ב"ה נעשה ונצליח

מסדי נתונים – DB

מרצה: ד"ר שרה כהן, מתרגלת: רחל בכר, סיכמה: עדי במברגר-אדרי

לתוכן העניינים

הרצאה 1 – ER Diagrams

מסד נתונים- אוסף של מידע.

מערכת לניהול מסד נתונים DBMS- כלי התוכנה שמשתמשים בו כדי לתחזק ולנהל את הנתונים.

למעשה כשאומרים מסד נתונים מתכוונים לכלי עצמו.

הנושאים המרכזיים:

- תכנון הסכימה, איך מחליטים באיזו צורה לשמור את הנתונים במסד
- איך אפשר לתשאל ולשלוף מידע
- איך המסד נתונים דואג לכך שהשאילתות יעבדו בצורה מהירה ויעילה
- איך מסד הנתונים מתמודד עם הרבה משתמשים בו זמנית ושומר על הנתונים תיקנים ושלא יתקלקלו בדרך
- איך מתמודדים עם התאוששות כשהמערכת נופלת/החשמל נופל- איך הנתונים לא ילכו לאיבוד

נלמד:

- איך להשתמש במסד נתונים באופן טכני
- איך מסד נתונים עובד מאחורי הקלעים
  - הבנה נכונה תאפשר להשתמש במסד בצורה נכונה
  - ייתן לנו כלים לפתח מסד נתונים חדש אם נרצה

<u>מידול הנתונים בעזרת דיאגרמות ישויות-קשרים</u>

יש 4 שלבים עיקריים לתכנון מסדי נתונים:

1. ניתוח הצרכים
   a. מה המידע שצריך לשמור?
   b. איך נשתמש בו?
   c. אילו אילוצים יש על המידע?
2. ניתוח קונספטואלי
   a. תרגום התיאור המילולי לדיאגרמה, בעלת משמעות סמנטית מדויקת
   b. נראה דיאגרמות ישויות קשרים, אך יש גם UML
3. תכנון לוגי
   a. מחליטים אילו טבלאות רוצים לשמור
   b. תרגום מהדיאגרמה והבנת האילוצים על הטבלאות
4. תכנון פיזי
   a. איפה כל פריט מידע ישב על הדיסק
   b. גישה לפריטים בצורה מהירה

ישויות, תכונות וקשרים

ישות היא אובייקט בעלם, שאפשר להבדיל בינו לבין ישות אחרת בעזרת התכונות שלו. (לדוגמא- אני, הקורס הזה. אנטי דוגמא- עלה)

קבוצת ישויות היא קבוצה של ישויות מאותו סוג (סטודנטים, קורסים). מצוין בדיאגרמה בעזרת מלבן.
תכונות כדי לתאר את הישויות בתוך הקבוצה. לכל הישויות חייב להיות ערך אמיתי (לא null) לכל אחת מהתכונות. למשל לקבוצת השחקנים- בן זוג לא יכול להיות תכונה, כי יש שחקנים שאין להם בן זוג. בנוסף, לכל תכונה חייב להיות ערך יחיד (שמעוניינים לשמור). אז תכונה של email תהיה תקינה אם לכל שחקן יש בדיוק כתובת email אחת (לא יותר ולא פחות).
מפתח היא קבוצה מינימלית של תכונות שמזהים באופן יחיד את הישויות בתוך הקבוצה. למשל תעודת זהות. לכל קבוצת ישויות חייב להיות מפתח, ואותו מציינים בעזרת הקו התחתון מתחת לשם התכונה.
קשר הוא אסוציאציה בן 2 או יותר ישויות.
קבוצת קשרים היא קבוצת קשרים מאותו סוג. מציינים בעזרת מעוין.
יכולות להיות גם קבוצות קשרים רקורסיבית- מקשרת בין אותה ישות לעצמה. (קשר בין עובד למנהל)
גם לקבוצת קשרים יכולות להיות תכונות, שמתארות את הזוגות בתוך הקשר.
קבוצות קשרים לא חייבות להיות בינאריים, אלא יכולות להיות גם של יותר מרכיבים- וגם אז כמות המרכיבים בקשר חייבת להיות קבועה.

כפילות בקשרים

לפעמים נרצה להוסיף אילוץ, שישות תוכל להשתתף רק פעם אחת בקבוצת קשרים כלשהי.
מסמנים את זה בעזרת חץ (מי שמצביע יוכל להשתתף לכל היותר פעם אחת).
יש קשרים של אחד להרבה, ויש של אחד לאחד (חץ דו כיווני).
דוגמא סטודנטים שלומדים באוניברסיטאות, אבות לילדים.
זה אפשרי גם בקבוצת קשרים לא בינאריים.
דנים בכל אחד מהחיצים בנפרד.

שלמות קשרים- האם ישות חייבת להשתתף באיזושהי קבוצת קשרים?

חץ עגול מציין שכל ישות שמצביעה חייבת להשתתף בדיוק פעם אחת.
במצב כזה, הקשר בעצם זהה לתכונה. מקובל שכשישה תכונה שמגיעה מתוך קבוצה קטנה ומוגדרת מראש של ערכים, נמדל אותה ע"י קבוצה של ערכים במקום ע"י תכונה.

ירושה

דומה לירושה בשפות תכנות (ואין ירושה כפולה), אבל שונה כי אין לקבוצות ישויות פעולות אלא רק תכונות. מצויר בעזרת משולש שכתוב בו ISA. קבוצה שיורשת מקבוצה אחרת, מוכלת בה ממש.
אם יש 2 קבוצות שיורשות מקבוצה אחרת, זה לא אומר בהכרח שהחיתוך שלהן ריק. בנוסף, הן לאו דווקא מכסות את כל הקבוצה ממנה הן יורשות.
למה שנרצה לשמור מידע על קבוצה שיורשת מקבוצה אחרת, אם היא לא מוסיפה עליה תכונות?

- עצם השמירה ככה נותנת לנו ידע נוסף
- השתתפות בקבוצת קשרים מיוחדת

קבוצת ישויות חלשה

המפתח שלה (ושאר התכונות) לבד לא מספיקות בשביל לזהות אותה, אלא צריך לדעת גם מפתח של קבוצת ישויות נוספת שהיא משויכת אליו (באופן יחיד). (מס' חשבון בנק ומס' הבנק)
זה מסומן בעזרת מרובע כפול סביב השם של קבוצת הישויות, ומעוין כפול סביב השם של קבוצת הקשרים. זיהוי של קבוצת ישויות כזו יהיה בעזרת המפתח שלה והמפתח של קבוצת הישויות אליהן היא קשורה.

קבוצת הישויות החלשה יכולה להשתתף בקשרים אחרים רגילים (ולא) עם קבוצת ישויות אחרות. **דוגמא** גדוד, פלוגה ומחלקה. זיהוי מחלקה הוא לפי מספר המחלקה, אות הפלוגה ומספר הגדוד. זה מאפשר למדל סוגי ישויות חדשות, שתלויות בישויות אחרות (כדי לא ליצור מצב שמידלנו את זה ששחקן יכול לזכות בפרס על סרט שלא שיחק בו).

<u>תרגום נתונים למבנה שאפשר לשמור בתוך מסד נתונים</u>

התכנון הלוגי. מעבר בין דיאגרמת ישויות למודל יחסי, ואז כתיבה של זה בסינטקס של מערכת הDB.

המודל היחסי הוא בעצם טבלאות. המושגים בהם זה:

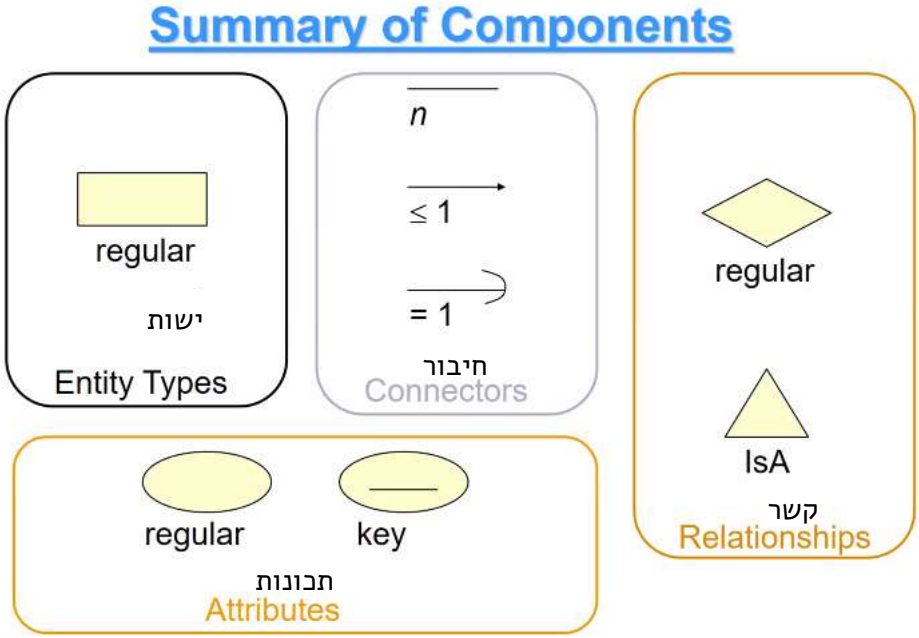
- **יחס**, בפועל זו פשוט טבלה. דומה בעולם של לוגיקה ליחס
- **מופע** של היחס הוא אוסף השורות שנמצאות בתוך היחס
- **סכימה** היא תיאור של שם היחס והתכונות שלו, כלומר העמודות בטבלה. את המפתח של היחס נציין עם קו תחתון
- **מפתח** היא קבוצה של תכונות אם לא יכולות להיות שתי שורות שונות עם אותן ערכים עבור כל הערכים שבמפתח. זוהי קבוצה מינימלית שבעזרתה אפשר לזהות מופעים ביחס.

בתרגום אנחנו צריכים להסתכל על כל אחד מהמרכיבים של הדיאגרמה ולתרגם אותו. נראה כללים לתרגום עבור כל אלמנט. לפעמים יש סיבות לסטות קצת מהתרגום הסטנדרטי.

1. כשנרצה לתרגם קבוצת ישויות, ניקח את שם קבוצת הישויות כשם היחס, את התכונות כתכונות של היחס ואז המפתח כמפתח של היחס.
2. כשנרצה לתרגם קבוצת קשרים, השם של היחס יהיה שם קבוצת הקשרים, התכונות שלו יהיו התכונות שלו עצמו ובנוסף המפתח של אחד מקבוצת הישויות שמשתתפות בקשר. כדי לייצר מפתח, ניקח את המפתחות של כל אחת מקבוצות הישויות שייצגנו.
3. עבור קבוצת קשרים רקורסיבית, נשים פעמיים את המפתח של קבוצת הישויות שמשתתפת בה. על מנת להבדיל, שינינו את השם של התכונה
4. תרגום קבוצת קשרים שבה יש חץ משולש (אחד להרבה)- מקבל את אותן תכונות כמו תרגום רגיל, אבל המפתח שונה עכשיו- המפתח של קבוצת הקשרים רק של זה שמוגבל להופיע לכל היותר פעם אחת.
5. אם יש כמה חיצים בדיאגרמה, יהיו לנו כמה אופציות שונות למפתחות שאפשר לבחור (ככמות החיצים בה).
6. עבור חץ מעוגל, לא חייבים לשמור את המידע ביחס נפרד אלא אפשר להכניס ישירות לתוך מי שמצביע אליו.
7. אם היחס הוא לא בינארי ויש חץ מעוגל, כן נצטרך ליצור לו יחס נפרד.
8. תרגום קבוצת ישויות חלשה- המפתח הוא של קבוצת הישויות החלשה יחד עם המפתח של קבוצת הישויות שבעזרתה מזהים את הישויות בקבוצה. לא צריך לייצר יחס עבור קבוצת הקשרים שדרכה מזהים את קבוצת הישויות החלשה.
9. תרגום ISA- כמה אופציות:
   a. E/R style Conversion - יצירת יחס עבור כל אחת מקבוצת הישויות. ישות אחת יכולה שהמידע שלה יהיה מפוזר על כמה יחסים. לקבוצות ישויות שיורשות, צריך לתת את התכונות של עצמן ואת המפתח של קבוצת הישויות ממנה הם יורשים.
   b. Object Oriented Approach - יצרת יחס עבור כל קומבינציה אפשרית של ישויות בקבוצה. אם יש ידע נוסף על התחום, זה יכול להשפיע על כמות היחסים שנייצר.
   c. Null Value Approach - יצירת יחס אחד שמכיל את כל התכונות, ואם יש ישות שחסרות לה חלק מהתכונות- הערך הזה פשוט יהיה null.

תרגול 1 – דיאגרמות ER

נרצה למדל מידע על עולם מסוים כדי שהיא תעזור לנו לבנות DB.



אם יש לי מגבלה על קשר בין קבוצת ישויות לעצמה, צריך לכתוב על החיבור פירוט.

הרצאה 2 – Relational Algebra

<u>יצירת טבלאות</u>

נדבר על פקודות שנשתמש בהן.

יצירת טבלה ;CREATE TABLE Name() כאשר בתוך הסוגריים ניתן את הפרטים על העמודות (מופרד בפסיקים)- שם העמודה, סוג והאילוצים עליה. לאחר מכן נציין אילוצים על הטבלה כולה.

SQL הוא case insensitive.

יש טיפוס varchar שהוא באורך משתנה, עד לאורך שציינו. והטיפוס char הוא בדיוק באורך שציינו.

כדי לשמור על המידע נכון, כל פעולה שסותרת אילוץ לא תתקבל, לכן כדאי לשים כמה שיותר אילוצים. אבל חשוב לדעת, שכשיש אילוצים פעולות של עדכון, הכנסה ומחיקה הן יותר איטיות.

אילוצים בתוך פקודת הcreate table:

- not null
- default() – נתינת ערך דיפולטיבי לשדה
- אילוצי בדיקה check- על ערך של שורה או על ערך של עמודה (נקרא אילוץ טבלה)
- unique – לא יכולות להיות שתי שורות בטבלה עם אותו ערך בדיוק בשדה הזה.
  אפשר גם להגדיר אילוץ unique על הטבלה כולה, למקרה שמדובר על יותר משדה אחד.
- primary key – גורר שלא ערך null וגם unique. ניתן להגדיר אותו רק פעם אחת עבור כל טבלה. כשמגדירים מפתח ראשי, מערכת הDB מסיקה הרבה דברים, כמו שהרבה מהגישה לטבלה תהיה דרך בדיקות על השדה הזה. לכן מייצר מבנה נתונים שיאפשר גישה יעילה דרך השדה הזה.
  ניתן לכתוב גם כאילוץ על הטבלה.
- foreign key – אילוץ מפתח זר. למקרה שנרצה שמידע שמופיע בשדה מסוים ישאב מטבלה אחרת, נשתמש בforeign key(x) references D(y) כאשר x שם השדה אצלנו, D שם הטבלה ממנה אנו שואבים את המידע וy שם השדה בטבלה ממנה שואבים.
  כדי שזה יהיה מותר, y חייב להיות Unique או מפתח ראשי בD.

- בגלל שמערכת הDB לא מאפשרת לסתור אילוצים, אם נרצה למחוק שורה של y בD, כל עוד יש שורה x שמשתמשת בy, המערכת לא תאפשר לפקודת המחיקה להתבצע (תעלה שגיאה).
- אפשר להוסיף בסוף האילוץ on delete cascade, שיגרום שמחיקה של y יגרור מחיקה של x.

מערכת הDB לא מאפשרת שום שינוי שגורם לאילוץ להיות מופר.

בעיה- אילוצי מפתח זר מעגליים, שלא מאפשרים הכנסה של נתונים. נראה את הפיתרון בהמשך, בעזרת אישור זמני להפר את האילוץ.

מחיקת טבלה DROP TABLE Name;. אם יש אילוצי מפתח זר, צריך לשים לב לסדר של מחיקת הטבלאות. כדי להיפתר מהבעיה הזו, ניתן לכתוב DROP TABLE Name cascade;, מה שימחק את האילוצים שהצביעו על הטבלה הזו (לא מוחק שורות נוספות אלא אילוצים).

הכנסת שורות לטבלה

- INSERT INTO TableName(column1, columns2) VALUES(value1, value2)
- או INSERT INTO TableName VALUES(value1, value2), וזה כשמכניסים את הערכים לפי סדר העמודות שציינו כשהגדרנו את הטבלה.
- אפשר גם להכניס ערכים רק לחלק מהשדות, ושאר הערכים יהיו null (אלא אם לא הכנסנו ערך לשדה שהוגדר כnot null. אם יש שדה עם ערך דיפולטיבי שלא הכנסנו לו ערך, הערך הדיפולטיבי הוא זה שירשם).
- אפשר גם להכניס שורות רבות לתוך טבלה, בעזרת bulk loader.

המודל הרלציוני

המודל הרלציוני הוא דרך אבסטרקטית (מתמטית) יותר לתאר המושג של טבלה.

לטבלה נקרא relation או **יחס**. יש בן instance, שזה אוסף השורות שנמצאות בו כרגע. בנוסף, יש schema שזה שם הטבלה יחד עם שמות העמודות attributes של היחס.

יחס זו קבוצה של שורות:

1. בפרט, אין שורות זהות (שורה לא יכולה להופיע כמה פעמים ביחס).
2. בנוסף, נניח שאין ערכי null.

סדר השורות הוא לא בר משמעות, וגם סדר העמודות לא בר משמעות עבורנו.

אלגברה רלציונית

מפעילים פעולות אלגבריות על יחסים, ומקבלים יחס חדש.

בפועל, משתמשים בשפת שאילתות שנקראת SQL- שמקבל טבלאות ופולטת טבלאות.

אלגברה רלציונית חשובה כי זה חלק ממה שקורה מאחורי הקלעים (שאילתה- תרגום לביטוי באלגברה רלציונית- אופטימיזציות של המערכת- חישוב).

אופרטורים אונאריים

הטלה $\Pi_{1,..,n}$- מקבלת יחס ו n אותיות שמייצגות עמודות מתוך היחס, ומחזירה רק את העמודות הרצויות מתוך היחס. זה יוריד כפילויות.

בחירה σ_{rule}- מקבלת יחס ותנאי בוליאני, ומחזירה רק את השורות מתוך היחס עבורן התנאי הבוליאני מתקיים. התנאים יכולים להיות תנאים מורכבים, אבל יבדק בכל פעם על שורה אחרת מתוך היחס.

אפשר לשלב בין השאילתות האלו (החישוב מתחיל מפנים החוצה).

אופרטורים בינאריים

איחוד U- מקבלת שני יחסים שחייבים להיות מתאימים- כלומר בדיוק את אותה הסכימה (אותן שמות העמודות), ומחזירה את איחוד השורות של שני היחסים שהתקבלו. אם יש שורה שמופיעה פעמיים, היא תופיע רק פעם אחת בתוצאה.

חיסור - - פועלת גם כן רק על יחסים מתאימים. מחזירה את כל השורות שמופיעות בראשון ולא מופיעות בשני.
מכפלה קרטזית ×- מקבלת שני יחסים, אחד עם n עמודות והשני עם m עמודות. מוציאה שורות עם n+m עמודות, שהן כל הצירופים האפשריים הקיימים מתוך השורות בשני היחסים. אם יש עמודות עם אותו השם בשני היחסים, נציין לאיזו מהן התכוונו בעזרת שימוש בשם_הטבלה.שם_העמודה.

אופרטור טכני- עוזר לנו לנסח שאילתות בצורה ברורה
שינוי שם renaming $\rho_{R(n_1,\dots,n_n)}$- מוחזר אותו יחס, אבל עם סכימה חדשה. יעיל כדי לשבור ביטוי ארוך אלגברית (לתת שם לתוצאות ביניים באמצע תהליך החישוב). בנוסף, יעיל כדי לפתור קונפליקטים בשמות העמודות.

אופרטורים נוספים- syntactic sugar

אלו אופרטורים שאפשר לבטא בדרכים אחרות- אבל הם נפוצים אז הם אופרטורים בפני עצמם.
חיתוך ∩-אפשר לבטא את $R \cap S$ בעזרת $R - (R - S)$. עובדת רק על יחסים מתאימים. התוצאה מכילה את כל השורות שנמצאות בשני היחסים שקיבלנו.
צירוף על תנאי $R \bowtie_{rule} T$ -תנאי בחירה על מכפלה קרטזית. שקול ל$\sigma_{rule}(R \times T)$.
צירוף טבעי $R \bowtie T$ -דרישת שוויון על כל אחד מהעמודות המשותפות. שקול ל:

1. מכפלה קרטזית של R וS
2. בחירה של הטאפלים (השורות) בהם יש את אותם הערכים בעמודות המשותפות בין R לS
3. הטלה, כך שנשאר רק עם העתק אחד של כל עמודה משותפת

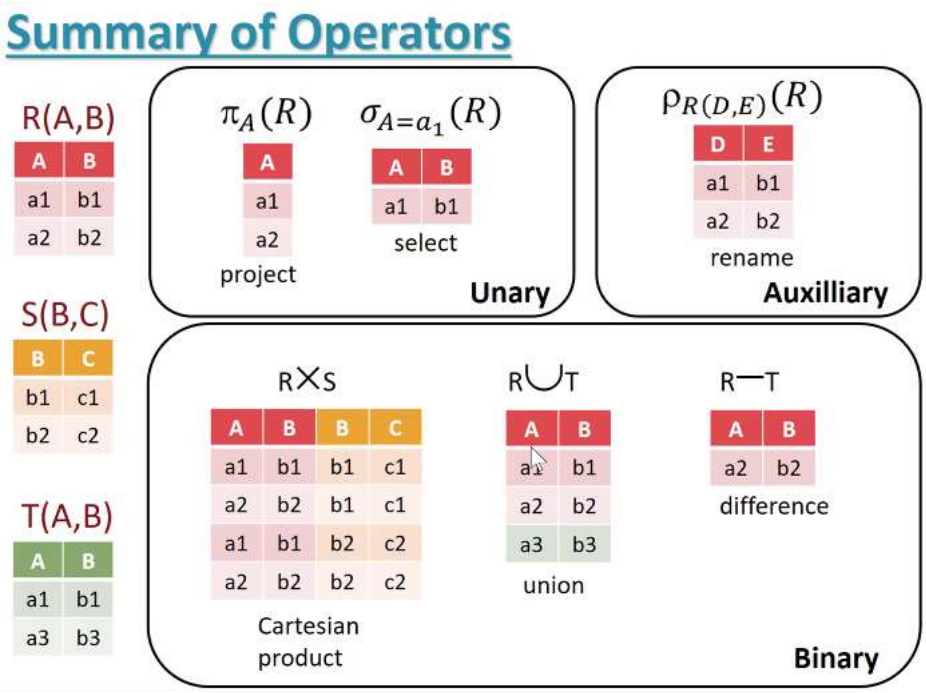
פעולה אסוציאטיבית וקומוטטיבית (כלומר אפשר לכתוב כמה פעולות כאלו בלי סוגריים אם נרצה).
יש שאילתות שקולות. כרגע לא משנה במה נשתמש. בהמשך נרצה לחשוב מה יותר יעיל למערכת הDB לבצע.
חילוק $R \div S$ (או $R \backslash S$)-הסכימה של S צריכה להיות מוכלת בסכימה של R. העמודות שיופיעו בתוצאה הן אלו שרק בR ולא בS. השורות $(a_1, \dots, a_n)$ שיופיעו בתוצאה הן אלו שעבור כל $(b_1, \dots, b_m)$ בS, $(a_1, \dots, a_n, b_1, \dots, b_m)$ בR. המילה all מרמזת לשימוש בפעולה הזו.
חשוב לעשות הטלות לפני פעולת החילוק!

<u>שקילות בין ביטויים אלגבריים</u>

ביטויים הם שקולים אם הם תמיד מחזירים את אותו דבר. $E_1 \equiv E_2$
כלומר ביטויים הם **שקולים** אם"ם לא משנה מה תוכן היחס, הם תמיד מחזירים את אותה התוצאה.
לפעמים משתמשים בשקילות כדי להחליט מי יותר יעיל לחישוב.
<u>הוכחת שקילות</u>- שימוש בהגדרות של הפעולות האלגבריות שהגדרנו.
כשמשתמשים בחילוק, חשוב לשים לב אם צריך לעשות הטלה לפני פעולת החילוק (ולעשות לפני ולא אחרי, זה מוביל לתוצאה שונה).
השימוש בכל זה במערכת הDB הוא על מנת לייעל חישוב של ביטויים, כדי לצמצם תוצאות ביניים של חישובים (לייצר יחסים קטנים בדרך). עושים את זה ע"י (תוך שמירה על שקילות):

- הפעלה מוקדמת של פעולות בחירה
- הפעלה מוקדמת של פעולות הטלה
- לשים לב לסדר ביצוע הצירופים

תרגול 2
תרגום ליחסים.



אופרטורים אונאריים- הטלה בוחרת עמודות |. בחירה בוחרת שורות _.

אופרטורים בינאריים- מכפלה קרטזית יכולה להתבצע על כל טבלאות שהן, בניגוד לאיחוד ולחיסור.

איחוד גם מוחק כפילויות.

אופרטור עזר- שינוי שם.

כל האופרטורים לא משנים את הטבלה המקורית אלא מחזירים עותק שלה.

אופרטור נוסף- צירוף טבעי.

לשאול את עצמנו- איזו טבלה צריכים? מה התנאי? איזה עמודות צריך להחזיר?

הרצאה 3 - SQL

אי תלות של אופרטורים באלגברה רלציונית

האם אפשר לבטא את אחד האופרטורים בעזרת האחרים? לא. אם נחסיר את אחת מהפעולות מהשפה, נקבל שפה שיכולה לבטא פחות דברים.

הוכחת אי תלות של פעולת ההטלה:

יהא R(A,B) קשר, ויהא E ביטוי רלציוני שמשתמש בR וב $\cup,-,\times,\sigma$. אזי $\pi_A R \not\equiv E$.

נראה באינדקוציה על הגודל של E- כמות הפעולות באלגברה של E. נראה שהתוצאה של E תמיד תכלול לפחות 2 עמודות, לעומת ההטלה שתגרום שתהיה רק עמודה אחת. זה מראה את התכונה המיוחדת של ההטלה- גורמת לסכימה לקטון.

בסיס האינדוקציה: n=0, אזי בהכרח E=R ויש בתוצאה 2 עמודות.

הנחת האינדוקציה: אם יש בE לכל היותר N אופרטורים, אזי E מחושב לביטוי עם לפחות 2 עמודות.

צעד האינדוקציה: כל אחת מהאופציות לפעולות הן ארבעת אלו, שבכל אחת מהן מספר השורות לא קטן.

