Chapter 9 & 10 : Part 1
Oracle 12c Database Data Concurrency : Transactions and Locking
B + tree

By David Itshak
shaked19@gmail.com
http://www.ildba.co.il/author/cimid/
http://www.sqlserver.co.il/?cat=940
Global Hebrew Virtual PASS Chapter :
https://www.youtube.com/watch?v=x4hGjYGBfkc
https://www.youtube.com/watch?v=eJO8G9if3EY
SqlSaturday Israel 2016 :
Reference and Credits

Oracle® Database Concepts
12c Release 1 (12.1)
E41396-13
https://docs.oracle.com/database/121/CNCPT/toc.htm

Oracle® Database Performance Tuning Guide
12c Release 1 (12.1)
E49058-06
https://docs.oracle.com/database/121/TGDBA/toc.htm

Oracle® Database SQL Language Reference
12c Release 1 (12.1)
E41329-20

Oracle Essentials(Oracle Database 12c), 5th; O'Reilly, 2013

Oracle OCA Oracle Database 12c Administrator Certified Associate Study Guide Exam

Pro Oracle Database 12c Administration, 2 edition ISBN 1430257288 2013

Apress Oracle Database Transactions and Locking Revealed (2014)

Oracle Learning Library

Pro SQL Server Internals 2014  Apress
SQL Server Concurrency  Locking, Blocking and Row Versioning  By Kalen Delaney
SQL 2016 Book Online
Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
ISBN 1-55860-508-8

“Teamwork is essential. It allows you to blame someone else.” (Anonymous)
Part II: Concurrency Control

- 3 Concurrency Control: Notions of Correctness for the Page Model
- 4 Concurrency Control Algorithms
- 5 Multiversion Concurrency Control
- 6 Concurrency Control on Objects: Notions of Correctness
- 7 Concurrency Control Algorithms on Objects
- 8 Concurrency Control on Relational Databases
- 9 Concurrency Control on Search Structures
- 10 Implementation and Pragmatic Issues
Chapter 10: Implementation and Pragmatic Issues

• 10.2 Data Structures of a Lock Manager

• 10.3 Multi-Granularity Locking and Lock Escalation

• 10.4 Transient Versioning

• 10.5 Nested Transactions for Intra-transaction parallelism

• 10.6 Tuning Options

• 10.7 Overload Control

• 10.8 Lessons Learned

“All theory, my friend, is grey; but the precious tree of life.”

(Johann Wolfgang von Goethe)
Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests.
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).
- The requesting transaction waits until its request is answered.
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests.
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.
Lock Table

- Dark blue rectangles indicate granted locks; light blue indicate waiting requests.
- Lock table also records the type of lock granted or requested.
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks.
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
- If transaction aborts, all waiting or granted requests of the transaction are deleted.
  - Lock manager may keep a list of locks held by each transaction, to implement this efficiently.
Organization of Lock Control Blocks

Transaction Control Blocks (TCBs)

<table>
<thead>
<tr>
<th>Transaction Id</th>
<th>Update Flag</th>
<th>Transaction Status</th>
<th>Number of Locks</th>
<th>LCB Chain</th>
</tr>
</thead>
</table>

Hash Table indexed by Resource Id

Resource Control Blocks (RCBs)

<table>
<thead>
<tr>
<th>Resource Id</th>
<th>Hash Chain</th>
<th>FirstInQueue</th>
<th></th>
</tr>
</thead>
</table>

Lock Control Blocks (LCBs)

<table>
<thead>
<tr>
<th>Transaction Id</th>
<th>Resource Id</th>
<th>Lock Mode</th>
<th>Lock Status</th>
<th>NextInQueue</th>
<th>LCB Chain</th>
</tr>
</thead>
</table>

...
Lock Table

- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently
Bucket Count and Performance

- A hash based on the first two letters of string and can return $26 \times 26 = 676$ different hash keys.
- Assuming that the hash table can accommodate all 676 different hash buckets and you have the data shown in Figure
- you will need to traverse at most two rows in the chain when you run a query that looks for a specific value.
Bucket Count and Performance

- Situation changes if your hash table does not have enough buckets to separate unique hash keys from each other.
- Figure illustrates the situation when a hash table has only 26 buckets and each of them stores multiple different hash keys.
- Now The same lookup of the Ann row requires you to traverse the chain of nine rows total.
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- **10.3 Multi-Granularity Locking and Lock Escalation**
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Reconciling Coarse- and Fine-grained Locking

**Problem:** For reduced overhead, table scans should use coarse locks
Detect conflict of page lock with tablespace lock

**Approach:** Set “intention locks” on coarser granules

**Multi-granularity locking protocol:**
- A transaction can lock any granule in S or X mode.
- Before a granule p can be locked in S or X mode, the transaction needs to hold an IS or IX lock on all coarser granules that contain p.

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IS</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>IX</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SIX</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

**Typical policy:**
- use coarse locks for table scans
- use fine locks otherwise
- escalate dynamically to coarse locks when memory usage for LCBs becomes critical
Intention Lock Modes

• In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  – *intention-shared* (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  – *intention-exclusive* (IX): indicates explicit locking at a lower level with exclusive or shared locks.
  – *shared and intention-exclusive* (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

• intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.
Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree.
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

**Granularity of locking** (level in tree where locking is done):

- **Fine granularity** (lower in tree): high concurrency, high locking overhead
- **Coarse granularity** (higher in tree): low locking overhead, low concurrency
The levels, starting from the coarsest (top) level are

- database
- area
- file
- record
SQL Server Locking Basics

- Unit of data locked (**lock resource**) – such as row, page, or table
- Type of locks acquired (**lock mode**) – shared, exclusive, update, and so on
- **Duration of the lock** – how long the lock is held
- **Lock ownership** – the "scope" of the lock (most locks are transaction scoped)
- **Lock metadata** – how to review current locking using the Dynamic Management
  
  - MSSQL : View (DMV) called sys.dm_tran_locks.
Lock resources

- When SQL Server locks a row in an index, it refers to it, and displays it, as a KEY lock, but keep in mind that SQL Server locks the entire index row, not just the key column.
- In some circumstances, SQL Server can also lock ranges of index rows.
- Locks on rows in a heap table (one without a clustered index) appear as RID (Row ID) locks in the sys.dm_tran_locks view.
- SQL Server supports two kinds of KEY locks, depending on the isolation level of the current transaction.
  - If the isolation level is **READ COMMITTED** or **REPEATABLE READ**, SQL Server attempts to lock the index rows it accesses while processing the query.
  - If the table has a clustered index, then the data rows are at the leaf level of the index, and so row locks for any data in a table with a clustered index will always appear as KEY locks.
- If the table is a heap, SQL Server might acquire KEY locks for the non-clustered index rows and RID locks for the data rows.
Lock duration

• **Default : Read Committed**
• SQL Server releases S locks as soon as it has read and processed the locked data.
• It holds an X lock until the end of the transaction, whether the transaction is committed or rolled back.
• It holds a U lock until the end of the transaction, unless it promoted the U lock to an X lock as with all X locks, remains for the duration of the transaction.
SQL Server Lock Resources

- Can lock user data resources at row, page, or table level.

- SQL Server will attempt to acquire row-level locks, to allow the highest degree of concurrency.

- However: lock escalation
Clustered index structure: Root level

- Table can have only one clustered index defined.
- Data pages are linked in a double-linked list where every page contains pointers to the next and previous pages in the chain. This list is called the *leaf level of the index, and it contains the actual table data*.
Clustered index structure: Leaf level

- Create a clustered index on the heap table with the data.
- First step SQL Server creates another copy of the data that is sorted based on the value of the clustered key.
- The data pages are linked in a double-linked list where every page contains pointers to the next and previous pages in the chain. This list is called the *leaf level of the index, and it contains the actual table data.*
Clustered index structure: Intermediate and root levels

- When the leaf level consists of the multiple pages, SQL Server starts to build an intermediate level of the index,
**Ordered index scan**

- Data on the leaf level of the index is already sorted based on *CustomerId column value*
- *SQL Server* can scan the leaf level of the index from the first to the last page and return the rows in the order they are stored.
**Ordered index scan execution plan**

- An ordered scan just means that SQL Server reads the data based on the order of the index key.
- SQL Server can navigate through indexes in both directions, forward and backward.
Clustered index structure: Leaf level

- Create a clustered index on the heap table with the data.
- First step: SQL Server creates another copy of the data that is sorted based on the value of the clustered key.
- The data pages are linked in a double-linked list where every page contains pointers to the next and previous pages in the chain. This list is called the *leaf level of the index, and it contains the actual table data.*

```
create table dbo.Customers
(
    CustomerId int not null,
    Name varchar(64) not null,
    -- Other columns
)
```

```
create clustered index IDX_CI
on dbo.Customers(CustomerId)
```
Clustered index structure: Intermediate and root levels

- When the leaf level consists of the multiple pages, SQL Server starts to build an intermediate level of the index,
Ordered index scan

- Data on the leaf level of the index is already sorted based on *CustomerId column value*
- *SQL Server* can scan the leaf level of the index from the first to the last page and return the rows in the order they are stored.
Ordered index scan

cREATE NONCLUSTERED INDEX IDX_NCI ON dbo.Customers(Name)

• While a clustered index specifies how data rows are sorted in a table, nonclustered indexes define a separate sorting order for a column or set of columns and persist them as a separate index structure.
• Leaf level of the nonclustered index is sorted based on the value of the index key—Name in our case.
• Every row on the leaf level includes the key value and row-id.
• For heap tables, row-id is the physical location of the row defined as file:page:slot address.
Problems:
Two transactions were both searching for the same row to modify (for example, the same customer row in the Customers table), using different access paths, and they could both reach the desired resource at the same time.

