transactions

م. عبد البديع مراد

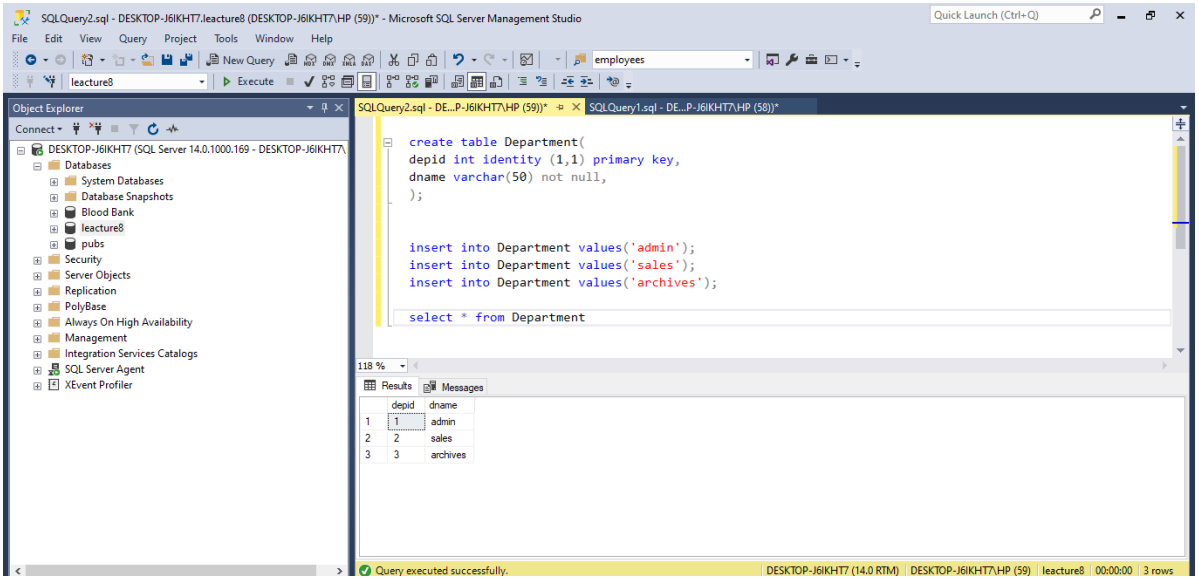
23/11/2023

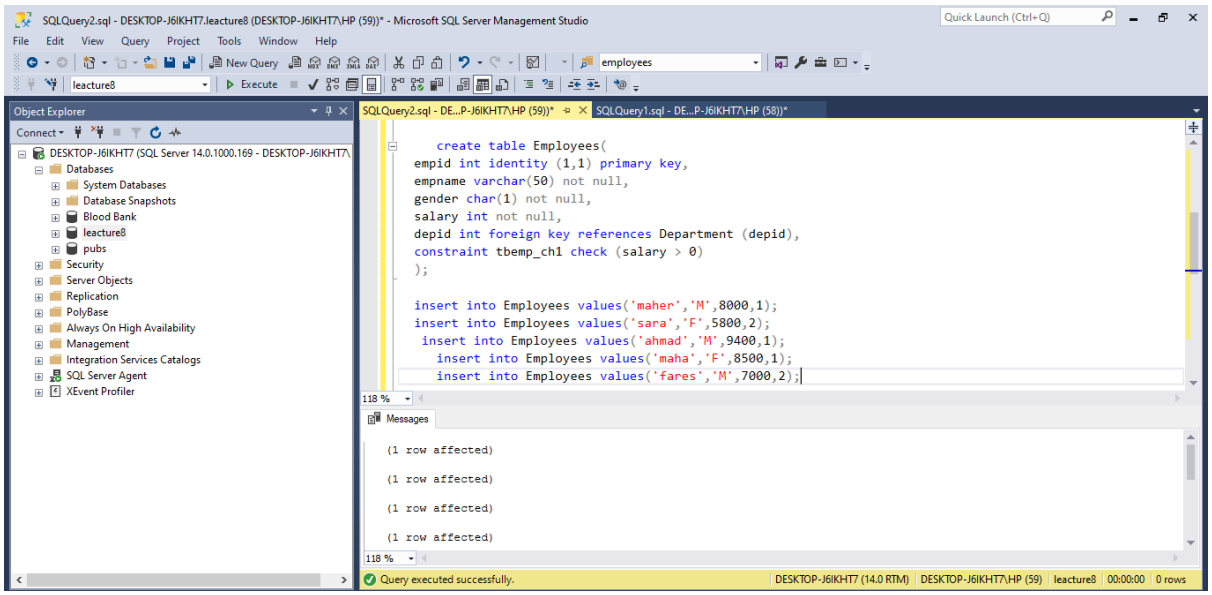
RB Informatics;

قواعد المعطيات (2)

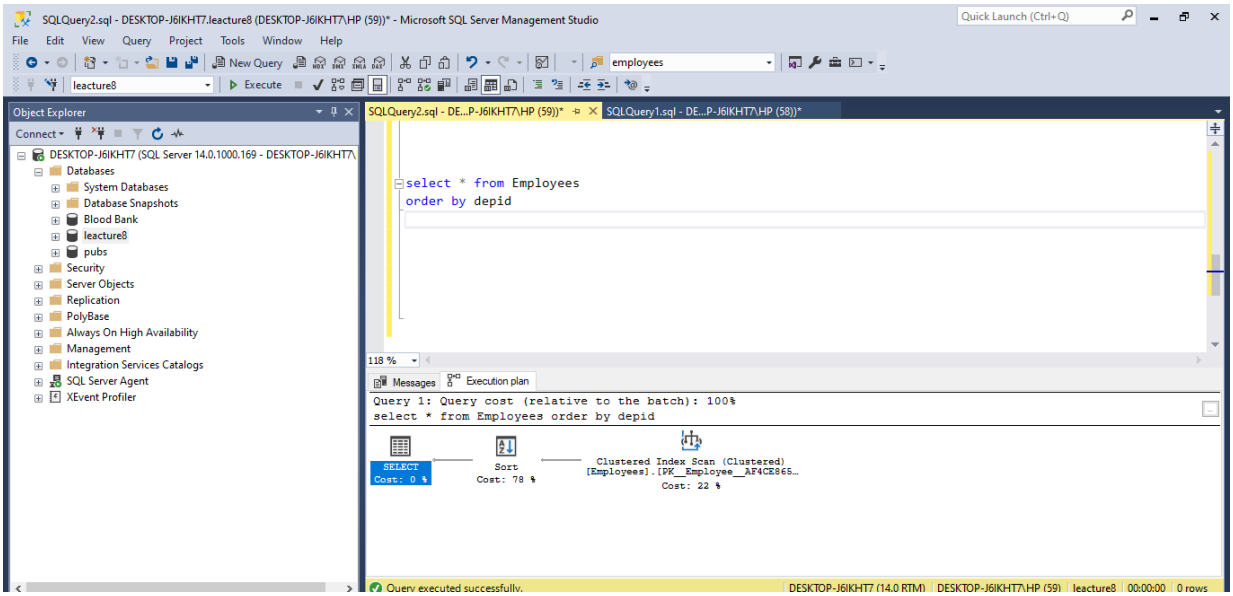
تكلمنا في المحاضرة السابقة عن ال *fragmentation* وتحدثنا أيضاً عن ال *join* وأنواعها وسنتحدث في هذه المحاضرة عن كيفية حفاظ ال *database* على المعلومات لتجنب حدوث أي نوع من أنواع التقارب عند القيام بأكثر من عملية على ال *database* في نفس الوقت.

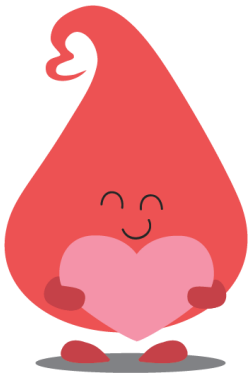
فلو قمنا بداية ببناء نفس ال *tables* الذين قمنا ببنائهم في المحاضرة السابقة:





لو قمنا بعد ذلك بتنفيذ ال *Query* التالية وعرض ال *execution plan*:





نلاحظ أنها استخدمت ال *clustered index* كالمعتاد.

لنقم بتوضيح فكرة التضارب بشكل أكبر من خلال المثال التالي:

لدينا ال *table* التالي:

table 1

ID	*Name*	*Salary*
1	*A*	1000
2	*B*	2000
3	*C*	3000

وقام شخص بالاتصال لهذه ال *database* وقام بعمل *connect* ومن ثم قام بتنفيذ تعليمة ال *select* على هذا ال *table*.