- $E = \sigma_\phi E_1$
- $E = E_1 \cup E_2$
- $E = E_1 - E_2$
- $E = E_1 \times E_2$

עבור הוכחת אי תלות של שאר האופרטורים:

מכפלה- מגדילה את גודל הסכימה.
איחוד- מגדילה את כמות השורות בתוצאה.
מינוס- לא מונוטונית (עבור R-S, אם נגדיל את S התוצאה תקטן).
בחירה- מסתכלת על תוכן השורות, בוחרת סימטריה/שורות שמקיימות תנאים מסויימים בלבד.

SQL- Structured Query Language

זוהי שפת השאילתות, שמטרתה להיות כמה שיותר פשוטה ודומה לשפה טבעית.
קבלת טבלאות כאינפוט, ומחזירה טבלה כפלט.
התוצאה מודפסת למסך, אין שום תופעת לוואי למסד מהשאילתה (לא משנה את התוכן של הטבלאות עצמן). אם לא ביקשנו במפורש, התוצאה לא נשמרת בשום מקום.

זוהי שפה דקלרטיבית- מצהירים מה רוצים לראות בתוצאה ולא איך לחשב במפורש. המסד יחשב בצורה שבה נח לו.

הבדלים בין אלגברה רלציונית לSQL:

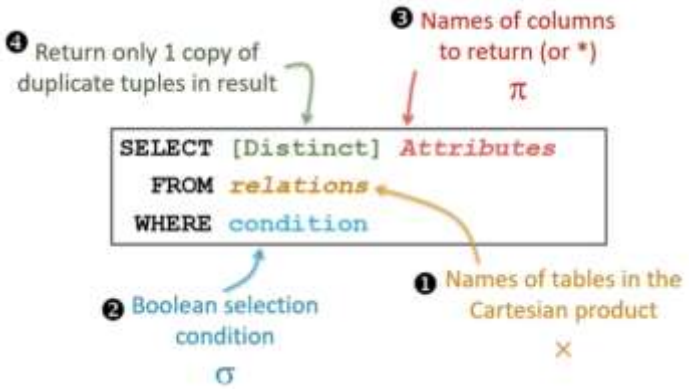
RA	SQL
מניחים שיש סט של שורות (בלי חזרות)	כל שורה יכולה להופיע מספר פעמים
אין ערכי null	הטבלאות יכולות להכיל ערכי null
מופעלת לפי לוגיקה ב2 ערכים (אמת/שקר)	לוגיקה ב3 ערכים (אמת/שקר/לא ידוע)
לא טיורינג-שלמה (לא שפה שבעזרתה אפשר לכתוב כל תוכנית אפשרית)	שפה טיורינג-שלמה (יש פיצ'רים נוספים שאין בRA)

צורה בסיסית של שאילתות SQL:
SELECT Attributes
FROM relations
WHERE condition

רכיבים בסיסים של שאילתות SQL

SELECT, FROM - חובה
WHERE, GROUP BY, HAVING, ORDER BY - לא חובה, ואם הם מופיעים- זה צריך להיות בסדר הזה.
השפה case insensitive, חוץ מאשר ביטוי בתוך מרכאות.
כל שאילתה חייבת להסתיים ב;.

כל אחד מהחלקים מתאים לפעולה אחרת באלגברה, והרעיון (לא מחייב) של הסדר בו הדברים מחושבים:



SELECT

<u>חישוב הטלה</u> SELECT x FROM y; כי שזה יהיה באמת הטלה, צריך להוסיף distinct.
אם בבחירה יש שורה עם מפתח, לא כדאי לכתוב distinct (כי אין צורך וזה מכביד על המסד).
<u>חישוב בחירה</u> SELECT * FROM y WHERE c;.
ניתן כאן גם להוסיף **חישוב על הערכים שיהיה בעמודות**- במקום * SELECT נכתוב SELECT c1/c2 לדוגמא- וזה יציג בכל שורה את הערך בעמודה c1 לחלק לערך בעמודה c2. בהצגה, הוא כותב את שמות המעודות שקיבל- אבל שם עמודה לא יכוללהיות עם / ולכן יציג משהומוזר. ניתן לקבוע גם את שם העמודה שנרצה שתופיע כך: SELECT c1/c2 AS c12.
אפשר גם **להפעיל פונקציות** על ערכים בעמודות. שוב, שם העמודה לא יהיה מוגדר היטב כי לא יכולה להופיע * בשם עמודה. אם נרצה שם בעל משמעות, נכתוב AS.
ניתן גם **להגדיר ערך** לעמודה חדשה.

WHERE

ניתן לשים **תנאים מורכבים**, שכוללים תנאים נומריים, לוגיים, בדיקת ערך null ותבניות בהן ניתן להשוות בין סטרינגים. ההשוואה קורית בעזרת LIKE שמשווה לביטוי רגולרי, ויכולות הביטוי שלו מוגבלות- ניתן לציין איזה אותיות רוצים שיהיו, % מציין 0 או יותר תווים כלשהם, _ מציין תו אחד כלשהו.

ORDER BY

ציון באיזה סדר נרצה לראות את התוצאה. אם לא נציין- מה שיוצא אני מרוצה. המיון הוא בסדר עולה לפי הדיפולט שלו. אפשר לציין במפורש אם רוצים מיון בסדר עולה ASC או יורד DESC.

FROM

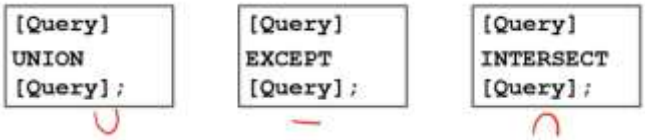
<u>חישוב מכפלה קרטזית</u> ציון כמה יחסים בFROM יכפיל אותם.
כדי להתייחס לשדה שיש לו את אותו שם על פני כמה טבלאות מהן אנחנו בוחרים, נשתמש בtableName.name, ואפשר להשתמש בזה גם עבור שמות שדות יחודים. אפשר גם להשתמש בaliasing- נכתוב FROM TableName aliasName, ונוכל להשתמש בשם שנתנו בהמשך.

דרכים מקוצרות לכתוב **תנאי צירוף**
<u>חישוב צירוף על תנאי</u> כך FROM Sailors S1 **INNER JOIN** Saliors S2 on **(condition)**.
<u>חישוב צירוף טבעי</u> כך FROM Sailors S **NATURAL JOIN** Reserves R.

DISTINCT

מחיקת כפילויות מהתוצאה. לפעמים לא יכולות להיות כפילויות, ואז חבל לכתוב את זה- כי המימוש במערכת הDB זה ע"י מיון התוצאה וחיפוש שורות כפולות, או בניית hashtable שבעזרתו הוא מחפש כפילויות.

<u>פעולות על קבוצות</u>



במסדי נתונים שונים, לא נכתוב EXCEPT אלא MINUS.
זו פעולה תקינה אם השאילתות מחזירות את אותה כמות העמודות, ויש להם טיפוס מתאימים (בדיוק אותו דבר או מספיק דומה- int וfloat).
שמות העמודות בתוצאה נקבעות לפי מה שכתוב בשאילתה העליונה.

בכל שלושת השאילתות, לא יהיו כפילויות בתוצאה. אם נרצה כפילויות, נצטרף לציין במפורש ליד פעולת הקבוצה את המילה ALL. בunion all הכמות תהיה הכמות בראשון+ הכמות בשני. אבל אף מערכת DB לא תומכת באמת בexcept all ובINTERSECT ALL.

צריך להיות זהירים כשעושים EXCEPT או INTERSECT על שדה שהוא לא מפתח, כי זה לא יחזיר בדיוק את מה שרצינו.

<u>הרכבה</u>

חסר לנו כדי לקבל את כל כח הביטוי של האלגברה.
אין דרך מקוצרת לבטא חילוק, לכן חייבים דרך לכתוב שאילתות עם הרבה תתי-שאילתות.
SELECT, FROM, WHERE, HAVING יכולות להכיל תתי שאילתות. GROUP BY, ORDER BY לא יכולות.
תתי-שאילתות יכולות להיות מתואמות- להתייחס לשאילתה שמוגדרת מחוץ אליה.

תת שאילתה בWHERE

תת שאילתה כחלק מביטוי בוליאני- שימוש בשאילתה כדי להגיד תנאי בוליאני מורכב.
c **IN** (Query) כשבהתחלה יש משתנה או ערך אחד, IN או NOT IN בתת השאילתה, שצריכה להכיל עמודה אחת בדיוק. עבור כל שורה, תת השאילתה מחושבת עליה (אלא אם מערכת הDB עושה את זה יעיל).
c **< ANY** (Query) כשאפשר לכתוב ANY או ALL. הערך הוא אמת אם הערך של C עונה על התנאי יחסית לANY או יחסית לALL התוצאות של השאילתה.
EXISTS (Query) מחזיר אמת אם תת השאילתה לא ריקה, ועבור NOT EXISTS יחזיר אמת אם תת השאילתה ריקה. כלור האם השורה הספציפית בתוך/לא בתוך תת השאילתה המבוקשת.

<u>פעולת החילוק</u>

גרסה 1: לא קיימת ספינה שלא נמצאת ברשימה הספינות שהימאי הזמין.

```
SELECT sid
FROM Sailors S
WHERE NOT EXISTS
      (SELECT B.bid
       FROM Boats B
       WHERE B.bid NOT IN
            (SELECT R.bid
             FROM Reserves R
             WHERE R.sid = S.sid));
```

Sailors who Reserved all Boats

גרסה 2: לא קיימת ספינה כך שלא קיימת הזמנה שלה על שם הימאי.

```
SELECT S.sid
  FROM Sailors S
 WHERE NOT EXISTS(
          SELECT B.bid
            FROM Boats B
           WHERE NOT EXISTS(
                    SELECT R.bid
                      FROM Reserves R
                     WHERE R.bid=B.bid and
                           R.sid=S.sid))
```

Sailors who Reserved all Boats

גרסה 3: לא קיימת ספינה ברשימת הספינות שהימאי לא הזמין.

Sailors who Reserved all Boats

```
SELECT S.sid
  FROM Sailors S
 WHERE NOT EXISTS((SELECT B.bid
                     FROM Boats B)
                   EXCEPT
                  (SELECT R.bid
                     FROM Reserves R
                    WHERE R.sid = S.sid));
```

תת שאילתה בFROM

חייב להינתן כalias.

יכולים להיות עוד מקומות עם תתי שאילתיות, אבל רובם קורים עם פעולות הקבצה- נלמד קודם את זה ואז נשלב את זה עם שאילתות בSELECT ובHAVING.

תרגול 3

SQL מורכב מ3 closeים, שפועלים לפי הסדר הזה: from, where, select.

הרצאה 4 - SQL

<u>פונקציות הקבצה Aggregation</u>

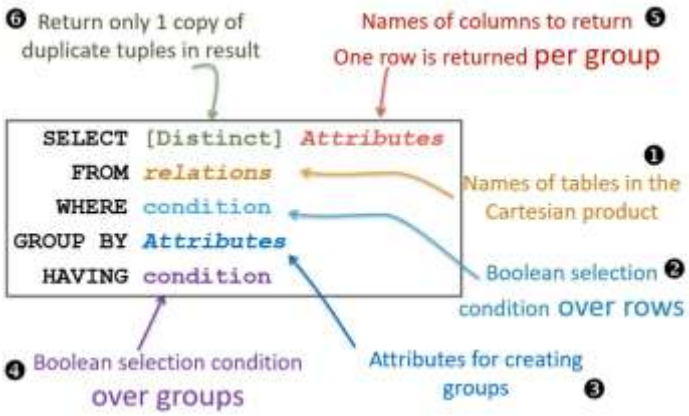
פונקציה שמקבלת מולטי-סט של ערכים ומחזירה ערך בודד. לרוב נראה אותן כחלק מהSELECT.
לדוגמא- סכום. לא צריך לקבל קבוצה אלא מולטי-סט, כי ערך יכול לחזור על עצמו.
בסטנדרט של SQL מוגדרות פונקציות הקבצה, ולרוב כל מסד נתונים תומך בפונקציות נוספות.

- COUNT(*) סופר את מספר השורות
- COUNT(A) סופר כמה ערכים יש בA, COUNT(DISTINCT A) סופר כמה ערכים שונים יש בA
- SUM([DISTINCT] A) סוכם את הערכים בA, או עם DISTINCT סופר רק את הערכים השונים
- AVG([DISTINCT] A) ממוצע של הערכים בA
- MIN(A) המינימום בA
- MAX(A) המקסימום בA

הפעולות מוגדרות היטב רק כשהן מעל עמודה שהגיוני להגדיר אותן עליה- SUM וAVG מעל מספרים, MIN וMAX לא רק על מספרים- על כל שדה שמוגדרת עליו השוואה מסוימת (לדוגמא ראשונה לפי סדר הא"ב).

כשנרצה להפעיל אותן על כל הטבלה

GROUP BY - בעזרתו מזהים קבוצות. כשאין כזה, מתייחסים לכל הטבלה כקבוצה אחת.
HAVING הוא תנאי בוליאני מעל הקבוצות שהגדרנו, אם הוא מקבל ערך אמת משתמשים בקבוצה על מנת לייצר שורה אחת בפלט (ואם מקבל ערך שקר זורקים את הקבוצה).
מתחילים מלחשב את הWHERE, ואז GROUP BY ולבסוף HAVING.
סדר החישוב סה"כ:



זה עוזר להבין איזה סוג שאילתות הן חוקיות ואילו הן לא הגיוניות.

כללים בסיסיים שלמדנו משגיאות בשאילתות:

- כל שדה בSELECT שלא נמצא בפונקציה הקבצה חייב להופיע בGROUP BY (כדי שכל השורות בקבוצה יקבלו את אותו הערך)
- כל שדה בHAVING שלא נמצא בפונקציה הקבצה חייב להופיע בGROUP BY (כדי שכל השורות בקבוצה יקבלו את אותו הערך)
- אי אפשר להפעיל פונקציית הקבצה על פונקציית הקבצה (כי מצפות לקבל כמה ערכים ולהוציא ערך בודד- ברגע שהפעלנו על ערך בודד זה חסר משמעות)
- HAVING צריך לחשב תנאי בוליאני
- מותר לכתוב תת שאילתה בSELECT אך ורק אם היא מחזירה ערך בודד (או NULL)

חישוב (דומה ל)חילוק

בעזרת תת שאילתה בHAVING.

כל עוד הטבלאות לא ריקות, ייתן את אותה התוצאה כמו פעולת החילוק.

<u>ערכי NULL</u>

נדבר על איך השאילתות מחושבות כאשר יש ערכי NULL.

ערכי NULL יכולים להופיע בתוך הטבלה.

בSQL משתמשים בלוגיקה של 3 ערכים- אמת, שקר, לא ידוע. אלו טבלאות האמת:

A	B	A and B	A or B
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

A	Not A
True	False
False	True
Unknown	Unknown

בWHERE רק שורות להן יש ערך אמת משמשות כדי לייצר את התוצאה.

התנהגות NULL בביטויים מתמטיים:

1. כל ביטוי מתמטי שמופעל על NULL, מחזיר NULL
2. אופרטור השוואה מחזיר את הערך הבוליאני לא ידוע
3. כדי להבין האם לעמודה יש ערך NULL נבדוק x IS NULL או x IS NOT NULL – יחזיר אמת או שקר בהתאם לבדיקה

התנהגות NULL בפונקציות הקבצה:

- COUNT(*) סופר את מספר השורות – לא משנה אם יש או אין NULL
- COUNT([DISTINCT] A) סופר כמה ערכים יש בA– בהם A לא NULL. אם כל השורות של A הם NULL, יחזיר 0
- SUM([DISTINCT] A), AVG([DISTINCT] A), MIN(A), MAX(A) – מתעלמים מערכי NULL. אם כל הערכים הם NULL, יחזירו NULL
- בהשוואה בין שורות (בGROUP BY ובDISTINCT), שורות זהות עם NULL באותם מקומות- מתלכדות

יש שורות שיכולות להעלם אחרי פעולה של צירוף טבעי. לפעמים נרצה שהן יישארו. בשביל להשאיר נשתמש באחת מהפעולות:

- NATUARAL LEFT OUTER JOIN – בתוצאה יהיו:
  - כל השורות של הצירוף של הטבלה השמאלית והימנית
  - כל שורה בטבלה השמאלית שלא הייתה לה שורה מתאימה בטבלה הימנית - פשוט נקבל אותה עם ערכי NULL בעמודות האלו
- יש גם NATUARAL RIGHT OUTER JOIN – בתוצאה יהיו:
  - כל השורות של הצירוף של הטבלה השמאלית והימנית
  - כל שורה בטבלה הימנית שלא הייתה לה שורה מתאימה בטבלה השמאלית - פשוט נקבל אותה עם ערכי NULL בעמודות האלו
- NATUARAL FULL OUTER JOIN – בתוצאה יהיו:
  - כל השורות של הצירוף של הטבלה השמאלית והימנית
  - כל שורה בטבלה הימנית שלא הייתה לה שורה מתאימה בטבלה השמאלית - פשוט נקבל אותה עם ערכי NULL בעמודות האלו
  - כל שורה בטבלה השמאלית שלא הייתה לה שורה מתאימה בטבלה הימנית - פשוט נקבל אותה עם ערכי NULL בעמודות האלו

<u>פיצ'רים נוספים</u>

Modifications – איך עושים שינויים לשורות שכבר נמצאות בDB

<u>ראינו איך עושים הכנסות INSERT</u>, ושאם לא מכניסים מפתח- מערכת הDB תזרוק שגיאה (ברגיל בשרוצים להפר אילוץ). ראינו גם איך מוחקים טבלה שלמה DROP TABLE TableName;.

אם נרצה רק למחוק שורה יחידה מטבלה, נשתמש ב-DELETE FROM TableName WHERE c (או בלי תנאי WHERE אם נרצה למחוק את כל השורות).

DROP TABLE מוחק את הטבלה כולה (את ההגדרה שלה, הטבלה כבר לא תהיה קיימת),

DELETE יכולה למחוק את הנתונים אבל להשאיר את הטבלה עצמה ככה שנוכל להכניס לה נתונים אחרים בהמשך.

בשביל לעדכן טבלה נשתמש ב-UPDATE TableName SET s.old = s.new WHERE c.

View מבט

טבלה וירטואלית שמוגדרת ע"י שאילתה. דומה לטבלה כי אפשר להשתמש בה בכל מקום שבו משתמים בהגדרה של טבלה (בפרט בFROM), אבל לא באמת קיימת כטבלה אלא כמעין MACRO- שאילתה שנשמרת ע"י הDB. כל פעם שמתייחסים לשם של הVIEW, השאילתה מחושבת מחדש.

מגדירים בעזרת CREATE VIEW <view-name> AS <query>;

שינויים בטבלה יעודכנו גם בVIEW (מהאופן בו הוא עובד- מחושב במקום).

למה שנשתמש בVIEW?

1. אם יש שאילתה שמשתמשים בה הרבה, והיא ארוכה ומורכבת לכתיבה. כמו refactoring של קוד.
2. אבטחת מידע- אפשר לתת למשתמשים יכולת לגשת לתוצאה של VIEW במקום לגשת לטבלה שלמה.

נתינת הרשאות זה בעזרת grant ... to user; - וניתן לתת הרשאות לVIEW מתאים.

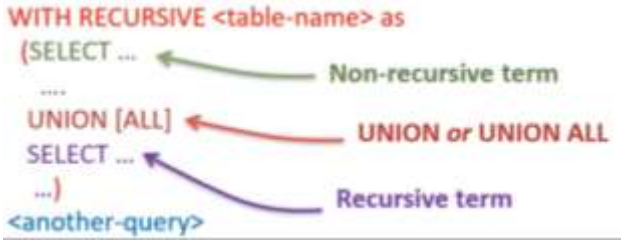
אפשר לפעמים **לשנות תוכן של VIEW**- והשינויים מבוצעים על הטבלה שעליה מוגדר הVIEW.
זה אפשרי רק כשהוא מוגדר להיות updateable – צריך לקיים **תנאים שיאפשרו לו לקבל עדכונים**:

1. השאילתה של הVIEW מוגדרת רק מטבלה אחת בFROM
2. אסור שהיא תכיל ברמה העליונה group by, having, distinct, union, intersect, except
3. בSELECT אסור שתופיע פונקציית הקבצה

אפשר גם לעשות INSERT לVIEW, שמכניס לטבלה המקורית – לפעמים אפשר להכניס בכה שיכנס רק לטבלה המקורית ולא לVIEW, אפשר להגדיר אילוצים על ה DBכך שלא יאפשר הכנסות מהסוג הזה.

רקורסיה בתוך שאילתות

טבלה זמנית היא טבלה שמגדירים לצורך חישוב שאילתה מסוימת, ובסוף החישוב היא נמחקת מהDB. משתמשים במילה WITH <table-name> as (<query>) <another-query>. בניגוד לVIEW, כאן הטבלה נוצרת באותו רגע, וקיימת רק עד סיום השאילתה שמשתמשת בה. קודם מחושבת הטבלה הזמנית, ולאחר מכן השאילתה איתה.
כדי להשתמש ברקורסיה, צריך להגדיר במפורש שטבלה יכולה להיות רקורסיבית WITH RECURSIVE.



שלבים בביצוע שאילתות רקורסיביות:

1. שלב ראשון
   a. חישוב החלק הלא-רקורסיבי
   b. אם כתוב UNION, מוחק כפילויות
   c. את השורות שמים ב2 טבלאות זמניות
      i. הטבלה הסופית
      ii. טבלת העבודה
2. שלב שני- החלק הרקורסיבי
   a. כל עוד טבלת העבודה לא ריקה, מחשבים את החלק הרקורסיבי. כשהחלק מתייחס לטבלה- לוקחים את המידע מטבלת העבודה
   b. אם יש UNION, מוחקים כפילויות (במה שנוצר וגם עם התוצאות הקודמות)
   c. מוסיפים את השורות שהתקבלו בטבלה הסופית, ומחליפים את התוכן של טבלת העבודה בשורות שהתקבלו

אם אנחנו משתמשים בunion all, זה יכול לגרום לכך שהשאילתה לא תסתיים לעולם (רקורסיה אינסופית). הDB במקרה כזה יעלה שגיאה בסגנון של stack overflow.

תרגול 4

תתי שאילתות בWHERE.

HAVING זה כמו WHERE רק לקבוצות ולא לשורות.

הרצאה 5 – Indexes (Evaluation) חישוב פעולת בחירה- תנאי בwhere

מה המסד עושה מאחורי הקלעים?

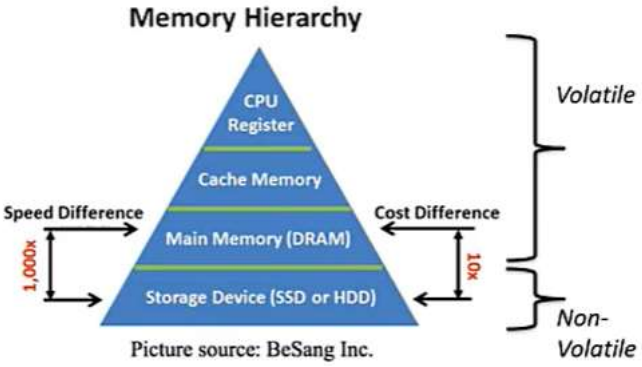
SQL היא שפה דקלרטיבית- בוחרת את המימוש הספציפי בעצמה.

בהמשך נדבר על איך לגרום למסד לעשות את מה שאנחנו רוצים.

השאלות הגדולות שצריך לענות עליהם כדי להבין איך מתרחש עיבוד השאילתות במערכת הDB:

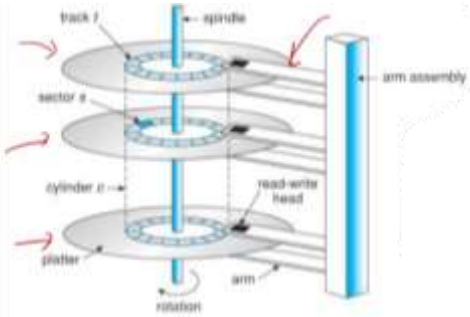
1. איך בכלל שומרים את המידע? מה קורה כשמכניסים שורות לטבלה?
2. מה קורה כשכותבים שאילתה?
3. מה גורם לחישוב השאילתה להיות מהיר או איטי?

איפה המידע שמור? בזיכרון. במחשב יש כמה סוגים של זיכרון, כאן נתייחס רק לזיכרון בדיסק ולזיכרון הראשי.



דיסק	זיכרון ראשי
גדול יותר	קטן יותר
זול יותר	יקר יותר לייצור ולקנייה
Non-volatile כלומר המידע נשמר	Volatile כלומר התוכן יכול ללכת לאיבוד (בזמן כיבוי המחשב)

בגלל שהמידע בDB חייב להישמר לאורך זמן, הוא חייב להישמר בדיסק (השכבה הנמוכה בציור).



איך נראה הדיסק? זהו HDD, שבנוי מראש קורא ומדיסקים. כשרוצים לקרוא משהו מהדיסק, צריך לסובב את הדיסק המתאים ולהזיז את הראש הקורא קדימה/אחורה כדי שיגיע למקום הנכון. זה לוקח זמן.

הסוגים הנפוצים ביותר של זיכרון דיסק הם HDD (יותר זול משמעותית אבל יותר איטי) וSSD.

בSSD הגישה מהירה יותר. בכל מקרה, שני הסוגים הרבה יותר איטיים מהזיכרון הראשי.

הזיכרון הראשי גם משמש תפקיד חשוב כאן- כדי להשתמש במידע ולעבד אותו חייבים להעביר אותו מהדיסק לזיכרון המרכזי. את המידע מעבירים מהדיסק לזיכרון הראשי ביחידות של דפים/בלוקים (4K לרוב). בדף יכולים להיות הרבה פרטי מידע, והם יעברו ביחד (אי אפשר להעביר חלק של דף או דף וחצי).

הזיכרון הראשי מנוהל ע"י הbuffer manager, שמחליט איפה למקם מידע שקיבל. לפעמים הזיכרון הראשי יהיה מלא, ואז הוא יצטרך להחליט את מה לדרוס בתוכו (כי הדיסק גדול ממנו בהרבה).

בדיסק המידע נשמר בקבצים, שזה בעצם סדרה של בלוקים. אם נרצה לקרוא קובץ שלם, נצטרך לקרוא בלוק בלוק ממנו לזיכרון הראשי.

לרוב מערכת הDB לא מפצלת טאפלים (שורות) בין בלוקים. כדי לחשב בכמה בלוקים שמורות T שורות, נעשה את החישוב הבא: $\left\lceil T / \left\lfloor \frac{size\ of\ block}{size\ of\ row} \right\rfloor \right\rceil$.

I/O complexity

הרבה פעמים הזמן שלוקח להעביר את המידע מהדיסק לזיכרון הראשי הוא משמעותית יותר ארוך מזמן העיבוד שלו בזיכרון הראשי (החישובים בדרך כלל מהירים). לכן מקובל לנהוג במודל חישובי שונה ממה שאנחנו רגילים- במקום למדוד פעולות, מודדים כמה פעולות I/O עושים מהדיסק במהלך התוכנית (כמה בלוקים קראנו מהדיסק וכתבנו לדיסק). זו הפשטה, באופן מעשי העלות תלויה גם במיקום של הבלוקים בדיסק.

לא נספור את כמות פעולות הכתיבה כדי לכתוב את התוצאה הסופית, כי תוצאת השאילתה היא רק משהו שרואים במסך ולא נכתבת לדיסק אם לא ביקשנו את זה במפורש.

נדון על איך עושים פעולות יעלות על מידע שלא נכנס כולו לזיכרון הראשי.

<u>ניהול קבצים וגישה יעילה לטבלאות</u>

טבלאות שמורות בקובץ ערימה heap- אוסף בלוקים שמוסיפים לסופו את השורות החדשות.

נניח שכל השורות בטבלה לוקחות את אותו המקום בדיסק.

כמה זמן ייקח לעשות פעולות בסיסיות על הערימה? נניח שהטבלה שמורה בN בלוקים

- מציאת שורה עם ערך sid מסוים- במקרה הגרוע נקרא את כל הבלוקים- N פעולות
- הוספת שורה- נצטרך לרשום לתוך הבלוק האחרון או לפתוח בלוק חדש אם נגמר בו המקום- צריך לקרוא את הבלוק לזיכרון, להוסיף את השורה לבלוק ולכתוב אותו לדיסק (כי אי אפשר לכתוב/לקרוא ביחידות ששונות מבלוק שלם)- 2 פעולות
  - אם לטבלה יש אילוץ מפתח צריך לוודא שלא שברנו אותו, וזה ייקח יותר מ2 פעולות כי צריך לקרוא את כל הבלוקים ולוודא שהאילוץ נשאר- N+1 פעולות
- מחיקת שורה- צריך למצוא את הבלוק שמכיל אותה (יעלה N פעולות), למחוק את השורה מהבלוק ולכתוב אותה בחזרה לדיסק- N+1 פעולות
  - השארנו אזור ריק בבלוק, אם זה לא רצוי צריך לעשות עוד עבודה כדי לדאוג לניצול מקסימלי של הזיכרון

אז שמירה בקובץ ערימה גורמת לזה שכמעט כל פעולה מצריכה מאיתנו לקרוא את כל הקובץ.

בגלל שהמידע גדול קריאה של כל הקובץ זו פעולה איטית מדי. נראה שפתרון לזה יהיה לשמור את המידע ממוין, ואז אפשר לעשות חיפוש יעיל כדי למצוא שורה מסוימת.

הבעיה היא שעבודה עם קבצים ממוינים היא לא יעילה מבחינת הכנסה של מידע לטבלה, כי צריך להכניס את השורה "לאמצע" ולהזיז את כל השורות הבאות. לכן בפועל הקבצים לא שמורים בצורה ממוינת, חוץ משני מקרים יוצאי דופן:

- כשיש טבלאות שרק מוסיפים להן מידע, ויש מיון טבעי שנוצר לפי סדר ההכנסה- log של פעילות המערכת שממוין לפי זמן
- index organized files, נדבר בהמשך הקורס

בכל זאת נרצה לגשת בצורה יעילה- איך נעשה את זה?

מבנה index

מבנה נתונים שמוגדר מעל הטבלה, שנותן דרך יעילה למצוא שורות שמתעניינים בהן בלי לעבור על כל הטבלה. יכול להיות hash table או עץ חיפוש בינארי, שהמבנה מצביע על השורות בטבלאות. מפתח החיפוש במבנה הוא השדה שלפיו נרצה להיות מסוגלים לגשת לטבלה.

המצביעים הם זוג של ערכים: הכתובת של הבלוק, מספר השורה בתוך הבלוק.

התקווה שלנו היא שאת המבנה הזה ניתן לקרוא באופן יעיל מהזיכרון.
מבנה אינדקס רגיל הוא:

- לא בינארי
- כל הכתובות יהיו בעלים (ולא לאורך הדרך)
- פיצול של ערכים כפולים

עץ B+

מערכת הDB תומכת בהרבה סוגים שונים של אינדקס, אנחנו נדבר על הסוג הפופולרי ביותר. זהו עץ, עם שורש, עלים וקודקודים פנימיים. העץ מאוזן- המרחק מהשורש לכל אחד מהעלים זהה. העלים הם רשימה משורשרת דו כיוונית (מכל עלה אפשר להמשיך לעלים הבאים בתור או לחזור אחורה לעלה הקודם). בכל קודקוד יש מספר ערכים ממוינים, וכל העלים גם כן ממוינים (מאפשר חיפוש יעיל בעץ). בעץ יש מפתח חיפוש (לדוגמא גיל, כתוב בכל אחד מהקודקודים) וערכים+ שורות שעליהן מצביעים (כתוב בעלים). אותו ערך יכול להופיע כמה פעמים, בכל פעם עם שורה אחרת שהוא מצביע עליה.
לכל שורה בטבלה יש ערך מתאים בעלה (את ערך מפתח החיפוש המתאים ואת הכתובת).
איך מחפשים ערך ספציפי בעץ כזה? מכל קודקוד יש 3 צלעות- השמאלית מובילה לערכים שקטנים או שווים לערך השמאלי, הימנית מובילה לערכים שגדולים מהערך הימני, והאמצעי לערכים שביניהם.
תמיד נתחיל לחפש דרך המצביע הראשון שיכול להתאים וכך מובטח לנו שנמצא את הערך. אם יש כמה מופעים, נמצא אותם ע"י כך שנמשיך ימינה בשכבת העלים. מציאת עלה כזה היא בעלות של גובה העץ.
לעץ B+ בעל דרגת פיצול d יש את התכונות הבאות:

1. כל העלים באותה רמה (מאוזן)
2. לכל קודקוד יש לכל היותר d ילדים
3. לכל קודקוד מלבד השורש יהיו לפחות $\left\lceil \frac{d}{2} \right\rceil$ ילדים
4. לכל קודקוד שהוא לא עלה שיש לו k ילדים, יהיו k-1 ערכי חיפוש
5. כל המפתחות בתוך הקודקוד מסודרים בסדר עולה

איך נבחרת דרגת הפיצול? אין לנו שליטה עליה, אלא מערכת הDB בוחרת בעצמה כך:

- עץ הB+ נשמר בדיסק כמו הטבלאות (בבלוקים)
- בכל פעם שעוברים דרך מצביע בעץ צריך להעלות את הבלוק המתאים מהדיסק- מצביע הוא בעצם כתובת של בלוק בדיסק שבה הקודקוד הבא נמצא
- אם יש דרגת פיצול גבוהה יותר, העץ יהיה פחות עמוק ויהיו פחות קריאות דיסק (כי העבודה דורשת לרדת בעומק העץ)
- מערכת הDB תבחר את רמת הפיצול המקסימלית שתאפשר לה עדיין לשמור כל קודקוד בבלוק אחד

עלות הI/O של חיפוש בעץ מורכבת משני דברים:

- עומק העץ, כדי למצוא את העלה הראשון הרלוונטי
- טיול על העלים, בעלות שתלויה בכמות המספרים של הערך הרלוונטי ובמספר הערכים שנכנסים לכל עלה

העץ חייב להישאר מאוזן תמיד, והתחזוק צריך להיות יעיל. את כל זה אפשר לעשות בזמן שפרופורציונלי לעומק העץ.

נוסחאות

- כדי למצוא את דרגת הפיצול d:
  - d ילדים = $(size\ of\ pointer) \cdot d$
  - d-1 ערכי חיפוש = $(search\ key\ size) \cdot (d-1)$
  - $(size\ of\ pointer) \cdot d + (search\ key\ size) \cdot (d-1) \leq block\ size$

- נוסחא יחידה: $\left\lfloor \frac{block\ size+search\ key\ size}{pointer\ size+search\ key\ size} \right\rfloor$ (הגדלים נמדדים בbytes)
- כדי למצוא את מספר העלים עבור t מספר הטאפלים (שורות) בטבלה:
  - מספר העלים המינימלי $\left\lceil \frac{t}{d} \right\rceil$
  - מספר העלים המקסימלי $\left\lceil \frac{2t}{d} \right\rceil$

<u>עלות חישוב שאילתה בעזרת אינדקס</u>

יש 3 דרכים שונות להשתמש במבנה אינדקס במהלך חישוב שאילתה (ואפשר גם לשלב ביניהן):

1. INDEX UNIQUE SCAN – לכל היותר צריך לעשות מעבר אחד מהשורש לעלה אחד (בלי להמשיך ימינה או שמאלה לעלים נוספים)
   a. לדוגמא מעבר לפי מפתח
   b. נשתמש עבור SELECT DISTINCIT "true" FROM Sailors WHERE age > 58, כי מעוניינים לדעת האם קיים ימאי שגילו מעל 58- נצטרך רק לדעת אם יש אחד כזה
2. INDEX RANGE SCAN – מעבר מהשורש לעלה, וגם צריך להמשיך לעבור ימינה או שמאלה בעלים
   a. לדוגמא מעבר לפי גיל
   b. נשתמש עבור SELECT COUNT(*) FROM Sailors WHERE age > 58, כי מעוניינים לדעת כמה ימאים כאלו יש
3. TABLE ACCESS BY ROWID – בדר"כ מגיע יחד עם אחד מהשניים הקודמים. בהגעה לעלה, צריך ממש להעלות את הערך של השורה מהטבלה (קריאת עוד בלוק מהדיסק)
   a. נשתמש עבור SELECT name FROM Sailors WHERE age > 58, כי מעוניינים לדעת מידע אחר עליהם מהשורה שלהם בטבלה
   b. במקרה הגרוע, כל שורה תהיה בבלוק אחר

נוסחאות

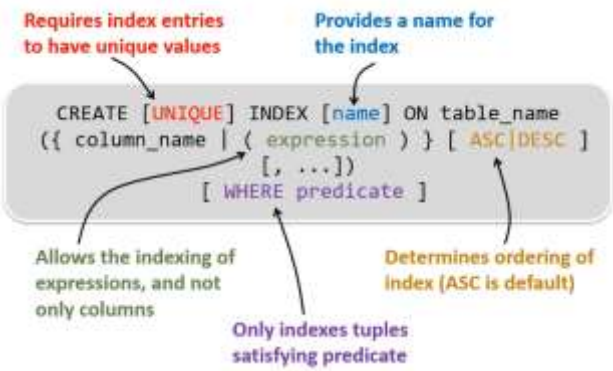
- גובה העץ עבור טבלה עם N שורות ודרגת פיצול d הוא לכל היותר $\left\lceil \log_{\left\lceil \frac{d}{2} \right\rceil} N \right\rceil$
  - כי לכל קודקוד יש לפחות $\left\lceil \frac{d}{2} \right\rceil$ ילדים
- כמות העלים שיכילו ערכים המתאימים לערך אותו אנחנו מחפשים, אם תנאי החיפוש מתאים לm שורות בטבלה ודרגת הפיצול היא d, היא לכל היותר $\left\lceil \frac{m}{\left\lceil \frac{d}{2} \right\rceil - 1} \right\rceil$ (זהו עלות המעבר על העלים)
  - כי $\left\lceil \frac{d}{2} \right\rceil - 1$ זה כמות הערכים המינימלית בכל עלה
- אם נרצה לקרוא ערך אחר מהטבלה עבור כל אחד מהערכים, זה יוסיף לנו m פעולות I/O של קריאת הבלוקים המתאימים

לא תמיד כדאי להשתמש במבנה אינדקס, גם אם יש לנו כזה. לפעמים סריקה של כל הטבלה תהיה זולה יותר. חלק מהייעול של מערכת הDB זה להבין מה יהיה לה זול יותר, ולהשתמש בזה.

<u>יצירת אינדקס (מצד המשתמש)</u>

נשתמש בפקודה CREATE INDEX ON table_name ({column_name, ...}). נדבר בהמשך על איך נראה אינדקס על יותר מעמודה אחת.

יש עוד אופציות להוספה לפקודה, אלו הנפוצות מביניהן:



שם לאינדקס יכול להיות שימושי כדי לבדוק בצורה אינפורמטיבית האם המסד באמת השתמש בו.

יש אינדקסים שנבנים באופן אוטומטי (לא את כולם צריך להגדיר באופן ידני). בכל פעם שמגדירים מפתח על טבלה, מערכת הDB מייצרת אינדקס שבעזרתו היא מוודאה שהמפתח נשאר ייחודי. באופן דומה, היא מייצרת עבור ערכים שהגדרנו אותם כunique (גם עבור סט של ערכים כאלו). אפשר גם להגדיר מה יהיה שם האינדקס בעזרת הכרזה שונה על המפתח לטבלה- CONSTRAINT index_name PRIMARY_KEY(pkey) בסוף הגדרת הטבלה.

איך נדע אם מסד הנתונים בחר להשתמש באינדקס שלנו כדי לייצר שאילתה? נוסיף את הפקודה EXPLAIN לפני השאילתה שלנו- מה שיגרום לכך שמסד הנתונים יכתוב איך הוא הולך לבצע את השאילתה בלי לבצע אותה. בהדפסה שלו, העובדה שכתוב Index Scan using מציינת שמשתמשים באינדקס. הוא גם כותב מה הוא הולך לבדוק (אבל לא את הגישה הנוספת כדי לשאוב מידע אחר מהטבלה המקורית), שיערוך של עלות ההתחלה לביצוע שאילתה (זה מספר יחסי ולא שמציין זמן), שיערוך עלות ביצוע הפעולה, כמות השורות שמעריך שיוחזרו ורוחב השורה- כמה בייט של נתונים הוא צופה שיוחזרו.
אפשר גם לבקש שהשאילתה גם תבוצע- בעזרת EXPLAIN ANALYZE, ואז בנוסף להערכות גם נכתב כמה היו העלויות בפועל. ניתן גם לראות את זמן התכנון לעומת זמן הביצוע במילישניות.
ניתן להשתמש בזה גם כשהשאילתה היא יותר מורכבת, ואז המסד יציין בנוסף למה הוא הולך לבדוק גם את הפילטר הנוסף.

לשים לב

- מבנה אינדקס לא נבנה בזמן שאילת שאילתה, אלא מי שמתכנן את מערכת הDB צריך להחליט איזה אינדקסים לייצר, ומייצר אותם הזמן ייצור הDB או בכל זמן אחר כך שירצה, ולאחר מכן מערכת הDB בזמן חישוב שאילתה יכולה לבחור האם להשתמש באינדקס או לא.
- כל אינדקס נוסף שנייצר מגדיל את כמות הזיכרון שנצרך כדי לשמור את האינדקס (הקצאת זיכרון נוסף), ויש לו עלויות תיחזוק כשעושים שינויים בטבלאות. לכן לא כדאי לייצר המון אינדקסים, אלא רק כאלו שיזרזו את השאילתות הנפוצות ביותר והחשובות ביותר.
- כל עמודה נוספת באינדקס תגדיל את כמות המקום שהוא יצרוך ואת עלות התחזוקה שלו.
- לא לייצר אינדקסים מיותרים!
  - אם ייצרנו על age, rating אין צורך ליצור אחד נוסף על age

<u>אינדקסים על יותר מעמודה אחת – Multicolumn/Concatenated Indexes</u>

המיון הראשוני יהיה לפי השדה הראשון באינדקס, והמיון השניוני יהיה לפי השדה השני בו.
לכן הסדר בו מגדירים את השדות לאינדקס מאוד חשוב.
עבור איזה סוג שאילתות נוכל להשתמש באינדקס כזה כדי לענות עליהן?

- תנאי על השדה הראשון
- אי אפשר ביעילות עבור תנאי על השדה השני
- יעיל ביותר כשיש תנאי שוויון על השדה הראשון באינדקס, ותנאי השוואה (גדול מ/קטן מ) על השדה הבא
- עבור חישוב בSELECT על שדה אחר מזה שהאינדקס עליו, יכול להיות יעיל כדי להוריד את הגישה הנוספת לטבלה

יש אפשרות גם לבנות אינדקס על שדה מסוים, ולהוסיף רק בעלים שדה נוסף כך

CREATE INDEX ON table_name ({column_name, ...}) include (column_name2)

למה זה טוב? חסכון גישה לטבלה עצמה כדי לשלוף שורה מתאימה. יש לזה גם יתרונות על פני בנייה של טבלה אינדקס על פני שתי העמודות- כי מפתח החיפוש הוא רק שדה אחד, ולכן תופס פחות מקום בדיסק מאשר שני שדות. לכן דרגת הפיצול תוכל להיות גבוהה יותר, ונוכל להגיע לעץ עם גובה נמוך יותר כך שהטיול באינדקס הוא מהיר יותר.

תרגול 5

<u>חישובים</u>

- כדי למצוא את דרגת הפיצול d: $\left\lfloor \frac{block\ size + search\ key\ size}{pointer\ size + search\ key\ size} \right\rfloor$
- גובה העץ עבור טבלה עם N שורות ודרגת פיצול d הוא לכל היותר $\left\lceil \log_{\lceil \frac{d}{2} \rceil} N \right\rceil$
  - כי לכל קודקוד יש לפחות $\left\lceil \frac{d}{2} \right\rceil$ ילדים (ולכל היותר d ילדים)
  - זה רק הגובה, לא כולל גישה לעלים (בחישוב צריך להוסיף את מספר העלים שעברנו בהם)
- כמות העלים שיכילו ערכים המתאימים לערך אותו אנחנו מחפשים, אם תנאי החיפוש מתאים לm שורות בטבלה ודרגת הפיצול היא d, היא לכל היותר $\left\lceil \frac{m}{\lceil \frac{d}{2} \rceil - 1} \right\rceil$ (זהו עלות המעבר על העלים)
  - כי $\left\lceil \frac{d}{2} \right\rceil - 1$ זה כמות הערכים המינימלית בכל עלה
- אם יש יותר שורות שנרצה לקרוא מאשר בלוקים בטבלה, האינדקס ממיין את השורות כדי שיקרא כל בלוק לכל היותר פעם אחת

הרצאה 6 - Joins (Evaluation)

<u>הקדמה</u>

נרצה להבין איך חישובים של צירוף טבעי (כל הצירופים שהם) מתבצעים בצורה יעילה.
כל אחת מהטבלאות תופסת כמה בלוקים בדיסק, וגם תוצאת הצירוף תופסת מקום (גם אם כותבים לדיסק וגם אם מייצרים לזיכרון המרכזי ופולטים אחר כך למסך).
כדי לעבד את המידע ולעשות איתו חישובים, צריך להעביר אותו מהדיסק לזיכרון המרכזי ביחידות של בלוקים/דפים.
למה זה מעניין? כי גודל הזיכרון המרכזי משפיע על אופן החישוב:

- אם יש מספיק מקום כדי לשמור את הטבלאות המקוריות+ טבלת הפלט בזיכרון המרכזי, נעשה את זה
- אם יש מספיק מקום כדי לשמור את הטבלאות המקוריות ורק עוד בלוק יחיד, אחרי שבלוק אחד של הפלט נוצר נציג אותו במסך או נעביר אותו לדיסק ונמשיך לייצר את המשך הפלט
- אם אין מספיק מקום- אפשר לקרוא את הטבלה הראשונה כולה וחלק מהטבלה השנייה, ולשמור בלוק אחד לתוצאה (ובה נפעל כמו קודם). אחרי שסיימנו לצרף חלק מהטבלה השנייה, נעלה את החלק הבא (נדרוס את המקום של הבלוק הקודם) ונצרף אותו עם

הטבלה הראשונה באותו האופן – זהו אלגוריתם חיצוני, שפועל תחת ההנחה שאי אפשר להכניס את כל הקלט לזיכרון המרכזי

- אם אין מספיק מקום אפילו לזה- אפשר לקרוא בכל פעם בלוק אחד של כל טבלת הקלט – צריך בשביל זה אלגוריתם חיצוני שיראה לנו איך לעשות את זה ביעילות

נראה 4 אלגוריתמים לחישוב פעולת הJOIN. בכל פעם שמערכת הDB מקבלת שאילתה עם פעולת הJOIN, היא מנסה לחשב כמה יעלה חישוב הצירוף (בפעולות I/O), ובוחרת את האלגוריתם שנראה שיהיה הזול ביותר ומפעילה אותו בפועל. נלמד אותם וננסה להבין איך מערכת הDB מחשבת כמה יעלה לחשב את כל אחד מארבעת האלגוריתמים.

BNL – Block Nested Loops Join

סימונים: B(R) כמות הבלוקים ביחס R, M כמות הבלוקים הפנויים בזיכרון המרכזי.

האלגוריתם פועל בעזרת לולאות מקוננות.

1. מחליטים מי יהיה היחס החיצוני בלולאה ומי הפנימי
2. נקרא את היחס החיצוני ביחידות של M-2 בלוקים
3. נקרא את היחס הפנימי בלוק בלוק, ונעשה צירוף של הבלוק הנוכחי עם הבלוקים שהעלנו ואת התוצאה נכתוב בבלוק של הפלט, ונעשה flash שלו (לדיסק/למסך) כשהוא יתמלא
4. כשסיימנו את כל הבלוקים של היחס הפנימי, נעלה את M-2 הבלוקים הבאים ביחס החיצוני ונחזור לשלב 3

להלן **פסאודו קוד**, בו S היחס החיצוני ו R הפנימי:

R(D, E) S(E,F)
R stored in blocks $b_0,\ldots,b_{B(R)-1}$
S stored in blocks $b'_0,\ldots,b'_{B(S)-1}$

```
left = 0
while left < B(S):
   right = min(left + M-2, B(S))
   read b_left,...,b_right-1 to memory
   for i = 0 to B(R)-1:
      read b'_i to memory
      for each tuple (e,f) of S in memory
         for each tuple (d,e') of R in memory
            if (e=e') add (d,e,f) to output
   left = left + M-2
```

כמה זה עולה לנו?

- את היחס החיצוני קוראים רק פעם אחת, לכן העלות היא B(S)
- את היחס הפנימי קוראים $\left\lceil\frac{B(S)}{M-2}\right\rceil$ פעמים, לכן העלות היא $B(R)\cdot\left\lceil\frac{B(S)}{M-2}\right\rceil$
- לכן העלות סה"כ היא $B(S)+B(R)\cdot\left\lceil\frac{B(S)}{M-2}\right\rceil$

כלומר עדיף לבחור את היחס הקטן יותר בתור היחס החיצוני, כי זה יחסוך לנו פעולות I/O.

עלות כתיבת התוצאה לא נחשבת כחלק מעלות הI/O, כי היא יכולה גם להיכתב למסך ולאו דווקא להישמר בדיסק.

ניתן לראות שמערכת הDB בחרה בזה ע"י שימוש בEXPLAIN, ואז לראות שעל שני היחסים מופעל Seq Scan, כאשר יש Nested Loop למעלה.

מה קורה אם יש גם תנאי בחירה? (נניח כרגע שאין לנו מבנה אינדקס)

1. חישוב כמו קודם ואז מחיקה של מה שלא עומד בתנאי- לא מוסיף עלות

2. דחיפת פעולת הבחירה לפני הצירוף- מקטין את אחד היחסים לפני פעולת הצירוף- מקטין מאוד את העלות ולכן עדיף כשאפשרי
   a. צריך לזכור בחישוב העלות לחשב את תנאי הבחירה (טעינת הטבלה והפעלת התנאי עליה)

INL - Index Nested Loops Join

סימונים: B(R) כמות הבלוקים ביחס R, T(R) כמות השורות ביחס R.
עובד בעזרת לולאה מקוננת שמניחה קיום של אינדקס על שדה הJOIN של היחס הפנימי.
משתמש רק ב4 בלוקים בזיכרון הראשי (בלי קשר לגודל הזיכרון הראשי, שיכול להיות גדול יותר):

1. בלוק של הטבלה הראשונה
2. בלוק של האינדקס
3. בלוק של הטבלה השנייה
4. בלוק של הפלט

פועל כך:

1. בוחרים מי יהיה היחס החיצוני ומי יהיה היחס הפנימי
2. נעבור על היחס החיצוני ונעבור שורה שורה בו
3. לכל שורה נשתמש במבנה אינדקס על היחס הפנימי כדי למצוא שורות מתאימות ביחס הפנימי

להלן **פסאודו קוד**, בו הנחנו שיש אינדקס על S.E (ולכן S הוא היחס הפנימי):

R(D, E) S(E,F)
Assuming index on S.E

```
for each block b of R:
   read b to memory
   for each tuple (d,e) in b:
      Search the index on S.E for e
      for each index entry (e,tupleId):
         access S at tupleId to retrieve (e,f)
         add (d,e,f) to the output
```

במה זה עולה לנו?

- את היחס החיצוני קוראים פעם אחת- עולה B(R)
- לכל שורה בR מפעילים את תנאי הבחירה, לכן עולה $T(R) \times cost\_of\_select$
  - העלות של הבחירה היא:
    - ירידה בBtree
    - טיול על העלים
    - גישה לבלוקים המתאימים לשורות עצמן
- לכן העלות סה"כ היא $B(R) + T(R) \times cost\_of\_select$

ניתן לראות שמערכת הDB בחרה בזה ע"י שימוש בEXPLAIN, ואז לראות שעל אחד מהיחסים מופעל Index Scan (ועל השני Seq Scan) תחת Nested Loop.

מה קורה אם יש גם תנאי בחירה?

1. חישוב כמו קודם ואז מחיקה של מה שלא עומד בתנאי- לא מוסיף עלות
2. דחיפת פעולת הבחירה לפני הצירוף- מקטין את אחד היחסים לפני פעולת הצירוף- הרבה יותר יעיל
   a. במקרה בו מדובר על בחירה בשדה של היחס הפנימי שאין עליו אינדקס זה בלתי אפשרי (אפשרי רק כשזה שדה של היחס החיצוני או חלק מהאינדקס של הפנימי)

לכן הכלל הוא- אם אפשר לדחוף את פעולת הבחירה, כדאי לעשות את זה כדי לחסוך בעלויות.

Hash Join

סימונים: B(R) כמות הבלוקים ביחס R, M כמות הבלוקים הפנויים בזיכרון המרכזי.

מתבסס על רעיון של שימוש בפונקציות Hash כדי למצוא שורות תואמות.

נפעיל פונקציית hash על הערך עליו הצירוף מתבצע בשני היחסים, ולפי התוצאה שנקבל נחלק את השורות ל"דליים" שונים (לכל יחס סט דליים שונה). כעת, אם שתי שורות אמורות להצטרף אחת לשנייה, אז הן נמצאות בדליים מתאימים. כעת במקום לעבור על כל הטבלה, נוכל רק לעבור על כל השורות בדלי התואם.