Each transaction could acquire an S lock on that row, but then each transaction would attempt to convert this lock to an X lock in order to perform the modification, but the S lock held by one transaction prevents the other from doing so.
<table>
<thead>
<tr>
<th>RESOURCE TYPE</th>
<th>EXAMPLE OF RESOURCE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>(3a01f8ac47a)</td>
<td>A lock on a single row on an index. This includes row locks taken on tables that have a clustered index on them. The resource is a hash value that can be retrieved against your table with <code>%lock%</code>.</td>
</tr>
<tr>
<td>PAGE</td>
<td>1:19216</td>
<td>A lock on an index or data page. Breaks down as &lt;FILE ID&gt; : &lt;PAGE NUMBER&gt; These map to the file_id and page_id fields in the sys.dm_os_buffer_descriptors DMV.</td>
</tr>
<tr>
<td>EXTENT</td>
<td>1:19216</td>
<td>A contiguous set of eight pages. Pages are allocated to tables in extents. &lt;FILE ID&gt; : &lt;FIRST PAGE NO&gt;</td>
</tr>
<tr>
<td>HoBT</td>
<td>72057594058637312</td>
<td>HoBT is a Heap or Balanced Tree (BTree). When a table is a heap (no clustered index), it protects the heap. Otherwise, it protects the BTree of the index.</td>
</tr>
<tr>
<td>OBJECT</td>
<td>2105058535</td>
<td>Normally a table lock but could be anything with an OBJECT_ID. If it's a table lock, then it will cover both data pages and all indexes on the table.</td>
</tr>
<tr>
<td>APPLICATION</td>
<td>0:[MyAppLock]: (6731ef3)</td>
<td>An application lock. Set by sp_getapplock.</td>
</tr>
<tr>
<td>METADATA</td>
<td>xml_collection_id = 65536</td>
<td>Used to lock SQL Server system metadata — e.g., when taking a schema stability lock on metadata of an XML column when querying a row.</td>
</tr>
<tr>
<td>ALLOCATION_UNIT</td>
<td>72057594039828480</td>
<td>Allocation Unit ID seen during deferred drop operations, such as on a large table. Also visible during minimally logged operations such as SELECT INTO.</td>
</tr>
<tr>
<td>FILE</td>
<td>0</td>
<td>Seen when adding or removing files from a database. No resource description information is published.</td>
</tr>
<tr>
<td>DATABASE</td>
<td>7</td>
<td>A lock against the entire database. This can be a shared transaction workspace lock to identify a connection in the DB or a transaction lock when altering the database. Changing from read_write to read_only would need an exclusive transaction against the database.</td>
</tr>
</tbody>
</table>
The levels, starting from the coarsest (top) level are

- *database*
- *table*
- *Page*
- *Row*
Locking Examples

• Example 1: SELECT with READ COMMITTED isolation level
• Example 2: SELECT with REPEATABLE READ isolation level
• Example 3: SELECT with SERIALIZABLE isolation level
• Example 4: Update with READ COMMITTED isolation level
• Example 5: Update with SERIALIZABLE isolation level (with an index)
• Example 6: Update with SERIALIZABLE isolation level not using an index
• Example 7: Creating a table
• Example 8: RID locks
Creation of the DBlocks view to display locks in the current database: SQL Server

```
IF EXISTS ( SELECT 1
       FROM sys.views
       WHERE name = 'DBlocks' )
   DROP VIEW DBlocks;
GO
CREATE VIEW DBlocks AS
SELECT request_session_id AS spid,
       DB_NAME(resource_database_id) AS dbname,
       CASE WHEN resource_type = 'OBJECT'
             THEN OBJECT_NAME(resource-associated_entity_id)
             WHEN resource-associated_entity_id = 0 THEN 'n/a'
             ELSE OBJECT_NAME(p.object_id)
       END AS entity_name,
       index_id,
       resource_type AS resource,
       resource_description AS description,
       request_mode AS mode,
       request_status AS status
FROM sys.dm_tran_locks t
LEFT JOIN sys.partitions p
       ON p.partition_id = t.resource-associated_entity_id
WHERE resource_database_id = DB_ID()
AND resource_type <> 'DATABASE';
```
[Production].[Product] table
USE [AdventureWorks2014]
GO

CREATE TABLE [Production].[Product]
(
    [ProductID] [int] IDENTITY(1,1) NOT NULL,
    [Name] [varchar](50), [Name] NOT NULL,
    [ProductNumber] [nvarchar](25), NOT NULL,
    [MakeFlag] [dbo].[Flag] NOT NULL CONSTRAINT [DF_Product_MakeFlag] DEFAULT (1),
    [FinishedGoodsFlag] [dbo].[Flag] NOT NULL CONSTRAINT [DF_Product_FinishedGoodsFlag] DEFAULT (1),
    [Color] [nvarchar](15), [Color] NOT NULL,
    [SafetyStockLevel] [smallint], [SafetyStockLevel] NOT NULL,
    [ReorderPoint] [smallint], [ReorderPoint] NOT NULL,
    [StandardCost] [money], [StandardCost] NOT NULL,
    [ListPrice] [money], [ListPrice] NOT NULL,
    [Size] [nvarchar](5), [Size] NOT NULL,
    [SizeUnitMeasureCode] [nchar](3), [SizeUnitMeasureCode] NOT NULL,
    [WeightUnitMeasureCode] [nchar](3), [WeightUnitMeasureCode] NOT NULL,
    [Weight] [decimal](8, 2), [Weight] NOT NULL,
    [DaysToManufacture] [int], [DaysToManufacture] NOT NULL,
    [Class] [nchar](2), [Class] NOT NULL,
    [Style] [nchar](2), [Style] NOT NULL,
    [ProductSubcategoryID] [int] NULL,
    [ProductModelID] [int] NULL,
    [SellStartDate] [datetime], [SellStartDate] NOT NULL,
    [SellEndDate] [datetime], [SellEndDate] NOT NULL,
    [DiscontinuedDate] [datetime], [DiscontinuedDate] NOT NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL NOT NULL CONSTRAINT [DF_Product_rowguid] DEFAULT (newid()),
    [ModifiedDate] [datetime], [ModifiedDate] NOT NULL CONSTRAINT [DF_Product_ModifiedDate] DEFAULT (getdate()),
    CONSTRAINT [PK_Product_ProductID] PRIMARY KEY CLUSTERED
    ( [ProductID] ASC )
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
GO
Example 1: SELECT with READ COMMITTED isolation level

USE AdventureWorks2014;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT *
FROM Production.Product
WHERE Name = 'Reflector';
SELECT *
FROM DBlocks
WHERE spid = @@spid;
COMMIT TRAN

• By default, SQL Server releases S locks as soon as it has finished reading the data.
• By the time we execute the SELECT from the view, SQL Server no longer holds the locks.
• As such OBJECT lock on the view (there is also a DATABASE lock on the AdventureWorks database, but the DBlocks view filtered out database locks).
Example 2: SELECT with REPEATABLE READ isolation level

```sql
USE AdventureWorks2014;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
SELECT * FROM Production.Product WHERE Name LIKE 'Racing Socks%';
SELECT * FROM DBlocks WHERE spid = @@spid AND entity_name = 'Product';
COMMIT TRAN
```

- This time, because the transaction isolation level is REPEATABLE READ, SQL Server holds the S locks until the transaction is finished and so we can see them in our results.
- Production.Product table has a clustered index, so the rows of data are all index rows in the leaf level.
- As such, output shows that the locks on the two individual data rows returned are KEY locks.
- The table also has a non-clustered index on the Name column and we can see two KEY locks at the leaf level of this non-clustered index, used to find the relevant rows.
- We can distinguish the clustered and non-clustered indexes by the value in the Index_ID column: the data rows have an Index_id value of 1, and the non-clustered index rows have an Index_ID value of 3.

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>PAGE</td>
<td>1:9057</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>(6b00b8eeda30)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>(6a00dd896688)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>PAGE</td>
<td>1:4606</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>6</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>(9502d56a217e)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>7</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>(9602945b3a67)</td>
<td>S</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Example 3: SELECT with SERIALIZABLE isolation level

USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT * 
FROM Production.Product 
WHERE Name LIKE 'Racing Socks%';
SELECT * 
FROM DBlocks 
WHERE spid = @@spid
AND entity_name = 'Product'; 
COMMIT TRAN
Example 3: SELECT with SERIALIZABLE isolation level

- S-mode KEY locks on the rows in the clustered index, and IS locks on the parent pages and object.
- The two-part mode RangeS-S indicates a key-range lock in addition to the lock on the key itself.

1. First part (RangeS) is the lock on the range of keys between and including the key holding the lock and the previous key in the index.

2. Key-range locks prevent other transactions from inserting any new rows into the table that meet the condition of this query; that is, it's not possible to insert any new rows with a product name starting with Racing Socks.

3. Key-range locks are held on ranges in the non-clustered index on Name (Index_id = 3) because that is the index used to find the qualifying rows.

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>PAGE</td>
<td>1:9057</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>(6b00b8eeda30)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>(6a00dd896688)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>PAGE</td>
<td>1:4606</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>6</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>(9502d56a217e)</td>
<td>RangeS-S</td>
<td>GRANT</td>
</tr>
<tr>
<td>7</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>(23027a50f6db)</td>
<td>RangeS-S</td>
<td>GRANT</td>
</tr>
<tr>
<td>8</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>(9602945b3a67)</td>
<td>RangeS-S</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Example 3: SELECT with SERIALIZABLE isolation level

- The two Racing Socks rows are Racing Socks, L and Racing Socks, M.

- 3 KEY locks in the non-clustered index because SQL Server must lock three different ranges of data:

  1. Range from the key preceding the first Racing Socks row in the index (Pinch Bolt) up to the first Racing Socks row (Racing Socks, L)
  2. Range between the two rows starting with Racing Socks.
  3. The range from the second Racing Socks row (Racing Socks, M) to the next key in the index (Rear Brakes).
Example 4: Update with READ COMMITTED isolation level

USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';
COMMIT TRAN

- 2 rows in the leaf level of the clustered index are locked with X locks.
- Page and table are then locked with IX locks.
Example 5: Update with SERIALIZABLE isolation level (with an index)

USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Name LIKE 'Racing Socks%';
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product';

• 2 rows in the leaf level of the clustered index are locked with X locks.
• Page and table are then locked with IX locks.
Example 5: Update with SERIALIZABLE isolation level (with an index)

- Key-range locks are on the non-clustered index, used to find the relevant rows.
- Range interval itself needs only an S lock to prevent insertions, but the searched keys have U locks, ensuring that no other process can attempt to UPDATE them.
- The keys in the table itself (index_id = 1) obtain the X lock when the actual modification is made.