وبالتأكيد فإن ذلك لن يحدث أي تغيير على ال *table* لأن تأثير ال *select* يكون على ال *view* فقط. ولكن ال *database* تبدأ في حماية نفسها عندما نبدأ تنفيذ تعليمات *dml* والتي هي *insert, update, delete*.

فلو قمنا الآن بتنفيذ هذه ال *query* على ال *table* السابق:

```
update table1
set Name =' D'
where ID = 1
```

عند تنفيذ هذه التعليمة سيصبح هنالك *lock* على ال *row* ذو ال $ID = 1$ أي يتم قفله ولكن لم ذلك؟

لنفترض في هذه الأثناء هنالك شخص آخر يقوم بتنفيذ ال *query* التالية على ال *table* السابق:

```
select *
from table1
where ID = 1;
```

فماذا سيكون ال $name$ في ال $result\ set$ المعادة هل سيكون $'A'$ أم $'D'$ ؟

الجواب انه إذا كان في هذه ال $query$ يستخدم $clustered\ scan$ فعندها لن يكون قادراً على قراءة ال $data$ لأن ال $lock$ يكون على كامل ال $table$.....

بينما لو أردنا الوصول إلى row معين باستخدام $query$ تستخدم ال $seek$ فذلك ممكن.

إذاً فإن ال $select$ التي قام بها الشخص الثاني في مثالنا تبقى في حالة $waiting$ وبالطبع ذلك لا يعني بالضرورة وجود مشاكل في ال $database$ أو ال $query$ المنفذة. ولكن ال $query$ تكون في حالة انتظار لل $transaction$ حتى ينتهي..... وال $data$ التي يتم التعديل عليها في هذه الأثناء يطلق عليها $dirty\ data$ وال $database$ بشكل عام تكون في حالة $dirty$.

> ملاحظة:
>
> عملية ال $update$ السابقة التي قمنا بها هي التي تقابل بدء عملية ال $transaction$.

وعند التطبيق العملي فإن ال $transaction$ يعبر عن بدايته ب $begin\ transaction$ وكل $transaction$ ينتهي إما ب: $commit$: لحفظ التغييرات التي طرأت.

$roll\ back$: للتراجع عن التغييرات لحد آخر حالة $stable$ لل$database$.

إذاً عندما يحصل تغيير على ال $database$ فإن هذا التغيير سيكون أثره واضحاً ضمن أي $session$ تنفذ $query$ على هذه ال $database$.

> ملاحظة:
>
> بالنسبة لعمليات ال $insert$ الاعتيادية التي نقوم بها على سبيل المثال فإن $sql\ server$ يحوي $feature$ يطلق عليها $auto\ commit$ تعني أن أي تعليمة dml تنفذ فإنه مباشرة بعد تنفيذها يقوم بعمل $commit$.

ولكن هذا ليس دائماً صحيح إذ اننا في الحياة العملية من الممكن أن نقوم بتنفيذ عدة تعليمات dml ككتلة واحدة فإما أن ينجح التنفيذ أو يفشل.

فنحن في مثل هذه الحالة بحاجة لتخزين كتلة التعليمات هذه ضمن $transaction$ وفي نهايته إما يكون $commit$ في حال نجاح التنفيذ لهم ككل او ان يفشل تنفيذ إحدى التعليمات على الأقل فيقوم بعمل $roll\ back$ حينها.

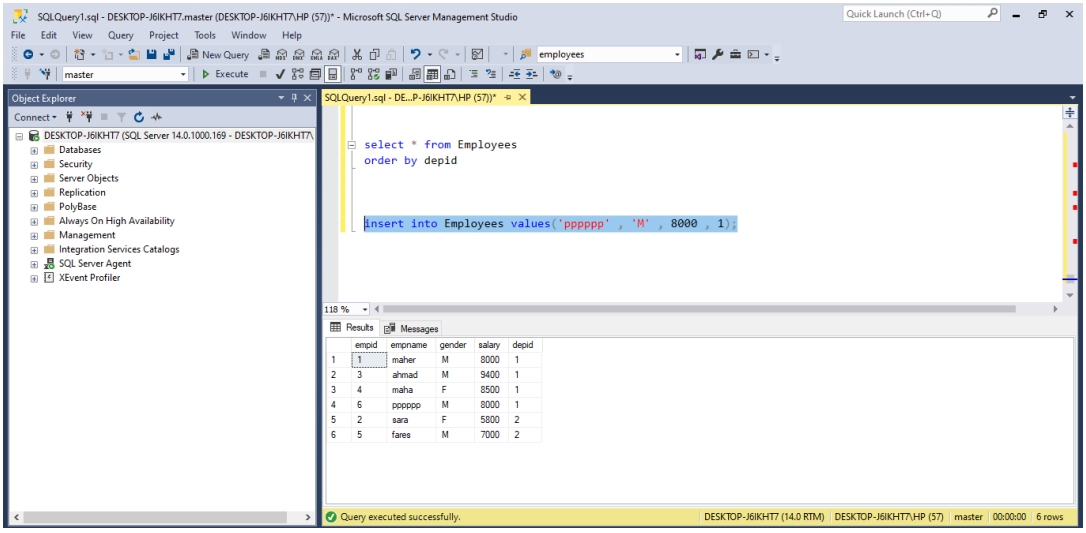
> ملاحظة:
>
> إن ال $session$ التي نقوم بتنفيذ تعليمات ال dml ضمنها بإمكانها رؤية التغييرات التي تحدث ضمن هذه ال $session$.

أي بالنسبة لعملية ال $update$ التي قمنا بتنفيذها على سبيل المثال فلو تمت ضمن $transaction$ معين وبدون أن ننهيه ب $commit$ أو $roll\ back$ وقمنا في نفس هذه ال $session$ بعمل $select$ فإن ال $result$ العائدة ستكون حاملة لل $Name\ 'D'$.

بينما لا يمكن رؤية هذه التغييرات في أي $session$ أخرى حتى يتم الانتهاء من هذه ال $transaction$.

الآن نذهب للتطبيق على ال $tables$ الذين قمنا ببنائهم سابقاً:

نقوم بتنفيذ تعليمة ال *insert* التالية على ال *Employees table*:



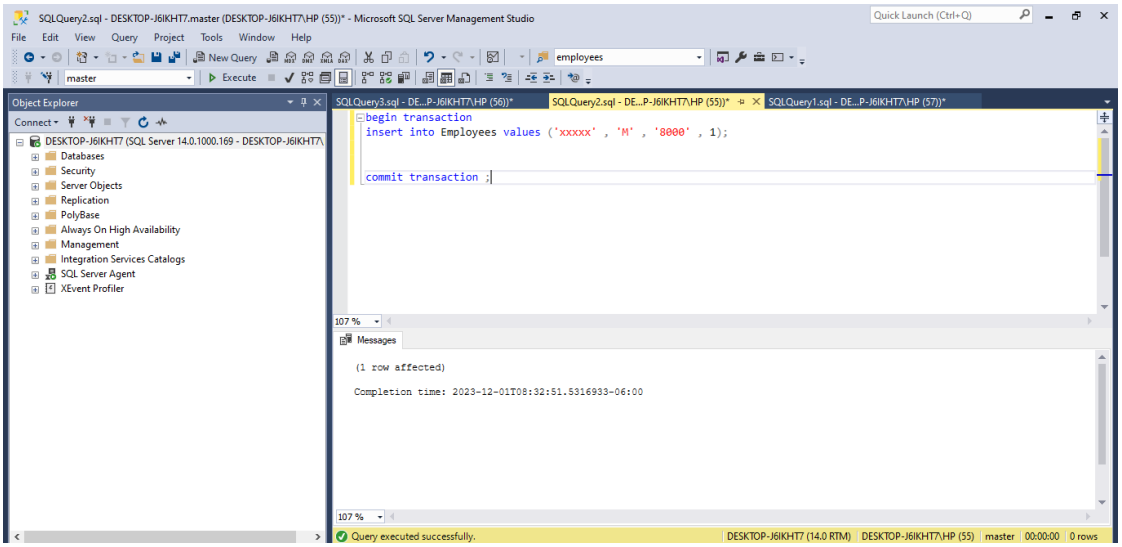
كما تكلمنا فإن هذه التعليمة يحصل لها *commit* مباشرة.

