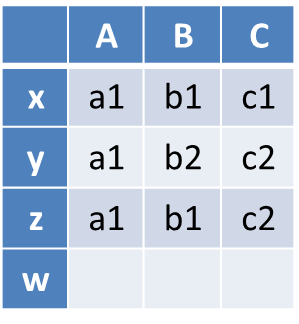
## הגדרות

### תלות (Dependency)

נאמר כי תכונה (כל תכונה מיוצגת בעמודה) או קבוצת תכונות, המסומנות ב-B, הן תלויות בקבוצת תכונות המסומנות ב-A, אם באמצעות הערכים ב-A ניתן לדעת בדיוק את הערכים ב-B. חייב להיות שמ-A ניתן להסיק אך ורק ערך אחד לכל תכונה ב-B. אם תנאי זה מתקיים נאמר כי B תלוי ב-A, ונסמן זאת .

לדוגמה, אם A היא ת"ז של סטודנט ו-B היא שם פרטי של סטודנט, נאמר כי B תלוי ב-A, כיוון שמת"ז ניתן לדעת מה השם הפרטי. ההיפך אינו נכון, שכן יכול להיות מספר ת"ז המתאימות לשם פרטי אחד.

### תלות מרובה (Multivalued Dependency)

בהינתן שתי קבוצות של תכונות, A ו-B, נסמן ב-C את כל שאר התכונות שאינן ב-A או B, . נאמר כי ל-B יש תלות מרובה ב-A, בסימון *, אם בהינתן שורות x ו-y בטבלה כך שמתקיים:*

* *– התא בשורה x בעמודה A שווה לתא בשורה y בעמודה A.*

*אזי קיימת שורה z שבה מתקיים:*

*תלות זו בעצם אומרת שיש בטבלה שורה אחת או יותר, המרמזות על נוכחות של שורה אחת או יותר אחרות באותה טבלה. כלומר, מהתבוננות בשורות מסוימות ניתן להסיק את נוכחותן של שורות אחרות.*

### Candidate keys

קבוצה מינימלית של תכונות, שכל שאר התכונות בטבלה תלויות בהן, נקראת Candidate keys (או (Minimal Super-Key. הכוונה במינימלית היא שאם נוריד תכונה אחת מקבוצה זו, תהיה לפחות תכונה אחת שאינה תלויה ב-Candidate keys. יכולות להיות מספר קבוצות כאלו בגדלים שונים. כל התכונות שהן חלק מאיזשהו Candidate key מכונות **prime**, וכל שאר התכונות שהן לא חלק מאף Candidate key מכונות **non-prime**.

לדוגמה, עבור טבלה שהתכונות שלה הן: שם אוניברסיטה, שם מחלקה, ושם ראש מחלקה, אזי תחת ההנחה שלכל מחלקה יש ראש מחלקה אחד, נוכל לבחור את ה- Candidate keysלהיות אחת משתי האפשרויות הבאות:

* ראש המחלקה.
* שם האוניברסיטה והמחלקה.

נשים לב שעבור כל Candidate key התכונות שנותרות אכן תלויות בה. עוד נשים לב שכל התכונות הן prime.

