# NoSQL

## הגדרה

NoSQL הן קבוצות של מערכות לניהול מסדי נתונים שאינן מסתמכות בלבד על אחסון הנתונים בטבלאות (רלציות) כמו מסדי נתונים רלציוניים. מקור השם הוא Non SQL, אך נקרא לעיתים Not Only SQL, כדי להדגיש את העובדה שהן עשויות לתמוך בשאילתות הדומות ל-SQL.

מסדי נתונים מסוג NoSQL יותר גמישים בשמירת עקרונות ה-ACID, ביחס למסדי נתונים רלציוניים. הם מתוכננים עבור מערכות מבוזרות שבהן המידע מתפרש על כמה שרתים, ומשמשים בעיקר למערכות של ביג-דאטה או מערכות זמן אמת.

## משפט CAP

משפט CAP מציין כי קיימות שלוש דרישות בסיסיות בעת תכנון יישומים למערכת מחשבים מבוזרת.

* **C**onsistency - כל המשתמשים מקבלים את המידע הכי מעודכן ממסד הנתונים. כל המשתמשים רואים את אותו המידע, ולכן כל פעולה על מסד הנתונים היא על הנתונים העדכניים ביותר.
* **A**vailability - כל בקשה מקבלת תגובה (לא בהכרח העדכנית ביותר).
* **P**artition tolerance - פירוש הדבר שהמערכת ממשיכה לתפקד אפילו כשהתקשורת בין השרתים אינה אמינה, כלומר השרתים עשויים להיות מחולקים למספר קבוצות שאינן יכולות לתקשר זו עם זו.

משפט CAP מוכיח שאי אפשר למלא את כל שלושת הדרישות אלא לכל היותר 2. לכן כל מסד נתונים של NoSQL עוקב אחר השילובים השונים של C, A, P מתוך משפט CAP. מסדי נתונים רבים ב-NoSQL בוחרים להתפשר על העקביות ולהרוויח זמינות וסובלנות לחלוקה. בכך הם לא שומרים על עקרונות ה-ACID, ומציעים במקום את עקרונות BASE:

* **B**asically **A**vailable - הנתונים זמינים לרוב.
* soft **S**tate - המצב עשוי להשתנות אפילו ללא עדכונים (מכיוון שעדכונים ישנים עדיין מתפשטים).
* **E**ventual consistency - אם נאפשר לנתונים להתפשט מספיק זמן, הם לבסוף יהפכו עקביים.

## יתרונות וחסרונות

### יתרונות

1. Scalability - ניתן בקלות להרחיב את כמות המידע המאוחסן. יש שני דרכים לעשות זאת: על ידי הוספת משאבים לשרתים הקיימים (Vertical scaling) או על ידי הוספת שרתים (horizontal scaling).
2. Availability - המידע במסדי נתונים של NoSQL משוכפל למספר שרתים, כך שאם שרת אחד או יותר נופלים נוכל לשחזר את המידע משרת אחר. תכונה זו הופכת אותו לזמין ביותר, מכיוון שבכמעט כל מצב נוכל לגשת למידע.
3. Speed - למערכות מבוזרות יש יותר כוח חישוב מאחר וכל שרת יכול לבצע חישובים בפני עצמו - חישוב מקבילי.
4. Flexibility - ניתן להוסיף תכונות מהר תוך כדי ריצה. לא מקפידים על עקביות.

### חסרונות

1. Narrow focus - מיועד בעיקר לאחסון ולכן מספק מעט מאוד פונקציונליות.
2. No standardization - אין סטנדרט אחיד למסדי נתונים של NoSQL.
3. Management challenge - ניהול נתונים ב-NoSQL הרבה יותר מורכב ממסד נתונים רלציוני. יש לו מוניטין שהוא מאתגר להתקנה ואפילו קדחתני יותר לניהול על בסיס יומי.
4. GUI - לרוב מסדי הנתונים מסוג NoSQL אין GUI נוח ופשוט.

## סוגים

מבני הנתונים בהם משתמשים מסדי הנתונים השונים ב-NoSQL שונים אחד מהשני, אמנם ניתן לחלק אותם למספר קטגוריות. הבחירה של מסד נתונים תלוי בבעיה שאותה אנו רוצים לפתור.

1. Key-Value Stores - מסד נתונים פשוט ומהיר, המאחסן נתונים בצורה של מפתח-ערך. עובד כמו Hashtable ענק. דוגמה: Redis (הנפוץ ביותר), ‏Riak, ‏Voldemort.
2. Wide-Column Stores - בסיס נתונים מבוססי עמודות (במקום שורות), בו משמשים בעיקר לשאילתות על מערכי מידע גדולים במיוחד הניתנים לשינוי מהיר. דוגמה: Cassandra.
3. Document Databases - בסיס נתונים המצמיד מפתח למבנה מורכב של מידע הנקרא "מסמך". מסמך יכול להכיל צמדים מורכבים של מפתח-ערך, מפתח-מערך, או אפילו מפתח-מסמך. דוגמה: MongoDB.
4. Search Engine Databases - כמו מסדי נתונים מסוג Document שבהם כל מפתח מזווג למסמך. אלא שהם מוסיפים לכל מסמך שמוחזר משאילתה, דירוג (scoring) כמה הוא מתאים למה שחיפשנו בשאילתה. דוגמה: Elastic Search.
5. Graph Stores - מאחסנים את המידע בצורה של גרף, עם צמתים וקשתות. דוגמה: Neo4J, Jena.