פועל כך:

1. נבנה hash table על כל אחד מהיחסים, שמשתמש ב-M-1 דליים
2. נקרא כל זוג של דליים מתאימים ונבצע את הצירוף ביניהם

להלן **פסאודו קוד**:

R(D, E)
S(E,F)

h is a hash function to [1,...,M-1]

```
for each block B∈R:
   read B
   for each tuple t∈B:
      add t to buffer block h(t[E]) //flush as fills
for each block B∈S:
   read B
   for each tuple s∈B:
      add s to buffer block h(s[E]) //flush as fills
for i=1 to M-1:
   read i-th partition Pi of R
   for each block B of i-th partition of S
      read B
      for each t∈Pi, s∈B:
         if t[E]=s[E]:
            add(t[D],t[E],s[F]) to output
```

במה זה עולה לנו?

- לייצר את הhash table עולה 2B(S) כי קראנו את S פעם אחת וכתבנו את S פעם אחת
- כמה גדולים יהיו הדליים? אם הערכים מפולגים באופן אחיד וגם פונקציית ההאש מפלגת את הערכים באופן אחיד, כל דלי יהיה בגודל של בערך $\left\lceil \frac{B(S)}{M-1} \right\rceil$ (משובך היונים יש לפחות אחד שבאמת בגודל כזה, ובקורס נניח שזה הגודל)
- כלומר סה"כ העלות היא:
  - יצירת ההאש טייבלס עולה 2B(R)+2B(S)
  - קריאת הדליים המתאימים עולה B(R)+B(S)
  - לכן סה"כ עולה 3B(R)+3B(S)

כדי להבטיח שלא נצטרך לקרוא דלי יותר מפעם אחת, נצטרך להבטיח שיש לנו מקום בזיכרון הראשי כדי לקרוא את אחד הדליים במלואו- נוכל לנצל M-2 בלוקים בזיכרון הראשי עבור זה.

אז כדי לקוות שנוכל לקבל זמן צירוף דליים שהוא לינארי, נצטרך שיתקיים $\left\lceil \frac{B(R)}{M-1} \right\rceil \leq M-2$ או $\left\lceil \frac{B(S)}{M-1} \right\rceil \leq M-2$, כי אז נוכל לקרוא את הדלי כולו במלואו ולהשאיר בלוק בזיכרון המרכזי לפלט ולקריאת הדלי השני. זה לא תמיד יתקיים, כמו במקרה בו הערכים לא מפולגים אחיד, אבל בקורס אנחנו נניח שהם כן.

ניתן לראות שמערכת הDB בחרה בזה ע"י שימוש בEXPLAIN, ואז לראות שכתוב Hash Join.

מה קורה אם יש גם תנאי בחירה?

1. חישוב כמו קודם ואז מחיקה של מה שלא עומד בתנאי- לא מוסיף עלות
2. דחיפת פעולת הבחירה לפני הצירוף- מקטין את אחד היחסים לפני פעולת הצירוף- הכתיבה שלו לhash table והדליים המתאימים יהיו קטנים יותר

כרגיל, אם אפשר אז כדאי לעשות את הבחירה קודם כדי לחסוך בעלויות.

<u>מיון</u>

השיטה האחרונה שנדבר עליה מתבססת על מיון היחסים.
לפעמים כשמבקשים לעשות מיון (ORDER BY) יתבצע מיון (יופיע sort אחרי שנעשה EXPLAIN), ולפעמים לא יהיה באמת מיון, אלא שימוש באינדקס על השדה אותו ביקשנו למיין (יופיע index only scan בEXPLAIN, או index scan אם ניגשים אחר כך למידע ביחס). אבל לרוב כן נצטרך לבצע מיון באמת.
מיון חשוב לא רק בשביל ORDER BY אלא גם בשביל מחיקת נתונים לDISTINICT, ובשביל תנאי בgroup by (אחת הדרכים לעשות את זה). בנוסף, בשביל איחוד, חיתוך או מינוס- לרוב נשתמש במיון (לפעמים אפשר להשיג את אותו אפקט בעזרת hash ולא מיון).

איך מיון ממומש?

בדר"כ נשתמש בפונקציה כמו quick sort ונזיז את הערכים במערך. הבעיה שאצלנו המידע הרבה יותר גדול מהזיכרון המרכזי, ואי אפשר להחזיק את כולו בזיכרון המרכזי. אם ננסה לעשות את זה, נקבל בעיה שאין מספיק מקום, או אם יש תמיכה רחבה בזיכרון וירטואלי- נצליח להקצות את המקום אבל הפעולה עצמה תהיה מאוד איטית כי כל הזמן נצטרך לקרוא בלוקים מהדיסק.
אז נצטרך אלגוריתם חיצוני (אלגוריתם שמטרתו לעבד נתונים שהם גדולים מהזיכרון המרכזי) שיעזור לכך, ואת הסיבוכיות נספור בכמות הקריאות והכתיבות לדיסק, בלי הזמן שלוקח לכתוב את התוצאה בחזרה לדיסק.

מיון חיצוני

מאוד חשוב ושימושי. נלמד מקרה פרטי של מיון חיצוני.
נרצה למיין את R, לו יש B(R) בלוקים, וM בלוקים פנויים בזיכרון המרכזי. נראה מקרה פרטי בו $\left\lceil\frac{B(R)}{M}\right\rceil < M$ (זה לא מאוד מורכב להרחיב אותו).
נשתמש במיון מיזוג מרכזי ב2 שלבים:

1. נייצר סדרות ממוינות קצרות מR
2. נמזג את הסדרות שייצרנו לקבל סדרה אחת גדולה וממוינת

להלן **פסאודו קוד**, המשתמש במיון במקום (בחלק של המיון בו):

$\left\lceil\frac{B(R)}{M}\right\rceil < M$

R stored in blocks $b_0,\dots,b_{B(R)-1}$

```
left = 0
while left < B(R):
    right = min(left + M, B(R))-1
    read blocks b_left,...,b_right to main memory
    sort b_left,...,b_right in place
    write sorted sequence to disk
    left = left + M
```

1. a. בסוף שלב זה, יש $\left\lceil\frac{B(R)}{M}\right\rceil$ סדרות ממוינות.

$\left\lceil\frac{B(R)}{M}\right\rceil < M$

Suppose there are n tuples in each block of R and N tuples in total

```
for i = 1 to ⌈B(R)/M⌉:
    read the first block of the i-th sorted
        sequence into memory
    set p_i to point to the first tuple in
        this block
```

2. לכל אחת מהסדרות הממוינות, קוראים את הבלוק הראשון מהדיסק לזיכרון הראשי.

a. נשים לב שבגלל ההנחה $\left\lceil \frac{B(R)}{M} \right\rceil < M$ (קטן ממש), נשאר בלוק פנוי בזיכרון המרכזי- נשתמש בו לכתיבה הפלט

Suppose there are n tuples in each block of R and N tuples in total

```
k = 0
while k < N:
  let pi be the pointer to the minimal
     tuple among all pointers
  copy the tuple of pi to the output block
  increment k
  if k mod n = 0
     flush the output block to screen/disk
  if pi points to the last tuple of the
     block, read the next block of the
     sequence and set pi to its first tuple
  else increment pi
```

b. K יהיה כמות הערכים שכבר הוצאנו במיון

כמה זה עולה לנו?

- עלות השלב הראשון היא 2B(R) (קריאה יחידה, מיון במקום וכתיבה יחידה)
- השלב השני עולה B(R) (קריאה של כל הבלוקים הממוינים שוב)
- סה"כ עולה 3B(R)

אם $\left\lceil \frac{B(R)}{M} \right\rceil \geq M$, לא נוכל לעשות מיון כזה בשלב אחד אלא נעשה כמה שלבים של מיזוג.

SM - Sort Merge Join

1. מיון היחס הראשון לפי השדה עליו הJOIN קורה
2. מיון היחס השני לפי השדה עליו הJOIN קורה
3. נקרא את הטבלאות הממוינות במקביל וניצור את השורות התואמות

להלן **פסאודו קוד**:

R(D, E)
S(E,F)

```
Sort R by E
Sort S by E
Tr = first tuple in R
Ts = first tuple in S
Gs = first tuple in S
while Tr ≠ eof and Gs ≠ eof:
  while Tr ≠ eof and  Tr[E]<Gs[E]:
     increment Tr
  while Gs ≠ eof and Tr[E]>Gs[E]:
     increment Gs
  while Tr ≠ eof  and Tr[E] == Gs[E]:
     Ts = Gs
     while Ts ≠ eof  and Ts[E] == Tr[E]:
        add (Tr,Ts) to output
        increment Ts
     increment Tr
  Gs = Ts
```

התפקיד של G_S הוא לאפשר לחזור אחורה לשורות שכבר עברנו עליהן כשיש את אותו ערך בכמה שורות.

מתי אפשר להשתמש באלגוריתם? כשאפשר למיין את R ($\left\lceil \frac{B(R)}{M} \right\rceil < M$) ואת S ($\left\lceil \frac{B(S)}{M} \right\rceil < M$).

מהי סיבוכיות הI/O שלו?

- מיון R עולה 3B(R), ואת התוצאה הממוינת כותבים לדיסק- סה"כ 4B(R)
- מיון S עולה 3B(S), ואת התוצאה הממוינת כותבים לדיסק- סה"כ 4B(S)
- קוראים אותם ועושים מעין צירוף מיון, עולה B(R)+B(S)
- סה"כ עולה 5B(R)+5B(S)

האם אפשר להיות יותר יעילים? נניח שR ממוין בM_1 רצפים ממוינים, S ממוין בM_2 רצפים ממוינים ו$M_1 + M_2 > M$. אזי יש מספיק מקום בזיכרון המרכזי לקרוא את הבלוק הראשון של כל

רצף ממוין של R ושל S ולהשאיר בלוק לתוצאה. כלומר נוכל לעשות את צירוף המיזוג תוך כדי מעבר על היחסים בזמן לינארי.

כלומר אם $\left\lceil\frac{B(R)}{M}\right\rceil + \left\lceil\frac{B(S)}{M}\right\rceil < M$ (כמות הרצפים הממוינים בשלב הראשוני של אלגוריתם המיון)

נוכל לשפר את הסיבוכיות ע"י ויתור על יצירת היחסים הממוינים במלואם:

- ניצור רצפים ממוינים בגודל M של R- לוקח 2B(R)
- ניצור רצפים ממוינים בגודל M של S- לוקח 2B(S)
- קוראים לזיכרון המרכזי את הבלוק הראשון של כל אחד מהרצפים הממוינים ויצירת המיזוג לצירוף- לוקח B(R)+B(S)
- סה"כ עולה 3B(R)+3B(S) (כמו הhash join)

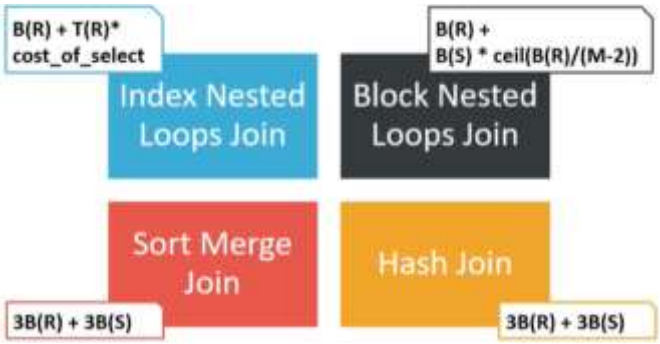
ניתן לראות שמערכת הDB בחרה בזה ע"י שימוש בEXPLAIN, ואז לראות שכתוב Merge Join.
מערכת הDB יכולה למיין ביותר מ2 שלבים ולפעמים תבחר לעשות את זה, אבל לא נתעסק בזה בקורס.

האם זה תמיד יהיה יעיל? האלגוריתם מתבסס על זה שאת השלב האחרון אפשר לעשות בזמן לינארי, אבל לא תמיד זה נכון. לשמחתנו המקרה הנפוץ הוא שזה כן נכון, כי לרוב עושים צירוף בין שדה שהוא מפתח או מפתח זר בטבלאות. לכן נפשט את הדיון ונחשוב שהבעיה לא קיימת.

מה קורה אם יש גם תנאי בחירה?

1. חישוב כמו קודם ואז מחיקה של מה שלא עומד בתנאי- לא מוסיף עלות
2. דחיפת פעולת הבחירה לפני הצירוף- מקטין את אחד היחסים לפני פעולת הצירוף- ברגיל זה עדיף

כשיש שאילתה עם צירוף, מערכת הDB מעריכה כמה כל מיון יעלה ובוחרת את היעילה ביותר.



הרצאה 7 - Optimal Plans (Evaluation)

<u>הקדמה</u>

נחבר את מה שלמדנו בשבועיים האחרונים ונבין איך מערכת הDB מחשבת שאילתה שלמה.
איך נבחר איך לעבור? אנחנו כאנשים יכולים לקבל הערכה טובה מהידע שלנו על ימאים (כנראה יהיו קצת שגילם גדול מ90 ולכן נעדיף מבנה אינדקס, אך לכולם גילם גדול מ0 ולכן נעדיף סריקה של הטבלה), אך איך מערכת הDB יכולה לקבל הערכה טובה של השיטה שכדאי לבצע בה את השאילתה? כדי לבחור תוכנית ביצוע, מערכת הDB צריכה להעריך את הסלקטיביות של הדברים, ובשביל זה צריכה לעשות סטטיסטיקות. כדי לבחור בין תוכניות ביצוע שונות, אנחנו צריכים להיות מסוגלים להעריך את הגודל של תוצאות הביניים (פעולות בחירה או צירוף).

סטטיסטיקות

מערכת הDB מחשבת ושומרת מידע סטטיסטי על הטבלות שלנו ועל האינדקסים שלהן.
בPostgres סטטיסטיקות מיוצרות ומעודכנים בזמן יצירת אינדקס, או כשמריצים את הפקודה

ANALYZE או VACUUM. אלו פקודות שרצות מדי פעם גם בלי שקראנו להן ואוספות סטטיסטיקות בזמן שמערכת הDB נחה.

תצוגה של הסטטיסטיקות עבור כל עמודה:

- אפשר לשנות את הפקודה כך שהשורות יופיעו לנו אנכית ולא אופקית- \x
- inherited זה האם הוא יורש את השדה מטבלה אחרת (מפתח זר)
- null_frac זה אחוז הערכים שערכם null
- avg_width זה הרוחב הממוצע לערכים בעמודה הזו בבייטים
- n_distinct זה כמות הערכים השונים שיש באותה עמודה- אם המספר חיובי הוא מציין את כמות הערכים השונים בעמודה, אם הוא שלילי זה אחוז הערכים השליליים מתוך כלל הערכים בטבלה
- most_common_vals ו-most_common_freqs מכיל את הערכים השונים הנפוצים ביותר והתדירויות שלהם
- histogram_bounds זה חלוקה של הערכים בעמודה כך שבכל טווח יש בערך אותה כמות של שורות
- correlation היחס בין מיקום של ערכים דומים בטבלה למיקום הפיזי שלהם בדיסק- אם הוא קרוב לו1 אז ערכים שערכם קרוב גם קרובים בדיסק (עונה על השאלה האם בגישה דרך מבנה אינדקס יש דרך לעשות פחות פעולות של קריאות לדיסק)

<u>הערכת גדלים</u>

איך ההערכות קורות? יש שיטות פשוטות לשערך את הגודל של תוצאה של שאילתה, בהינתן סטטיסטיקות על הטבלאות. יש שיטות מסובכות יותר, אבל לא נדבר עליהן.

סימונים- B(R) כמות הבלוקים בדיסק שR תופס, T(R) כמות השורות של R, V(R,A) כמות הערכים השונים בשדה A של הטבלה R. אם A הוא מפתח אז V(R,A)=T(R).

איך אפשר להעריך גודל תוצאה של שאילתה? נניח התפלגות אחידה על הערכים בטבלה. כרגע נדבר על כמות השורות בתוצאה, אך חשוב לשים לב שלפעמים נתעניין בכמות הבלוקים בתוצאה.

1. תנאי בחירה (שוויון)
   a. **השאילתה:** SELECT * FROM R WHERE A = 'a' ($\sigma_{A=\text{"a"}}(R)$)
   b. **הנחה:** הערכים בA מתפלגים באופן אחיד
   c. **מספר השורות בתוצאה:** $\frac{T(R)}{V(R,A)}$
2. תנאי השוואה (אי שוויון)
   a. **השאילתה:** SELECT * FROM R WHERE A <= x ($\sigma_{A\leq x}(R)$)
   b. **הנחה:** הערכים בA הם בטווח [y,z]
   c. **מספר השורות בתוצאה:** $T(R)\frac{x-y+1}{z-y+1}$. נשים לב שz-y+1 זה גודל הטווח כולו, x-y+1 אלו הערכים שמקיימים את התנאי.
3. תנאי "קטן מ" כשלא יודעים כלום על הטווח של A
   a. **השאילתה:** SELECT * FROM R WHERE A < x ($\sigma_{A<x}(R)$)
   b. **הנחה:** הנחה מפשטת ששליש מהערכים עונים על התנאי. למה? כובע. (יש כאלו שמשתמשים בהערכה של חצי)
   c. **מספר השורות בתוצאה:** $\frac{T(R)}{3}$
4. שני תנאי בחירה
   a. **השאילתה:** SELECT * FROM R WHERE A = 'a' and B = 'b' ($\sigma_{A=\text{"a" and B="b"}}(R)$)
   b. **הנחה:** הערכים מתפלגים באופן אחיד והם בלתי תלויים
   c. **מספר השורות בתוצאה:** $\frac{T(R)}{V(R,A)\times V(R,B)}$

5. שני תנאים (שילוב של מה שהיה קודם)
   a. **השאילתה:** SELECT * FROM R WHERE A = 'a' and B < 'b' ($\sigma_{A=\text{"a" and B}<b}(R)$)
   b. **הנחה:** הערכים בלתי תלויים והטווח של B לא ידוע
   c. **מספר השורות בתוצאה:** $\frac{T(R)}{V(R,A)\times 3}$
6. שתי השוואות
   a. **השאילתה:** SELECT * FROM R WHERE A < 'a' and B < 'b' ($\sigma_{A<\text{a and B}<b}(R)$)
   b. **הנחה:** התנאים בלתי תלויים ושני הטווח לא ידועים
   c. **מספר השורות בתוצאה:** $\frac{T(R)}{3\times 3}$
7. תנאי **או** (ולא וגם)
   a. **השאילתה:** SELECT * FROM R WHERE A = 'a' or B 'b' ($\sigma_{A=\text{a or B}=b}(R)$)
   b. **הנחה:** התנאים בלתי תלויים
   c. **מספר השורות בתוצאה:** $T(R)\left(1-\underbrace{\underbrace{\left(1-\frac{1}{(V(R,A)}\right)}_{A\neq' a'}\times\underbrace{\left(1-\frac{1}{V(R,B)}\right)}_{B\neq' b'}}_{\text{לא עונה על אף תנאי}}\right)$
8. פעולה של צירוף (למעשה הפעלה חוזרת של תנאי בחירה)
   a. **השאילתה:** SELECT * FROM R,S WHERE R.A=S.A ($R \bowtie S$)
   b. **הנחה:** הערכים מתפלגים באופן אחיד בשתי הטבלאות
   c. **מספר השורות בתוצאה:** $\frac{T(R)\times T(S)}{\max\{V(R,A),V(S,A)\}}$
   d. מה קורה מתי שA מפתח בR ומפתח זר בS? כל שורה בS תצטרף בדיוק עם שורה אחת בR. **מספר השורות בתוצאה** T(S)- מתאים לנוסחה בc
9. פעולת צירוף וגם תנאי בחירה
   a. **השאילתה:** SELECT * FROM R,S WHERE R.A=S.A and R.B='b' ($\sigma_{B='b'}(R \bowtie S) = (\sigma_{B='b'}R) \bowtie S$)
   b. **הנחה:** הערכים מתפלגים באופן אחיד בשתי הטבלאות
   c. **מספר השורות בתוצאה:** $\frac{T(R)\times T(S)}{\max\{V(R,A),V(S,A)\}\times V(R,B)}$
10. צירוף על שתי עמודות
   a. **השאילתה:** SELECT * FROM R,S WHERE R.A=S.A and R.B=S.B ($R \bowtie S$)
   b. **הנחה:** הערכים מתפלגים באופן אחיד בשתי הטבלאות
   c. **מספר השורות בתוצאה:** $\frac{T(R)\times T(S)}{\max\{V(R,A),V(S,A)\}\times\max\{V(R,B),V(S,B)\}}$
11. הטלה ללא מחיקת כפילויות
   a. **השאילתה:** SELECT A FROM R ($\pi_A(R)$)
   b. **מספר השורות בתוצאה:** $T(R)$
12. הטלה עם מחיקת כפילויות
   a. **השאילתה:** SELECT DISTINICT A FROM R ($\pi_A(R)$)
   b. **מספר השורות בתוצאה:** $V(R,A)$

<u>Query Plans תוכנית ביצוע שאילתה</u>

אחרי שיודעים לחשב סטטיסטיקות, אפשר לעבור ליצור תוכנית ביצוע שאילתה בצורה יעילה.
תוכנית ביצוע שאילתה זה תיאור מדויק של איך הולכים לבצע שאילתה.
ננסה להבין איך מערכת הDB בוחרת את התוכנית, כדי לעשות את זה:

1. נתרגם את השאילתה לאלגברה רלציונית
   a. במקום לרשום את הביטוי באלגברה בצורה רגילה- נציג אותו כעץ של פעולות, כאשר החלק התחתון זה היחס הכי פנימי.
2. לכל אחת מהפעולות באלגברה רלציונית נבחר תוכנית ביצוע (אלגוריתם)
   a. נכתוב ליד אופרטורים אלגבריים את תוכנית הביצוע
   b. היחס החיצוני הוא זה שכתוב בצד שמאל
   c. אם לא רשמנו אופרטור, נניח שהתוצאות עוברות בצורה ישירה לפעולה הבאה בצורה שנקראת pipelined- בלי כתיבה של תוצאות הביניים לדיסק, כלומר כל שורה עוברת ישירות לפעולה הבאה
   d. על בחירה צריך לכתוב אם עשינו index scan או full table scan לפי האם יש אינדקס על השדה הזה
3. תמיד נדחוף פעולות של הטבלה ובחירה (כי ראינו שזה תמיד מקטין את תוצאות הבחירה ועושה חישוב יעיל יותר)

כרגע נדבר רק על פעולת צירוף אחת (ולא על כמה צירופים).
אם נצטמצם לתוכניות ביצוע בהן דוחפים את פעולות הבחירה וההטלה פנימה בצורה מקסימלית, **כמה תוכניות שונות נראה עבור אותה שאילתה?**
מבני האינדקסים הקיימים נותנים לנו מידע חשוב לגבי איזה תוכניות ביצוע כדאי ואפשר לעשות.

- בחירה האם להשתמש בindex scan או בfull table scan עבור כל אחת מהטבלאות עליהן קיים אינדקס
- בחירה מי היחס הפנימי ומי היחס החיצוני
  - רק בBNL אפשר להחליף בלי בעיה
- בחירה באיזו דרך לעשות את הצירוף
  - 4 אפשרויות (BNL, MS, HJ, INL), לא תמיד אפשר לעשות את כולן
  - בINL יש פחות אפשרויות כי לטבלה הפנימית חייב להיות אינדקס על שדה הצירוף. בנוסף, לא ניתן לדחוף את הבחירה אם האינדקס לא מאפשר את זה (האינדקס צריך להיות גם על השדה עליו התנאי)

הבחירה הטובה ביותר מבין האפשרויות (בדוגמא מהמצגת יש 18 כאלו) תלויה בנתונים.

<u>בחירת תוכנית ביצוע</u>

מהי הצורה הנכונה ביותר לגשת לכל אחד משני היחסים? מהי שיטת הצירוף האופטימלית?
איך נבחר את התוכנית הטובה ביותר מבין כל האופציות שיש לנו? התשובה בנויה מ:

1. נבין מהי הדרך הזולה ביותר לבצע את פעולת הבחירה- סריקה מלאה של הטבלה או שימוש באינדקס (בשהוא קיים)
   a. חישוב עבור כל אחת מהטבלאות
2. נמצא מהי שיטת הצירוף הזולה ביותר
3. נניח דחיפה מקסימלית של פעולות הבחירה וההטלה (בכלל לא נסתכל על מקרים בהם היה אפשר לדחוף ובחרנו לא לעשות את זה)

עלות פעולת הבחירה

למעשה אנחנו כבר יודעים איך לענות על שאלות כאלו- דיברנו על חישוב עלויות של שימוש באינדקס לפני שבועיים.
אחרי השלב הראשון, נוכל לצמצם משמעותית החוצה את כל התוכניות שהשתמשו בסריקה היקרה יותר עבור כל אחת מהטבלאות.

עלות הצירוף

לכאורה אנחנו יודעים לבצע את זה- יש לנו נוסחאות משבוע שעבר שנוכל לחשב בעזרתן. אבל, שבוע שעבר הנחנו שאנחנו מצרפים יחסים R ו-S, אבל בפועל נצרף תוצאות של ביטויים E_R, E_S שמכילים פעולות של בחירה/הטלה וזה משנה את הזמנים.

אז מהי בדיוק עלות נוסחאות החישוב מעל ביטויים כללים ולא רק מעל יחסים?

איפה נכנס חוסר הדיוק? השתמשנו ב-B(R) כגודל של R ובעלות הקריאה שלו. כשיש ביטוי, הם יכולים להיות שונים. נרצה להבדיל בין גודל הביטוי לבין עלות קריאת הבחירה.

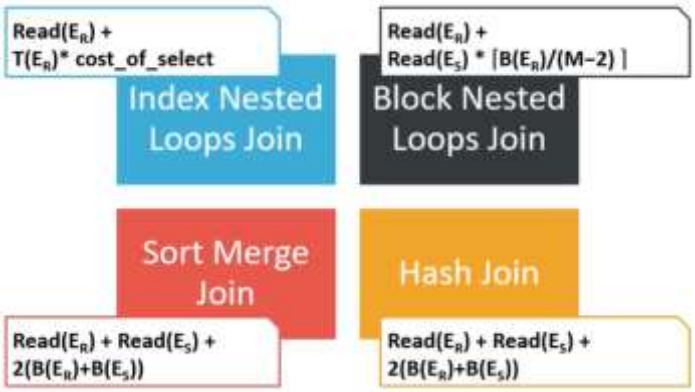
נדבר על ביטויים שיכולים להכיל פעולות של בחירה/הטלה/שניהם.

נגדיר: Read(E) עלות קריאת E מהדיסק, B(E) כמות הבלוקים ב-E, T(E) כמות השורות ב-E.