### Super key

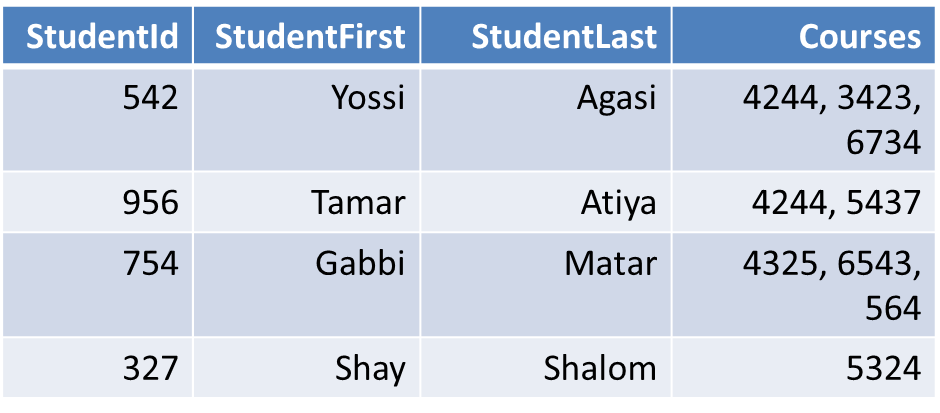
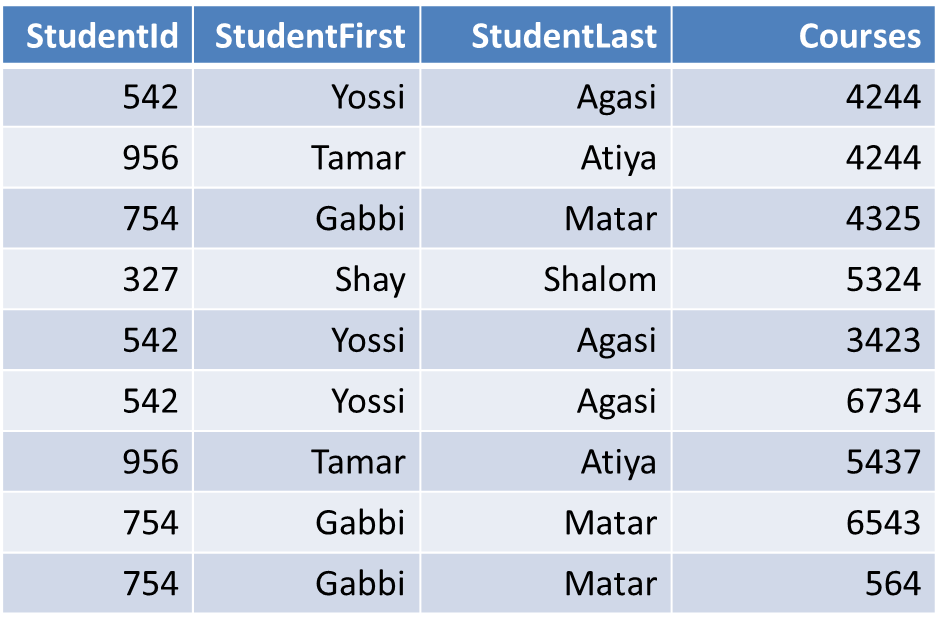
קבוצת של תכונות, שכל שאר התכונות בטבלה תלויות בהן, נקראת Super key. קבוצות מסוג זה מוגדרות כמו Candidate keys רק בלי המינימליות. נשים לב שכל super key מכיל בתוכו איזשהו Candidate key.

## Normalization

נציג מספר צורות של רמות נרמול ואת הכללים הנדרשים כדי לבצע אותם. כל רמת נרמול כוללת את כל הרמות הקודמות לה ומוסיפה עליהן.

### 1NF

בדרגת נרמול זו כל תא בטבלה של מסד הנתונים מכיל ערך אחד. במידה ובמסד הנתונים יש תא שמכיל יותר מערך אחד, נוסיף שורות לטבלה, כמספר הערכים בתא זה, כך שבכל שורה יהיה ערך אחר מהתא. הרעיון מאחורי נרמול זה הוא שכך יהיה יותר קל לחפש את כל השורות המשותפות לערך כלשהו.