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>PAGE</td>
<td>1:9057</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>680b9eeb3a30</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>1</td>
<td>KEY</td>
<td>6a00dd896688</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>PAGE</td>
<td>1:4606</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>6</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>9502d56a217e</td>
<td>RangeS-U</td>
<td>GRANT</td>
</tr>
<tr>
<td>7</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>23027a50f6db</td>
<td>RangeS-U</td>
<td>GRANT</td>
</tr>
<tr>
<td>8</td>
<td>AdventureWorks</td>
<td>Product</td>
<td>3</td>
<td>KEY</td>
<td>9602945b3a67</td>
<td>RangeS-U</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Example 6: Update with SERIALIZABLE isolation level not using an index

```
USE AdventureWorks ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.6
WHERE Color = 'White' ;
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'Product' ;
COMMIT TRAN
```

387 52 AdventureWorks Product 1 KEY (7800a7c21f271) RangeS-U GRANT
388 52 AdventureWorks Product 1 KEY (0a00de7e40b3) RangeS-U GRANT
389 52 AdventureWorks Product 1 KEY (74001f7d243b) RangeS-U GRANT
390 52 AdventureWorks Product 1 KEY (7e0015491d57) RangeS-U GRANT
391 52 AdventureWorks Product 1 KEY (7200adf6cb1d) RangeS-U GRANT
392 52 AdventureWorks Product 1 KEY (66006536b0c2) RangeS-U GRANT
393 52 AdventureWorks Product 1 KEY (03000d8f0ecc) RangeS-U GRANT
394 52 AdventureWorks Product 1 KEY (6a00dd896688) RangeX-X GRANT
395 52 AdventureWorks Product 1 KEY (ac0069ecc4d8) RangeS-U GRANT
396 52 AdventureWorks Product 1 KEY (c5009eaac9c) RangeX-X GRANT
397 52 AdventureWorks Product 1 KEY (d200a86fa050) RangeS-U GRANT
398 52 AdventureWorks Product 1 KEY (a000d1531292) RangeS-U GRANT
399 52 AdventureWorks Product 1 KEY (c90001557ad6) RangeS-U GRANT
400 52 AdventureWorks Product 1 KEY (d00106576f1c) RangeS-U GRANT
Example 6: Update with SERIALIZABLE isolation level not using an index

- As there was no useful index, a clustered index scan on the entire table was required, and so all keys initially received the RangeS-U lock;
- Four rows were eventually modified, the locks on those keys escalated to the RangeX-X lock.
- Two of the RangeX-X locks, and a few of the RangeS-U locks.
- Complete output has 501 RangeS-U locks, as well as IU locks on several pages, IX locks on two pages, and an IX lock on the table.
Example 7: Creating a table

```
USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN
SELECT * INTO newProducts
FROM Production.Product
WHERE ListPrice BETWEEN 1 AND 10;
SELECT * FROM DBlocks
WHERE spid = @@spid;
COMMIT TRAN
```

- When creating the new table, SQL Server acquires locks on six different system tables to record information about the new table.
- In addition, schema modification (Sch-M) locks are acquired.
Example 8: RID locks

USE AdventureWorks;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
UPDATE newProducts
SET ListPrice = 5.99
WHERE Name = 'Road Bottle Cage';
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'newProducts'
COMMIT TRAN

- There are no indexes on the newProducts table
- X lock on the row (RID).
  For RID locks: File Number:Page number:Slot number.
- As expected, SQL Server takes IX locks on the page and the table.
<table>
<thead>
<tr>
<th>Requested lock mode</th>
<th>IS</th>
<th>S</th>
<th>U</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent shared (IS)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Update (U)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Intent exclusive (IX)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive (X)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Exclusive locks

- SQL Server automatically acquires exclusive (X) locks on data in order to modify that data, during an INSERT, UPDATE, or DELETE operation.

- Only one transaction at a time can hold an X lock on a particular data resource, and X locks remain until the end of the transaction.

- The changed data is usually unavailable to any other process until the transaction holding the lock either commits or rolls back.

- READ UNCOMMITTED transaction isolation level, can read data exclusively locked by another transaction.
Intent locks

- you can have intent shared (IS) locks, intent exclusive locks (IX), and even intent update locks (IU), indicated in the request_mode column of the sys.dm_tran_locks view by IS, IX and IU, respectively.

- SQL Server can acquire locks at different levels of granularity (i.e., at the row, page, or table level), and so needs some mechanism that signals whether a component of a resource is already locked.

- Example: if one transaction attempts to lock a table, SQL Server must be able to determine whether it has already locked a row or a page of that table. Intent locks serve that purpose.
Intent locks

• Whenever a transaction acquires a lock at a lower level of granularity, it also acquires higher-level intent locks for the same object.

• For example, a transaction that holds an X lock on a row in the Customers table will also hold IX locks on both the page containing that row, and the Customers table.

• These Intent locks will prevent another transaction from locking the entire Customers table (acquiring an X lock on the table).
SQL server : U (Update) Lock

• A transaction acquires a U lock when SQL Server executes a data modification operation, but first needs to perform a search to find the resource to modify. (Ex: row of data)

• SQL Server doesn't need to place an X lock on the row until it is ready to perform the modification, but it does need to apply some sort of lock as it searches, to protect that same data from modification by another transaction in the time between finding the data and modifying it.

• Therefore, SQL Server places a U lock on the row, checks the row and, if it meets the criteria, converts it to an X lock.
SQL server : U (Update) Lock

• SQL Server is searching for the data, it could acquire an S lock on each row it encountered and then determine whether it had found the data it needed.

• Problems : two transactions were both searching for the same row to modify (for example, the same customer row in the Customers table), using different access paths, and they could both reach the desired resource at the same time. Each transaction could acquire an S lock on that row, but then each transaction would attempt to convert this lock to an X lock in order to perform the modification, but the S lock held by one transaction prevents the other from doing so.

• At this point, we have a deadlock since neither transaction can proceed.
Types of deadlock

• Cycle deadlock
• Conversion deadlock
A cycle deadlock

- Transaction 1 starts, acquires an exclusive table lock on the Supplier table, and requests an exclusive table lock on the Part table.
- Simultaneously, Transaction 2 starts, acquires an exclusive lock on the Part table, and requests an exclusive lock on the Supplier table.
- The two transactions become deadlocked – caught in a "deadly embrace." Each transaction holds a resource needed by the other process. Neither can proceed and, without intervention, both would be stuck in deadlock forever.
Generating a cycle deadlock

USE AdventureWorks
-- On one connection, start Transaction 1:
BEGIN TRAN
UPDATE Purchasing.PurchaseOrderDetail
SET OrderQty = OrderQty + 200
WHERE ProductID = 922
     AND PurchaseOrderID = 499;
GO

-- Open a second connection, and start Transaction 2:
BEGIN TRAN
UPDATE Production.Product
SET ListPrice = ListPrice * 0.9
WHERE ProductID = 922;
GO

-- Go back to the first connection, and execute this update statement:
UPDATE Production.Product
SET ListPrice = ListPrice * 1.1
WHERE ProductID = 922;
GO

-- At this point, this first connection should block.
-- It is not deadlocked yet, however. It is waiting for a lock
-- on the Production.Product table, and there is no reason
-- to suspect that it won't eventually get that lock.
Generating a cycle deadlock

-- Now go back to the second connection,
-- and execute this update statement:
UPDATE Purchasing.PurchaseOrderDetail
SET OrderQty = OrderQty - 200
WHERE ProductID = 922
    AND PurchaseOrderID = 499;
GO
-- At this point a deadlock occurs.

One of the processes will receive the following error message. (Of course, the actual process ID reported will probably be different.)

Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
A conversion deadlock.

Problems:
Two transactions were both searching for the same row to modify (for example, the same customer row in the Customers table), using different access paths, and they could both reach the desired resource at the same time.

Each transaction could acquire an S lock on that row, but then each transaction would attempt to convert this lock to an X lock in order to perform the modification, but the S lock held by one transaction prevents the other from doing so.
SQL server : U (Update) Lock

• In order or avoid such deadlocks, if a transaction begins a search operation with the intention of eventually modifying data, then SQL Server acquires U locks until it finds the data to modify.

• U locks are compatible with S locks, but are incompatible with X locks or other U locks.

• So if two transactions were searching for the same resource, each with the intention of modifying it, then the first one to reach it would acquire a U lock, and then the second one would be blocked until the first was finished. Since the second transaction was blocked, the first is free to convert its U lock to an X lock, make the data modification and release its locks. Then the second transaction could make its change.

• In the sys.dm_tran_locks view, a request_mode of 'U' indicates an update lock.
Multiple Granularity Locking Scheme

- Transaction $T_i$ can lock a node $Q$, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node $Q$ can be locked by $T_i$ in S or IS mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or IS mode.
  4. A node $Q$ can be locked by $T_i$ in X, SIX, or IX mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or SIX mode.
  5. $T_i$ can lock a node only if it has not previously unlocked any node (that is, $T_i$ is two-phase).
  6. $T_i$ can unlock a node $Q$ only if none of the children of $Q$ are currently locked by $T_i$.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock.
Multiple granularity for locks

- Explicit Lock
- Implicit Lock – Lower in the hierarchy
Multiple granularity for locks

- $T_1$ lock $f_2$: Allowed
- $T_2$ write $a_{24}$: Not Allowed
- $T_3$ write $r_1$: Not Allowed
Intention lock mode

1. shared (S)
2. exclusive (X)
3. intention-shared (IS): at least one descendant has a S lock
Rules of Locking

one lock
meet restriction

↓ locks acquired root - leaf order

↑ locks released leaf - root order
Ex : Locks Modes

\[ T_1 \rightarrow \text{read} \ a_{12} \]

\[ T_2 \rightarrow \text{write} \ a_{14} \]

\[ T_3 \rightarrow \text{read} \ f_1 \]

\[ T_4 \rightarrow \text{read} \ d \]

\[ T_1, T_2 \rightarrow \text{concurrently} \]

\[ T_1, T_3, T_4 \rightarrow \text{concurrently} \]

\[ T_2, T_3 \rightarrow \text{not concurrently} \]
• Example: Escalate dynamically to coarse locks when memory usage for LCBs becomes critical
Lock conversion from S to X.