העלויות:

1. BNL: עבור R היחס החיצוני ו-S הפנימי (E_R היחס החיצוני ו-E_S הפנימי)
   a. רגיל: $B(R) + B(S) \times \left\lceil \frac{B(R)}{M-2} \right\rceil$
   b. על ביטויים: $Read(E_R) + Read(E_S) \times \left\lceil \frac{B(E_R)}{M-2} \right\rceil$
2. INL: עבור R היחס החיצוני ו-S הפנימי (E_R היחס החיצוני ו-E_S הפנימי)
   a. רגיל: $B(R) + T(R) \times cost\_of\_select$
   b. על ביטויים: $Read(E_R) + T(E_R) \times cost\_of\_select$
3. HJ:
   a. רגיל: $3B(R) + 3B(S)$
   אפשר להשתמש כש-$\left\lceil \frac{B(R)}{M-1} \right\rceil < M - 1$ או $\left\lceil \frac{B(S)}{M-1} \right\rceil < M - 1$
   b. על ביטויים: $Read(E_R) + Read(E_S) + 2\big(B(E_R) + B(E_S)\big)$
   אפשר להשתמש כש-$\left\lceil \frac{B(E_S)}{M-1} \right\rceil < M - 1$ או $\left\lceil \frac{B(E_R)}{M-1} \right\rceil < M - 1$ (כי כותבים חזרה רק את מי שמעניין אותנו)
4. SM גרסה 1:
   a. רגיל: $5B(R) + 5B(S)$
   אפשר להשתמש כש-$\left\lceil \frac{B(R)}{M} \right\rceil < M$ וגם $\left\lceil \frac{B(S)}{M} \right\rceil < M$
   b. על ביטויים: $Read(E_R) + Read(E_S) + 4\big(B(E_R) + B(E_S)\big)$ (רק בשלב הראשון קוראים את כל R ו-S)
   אפשר להשתמש כש-$\left\lceil \frac{B(E_R)}{M} \right\rceil < M$ וגם $\left\lceil \frac{B(E_S)}{M} \right\rceil < M$
5. SM גרסה 2:
   a. רגיל: $3B(R) + 3B(S)$
   אפשר להשתמש כש-$\left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil < M$
   b. על ביטויים: $Read(E_R) + Read(E_S) + 2\big(B(E_R) + B(E_S)\big)$ (2 הפעולות זה קריאת ומיזוג)
   אפשר להשתמש כש-$\left\lceil \frac{B(E_R)}{M} \right\rceil + \left\lceil \frac{B(E_S)}{M} \right\rceil < M$

לסיכום:



מציאת תוכנית אופטימלית

חשוב לזכור להקטין את הגודל אחרי ההטלה (לחשב את כמות השורות כפול הגודל של השדות בהטלה).

כשאנחנו מחשבים, אפשר לשים לב שעל חלק מהחישובים אפשר לדלג אחרי שהגענו לתוצאה טובה באחד מהחישובים. בדוגמא, בBNL עלות $\left\lceil \frac{B(E_R)}{M-2} \right\rceil$ הייתה 1 ולכן העלות הכוללת הייתה $Read(E_R) + Read(E_S)$, אז כל חישוב שעושה יותר מזה יהיה פחות יעיל בוודאות ולא נצטרך לחשב אותו.

כעת אנחנו יודעים למצוא תוכנית ביצוע אופטימלית לשאילתה עם פעולה אחת של צירוף ופעולות של הטלה. מערכת הDB צריכה להתמודד גם עם דברים יותר מסובכים, אבל לא נלמד את זה. כן נוכל להבין איזו תוכנית מערכת הDB בחרה, וכך לייעל שאילתות שרצות באיטיות.

הרצאה 8 – תאוריית התכנון Framework (Design Theory)

מבוא

תאוריית התכנון עוסקת בנרמול של מסד הנתונים.

תיאוריית התכנון נותנת מסגרת פורמלית ומדוייקת למה הוא תכנון טוב של מסד נתונים.

נרמול של מסד נתונים הוא מושג שמתייחס לתכנון סכמה שמקיימת עקרונות מסוימים. נבין מהם ואיך מיישמים אותם. היתרונות שלהם זה שיהיה מהיר יותר לבצע פעולות של עדכון, הכנסה, הטבלאות יהיו קטנות יותר ויהיה קל יות לשמור על נכונות המידע. החסרון הוא שכשנצטרך לחשב שאילתות יהיו לנו יתר פעולות של צירוף, לכן יש גישה שאומרת לנרמל עד שזה "באוב", ואז תעשו את ההפך עד שזה "עובד". אנחנו נרצה לבצע נרמול כמה שיותר טוב למסד, ורק אם נראה שנוצרת בעיה נרצה לחזור אחורה.

בעיות בטבלה שבמצגת: הגבלת כמות הלקוחות לכל עובד. קשה לחפש מידע וזה לא מבטא את העובדה שיש עובד אחד לכל לקוח. פתרון יכול להיות שהעמודה של לקוח מכילה קבוצה של ערכים (מערכת הDB מאפשרת את זה), אבל זה עדיין לא מבטא את היחידות וכתיבה של שאילתות על העמודה הזו היא פחות יעילה.

לכן, נעדיף שמסד הנתונים יהיה ב**first normal form**.

טבלה היא בצורה נורמלית ראשונה אם כל עמודה מכילה רק ערכים אטומיים (לא קבוצות או מערכים) ואין עמודות שחוזרות על עצמן (לקוח 1, לקוח 2 וכו'). תמיד כדאי לבנות מסד כזה. נשיג את זה ע"י שמירה בשתי טבלאות.

עדיין יש חזרתיות במסד. יש מידע משוכפל, וזה דבר רע- מגדיל את הצורך בעדכון, ושינוי צריך להעשות בזהירות כי צריך לשמור על אחידות. לזה קוראים **אנומליית עדכון**.

בעיה נוספת היא **אנומליית הוספה**- אם פתחנו משרד נוסף אבל עוד אין לנו עובדים בו, אין לנו יכולת להכניס את המידע הזה לטבלה.

בעיה נוספת היא **אנומליית מחיקה**- אם העובד האחרון עזב נאבד גם את המידע על העובד (שזה בסדר כי הוא עזב) וגם את המידע על המשרד. כלומר, מוחקים מידע שלא אמור להמחק רק בגלל הצורה שבה שמרנו את הנתונים.

כל זה יכול לייצר מצב של חוזר עקביות בנתונים במסד. בכל מסד נתונים גדול יש חוסר עקביות מסוים ויכולות להיכנס אליו שגיאות. בתכנון המסד ננסה להבין איך להימנע מהן כמה שיותר.

נרצה דרך להוכיח בצורה חד משמעית שמסד מתוכנן טוב או לא.

אם נחפור ביסוד, נבין מזה שידענו שלמשרד יש רק טלפון אחד. זה נקרא **תלות פונקציונאלית**.

בגלל שיש הרבה עובדים, המספר נשמר הרבה פעמים. יכול להיות שזו לא בעיה בתכנון- למשל אם יודעים שגם לכל עובד יש מספר טלפון יחיד וגם בכל משרד יש עובד יחיד.

כדי שנוכל להכריע אם תכנון הוא טוב, נצטרך מידע "על העולם".

מטרות

המטרה הראשונה היא לתת הגדרה פורמלית להחלטה האם סכמה שקיבלנו מתוכננת ובנויה היטב.

המטרה השנייה היא לראות איך אפשר בצורה אוטומטית לתקן סכמה שלא עונה על הקריטריונים (לא מתוכננת היטב) לסכמה שתענה על הקריטריונים ותהיה מתוכננת לפי העקרונות הנכונים.

תלות פונקציונאלית

סימונים:

- A, B, C... – עמודה בודדת בתוך סכמה של יחס
- X, Y ,... – קבוצות של עמודות
- כתיבה של שתי אותיות אחת ליד השנייה – הכוונה לאיחוד

$$XY = X \cup Y$$
$$XA = X \cup \{A\}$$
$$ABC = \{ABC\}$$
$$XX = X \cup X = X$$

- R – שם של יחס
  - לדוגמא R(A,B,C) מכיל את העמודות A,B,C
- r – מופע של יחס (דוגמא לתוכן של היחס – אוסף של שורות)
- s, t – שורות במופע של יחס
  - נשתמש בסימון s[A] כדי לציין הטלה של הערך בשורה של s רק על העמודה A

הגדרה (תלות פונקציונאלית):

בהינתן:

- סכמה של יחס $R(A_1, \dots, A_n)$
- שתי קבוצות של עמודות $X \subseteq A_1, \dots, A_n, Y \subseteq A_1, \dots, A_n$
- מופע r של R

הגדרה: התלות הפונקציונאלית $X \to Y$ מתקיים במופע r אם **לכל** שתי שורות $s, t \in r$ מקיים שאם $s[X] = t[X]$ אזי $s[Y] = t[Y]$. כלומר הערך שכתוב בX קובע פונקציונאלית את הערך בY.

הגדרה שקולה (על דרך השלילה): $X \to Y$ לא מתקיים במופע r אם **קיימות** שורות $s, t \in r$ כך ש$s[X] = t[X]$ אבל $s[Y] \neq t[Y]$. כלומר אם מצאנו לה סתירה.

מהדוגמאות ראינו שהתלות יכולה גם להיות טאוטולוגיה ($CF \to F$).
ברגע שכל השורות שונות – כל תלות שנכתוב מתקיימת באופן ריק. כדי לדעת מה אמור להתקיים, נצטרך שמתכנן הסכמה יגיד לנו מה חייב לקרות.
מדוגמא של יחס אפשר לראות איזה תלויות אמורות להיות, אבל זה לא אומר לנו לגבי כל המקרים.

נביעה לוגית של תלות פונקציונאלית implications

מתכנן הסכמה מגדיר אלו תלויות אמורות להתקיים, אבל יכול להיות שהוא יגדיר רק חלק מהן (יכול להיות מייגע לציין הכל). נרצה להסיק מתוך קבוצה של תלויות, איזה עוד תלויות אמורות להתקיים.
נסתכל על קבוצה של תלויות פונקציונאלית F. נגיד שF משתמע $X \to Y$ אם לכל מופע r של R, אם כל F מתקיים בr אז גם $X \to Y$ מתקיים בr. לכן נגיד ש$\boldsymbol{X \to Y}$ **נובע מF**.
כדי להוכיח נרצה להראות שלכל מופע של היחס בו F מתקיים אזי גם $X \to Y$ מתקיים.
הוכחה: עבור $F = \{A \to B, A \to C, C \to D\}$ נרצה להוכיח ש$A \to D$. יהי r מופע בו F מתקיים.
נראה שגם $A \to D$ מתקיים. יהיו s, t שורות בr כך ש$s[A] = t[A]$. מכיוון שF מתקיים אזי $s[C] = t[C]$. שוב, מכיוון שF מתקיים אזי$s[D] = t[D]$. לכן s, t לא סותרות את התלות. אין זוג שסותר את התלות ולכן היא מתקיימת.
הפרכה (דוגמא נגדית): בשביל לסתור תלות, נבנה מופע של היחס שסותר אותה.
הוכחה של טענה יכולה להיות בעזרת טרנזיטיביות, או אם הוספנו עמודה לX כשידוע ש$X \to Y$.

תלות טריוויאלית היא טאוטולוגיה: כש$Y \subseteq X$ נגיד ש$X \to Y$ היא תלות טריוויאלית. לדוגמא $S \to S, SD \to S$.

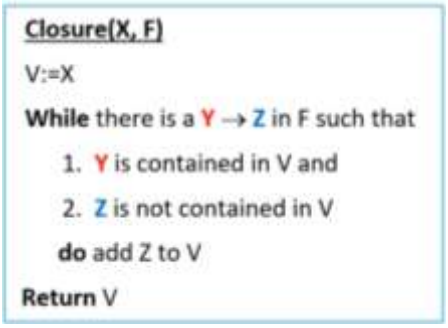
איך נבדוק נביעה? כדי להפריך נביא דוגמא נגדית (נתחיל מלהכניס ערכים שתומכים במה שנרצה להפריך, ואז נכניס ערכים שנכונים בF). כדי להוכיח נכתוב הוכחה מילולית. בחלק הבא נמצא דרך לעשות את שני הדברים בלי הדרך הארוכה.

סגור closure

בהינתן קבוצה של תלויות פונקציונאליות F וקבוצה של עמודות X, נגדיר את X_F^+ **הסגור של X יחסית לF** להיות קבוצת העמודות A כך ש$X \to A$ נובע מF (נסמן $F \models X \to A$).

כשF ברור מההקשר, נכתוב רק X^+ (X סגור, בלי לציין את הF למטה).

אלגוריתם לחישוב X_F^+:



הוא אלגוריתם פולינומיאלי, ואפשר גם לשפר אותו כדי שירוץ בזמן לינארי. בכל אופן, הוא יעיל. האם האלגוריתם נכון? נבדוק האם מה שהוא מחזיר מתאים להגדרה. כדי להוכיח, נראה הכלה דו כיוונית.

הוכחה:

- $Closure(X,F) \subseteq X_F^+$ (הצד הקל) נוכיח באינדוקציה על האיטרציה בה הוספנו עמודות לקבוצה V (השורה הראשונה- ברור שאת עצמו הוא קובע, בהמשך- מטעמי טרנזיטיביות).

  טענה: אם הוספנו את A לV לאחר k איטרציות של הלולאה, אז $X \to A$ נובע מF.

  בסיס האינדוקציה $k = 0$ אז $A \in X$. לכן, בוודאי $X \to A$ נובע מF (תלות טריוויאלית).

  צעד האינדוקציה נניח שהוספנו את A לאחר k איטרציות. אז יש $Y \to Z \in F$ כך ש$Y \subseteq V$ ו$A \in Z$. נרשום $Y = B_1, \dots, B_n$. לפי הנחת האינדוקציה, $X \to B_i$ נובע מF לכל i. יהי r מופע של יחס שמקיים את F ויהיו s, t שורות בr כך ש$s[X] = t[X]$. מכיוון שF מתקיים ו$X \to B_i$ נובע מF, אזי $\forall i\ s[B_i] = t[B_i]$. מכיוון שr מקיים את F, הוא בפרט מקיים את $Y \to Z$. מכיוון ש$s[T] = t[Y]$ אז $s[Z] = t[Z]$ ובפרט $s[A] = t[A]$. כלומר, $X \to A$ נובע מF כנדרש.

- $Closure(X,F) \supseteq X_F^+$ (פחות טריוויאלי) נצטרך להראות שאם $X \to A$ נובע מF, אזי $A \in Closure(X,F)$. נוכיח טענה שקולה (על דרך השלילה): אם $A \notin Closure(X,F)$ אזי $X \to A$ לא נובע מF. כדי להראות את זה, נראה שאם $A \notin Closure(X,F)$ אזי יש מופע של יחס בו F מתקיים אבל $X \to A$ לא. **אנחנו נניח ש$A \notin Closure(X,F)$ ונייצר מופע של יחס בו F מתקיים אבל $X \to A$ לא.**

  במופע של היחס נציג 2 שורות, ובהן נקבע את הערכים בעמודות המתאימות ל$Closure(X,F)$ להיות זהים (0 במקרה שלנו), ובשאר העמודות להיות שונים (באחת מהן 0 ובשנייה 1). נראה שזהו אכן מופע של יחס בו F מתקיים ו$X \to A$ לא מתקיים:

  - F מתקיים: נניח ש$Y \to Z$ היא תלות בF.
    - מקרה 1: $Y \subseteq Closure(X,F)$. בגלל האופן בו האלגוריתם עובד, $Z \subseteq Closure(X,F)$. לכן השורות זהות גם בY וגם בZ ולכן התלות מתקיימת.

- <u>מקרה 2:</u> $Y \notin Closure(X,F)$. שתי השורות שונות בY, והתלות מתקיימת (כדי שהיא לא תתקיים צריך שהן יהיו שונות בY ושוות בZ, אבל זה לא קורה כי הן שונות כבר בY).
  - <u>X → A לא מתקיים:</u> נשים לב ש$X \subseteq Closure(X,F)$. לפי ההנחה, $A \notin Closure(X,F)$. לכן, השורות זהות בX ושונות בA. לכן, התלות לא מתקיימת.

לסיכום:

<u>הגדרה:</u> $X^+ = \{A | X \to A \ follows \ from \ F\}$.

<u>טענה (שהוכחנו):</u> $X^+ = Closure\{X,F\}$.

<u>למה:</u> $X \to Y \ follows \ from \ F \Leftrightarrow Y \subseteq X^+$.

<u>מסקנה:</u> $X \to Y \ follows \ from \ F \Leftrightarrow Y \subseteq Closure(X,F)$.

זה חשוב כי אם נרצה לדעת האם X → Y נובע מFn, רק נצטרך לחשב את X^+ בעזרת אלגוריתם הסגור ולבדוק אם Y מוכל בתוצאה.

<u>מפתחות ומפתחות על</u>

<u>הגדרה:</u> קבוצת העמודות X היא **מפתח על** בR אם $X^+ = R$.

<u>הגדרה:</u> קבוצת העמודות X היא **מפתח** בR (הגדרה חזקה יותר) אם:

1. $X^+ = R$ הוא מפתח על
2. $\forall Y \subset X, Y^+ \subset R$ הוא מינימלי- אם נוריד ממנו עמודה נקבל קבוצה שהיא לא מפתח על

יכולים להיות מספר מפתחות, אפילו בגדלים שונים.

אלגוריתם למציאת מפתח:

```
Minimize(X, F):
for each A∈X
        if A∈{X-A}⁺
            then X = X – {A}
Return X

FindKey(R,F):
Return Minimize(R,F)
```

הוא אכן מחזיר מפתח. הסדר שבו נרוץ על האלגוריתם משפיע על איזה מפתח נראה בתוצאה. Minimize מחזיר רק מפתח יחיד. אז איך נוכל למצוא את כלל המפתחות? זאת ועוד בחלק הבא.

תרגול 8

סיימנו לדבר תכנון שאילתה והערכת גודל של צירופים.

תלות פונקציונאלית

<u>הגדרה:</u> התלות הפונקציונאלית $X \to Y$ מתקיים במופע r אם **לכל** שתי שורות $s,t \in r$ מקיים שאם $s[X] = t[X]$ אזי $s[Y] = t[Y]$. כלומר הערך שכתוב בX קובע פונקציונאלית את הערך בY.

<u>הגדרה שקולה (על דרך השלילה):</u> $X \to Y$ מתקיים במופע r אם **לא קיימות** שורות $s,t \in r$ כך ש$s[X] = t[X]$ אבל $s[Y] \neq t[Y]$. כלומר אם לא מצאנו לה סתירה.

סגירות

הסגור של A, A^+, זה כל העמודות אליהן ניתן להגיע בעזרת תלויות פונקציונליות מA.

אם בסגור של AB יש את כל R, אז הוא מפתח על.

הרצאה 9 – Normal Forms

מציאת כל המפתחות

אנחנו יודעים למצוא מפתח אחד עבור יחס, ובעת נרצה למצוא את כולם.
נעזר באלגוריתם למציאת מפתח בודד, במקבל קבוצה של עמודות X (למעשה אנחנו נשלח את R כולו) ומנסה להוריד עמודות X כל עוד הורדתם לא מפריעה לסגור. כלומר נוריד את A מ-A אם A נמצא בסגור של $\{X - A\}^+$.

האלגוריתם למציאת כל המפתחות:

```
All Keys(R,F):
K := FindKey(R,F)
KeyQueue := {K}
Keys := {K}
While KeyQueue.isNotEmpty()
    K := KeyQueue.dequeue()
    Foreach X→Y ∈ F for which Y ∩ K is not empty do
        S:=K-{Y} ∪ X          //S is a superkey!
        If S does not contain any J ∈ Keys then
            S' := Minimize (S,F) //S' is a new key
            Add S' to Keys and to KeyQueue
Return Keys
```

כמות המפתחות בR יכולה להיות אקספוננציאלית. לכן זמן הריצה של האלגוריתם לא יכול להיות פולינומיאל. הוא רץ בזמן ריצה פולינומיאלי בקלט (R,F) ובפלט (כמות המפתחות בR). זה אלגוריתם מאוד יעיל, כי במקרים בהם יש קצת מפתחות זמן הריצה הוא קצר, ובמקרים בהם יש הרבה מפתחות הוא רץ בזמן ארוך- אבל אי אפשר לעשות טוב יותר מזה.
אטריביוטים שלא מופיעים בכלל בצד ימין תמיד יהיו חלק מהמפתח.

צורות נורמליות

הן צורות שבהן היינו רוצים לראות את הטבלאות כדי לוודא שהן בנויות בצורה הגיונית.
דיברנו על צורה נורמלית ברשימה ($1^{st}\ Normal\ Form$)- שבכל עמודה יש ערך בודד ואין חזרה על עמודות. כעת נרצה לדבר על שני סוגים חזקים יותר של צורות נורמליות: 3NF וגם BCNF=4NF. על הצורה נורמלית השנייה לא נדבר כי היא לא מעניינת באופן מעשי.
הצורות הנורמליות באות כדי לפתור את בעיית היתירות בטבלאות, שקורית כשיש ערכים שרשומים בה בלי צורך (שכפולים של ערכים אחרים שיכולנו להבין אותם בלי שהם יהיו כתובים).
ראינו בשבוע שעבר שערכים כאלו דורשים יותר זיכרון לשמירה, ויכולים לעשות בעיות בעדכון.
נגדיר שערך בטבלה הוא **מיותר** אם אפשר היה להסיק אותו בעזרת השורות האחרות בהינתן התלויות הפונקציונאליות.
הנושא של תלויות פונקציונאליות עוסקת בבעיית היתירות של נתונים, ומנסה למנוע מצב בו יש ערכים מיותרים בטבלאות.
צריך לזכור שיחסים הם שורות ללא חזרות, וזה משפיע על האם יכולה להיות יתירות. אם צד שמאל של התלות הוא מפתח, לא יכולה להיות יתירות. כשצד שמאל של התלות הוא לא מפתח, יכולה להיות יתירות.

BCNF Boyce-Codd Normal Form

<u>הגדרה:</u> R הוא בצורה נורמלית BCNF אם לכל תלות $X \to Y$ **שנובעת מF** מתקיימת אחת משתי התכונות:

1. התלות היא טריוויאלית (כלומר $Y \subseteq X$)
2. צד שמאל (X) הוא מפתח על של R

זה לא נותן דרך נוחה לבדוק מהי הצורה הנורמלית הנוחה של היחס, כי צריך בשביל זה לבדוק את כל התלויות שנובעות מF, ולא רק את F עצמה.

למזלנו, <u>הגדרה שקולה</u> היא שR הוא בצורה נורמלית BCNF אם לכל תלות $X \to Y$ **בF** מתקיימת אחת משתי התכונות:

1. התלות היא טריוויאלית (כלומר $Y \subseteq X$)
2. צד שמאל (X) הוא מפתח על של R

אפשר להוכיח שההגדרות זהות. זו הגדרה נחמדה כי הבדיקה של זה היא פולינומיאלית.
בשאין יתירות, אלו יחסים בBCNF (ובשיש הם לא בה).

Third Normal Form

זה צורה נורמאלית חלשה יותר.

<u>הגדרה:</u> R היא ב3NF אם לכל תלות $X \to Y$ **שנובעת מF** מתקיים אחד משני התנאים הבאים:

1. צד שמאל (X) הוא מפתח על של R
2. לכל $A \in Y$, או ש$A \in X$ או שA שייך לאיזשהו מפתח

זה לא נותן דרך יישומית לבדוק מה <u>נמצא</u> בה אלא מדבר על מה נובע ממנה.

למזלנו, <u>הגדרה שקולה</u> היא R היא ב3NF אם לכל תלות $X \to Y$ **ב**F מתקיים אחד משני התנאים הבאים:

1. צד שמאל (X) הוא מפתח על של R
2. לכל $A \in Y$, או ש$A \in X$ או שA שייך לאיזשהו מפתח

הבדיקה של 1 היא פולינומיאלית, וגם הבדיקה אם $A \in X$. אבל הבדיקה אם A הוא חלק ממפתח היא בעיה NP שלמה. לכן סה"כ הבדיקה האם R הוא בצורה נורמלית שלישית זו בעיה NP שלמה.
אבל לפעמים נרצה לבדוק את ע"י הפעלה של האלגוריתם למציאת כל המפתחות.
בדאי לשים לב שאם יחס הוא בBCNF אז הוא גם ב3NF, אבל ההפך לא מתקיים.

לסיכום, השוואה:

	BCNF	3NF
סיבוכיות	PTIME	NP-compelte
יתירות	אין	יכול להיות קצת
Achievable with Lossless Join Decompositions	כן	כן
Achievable with Lossless Join, Dependency Preserving Decompositions	לא	כן

ה"לא" בשורה התחתונה הוא הסיבה שיש את 3NF.
אז כעת אנחנו יודעים להכריע האם סכמה מתוכננת היטב. כעת נלמד לתכנן סכמה "לא טובה".

<u>פירוקים Decompositions</u>

היינו רוצים שכל היחסים שלנו יהיו בBCNF, או לפחות ב3NF. מה נעשה אם זה לא המצב? ננסה להבין מהם פירוקים (מהו פירוק ואילו תכונות הוא צריך לקיים) ואיזה סוגי פירוק הם טובים.
ראשית, ננסה להבין באיזה צורה נורמאלית היחס שלנו נמצא. אם הוא לא באף צורה, היינו רוצים לשפר את הטבלה כדי שתהיה בצורה נורמאלית טובה יותר. השיפור נעשה ע"י פירוק היחס לתתי-יחסים כדי להקטין את כמות יתירות המידע שיש בטבלה, ואז צריך לדון על האם הפירוק הוא טוב.
הוכחנו בתרגול שכל יחס עם שתי עמודות הוא בBNCF ולכן הפירוק בדוגמא במצגת הוא טוב.
כעת, צריך לראות איך אפשר לשחזר את היחס המקורי. כאן, צירוף טבעי יחזיר את היחס המקורי.
בנוסף, צריך לוודא שהתלויות הפונקציונאליות שהתקיימו ביחס המקורי יתקיימו גם כאן כעת ולאחר הצירוף.
כלומר ראינו כאן כמה דברים טובים:

1. הפירוק יצר 2 טבלאות בצורת BCNF
2. ניתן לשחזר את היחס המקורי ע"י פעולת הצירוף
3. היה אפשר לוודא שתלויות פונקציונאליות מתקיימות בזמן ההכנסה

האם כל פירוק הוא טוב? בדוגמא של הפירוק השני, לא ניתן לבדוק בזמן ההכנסה שהתלויות הפונקציונאלית מתקיימות. בדוגמא של הפירוק השלישי, לא ניתן לשחזר את R.

באופן פורמלי, **פירוק של R** הוא קבוצה של יחסים $R_1, \dots, R_n$ כך שסך העמודות ב$R_1, \dots, R_n$ זהות לאלו שבR, ונתעניין ב3 תכונות:

1. (הכרחית) ללא אובדן lossless – נוכל לשחזר את היחס המקורי ע"י צירוף טבעי
2. (רצויה) שימור תלויות dependency preserving- רוצים להיות מסוגלים לבדוק שהתלויות הפונקציונאליות מתקיימות על תתי-הסכמות כדי להבטיח אותן על היחס המלא אחרי הצירוף
3. (רצויה) כל תת סכמה תהיה בBCNF או ב3NF

כעת ננסה להבין איך בודקים עבור כל אחת מהתכונות אם היא מתקיימת, ואז נוכל להבין איך מייצרים פירוק שמייצר את שלושתן.

Lossless Join (2 Relations)

נדבר על התכונה של היות היחס החדש ללא אובדן, כלומר אפשר לחשב את היחס המקורי מהיחסים שפירקנו, ע"י צירוף טבעי.

הגדרה: פירוק של יחס R לתתי יחסים $R_1, \dots, R_n$ הוא **ללא אובדן** אם יחסית לקבוצת התלויות הפונקציונאליות F אם לכל מופע r של R שמקיים את F מתקיים $r = \pi_{R_1} r \bowtie \cdots \bowtie \pi_{R_n} r$.

אם התכונה הזו מתקיימת, נוכל לשמור רק את ההטלות של r על כל אחד מתתי היחסים, בלי לשמור את r כולו, ולהיות בטוחים שניתן לשחזר בחזרה את r המקורי. זו תכונה הכרחית.

נשים לב שהוכחנו (כשדיברנו על ביטויים אלגבריים) ש$r \subseteq \pi_{R_1} r \bowtie \cdots \bowtie \pi_{R_n} r$ תמיד מתקיים, והכיוון השני הוא זה שלא תמיד מתקיים.

יש מקרה מיוחד, בו הפירוק הוא לשני יחסים (הוא קל לבדיקה ולכן נתחיל מלדבר עליו).

הגדרה: פירוק של R לR_1, R_2 הוא ללא אובדן יחסית לF אם לפחות אחד משני התנאים מתקיימים:

1. $R_1 \subseteq (R_1 \cap R_2)^+$
2. $R_2 \subseteq (R_1 \cap R_2)^+$

אם נרצה לבדוק בהינתן שניהם אם החיתוך הוא ללא אובדן, נצטרך רק לחשב את החיתוך ביניהם ואת הסגור שלו, ולראות אם אחד מהם מוכל בו.

למה זה נכון? למה זה מבטיח $\pi_{R_1} r \bowtie \cdots \bowtie \pi_{R_n} r \subseteq r$? הוכחה במצגת.

Lossless Join (N Relations)

כעת נרצה להבין איך פירוק ליותר משני תתי יחסים יכול להיות פירוק ללא אובדן.

כעת נרצה למצוא תנאי כאשר הפירוק הוא למספר כלשהו של תתי יחסים.

כדי לעשות את זה:

1. נייצר טבלת בגודל $n \times k$ עבור k מספר היחסים וn מספר העמודות

```
CreateTable(R=(A1,...,Ak),R1,...,Rn):
T := new Table[n,k]
for i = 1 to n
   for j = 1 to k
       if Aj∈Ri
           then T[i,j] = aj
            else T[i,j] = bij
return T
```

2. נתקן את התלויות הפונקציונאליות בה עד שהן יתקיימו

```
ChaseTable(T,F):
While there are rows t, s in T and X→Y in F such that
            t[X]=s[X] and t[Y]≠s[Y]
  for each A_i in Y do
      if t[A_i]=a_i then replace s[A_i] with a_i
      else if s[A_i]=a_i then replace t[A_i] with a_i
      else replace t[A_i] with s[A_i]
```

3. בודקים האם יש שורה שכולה a- אם כן, הפירוק הוא ללא אובדן. אחרת, הוא עם אובדן.

```
TestDecomposition(R,R_1,...,R_n,F):
T = CreateTable(R,R_1,...,R_n)
ChaseTable(T,F)
If T contains a row with only "a" values
   return "lossless"
else return "not lossless"
```

מה שמיוחד בטבלה הזו זה שהיא מנסה לייצר דוגמא נגדית להיותו של הפירוק ללא אובדן, ואם היא לא תהווה דוגמא נגדית- היא תוכיח את זה שהפירוק הוא ללא אובדן.
אם נעשה הטלה ואחר כך נצרף בחזרה, נקבל שורה שכולה a.
תיקנו את המופע שלנו בעזרת התלויות הפונקציונאליות כדי לוודא שיש לנו יחס תקין, שמקיים את כל התלויות. לאחר התיקון, אם נעשה על היחס הטלה וצירוף, נקבל בחזרה שורה שכולה a.
אם סיימנו את האלגוריתם עם יחס שבו לא הייתה שורה שכולה a מצאנו דוגמא לטבלה שמקיימת את התלויות ולא מכילה שורה שכולה a, שכשנעשה עליה הטלה וצירוף, נמצא שורה שלא הייתה ביחס המקורי (שכולה a). זו דוגמא נגדית שמראה ש $r \neq \pi_{R_1} r \bowtie \cdots \bowtie \pi_{R_n} r$.
אם לעומת זאת ביחס שבנינו יש שורה שכולה a, אז למעשה התהליך שהאלגוריתם עובר בתיקון התלויות הפונקציונאליות יכול לשמש לסימלוץ הוכחה ששורה נמצאת בתוצאה של צירוף ההטלות אם"ם הייתה ביחס המקורי.

תרגול 9

מפתח של דיאגרמות ER ומפתח בתלויות פונקציונאליות מתנהגים אותו דבר.
נדבר על מציאת כל המפתחות. זה חשוב להגדרת צורות נורמאליות.
העיקרון מאחורי האלגוריתם למציאת כל המפתחות זה שאם E הוא מפתח ו $X \to E$, אז X הוא מפתח על. כך ננסה להחליף את E בעמודות אחרות ולנסות לקבל מפתחות נוספים, בעזרת בניית עץ החלפות, שיסתיים כשלא יהיו עוד ענפים שניתן להמשיך אותם בעץ.

צורה נורמאלית

יש לנו 2 צורות נורמאליות רצויות, ואנחנו מדברים על איך לבדוק אם יחס הוא באחת מהצורות.
הגדרה: R הוא בצורה נורמלית BCNF אם לכל תלות $X \to Y$ בF מתקיימת אחת משתי התכונות:

1. התלות היא טריוויאלית (כלומר $Y \subseteq X$)
2. צד שמאל (X) הוא מפתח על של R

הגדרה: R היא ב3NF אם לכל תלות $X \to Y$ בF מתקיים אחד משני התנאים הבאים:

1. צד שמאל (X) הוא מפתח על של R
2. לכל $A \in Y$, או ש $A \in X$ או שA שייך לאיזשהו מפתח

נשים לב ש $BCNF \subset 3NF$.
צריך תלות אחת שסותרת את התנאי כדי להוכיח שאנחנו לא בBCNF או לא ב3NF (בלומר כדי להיות באחד מהם צריך שזה יתקיים לכל התלויות).
טענה: נניח שR הוא יחס שבו יש בדיוק שתי עמודות, ונניח שF היא קבוצת התלויות הפונקציונאליות. אזי R הוא בBCNF.
הוכחה: נחלק למקרים של כל האופציות שיכולות להיות בF ונראה שלכולן זה מתקיים.

נניח $R = (A, B)$ וקבוצת תלויות F. אם $F = \emptyset$ אז R בBCNF כי אין תלות שסותרת. אחרת, בF יכולות להיות מלבד התלויות הטריוויאליות (שאינן סותרות BCNF) רק התלויות הבאות: $A \to B, B \to A, A \to AB, B \to AB$. בכל אחת מהתלויות האלו צד שמאל הוא מפתח על ולכן אין סתירה לBCNF. מכאן נובע שR בBCNF.

<u>טענה (לא נכונה):</u> נניח שR הוא יחס, ונניח שF היא קבוצת התלויות הפונקציונאליות. נניח שכל מפתח בF הוא עמודה יחידה. אזי R הוא בBCNF.

<u>דוגמא נגדית:</u> $R = (A, B, C), F = A \to BC, B \to C$.

<u>טענה:</u> נניח שR הוא יחס, ונניח שF היא קבוצת התלויות הפונקציונאליות. נניח שכל מפתח בF הוא עמודה יחידה. אזי R הוא בBCNF אם ורק אם R הוא ב3NF.

<u>הוכחה:</u> נניח שR יחס עם קבוצת תלויות F ונניח שכל מפתח בR הוא עמודה בודדת.

- $\Leftarrow$ אם R בBCNF אז R גם ב3NF לפי הגדרה
- $\Rightarrow$ אם R ב3NF, נסתכל על כל תלות בF שאינה טריוויאלית $X \to A$. אם X הוא מפתח על אז אין סתירה לBCNF, אחרת חייב להיות שA נמצא במפתח. מכיוון שכל מפתח הוא עמודה בודדת, נקבל שA מפתח וX הוא מפתח על. לכן אין סתירה לBCNF בכל מקרה, וR בBCNF.

<u>טענה (לא נכונה):</u> נניח שR הוא יחס, ונניח שF היא קבוצת התלויות הפונקציונאליות. נניח שלכל תלות בF, בצד שמאל יש רק עמודה אחת. אזי R הוא בBCNF אם ורק אם R הוא ב3NF.

<u>דוגמא נגדית:</u> $R = (A, B, C), F = A \to B, B \to A$, אזי המפתחות הם AC וBC, לכן R רק ב3NF.

הרצאה 10 – Decompositions פירוקים

<u>שימור תלויות</u>

דיברנו על איך בודקים אם פירוק הוא ללא אובדן. כעת נעבור לתכונה הבאה של פירוק – שימור תלויות. כלומר נוכל לבדוק שתלויות נשמרות בתתי הסכמות בלי שנצטרך לצרף אותן כדי לבדוק את שימור התלויות.

נגיד שפירוק הוא **משמר תלויות** אם מספיק לבדוק שכל אחת מהתלויות המוגדרות לוקאלית (על תתי היחסים) מתקיימות, כדי שהתלויות יתקיימו על כלל היחס.

נגדיר מהן **התלויות שמוגדרות מעל כל אחד מתתי היחסים**. נתון היחס המקורי R, קבוצת תלויות פונקציונאליות F ותתי-יחסים $R_1, \dots, R_n$. התלויות שאמורות להתקיים מעל $R_1, \dots, R_n$ הן:

נגדיר **הטלה של F על תת סכמה $\mathbf{R_i}$**- אוסף התלויות הפונקציונאליות שנובעות מF ומתייחסות רק לעמודות בR_i. כלומר $F_{R_i} = \{V \to W \; s.t. V, W \subseteq R_i \; and \; V \to W \; follows \; from \; F\}$.

הקבוצה הזו יכולה להיות גדולה מאוד (אפילו אקספוננציאלית), כי מכניסים גם את כל מה שנובע מF, גם אם הוא טריוויאלי.

נגדיר באופן פורמלי מתי פירוק משמר תלויות: הפירוק של R לתתי היחסים $R_1, \dots, R_n$ משמר תלויות אם לכל $X \to Y$: $X \to Y$ נובע מF אם ורק אם $X \to Y$ נובע מ$(F_{R_1} \cup \dots \cup F_{R_n})$.

נשים לב שמספיק לבדוק את התלויות בF עצמו ולא בכל מה שנובע ממנו.

בפועל, זה אומר שכשנכניס מידע לטבלה R_i נבדוק שהיא מקיימת את התלויות בF_{R_i}, וזה יבטיח שכשנצרף בחזרה את השורות נקיים גם את F כולה.

חשוב לשים לב שאם X,Y נמצאים בעצמם בR_i כלשהו אז בוודאי ש$X \to Y$ נשמר.

אלגוריתם לחישוב שימור תלויות:

```
IsDependencyPreserving(R,R1,..,Rn,F):
For each X→Y in F do:
        Z:= X
        repeat
                Z' := Z
                for i = 1 to n do
                        Z := Z ∪ ((Z ∩ Ri)+ ∩ Ri)
        until Z = Z'
        if Y is not contained in Z then return "NO"
Return "YES"
```

הוא מקסים כי הוא רץ בזמן פולינומיאלי, בלי לחשב את $F_{R_1} \cup \ldots \cup F_{R_n}$ שיכול להיות ענק.
השורה הארוכה מחשבת את הסגור של Z יחסית לתלויות בR_i.
הסדר של המעבר על תתי הסכמות לא חשוב, כי חוזרים על הפעולה שוב ושוב.
יכול להיות פירוק ללא אובדן ובלי שמירת תלויות, וגם להפך.

Decomposition Normal Forms

נרצה לדון בצורה הנורמלית של היחסים בפירוק. נשים לב שכדי לדעת מהי הצורה הנורמלית של היחס R_i, אנחנו צריכים לדעת מהן התלויות F_{R_i} שמוגדרות מעליו. לחשב את כל התלויות האלו יכול לקחת זמן אקספוננציאלי, נעשה את זה בכל מקרה.
זו פרוצדורה שמחשבת קבוצה ששקולה לF_{R_i}:

```
ComputeDependenciesInProjection (R,Ri,F):
G := ∅
For each X ⊆ Ri do:
        Add the dependency X→(X+ ∩ Ri)  to G
```

זהו אכן אלגוריתם אקספוננציאלי. יש אפשרות לעשות פחות חישובים, אבל לפחות בהתחלה עדיף לעשות את החישוב במלואו בלי קיצורי דרך.

ביסוי מינימלי

אנחנו רוצים למצוא פירוק ללא אובדן, משמר תלויות ועם צורה נורמלית טובה. לצורך זה נצטרך את המושג ביסוי מינימלי. כשמקבלים קבוצה של תלויות פונקציונאליות, יכול להיות שחלק מהתלויות מיותרות כי הן מכילות תלויות אחרות. **ביסוי מינימלי** מכיל אך ורק את התלויות הפונקציונאליות ההכרחיות- שלא נובעות מהתלויות האחרות, ושאם נוריד אותן נאבד מידע.
הקטנה של הביסוי יכולה להיות גם ע"י הורדה של אטריביוט מתלות, ולא רק הורדה של תלות.
הגדרה: **ביסוי מינימלי** לקבוצה של תלויות פונקציונאליות F היא קבוצה של תלויות פונקציונאליות G, כך ש:

1. **צד ימין הוא קטן**- לכל תלות בG יש רק אטריביוט אחד בצד ימין
2. **שקילות לF**- כל תלות בG צריכה לנבוע מF ולהפך
3. **אי אפשר להקטין את G יותר ע"י מחיקות נוספות** – אם נקבל את H מG ע"י מחיקת תלות אחת או יותר, או ע"י מחיקת אטריביוט אחד או יותר, אזי H כבר לא יהיה שקול לF

אלגוריתם למציאת ביסוי מינימלי:

```
ComputeMinimalCover(F):
G := ∅
for each X→Y in F
   for each A in Y
        add X→ A to G
for each X → A in G
   for each B in X
        if A ∈ (X-B)+ then  remove B from X → A
for each X → A in G
        if X → A follows from the other dependencies
        then remove X → A
```

אלגוריתם פולינומיאלי שרץ ב3 שלבים:

1. נדאג לזה שבכל תלות יהיה רק אטריביוט אחד בצד ימין
2. נמחק אטריביוטים מיותרים מצד שמאל של תלויות
3. מורידים תלויות מיותרות (אם יש תלות שהיא היחידה בה אטריביוט מסוים מופיע בצד ימין, היא אף פעם לא תהיה מיותרת)

פירוק 3NF

כעת אנחנו יכולים בהנתן סכמה למצוא פירוק שלה שהוא ללא אובדן, משמר תלויות ועם צורה נורמלית שלישית. אלגוריתם שעושה את זה:

```
Find3NFDecomposition(R, F):
G := ComputeMinimalCover(F)
for each X→A in G
   add the schema XA
If no schema created contains a key, add a key as a schema
Remove schemas that are contained in other schemas
```

לבדוק אם סכמה היא ב3NF זו בעיה NP קשה, אבל זה אלגוריתם פולינומיאלי שמוצא פירוק ל3NF (אם הוא מקבל יחס שכבר ב3NF, יכול להיות שהוא עדיין יחזיר פירוק שלו ולא אותו).

למה האלגוריתם עובד?

1. משמר תלויות- כל תלות נמצאת בתוך תת סכמה בכה שהיא חייבת להישאר
2. ללא אובדן – כל תלות הפכה להיות תת סכמה, ויש גם תת סכמה שהיא מפתח על
3. כל תת סכמה היא לפחות ב3NF- נשים לב שאנחנו מייצרים תתי סכמות:
   a. מתוך תלויות: נניח ש$X \to A, R_1 = XA$. יהי $Y \to B$ תלות בF_{R_1}. בR_1, בהכרח X הוא מפתח.
      i. מקרה א': $B \in X$. אם כך, B שדה במפתח
      ii. מקרה ב': $B \notin X$. אז $B = A$. אם כך, $Y = X$. (אחרת, אם $Y \subsetneq X$ אז $X \to A$ לא היה מינימאלי). Y הוא מפתח.
   b. כי היא מפתח- קל להוכיח שזה בצורה הרצויה

פירוק BCNF

נראה אלגוריתם למציאת פירוק שהוא ללא אובדן וכל אחת מתתי הסכמות היא בBCNF, אבל לא נוכל להבטיח שהפירוק משמר תלויות.

הנה אלגוריתם: הוא לא פולינומיאלי, אבל קיים אלגוריתם פולינומיאלי שפותר את הבעיה. הוא מסובך ולכן נעדיף את זה.

```
FindBCNFDecomposition(R, F):
If R is in BCNF
   then return R
   else let X→Y be a BCNF violation
      R1 = X+
      R2 = X ∪ (R-X+)
      return FindBNCFDecomposition(R1,FR1) ∪
             FindBNCFDecomposition(R2,FR2)
```

תרגול 10

אנחנו רוצים למצוא פירוק שהם בצורות הנורמאליות שלמדנו, כך שכשנצרף את הטבלאות האלו לא נרצה שיופיעו שורות שלא היו קודם (ללא אובדן) ונרצה שהתלויות שלנו ישמרו.

בדיקה שאין אובדן- ע"י מילוי הטבלה כמו שלמדנו (שורה לכל יחס, עמודה לכל אטריביוט, שינוי בהתאם לתלויות). אם יש אובדן- נעבור על כל התלויות ולא נוכל לעדכן עוד את הטבלה, אבל לא תהיה שורה שכולה a.
בדיקה ששומר תלויות- האלגוריתם עובר דרך התלויות ובודק לכל אחת האם היא נשמרת בכל היחסים שפירקנו.

איך מוצאים פירוק ל3NF? בעזרת **מציאת ביסוי מינימלי**:

1. מפרקים תלויות כך שבתלות יהיה רק אטריביוט אחד בצד ימין
2. נמחק אטריביוטים מיותרים מצד שמאל של תלויות
   a. מותר להשתמש בתלות אותה אנחנו בודקים
3. מורידים תלויות מיותרות
   a. אסור להשתמש בתלות אותה אנחנו בודקים

ואז:

- אם אין מפתח שמוכל באף תת סכמה, תוסיף מפתח כתת סכמה
- תסיר תת סכמות שמוכלות ממש בתתי סכמות אחרות

הרצאה 11 - Framework (Transaction Management)

מבוא

עד עכשיו חשבנו על הגישה לDB כאוסף פקודות שמישהו מקליד בחלון של המערכת. למעשה, כל תוכנית שעובדת מול הDB עובדת בצורה של אוסף פקודות שמדי פעם ניגשות למערכת הDB ומפעילות פעולות של עדכון/ הכנסה/ בחירה/ מקבלות את הערכים ועושות איתם משהו. יש אוסף של תוכניות שרצות מול מערכת הDB כל הזמן ומנסות להשיג משהו.
נדון בשאלה מה יכולות להיות הבעיות שקורות במערכת כזו, ואיך אפשר לדאוג לזה שלא יהיו בעיות בגלל ריצה של מספר תוכניות מול מערכת הDB.
אילו בעיות יכולות להיווצר כשמריצים תוכניות מול מערכת הDB? סוגים:

1. חישוב חלקי של התוכנית
   - התוכנית רצה ומתישהו באמצע בגלל בעיה כלשהי/משהו בלתי צפוי התוכנית נכשלת ומפסיקה לעבוד. יכול להשאיר את הנתונים בצורה לא הגיונית.
   - נשתמש במנגנון שנקרא מנגנון הtransaction שיבטיח שאף פעם אין ביצוע חלקי של תוכניות, אלא הן תמיד מתבצעות במלואן.
   - <u>התכונה שנרצה להבטיח היא atomicity</u>- התוכנית תבוצע בצורה אטומית, לא יהיה ביצוע חלקי.
2. באג בתוכנית
   - יכול לגרום להכנסת שגיאות לDB.
   - לא נתמקד בזה, זו בעיה שמערכת הDB לא תוכל להתמודד איתה, אלא בעיה של המתכנת.
   - <u>התכונה שנרצה להבטיח היא consistency</u>.
3. יש הרבה תוכניות שרצות בו זמנית מול מערכת הDB
   - הרבה אנשים מנסים לגשת בו זמנית, ותוכניות יכולות להפריע אחת לשנייה או לראות נתונים שקשורים למשהו שתוכנית אחרת מריצה.
   - מנגנון הtransactions יאפשר לדאוג לכך שתוכניות שונות שרצות בו זמנית לא יפריעו אחת לשנייה.
   - <u>התכונה שנרצה להבטיח היא isolation</u>- תוכנית אחת לא תראה את האפקטים של תוכניות אחרות שרצות בו זמנית, אלא יהיה ממש כאילו היא רצה לבד.
4. נפילת המערכת

- אחרי שינויים וסיום בהצלחה- נפילת חשמל, ולכן חשבנו שעשינו משהו (רכשנו כרטיס טיסה) אבל בפועל המידע לא נשמר במערכת הDB.
- מערכת הDB תדאג שעדכונים לא ילכו לאיבוד. נדבר על זה בהמשך הקורס, כשנדבר על מנגנון של התאוששות מנפילות.
- התכונה שנרצה להבטיח היא durability - גם במקרה של נפילה של המערכת, שינויים שעשינו ישמרו

נדבר על הצורה בה מערכת הDB דואגת לזה שהבעיות האלו לא יקרו.
אם המתכנת דואג לתכונת הקונסיסטנטיות (אין באגים בתוכנית) אז מערכת הDB תבטיח את 4 התכונות- ACID.

Transactions

בשביל להגיד למערכת הDB שיש אוסף פעולות שמבחינה לוגית צריכות לקרות יחד, נשתמש במנגנון transaction- תשואה/עסקה, שאומר לDB שיש יחידות שצריכות להתבצע במלואן או לא להתבצע במלואן, כדי שמערכת הDB תדע מה למחוק כשמתאוששת משגירות.
זו דרך להגדיר שיש אוסף פעולות שקורות יחד, ותוכנית הביצוע צרירה לחשוב עליהן כאילו קורות ביחד, בצורה מנותקת משאר הפעולות שרצות על הDB.

- כדי להגדיר transaction נכתוב ;BEGIN TRANSACTION לפני הפעולות.
- בסיום נכתוב ;COMMIT אם נרצה לציין שכל הפעולות הסתיימו והשינויים צריכים להישמר בDB.
- נוכל לכתוב ;ROLLBACK במקרה שרואים ערך לא תקין.
  אם מתבצעת שגיאה כלשהי, מערכת הDB באופן אוטומטי עושה את פעולת הROLLBACK- חזרה למצב שהיה לפני.
- אפשר גם לסיים transaction עם ;END TRANSACTION ששקול לcommit.

Postrgres מודיע כשהוא עושה ROLLBACK.
מנגנון הtransactions יבטיח לנו אטומיות.

ראינו בעיה שקרתה כי הייתה תוכנית שהתבצעה לא בצורה מלאה (העברת כסף ללא ביסוי).
בנוסף, יכולות להיות בעיות של מקביליות, שנובעות מזה שכמה אנשים ניגשים בו זמנית למערכת הDB. **סוגים של בעיות מקביליות**:

1. **Dirty write**- transaction אחת כותבת מידע שנכתב כבר ע"י transaction נוכחית אחרת שרצה ועוד לא עשתה COMMIT
2. **Dirty read**- transaction קוראת ערך שהשתנה ע"י transaction אחרת שעוד לא עשתה COMMIT (ואחר כך עושה ROLLBACK לדוגמא)
3. **Non-repeatable reads**- טרזנקציה אחת קוראת ערך שהיא כבר קראה ומגלה שהוא השתנה ע"י טרזנקציה אחרת (מאוד בעייתי כי אי אפשר להסתמך על זה שהערכים שקראנו הם הערכים האמיתיים).
4. **Phantom reads** – transaction מבצעת שאילתה ומקבלת קבוצה של ערכים כתגובה, וכשמבצעת את השאילתה שוב מקבלת קבוצה שונה של שורות. יכול להיות שהערכים עצמם לא השתנו, אבל התווספה שורה למשל.
5. **Serialization anomalies** – תופעה שקורית מהרצה מקבילית, ששונה ממה שהיה קורה בהרצה סדרתית של הפעולות.

ננסה להבין איך מערכת הDB משיגה את תכונת הבידוד.

בידוד

רוצים שתוכניות שונות שרצות מול הDB (טרנזקציות שונות) יהיו מבודדות מהאפקטים אחת של השנייה. איך אפשר להשיג את זה?

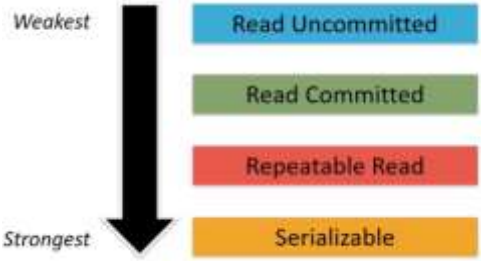
- לא להריץ טרנזקציות שונות בו זמנית

- הכי קל ובהכרח נכון
- אבל אז צריך לחכות שטרנזקציה תסיים כדי להתחיל לרוץ, מה שיאט את הביצועים שלנו

שימוש במנגנון בקרת מקביליות שידאג לזה שהטרנזקציות לא ירגישו אחת את השנייה ובכל זאת ירוצו במקביל. בגדול יש 2 סוגים של בקרות מקביליות

- בכל ערך שומרים גרסה אחת של כל ערך וכל שורה ומתמרנים איכשהו
- שמירת הרבה העתקים של אותם פריטים מידע, וכל טרנזקציה תשתמש בהעתק אחר
  - בסוף נדאג לזה שהמידע יעודכן כמו שצריך לDB

למעשה, בידוד זה לא הכל או כלום. יש בSQL 4 רמות בידוד שונות, שמגדירות עד כה חשוב שהיא תהיה מבודדת. ככל שנבודד יותר, המקביליות תפגע אבל נוכל לשמור על יתר המבניות בDB כי נוכל להיות בטוחים שאין השפעה של טרנזקציה אחת על אחרת.