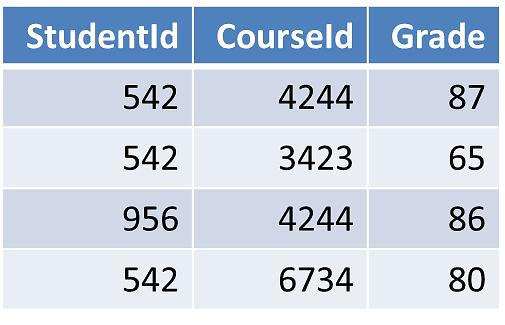
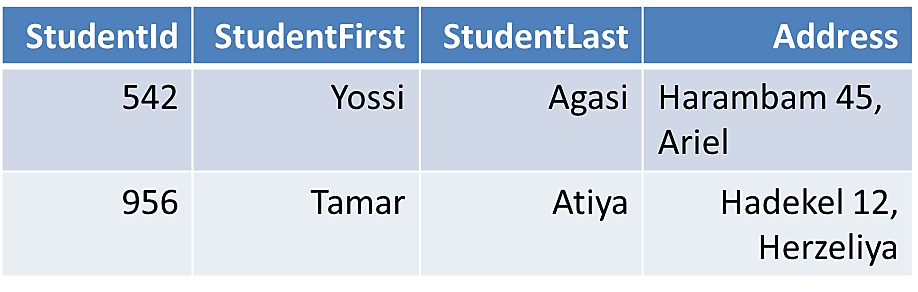
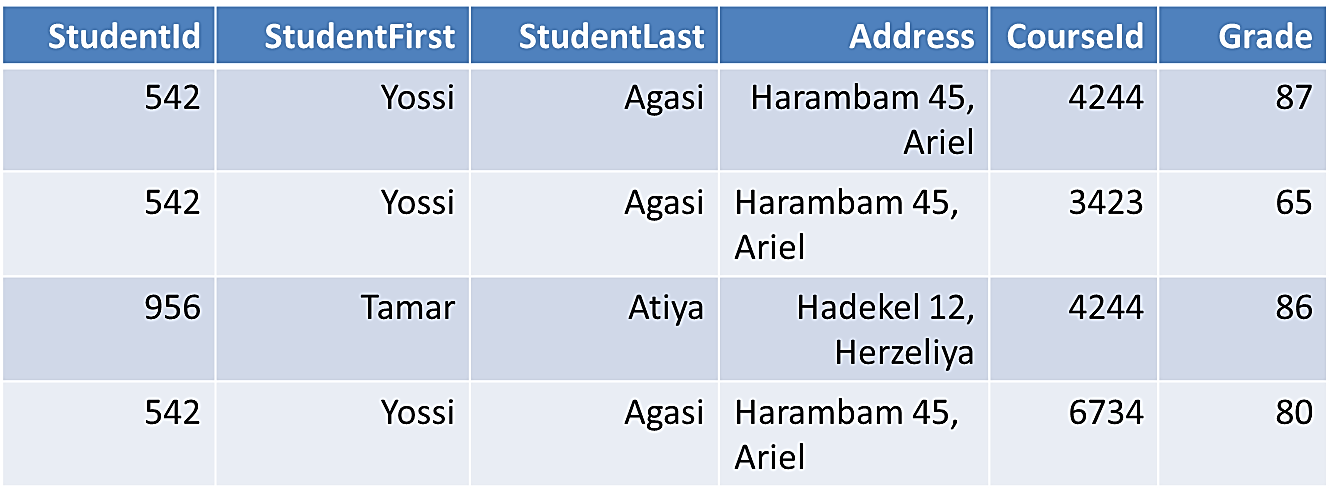


לדוגמה, טבלה שבה עמודה המכילה את כל הקורסים שסטודנט עושה נפצל למספר שורות כך שכל שורה היא קורס אחר.

### 2NF

הטבלה חייבת להיות 1NF ובנוסף כל תכונה שהיא non-prime אינן תלויה בקבוצה חלקית ממש של ה-candidate key, אלא תלויה ב-candidate key כולו. כדי לגלות אם טבלה היא 2NF, יש לעבור על כל התכונות שהן non-prime ולבדוק באיזה תכונות הן תלויות. מספיקה תכונה אחת שלא מקיימת תנאי זה כדי שהטבלה לא תהיה 2NF. הפתרון במצב זה הוא לפצל את הטבלה לשתי טבלאות, כך שכל תכונה תהיה בטבלה שבה נמצאת הקבוצה החלקית של ה-candidate key שהיא תלויה בה.

