**Transaction Isolation Level**

טרנזקציה היא רצף של פעולות שבוצעו כיחידה לוגית אחת. יחידת לוגית חייבת להציג ארבעה מאפיינים ACID, הנקראים תכונות האטום, עקביות, בידוד ועמידות, כדי להיות זכאים כטרנזקציה.

* *ACID - Atomicity, Consistency, Isolation, Durability -* *תכונות אשר בלעדיהן לא ניתן להבטיח את עקביות הנתונים במערכות.*

כאשר מבצעים טראנסאקציה, נועלים את הרשומות, שמשתתפות בטראנסאקציה, עד לסיום הטראנסאקציות. כשטרנזקציה שמתעסקת עם המידע לוגית , את המידע הזה כנעול על ידה וטרנזקציה אחרת שמגיעה להתעסק עם אותו המידע מכבדת את הנעילות שלקחה הטרנזקציה הקודמת. בצורה כזאת, אנחנו משיגים אפקט של סנכרון שמבוסס על זה שבמקרה של “התנגשות רצונות” בין טרנזקציות, אחת ממתינה לשנייה שתסיים, תשחרר את הנעילות שלה – ואז ממשיכה.

ישנם מספר סוגים של נעילות, אבל לצורך ההסבר נדבר רק על 2 סוגים:

Shared lock - כל ביצוע קריאה של נתונים ע"י משפט select שם נעילה מסוג shared lock על אותם רשומות.  
אין הגבלה על כמות הshared lock על רשומה, כך שמספר משתמשים יכולים לקרוא את אותה רשומה בו זמנית.

Exclusive lock - כל שינוי של הנתונים (update, insert, delete) חייבים לקבל exclusive lock על הרשומות, שמשנים. ברגע שיש על רשומה exclusive lock אחד, לא ניתן לשים על הרשומה שם נעילה נוספת (כולל shared lock), ורק הprocess, שנעל את הרשומה, יכול לעבוד עם הרשומה. הנעילה משתחררת בסיום הטראנסאקציה.

**Isolation level** קובע את הגישה לנתונים, שנמצאים בתוך טראנסאקציה. מדובר למעשה בהגדרה פשוטה: ההבטחה שגם אם שתי טרנזקציות רצות במקביל, תוצאת הביצוע שלהן בפועל תהיה כאילו הן רצו אחת אחרי השנייה. ב- SQL Server קיימים מספר Isolation Levels שאפשר לקבוע לטרנזקציה.

הקביעה מתבצעת באמצעות הוספת השורה בתחילת הטרנזקציה, כאשר במקום xxx מופיע שם ה- Isolation Level:

SET TRANSACTION ISOLATION LEVEL xxx

בכל Session ניתן לשנות את ה- Isolation Level, אלא אם כן רוצים להשאיר את הגדרות ברירת המחדל ללא שינוי (Read Committed). ההגדרה של ה - Isolation Level משפיעה על שלוש הבעיות הבאות:

Dirty Reads - המונח הזה מדבר על קריאות של מידע ששונה ע”י טרנזקציה כלשהי שטרם הסתיימה. כלומר, אנחנו קוראים מידע שלא מובטח שאי פעם היה נכון. לא מדובר על מידע שאולי לא יהיה נכון עוד שנייה (ואז אנחנו יכולים להגיד שהוא נכון לרגע הקריאה), אלא מידע שייתכן שלעולם לא היה אמור להיקרא כי הוא שונה ע”י טרנזקציה שעדיין בתהליך ואז ייתכן שיש לה עוד שינויים שהיא רוצה לעשות על המידע, או שהיא סיימה לעשות את השינויים עם המידע שלנו, אבל תיכשל בהמשך הפעולה שלה, תעשה rollback ואז המידע שלנו מעולם לא היה נכון.

Phantom Reads - דמיינו שיש לנו טרנזקציה שמריצה את אותה שליפה בהתחלה ובסוף, כאשר השליפות הן על טווח של שורות. למשל, אנחנו מביאים את כל העסקאות שבוצעו מישראל בתחילת הטרנזקציה, ואז בסוף הטרנזקציה שוב עושים את אותה השליפה. נניח גם שבמקביל רצה עוד טרנזקציה שהוסיפה עוד עסקה מישראל. כלומר, השורות שראינו בטרנזקציה A בתחילת הטרנזקציה ובסוף הטרנזקציה הן לא אותן שורות, כי התווספה שורה באמצע. האם מצב זה תקין מבחינתנו? אולי כן ואולי לא, ובכל אופן יש לזה השפעה על רמת הבידוד הנדרשת בין טרנזקציות.