USE AdventureWorks2014;
-- Create a new table
IF OBJECTPROPERTY(OBJECT_ID('NewOrders'), 'IsUserTable') = 1
DROP TABLE NewOrders;
GO
SELECT *
INTO NewOrders
FROM Sales.SalesOrderHeader;
GO
CREATE UNIQUE INDEX NewOrder_index ON NewOrders(SalesOrderID);
GO
Lock conversion from S to X.

-- Change isolation level and start transaction
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN

-- SELECT data and examine the locks
SELECT *
FROM NewOrders
WHERE SalesOrderID = 55555;

SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'NewOrders';

-- UPDATE data and examine the locks
UPDATE NewOrders
SET SalesPersonID = 277
WHERE SalesOrderID = 55555;

Results | Messages
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SalesOrderID</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>55555</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>AdventureWorks2014</td>
<td>NewOrders</td>
<td>0</td>
<td>RID</td>
<td>1.24738:25</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>AdventureWorks2014</td>
<td>NewOrders</td>
<td>2</td>
<td>KEY</td>
<td>(fc81b91b4150)</td>
<td>S</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>AdventureWorks2014</td>
<td>NewOrders</td>
<td>0</td>
<td>PAGE</td>
<td>1.24738</td>
<td>IS</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>AdventureWorks2014</td>
<td>NewOrders</td>
<td>2</td>
<td>PAGE</td>
<td>1.26549</td>
<td>IS</td>
<td>GRANT</td>
<td></td>
</tr>
<tr>
<td>79</td>
<td>AdventureWorks2014</td>
<td>NewOrders</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td></td>
<td>IS</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Lock conversion from S to X.

- Open REPEATABLE READ transaction and selects one row from the NewOrders table.
- When querying our DBlocks view we see an S lock on a RID in the database. (SalesOrderID = 55555)
- S lock is held for a key in the non-clustered index.
- Also IS locks for the page in the table that contains the selected row, and for the page in the index that contains the key for the selected row as well as an IS lock on the table itself.

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>RevisionNumber</th>
<th>OrderDate</th>
<th>DueDate</th>
<th>ShipDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55555</td>
<td>2003-10-05</td>
<td>2003-10-17</td>
<td>2003-10-12 00:00:00.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>PAGE</td>
<td>1:70144</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>RID</td>
<td>1:70144:30</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>KEY</td>
<td>(0300b2...</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>PAGE</td>
<td>1:70886</td>
<td>IS</td>
<td>GRANT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>PAGE</td>
<td>1:70144</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>RID</td>
<td>1:70144:30</td>
<td>X</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>KEY</td>
<td>(0300b2...</td>
<td>U</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>PAGE</td>
<td>1:70886</td>
<td>IU</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Lock conversion from S to X.

- After we update the row, we query the DBlocks view again, and this time there are different locks on the same resources.
- RID now has an X lock, and the page in the table, and the table itself, both have IX locks.
- Key in the index has a U lock. SQL Server acquired the U lock while searching for the row to update, and this is necessary because, until the modification to the data row happens SQL Server doesn't know whether the modification will also require a change to the index.
- The page in the index containing the key has an IU lock.

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>RevisionNumber</th>
<th>OrderDate</th>
<th>DueDate</th>
<th>ShipDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>55555</td>
<td>2</td>
<td>2003-10-05</td>
<td>2003-10-17</td>
<td>2003-10-12 00:00:00.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>PAGE</td>
<td>1:70144</td>
<td>IS</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>RID</td>
<td>1:70144:30</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>KEY</td>
<td>(0300b2...)</td>
<td>S</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>PAGE</td>
<td>1:70886</td>
<td>IS</td>
<td>GRANT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spid</th>
<th>dbname</th>
<th>entity_name</th>
<th>index_id</th>
<th>resource</th>
<th>description</th>
<th>mode</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>NULL</td>
<td>OBJECT</td>
<td></td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>2</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>PAGE</td>
<td>1:70144</td>
<td>IX</td>
<td>GRANT</td>
</tr>
<tr>
<td>3</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>0</td>
<td>RID</td>
<td>1:70144:30</td>
<td>X</td>
<td>GRANT</td>
</tr>
<tr>
<td>4</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>KEY</td>
<td>(0300b2...)</td>
<td>U</td>
<td>GRANT</td>
</tr>
<tr>
<td>5</td>
<td>AdventureWorks</td>
<td>NewOrders</td>
<td>2</td>
<td>PAGE</td>
<td>1:70886</td>
<td>IU</td>
<td>GRANT</td>
</tr>
</tbody>
</table>
Chapter 10: Implementation and Pragmatic Issues

• 10.2 Data Structures of a Lock Manager
• 10.3 Multi-Granularity Locking and Lock Escalation

10.4 Transient Versioning

• 10.5 Nested Transactions for Intra-transaction parallelism
• 10.6 Tuning Options
• 10.7 Overload Control
• 10.8 Lessons Learned
Storage Organization for Transient Versioning

- update on current data moves old version to version pool
- read-only transactions follow version chains
- old versions are kept sorted by their successor timestamps → garbage collection simply advances begin pointer
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- **10.5 Nested Transactions for Intra-transaction parallelism**
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Multi-threaded Transactions

Example:

\( t_1: t_{11} \ t_{12} \ t_{13} \ t_{14} \) with \( t_{12} \) and \( t_{13} \) as parallel threads

\( t_{11}: \textcolor{red}{r(t)} \ \textcolor{red}{r(p)} \ \textcolor{red}{w(p)} \ /* \text{store new incoming e-mail} */\)

\( t_{12}: t_{121} \ t_{122} \ t_{123} \ t_{124} \) with \( t_{122}, \ t_{123}, \ t_{124} \) as parallel threads

\( t_{121}: \textcolor{red}{r(t)} \ \textcolor{red}{r(s)} \ \textcolor{red}{w(s)} \ /* \text{update folder by subject} */\)

\( t_{122}: \textcolor{green}{r(r)} \ \textcolor{green}{r(n)} \ \textcolor{green}{r(l)} \ \textcolor{green}{w(l)} \ /* \text{update text index for descriptor}_1 */\)

\( t_{123}: \textcolor{blue}{r(r)} \ \textcolor{blue}{r(n)} \ \textcolor{blue}{r(m)} \ \textcolor{blue}{w(m)} \ \textcolor{blue}{w(n)} \ /* \text{update text index for descriptor}_2 */\)

\( t_{124}: \textcolor{green}{r(r)} \ \textcolor{green}{r(n)} \ \textcolor{green}{r(l)} \ \textcolor{green}{w(l)} \ /* \text{update text index for descriptor}_3 */\)

\( t_{13}: \textcolor{blue}{r(t)} \ \textcolor{blue}{r(f)} \ \textcolor{blue}{w(f)} \ \textcolor{blue}{w(g)} \ \textcolor{blue}{w(t)} \ /* \text{update folder by sender} */\)

\( t_{14}: \textcolor{green}{r(t)} \ \textcolor{green}{r(p)} \ \textcolor{green}{w(p)} \ \textcolor{green}{r(g)} \ \textcolor{green}{w(g)} \ /* \text{assign priority} */\)
Locking for Nested Transactions

2PL protocol for nested transactions:
• Leaves of a transaction tree acquire locks as needed, based on 2PL for the duration of the transaction.
• Upon terminating a thread, all locks held by the thread are inherited by its parent.
• A lock request by a thread is granted if no conflicting lock on the same data item is currently held or the only conflicting locks are held by ancestors of the thread.

Theorem 10.1:
2PL for nested transactions generates only schedules that are equivalent to a serial execution of the transactions where each transaction executes all its sibling sets serially.
Layered Locking with Intra-transaction Parallelism

- search (CityIndex, “Austin”) fetch(x) modify(x)
- delete (CityIndex, "Austin", @x) insert (CityIndex, “Dallas”, @x)

Transaction Flow:
- Transaction t1: L0
  - t11: r(r) r(n) r(l) r(p) w(p)
  - t12: r(p) w(p)
  - t13: r(p) w(p)
  - t14: r(r)
- Transaction t2: L1
  - t2: r(r) r(n) r(l) w(l)
  - t21: r(r) r(n) r(l) w(l)
  - t22: r(p) w(p)
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Tuning Repertoire

• Manual locking (or manual preclaiming)
• Choice of SQL isolation level(s)
• Application structuring towards short transactions
• MPL control
Oracle Manual Locking and User-Defined Locks

- Manually lock data via a SQL statement.
  - SELECT...FOR UPDATE
  - LOCK TABLE statement

- Create our own locks via the DBMS_LOCK package.
To avoid the lost update issue whereby one session would overwrite another session’s changes.

A method to serialize access to detail records to enforce business rules.

```
SCOTT@ORA12CR1> select empno, ename, sal
2 from emp
3 where empno = :empno
4 and decode( ename, :ename, 1 ) = 1
5 and decode( sal, :sal, 1 ) = 1
6 for update nowait
7 /
EMPNO ENAME SAL
---------- ---------- ----------
7934 MILLER 130
SCOTT@ORA12CR1> update emp
2 set ename = :ename, sal = :sal
3 where empno = :empno;
1 row updated.
SCOTT@ORA12CR1> commit;
Commit complete.
```
LOCK TABLE statement

• Simply locks the table, not the rows in the table.