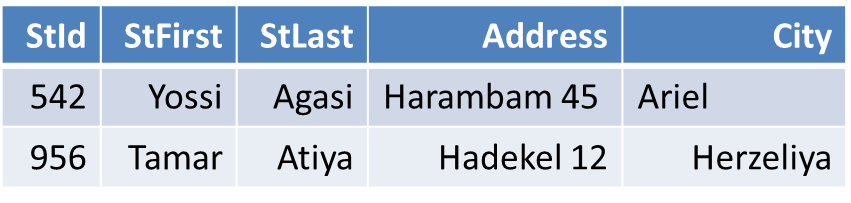
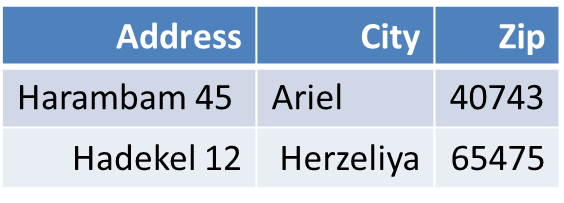
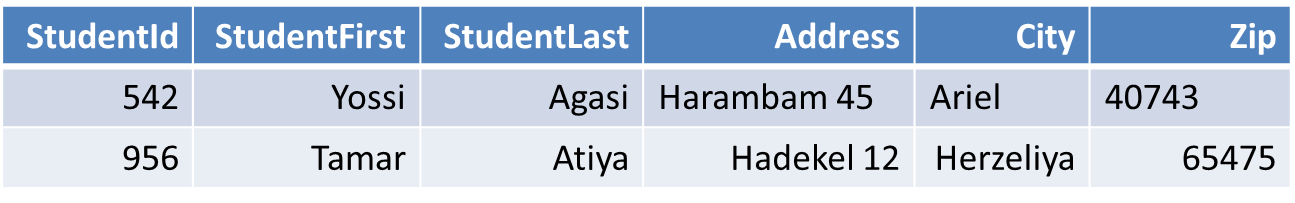
לדוגמה, בטבלה שבה כל רשומה היא קורס אחר של סטודנט, ה-candidate key הם ה-ID של הסטודנט וה-ID של הקורס. שם פרטי, משפחה וכתובת תלויים ב-ID של הסטודנט, ואילו הציון תלוי ב-ID של הקורס, לכן נפצל את הטבלה כך:



### 3NF

הטבלה חייבת להיות 2NF, ובנוסף כל התכונות שהם non-prime תלויות ב-super key, כלומר בקבוצת תכונות המכילה איזשהו candidate key. כדי לגלות אם טבלה היא 3NF, יש לעבור על כל התכונות שהם non-prime ולבדוק באיזה תכונות הן תלויות. מספיקה תכונה אחת שלא מקיימת תנאי זה כדי שהטבלה לא תהיה 3NF. הפתרון במצב זה הוא לפצל את הטבלה לשתי טבלאות, כך שהתכונה שאינה מקיימת את התנאי תהיה עם התכונות בהן היא תלויה.

לדוגמה, בטבלה שבה כל רשומה מכילה פרטים על סטודנט, וה-candidate key הוא ה-ID של הסטודנט. המיקוד יכול להיות תלוי ב-ID של הסטודנט, אך גם יכול להיות תלוי בכתובת והעיר בה הוא גר, לכן אינה 3NF. נפצל את הטבלה כך:

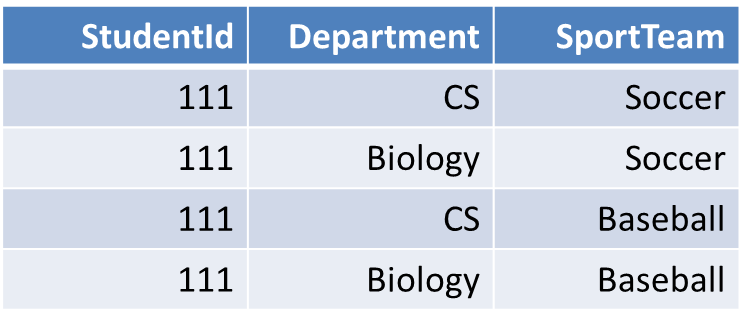


### BCNF (3.5NF)

כדי שהטבלה תהיה בצורת BCNF (Boyce and Codd Normal Form, נקרא גם 3.5NF), היא חייבת להיות 3NF ובנוסף כל התכונות שהם prime תלויות ב-super key, כלומר בקבוצת תכונות המכילה איזשהו candidate key. נשים לב שהשינוי היחיד מ-3NF הוא שגם תכונות prime צריכות להיות תלויות ב-super key, ולא רק תכונות non-prime. כדי לגלות אם טבלה היא BCNF, יש לעבור על כל התכונות ולבדוק באיזה תכונות הן תלויות. מספיקה תכונה אחת שלא מקיימת תנאי זה כדי שהטבלה לא תהיה BCNF.