## Map-Reduce

זוהי תבנית שבה מריצים מספר אלגוריתמים במערכת מבוזרת המאפשרת לבצע חישוב מקבילי על כמות גדולה של מידע. באמצעות האלגוריתמים מחלקים את המידע למספר מחשבים (workers), כל מחשב מקבל חלק מהמידע ומבצע עליו חישוב. מאחדים את כל המסקנות ומחזירים תוצאה. כל תבנית זו מחולקת לשלושה שלבים, שכל שלב הוא אלגוריתם אחר:

1. Map - האלגוריתם הראשון תפקידו לסנן את המידע ולקחת רק את החלקים עליהם צריך להפעיל פעולת חישוב. בנוסף מבצע מיפוי על המידע הנותר ל-key ו-value, כך שנוכל לגשת אליו בצורה נוחה.
2. Sort/Grouping - מפרק את המידע לקבוצות הגיוניות, ומקצה לכל worker קבוצת מידע.
3. Reduce - כל worker מפעיל אלגוריתם זה על המידע שקיבל ומחזיר את התוצאה.

בחלק מהמקרים איחוד המידע שמוחזר מכל worker הוא עצמו התשובה, אמנם ישנם מקרים בהם צריך אלגוריתם נוסף שיעבד את כל המידע שהתקבל ויחזיר את התשובה. שלב זה נקרא finalize.

דוגמה לשימוש ב-Map Reduce. נתון מסד נתונים המחזיק מידע על סטודנטים כמו: ID, שם פרטי ומשפחה, גיל, כתובת וממוצע ציונים. נרצה למצוא את הציון הממוצע לכל שכבת גיל. בשלב ה-map נמפה בין הגיל לציון ונסנן את כל שאר המידע. בשלב ה-Grouping נאחד את הזוגות עם אותו גיל ונשלח ל-worker כלשהו. בשלה ה-Reduce כל worker ימצא את הממוצע ציונים בקבוצת הסטודנטים שקיבל ויחזיר את המידע. במידה וקבוצת הסטודנטים בגיל מסוים גדולה מכדי לשלוח את כולה ל-worker אחד, היא תפוצל כך שכל worker יחשב את הממוצע של הקבוצה שקיבל. לבסוף מכל הערכים שהוחזרו יעשה ממוצע ביחס לגודל כל קבוצה.

כיוון שמסדי נתונים מסוג NoSQL עובדים במערכת מבוזרת על כמויות גדולות של מידע, רבים מהם מאפשרים למתכנת לבצע Map-Reduce.

# Key-Value Stores

## הגדרה

מסד נתונים שבו כל הנתונים מאוחסנים בצורה של מפתח וערך. השליפה ממסד הנתונים תמיד תהיה לפי המפתח, כלומר נכניס את המפתח ונקבל את הערך או ערכים המתאימים לו. אין סדר על הנתונים, ניתן להכניס כל סוג מפתח שיצביע לכל סוג ערך, כמו XML, Json, תמונות, וידאו, וכו'. מסד נתונים זה הוא פשוט ומהיר.

## Redis

מערכת ניהול מסדי נתונים מסוג key-value. זוהי המערכת הפופולארית ביותר מסוג זה.

## פקודות Redis

### פקודות בסיסיות

SET key value – מכניסה למסד נתונים מפתח וערך המתאים לו.

GET key – מקבלת מפתח ומחזירה את הערך המתאים לו.

DEL key – מוחקת מפתח ממסד הנתונים.

INCR key – במידה וערך המתאים למפתח הוא מספר יגדיל אותו ב-1. אם הערך אינו מספר יחזיר שגיאה.

INCRBY key num – מגדיל את הערך המתאים למפתח ב-num.

### רשימות

כל מפתח יכול גם להכיל רשימה של ערכים.

RPUSH key value1, value2, … - דוחף את הערכים לסוף רשימת המפתח.

LPUSH key value1, value2, … - דוחף את הערכים לתחילת רשימת המפתח.

LRANGE key num1 num2 – מחזיר את כל הערכים ברשימה שמיקומם בין [num1,num2]. הערך האחרון מוגדר גם -1.

### Hashes

כל מפתח יכול להיות hashtable בעצמו, כלומר נוכל להכניס לו ערך שהוא בעצם מפתח שגם לו יש ערך.

HSET key innerKey value – מכניס לתוך המפתח החיצוני מפתח פנימי עם ערך.

HGET key innerKey – מחזיר את הערך המתאים למפתח הפנימי.

HGETALL key – מחזיר רשימה של מפתחות וערכים. המתאימים למפתח זה.

### קבוצות

כל מפתח יכול לשמש גם כ-set או ordered set. ההבדל בין set לרשימה הוא שב-set לא יכול להיות או ערך פעמיים.

SADD set item – מוסיף פריט לקבוצה.

SREM set item – מוחק פריט מהקבוצה.

SISMEMBER set item – בודק אם הפריט נמצא בקבוצה. אם כן מחזיר true, אחרת, false.