• If you start modifying the rows, they will be locked as normal. So, this is not a method to save on resources (as it might be in other RDBMSs).

• **Locking a Table: Example for a large batch update**
  - locks the employees table in exclusive mode but does not wait if another user already has locked the table:

  LOCK TABLE employees IN EXCLUSIVE MODE NOWAIT;
**DBMS_LOCK package**

- **Use this package to serialize access to some resource external to Oracle.**
  - Using the UTL_FILE routine that allows you to write to a file on the server’s file system.

- File is external, Oracle won’t coordinate the many users trying to modify it simultaneously.

- Before you open, write, and close the file, you will request a lock named after the file in exclusive mode, and after you close the file, you will manually release the lock.

- Only one person at a time will be able to write a message to this file. Everyone else will queue up.

- **DBMS_LOCK package allows you to manually release a lock when you are done with it, or to give it up automatically when you commit, or even to keep it as long as you are logged in.**
DBMS_LOCK package

• ALLOCATE_UNIQUE Procedure: Allocates a unique lock identifier
  
  Syntax

  ```
  DBMS_LOCK.ALLOCATE_UNIQUE ( lockname IN VARCHAR2, lockhandle OUT VARCHAR2, expiration_secs IN INTEGER DEFAULT 864000);
  ```

• CONVERT Function: converts a lock from one mode to another.

  Syntax

  ```
  DBMS_LOCK.CONVERT ( id IN INTEGER || lockhandle IN VARCHAR2, lockmode IN INTEGER, timeout IN NUMBER DEFAULT MAXWAIT) RETURN INTEGER;
  ```

• RELEASE Function: explicitly releases a lock previously acquired using the REQUEST function

  Syntax

  ```
  DBMS_LOCK.RELEASE ( id IN INTEGER) RETURN INTEGER;
  DBMS_LOCK.RELEASE ( lockhandle IN VARCHAR2) RETURN INTEGER;
  ```
Definition 10.1 (Isolation Levels):
• A schedule $s$ runs under isolation level **read uncommitted** (aka. dirty read or browse mode) if write locks are subject to S2PL.
• A schedule $s$ runs under isolation **read committed** (aka. cursor stability) if write locks are subject to S2PL and read locks are held for the duration of an SQL operation.
• A schedule $s$ runs under isolation level **serializability** if it can be generated by S2PL.
• A schedule $s$ runs under isolation level **repeatable read** if all anomalies other than phantoms are prevented.

Remark: A scheduler can use different isolation levels for different transactions.

Observation: **read committed is susceptible to lost updates**

Example: $r_1(x)\ r_2(x)\ w_2(x)\ c_2\ w_1(x)\ c_1$
### Which isolation levels permit which behaviors?

#### SQL Server

<table>
<thead>
<tr>
<th>Transaction Isolation Level</th>
<th>Behaviors Allowed</th>
<th>Concurrency Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dirty Read</td>
<td>Non-repeatable Read</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>(default for SQL Server)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>REPEETABLE READ</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SNAPSHOT</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
-- Create a database and table for testing the isolation levels
USE master
GO
IF EXISTS ( SELECT 1
FROM sys.databases
WHERE name = 'IsolationDB' )
DROP DATABASE IsolationDB ;
GO
CREATE DATABASE IsolationDB ;
GO
USE IsolationDB ;
GO
CREATE TABLE IsolationTest
(
  col1 INT PRIMARY KEY ,
  col2 VARCHAR(20)
) ;
GO
INSERT INTO IsolationTest
VALUES ( 10, 'The first row' ) ;
INSERT INTO IsolationTest
VALUES ( 20, 'The second row' ) ;
INSERT INTO IsolationTest
VALUES ( 30, 'The third row' ) ;
INSERT INTO IsolationTest
VALUES ( 40, 'The fourth row' ) ;
INSERT INTO IsolationTest
VALUES ( 50, 'The fifth row' ) ;
GO
MSSQL Isolation level : Read Uncommitted
Dirty read  example

\[\text{t1}\]
--- Step 1:
- Start a transaction but don't commit it
  USE IsolationDB;
  GO
  BEGIN TRAN
  UPDATE IsolationTest
  SET col2 = 'New Value';
  --<EXECUTE>

\[\text{t2}\]:
All the values in col2 are the string 'New Value', even though the transaction in the first connection has not yet committed.
MSSQL Isolation level: Read Uncommitted
Dirty read example

**t3**
Back in the first connection, rollback the transaction, as shown in Step 3

```
USE IsolationDB ;
GO
BEGIN TRAN
UPDATE IsolationTest
SET col2 = 'New Value' ;
EXECUTE
```

**t4**
For Step 4, rerun the SELECT statement in the second connection to see that all the values in col2 have reverted to their original values

```
USE IsolationDB ;
GO
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED ;
SELECT *
FROM IsolationTest ;
EXECUTE
```

```
SELECT *
FROM IsolationTest ;
EXECUTE
```
Dirty reads problems

• If a query running under READ UNCOMMITTED isolation level performs a scan of a table it is possible that a separate transaction could update a row of data, causing that row to move to a new location.

• If the scan started before the update and read the initial version of the row, the row might move to a page not yet read, and the query could end up reading it again, later on in the same scan.
MSSQL Isolation level : Read committed (Default)

**Blocking Example**

**t1**

Isolation level is the default READ COMMITTED, and then it will start a transaction that reads data from the IsolationTest table to compute an average.

```sql
-- Step 1:
-- Start a transaction but don't commit it
USE IsolationDB;
GO
BEGIN TRAN
UPDATE IsolationTest
SET col2 = 'New Value';
--<EXECUTE>
```

Notes:

- The SELECT statement blocks with READ COMMITTED isolation level.

**t2**

Step 2 will UPDATE the table. Assuming that the query in Step 1 has finished processing, the UPDATE will succeed, even though the first connection is still inside a transaction. Note that the UPDATE is an auto-commit transaction and so SQL Server will commit the UPDATE and release the locks as soon as it completes.

```sql
-- Step 2:
-- Start a new connection and change your isolation level
USE IsolationDB;
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT *
FROM IsolationTest;
--<EXECUTE>
```

Notes:

- You should notice that the process blocks, and returns
- No data or messages!

A SELECT statement blocking with READ COMMITTED isolation level.
MSSQL Isolation level: Read committed (Default)

Blocking Example

```sql
USE IsolationDB;
GO
BEGIN TRAN
UPDATE IsolationTest
SET col2 = 'New Value';

-- To finish up, perform the following two steps:
-- Step 3:
-- Return to the connection from Step 1 and issue a ROLLBACK
ROLLBACK TRANSACTION;

-- Step 2:
-- Start a new connection and change your isolation level
USE IsolationDB;
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM IsolationTest;

-- You should notice that the process blocks, and returns
-- no data or messages!

-- Step 4:
-- Rerun the SELECT statement in the connection from Step 2
SELECT * FROM IsolationTest;

-- Verify that the data is available
```
MSSQL Isolation level : Read committed (Default) non-repeateable reads example

**t1**

Make sure the isolation level is the default READ COMMITTED, and then it will start a transaction that reads data from the Isolation Test table to compute an average.

```sql
-- Step 1:
-- Read data in the default isolation level
USE IsolationDB
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRAN
SELECT AVG(col1)
FROM IsolationTest;
--<EXECUTE>|
```

**t2**

In the second connection, Step 2 will UPDATE the table. Assuming that the query in Step 1 has finished processing, the UPDATE will succeed, even though the first connection is still inside a transaction. Note that the UPDATE is an auto-commit transaction and so SQL Server will commit the UPDATE and release the locks as soon as it completes.

```sql
-- Step 2:
-- In a new connection, update the table:
USE IsolationDB;
UPDATE IsolationTest
SET col1 = 500
WHERE col1 = 50;
--<EXECUTE>|
```
In Step 3, return to the first connection and run the same SELECT statement. The average value is now different and we have a non-repeatable read. The default READ COMMITTED isolation level prevents other connections from reading data being modified, but only prevents other connections from changing data being read, while the read operation is in progress. Once it is complete, other transactions can change the data, even if the reading transaction is still open. As a result, there is no guarantee that we'll see the same data if we rerun the SELECT within the transaction.
READ COMMITTED and non repeatable reads

- READ COMMITTED isolation level prevents other connections from reading data being modified, but only prevents other connections from changing data being read, while the read operation is in progress.

- Once it is complete, other transactions can change the data, even if the reading transaction is still open.

- As a result, there is no guarantee that we'll see the same data if we rerun the SELECT within the transaction.
READ_COMMITTED_SNAPSHOT

- The same as the default in terms of the read phenomena, i.e. it prevents dirty reads

- Allows non-repeatable reads and phantom reads.

- Optimistic implementation of the READ COMMITTED level prevents dirty reads, *without blocking other* transactions.
t1

--- Step 1:
-- First close all other connections to make sure no one is using
-- the IsolationDB database
-- Step 2:
-- Change the database option to enable "read committed snapshot"
ALTER DATABASE [IsolationDB] SET READ_COMMITTED_SNAPSHOT ON ;

--<EXECUTE>

--- Step 3:
-- Start a transaction but don't commit it
USE IsolationDB ;
GO
BEGIN TRAN
UPDATE IsolationTest
SET col2 = 'New Value' ;
--<EXECUTE>

5/25/2016
READ_COMMITTED_SNAPSHOT

-- Step 4:
-- Start a new connection and change your isolation level
USE IsolationDB;
GO
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM IsolationTest;
--<EXECUTE>
-- You should notice that the second connection is not blocked, but
-- it does not return the changed data. The results you get are the
-- original committed data, before the UPDATE in Step 3 was performed
-- no data or messages!