### 4NF

הטבלה חייבת להיות BCNF, ובנוסף אין בטבלה אף תלות מרובה (multivalued dependencies), כלומר אין שורה שממנה נוכל להסיק שורה אחרת בטבלה. במידה ואכן יש תלות מרובה בין שני תכונות, נפצל את הטבלה ל-2 טבלאות כך שכל קבוצה בתלות המרובה תהיה בטבלה נפרדת. המטרה בנרמול זה היא לגרום לכך שכל טבלה תחזיק מידע מסוג אחד ולא לערבב בין נתונים שאינם קשורים באותה הטבלה.



לדוגמה, טבלה שבה כל שורה מייצגת סטודנט. מכילה ID של הסטודנט, את המחלקה בה נמצא, ואת קבוצת הספורט בה משתתף. במידה והסטודנט משתתף במספר קבוצות ספורט, נצטרך שורה נפרדת לכל קבוצה כזו. נמצא כי שם המחלקה לה משתייך יוכפל לחינם בכל שורה כזו. המסקנה היא שישנה תלות מרובה בין קבוצת הספורט ל-ID (studentId ->> sportTeam), ולכן טבלה זו אינה 4NF. הפתרון הוא לפצל את הטבלה לשני טבלאות: אחת תכיל קבוצות ספורט והשנייה את שם המחלקה.

### 5NF

ברמת נרמול זו מיישמים כל מיני חוקים על הרשומות שבאמצעותם נוכל להסיק שיש רשומות שתלויות ברשומות אחרות. נוכל לצמצם את הרשומות התלויות ובכך להקטין את גודל המידע השמור במסד נתונים. כיוון שכאשר מסד נתונים מנורמל בצורת 4NF הוא בדרך כלל גם יהיה מנורמל ברמת 5NF, למעט מקרים מאוד נדירים שלא נגיע אליהם, לא נפרט על רמת נרמול זו.

# Connectors

## JDBC

JDBC (**J**ava **D**ata**B**ase **C**onnectivity) זוהי ספרייה ב-Java באמצעותה ניתן, דרך קוד Java, להתחבר למסד נתונים ולהפעיל עליו פקודות SQL. כדי שנוכל לעשות זאת יש לבצע מספר שלבים:

1. להתחבר למסד הנתונים. לשם כך צריך ליצור אובייקט מסוג Connection המייצג את החיבור אל מסד הנתונים, באמצעות שיטת getConnection, המקבלת את מיקום מסד הנתונים, שם משתמש וסיסמה. חיבור זה צריך להיסגר בסוף. הדרך הכי נוחה היא פשוט להצהיר על החיבור באמצעות try with resources.
2. ליצור הצהרת SQL. יש שלושה סוגי אובייקטים לשם כך: Statement, PreparedStatement, CallableStatement. אובייקט Statement מיועד לפקודות רגילות, והשניים האחרים משמשים כדי להפעיל Stored Procedure במסד נתונים המקבלים פרמטרים.
3. ביצוע ההצהרת SQL שהכנו. יש שלושה סוגי פקודות לשם כך: executeQuery(), executeUpdate(), execute(). Execute משמש להרצת כל סוג הצהרת SQL, executeQuery משמש כדי להחזיר מידע ממסד הנתונים, ו-executeUpddate משמש כדי לעדכן או לשנות את מסד הנתונים.

כאשר מחזירים מידע ממסד הנתונים, הערך המוחזר ישמר באובייקט מסוג ResultSet ששומר את הערך המוחזר. ערך זה מזכיר איטרטור בכך שהוא מצביע כל פעם לשורה (רשומה) אחרת בערך שהוחזר, וכדי לגרום לו להצביע לשורה הבאה צריך להפעיל עליו שיטת next. השורה הראשונה בערך שהוחזר ממוספרת בתור 1. ה-ResultSet בהתחלה מצביע לשורה 0, לכן כדי שיצביע על השורה הראשונה, יש להפעיל עליו שיטת next. כאשר עבר על כל השורות מחזיר null. על ה-ResultSet ניתן להפעיל שיטות נוספות כדי לקבל מטה-דאטה על הערך שהוחזר מהשליפה.