Repetable Reads - בדומה ל- Phantom Reads, אבל עכשיו אנחנו לא מסתכלים על טווח שורות אלא על שורה ספיציפית. למשל, קראתי את הנתונים על לקוח מסויים בתחילת הטרנזקציה, ולאחר מכן בסוף הטרנזקציה אני שוב קורא את הנתונים של אותו הלקוח. מה ההתנהגות שאני רוצה שתהיה אם במקביל רצה טרנזקציה אחרת, שמעדכנת את הנתונים של אותו הלקוח? אני רוצה לראות את אותם הנתונים שראיתי בהתחלה? את הנתונים העדכניים אבל ראיתי בתוך אותה טרזנקציה שתי גרסאות שונות של אותה השורה.   
אפשר להתייחס לכל אחד מהמקרים האלה כאל מקרים בעיתיים שיש למנוע. שימוש נכון בטרנזקציות - מונע.

SQL Server יכול לעבוד בארבע Isolation levels שונים. ארבעת הרמות הם:

SERIALIZABLE

נניח שהיינו רוצים לפתור איזושהי בעיית סנכרון בין שני threads בקוד שאנחנו כותבים. אחת הדרכים הבסיסית ביותר הייתה לשים את כל הבלוק של הקוד שעלול להתנגש תחת lock, כאשר הוא כתוב ברצף. ואז מובטח לנו שעבור רצף הפעולות שלנו אנחנו מחזיקים את הנעילה, והיא לא הולכת לשום מקום עד שאנחנו מסיימים. את התרחיש הזה אפשר להקביל ל Serializable Isolation Level.

* *Thread הוא כמו תת תהליך של תהליך (process) כלשהו במערכת. משתמשים בו כדי להריץ כמה חלקי קוד בתוכנה אחת במקביל או את אותו הקוד מספר פעמים. process מכיל אחד או יותר תהליכונים, כולם חולקים את אותו מרחב הזכרון הוירטואלי של התהליך, חולקים משאבים משותפים.*

הקונספט פשוט: אנחנו רוצים להבטיח שאנחנו רואים רק את המידע הכי עדכני, ושהוא נשאר אותו הדבר לכל אורך הריצה של הטרנזקציה שלנו – אלא אם כן אנחנו שינינו אותו. בנוסף, אנחנו לא מאפשרים לטרנזקציות אחרות לראות data

איך אנחנו משיגים את זה? מפזרים lock על כל שורה שקראנו (ובוודאי שעל כל שורה שעדכנו). לא רק זה, אלא מפזרים lock על כל הטווח שבין שתי שורות שקראנו. כלומר, אם קראנו שורה שה- ID שלה הוא 10 ושורה שה- ID שלה הוא 100, נשים lock על כל הטווח שבאמצע.

בצורה הזאת אנחנו הולכים למעשה בגישה הכי מחמירה. אנחנו כמובן משלמים על זה במקביליות, שאותה אנחנו מורידים ל-0. לא תהיה מקביליות על אותו המשאב בין טרנזקציות שמוגדרות כ- Serializable.

REPEATABLE READ

הקונספט של REPETABLE READ הוא להיות כמו SERIALIZABLE, רק קצת פחות מחמיר. איך זה בא לידי ביטוי שזה “פחות מחמיר”? אמנם נועלים על כל שורה שקוראים, בצורה כזאת שאחרים לא יכולים לקרוא (ובטח שלא לעדכן) אותה. אבל, לא נועלים על הטווח בין שתי שורות שקוראים. כלומר, אם קוראים שורה שה- ID שלה הוא 5 ושורה שה- ID שלה הוא 100, תהיה נעילה על שתי השורות הללו שתמנע עדכון וקריאה שלהן מטרנזקציות אחרות, אבל על השורות 6-99 לא תהיה נעילה – בניגוד ל- SERIALIZABLE.

כלומר, מחמירים פחות: אנחנו אמנם מבטיחים שאם אנחנו קוראים את אותה השורה פעמיים אנחנו נראה את אותו המידע, אבל לא מבטיחים שלא התווסף מידע בין קריאות שונות.