SUNIOUN set1 set2 – מחזיר איחוד של שתי הקבוצות.

ZADD sortSet score item – מכניס פריט עם תוצאה אל הקבוצה הממוינת. התוצאה נצרכת לקביעת המיקום של הפריט.

ZRANGE sortSet item num – מחזיר num פריטים מהקבוצה החל הפריט item.

### פקודות נוספות

EXPIRE key time – קובע כמה זמן הערך ישמר במפתח שצוין.

TTL key – מחזיר את הזמן שנותר למפתח להכיל את הערך.

KEYS pattern – מקבל שם חלקי של מפתח ומחזיר את שמות כל המפתחות שמתחילות בתבנית זו.

# Wide-Column Stores

## הגדרה

מסדי נתונים מסוג Wide-Column משתמשים בטבלאות, שורות ועמודות כדי לאחסן את המידע, כמו במסדי נתונים רלציוניים, אלא שכאן לכל שורה יכולים להיות עמודות שונות, אפילו באותה טבלה.

הרעיון המרכזי במסדי נתונים אלו הוא שמגדירים עמודה אחת או יותר המשמשת כמפתח, כך שלכל שורה יש מפתח ייחודי המזהה אותה. ניתן להסתכל על מסדי נתונים אלו כעין Hashtable שבו הערך לכל מפתח הוא שורה בטבלה. המפתח יכול להיות מספר עמודות, כך שכל Hashtable מצביע אל Hashtables פנימיים. בהכנסת שורה לטבלה ניתן לבחור לאיזה עמודות לתת ערכים, כך שניתן לקבל שורות עם עמודות שונות. בנוסף, הוספת עמודות חדשות לטבלה אינו מצריך לשנות את כל השורות הקיימות.

## Cassandra

זהו מערכת לניהול מסדי נתונים מסוג Wide-Column. השימוש בה מאוד נפוץ ובין היתר משתמשים בה חברות כמו: Facebook, Twitter, eBay, Netflix. הייחודיות של Cassandra הוא שהכתיבה בה ממש מהירה אמנם הקריאה ממנה יכולה להיות איטית. לכן בניגוד למסדי נתונים רלציוניים, שבהם רצינו שכל המידע יהיה רשום פעם אחת ולא יהיה תלוי אחד בשני, ב-Cassandra דווקא כן נרצה שהמידע יהיה משוכפל במספר מקומות כך שהגישה אליו תהיה יותר מהירה.

שפת השאילתות של Cassandra, הנקראת CQL (Cassandra Query Language), נראית כמו גישה פשוטה יותר של SQL. אמנם יש ביניהם מספר מצומצם של הבדלים. מלבד שינויים אלו, הפעולות ב-CQL זהות לפעולות של SQL.

1. כדי ליצור את מסד הנתונים משתמשים בפקודה CREATE KEYSPACE ובה יש לציין כמה שכפולים של המידע אנו רוצים. במערכות מבוזרות נרצה לשמור את המידע במספר שרתים, כך שאם שרת אחד נפל המידע לא נאבד ונוכל לשחזר אותו משרת אחר. נוכל לבחור האם להשתמש ב-SimpleStrategy שבו כל שכפול נשמר בשרת אחד, או ב- NetworkTopologyStrategyשבו כל שכפול מבוזר בכמה שרתים. דוגמה, להגדרת מסד נתונים:

CREATE KEYSPACE university WITH REPLICATION = {'class':'SimpleStrategy', 'replication\_factor':2};

1. לא ניתן לבצע שאילתות מקוננות, וכן לא ניתן להשתמש ב-join כדי לאחד שליפות.
2. הגבלות נוספות על תנאי WHERE עליהן נדבר בסעיף הבא.

## תנאי WHERE ב-Cassandra

ביצירת טבלה, כאשר מגדירים את העמודות המהוות את המפתחות, בניגוד לSQL, כאן יש חשיבות לסדר הגדרת המפתחות. המפתח הראשון נקרא Partition key, ואילו כל שאר המפתחות נקראים Clustering Keys. לדוגמה, ביצירת הטבלה למטה studentid הוא Partition key, ואילו course הוא Clustering Key.

CREATE TABLE grades (studentId INT, course TEXT, grade FLOAT, PRIMARY KEY(**studentId, course**));

כאשר רוצים לגשת למידע, סדר המפתחות מהווה את הכתובת באמצעותה המערכת יודעת את מיקום המידע. כמו path של קובץ המתאר היררכיה של ספריות במערכת הקבצים. מסיבה זו סדר המפתחות חשוב. הדרך שבו Cassandra מאחסנת את המידע גורמת לכך שיש מספר הגבולות על תנאי WHERE:

* ה-Partition key משמש כמו מפתח של hashtable שבאמצעותו ניגשים אל המקום שבו נמצא המידע. לכן בתנאי WHERE ה-Partition key חייב להיות העמודה הראשונה לאחר ה-WHERE, והתנאי עליו חייב להיות שווה '=' ולא אי-שוויון אחר.
* ה- Clustering Keysמסודרים בצורה היררכית, כמו עץ, שבסופו נמצא המידע. מסיבה זו, הסדר שלהם, שהוגדר ביצירת הטבלה, חשוב. ולכן בתנאי WHERE בכל פעם שרוצים לשים תנאי על אחד ה-Clustering Keys יש גם לשים תנאי על כל ה-Clustering Keys שנמצאים לפניו בהיררכיה.
* בכל תנאי WHERE שבו שמנו תנאי על מספר Clustering Keys, רק על האחרון בהיררכיה מבניהם יכול להיות תנאי שהוא אי-שוויון, ואילו על כל ה-Clustering Keys שלפניו התנאי הוא שווה '='.

מספר דוגמאות של שליפות עם תנאי WHERE על טבלה T שבה x הוא Partition key, ואילו y ו-z הם Clustering Keys.

SELECT \* FROM T WHERE x = 5; // correct

SELECT \* FROM T WHERE x = 5 AND z = 7; // wrong. y before z

SELECT \* FROM T WHERE x = 5 AND y < 3 AND z > 0; // wrong. y must be equal

SELECT \* FROM T WHERE x = 5 AND y < 6 AND y > 0; // correct

SELECT \* FROM T WHERE x = 5 AND z = 4 AND y > 0; // wrong. y must be equal

SELECT \* FROM T WHERE x < 10; // wrong. x must be equal

SELECT \* FROM T WHERE x =2 AND z <4 AND y = 3; // correct

SELECT \* FROM T WHERE y = 2 AND z < 8; // wrong. x must appear

# Document Store

## הגדרה

מסד נתונים מסוג Document הוא מסד נתונים שבו כל מפתח מזווג למסמך. כל מסמך הוא מבנה נתונים מורכב היכול להכיל זוגות של מפתח-ערך, מפתח-רשימת ערכים, ואף מפתח-מסמך. מסמכים אלו בדרך כלל יהיו קבצי XML או JSON, או קבצים מסוג דומה. בניגוד למסדי נתונים מסוג Key-Value, כאן ניתן לבצע שאילתה לא רק על המפתחות אלא גם על המסמכים ובערכים שבתוך המסמכים.

## MongoDB

זהו מסד הנתונים מסוג NoSQL עם שימוש הנפוץ ביותר. כל מסד נתונים מסוג MongoDB מורכב מאוספים של מסמכים. כל מסמך הוא בדרך כלל קובץ Json.

השפה המשמשת לבצע פעולות על MongoDB רגישה לאותיות גדולות וקטנות (Case sensitive). תכונה חשובה של שפה זו היא שכאשר נבצע פעולה על מסד נתונים או אוסף, אם הם קיימים נעבור אליהם, אך אם לא קיימים MongoDB ייצור אותם.

באתר [www.mongodb.com](https://www.mongodb.com/download-center) ניתן למצוא קישור להורדת שרת ולקוח של MongoDB, וכן ניתן להוריד GUI בשם Robomongo כדי להציג את השליפות שעשינו.

## יצירת מסד נתונים ואוסף

כדי ליצור מסד נתונים, יש לרשום use myDataBase או כל שם אחר למסד הנתונים, אם קיים מסד נתונים בשם זה תעבור אליו, ואם לא תיצור אותו. כדי למחוק מסד נתונים נשתמש בפקודה db.dropDatabase().

כדי ליצור אוסף, יש להשתמש בפקודת createCollection המקבלת שני פרמטרים: שם האוסף, ומספר אפשרויות להגדרת המסמך בשיטה זו. האפשרויות כוללות:

* size - מספר הבתים שהאוסף יכול להכיל.
* max - מספר מקסימלי של מסמכים שהאוסף יכול להכיל.
* capped - כאשר האוסף מגיע לגבול הקיבולת שלו ורוצים להכניס מסמך נוסף, המסמך הכי ישן ימחק והחדש יכנס במקומו.
* autoIndex - האם לתת אוטומטית ID ייחודי לכל מסמך במסד הנתונים.

דוגמה ליצירת אוסף של מסמכים שכל אחד מייצג סטודנט:

db.createCollection("students", { capped : true, size : 6142800, max : 10000, autoIndexID : true } )

כדי למחוק אוסף נשתמש בשיטת db.myDataBase.drop().

## הכנסה - insert

הכנסת המסמכים לאוסף היא באמצעות פונקציה insert, המקבלת כפרמטר אובייקט בפורמט JSON או מערך של אובייקטים בפורמט JSON, שכולם יכנסו למסד הנתונים. לדוגמה, הכנסות מסמכים מסוג סטודנט.

db.students.insert({

"FirstName": "Chaya",

"LastName": "Glass",

"id": "111",

"age": "21",

"Address": {

"Street": "Hatamr 5",

"City": "Ariel",

"Zip": "40792"

}

})

db.students.insert([{

"FirstName": "Tom",

"LastName": "Glow",

"Address": {

"Street": "Mishmar 5",

"City": "Ariel"

}

}, {

"FirstName": "Tal",

"LastName": "Negev",

"Address": {

"Street": "Yarkon 26",

"City": "Jerusalem"

}

}])

## חיפוש - find

כדי לחפש מידע ב-MongoDB נשתמש בפונקציה find. פונקציה זו יכולה לקבל עד שני פרמטרים. הפרמטר הראשון הוא כדי לסנן בין מסמכים, כמו תנאי WHERE, והשני הוא כדי לבחור איזה מידע להציג מתוך המסמכים שנבחרו, כמו SELECT. לא חייבים לציין את שני הפרמטרים, אלא אפשר להכניס רק את הראשון או לא להכניס אף פרמטר. אם לא נכניס אף פרמטר יוחזרו כל הנתונים באוסף, ואם נכניס רק את הראשון יוחזרו כל המסמכים המקיימים את התנאי.

המידע המוחזר מ-find יוצג כשורה אחת ארוכה בפורמט JSON, ללא הזחות. אם נרצה שהמידע יוחזר בצורה קריאה עם הזחות, נפעיל על find את הפונקציה pretty.

db.students.find().pretty()

### הפרמטר הראשון (WHERE)

כדי לבחור מסמך מסוים לפי התכונות הפנימיות בו, נציין בפרמטר הראשון את שם התכונה (מפתח) ואת הערך של התכונה אותו אנו מחפשים. ישנם כל מיני אופרטורים כדי לקיים תנאים שונים ולשלב בין תנאים:

* Equality - find({"<key>": "<value>"})
* Less Than - find({"<key>":{**$lt**: "<value>"}})
* Less Than Equals - find({"<key>":{**$lte**: "<value>"}})
* Greater Than - find({"<key>":{**$gt**: "<value>"}})
* Greater Than Equals - find({"<key>":{**$gte**: "<value>"}})
* Not Equals - find({"<key>":{**$ne**: "<value>"}})
* AND - find({**$and**: [{"<key1>": "<value1>"} , {"<key2>": "<value2>" }]}).

More convenient to use: find({"<key1>": "<value1>" , "<key2>": "<value2>" })

* OR - find({**$or**: [{"<key1>": "<value1>"} , {"<key2>": "<value2>" }]}).

דוגמאות לסינון של מסמכים:

db.students.find() // gets all documents collection

db.students.find({"Address.City": "Ariel"}) // gets all documents where city is Ariel

db.students.find({"FirstName":"Tom", $or: [{"LastName":"Negev"},{"LastName":"Glow"}]}) // combine and with or

### הפרמטר השני (SELECT)

בפרמטר השני, כדי לבחור תכונות ספציפיות מכל המסמכים שנבחרו בפרמטר הראשון, נוסיף ליד כל תכונה כזו את הערך את הערך true. כאשר הגדרנו את מסד הנתונים עם autoIndex, בכל מידע שמוזר ממסמך כלשהו יהיה בהתחלה את ה-ID של המסמך. כדי שה-ID לא יוצג נוסף בפרמטר השני \_id:false.

דוגמאות לבחירת תכונות:

db.students.find({"FirstName":"Tim"},{"LastName":true}) // gets last name of all students their first name is Tim

db.students.find({},{"FirstName":true, \_id:false}) // gets the first name of all students. No ID.

## עדכון - update

כדי לעדכן מסמכים הנמצאים באוסף נשתמש בפונקציה update. בפונקציה זו הפרמטר הראשון משמש כדי לבחור את המסמכים אותם אנו רוצים לשנות, כמו שראינו בפרמטר הראשון ב-find. בפרמטר השני נכניס את הערך שאנו רוצים שיכנס למסמך. בברירת מחדל, update מעדכן רק את המסמך הראשון שמוצא שמקיים את התנאי בפרמטר הראשון. כדי שיעדכן את כל המסמכים המקיימים את התנאי, יש להכניס פרמטר שלישי שבו נציין multi:true.

הערך שנכניס בפרמטר השני יחליף את המסמך כולו. אם נרצה לשנות תכונות ספציפיות במסמך ולא את כולו, נרשום בתחילת הפרמטר השני $set, ולאחר מכן את התכונות שנרצה לעדכן ואת הערכים המתאימים להם.

דוגמאות לעדכון מסמכים:

db.students.update({"FirstName":"Tom"}, {"FirstName" : "Tim", "LastName": "Glow", "Address" : { "Street" :

"Mishmar 5", "City" : "Ariel" }}) // update student Tom document to new document

db.students.update({"FirstName":"Tom"}, {$set:{"FirstName":"Tim"}},{multi:true}) // update all the students whose

current name is Tom to Tim

## Map Reduce

MongoDB מאפשר לבצע שאילתה בשיטת Map Reduce. כדי לעשות זאת יש להפעיל את הפונקציה mapReduce על מסד הנתונים. התחביר של השימוש בפונקציה הוא בשפת JavaScript. פונקציה זו מקבלת שלושה פרמטרים:

1. מצביע לפונקציה המבצעת map (באמצעות הפונקציה emit המקשרת key ל-value).
2. מצביע לפונקציה המבצעת reduce, פונקציה זו צריכה לקבל key ו-value, והיא תופעל אוטומטית על כל מסמך שלא סונן בשלב ה-map.
3. בפרמטר זה נגדיר אפשרויות נוספות כמו: query המבצע סינון על המסמכים כמו בפרמטר הראשון ב-find, out שבו נותנים שם לתוצאות שמוחזרות מה-map reduce, ו-finalize שהוא מצביע לפונקציה שתבצע חישובים סופיים על הערכים שהוחזרו מ-reduce.