דוגמה לשליפת טבלה שלמה והדפסת כל השורות בטבלה:

import java.sql.\*;

public class Main{

public static void main(String[] args){

try{

Class.*forName*("com.mysql.jdbc.Driver");

try(Connection con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test", "user", "pwd")){

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("*SELECT \* FROM students*");

int numOfColumns = rs.getMetaData().getColumnCount();

while (rs.next()){

for (int col = 1; col <= numOfColumns; col++){

System.*out*.print(rs.getString(col) + " ");

}

System.*out*.println();

}

}} catch (Exception ex){ex.printStackTrace();}

}

}

להשלים: לעבור על כל סוגי האובייקטים, למה משמשים והשיטות עליהם.

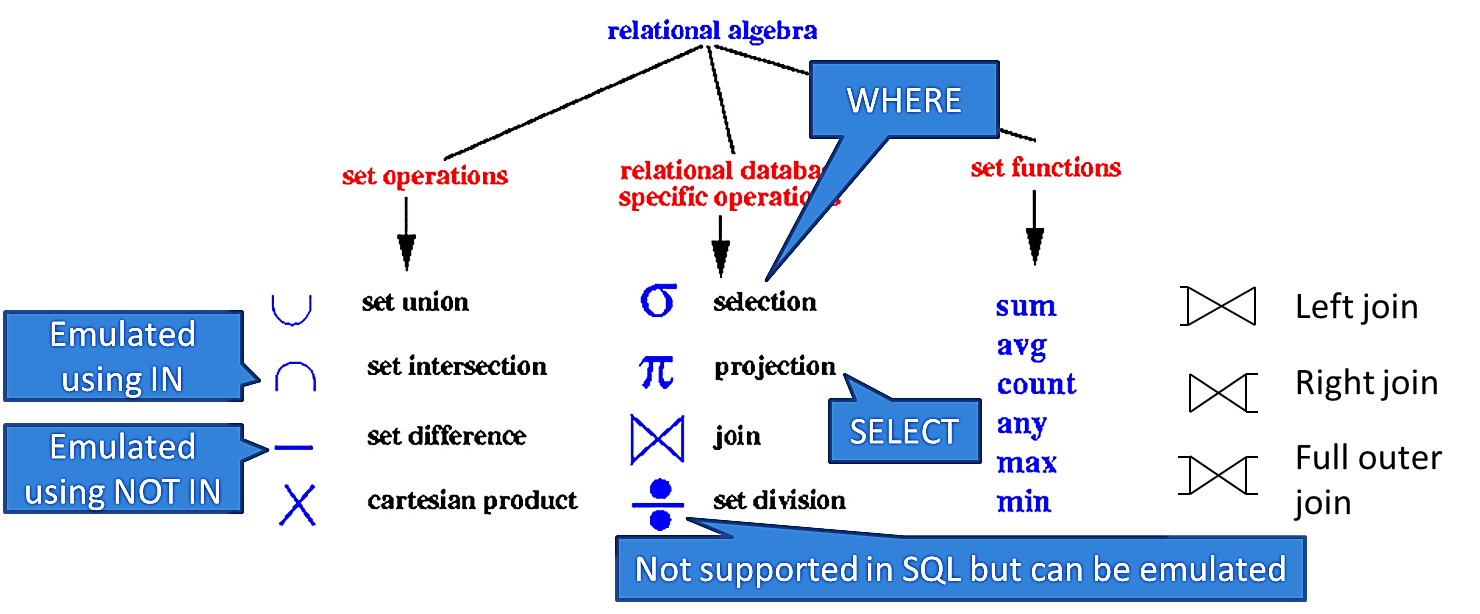
<https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm>

# אלגברה רלציונית

## הגדרה

אלגברה רלציונית היא שפה מתמטית מאוד דומה לשפה שלמדנו של SQL, אך עם שמות שונים. בשפה זו לכל פעולה יש סימון.

## סימונים

הסימונים הנפוצים הם: