Seminar 4

Multiversioning

Monitoring locks

- *SQL Server Profiler*
- *sp_lock, sys.dm_tran_locks*
- *sys.dm_tran_active_transactions*
  - Resource types:
    - RID – row identifier
    - Key – range of keys in an index (key-range locks)
    - Page – 8 KB page from tables/indexes
    - HoBT – Heap or balanced tree
    - Table, File, Database
    - Metadata
    - Application

Query Governor and DBCC LOG

- SET QUERY_GOVERNOR_COST_LIMIT: query optimizer estimates the number of seconds needed to run a query. If it is bigger than setting, the query will not run (0 value allows all queries to run)
- DBCC LOG – returns information about transactions log

DBCC LOG (<databasename>,<output id>)

  - Output id: 0-4 specifies level of detail

Isolation Levels in SQL Server

- **`READ UNCOMMITTED`**: no lock when reading;
- **`READ COMMITTED`**: holds locks during statement execution (default) (*Dirty reads*);
- **`REPEATABLE READ`**: holds locks for the duration of transaction (*Unrepeatable reads*);
- **`SERIALIZABLE`**: holds locks and key range locks during entire transaction (*Phantom reads*);
- **`SNAPSHOT`**: work on data snapshot
- SQL syntax:

```
SET TRANSACTION ISOLATION LEVEL …
```

Multiversioning

- In a multiversion DBMS, each write on a data item **x** produces a new copy (or version) of **x**.
- For each read on **x**, the DBMS selects one of the versions of **x** to be read.
- Since writes do not overwrite each other, and since reads can read any version, the DBMS has more flexibility in controlling the order of reads & writes.

Row-Level Versioning (RLV)

- Introduced in SQL Server 2005
- Useful when *committed* data is needed, but not mandatory *the most recent version*
  - *Read Committed Snapshot Isolation* & *Full Snapshot Isolation* – the reader never blocks. It gets the previously committed value
- All older versions are stored in *tempdb* database
  - A snapshot of database could be constructed by old versions

Read Committed Snapshot Isolation

```
ALTER DATABASE MyDatabase
SET READ_COMMITTED_SNAPSHOT ON
```

- All operations see committed records when SQL command execution started =>
  - snapshot of data at command level
  - consistent read at command level
  - Available if READ COMMITED isolation level is used (default)

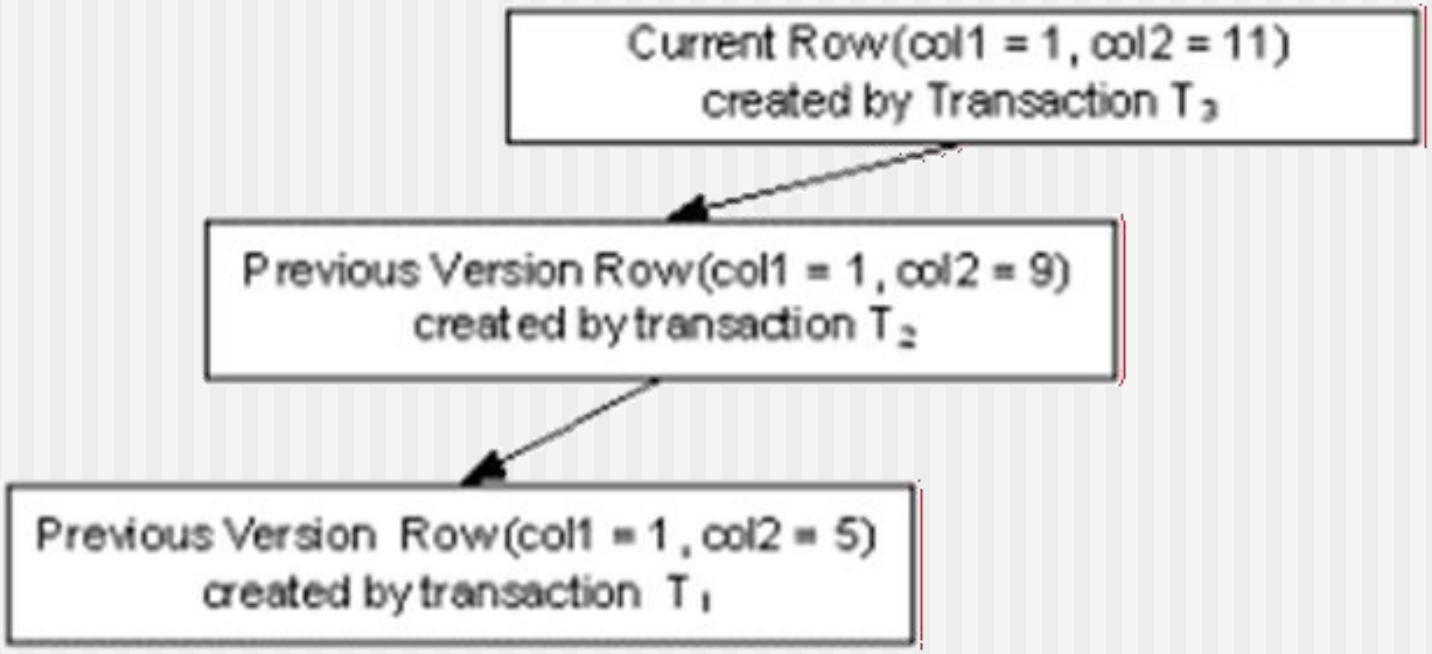
Full Snapshot Isolation

```
ALTER DATABASE MyDatabase
SET ALLOW_SNAPSHOT_ISOLATION ON
```