الآن نقوم بفتح *session* جديدة وبدء *transaction* جديد:

Begin transaction

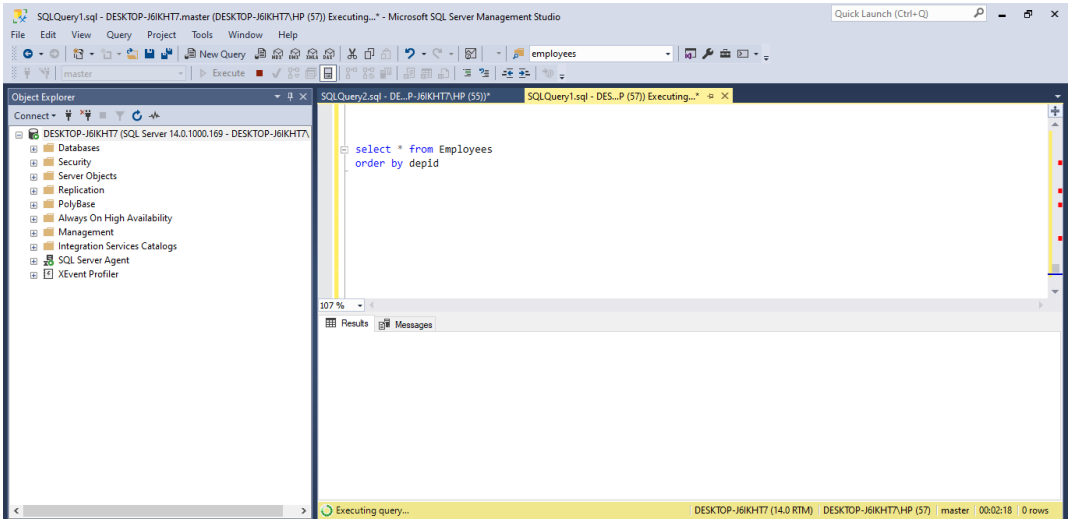
> **ملاحظة:**
>
> فور كتابة *begin transaction* فإن ال *auto commit* الموجود *by default* يتم إلغاؤه.

وضمن هذا ال *transaction* سنقوم بعملية *insert* أخرى ل *record* ونعطيه *Name* وليكن *'xxxx'*:

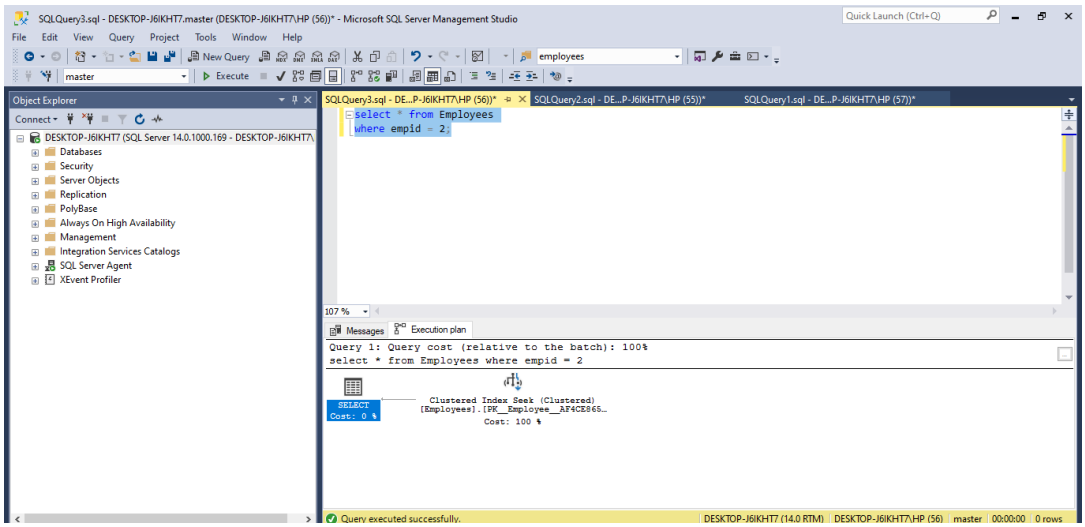


فعملية ال *insert* الأخيرة أصبحت معلقة الآن وال *database* أصبحت بحالة *dirty* وأصبح هنالك *lock* على ال *Employees table* لأنه لا يستطيع توقع ما سيحصل في ال *transaction* إما *DML* أخرى او *commit* أو *roll back*.

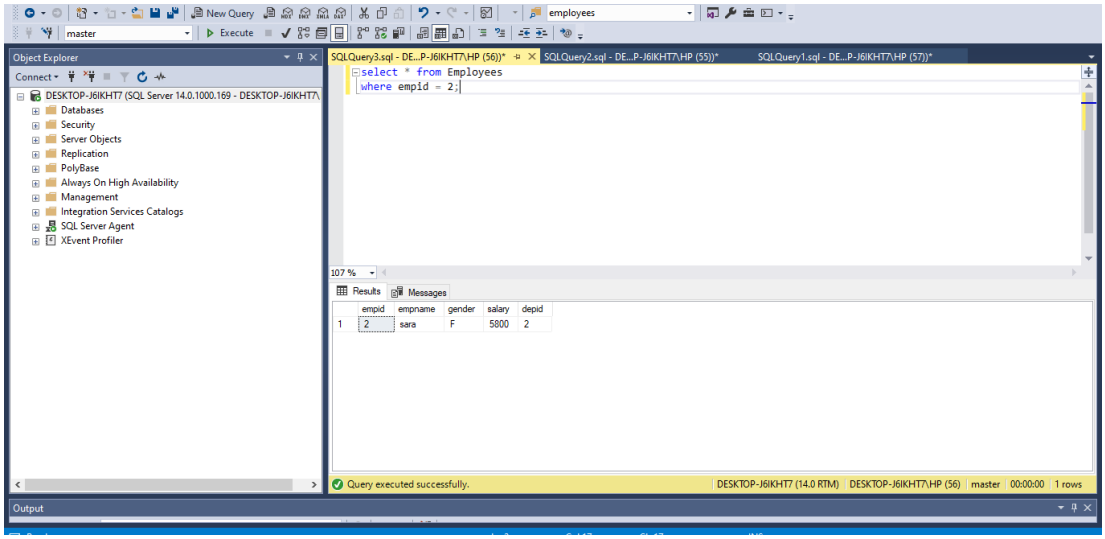
لذلك فلو ذهبنا إلى ال *session* الأخرى وقمنا بكتابة تعليمة *select* من هذا ال *table* فإن ال *query* ستبقى في حالة *waiting* وذلك يعود سببه كما تحدثنا أن ال *operation* هنا هي *scan clustered*:



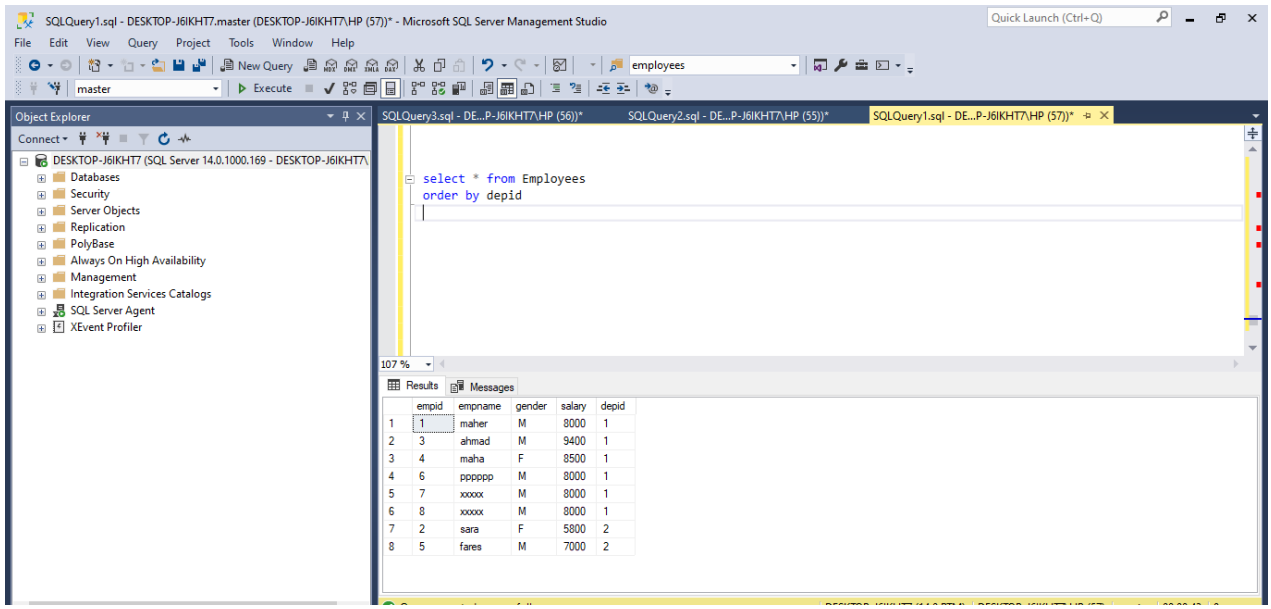
في الوقت ذاته سنقوم بفتح *session* ثالثة وننفذ فيها ال *query* التالية والتي تستخدم ال *seek* في ال *execution plan* الخاص بها كما نشاهد...



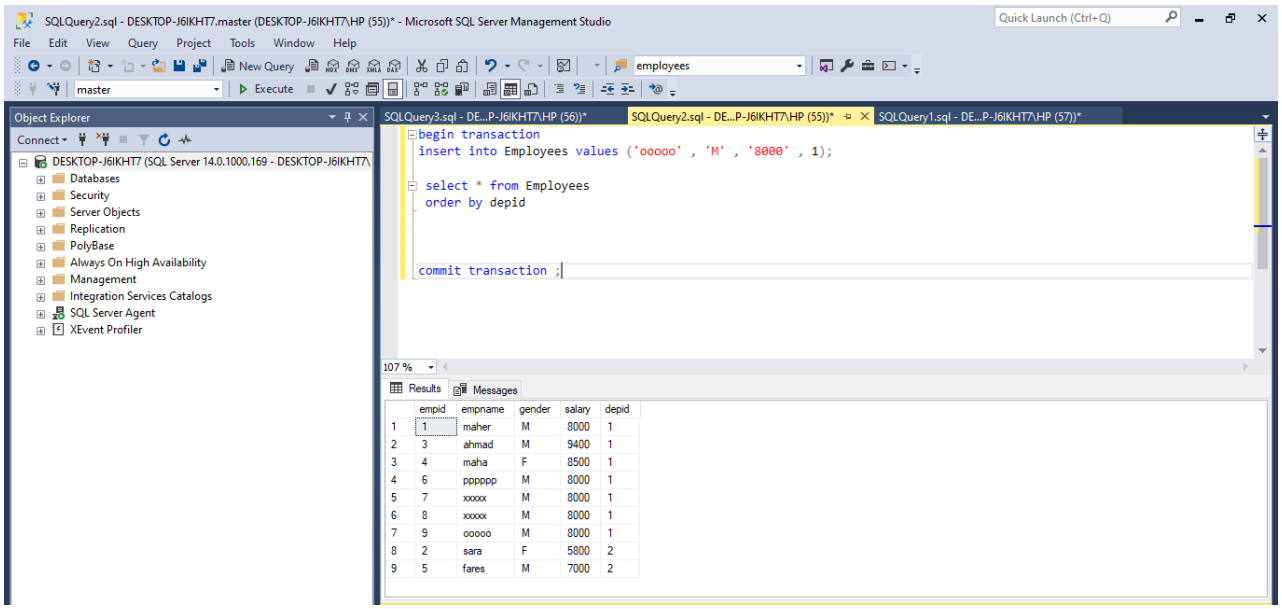
نقوم بتنفيذ هذه ال *query* للتأكد من أنها تتنفذ وتحضر *result set*:



الآن نعود للـ *session* التي تحوي الـ *transaction* ونقوم بإنهائه إما بـ *commit transaction* أو *roll back transaction* وفي مثالنا نستخدم *commit* وبعد ذلك نعود للـ *select* التي كانت في حالة *waiting* فنرى أنها تنفذت وأحضرت *data*:



وكما تحدثنا أن التغييرات هي مرئية بالنسبة للـ *session* الواحدة نفسها ونرى مثالنا على ذلك:



ملاحظة:

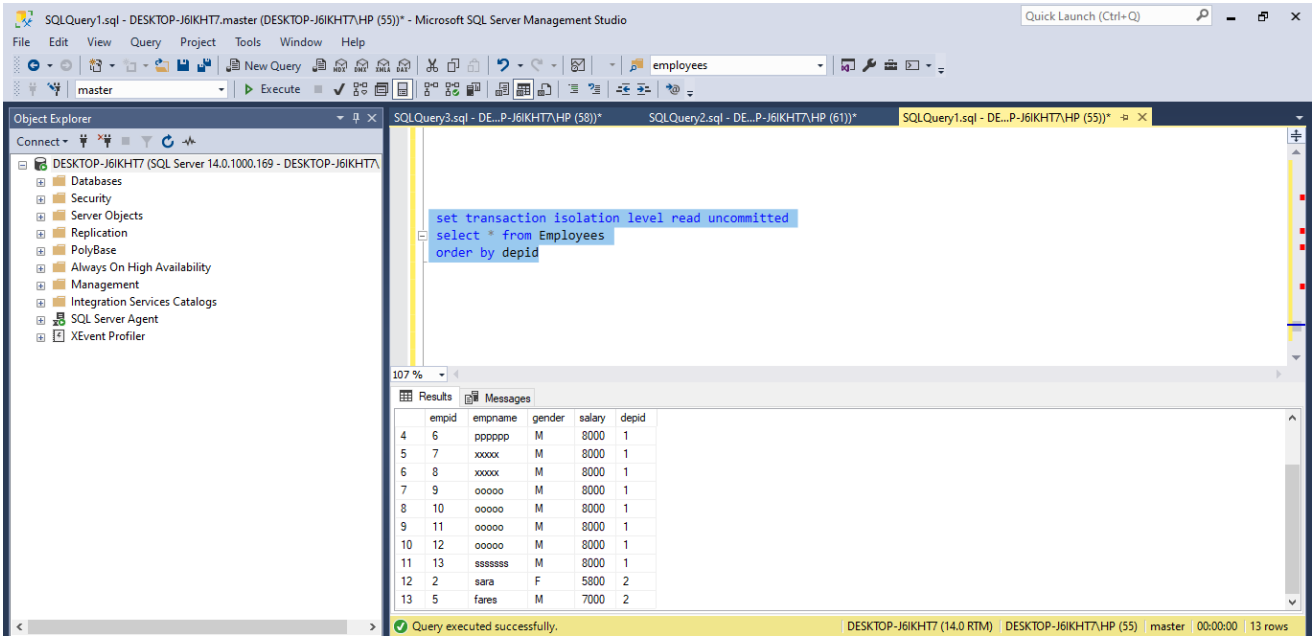
في الـ *transactions* يوجد *options* معينة قليلة الاستخدام وفي كثير من الأحيان يكون استخدامها خاطئ.

"If you get tired, learn to rest.
Not to quit"

set transaction isolation level read uncommited

وفي هذه الحالة ستقرأ ال data ولو كانت uncommited أي dirty data.

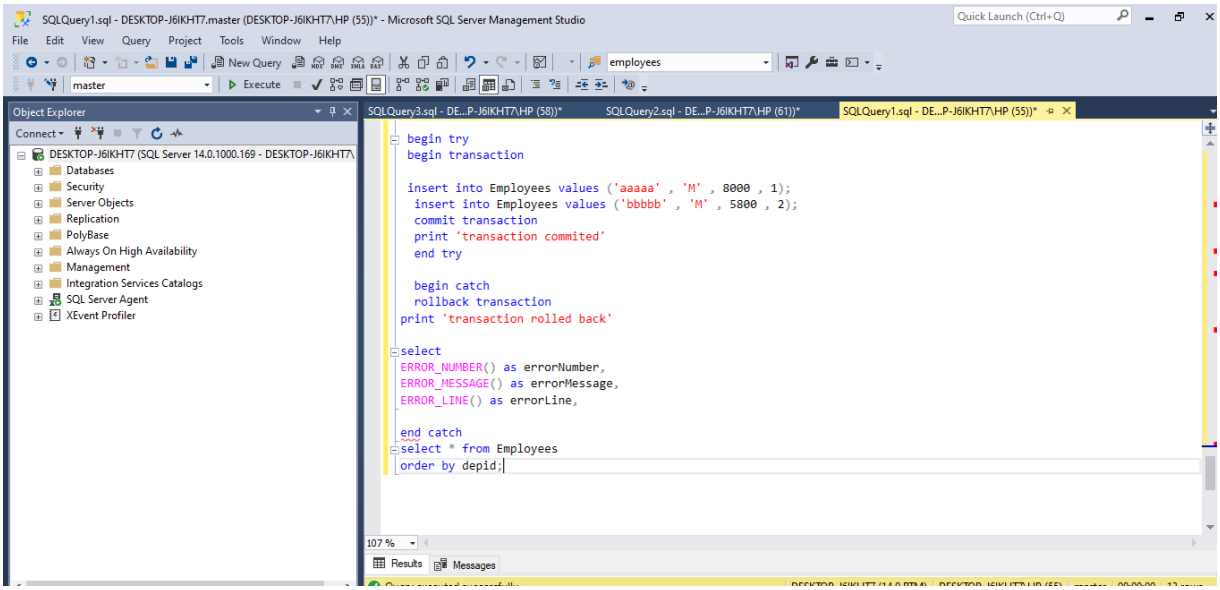
أي لو قمنا بتشغيل ال transaction السابق مع هذه ال feature ونفذنا ال select query فإنه سيقوم مباشرة بإحضار ال data وبدون أن يدخل في حالة waiting ولكن هذا بالتأكيد أمر خاطئ جداً فال result التي عادت وأصبحت موجودة لدينا وقمنا باتخاذ قرارات وتنفيذ تعليمات على أساسها يمكن أن يحدث roll back لها ضمن ال transaction في أي لحظة... لذلك فإن هذه ال feature لا يجب استخدامها ابداً.