השם שהגדרנו ב-out מהווה אוסף בפני עצמו המכיל את כל המידע שהוחזר מה-map reduce. לכן כדי לקבל את המידע נצטרך להפעיל על שם זה את פונקציית find.

דוגמה לשימוש ב-map reduce על מסד נתונים המתאר הזמנות של מוצרים. המטרה היא למצוא עבור כל מוצר מהי הכמות הכוללת שהוזמנה ממנו בכל ההזמנות יחד, ומהי הכמות הממוצעת בכל הזמנה.

**db.orders.mapReduce(**

**mapFunction,**

**reduceFunction,**

**{**

**out: { merge: "sku\_info" },**

**query: { ord\_date: { $gt: new Date('01/01/2012') } }**

**finalize: finalizeAddAvgField**

**}**

**)**

var **mapFunction** = function() {

for (var idx = 0; idx < this.items.length; idx++) {

var key = this.items[idx].sku;

var value = {

count: 1,

qty: this.items[idx].qty

};

emit(key, value);

}

};

var **reduceFunction** = function(keySKU, countObjVals) {

reducedVal = { count: 0, qty: 0 };

for (var idx = 0; idx < countObjVals.length; idx++) {

reducedVal.count += countObjVals[idx].count;

reducedVal.qty += countObjVals[idx].qty;

}

return reducedVal;

};

var **finalizeAddAvgField** = function (key, reducedVal) {

reducedVal.avg = reducedVal.qty/reducedVal.count;

return reducedVal;

};

# Search Engine Databases

## הגדרה

מסדי נתונים בשיטת Search Engine Databases עובדים כמו מסדי נתונים מסוג Document שבהם כל מפתח מזווג למסמך. אלא שהם מוסיפים לכל מסמך שמוחזר משאילתה, דירוג (scoring) כמה הוא מתאים למה שחיפשנו בשאילתה. מסדי נתונים אלו, כמו שמציין השם שלהם, מתאימים בעיקר למנועי חיפוש, שבהם אנו רוצים שיופיעו קודם התוצאות הכי רלוונטיות לחיפוש שלנו.

## Elastic Search

זהו מסד נתונים מסוג Search Engine Database הנפוץ ביותר. משתמשים בו: Wikipedia, StackOverflow, GitHub, ועוד. ניתן להשתמש בו במודל של REST או ב-Java API. נבנה על מערכת Lucene. הוא נחשב למערכת שקרובה לזמן אמת, מאחר ולוקחת שנייה אחת מאז שמסמך נוצר או מעודכן ועד שניתן לחפש אותו. אנו נשתמש בשפת cURL כדי להפעיל עליו פקודות HTTP.

## פקודות

כמו ב-MongoDB אין צורך ליצור מסד נתונים או מסמך, נוכל פשוט להשתמש בהם והם ייוצרו לבד.

* XPUT - משמש להוספת מסמך. צריך לציין את מיקום המסד נתונים, שם המסד נתונים, שם האוסף וה-ID של המסמך.

curl -XPUT "http://localhost:9200/university/students/111" -d

"{\"FirstName\": \"Chaya\", \"LastName\": \"Glass\", \"age\": \"21\", \"Address\": { \"Street\":\"Hatamr 5\",

\"City\": \"Ariel\",\"Zip\": \"40792\"}}"

* XPOST – כמו XPUT משמש כדי להכניס מסמך, אלא שלא צריך לציין ID של מסמך הוא תמיד ייצור אחד חדש. ההבדל בין XPOST ל-XPUT שאם נפעיל את אותה פקודת XPUT מספר פעמים תמיד הוא יפנה לאותו מסמך, ואילו אם נפעיל את XPOST מספר פעמים הוא ייצור על פעם מסמך חדש.

curl -XPOST "http://localhost:9200/university/students" -d

"{\"FirstName\": \"Tal\", \"LastName\": \"Negev\", \"age\": \"28\"}"

* XGET – משמש כדי לשלוף מידע. אפשר לבצע חיפוש רגיל כמו שאנו מכירים עם תנאים, אך גם אפשר לבצע חיפוש עם filter, שבו אנו מספקים איזשהו טקסט המתאר את מה שאנו מחפשים. Elastic Search יחזיר את המסמך שהמידע בו הכי מתאים לטקסט שסיפקנו. ניתן גם לחפש התאמה מדויקת באמצעות match\_phrase.

curl -XGET "http://localhost:9200/university/students/\_search? q=LastName:Negev"

curl -XGET http://localhost:9200/university/students/\_search -d

"{ \"query\" : { \"match\" : { \"description\" : \"doesn't show-up learns\" } }}

* XHEAD – אותו תחביר כמו XGET אלא שרק בודק אם המסמך קיים במסד נתונים ולא מחזיר את מה שנמצא בו.

curl -XGET "http://localhost:9200/university/students/333"

* XDELETE – מוחק את המסמך.

## TF-IDF

השיטה שבאמצעותה Elastic Search יודע לתת ניקוד לכל מסמך, ובכך להחזיר את המסמכים מדורגים מהטוב ביותר לגרוע ביותר, היא TF-IDF. לפי שיטה זו בכל מסמך עוברים על כל המילים בטקסט שאנו מחפשים, ולכל מילה מחשבים ניקוד בשני שלבים. הניקוד הכולל של המסמך יהיה סכום הניקוד של כל המילים בחיפוש. שני השלבים שבאמצעותם כל מילה k מקבלת ניקוד הם:

1. TF (Term Frequency) - מחשבים מספר הפעמים שהמילה k מופיעה במסמך חלקי מספר המילים במסמך כולו.
2. IDF (Inverse Document Frequency) - מכניסים לפונקציה log את תוצאת החילוק של מספר המסמכים הכולל במספר המסמכים ש-k מופיעה בהם.

נניח כי Q היא קבוצת כל המילים שאנו מחפשים, k היא מילה מ-Q, D היא קבוצת כל המסמכים, ו-d מסמך נוכחי. הניקוד של d יחושב לפי הנוסחה:

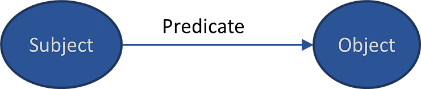
*ישנם מספר דרכים להגדיר את* TF-IDF*, הדרך שהגדרנו היא הנפוצה ביותר.* TF-IDF *מתחשבת גם במספר המופעים של המילה במסמך ביחס למספר המילים בו, אך גם בשכיחות של כל מילה בכל המסמכים. ככל שמילה נדירה יותר כך היא נחשבת מילת מפתח חשובה יותר, ולכן ניתן לה ערך גדול יותר.*

# Graph Databases

## הגדרה

אלו מסדי נתונים שמאחסנים את המידע בצורה של גרף, עם צמתים וקשתות. מסדי נתונים אלו מאפשרים שאילתות מסוג של "מצא את כל החברים" של אובייקט כלשהו, את החברים שלהם, וכן הלאה עד רמת קרבה 10. משמשים בעיקר לשמירת מידע הקשור לרשתות וקשרים חברתיים.

## Apache Jena

זהו מסד נתונים שמאחסן את הנתונים בצורה של גרפים. הוא מבוסס על מודל RDF (Resource Description Framework) כדי לאחסן את המידע. לפי מודל RDF כל הנתונים מאוחסנים משלישיות של subject, predicate, object (בעברית נושא, נשוא, מושא). ה-subject הוא אלמנט שמכיל את המידע, ה-predicate היא תכונה של האלמנט, וה-object הוא הערך המתאים לתכונה של האלמנט. כל שלישיות אלו מאחסנות במין טבלה אחת ענקית.

שפת השאילתות של RDF נקראת SPARQL. המימוש של SPARQL המתאים ל-Jena נקרא ARQ. כל השאילתות בשפה זו מורכבות מהשלישייה שצוינה לעיל.

## ARQ - SELECT

בכל שליפה ממסד הנתונים צריך לתת שם לאלמנטים שאנו רוצים לשלוף, לאחר מכן לציין בתנאי WHERE את שם האלמנט שציינו, ולאחריו את ה-predicate וה-object בהם אנו מעוניינים. כל שליפה מסתיימת באופרטור נקודה. נוסיף לשאילתה DISTINCT במקרה שיש מספר אלמנטים זהים ואנו לא רוצים שכפולים.

SELECT ?something WHERE {

?something predicate object .}

אפשר בפקודה אחת לשרשר שליפות, כלומר לבצע שליפה מהאלמנטים שקיבלנו בשליפה הראשונה. כדי לעשות זאת יש להוסיף שלישייה נוספת לאחר אופרטור נקודה של השליפה הראשונה.

SELECT ?something WHERE {

?something predicate1 object1 .

?something predicate2 object2 .}

במקרה שיותר קל לציין אילו אלמנטים אנו לא מעוניינים, נוכל לבצע שליפה על אלמנטים אילו ולבקש את האלמנטים המשלימים שלהם, על ידי שנרשום לפני השלישייה Filter NOT EXISTS.

דוגמאות:

SELECT ?something WHERE { // Select all Persons

?something rdf:type dbo:Person .}

SELECT ?thing WHERE { // Select all Women

?thing rdf:type dbo:Person .

?thing tto:sex "female" .}

SELECT DISTINCT ?person WHERE { // Select Persons That Have **Cats**

?person rdf:type dbo:Person .

?person tto:pet ?type .

?type rdf:type tto:Cat .}

SELECT ?person WHERE { // Select Persons That do **not** Have any Pets

?person rdf:type dbo:Person .

FILTER NOT EXISTS {?person tto:pet ?pet } .

## Neo4J

מסד נתונים שבו המידע נשמר בצמתים המקיימים ביניהם קשרים. הצמתים מחולקים לקבוצות, וכל צומת היא בדרך כלל מסמך JSON. במקור השימוש ב-Neo4J היה רק דרך ספריית Java (ולכן בשם שלו יש 4J), אמנם בפועל ניתן להשתמש בו גם באמצעות REST. שפת השאילתות שמשתמשים בה במסד נתונים זה נקראת Cypher. כל המושגים ב-Neo4J לקוחים מהסרט "מטריקס". שפה זו אינה רגישה לאותיות (case insensitive).