-- Step 3:
-- Start a transaction but don't commit it
USE IsolationDB;
GO
BEGIN TRAN
UPDATE IsolationTest
SET col2 = 'New Value';
--<EXECUTE>
-- To finish up, perform the following steps:
-- Step 5:
-- Return to the connection from Step 1 and issue a ROLLBACK
ROLLBACK TRANSACTION;
--<EXECUTE>

-- Step 6:
-- Now close all other
connections to make sure no one
is using the IsolationDB
database:
5/25/2016
READ_COMMITTED_SNAPSHOT

-- Step 7:
-- Change the database option to disable "read committed snapshot"

ALTER DATABASE [IsolationDB] SET READ_COMMITTED_SNAPSHOT OFF ;

--<EXECUTE>

Command(s) completed successfully.
REPEATABLE READ

• REPEATABLE READ isolation level adds to the properties of READ COMMITTED by ensuring that if a transaction re-reads data, or if a query is reissued within the same transaction, then the same data will be returned.

• Issuing the same query twice within a transaction won't pick up any changes to data values that were made by another transaction.

• A second transaction cannot modify the data that a first transaction has read, as long as that first transaction has not yet committed or rolled back.
REPEATABLE READ

-- Step 1:
-- Read data in the Repeatable Read isolation level
USE IsolationDB;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN
SELECT AVG(col1)
FROM IsolationTest;
--<EXECUTE>

-- Step 2:
-- In the second connection, update the table:
USE IsolationDB;
UPDATE IsolationTest
SET col1 = 5000
WHERE col1 = 500;
--<EXECUTE>

-- You should notice that the UPDATE process blocks,
-- and returns no data or messages
**REPEATABLE READ**

--- Step 1:  
- Read data in the Repeatable Read isolation level

```sql
USE IsolationDB ;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
BEGIN TRAN
SELECT AVG(col1) 
FROM IsolationTest ;
--<EXECUTE>
```

--- Step 3:  
- Go back to the first connection and 
- run the same SELECT statement:

```sql
SELECT AVG(col1) 
FROM IsolationTest ;
--<EXECUTE>
```

--- Step 4:  
- issue a ROLLBACK

--- ROLLBACK TRANSACTION ;
```sql
--<EXECUTE>
```
REPEATABLE READ

• However, REPEATABLE READ isolation doesn't prevent all possible read phenomena.

• It protects only the data that has already been read.
REPEATABLE READ: phantoms

**t1**

-- Close all connections and open two new ones
-- Step 1:

```
USE IsolationDB;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40
```

**t2**

-- Step 2:
-- In the second connection, insert new data

```
USE IsolationDB;
INSERT INTO IsolationTest
VALUES (25, 'New Row');
```

REPEATABLE READ: phantoms

Upon the second execution of the same SELECT statement, the new row appears, called a phantom. The row didn't even exist the first time we ran the SELECT statement, so it wasn't locked. We can prevent phantoms with the SERIALIZABLE isolation level.

T4
Issue a ROLLBACK
SERIALIZABLE

- We pay a price to prevent phantoms. In addition to locking all the data has been read, enforcing the SERIALIZABLE isolation level, and so preventing phantoms, requires that SQL Server also lock data that doesn't exist (*Key-range Locks*).

- Running multiple SERIALIZABLE transactions at the same time is the equivalent of running them one at a time – that is, serially.
SERIALIZABLE

-- Open two new connections
-- Step 1:
-- In the first connection, start a transaction

USE IsolationDB;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40;
--<EXECUTE>

-- Step 2:
-- In the second connection, insert new data

USE IsolationDB
INSERT INTO IsolationTest
VALUES ( 35, 'Another New Row' );
-- Notice that the INSERT will block
--<EXECUTE>
-- In the first connection, start a transaction
USE IsolationDB ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
BEGIN TRAN
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40 ;
--<EXECUTE>

-- Step 3:  
-- Go back to the first connection and rerun the SELECT
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40 ;
--<EXECUTE>  
-- Notice no new rows
SERIALIZABLE

T4

Rollback

```sql
USE IsolationDB;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40;
--<EXECUTE>

-- Step 3:
-- Go back to the first connection and rerun the SELECT
SELECT *
FROM IsolationTest
WHERE col1 BETWEEN 20 AND 40;
--<EXECUTE>
-- Notice no new rows

-- Step 4:
-- issue a ROLLBACK
ROLLBACK TRANSACTION;
--<EXECUTE>
```

Command(s) completed successfully.

---

5/25/2016  Transactional Information Systems  10-109
Definition 10.2 (Multiversion Read Committed and Snapshot Isolation Levels):

- A transaction runs under isolation level multiversion read committed if it reads the most recent committed versions as of the transaction’s begin and uses S2PL for writes.
- A transaction runs under snapshot isolation if it reads the most recent versions as of the transaction’s begin and its write set is disjoint with the write sets of all concurrent transactions.

Observation: snapshot isolation does not guarantee MVSR

Example:

\[ r_1(x_0) \ r_1(y_0) \ r_2(x_0) \ r_2(y_0) \ w_1(x_1) \ c_1 \ w_2(y_2) \ c_2 \]

Possible interpretation:

constraint \( x + y \geq 0, x_0 = y_0 = 5, \)

\( t_1 \) subtracts 10 from \( x \), \( t_2 \) subtracts 10 from \( y \)
Advanced Locking Concepts

- Lock Compatibility
- Lock Mode Conversion
- Special Intent Locks
- Shared intent exclusive (SIX)
- Update intent exclusive (UIX)
- Shared intent update (SIU)
Advanced Locking Concepts

- **lock compatibility** – which lock types are compatible, and so can exist simultaneously on the same resource, and which are incompatible, and so will lead to blocking

- **lock mode conversion** – how SQL Server converts lock modes in response to the operations being performed by a given transaction, in order to ensure enforcement of the ACID transaction properties

- **special intent locks** – acquired when a non-intent lock is requested on a resource on which either an IX or an IU lock is already held

- **key-range locks** – a type of lock acquired in SERIALIZABLE isolation level when scanning or modifying a range of data;

- **We take a closer look at the four most common modes of key-range lock.**
<table>
<thead>
<tr>
<th>Requested lock mode</th>
<th>Existing granted lock mode</th>
<th>IS</th>
<th>S</th>
<th>U</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent shared (IS)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Update (U)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Intent exclusive (IX)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive (X)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
3 Schema Locks

1. Sch-S: schema stability lock
2. Sch-M: schema modification lock
3. BU: bulk update lock
Sch-M: schema modification lock

- SQL Server acquires a Sch-M lock when performing certain DDL operations that change a table's definition (its schema).
- Operations include adding and dropping columns from the table, or changing a column's data type.
- When a session is making a schema change to a table, no other sessions can do anything with the table.
- “Super-lock.” It is incompatible with any other lock types.
- Similar to exclusive (X) locks, schema modification (Sch-M) locks are held until the end of the transaction.
- You run DDL statements within explicit transactions. While that allows you to rollback all of the schema changes in case of an error, it also prevents any access to the affected objects until the transaction is committed.
Sch-S: schema stability lock

- Schema stability (Sch-S) locks are used during DML query compilation and execution.

- SQL Server acquires them regardless of the transaction isolation level, even in READ UNCOMMITTED mode.

- Schema stability (Sch-S) locks are compatible with any other than schema modification (Sch-M) lock types.

- It's perfectly fine for one session to be modifying data in a table while SQL Server is optimizing a query for another session that is accessing that table.
1. In the first session, we started the transaction and altered the table acquiring a schema modification (Sch-M) lock.
2. In the next step in the other sessions, we ran a SELECT statement in READ UNCOMMITTED transaction isolation level and a DELETE statement.
3. Both of these sessions were blocked while waiting for schema stability (Sch-S) locks that were required for query compilation. Figure illustrates this behavior.
Schema locks: Query execution when plans are in cache

run that example the second time, when queries are compiled and plans are in cache.
The second session would still wait for the schema stability (Sch-S) lock. There are no shared (S) locks in the
READ UNCOMMITTED transaction isolation level, and the schema stability (Sch-S) lock is the only way to keep a schema
stable during execution. However, the session with the DELETE statement would wait for an intent exclusive (IX) lock
instead. That lock type needs to be acquired anyway, and it can replace a schema stability (Sch-S) lock because it is
also incompatible with schema modification (Sch-M) locks and prevents schema from being altered
Deadlock due to mixed DDL and DML statements (Step 1)

Mixing schema modification locks with other lock types in the same transaction increases the possibility of the deadlocks. Let’s assume that we have two sessions: the first one starts the transaction, and it updates the row in the table. At this point, it holds exclusive (X) lock on the row and two intent exclusive (IX) locks on the page and table. If another session tries to read (or update) the same row, it would be blocked. At this point, it will wait for the shared (S) lock on the row and have the intent shared (IS) locks held on the page and the table. (Page-level intent locks are omitted.)
Deadlock due to mixed DDL and DML statements (Step 2)

- If at this point the first session wants to alter the table, it would need to acquire a schema modification (Sch-M) lock.
- That lock type is incompatible with any other lock types, and the session would be blocked by the intent shared (IS) lock held by the second session.
- We would then have a deadlock, as shown in Figure.
Multiple Sessions and Lock Compatibility – EX1

As you can see in Figure, the first session (SPID=55) holds the shared (S) lock on the row. The second session (SPID=54) is trying to acquire exclusive (X) lock on the same row, and it is being blocked due to lock incompatibility. The third session (SPID=53) is reading the same row in the READ COMMITTED transaction isolation level. This session has not been blocked.

Note third session ran in READ COMMITTED transaction isolation level and did not acquire the lock on the resource, can be considered an internal optimization.
As you can see in Figure, the third session did not even try to acquire the shared (S) lock on the row. There is already a shared (S) lock on the row held by the first session (SPID=55), which guarantees that the row has not been modified by uncommitted transactions. In a READ COMMITTED isolation level, a shared (S) lock releases immediately after a row is read. As a result, Session 3 (SPID=55) does not need to hold its own shared (S) lock after reading the row, and it can rely on the lock from Session 1.
Let’s change our example and see what happens if the third session tries to read the row in a REPEATABLE READ transaction isolation level where a shared (S) lock needs to be held until the end of the transaction.