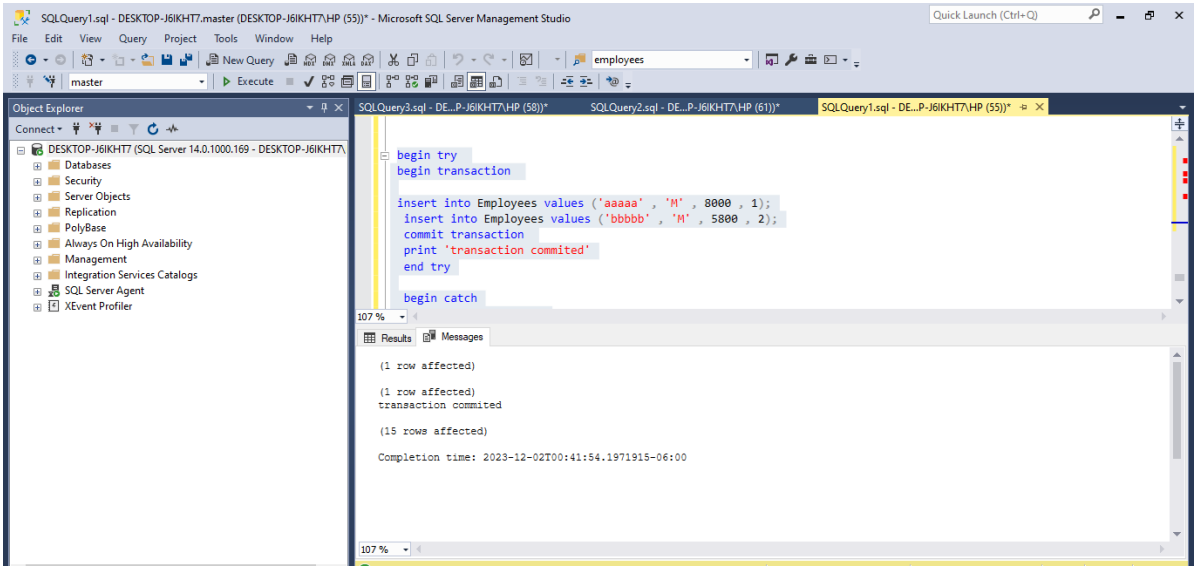


ملاحظة:

عادة يتم في ال transaction استخدام try و catch لتحديد الخطأ والعملية التي ستنفذ على هذا الأساس.

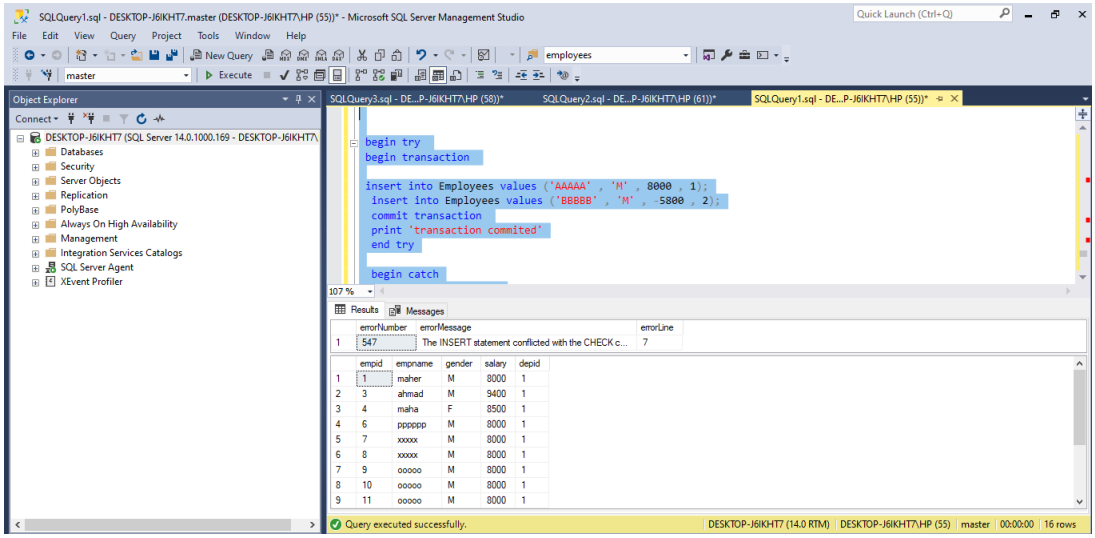
مثال:

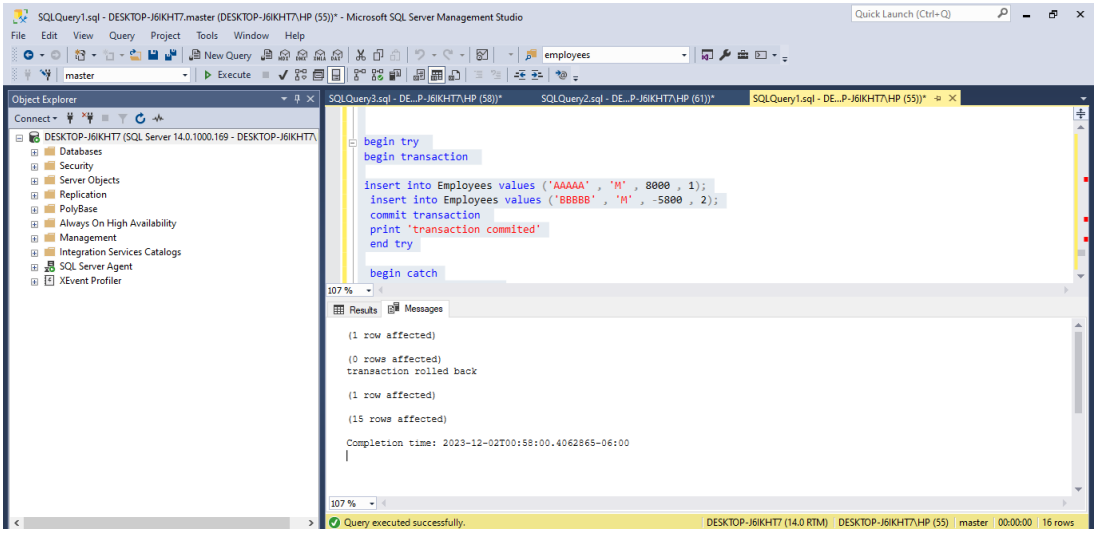




كما نلاحظ أن هذا ال *transaction* تمت معالجته بشكل صحيح.

الآن سنقوم بتنفيذ *transaction* آخر يحوي على عملية *insert* خاطئة او مخالفة للشرط *constraint* وعندها بالتأكيد سيقوم بعمل *roll back* :





نلاحظ طباعة *transaction roll back* وعدم إضافة ال *rows* 2 الى ال *table*.