מערכת DB שונות ממשות את הרמות בדרכים שונות, ואת מה שהן מבודדת. זה מוזר אבל דורש זהירות רבה שלנו.

רמות הבידוד בPostgres:

Level	Dirty Write	Dirty Read	Unrepeatable Read	Phantom	Serialization Anomoly
READ UNCOMMITTED	No	Possible, not in Postgres	Possible	Possible	Possible
READ COMMITTED	No	No	Possible	Possible	Possible
REPEATABLE READ	No	No	No	Possible, not in Postgres	Possible
SERIALIZABLE	No	No	No	No	No

כשנדבר על איך מערכת הDB דואגת לבידוד, נדבר רק על איך היא דואגת לרמת הserializable.
הדיפולט בPostgres, SQLServer, Oracle הוא שאם כותבים transaction, רצים ברמת בידוד של read committed. בגלל זה יכולות להיווצר שגיאות בנתונים. בMySQL הדיפולט הוא repeatable read. חשוב לשים לב באיזו רמת בידוד אנחנו רוצים לרוץ, ולהגדיר את זה.
כל רמות הבידוד מעבר לserializable מבוססות עליו, והן לא נדרשות לקורס.
כדי להגדיר את רמת הבידוד של הטרנזקציה, מגדירים אותה אחרי הגדרת הטרנזקציה.

```
BEGIN TRANSACTION
ISOLATION LEVEL ....
queries, updates, ...
COMMIT | ROLLBACK | END TRANSACTION
```

בנוסף, ניתן להגדיר מה תהיה רמת הבידוד לכל הטרנזקציות הבאות שיגיעו.

```
SET TRANSACTION
ISOLATION LEVEL { SERIALIZABLE |
                  REPEATABLE READ |
                  READ COMMITTED |
                  READ UNCOMMITTED }
READ WRITE | READ ONLY
```

איך מערכת הDB משיגה את הבידוד בעזרת העתק אחד של כל ערך (Single-Version Concurrency Control)? ניתן הגדרות תיאורתיות כדי להבין מה אנחנו רוצים להשיג.

תיזמונים

ניתן הגדרות פורמליות כדי להוכיח מה מבטיחים ומה לא מבטיחים במערכת הDB.
הגדרה: **טרנזקציה** transaction היא ביצוע אחד של תוכנית של המשתמש בתוך מערכת הDB.
כאבסטרקציה, נסתכל על טרנזקציה כאוסף של פעולות של קריאה וכתיבה של אובייקטים לDB (כשאובייקט יכול להיות ערך, שורה, דף, טבלה וכו').
הנחות מפשטות:

- נניח שטרנזקציות לא מדברות אחת עם השנייה, אלא רק יכולות לראות את האפקטים אחת של השנייה בתוך מערכת הDB.
- נניח שהDB הוא קבוצה קבועה של אובייקטים שהם בלתי תלויים אחד בשני.
- נניח שיש רק העתק אחד של כל אובייקט בDB. בכל פעם שטרנזקציה קוראת אובייקט מסוים, היא רואה את הערך האחרון שנכתב אליו.

אפילו אם טרנזקציה שינתה ערך ולא עשתה commit, שאר הטרנזקציות רואות את הערך הזה. בשלב הזה אין לנו פרוטוקול שמטפל בבידוד.

הגדרה: **תיזמון** של קבוצה של טרנזקציות $T_1, \dots, T_n$ הוא סידור של הפעולות של כל $T_1, \dots, T_n$ בצורה קונסיסטנטית עם שאר הטרנזקציות (בוחרים מי מתחיל, מי מתפרץ וכן הלאה).
אם כל הטרנזקציות בתיזמון עושות בסוף commit או abort, התיזמון הוא **תיזמון מלא**.
אם הטרנזקציות מתבצעות אחת אחרי השנייה בצורה מלאה (בלי להתפרץ) **התיזמון הוא serial סדרתי**.

למה זה נח שהטרנזקציות עושות interleaving – משתלבות זו בזו? כי זה מאפשר לנו לגשת בצורה מהירה יותר לנתונים, בעיקר כשאין קשר בין האובייקטים שכל טרנזקציה מטפלת בו.
למה תיזמון סדרתי הוא טוב? הוא מאוד נח ושימושי כי אפשר להיות בטוחים שהטרנזקציות רצות בבידוד אחת מהשנייה (כי הן לא רצות באותו זמן).

הנחת הריצה הסריאלית (הסדרתית): תיזמון הוא סריאלי אם הטרנזקציות רצות אחת אחרי השנייה ללא שילוב. ההנחה שלנו היא שאם המשתמש כתב תוכנית בלי באגים ונריץ אותה מעל DB תקין וקונסיסטנטי, הDB יישאר תקין וקונסיסטנטי.
אבל בפועל, חייבים להשתמש בשילוב בין הפעולות של הטרנזקציות כדי להשיג זמן ריצה טוב מספיק.
נרצה להבין מה בסדר מבחינת תיזמון ומה לא.

Serializable בר סידור

כל תיזמון סדרתי של טרנזקציות משאיר את המסד במצב קונסיסטנטי (מצב טוב).
הגדרה: נגיד שתיזמון של טרנזקציות שעושות commit הוא **בר סידור** אם האפקט שלו על הDB מובטח להיות זהה לאפקט של תיזמון סדרתי.
כלומר התוצאה של הריצה זהה לריצה סדרתית. זה מה שמובטח ע"י רמת הבידוד serializable, ולכן זה מה שנרצה להגדיר ולראות איך מסיקים.

מה יכול לגרום לתיזמון לא להיות בר סידור? אילו בעיות יכולות להיווצר?

- אם הטרנזקציות קוראות וכותבות לקבוצות שונות של אובייקטים- אין בעיה
- אם יש טרנזקציה שרק קוראת אובייקטים ולא כותבת אף פעם- אין בעיה
- הבעיה נוצרת כשטרנזקציה אחת כותבת לאובייקט A, וטרנזקציה אחרת קוראת/דורסת את האובייקט A- וזה יכול להיות שונה מריצה סדרתית

בעיות כאלו נקראות **קונפליקטים**. סוגים:

1. **קונפליקט כתיבה קריאה** מתרחש כשטרנזקציה T כותבת ערך A, ו'T קוראת את A. יכול לגרום לתיזמון להיות לא בר סידור.
2. **קונפליקט קריאה כתיבה** מתרחש כשטרנזקציה כותבת ערך שטרנזקציה קודמת קראה. אם הטרנזקציה הראשונה תנסה לקרוא שוב היא תקבל unrepeatable read. יכול לגרום לתיזמון להיות לא בר סידור.
3. **קונפליקט כתיבה כתיבה** מתרחש כש'T כותבת ערך על ערך אותו T כתבה.

בהמשך, נרצה למנוע ע"י הפרוטוקול את מה שיכול לגרום לקונפליקטים.

דיברנו על תיזמונים בהם טרנזקציות עושות commit. אם יש תיזמון בו חלק עושים commit וחלק עושים abort, נגיד שהתיזמון הוא **בר סידור** אם האפקט שלו על הDB זהה לריצה סדרתית של התיזמונים שעשו commit (כלומר אלו שמשנים את הDB).

<u>Recoverable בר התאוששות</u>

תכונה נוספת קריטית למסד. נרצה למצוא פרוטוקולים שהם גם ברי סידור וגם ברי התאוששות.
יכולות להיות בעיות מאוד חמורות עם טרנזקציות שרוצות לעשות abort.
יש תיזמונים בהם בלתי אפשרי לעשות roll back כי פעולות שעושות commit התבססו על הפעולות שרוצות לעשות abort.
<u>הגדרה:</u> נגיד שהתיזמון S הוא **בר התאוששות** אם ורק אם כל הטרנזקציות מבצעות commit רק אחרי שהטרנזקציות שאת השינויים שלהן הן קוראות עושות commit.
בתיזמון כזה יכול לקרות cascading aborts- הabort בו גורמים אחד לשני לקרות. נוכל לגרום לזה שהשינויים שהתבצעו אחר כך לא יישמרו.
<u>הגדרה:</u> נגיד שהתיזמון S **נמנע מcascading aborts** אם טרנזקציות קוראות רק שינויים של טרנזקציות שכבר עשו commit.

נדון בשאלות:

- איך נוכל לזהות בצורה מסודרת אם תיזמון הוא בר סידור?
- כשמקבלים בקשות קריאה וכתיבה בזמן אמת, איך מערכת הDB תוכל להפעיל פרוטוקול שהוא בר סידור ובר התאוששות, ושנמנע מcascading aborts? איך נוכל לדאוג בעזרת פרוטוקול לייצר רק תיזמונים טובים?

קודם נבין איך נוכל לזהות אם תיזמון הוא בר סידור וכו', ואז ניצור פרוטוקול מתאים.

<u>Conflict Serializable</u>

מעוניינים להיות מסוגלים לזהות האם תיזמון הוא בר סידור. כלומר שהאפקט שלו על הDB הוא זהה לאם היינו מריצים את הטרנזקציות אחת אחרי השנייה. אחרי שנוכל לזהות האם תיזמון הוא בר סידור, נוכל לגשת לשאלה- איך מערכת הDB תוכל לדאוג לכך שהתיזמונים שיווצרו בפועל ע"י הבקשות של הקריאה והכתיבה לנתונים בDB יהיו ברי סידור.
בשביל לזהות תיזמון בר סידור, נצטרך להזכר ברעיון של **קונפליקטים**. קופליקט מתבצע כאשר יש טרנזקציה אחת שכותבת/קוראת אובייקט A וטרנזקציה אחרת שכותבת/קוראת את אותו האובייקט A אחר כך. הקונפליקטים האלו הם אלו שיכולים לעשות לנו בעיות, ולגרום לכך שהתיזמון לא יהיה שקול לתיזמון סדרתי.
אפשר לשאול האם 2 תיזמונים שונים הם שקולים מבחינת הקונפליקטים. כלומר, אם נסתכל על כל זוג פעולות בתיזמון הראשון שנמצאים בקונפליקט (כלומר על אותו אובייקט, מבוצעות ע"י שתי טרנזקציות שונות ולפחות אחת מהן היא כתיבה), האם הזוג הזה נמצא באותו סדר בשני התיזמונים?

הגדרה: אם לכל זוג של פעולות בתיזמון הראשון, הוא נמצא באותו סדר בתיזמון השני, נגיד שהתיזמונים הם **שקולי קונפליקט**.
נניח שכל הטרנזקציות מבצעות את פעולת הcommit בסוף, ולכן לפעמים נשמיט אותה מהתיזמונים שנראה בדוגמאות.
טענה: אם 2 תיזמונים שהם שקולי קונפליקטים, יהיו להם את אותו אפקט סופי על הDB.
הגדרה: נגיד שהתיזמון S הוא **בר סידור קונפליקטים** אם הוא שקול קונפליקטים לתיזמון סדרתי כלשהו.
זה יגרור את זה שהוא בר סידור- האפקט שלו על הDB יהיה כמו הרצה סדרתית של הטרנזקציות.
כדי לבדוק במקרה הכללי אם תיזמון S הוא בר סידור קונפליקטים:

1. נייצר גרף קדימויות בשביל התיזמון, שבו יש בו קודקוד לכל אחת מהטרזנקציות.
2. מוסיפים צלע מכוונת מT_i לT_j אם יש זוג פעולות קונפליקטים בתיזמון שמערבים את T_i וT_j, וT_i הוא הראשון שמתבצע בתיזמון S.
3. S הוא בר סידור קונפליקטים אם אין מעגל מכוון בגרף הקדימויות G, והוא שקול קונפליקטים לכל סידור טופולוגי של G.

נזכיר ש**סדר טופולוגי** בגרף G הוא סידור של הקודקודים V כך שאם יש צלע מv_i לv_j, אזי v_i יופיע לפני v_j בסידור.

תרגול 11

בשמחשבים את הסגור (באלגוריתם למציאת פירוק לBCNF) כן משתמשים בכל האטריביוטים של האבא (R, ולא רק בשל R_1, בשביל להשתמש בתלויות שמשתמשות בהם).
כדי להראות שהפירוק הוא בBCNF צריך להראות שאין בו או בביסוי המינימלי שלו אף תלות שמפרה BCNF.

קריאה מלוכלכת- קריאה של נתונים שטרנזקציה אחת שינתה לפני שהיא עשתה commit (כי יכול להיות שהיא תעשה abort בסוף).
קריאה nonrepeatable- שתי קריאות שמצפים שהן יהיו זהות (שהטרנזקציה הנוכחית לא שינתה כלום ביניהן, ואיכשהו המידע השתנה).

הרצאה 12 - Protocols (Transaction Management)

בשבוע שעבר שנו במושג של תיזמון, וניסינו להבין אילו תזמונים הם טובים, כלומר הריצה שלהם זהה לריצה סדרתית מבחינת ההשפעה על הDB. הראנו שאפשר לזהות מתי תזמון הוא בר סידור קונפליקטים, ואז בפרט האפקט שלו על הDB זה כמו ריצה סדרתית של הטרנזקציות. כל זה קורה בהסתכלות אחורה על סדרה של קריאות שכבר קרו, וזה לא מה שקורה במציאות. במציאות, למערת הDB מגיעות בקשות מהטרנזקציות השונות וברגע האמת הוא צריך להחליט אם לאפשר לבצע את הפעולות שהגיעו אליו (קריאה, כתיבה, commit). הוא צריך פרוטוקול שיאפשר לו להחליט האם לאפשר פעולות שהטרנזקציות מבקשות. נרצה שהפרוטוקול יעבוד כך שגם במבט לאחור נראה שהתזמון היה בר סידור קונפליקטים.
יש פרוטוקולים שונים שאפשר להשתמש בהם, אנחנו נראה שניים כאלו. הראשון הוא פרוטקוול מבוסס נעילה.

מנעולים

לכל אובייקט בDB יש מנעול ששייך אליו. יש שני סוגים של מנעולים לכל אובייקט:

- מנעול משותף – לקריאה
  - כמה יכולים להחזיק בו במקביל
- מנעול אקסלוסיבי – לכתיבה
  - מי שיש לו מנעול אקסלוסיבי יכול גם לקרוא את האובייקט.

- טרנזקציה יכולה להחזיק במנעול אקסלוסיבי על אובייקט A אך ורק אם אין לאף טרנזקציה אחרת מנעול מסוג כלשהו על A.

אם נשתמש בפרוטוקול שמבוסס על נעילה, לפני שנבצע פעולה כלשהי על אובייקט A, נצטרך להשיג מנעול מתאים. אם הטרנזקציה לא יכולה להשיג את המנעול המתאים- נחכה.

פרוטוקול 2PL

PostgreSQL לא משתמש בפרוטוקול הזה (אבל בSQLServer, mySQL כן משתמשות במחמיר, תלוי ברמת הבידוד). PostgreSQL משתמש בפרוטוקול שמבוסס על שכפול ערכים, לא נדבר עליו השנה.

זהו פרוטוקול מבוסס מנעולים. פרוטוקול 2PL - 2 pace locking (נעילה בשני שלבים), פועל כך:

- אם T רוצה לקרוא אובייקט, הוא קודם צריך לבקש מנעול משותף על האובייקט.
  - אם הוא יודע שבהמשך ירצה גם לכתוב, הוא יכול מההתחלה לבקש מנעול אקסלוסיבי, או לבקש לשדרג את המנעול לאקסלוסיבי מאוחר יותר.
- אם T רוצה לכתוב אובייקט, הוא קודם צריך לבקש מנעול אקסלוסיבי על האובייקט.
- ברגע שטרנזקציה משחררת מנעול כלשהו, היא לא יכולה לבקש מנעולים נוספים.

כלומר לכל טרנזקציה יש את שלב ה"גדילה" בו היא מבקשת מנעולים, ואת שלב ה"הקטנה" בו היא מתחילה לשחרר מנעולים (ולא מבקשת עוד מנעולים).

- אם טרנזקציה מנסה לבקש מנעול ולא יכולה לקבל אותו, אז מנהל המנעולים עושה אחד משני דברים:
  - אומר לה לחכות- שם את הטרנזקציה הזו בתור שמחכה למנעול עד שישתחרר
  - הפלת הטרנזקציה- ונעשה roll back של מה שהטרנזקציה ביקשה ומשחררים את המנעולים. בדר"כ אחר כך מריצים את הטרנזקציה שוב באופן אוטומטי עד שבסוף היא מצליחה.

נגיד ש**תזמון ניתן להשגה ע"י 2PL** אם יכולות להיות בקשות של נעילות ושחרורים, כך שהפעולות יתבצעו בדיוק בסדר בו הן כתובות בתזמון.

<u>טענה:</u> כל תזמון שאפשר להשיג אותו ע"י 2PL הוא בר סידור קונפליקטים.

<u>הוכחה:</u> נניח בשלילה שאת התזמון השגנו ע"י 2PL אבל היה מעגל בגרף הקדימויות (לא בר סידור קונפליקטים). אזי, נסתכל על אחת הצלעות במעגל בגרף הקדימויות. הצלע הזו (מT_1 לT_2) מופיעה בגרף אם יש קונפליקט- קיים אובייקט A כך שיש קונפליקט בין T_1 לT_2, וT_1 עושה את הפעולה ראשון בקונפליקט. אם כך, אחת הפעולות לפחות היא כתיבה (כי מדובר על קונפליקט). כלומר, T_1 חייב לשחרר את המנעול על האובייקט A לפני שT_2 ישיג את המנעול הזה. כלומר T_1 משחרר איזשהו מנעול לפני שT_2 משיג אותו (לפחות פעם אחת). באופן דומה לT_1, גם T_2 צריך לשחרר איזשהו מנעול לפני שT_3 משיג אותו, וכן הלאה- T_{n-1} חייב לשחרר איזשהו מנעול לפני שT_n משיג אותו. בגלל הצלע מT_n לT_1, T_n חייב לשחרר איזשהו מנעול לפני שT_1 משיג אותו. מטרנזיטיביות נקבל שT_1 משחרר איזשהו מנעול לפני בקשה כלשהי שלו עבור מנעול אחר. לכן קיבלנו סתירה לכך שפעלנו לפי 2PL.

חדשות טובות כל תזמון שאפשר להשיג ע"י 2PL הוא בר סידור קונפליקטים, אם כל הטרנזקציות עושות commit.

חדשות רעות תזמונים שפועלים לפי 2PL יכולים לא להיות ברי התאוששות (בר התאוששות- עושה commit רק אחרי שכל התיזמונים שאת השינויים שלהם הוא קרא עושים commit). זה בעייתי כי יכול להיות שטרנזקציה תעשה commit לפני commit (או abort) של טרנזקציה אחרת שאת השינויים שלה היא קראה.

פרוטוקול 2PL מחמיר – Strict 2PL

וריאציה של 2PL, שמבטיחה שהסידורים יהיו ברי התאוששות.

נגיד ש**תזמון הוא מחמיר** אם אף ערך שנכתב ע"י T לא נקרא או נדרס ע"י טרנזקציה אחרת עד שT עושה commit או abort.
קל לראות שתזמון הוא בר התאוששות וגם נמנע בcascading aborts אם התזמון הוא מחמיר (כי כל הבעיות שנוצרו בהם הן בגלל שקראנו/כתבנו ערכים ששונו ע"י טרנזקציה אחרת לפני שהיא סיימה).
ראינו שלא 2PL מבטיח שהתוצאה תהיה תזמון מחמיר, ולכן נצטרך להוסיף לו עוד תנאי כדי שהוא יהיה מחמיר, ובפרט בר התאוששות.
הפרוטוקול החדש:

- אם T רוצה לקרוא אובייקט, הוא קודם צריך לבקש מנעול משותף על האובייקט.
- אם T רוצה לכתוב אובייקט, הוא קודם צריך לבקש מנעול אקסלוסיבי על האובייקט.
- (שינוי) טרנזקציה משחררת את המנעולים שלה רק כשהיא מסיימת לרוץ.

כל תזמון שאפשר להשיג ע"י 2PL המחמיר יהיה בר סידור קונפליקטים וגם בר התאוששות (כי מחמיר).
כלומר ברגע שיש בקשה למנעול שאי אפשר לקבל- הפעולה תכה עד שהמנעול ישתחרר (בסוף של הטרנזקציות שמשתמשות בו), או שהטרנזקציה תיפול ותרוץ שוב בסוף.

Deadlocks

במערכות שמשתמשות בנעילה, צריך להתייחס גם לקיפאון- מצב בו אי אפשר להתקדם (כל אחד מחכה למנעול ולא יכול להתקדם בפעולה שלו).
יש 2 גישות בהתמודדות עם קיפאון:

1. מניעה- אף פעם לא יוכל להיווצר קיפאון
2. גילוי- נאפשר למצבי קיפאון לקרות, אבל נדאג לגלות אותם ולטפל בהם

מניעת דדלוק

יש 2 גישות עיקריות. בשתיהן נותנים לטרנזקציה זמן התחלתי, והוא מגדיר את העדיפות שלה.
נניח שT_i מבקש מנעול, וטרנזקציה אחרת T_k מחזיקה במנעול שיש לו קונפליקט אליו.
בשיטת **wait-die** פועלים כך:

- בודקים האם $Start(T_i) < Start(T_k)$: אם כן, T_i יחכה למנעול
- אחרת, נעשה abort לT_i ונפעיל אותה מחדש עם אותו זמן התחלתי שקיבל בהתחלה
  - כך בשלב מסוים הוא יהיה הטרנזקציה הותיקה ביותר במערכת, יוכל לקבל את המנעולים שזקוק להם, ויצליח לסיים

בשיטת **wound-wait** פועלים כך:

- בודקים האם $Start(T_i) < Start(T_k)$: אם כן, נפיל את T_k ונפעיל אותו מחדש עם אותו זמן התחלתי
- אחרת, T_i יחכה למנעול

כל השיטות למניעה מבטיחות שלא יהיה דדלוק, אבל לפעמים הן מפילות טרנזקציות בלי צורך, גם אם לא היה נוצר דדלוק בהמשך.
אפשר לחשוב על זה כעל גרף של "מי מחכה למה", שתי השיטות דואגות לזה שלא יהיה מעגל בגרף. בגלל ששומרים את זמן ההתחלה, אנחנו דואגים לזה שכל הטרנזקציות יסיימו לרוץ.

זיהוי דדלוק

כאן מנסים לזהות, ואז מפילים את אחת הטרנזקציות. כאן פועלים רק אם יש דדלוק.
זה חשוב כי דדלוקים יכולים להיות נדירים, ואז לא כדאי לעשות את עבודת המניעה אלא לטפל בזה כשזה קורה. נשתמש בwaits-for-graph (גרף של מי מחכה למי), שיש בו צלע מT_i לT_j, אם T_i מחכה לT_j שישחרר מנעול. מעגל בגרף מצביע על זה שיש קיפאון, ואז הDB יבחר להפיל טרנזקציה כלשהי מהמעגל ולהתחיל אותו מחדש.
יש גישות שונות בבחירה את מי להפיל:

- טרנזקציה שיש לה הכי פחות מנעולים

- טרנזקציה שעשתה הכי פחות עבודה
- הטרנזקציה הצעירה ביותר

<u>פרוטוקול חותמת הזמן הפשוט</u>

נראה דרך אחרת לנהל את התזמונים, כך שלא נשתמש במנעולים ובעיית הדדלוק לא תופיע. יש הרבה חסרונות בלהשתמש במנעולים כדי לנהל את הטרנזקציות- עלות של שמירת מנעולים לאובייקטים שונים, בדיקת האם המנעול פנוי, ניהול ההשהייה והabort יכול להיות מאוד יקר. לכן מקובל להשתמש בשיטות בלי מנעולים. הם מבוססים על הרעיון של שימוש בtimestamp.
דיברנו על זה שתזמון 2PL שקול קונפליקטים לסידור סדרתי, המסדר אותן לפי הרגע הראשון בו הטרנזקציות עושות unlock.
בפרוטוקול חותמת הזמן הפשוט נרצה להגיע לתזמון שקול קונפליקטים לתזמון סדרתי, המסדר אותן לפי זמן ההתחלה של הטרנזקציות.
אינטואיציה: לכל טרנזקציה T ניתן חותמת זמן TS(T) כשהיא מתחילה. אם החותמת זמן של T_i קטנה מהחותמת זמן של T_k, אזי נרצה להיות שקולים לריצה סדרתית בה T_i פעל לפני T_k.
בפרט, אם שתיהן רוצות לכתוב לאובייקט A, בגלל שT_k רץ אחרי T_i, נצפה שהערך שיהיה כתוב הוא הערך שT_k כתב.
כדי לנהל את התזמון (לוודא בכל רגע נתון שהפעולות תקינות וסבירות), לכל אובייקט נשמור חותמות זמן:

- לקריאה RTS(A) – חותמת הזמן של הטרנזקציה המאוחרת ביותר שקראה את A
- לכתיבה WTS(A) – חותמת הזמן של הטרנזקציה המאוחרת ביותר שכתבה לA

אם טרנזקציה תנסה לגשת לאובייקט שחותמת הזמן שלו היא מאוחרת יותר, נעשה abort לטרנזקציה ונפעיל אותה מחדש עם חותמת זמן חדשה.

הפרוטוקול:

במקרה של קריאה:

- $TS(T_i) < WTS(A)$: abort and restart with a **newer** TS
- Otherwise:
  - Allow read
  - $RTS(A) := \max\big(RTS(A), TS(T_i)\big)$
  - Make local copy of A to allow repeatable reads

ההעתק הלוקאלי נועד לכך שאם טרנזקציה תרצה לקרוא ערך שוב, היא תקרא את הערך שנמצא אצלה ולא את זה שנמצא בDB.

במקרה של כתיבה:

- $TS(T_i) < RTS(A)$: abort and restart with a **newer** TS
- $TS(T_i) < WTS(A)$: abort and restart with a **newer** TS
- Otherwise:
  - Allow write
  - $WTS(A) := TS(T_i)$
  - Make local copy of A to allow repeatable reads

יש אפשרות לשנות מעט את הפרוטוקול בעזרת **כלל הכתיבה של תומאס**. הוא מעט לא אינטואיטיבי, ועושה שינוי במקרה בו $TS(T_i) < WTS(A)$- את הכתיבה במקרה כזה נבצע על העתק לוקאלי, כי בכל מקרה אם היינו כותבים לDB הערך היה נדרס ע"י הטרנזקציה המאוחרת יותר.