- All operations see committed records when transaction execution started =>
  - snapshot of data at transaction level
  - consistent read at transaction level
  - SNAPHOT isolation level could be used

Row-Level Versioning

- One record contains TSN (transaction sequence number)
- All versions are stored in a linked list


Current Row(col1 = 1, col2 = 11) created by Transaction T3
Previous Version Row(col1 = 1, col2 = 9) created by transaction T2
Previous Version Row(col1 = 1, col2 = 5) created by transaction T1

Row-Level Versioning (cont.)

- Advantages
  - The level of concurrency is increased
  - Affects performance of triggers/index creation
- Drawbacks
  - Extra management requirements to monitor the usage of *tempdb*
  - Slower performance of update operations
  - Readers speed is affected by the cost of browsing the linked lists
  - Solves the conflict between writer and readers but simultaneous writers are still not allowed

Triggers & RLV

- Triggers have access to 2 *pseudo-tables*:
  - ‘*deleted*’ table – contains deleted rows or old versions of updated rows
  - ‘*inserted*’ table – contains inserted rows or new versions of updated rows
- Before SQL Server 2005:
  - ‘*deleted*’ table created based on transaction logs – affects performance
- Using RLV:
  - For tables with relevant triggers the changes are versioned

Index Creation & RLV

- In previous versions of MS SQL Server: index creation or rebuilding meant
  - Table exclusively locked, data completely inaccessible (*clustered indexing*)
  - Table shared for reading only. Index not available (*non-clustered indexing*)
- With Row-Level Versioning:
  - Indexes are created and rebuilt online
  - All requests will be processed on versioned data

Isolation Levels and Concurrency Anomalies

Isolation Level	**Dirty Read**	**Non-repeat. Reads**	**Phantom read**	**Update Conflict**	**Conc. Model**
Read Un-committed	Yes	Yes	Yes	No	Pessimistic
Read committed Locking	No	Yes	Yes	No	Pessimistic
Read committed Snapshot	No	Yes	Yes	No	Optimistic
Repeatable read	No	No	Yes	No	Pessimistic
Row-Level Versioning	No	No	No	Yes	Optimistic
Serializable	No	No	No	No	Pessimistic

OUTPUT clause

- provides access to *inserted, updated* or *deleted* records
- can implement certain functionalities performed only through triggers

```
UPDATE Categories
SET CategoryName = ‘Dried Produce’
OUTPUT inserted.CategoryID,
       deleted.CategoryName,
       inserted.CategoryName, getdate(),
       SUSER_SNAME()
INTO CategoryChanges
WHERE CategoryID = 7
```

MERGE statement

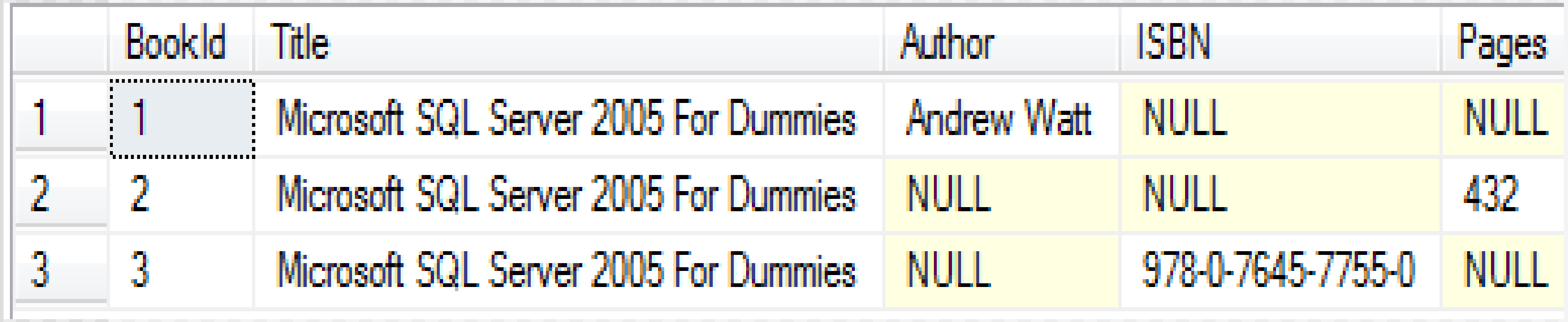
- MERGE – gives the ability to compare rows in a source and a destination table. INSERT, UPDATE or DELETE commands could be performed based on the result of this comparison

MERGE – General syntax

```
Merge Table definition as Target
Using ( Table Source ) as Source
(
Column Keys
)
ON (
Search Terms
)
WHEN MATCHED THEN
      UPDATE SET
        or
      DELETE
WHEN NOT MATCHED BY TARGET/SOURCE THEN
INSERT
```

MERGE sample

Books table



	BookId	Title	Author	ISBN	Pages
1	1	Microsoft SQL Server 2005 For Dummies	Andrew Watt	NULL	NULL
2	2	Microsoft SQL Server 2005 For Dummies	NULL	NULL	432
3	3	Microsoft SQL Server 2005 For Dummies	NULL	978-0-7645-7755-0	NULL

MERGE sample

```
MERGE Books
USING
 ( SELECT MAX(BookId) BookId, Title,MAX(Author)
     Author, MAX(ISBN) ISBN, MAX(Pages) Pages
 FROM Books
 GROUP BY Title
 ) MergeData ON Books.BookId = MergeData.BookId
 WHEN MATCHED THEN
  UPDATE SET Books.Title = MergeData.Title,
   Books.Author = MergeData.Author,
   Books.ISBN = MergeData.ISBN,
   Books.Pages = MergeData.Pages
 WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

PIVOT / UNPIVOT

- change a table-valued expression into another table.
- PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output, and performs aggregations where they are required on any remaining column values that are wanted in the final output.
- UNPIVOT performs the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.

PIVOT

```
SELECT <non-pivoted column>,
    [first pivoted column] AS <column name>,
    [second pivoted column] AS <column name>,
    ...
    [last pivoted column] AS <column name>
FROM
    (<SELECT query that produces the data>) AS <source query>
PIVOT
(
    <aggregation function>(<column being aggregated>)
FOR
[<column that contains values that become column headers>]
    IN ( [first pivoted column], [second pivoted column],
    ... [last pivoted column])
) AS <alias for the pivot table>
<optional ORDER BY clause>;
```