As you can see in Figure, the third session is blocked. It needs to acquire and hold its own shared (S) lock on the row, and there is an incompatible exclusive (X) lock from the second session in the queue.
• SQL Server acquires a BU lock on a table only if a session explicitly requests one during a bulk insert operation into the table.

1. With the BULK INSERT command we can specify a BU lock, using the TABLOCK hint, and with the bcp utility we can use the –h "TABLOCK" option.

2. We can specify that SQL Server take BU locks, by default, for a particular table, during bulk updates

3. BU locks allow multiple threads to load data into the same table concurrently, and they are only compatible with other BU locks and with Sch-S locks.
Creation of the DBlocks view to display locks in the current database.

```sql
IF EXISTS ( SELECT 1
    FROM sys.views
    WHERE name = 'DBlocks' )
    DROP VIEW DBlocks ;
GO
CREATE VIEW DBlocks AS
SELECT request_session_id AS spid ,
    DB_NAME(resource_database_id) AS dbname ,
    CASE WHEN resource_type = 'OBJECT'
        THEN OBJECT_NAME(resource_associated_entity_id)
        WHEN resource_associated_entity_id = 0 THEN 'n/a'
        ELSE OBJECT_NAME(p.object_id)
    END AS entity_name ,
    index_id ,
    resource_type AS resource ,
    resource_description AS description ,
    request_mode AS mode ,
    request_status AS status
FROM sys.dm_tran_locks t
LEFT JOIN sys.partitions p
    ON p.partition_id = t.resource_associated_entity_id
WHERE resource_database_id = DB_ID()
AND resource_type <> 'DATABASE' ;
```
USE AdventureWorks;

-- Create a new table
IF OBJECTPROPERTY(OBJECT_ID('NewOrders'), 'IsUserTable') = 1
    DROP TABLE NewOrders;
GO
SELECT * INTO NewOrders FROM Sales.SalesOrderHeader;
GO
CREATE UNIQUE INDEX NewOrder_index ON NewOrders(SalesOrderID);
GO

-- Change isolation level and start transaction
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN

-- SELECT data and examine the locks
SELECT * FROM NewOrders WHERE SalesOrderID = 55555;

SELECT * FROM DBlocks WHERE spid = @@spid
    AND entity_name = 'NewOrders';

-- UPDATE data and examine the locks
UPDATE NewOrders SET SalesPersonID = 277
WHERE SalesOrderID = 55555;

SELECT * FROM DBlocks WHERE spid = @@spid
    AND entity_name = 'NewOrders';

ROLLBACK TRAN
Creation of the DBlocks view to display locks in the current database.

The code in Listing will drop the NewOrders table if it already exists, then re-create it and build a non-clustered index on the SalesOrderID column. It then sets the isolation level to REPEATABLE READ in order that SQL Server holds S locks until the end of the transaction rather than just the end of the current statement, as is the case in the default READ COMMITTED level.
The code then opens a REPEATABLE READ transaction and selects one row from the NewOrders table. When querying our DBlocks view, we see an S lock on a RID in the database. This is the RID for the row that was selected, with SalesOrderID = 55555. Also, note that an S lock is held for a key in the non-clustered index. There are also IS locks for the page in the table that contains the selected row, and for the page in the index that contains the key for the selected row as well as an IS lock on the table itself.
After we update the row, we query the DBlocks view again, and this time there are different locks on the same resources. The same RID now has an X lock, and the page in the table, and the table itself, both have IX locks. The key in the index has a U lock. SQL Server acquired the U lock while searching for the row to update, and this is necessary because, until the modification to the data row happens, SQL Server doesn't know whether the modification will also require a change to the index. The page in the index containing the key has an IU lock.
Lock conversion from S to X.

Locks acquired by a SELECT and then UPDATE in the same transaction.
Special Intent locks

- Shared intent exclusive (SIX).
- Update intent exclusive (UIX)
- Shared intent update (SIU)
Special Intent Locks

- SQL Server holds an S lock on a row, then it also holds IS locks on the page and the table containing that row.

- SQL Server acquires an IU lock on an index page, when the component (a key) of that index had a U lock.

- In addition to the IS, IX, and IU there are three more types of intent locks that can be considered conversion locks.
SIX Mode

- Compromise
- Both SI and X. Safer.
- Something inside is written
- When SQL Server has one or more rows locked with X locks, the pages and the table that contains the rows will acquire IX locks. When the same transaction performs an operation that requires an S lock, SQL Server will acquire a SIX lock on the table.
USE AdventureWorks2014;
--Step 1: Create a new table and set the isolation level
IF OBJECTPROPERTY(OBJECT_ID('NewOrders'), 'IsUserTable') = 1
DROP TABLE NewOrders;
GO
SELECT *
INTO NewOrders
FROM Sales.SalesOrderHeader;
GO
CREATE UNIQUE INDEX NewOrder_index ON NewOrders(SalesOrderID);
GO
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
Special Intent Locks: SIX MODE

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
-- Step 2: Generate an SIX lock
BEGIN TRAN
UPDATE dbo.NewOrders
SET ShipDate = ShipDate + 1
WHERE SalesOrderID = 55555;
GO
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'NewOrders';
GO
SELECT *
FROM dbo.NewOrders WITH (TABLOCK, REPEATAuckleread )
WHERE SalesOrderID = 55555;
GO
SELECT *
FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
GO
Special Intent Locks : SIX MODE

- starts with the UPDATE statement, which will acquire an X lock on updated row, and IX locks on the page and the table containing the row.
- Subsequent query against the NewOrders table, in the same transaction, will obtain an S lock on the table and hold it to the end of the transaction, thanks to the REPEATABLE READ hint.
- Query against the DBlocks view reveals that SQL Server has acquired an SIX lock on the table.
Key-Range Locks

• when considering lock resources and SERIALIZABLE isolation level.

• If the isolation level is SERIALIZABLE and a query scans a range of data within a transaction, SQL Server needs to lock enough of the table to ensure that another transaction cannot insert a new value into the range currently being scanned, because if we reissued the same query that value would then appear as a phantom.

• A key-range lock is associated with a specific index key, but includes the range of possible values less than or equal to the key with which the lock is associated, and greater than the previous key in the index leaf level.

• Another way to say it would be that a key-range lock spans the range between two keys, and includes the key at the end, but not the key at the beginning.

• For example, if an index leaf level included the sequential values "James" and "Jones," a key-range lock on "Jones" would lock out all key values greater than "James" and less than or equal to "Jones."
USE AdventureWorks;

-- Step 1: Create a new table and set the isolation level
IF OBJECTPROPERTY(OBJECT_ID('NewOrders'), 'IsUserTable') = 1
    DROP TABLE NewOrders;
GO
SELECT *
INTO NewOrders
FROM Sales.SalesOrderHeader;
GO
USE AdventureWorks;

-- Step 1: Create a new table and set the isolation level
IF OBJECTPROPERTY(OBJECT_ID('NewOrders'), 'IsUserTable') = 1
    DROP TABLE NewOrders;
GO

SELECT *
INTO NewOrders
FROM Sales.SalesOrderHeader;
GO

CREATE UNIQUE INDEX NewOrder_index ON NewOrders(SalesOrderID);
GO

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
GO
-- Step 2: Generate RangeS-S locks
BEGIN TRAN
SELECT *
FROM dbo.NewOrders
WHERE SalesOrderID BETWEEN 55555 AND 55557;
GO
SELECT *
FROM DBlocks
WHERE spid = @@spid
   AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
GO
Generate RangeS-S locks

- Running as a SERIALIZABLE transaction, which requests a range of orders from the NewOrders table, based on SalesOrderID

- Results from the DBlocks view shows that SQL Server acquired four RangeS-S locks on KEY resources and that the SELECT statement returns three rows. It is normal to see one more key-range lock than the number of rows affected, because the ranges are open at the lower-valued end..

```
-- Step 2: Generate RangeS-S locks
BEGIN TRAN
SELECT * 
FROM    dbo.NewOrders
WHERE   SalesOrderID BETWEEN 55555 AND 55557;
GO
SELECT * 
FROM    DBlocks
WHERE   spid = @@spid 
        AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
GO
```
Generate RangeS-S locks

- The four key-range locks, indicated by the four RangeS-S locks in DBlocks view, cover:

1. Range starting just after the key 55554 up to and including the key 55555
2. Range starting just after the key 55555 up to and including the key 55556
3. Range starting just after the key 55556 up to and including the key 55557
4. Range starting just after the key 55557 up to and including the key 55558.
Generate RangeS-S locks

The four key-range locks, indicated by the four RangeS-S locks in DBlocks view, cover:

1. Range starting just after the key 55554 up to and including the key 55555

2. Range starting just after the key 55555 up to and including the key 55556

3. Range starting just after the key 55556 up to and including the key 55557

4. Range starting just after the key 55557 up to and including the key 55558.
Generate RangeS-S locks

• SQL Server will try to store any newly-inserted values, and remember that a range lock prevents SQL Server from inserting a new row into the locked range.

• Since the SalesOrderID column has an index, the rows will be stored in order of the SalesOrderID.

• Key-range locks will have to include the range from the key just prior to the first one selected up to and including the first key, so that the first key itself cannot be modified.

• Key-range locks will also have to include a range starting just after the highest-valued key selected up to the next key in the index, so that no values equal to the highest key selected can be inserted.
Generate RangeS-U locks

BEGIN TRAN
UPDATE   dbo.NewOrders
SET       ShipDate = ShipDate + 1
WHERE     SalesOrderID BETWEEN 55555 AND 55557;
GO
SELECT   *
FROM      DBlocks
WHERE     spid = @@spid
          AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
Generate RangeS-U locks

- If a non-clustered index is used to locate and update rows in a heap, while in SERIALIZABLE isolation level, and if the column being updated is not the indexed column used for access, the SQL Server will acquire a lock of Type RangeS-U.
- This means that there is an S lock on the range between the index keys, but the index key itself has a U lock.
- The rows in the heap will have the expected X lock on the RID.
Generate RangeS-U locks

- If a non-clustered index is used to locate and update rows in a heap, while in SERIALIZABLE isolation level, and if the column being updated is not the indexed column used for access, the SQL Server will acquire a lock of Type RangeS-U.