- $TS(T_i) < WTS(A)$: perform write on local copy and continue

פרוטוקול חותמות הזמן הפשוט בלי כלל הכתיבה של תומאס הוא שקול קונפליקטים לתזמון סדרתי לפי חותמות הזמן. עם חוק הכתיבה של תומאס, זה לא מתקיים- אבל היא עדיין שקולה מבחינת האפקט על הDB לריצה סדרתית (אבל לא שקול קונפליקטים אליו).
אם ננסה להריץ את פרוטוקול חותמות הזמן הפשוט התזמון לא יהיה בר התאוששות. לכן פרוטוקול חותמות הזמן האמיתי מסובך יותר- צריך לזכור על איזה ערכים בוצעה commit ועל איזה לא.

אינטואיציה לMVCC – multi version concurrency control

יש בו הרבה העתקים של אותם פריטים, וטרנזקציות שונות רואות העתקים שונים של אותם נתונים.
לכל טרנזקציה יש תמונת מצב של הDB, שמודרת לפי הזמן בו היא התחילה לרוץ. כל הקריאות והכתיבות נעשות לעותק המקומי של אותה טרנזקציה. כשרוצים לעשות commit מוודאים שזה תקין (לא מקלקל כלום מבחינת הנתונים), ואז מבצעים את פעולת הcommit או rollback.
היתרונות העיקריים הם שאף פעם קוראים לא יחכו לכותבים ולהפך, כי לכל טרנזקציה יש עותק משלה. יכולים להפסיד בסוף, אבל לפחות בזמן ביצוע הזמנים טובים.
לכל טרנזקציה יש מזהה ייחודי.
טבלאות בDB מכילות עמודות נסתרות, לדוגמא xmin שמכילה את המזהה של הטרנזקציה שהוסיפה את השורה, וxmax שמכילה את המזהה של הטרנזקציה שמחקה/ניסתה למחוק את השורה. 0 מציין שהשורה לא נמחקה.
אז יכול להיות מצב בו קיימות שתי שורות עם אותו מפתח, אך אף טרנזקציה לא תראה סתירה, כי יהיה לה עותק רק עם המידע שנכון אליה. אף פעם לא מוחקים או משנים שורה בטבלה- רק משנים את xmax או מייצרים העתק (כותבים בשורה המקורית שמחקנו אותה ויוצרים שורה חדשה עם הערכים המעודכנים). אז הטבלה יכולה להשאיר שורות "ישנות", שמבחינת חלק מהטרנזקציות נמחקו, ומבחינת חלק אחר הן עדכניות.
יש פקודה שנקראת VACUUM (והיא גם רצה אוטומטית מדי פעם) שמוחקת את השורות שברור שלא ישתמשו בהן בהמשך- כלומר זה בעבר עבור כל הטרנזקציות שרצות.
אם נעשה SELECT * xmin, xmax FROM a; נראה גם את העמודות האלו. עוד עמודה נסתרת היא ctid המיקום הפיזי של השורה- מספר הבלוק ומיקום השורה בבלוק.
פרטים חשובים שלא נדבר עליהם:

- איך ממשים את בקרת המקביליות מעל נתונים משובפלים
- איך מחליטים מתי אפשר למחוק שורות בפעול
- איך נותנים ID לטרנזקציות
- איך מתמודדים עם רמות שונות של בידוד

תרגול 12

קונפליקט (פוטנציאל לבעיות)- שתי פעולות שנעשות על אותו אובייקט ע"י טרנזקציות שונות, באשר לפחות אחת הפעולות היא כתיבה.
קונפליקטים אפשריים: $R_i(A)W_j(A)$, $W_i(A)R_j(A)$, $W_i(A)W_j(A)$.
שני תזמונים שונים יכולים להיות שקולים מבחינת הקונפליקטים בהם, אם כל הקונפליקטים שלהם מופיעים באותו הסדר (אם החיצים לא מתהפכים).
נרצה שהתזמונים יהיו שקולים מבחינת הקונפליקטים לתזמון סדרתי, כדי לקבל הבטחה על סדר הפעולות. נבנה גרף קונפליקטים, בו הקודקודים הם הטרנזקציות והחיצים הם הקונפליקטים. אם בגרף אין מעגלים, אז בוודאות הגרף הוא conflict serializable. נמצא את זה בעזרת מיון טופולוגי (סידור של הקודקודים בצורה כך שכל החיצים פונים רק בכיוון אחד- מימין לשמאל), והוא שקול לסדר מימין לשמאל.

כל התזמונים כאן הם לאחר מעשה, וכעת נרצה פרוטוקול כך שהתזמונים יצאו conflict serializable. בשביל זה יש את פרוטוקול 2PL המשתמש במנעולים- משותף, הנועל לקריאה בלבד (יכול להיות אצל כמה בו זמנית), ואקסקלוסיבי, הנועל לכתיבה (יכול להיות רק אצל אחד כשאין לאחרים מנעול משותף על הנתונים). אפשר לבקש כמה מנעולים שרוצים, וכשמשחררים מנעול אי אפשר לבקש עוד מנעול אחר כך.
נשים לב שתזמון כזה לא מבטיח לנו recoverable (שטרנזקציה עושה commit רק אחרי שטרנזקציות שהיא קראה את השינויים שלהן עושות commit או abort) ו avoiding cascading aborts (לא עושים קריאות מלוכלכות). תזמון strict 2PL כן מבטיח את זה- לא משחררים מנעולים עד שלא מבצעים את כל הפעולות.
בעקבות ההכנסה של מנעולים, גרמנו לזה שאולי יהיה deadlock. איך נזהה שיש דדלוק? כל הזמן עוקבים ברקע אחרי wait-for graph. אם יש מעגל בגרף, יש דדלוק. אם מזהים דדלוק, מנהל התזמונים בוחר להרוג את אחת הטרנזקציות. איך הוא בוחר? הכי צעיר, הכי פחות מנעולים, עשה הכי קצת.
זו גישה של זיהוי דדלוק. גישה אחת היא מניעה- נדבר עליה בשבוע הבא.

הרצאה 13- התאוששות

נפילות

דיברנו על 4 תכונות שמסד הנתונים מספק לטרנזקציות, הנה 2 מהן:

- אטומיות- הפעולות מקובצות יחד, ואם כולן לא יכולות להסתיים בהצלחה, כולן צריכות להימחק
- עמידות- אם הטרנזקציה ביצעה commit, אז כל מה שהיא כתבה אכן יישאר בDB

בעיית ההתאוששות צריכת להתמודד עם בעיה של עמידות, וגם איך עושים rollback כשיש בעיה בטרנזקציה. צריך להבטיח שאכן התוצאות שנעשו עליהן commit יישארו בתוך המסד, גם אם היו קריסות שלו.
התאוששות מבעיות מורכבת מ2 שלבים:

- לפני ההתאוששות צריך לעשות פעולות שיעזרו להתמודד עם בעיות כשהן יגיעו
- הבנה איך אפשר לשחזר את המידע אחרי הנפילה

זה ממש קריטי בכל מערכת. איבוד של מידע הוא נוראי במקרה של מסד נתונים.
זה דיון נפרד משיטת ניהול הטרנזקציות- איך מבצעים את הrollback ולא מתי.

עם איזה סוג נפילות צריך להתמודד?

יש 3 סוגים שונים של בעיות:

- נפילות ברמת הטרנזקציה
  - טרנזקציה יכולה להחליט לעשות abort בגלל בעיה לוגית
  - מערכת הDB יכולה להחליט באופן יזום לעשות abort לטרנזקציה, כמו שראינו במקרה של דדלוק
- נפילות ברמת המערכת
  - באג שגורם למערכת ליפול, המחשב שבו הDB נמצא נופל (בעיית חשמל). קורה בזמן בלתי צפוי, ולכן צריך כל הזמן להיערך לזה
- נפילות ברמת הדיסק
  - דיסק מקולקל ולא עובד. אי אפשר להתאושש. לכן הרבה פעמים יהיו הרבה העתקים לDB בדיסקים שונים

ניהול הbuffer pool

כדי להבין איך מערכת הDB מתאוששת מנפילות, נצטרך לדבר על ניהול הזיכרון המרכזי.
האם במקרה של commit דוחפים מיד ערכים לדיסק?
האם מאפשרים לשינויים של טרנזקציות אחרות להיכנס לדיסק עוד לפני commit?

והאם המידע של כל הבלוק נכנס לדיסק או לא?

מנהל הזיכרון המרכזי צריך להחליט על מדיניות:

- גניבה- הDB יכול לכתוב ערכים לדיסק גם אם עוד לא נעשה עליהם commit
- ללא גניבה- רק ערכים שנעשה עליה commit יכולים להכנס לדיסק
  יותר פשוט כשטרנקציה עושה rollback
  אבל לא תמיד יש מספיק זיכרון בשביל זה

וגם:

- הכרח- ברגע שעושים commit, מכריחים את הערכים להיכתב לדיסק
  מקל על התאוששות, כי השינויים כבר נמצאים על הדיסק
  אבל זה מאוד איטי, גם אם נשאר מקום בזיכרון המרכזי
- ללא הכרח- יכול להיות שערך שעשינו עליו לא יכנס מיד לדיסק

מדיניות של ללא גניבה + הכרח היא:

- קלה למימוש- לא מורידים מידע לדיסק אלא אם עשינו עליו commit
- אף פעם לא נצטרך לעשות undo לטרנזקציה שעשתה abort
- החישוב יכול להיות איטי
- בכל מקרה זה לא ישים בשביל מערכות בהן צריך לעשות כתיבות גדולות

לכן בפועל מערכת הDB משתמשת במדיניות של גניבה + ללא הכרח. צריך לעבוד קשה בשביל עמידות, וגם שתהיה יכולת לעשות rollback.

Write Ahead Logging (WAL)

שיטה שמיועדת לאפשר התאוששות.

הניהול של הבאפר נעשה בפוליסה של גניבה + ללא הכרח. כדי לאפשר את המדיניות הזו מייצרים Log של כל השינויים בDB, ואותו כותבים לדיסק. חייבים לכתוב ללוג את השינויים ולכתוב אותם לדיסק, עוד לפני שהשינוי נעשה בפועל (לפני שנשנה ערך נכתוב את הלוג שלו לדיסק). כשרוצים לעשות commit של טרנזקציה, קודם נכתוב את הערכים שבלוג לדיסק. רק אחרי כתיבה של הלוג לדיסק מאשרים שהcommit התבצע.

כך יש על הדיסק מידע איך השינויים אמורים להתבצע. יודעים לפי מה שיש בלוג מה אמור להיות הערך הסופי (גם אם הוא לא מעודכן כרגע), וזה מה שיעזור להתאושש במקרה של נפילות.

מתי כותבים את הלוג לדיסק?

1. כשעושים commit (כותבים את זה שהיא עשתה לוג וכותבים את הלוג לדיסק)
2. לפני שכותבים בלוק לדיסק (בגלל זה אפשר לגנוב)
3. כשהמקום שהוקצה ללוג בזיכרון המרכזי התמלא
4. מדי פעם, כשהDB נח ויש זמן טוב לזה

איך הדברים באמת נעשים?

לכל כניסה ללוג יש log sequence number (LSN), ושומרים את המספר האחרון שעד אליו הכנסנו לדיסק flushedLSN. כשטרנזקציה מבקשת לעשות commit, צריך לכתוב לדיסק את כל השורות עד לשורה שבה הטרנזקציה עשתה commit (למעשה את כל השורות בהן הטרנזקציה פעלה). שומרים לכל טרנזקציה את הLSN האחרון שבו עשה משהו (lastLSN של הטרנזקציה), וכך אפשר לדעת האם החלקים הרלוונטים בלוג כבר עברו לדיסק- אם ה$lastLSN > flushedLSN$ אז כותבים את הלוג בדיסק לפחות עד הlastLSN, ומעדכנים את הflushedLSN בהתאם.

באופן דומה, כשרוצים לכתוב דף מלוכלך לדיסק (השתנה בזיכרון המרכזי מאז שקראנו אותו מהדיסק), לפני שכותבים אותו לדיסק (בגלל המדיניות של גניבה) צריך לכתוב לדיסק את הלוג עד למקום האחרון בו עשינו שינוי לדף הזה. כדי לעשות את זה בצורה טובה, שומרים לכל דף מלוכלך

את הpageLSN. אם רוצים לגנוב את הדף הi, בודקים אם $pageLSN_i > flushedLSN$ ואם כן עושים flush לlog לפחות עד ה$pageLSN_i$ ומעדכנים את הflushedLSN.
בנוסף, לכל דף מלוכלך שומרים את השורה הראשונה בלוג שגרמה לו להיות מלוכלך recLSN.

<u>אלגוריתם ARIES להתאוששות</u>

משתמשים במדיניות של גניבה + ללא הכרח. משחזרים את ההיסטוריה בזמן של redo- כלומר אחרי התאוששות, משחזרים כדי להיות בדיוק באותו מצב שהיינו לפני הנפילה. כשצריכים לעשות undo לטרנזקציות של עשו commit, ובזמן הזה כותבים את השינויים האלו ללוג- כי המערכת יכולה ליפול שוב גם בזמן השיחזור.
תוך כדי ביצוע רגיל של הDB נשמור טבלה של הטרנזקציות הפעילות active transaction table **ATT**. לכל טרנזקציה פעילה נשמור ID, סטטוס (בפעולה, ביקש commit, בתהליכי abort) וlastLSN. אחרי שהטרנזקציות הסתיימו לגמרי (כולל פעולות הניקוי ושחרור המנעולים של הcommit או abort), הן יצאו מטבלת הATT.
טבלה נוספת שנשמרת היא טבלה עם הדפים המלוכלכים **DPT** .dirty page table, שמכילה מידע על כל הדפים המלוכלכים- שנקראו לזיכרון המרכזי והשתנו בו. לכל דף כזה שומרים ID, recLSN (השורה הראשונה בלוג בה שינינו את הדף), pageLSN (השורה האחרונה בלוג בה שינינו את הדף). דף שרק קוראים ולא כותבים הוא לא מלוכלך ולכן לא ייכתב ללוג.
בין הכתיבה של הCOMMIT לכתיבה של הTX-END ללוג, יש פעולות של שחרור משאבים וניקיון.
אם החלטנו לכתוב דף מלוכלך לדיסק, נוציא אותו מטבלת הDPT, ונכתוב את הלוג עד השורה של הpageLSN לדיסק.

מה קורה בcommit?

- כותבים COMMIT ללוג
- נעשה flush לזיכרון המרכזי של כל הלוג עד הcommit
- מודיעים שקרה commit
- נכתוב TXNEND ללוג (לא צריך לכתוב לדיסק)

אם טרנזקציה רוצה לעשות abort, נצטרך למחוק את הפעולות שלה. כדי לעשות את זה יעיל, צריך לדעת אילו שינויים צריך לעשות. לכל טרנזקציה שמרנו את הlstLSN – אבל זה לא עוזר למחוק את כל השינויים. לכן לכל שורה בלוג שומרים את מספר השורה הקודמת של אותה הטרנזקציה. זו מעין רשימה משורשרת בלוג. כל השורות עם אורך קבוע ולכן נח לעבור ביניהן.

כשטרנזקציה מבקשת לעשות abort:

- מודיעים שרוצה לעשות abort
- מוסיפים את המידע על הabort ללוג בזיכרון
- עוברים על כל השורות ומוחקים את הפעולות- כותבים ללוג עצמו ב compensation log record CLR, שבה כתוב השינוי והשורה הבאה שיש לעשות לה abort
- מודיעים שסיימנו

עד עכשיו למדנו איך להתאושש מנפילה של טרנזקציה בודדת. איך מתאוששים מנפיהל של כל המערכת? בנוי מ3 שלבים:

1. **אנליזה**- שחזור טבלת ATT וטבלת DPT.
2. **Redo**- חוזרים על כל ההיסטוריה, מהrecLSN הקטן ביותר בטבלת הDPT- כלומר מתקנים את כל מה שצריך כדי להביא את המצב בDB להיות כמו שהיה ברגע הנפילה.
3. **Undo**- לכל טרנזקציה שלא עשתה commit. בוחרים באופן סדרתי את הLSN הגדול ביותר של טרנזקציה כזו ועושים לה undo.

כשרוצים לשחזר את המערכת, צריך להתחיל מתחילת הלוג. אבל לא רוצים להתחיל מהלוג של תחילת כל הזמנים, אלא להתחיל ממצב מאוחר יותר בלוג, שבו אנחנו כבר בטוחים של השינויים

כבר נכתבו לדיסק. בשביל זה צריך נקודת checkpoint, שבה אנחנו יודעים שכל המידע עד לרגע זה נכתב לדיסק. כדי לכתוב checkpoint צריך:

1. לא לאפשר לטרנזקציות חדשות להתחיל, ולחכות עד שכל הטרנזקציות הנוכחיות יסיימו
2. נכתוב לדיסק את כל הלוג
3. נכתוב את כל הדפים המלוכלכים לדיסק
4. נכתוב <CHECKPOINT> ללוג, ונעשה לזה flush לדיסק

כעת את השחזור של הDB ניתן לעשות החל מהנקודה הזו.

בפועל, זה מאוד לא יעיל לעצור טרנזקציות חדשות ולא לאפשר להן לעבוד כשעושים checkpoint. לכן התהליך מורכב יותר וכן מאפשר להן לרוץ.

שלב האנליזה

מתחילים מנקודת הcheckpoint האחרונה, או מתחילת הלוג

1. מוצאים את נקודת הזמן ממנה צריך להתחיל לעשות redo.
2. רוצים לשחזר את הDPT- לא נוכל לשחזר בצורה מדויקת, אבל נמצא קבוצה של דפים שמכילה את כל הדפים שהיו מלוכלכים ברגע הנפילה. כלומר בסופו של דבר, יהיו לנו בה גם דפים לא מלוכלכים (כאלו שנכתבו לדיסק אחרי השינוי), אבל אנחנו לא יודעים אם הם מלוכלכים או לא.
   a. יכול להיות שכתבנו את אחד הדפים לדיסק כבר. מכיל דפים מלוכלכים שאולי נכתבו לדיסק.
   b. יכול להיות שהיו דפים נוספים, אבל נוכל לדעת בוודאות שהם לא נכתבו לדיסק, כי אז המידע על זה היה נכתב ללוג בדיסק. לא מכיל דפים שבוודאי לא נכתבו לדיסק.
3. נשחזר את הATT- גם כאן יכול להיות שמה שנשחזר יהיה שונה מהטבלה המקורית, זה יכיל את כל מי שהיו פעילות ואולי גם כאלו שסיימו. זה יספיק בשביל להבין למי צריך לעשות undo.
   a. יכול להיות שטרנזקציה שעשתה קומיט סיימה אותו ולנו זה לא מופיע כבה
   b. יכול להיות שהייתה טרנזקציה נוספת, שבהכרח לא שינתה שום דבר בדיסק

הלוג שעוברים עליו הוא זה שנמצא בדיסק. יכול להיות שבזיכרון המרכזי היה עוד מידע, אבל לא נוכל לגשת אליו.

בשלב זה עוברים על הלוג ורק משנים את טבלת הATT והDPT, בלי לשנות את הבלוקים עצמם.

כעת נצטרך לגזור מתוך הDPT והATT את המידע המדויק בשביל ההתאוששות:

1. הנקודה הראשונה ממנה צריך לעשות redo. זה לפי הערך הקטן ביותר בrecLSN בDPT
2. למצוא את קבוצת הדפים שעשויים להיות מלוכלכים בבאפר
3. למצוא את הטרזנקציות שלא סיימו ולעשות להן undo

שלב הRedo

- מתחילים מהrecLSN הקטן ביותר בDPT
- מסתכלים עד סוף הלוג
- לכל שורה בה משנים ערך- עושים את העדכון:
  - מביאים את הדף הרלוונטי לזיכרון המרכזי
  - בודקים אם ה$pageLSN < current\ LSN$:
  - אם כן, עושים את העדכון

נשים לב שכותבים לזיכרון המרכזי ולא דוחפים ישר לדיסק, כי רוצים לשחזר את המצב שהיה.

צריך לבדוק מול הpageLSN כי יכול להיות שהכתיבה לדיסק היא לא של ערכים פשוטים אלא של שאילתות SQL, ולכן זה יכול להיות מסובך לבדוק. בנוסף, יכול להיות שנכתב ערך מאוחר יותר.

שלב הUndo

- עושים לכל הטרנזקציות שלא עשו קומיט לפי הלוג בדיסק (ולכן לא עו קומיט עד הנפילה). אלו נקראות <u>הטרנזקציות המפסידות</u>

- עושים באותו אופן בו עושים rollback לטרנזקציה בודדת- רק שבכל פעם עושים undo לפעולה האחרונה בלוג שזקוקה לזה

למעשה, התהליך הוא יותר מורכב- בגלל תהליך הcheckpoint, וכי השורות בלוג יכולות להיות פעולות לוגיות.

תרגול 13

נרצה להימנע מראש מדדלוקים.
בWait-Die טרנזקציה הורגת את עצמה.
בWound-Wait הותיקים הורגים את הצעירים.
העדיפות הנמוכה ביותר היא הטובה ביותר. לכן נשארים עם אותה העדיפות.
בפרוטוקול חותמות הזמן לעומת זאת מתחילים עם TS חדש, כדי שנוכל לקרוא שינויים מאוחרים יותר.

הרצאה 14- NoSQL

עד כה עסקנו במסדי נתונים רלציוניים- טבלאיים.
כעת נדבר על מסדי נתונים בגישה הNoSQL. אפשר להבין מהו מסד כזה, מה היתרונות והחסרונות שלו לעומת מסד רלציוני, רק בשלב זה של הקורס.
NoSQL זו משפחה של טכנולוגיות מסדי נתונים, עם סוגים שונים של מודלי נתונים (דרכים שונות לחשוב על הdata), שפות שונות והבטחות שונות.

מהם הפיצ'רים העיקריים של מסד נתונים רלציוני?

- מבוססים על אבסטרקציה של טבלאות
- משתדלים שהטבלאות יהיו מנורמלות- שתהיה כמה שפחות יתירות וחזרתיות בתוך המסד
- סטנדרטי (החלפה בין מערכות היא לא מסובכת)
- שאילתות נכתבות בSQL, ואחר הפיצ'רים החשובים ביותר זה פעולת הצירוף
- עקביות (אפשרות לתת לטרנזקציות לרוץ במקביל ולהישאר עקבי)

איך מסדי NoSQL ביחס לפיצ'רים האלו?

- לרוב לא מדובר באבסטרקציה טבלאית
- לא מנורמלים- להפך, יש רמה גבוהה יש יתירות של מיעד
- אין סטנדרטיזציה- כל מסד בנוי בצורה שונה ועם שפה שונה (לכן אם רוצים להחליף מסד, כנראה נצטרך לכתוב את כל הקוד מחדש)
- בדרך כלל אין תמיכה בצירוף של נתונים שונים (אם אין להן תמיכה בשפה של המסד, נצטרך לכתוב תוכנית חיצונית בשפה עילית אחרת שניגשת למסד, קוראת נתונים ומבצעת את הצירוף)
- אין הבטחה על עקביות- מסד נתונים כזה הוא מאוד מוגבל. כשכותבים למסד כזה, צריך להבין היטב אילו סוג טרנזקציות נתמכות ואיזה הבטחות נותנים לנו. לכן צריך לחשוב על איך טרנזקציות שונות ישפיעו אחת על השנייה.

מה החסרונות של מסד נתונים רלציוני?

- impedance mismatch- יש שינוי מחשבתי בין מודל הנתונים הרלציוני לבין אבסטרקציה של הנתונים בצורה בה אנחנו חושבים עליה (לדוגמא שמירת מסמכים- יש להם אופי שונה מלטבלה ולכן זה לא יהיה מתאים כנראה).
- scaling out- בנויים על סמך היכולות של המחשב עליו הם נמצאים. אם אנחנו במקסימום של הscale up- משתמשים במחשב הכי טוב שאפשר לצורך זה, נרצה לעשות scale out- לפזר את המידע על הרבה מחשבים שונים ולהתייחס לזה כמסד אחד, ופתרונות של ביזור מאוד יקרים ומסובכים.
זה קשור ליתרון לעקביות- המסד נמצא רק במקום אחד ולכן קל לשמור בו על עקביות.

מודל נתונים

כל מסד נתונים שהוא לא SQL הוא שונה, ולכן המטרה שלנו היא ללמוד איך להיות צרכנים נבונים של מסדים כאלו.

צריך לחשוב על מהו מודל הנתונים של המסד שנרצה להשתמש בו.

מסד הנתונים הרלציוני מבוסס על המודל הרלציוני- המודל הטבלאי. זה מודל שטוח. יש בו 2-3 רמות של היררכיה- טבלה, שורה, ערך.

סוגים נפוצים של מודלי נתונים: לא צריך לדעת סינטקס למבחן

1. מודל key value – hash table מבוזר (מה שמאפשר לשמור על הרבה מחשבים שונים). יש מפתח שלו יש ערך שמתאים לו, שיכול להיות מכל סוג.
   a. פופולארי ביותר- redis – אפשר להשתמש בפקודות של SET, GET, INCR (פעולה אטומית של הגדלה), DEL.
   b. השאר (של אמזון ומייקרוסופט) הם יכולים לתמוך גם במודלים נוספים
2. מודל document store – כמו הקודם, רק שהערכים שמאוחסנים הם קבצים, לרוב קבצי JSON.
   a. שימוש בmongoDB – ממש שונה מDB רלציוני, דומה יותר לקוד
   b. וגם בשל אמזון ומייקרוסופט
3. מודל column store – דומה למודל טבלאי, אבל מתאים לטבלאות שיש להם הרבה ערכי Null. המידע כאילו נמצא בhash table שנותנים לו שורה ועמודה ומחזיר ערך.
   a. שימוש בcassandra – סינטקס דומה לSQL, אבל בCQL:
      i. אין צירוף
      ii. אין טרנזקציות
      iii. את הWHERE אפשר לעשות רק על עמודות עליהן יש אינדקס
      iv. יש רק AND (בלי OR/NOT)
      v. כל העדכונים מבוססים על המפתח הראשי
4. מודל graph databases – המודל הוא אוסף של קדקודים וצלעות, יכול להיות שיש תוויות, אטריביוטים ומשקולות עליהן.
   a. הפופולארי בכל הנראה הוא neo4j – נראה דומה לSQL במובן הדקלרטיבי

רוב מסדי הנתונים NoSQL הם schema-less. במסד נתונים רלציוני נצהיר על הטבלאות והעמודות בהן. היתרון הוא שאפשר להוסיף כל סוג של מידע שנרצה. המחיר הוא שלא נדע בפועל אילו תכונות יש בהכרח לכל פריט מידע במסד. יש מערכות שלמות שהתפקיד שלהן זה לעקוב אחרי סוג האובייקטים והתכונות שמתקיימות.