READ COMMITTED

בעוד שגם ב- SERIALIZABLE וגם ב- REPEATABLE READ מובטח לנו שאם אנחנו קוראים את אותה השורה אנחנו נקבל את אותו המידע, פה מובטח לנו משהו חלש משמעותית – שנקבל מידע שהוא committed. כלומר, אם קראנו את אותה שורה בתחלת ובסוף הטרנזקציה, יכול להיות שנקבל ערכים שונים אם טרנזקציה אחרת שרצה בתור READ COMMITTED עדכנה את המידע בין לבין. עם זאת, מובטח שבכל אופן לא נקבל Dirty Read.

כלומר, יכולה להיות טרנזקציה A שקוראת שורה מסויימת, ואז טרנזקציה B מתחילה ומעדכנת את השורה הזאת. נניח שבאותו רגע טרנזקציה A מנסה לקרוא שוב את השורה. היא תצטרך להמתין עד שטרנזקציה B תעשה COMMIT אבל ברגע שטרנזקציה B תעשה COMMIT, היא תקרא שוב את השורה ותקבל את הערך העדכני.

READ COMMITTED הוא ה- isolation level הדיפולטי ב- SQL Server, כי הוא מצד אחד מבטיח נכונות של המידע במקרה של הרצות מקביליות, ומצד שני מאפשר רמה גבוהה יותר של מקביליות מ- REPEATABLE READ ובטח מ-SERIALIZABLE.

READ UNCOMMITEED

מדובר ב- Isolation Level המינימליסטי ביותר, שלא מבטיח כלום. הוא יכול לשמש רק לקריאות (לא ניתן לבצע פעולות CRUD) ומאפשר לקרוא מידע שטרנזקציה פעילה שינתה, אבל עדיין לא עשתה commit (כלומר, ייתכן שהוא מעולם לא היה נכון ולעולם לא יהיה נכון). במילים אחרות – מאפשר Dirty Reads. מדובר ב- Isolation Level שלא מתאים אם רוצים לקבל מידע נכון, אבל הוא כן מבטיח שנוכל לקבל מידע בלי קשר לטרנזקציות אחרות שרצות ומעדכנות מידע. כלומר, הוא מאפשר לנו לקבל מידע כלשהו.

* *CRUD –, Delete , Update , Read Create .*

כדאי לציין שלא רק שלא מובטח שנקבל מידע שאי פעם היה נכון, אלא יש עוד כמה תופעות לוואי ל- Dirty Reads: אנחנו יכולים לקבל את אותה שורה מספר פעמים, לקבל שורה שנמחקה, לקבל מידע “חצי נכון” – ובקיצור, לקבל מידע שאינו אמין בהגדרה.

**Read Committed Snapshot & Allow Snapshot Isolation**

אם נסתכל על ה- Isolation Levels שראינו בינתיים: SERIALIZABLE, REPEATABLE READ, READ COMMITTED ו- READ UNCOMMITTED הם כולם מבוססים על נעילות (ועל התעלמות מנעילות, במקרה של READ UNCOMMITTED). עם זאת, נעילות זאת לא הדרך היחידה להבטיח isolation בין טרנזקציות שונות

ב- SQL Server נעשה שימוש ב- Optimistic Concurrency משולב במנגנון של versioning גם בעת עבודה עם SNAPSHOT ISOLATION LEVEL ו- READ COMMITTED SNAPSHOT ISOLATION LEVEL.

* *Optimistic Concurrency Control היא שיטה לבקרת מקביליות במסדי נתונים יחסיים ללא שימוש בנעילות.*

**שמירת גרסאות -** אם תשימו לב ל- Isolation Levels מבוססי הנעילות, תשימו לב שיש שם בעייה מהותית: יש מצבים שבהם מישהו רוצה לקרוא מידע – רק לקרוא אותו – וננעל כי מישהו אחר עדכן אותו ועדיין לא עשה commit. ואז, כדי למנוע dirty reads, אם אנחנו סתם רוצים לקרוא מידע, אפילו אם נעשה את זה בטרנזקציה שהיא ב- READ COMMITTED (הדיפולט, ההכי פחות מחמיר שעדיין מבטיח נכונות של המידע) – נינעל ונצטרך להמתין שמישהו יסיים לכתוב. וכל מה שאנחנו רוצים לעשות זה רק לקרוא.

דרך אחת “לפתור” את הבעייה היא לקרוא באמצעות READ UNCOMMITTED. אבל אז – לא מובטחת נכונות של המידע. דרך אחרת, היא להשתמש ב- isolation level מבוסס גרסאות (versioning). כלומר, כשאני אקרא את המידע, אני לא רוצה לקרוא את המידע שחלקו מעודכן וחלקו לא – אני רוצה לקרוא את הגרסא הנכונה האחרונה של המידע. אני לא רוצה לראות “שברי מידע” מטרנזקציה שפועלת ברגע זה, אבל אני לא רוצה להמתין שהיא תסתיים (ובטח לא רוצה לעכב טרנזקציה אחרת מלהתחיל). אני רוצה לקרוא את הגרסא הנכונה האחרונה של המידע, ואם הוא משתנה באמצע, להמשיך ולהבטיח שכל הקריאות שאני עושה יהיו שייכות לאותה גרסא שראיתי בהתחלה.

למעשה, אנחנו רוצים “הפרדת רשויות” בין מי שקורא למי שכותב. שמי שקורא לא יחסום את מי שכותב, ושמי שכותב לא יחסום את מי שקורא. lock-free, אבל שהמידע בכל זאת יהיה נכון.

* *Lock Free Algorithm הוא אלגוריתם המיועד לביצוע בידי מספר תהליכונים (Threads) באופן מקבילי ואינו משתמש לשם כך בנעילות חוסמות.*

SNAPSHOT ISOLATION

כדי לעשות שימוש ב- SNAPSHOT ISOLATION יש לאפשר את זה קודם ברמת ה-DB באמצעות הפקודה:

Alter Database [DBName] Set Allow\_Snapshot\_Isolation On

Go

כאשר אנחנו עושים שימוש ב- SNAPSHOT ISOLATION מובטח לנו שנוכל לקרוא כל שורה שנרצה. מובטח לנו גם שכל שורה שנקרא שייכת לאותה “גרסא” של מידע, כי כל שינוי שקורה במקביל למעשה יוצר “version” של השורה שלנו ב- tempdb (ובכך בא לידי ביטוי אחד החסרונות של עבודה עם row-versioning, היא יוצרת יותר עומס על tempdb וצריך לוודא שהחומרה שלנו תעמוד בכך, בהתאם לעומס שיש לנו וכמות הגרסאות שיהיו). כלומר, אנחנו יכולים לקרוא כל מידע שאנחנו רוצים, ולא מונעים מאף עדכון לקרות בטרנזקציות מקבילות.

מה לגבי עדכון בטרנזקציה שלנו? אז אנחנו יכולים לעדכן כמה שאנחנו רוצים, ואולי זה יצליח ואולי לא. פה בא לידי ביטוי אלמנט ה- Optimistic Concurrency. במצבים מסויימים אנחנו עלולים להיתקל ב- update conflict.

למשל, אם טרנזקציה אחרת עדכנה את הערכים “מתחת לרגליים” שלנו. יש כל מיני דרכים להתגבר על הבעייה, למשל באמצעות שימוש ב- hint בשם UPDLOCK כדי לסמן כבר בזמן הקריאה שאנחנו עומדים לעדכן ערכים מסויימים. עם זאת, בפועל, SNAPSHOT ISOLATION מתאים בעיקר לתרחיש שיש לנו הרבה reader-ים שאנחנו לא רוצים שייחסמו מ- writer-ים שפועלים ב- Isolation Level שונה. או לחלופין, יש לנו גם writer-ים שפועלים ב SNAPSHOT ISOLATION, אבל אנחנו יודעים שלא יהיו בינהם התנגשויות, לא בינם לבין עצמם ולא כלפי אחרים.

* *Table hints - מרמז למערכת לפעול בדרך מסוימת. המתכנת אינו מגדיר איך לבצע את המשימה וזה נעשה אוטומטית על ידי המערכת, אך יש דרכים להשפיע עליה במקרה הצורך . UPDLOCK NOLOCK , HOLDLOCK ועוד, ניתן לקרוא עליהם* [*כאן*](https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table?view=sql-server-2017) *.*

READ COMMITTED SNAPSHOT ISOLATION

אחת האפשרויות שקיימות ב- SQL Server היא להחליף את ההתנהגות של READ COMMITTED בהתנהגות מבוססת versioning. ההחלפה הזאת מתבצעת באופן גורף, ברמת דטאבייס, ולאחר שעושים אותה READ COMMITTED יתפקד למעשה בתור “READ COMMITTED SNAPSHOT”.