### קשרים

כמו שנראה בהמשך, השאילתות יהיו על קשרים בין צמתים. כדי לייצג קשר בין צמתים נסמן את הצמתים בסוגריים עגולות, את הקשר בסוגריים מרובעות, ונוסיף חיצים כדי לסמן את כיוון הקשר. תצוגת הקשרים תיראה כך:

(node1)-[:relation}->(node2) //node1 is related to node2, but not the opposite

(node1)-->(node2) / (node1)-[}->(node2) // no relation

## CREATE

פונקציה המשמשת כדי ליצור צומת. בתוך הסוגריים של הפונקציה נציין את שם הצומת, נקודותיים, הקבוצה לה משתייכת הצומת, ולבסוף את המסמך JSON שהצומת מכילה בתוכה. נוכל ליצור יותר מצומת אחת בפקודה אחת, על ידי רצף של סוגריים, כמו שהגדרנו לעיל, שהמפריד ביניהם הוא פסיק.

השם של הצומת הוא לא רלוונטי לאחר שסיימנו את פקודת CREATE, השימוש בשם זה הוא כדי להגדיר קשרים בצמתים חדשים. הגדרת הקשרים נעשית גם היא בתוך פקודת CREATE, מיד לאחר יצירת המשתנים, נרשום סוגריים ולאחריהם נציין את כל הקשרים בפורמט שהגדרנו בסעיף קודם. נראה מספר דוגמאות:

CREATE //

(glass:student {name: 'Chaya Glass', id:111, age:21, degree:'1'})

CREATE // create 3 students and define relation between them.

(glass:student {name: 'Chaya Glass', id:111, age:21, degree:'1'}),

(negev:student {name: 'Tal Negev', id:222, age:28, degree:'3'}),

(golan:student {name: 'Gadi Golan', id:333, age:24, degree:'1'}),

(negev)-[r1:teaches]->(glass), (golan)-[:in\_class\_with]->(glass), (glass)-[:in\_class\_with]->(golan)

## MATCH

פקודת MATCH משמשת כדי לשלוף מידע ממסד הנתונים. בפקודה זו צריך לתת שמות לצמתים שאנו שולפים ולבחור איזה קשר אנו מעוניינים שהצמתים יקיימו. לבסוף, באמצעות המילה השמורה RETURN, להחזיר את הצמתים עצמם או תכונות פנימיות שלהם. דוגמאות:

* Find all nodes who have any relation with Chaya Glass

MATCH (a)-->(b{name:'Chaya Glass'}) RETURN a

* Find all names of those who teach the student Chaya Glass:

MATCH (a)-[:teaches]->(b:student {name:'Chaya Glass'}) RETURN a.name

### Path

לעיתים לא נרצה רק את הצמתים בסוף החיפוש אלא גם את מסלול הקשרים בין שני צמתים. לשם כך בתחילת פקודת ה-MATCH נגדיר משתנה שלתוכו נכניס את המסלול. בסוגריים המרובעות, המציינות את הקשר, נציין אילו רמות במסלול אנו מחפשים. ציון הרמות בקשר יהיה בצורה הבאה: [:relation\*val1..val2]. פקודה זו תמצא את כל המסלולים מהשכנים שהם באורך val1 עד val2, לא יותר ולא פחות. אם נציין רק כוכבית ולא את המספרים תחזיר את כל המסלולים בין הצמתים. נוכל גם למצוא דווקא את המסלול הכי קצר, אם נרשום לאחר המשתנה שאליו יכנס המסלול ShortestPath. דוגמאות:

* Will return all paths of length 2 to 4 of type KNOWS, between Gadi Golan and others.

MATCH p=(a {name:'Gadi Golan'})-[:KNOWS\*2..4]->(b) RETURN p

* Will return the shortest path (using any type of relation, and in any direction) between Gadi Golan and Tal Negev

MATCH p=shortestPath((s1:student {name:'Gadi Golan'})-[\*]-(s2:student {name:'Tal Negev'})) RETURN p

### אפשרויות נוספות

ניתן להוסיף כל מיני מילים שמורות כדי שנוכל לרשום שאילתות מורכבות יותר.

WITH – הכנסת קבוצת צמתים למשתנה.

IN – כדי לחפש בתוך קבוצה של צמתים.

ALL – מצריך שיתקיימו כל התנאים.

ANY – מספיק שתנאי אחד מקויים.

AND – כדי להוסיף תנאי המסנן קבוצת צמתים.

OR – כדי לאחד בין צמתים המקיימים חלק מהתנאים.

COLLECT – מאחד קבוצת צמתים לאוסף אחיד. משמש כדי לשמור תוצאות שליפה במשתנה כדי שיהיה אפשר לבצע עליו שליפות נוספות.

COUNT – סוכם מספר צמתים בקבוצה.

דוגמאות:

* Returns all students that study all courses.

MATCH (c:course) WITH COLLECT(c) AS courses MATCH (s:student) WHERE ALL (x IN courses WHERE (s)-[:studies]->(x)) RETURN s.name

* Returns all students that study at most 4 courses.

MATCH (s:student)-[:studies]->(c:course) WITH s, COUNT(c) as num\_courses WHERE num\_courses <= 4 RETURN s.name

* Returns all students that are Tal Negev's friend and know more than 3 students.

MATCH (negev { name:"Tal Negev" })-[:friend]->(frOfNegev:student)-[:knows]->(st:student) WITH frOfNegev, COUNT(st) AS frCount WHERE frCount > 3 RETURN frOfNegev