Dead Lock

ليكن لدينا هذين ال $tables$ 2 البسيطين على سبيل المثال t_1, t_2:

$t1$

ID	$Name$
1	t_1

$t2$

ID	$Name$
1	t_2

لنفترض وجود شخص قام ببدء $transaction$ على ال $table$ الأولى t_1 وقام خلالها بعملية $update$ على ال $table$ ولتكن من الشكل:

$$Begin\ transaction$$
$$update\ t_1$$
$$set\ Name =' t_1t_1'$$
$$where\ ID = 1$$
$$\vdots$$

تحدثنا مسبقاً أن هذه التعليمة ستقوم بعمل $lock$ على ال t_1table.

وسنفترض أيضاً وجود شخص ثاني في $session$ ثاني أيضاً قام ببدء $transaction$ على ال $table$ الثانية t_2 وقام خلالها بعملية $update$ على ال $table$ ولتكن من الشكل:

$$Begin\ transaction$$
$$update\ t_2$$
$$set\ Name =' t_2t_2'$$
$$where\ ID = 1$$
$$\vdots$$

وأيضاً هذه التعليمة ستقوم بعمل $lock$ على ال $t_2\ table$.

ولحد هذه النقطة فإن التنفيذ سيكون سليم وليس هناك مشاكل ولكن المشكلة الحقيقة ستكون لو قمنا في تتمة $transaction$ الأول بالقيام بالعملية التالية:

"Your simplicity makes you beautiful"

$$update\ t_2$$
$$set\ Name =' ttt_2'$$
$$where\ ID = 1;$$

وبنفس الطريقة لو قمنا بال $transaction$ الثاني بالعملية بالتالية:

$$update\ t_1$$
$$set\ Name =' ttt_1'$$
$$where\ ID = 1;$$

وفي هذه الحالة نكون قد دخلنا بحالة $dead\ lock$ لا نهائية فكلا ال $transaction$ ينتظر الآخر كي ينتهي.

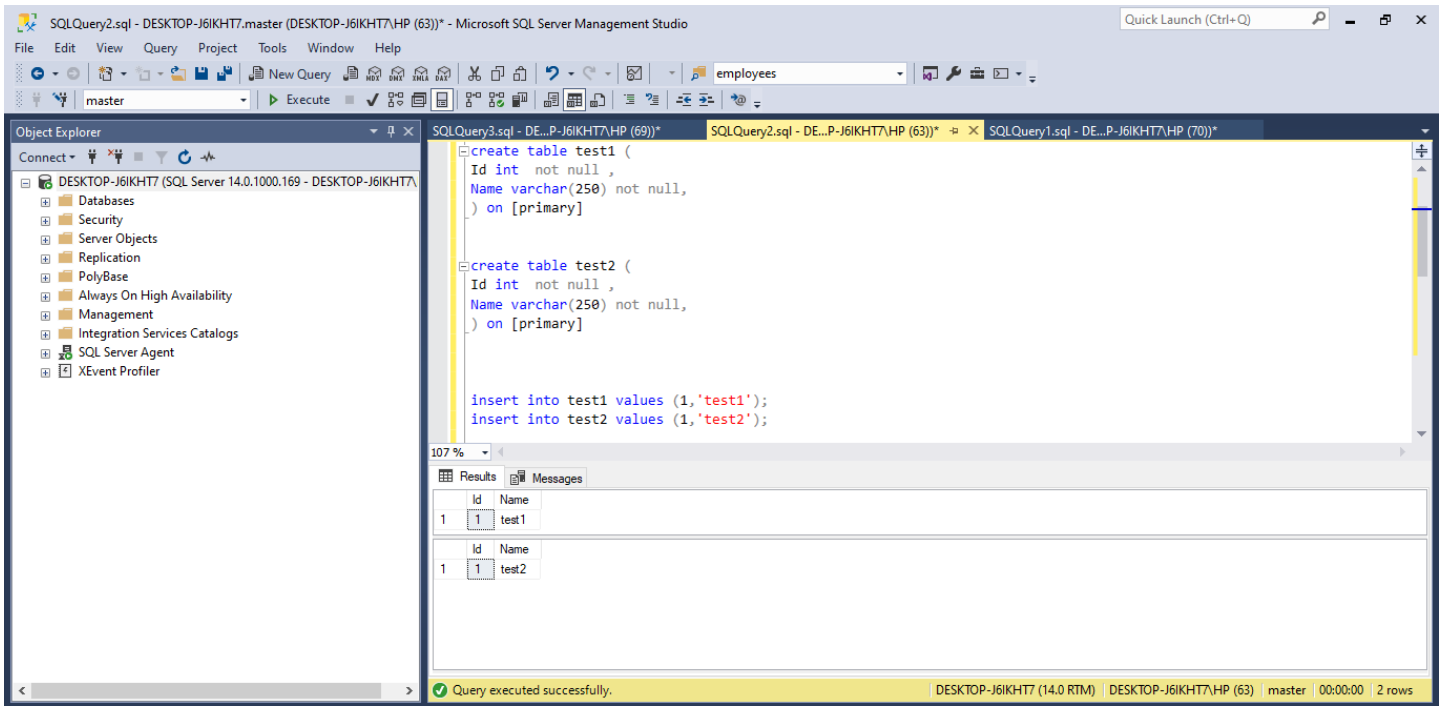
في هذه الحالة يقوم ال *engine* الخاص بال *database* بالانتظار لمدة 5 ثواني ..... إما أن تحقق إحدى ال *sessions* على حساب الأخرى أو تحصل على حالة *dead lock* وعندها فإن واحداً من هذين ال *sessions* 2 سيموت حتماً.
ويتم تحديد ذلك بالنظر إلى ال *transaction* لكل منهما فال *session* ذو *transaction* الأكبر ينتصر ويتم تنفيذه وعمل *commit* ويغلب ال *session* الآخر ويتم عمل *roll back* لل *transaction* الخاص به باعتباره الأضعف.

> **ملاحظة:**
>
> إن ال *dead lock error* هو من أكثر الأخطاء شيوعاً عند التعامل مع ال *database* وهو من أكثرها تأثيراً على ال *performance* وهو أيضاً من أكثرها تعقيداً في الحل.

ولتجنب مواجهته:

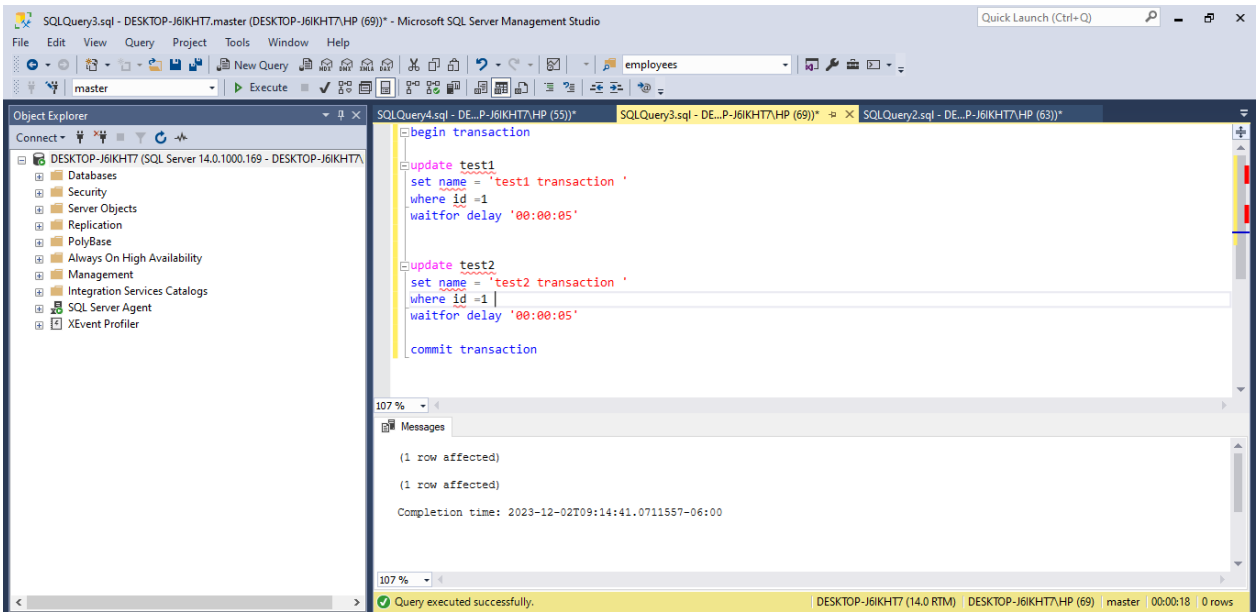
- من الخطوة الأولى لبناء ال *database* يجب أن يكون ال *design* بشكله الصحيح والمنتظم.
- تقليل زمن ال *transactions* قدر الإمكان أي تقسيم ال *code* او ال *logic* لأكثر من *transaction*.
- ألا يكون *DML Block* كبير جداً.
- معالجة ال *queries* التي تستهلك زمناً طويلاً كبناء *index* يخدم هذه ال *query* على سبيل المثال:

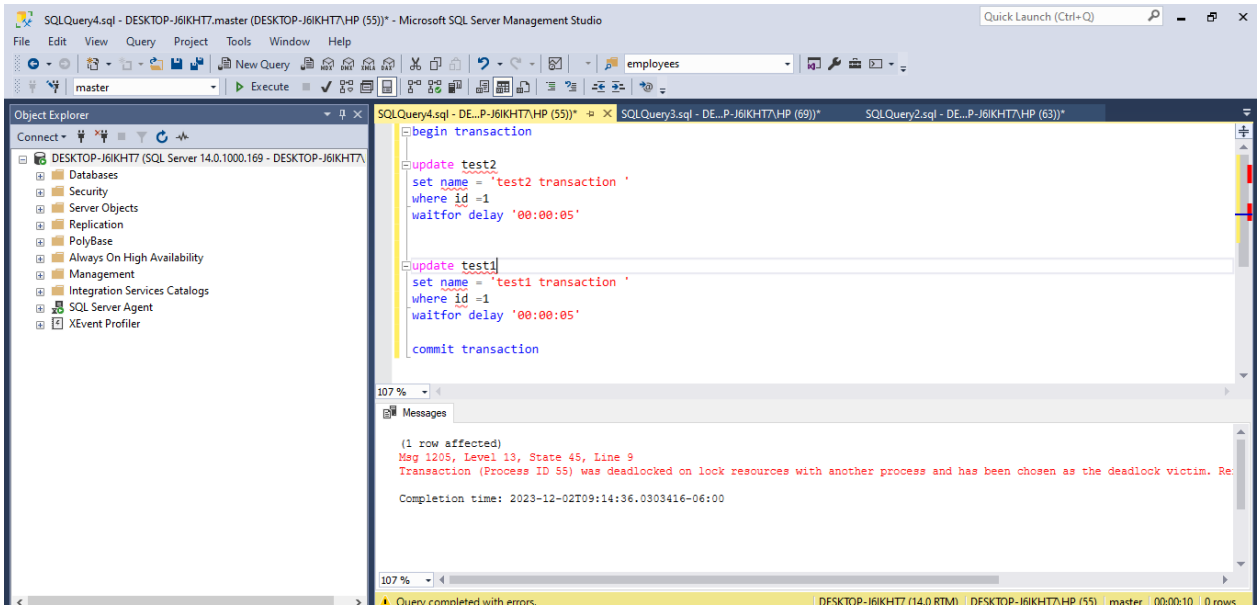
لنشاهد التنفيذ الفعلي لما سبق.
سنقوم ببناء هذين ال *tables* 2 اولاً:



"All I want for Christmas is to study Database"

الآن سنقوم بفتح *sessions* 2 وبدء *transaction* في كل واحدة منهما ونقوم من خلالهما بتنفيذ تعليمات بنفس الشكل الذي ذكرناه في المثال السابق لرؤية حالة ال *dead lock*.



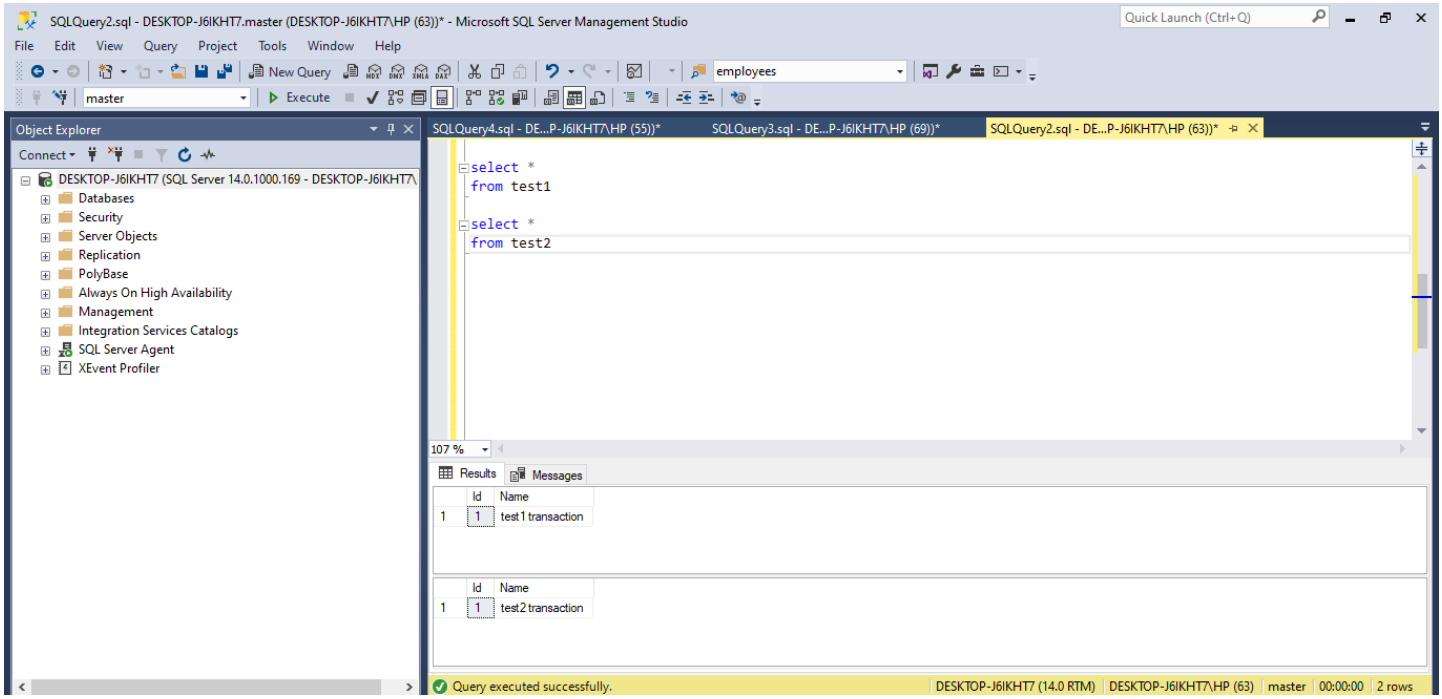


ملاحظة:

تعليمة التأخير *wait for* تم وضعها لنفترض وجود العديد من التعليمات بدلاً منها.

"Coffee in one hand, confidence in another"

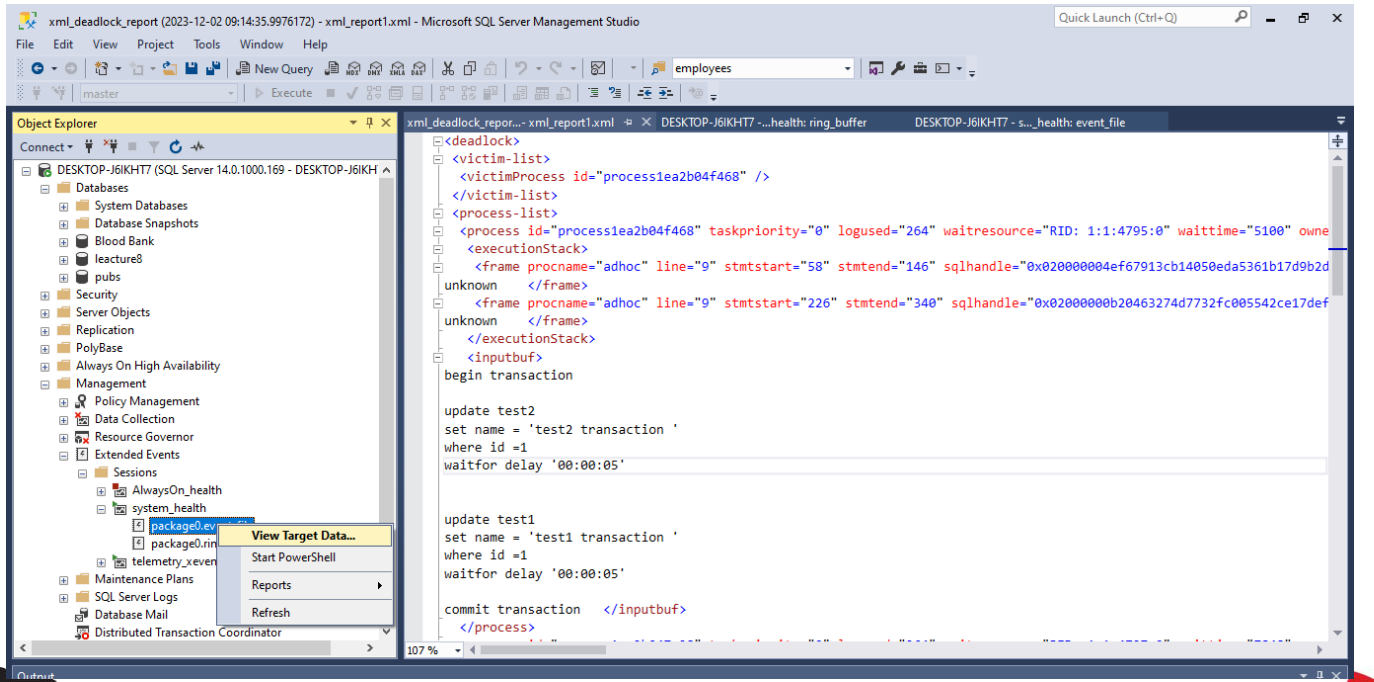
وكما لاحظنا أن ال *session* الأولى تم تنفيذها ونجحت على حساب ال *session* الثانية التي فشلت واعادت *dead lock error* وتم عمل *roll back* لل *transaction* الخاص بها.