אז מה בעצם אומר READ COMMITTED SNAPSHOT? הבסיס דומה ל- SNAPSHOT ISOLATION (במובן של הניהול גרסאות למידע) למעט ההבדלים הבאים:

בעוד שב-SNAPSHOT ISOLATION ניהול הגרסא מתבצע ברמת הטרנזקציה ב- READ COMMITTED SNAPSHOT ISOLATION הוא מתבצע ברמת ה-statement. כלומר, אם אנחנו פותחים טרנזקציה, בודקים את מס’ השורות בטבלה A ולאחר מכן את מס’ השורות בטבלה B, כאשר במקביל, בין בדיקת השורות בשתי הטבלאות – טרנזקציה אחרת מכניסה שורות לטבלה B ועושה commit.

ב-SNAPSHOT ISOLATION: נקבל את מס’ השורות בטבלה B כמו שהוא היה בתחילת הטרנזקציה, ולכן כשבוצעה ההוספה מקודם לטבלה B, עדיין נשמר המידע על הגרסא שבה אין את השינוי הזה – וזה המידע שנקרא.

ב-READ COMMITTED SNAPSHOT ISOLATION: נקבל את מס’ השורות בטבלה B לאחר ההוספה, כי ניהול הגרסאות הוא ברמת ה- statement ומכיוון שזה statement חדש אנחנו נסתכל על הגרסא הכי עדכנית עכשיו.

כפועל יוצא מה-bullet הקודם, העדכונים הם למעשה pessimistic concurrency, ולא optimistic concurrency. מכיוון שכל statement עומד בפני עצמו מבחינת versioning, כאשר מגיעים להריץ את ה-UPDATE, אז SQL Server לא מחוייב בגלל הגדרת ה- ISOLATION LEVEL לקרוא את הגרסא שמתאימה לתחילת הטרנזקציה.

ההתנהגות במקרה של עדכונים שונה. בפעולות עדכון לא מתבצעת גישה לגרסא ספיציפית, אלא מתבצעת גישה לנתון העדכני ביותר ונעשה שימוש בנעילות על מנת להבטיח עדכון.

כלומר, בכל הקשור לעדכונים, READ COMMITTED SNAPSHOT ISOLATION מתבסס על נעילות ולא על optimistic concurrency, כך ש-writer-ים ימתינו אחד לשני מצד אחד, אך מצד שני – לא ניתקל בעדכונים שנכשלים בגלל שהערך שעודכן הוא לא הערך העדכני ביותר, כפי שעלול לקרות ב- SNAPSHOT ISOLATION.

למעשה, השילוב של שתי הנקודות הללו הופך את השימוש ב- READ COMMITTED SNAPSHOT ISOLATION למתאים לרוב סוגי האפליקציות, ללא יותר מדי התאמות ושינויים. בנוסף, באפליקציות שמערבות קריאות וכתיבות מקבילות, או שבמקביל אליהן נעשה מול ה-DB גם קריאות ועדכונים ע”י אפליקציות אחרות – היתרון בא אפילו יותר לידי ביטוי. מרוויחים שקריאות לא חוסמות כתיבות (ולהיפך), אולם גם מקבלים התנהגות עדכון “נורמלית” (במובן שהיא יותר כמו שאנחנו רגילים), שלא דורשת שינויים אפליקטיביים רבים.

למעשה, קיימים בשוק שרתי DB שונים שבהם ההתנהגות הדיפולטית היא למעשה ההתנהגות של READ COMMITTED SNAPSHOT. ב-SQL Server זה לא מופעל כברירת מחדל, ויש לאפשר את ההתנהגות באמצעות הפקודה:

Alter Database [DBName] Set Read\_Committed\_Snapshot On

Go

צריך כמובן להיות מודעים להשפעות הנלוות, בעיקר העומס הגבוהה יותר שיווצר על tempdb בשל ה- versioning. בכל אופן, למרבית האפליקציות כנראה יש יתרון רב במעבר לעבודה עם READ COMMITTED SNAPSHOT ISOLATION

**הגדרת הטרנזקציה ובחירת ה- IsolationLevel**

עבודה עם טרנזקציות מקוד מתבצעת באמצעות שימוש ב- System.Transactions.TransactionScope, או באמצעות יצירת SqlTransaction עם המתודה BeginTransaction של SqlConnection.

לשני האובייקטים ניתן להעביר (ב- constructor של TransactionScope או כפרמטר של BeginTransaction) את ה- Isolation Level המבוקש באמצעות ה- enum המתאים.

אם עושים שימוש בטרנזקציות בקוד, הדיפולט הוא IsolationLevel.Serializable, כלומר לעשות שימוש ב- Isolation Level המחמיר ביותר שיש.