- This means that there is an S lock on the range between the index keys, but the index key itself has a U lock.

- The rows in the heap will have the expected X lock on the RID.

```sql
-- Step 3: Generate RangeS-U locks
BEGIN TRAN
UPDATE dbo.NewOrders
SET ShipDate = ShipDate + 1
WHERE SalesOrderID BETWEEN 55555 AND 55557;
GO
SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
```
--- STEP 4: Generate RangeX-X locks
--- We need a clustered index to see these locks

CREATE UNIQUE CLUSTERED INDEX NewOrder_index ON NewOrders(SalesOrderID) WITH DROP_EXISTING;

GO

BEGIN TRAN

UPDATE dbo.NewOrders
SET ShipDate = ShipDate + 1
WHERE SalesOrderID BETWEEN 55555 AND 55557;

GO

SELECT * FROM DBlocks
WHERE spid = @@spid
AND entity_name = 'NewOrders';

GO

ROLLBACK TRAN
Generate Range X-X locks

- If updating rows in an index while in SERIALIZABLE isolation level, the session will acquire exclusive key-range locks.

- Starts by converting the non-clustered index on SalesOrderID to a clustered index, and then updates the same range of rows as previous.

- In order to observe RangeX-X locks, the updated rows must be index keys, which is true when the table has a clustered index, and would also occur when updating one of the key columns of a non-clustered index.
Generate Range X-X locks

- If updating rows in an index while in SERIALIZABLE isolation level, the session will acquire exclusive key-range locks.
- Starts by converting the non-clustered index on SalesOrderID to a clustered index, and then updates the same range of rows as previous.
- In order to observe RangeX-X locks, the updated rows must be index keys, which is true when the table has a clustered index, and would also occur when updating one of the key columns of a non-clustered index.

```sql
-- STEP 4: Generate RangeX-X locks
-- We need a clustered index to see these locks
CREATE UNIQUE CLUSTERED INDEX NewOrder_index ON NewOrders(SalesOrderID)
WITH DROP_EXISTING;
GO
BEGIN TRAN
UPDATE dbo.NewOrders
SET ShipDate = ShipDate + 1
WHERE SalesOrderID BETWEEN 55555 AND 55557;
GO
SELECT *
FROM DBlocks
WHERE spid = @@spid
  AND entity_name = 'NewOrders';
GO
ROLLBACK TRAN
```
Rangel-N  (insert key-range and no resource lock)

We can observe this behavior by running, on two separate connections.

In the original connection, we first DELETE a row in the range we will be scanning so that there is room for an INSERT.

We then begin a transaction and select from a range of rows, but without committing or rolling back the transaction.
Opening a new connection, we attempt to INSERT a new row with same key as the row that we just deleted.

This insert blocks because the transaction on the first connection is still open and has the range locked. Return to the original connection, and run the query against DBlocks view, to reveal that the second connection has a lock request in a WAIT state with the RangeI-N mode.
RangeI-N (insert key-range and no resource lock)

- This kind of lock indicates an exclusive lock to prevent inserts on the range between keys and no lock on the keys themselves.
- The lock on the range is a special type, I, which only occurs as part of a key-range lock, and since there is no existing resource to lock, the second part of the name is N (for Null).

- SQL Server acquires RangeI-N locks when it attempts to insert values into the range between keys in SERIALIZABLE isolation level. We don't often see this type of lock because it is typically transient, held only until the correct location for insertion is found, and then escalated into an X lock.

- However, if one transaction scans a range of data using the SERIALIZABLE isolation level and then another transaction tries to INSERT into that range, the second transaction will have a lock request in a WAIT state, with the RangeI-N mode.
# Conversion key-range locks

<table>
<thead>
<tr>
<th>Lock 1</th>
<th>Lock 2</th>
<th>Conversion Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Rangel-N</td>
<td>Rangel-S</td>
</tr>
<tr>
<td>U</td>
<td>Rangel-N</td>
<td>Rangel-U</td>
</tr>
<tr>
<td>X</td>
<td>Rangel-N</td>
<td>Rangel-X</td>
</tr>
<tr>
<td>Rangel-N</td>
<td>RangeS-S</td>
<td>RangeX-S</td>
</tr>
<tr>
<td>Rangel-N</td>
<td>RangeS-U</td>
<td>RangeX-U</td>
</tr>
</tbody>
</table>
Lock Escalation : SQL Server

- Escalation based on SQL Server instance resource usage
- Escalation based on number of locks held by a single statement
- *Controlling Locking*, we can force SQL Server to change the default granularity of its locks with hints or index options.
Lock Escalation: SQL Server

- By default, SQL Server will acquire the finest-grain lock possible, in order to attain the greatest concurrency.

- In most cases, this means SQL Server will acquire row (RID or KEY) locks.

- SQL Server can acquire hundreds or thousands of individual locks on data in a single table without causing any problems.

- In some cases, however, if SQL Server determines that a query will access a range of rows within a clustered index, it may instead acquire page locks. After all, if every row on a page is going to be accessed, it's easier to manage a single page lock than dozens, or hundreds, of row locks.

- In other cases, primarily when there is no usable index to help process a query, SQL Server may lock an entire table right at the beginning of processing a query.
Escalation based on SQL Server instance resource Usage

• In some cases, acquiring individual locks on rows may end up consuming too much of SQL Server's memory.

• Although the memory required for each lock is quite small (about 96 bytes per lock), this still adds up to a sizeable portion of the total available memory, when thousands of locks are acquired.

• When SQL Server ends up using more than 24% of its buffer pool (excluding AWE memory) to keep track of locks acquired and lock requests waiting, it will choose any session holding locks and escalate its fine-grained (row or page) locks into a table lock.
Escalation based on SQL Server instance resource Usage

• Alternatively, we can specify that we want server-wide lock escalation to be triggered based on the total number of locks held by all sessions on the instance.

• If we change the value of the LOCKS configuration option to something other than the default value of zero, SQL Server will start choosing sessions to have their locks escalated as soon as it has acquired 40% of that configured total number of locks.

• For example, if we configure LOCKS to be 10,000, then escalation will start as soon as there are 4,000 locks held or requested
Escalation based on SQL Server instance resource Usage

• When the instance-wide escalation is triggered by crossing the memory threshold or by acquiring too many locks, we have no control over which sessions will have their locks escalated to table locks, and should just consider it a random selection.

• In addition, as long as the memory use remains over the instance-wide threshold, for every 1,250 new locks, SQL Server will again start escalating fine-grained locks into table locks.
Lock Escalation: Microsoft SQL Server
Lock Hierarchy

• Always have a Shared Lock (S) on DB level.

• When your query is connected to a DB (USE MyDatabase), Shared Lock prevents the dropping of the DB, or that backups are restored over that database.

• You have locks on the table, on the pages, and the records when you are performing an operation.
SQL Server Lock Escalation

- In DML: Intent Exclusive or Update Lock (IX or IU) on the table and page level, and a Exclusive or Update Lock (X or U) on the changed records.

- SQL Server always acquires locks from top to bottom to prevent Race Conditions, when multiple threads trying to acquire locks concurrently within the locking hierarchy.
SQL Server Lock Escalation

• DELETE operation on a table against 20,000 rows.

• Let’s assume that a row is 400 bytes long, means that 20 records fit onto one page of 8kb:
SQL Server Lock Escalation

- One S Lock on the database, 1 IX Lock on the table, 1.000 IX locks on the pages (20.000 records are spread across 1.000 pages), and you have finally 20.000 X locks on the records itself.

- In sum you have acquired 21.002 locks for the DELETE operation.
- Every lock needs in SQL Server 96 bytes of memory, so we look at 1.9 MB of locks just for 1 simple query.

- This will not scale indefinitely when you run multiple queries in parallel.

- For that reason SQL Server implements now the so-called Lock Escalation:
SQL Server Lock Escalation

- For more than 5,000 locks on one level in your locking hierarchy, SQL Server escalates into a simple coarse-granularity lock.
- SQL Server will by default *always* escalate directly to the table level.
- An escalation policy to the page level just doesn’t exist.
- One Exclusive Lock (X) on the table level. Concurrency of your database in a very negative way.
- No other session is able any more to access that table – every other query will just block.

![Diagram](image.png)
Escalation based on number of locks held by a single Statement

- SQL Server will also escalate locks when any individual session acquires more than 5,000 locks in a single statement.
- In this case, there is no randomness in choosing which session will get its locks escalated; it is the session that acquired the locks.
Escalation based on number of locks held by a single Statement

- First, we start a transaction and perform two UPDATE statements.
- Together, the two statements acquire more than 5,000 locks, but neither one, individually, acquires that many. Lock escalation does not occur.
- DBlocks query should reveal that the number of X locks held by the connection is 6,342.
Escalation based on number of locks held by a single Statement

- In the second transaction, we update the same 6,342 rows in a single statement. In this case, the DBlocks query reveals that the total number of locks held, right before the end of the transaction, is only one (a table lock). DBlocks query should reveal that the number of X locks held by the connection is 6,342.

```sql
USE AdventureWorks2014;
-- reset the isolation level
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO
BEGIN TRAN
-- Now show that if the same total number of locks are acquired in a single statement, we will get escalation and the sys.dm_tran_locks query will only show 1 lock
UPDATE Sales.SalesOrderHeader
SET DueDate = DueDate + 1
WHERE SalesOrderID <= 50000;
SELECT *
FROM DBlocks
WHERE mode = 'X'
AND spid = @@spid;
-- 1 lock
ROLLBACK TRAN;
GO
```