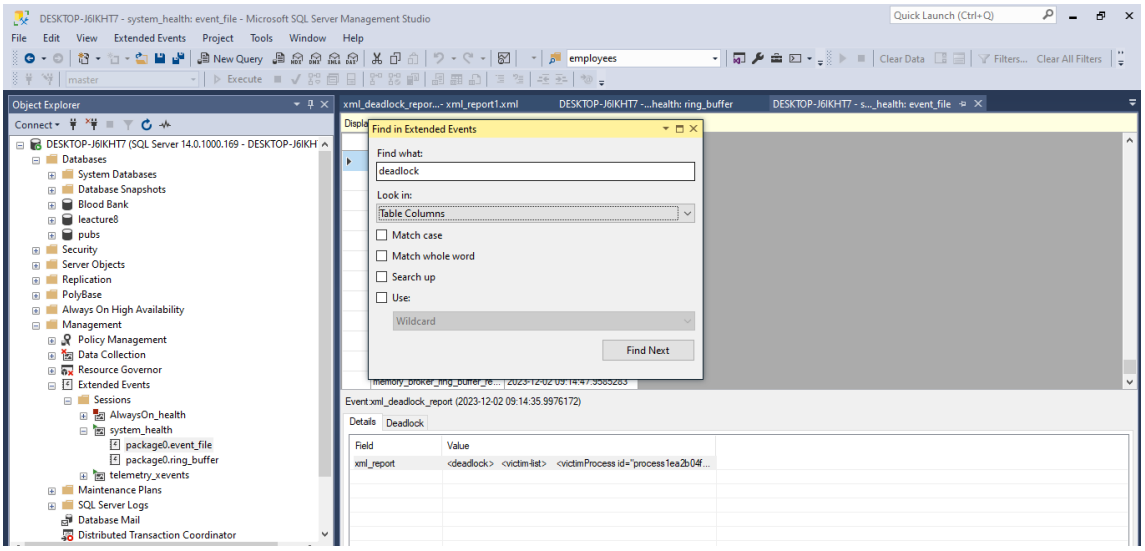
ملاحظة:

هنالك عدة طرق لمشاهدة ال *dead lock* ضمن ال *database* ومشاهدة تأثيرها ومن هو ال *transaction* الذي فشل او كما يسمى *victim transaction* (الضحية) ومن هو ال *transaction* الذي قام بقتله.

وإحدى الطرق لرؤية ال *dead lock* هي من خلال ال *side bar* بالضغط على *management* ← *Extended Events* ← *sessions* ← *system health* ومن ثم *click* يميني على *package0.event_file* ومن ثم *view target data*:

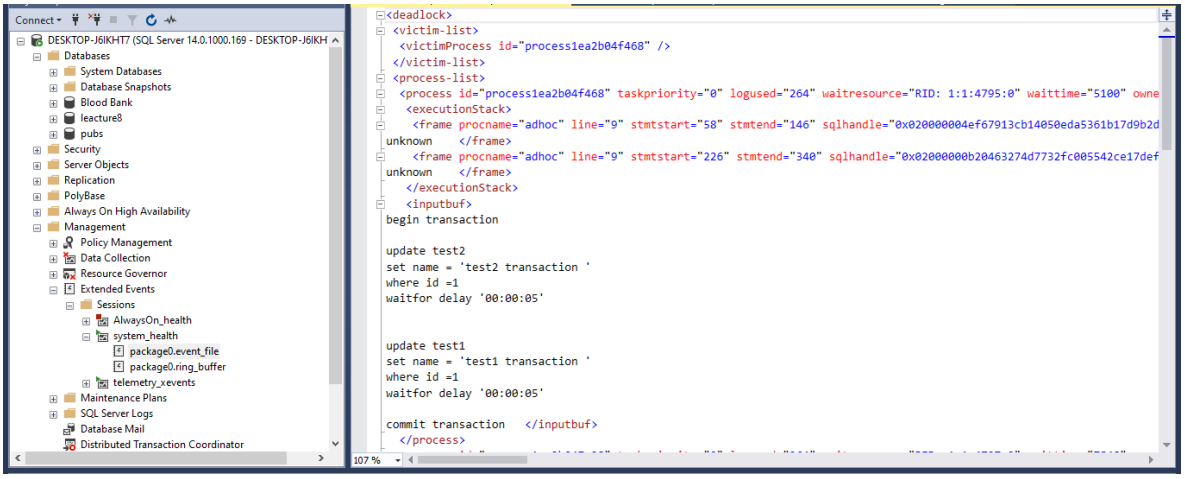


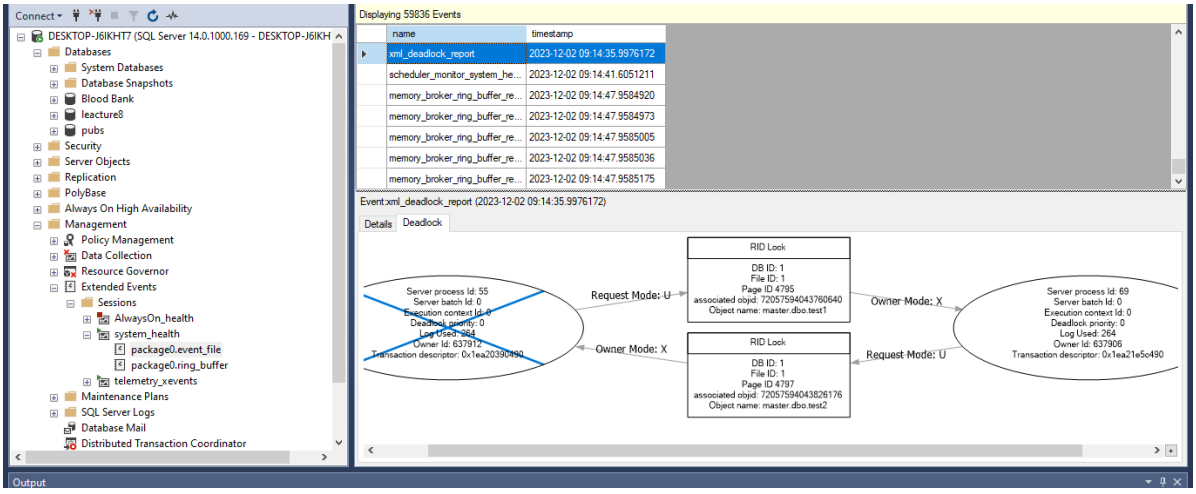
كما نلاحظ أنه هنا قام بإعطائنا كل التغيرات على ال *database* ويمكننا ضمن هذا الملف عبر الاختصار *ctrl + f* بالبحث عن *dead lock* وايضاً يمكننا تحديد ال *table* الذي نريد البحث فيه:



وفي حالتنا ظهر لدينا هذا ال *dead lock* الذي حصل نتيجة التنفيذ الذي قمنا به في المثال السابق.

نضغط عليه لنشاهد ال *block of code* يشرح عملية ال *dead lock* التي حصلت:





The End