ייתכן שזאת אופציה טובה לברירת-מחדל של ה- framework (פחות סיכוי לטעויות, תואם לסטנדרט של SQL), אבל מצד שני – מדובר בבחירה שהיא לא מתאימה למרבית האפליקציות. במרבית המקרים רצוי לעבוד עם ReadCommitted (כאשר אם רוצים לעשות שימוש ב- row version, רצוי להפעיל ברמת ה-DB את הפיצ’ר של READ COMMITTED SNAPSHOT).

אם לא מגדירים טרנזקציה בעצמכם, ולמשל עובדים עם Entity Framework, אז כשקוראים ל- SaveChanges נעשה שימוש דווקא ב- default הגיוני יותר בעבודה מול SQL Server, שהוא READ COMMITTED.

**השפעות הגדרת טרנזקציה על שאילתות עתידיות**

כאשר מייצרים connection חדש באמצעות SqlConnection, למעשה לא בהכרח מייצרים connection חדש. כדי לשפר את ביצועי האפליקציות שעובדות מול מסדי נתונים, מופעל ע”י ADO.NET מאחורי הקלעים מנגנון של connection pool, ש-“ממחזר” למעשה connections בהם נעשה שימוש, כדי לא להקים ולסגור כל הזמן connection-ים מול ה-DB תוך תשלום ה- overhead של פעולות אלה.

כדי לייצר הפרדה לוגית בין instance-ים שונים של SqlConnection, או במילים אחרות כדי לעשות אשלייה של connection חדש, מורץ בין “מחזורים” שונים של ה- connnection פעולה שמטרתה “לאפס” את ה- connection, באמצעות קריאה ל- sp\_reset\_connection.

עד SQL Server 2014 לא התאפס ה- Isolation Level בעת “מיחזור” ה- connection. כלומר, אם בקוד הגדרנו טרנזקציה (ונניח לא העברנו Isolation Level, כך שהיא רצה כברירת מחדל בתור SERIALIZABLE), ולאחר מכן יצרנו SqlConnection אחר ואפילו לא הגדרנו בו טרנזקציה – עדיין הוא “יסחוב” את ה- Isolation Level שהגדרנו קודם. הרבה פעמים, זה ממש לא מה שנרצה. במיוחד אם לא היינו מודעים גם ל- default המחמיר.

החל מ- SQL Server 2014, בוצע שינוי כך ש- sp\_reset\_connection מאפס את ה- Isolation Level להיות READ COMMITTED, הדיפולט של SQL Server (שכמובן, יתפקד כ- READ COMMITTED SNAPSHOT אם הגדרנו זאת).

* *ActiveX Data Objects) ADO.NET) הינה ספריה של רכיבים שמטרתם לעבוד עם מידע. השימוש העיקרי של ADO.NET הוא עבודה מול Database לשליפה ועדכון של מידע.*

**שילוב בין התנהגויות שונות באותה טרנזקציה**

על אף שלטרנזקציה מוגדר ISOLATION LEVEL אחד, ניתן עדיין לשלב כדי לקבל התנהגויות של Isolation Level מסוג Y בתוך טרנזקציה שעובדת עם Isolation Level מסוג X.

דוגמא טובה לזה, היא שאם יש לנו טרנזקציה שרצה ב- READ COMMITTED ISOLATION, ואנחנו רוצים לקרוא מידע (שאילתת SELECT) ולאפשר dirty-reads, אנחנו יכולים לעשות שימוש ב- hint שנקרא NOLOCK, כלומר לכתוב את השאילתה כך:

BEGIN TRANSACTION

SELECT col1, col2

FROM tbl1 WITH(NOLOCK(

SELECT col1, col2

FROM tbl2

COMMIT

כך נשיג שהשליפה של tbl1 מתבצעת בתור READ UNCOMMITTED, בזכות ה- hint של ה- NOLOCK, ואילו השליפה מול tbl2 מתבצעת READ COMMITTED (ברירת המחדל, כי לא הגדרנו isolation level).

דוגמא אחרת, היא שאם הפעלנו את הפיצ’ר של READ COMMITTED SNAPSHOT ISOLATION, והיינו רוצים לעשות שליפה ספיציפית שמתנהגת בתור READ COMMITTED מבוסס הנעילות, היינו יכולים להשתמש ב- hint שנקרא READCOMMITTEDLOCK (צורת השימוש דומה לדוגמא מקודם, פשוט להחליף את NOLOCK) כדי לקבל התנהגות של READ COMMITTED מבוסס נעילות, על אף שלאחר הפעלת הפיצ’ר הזה ה-READ COMMITTED הופך להיות מבוסס versioning.

.