Seminar 4

Multiversioning

Monitoring Locks

- SQL Server Profiler
- sp_lock
- sys.dm_tran_locks
- sys.dm_tran_active_transactions

Resource Types

- RID – row identifier (lock one row in a heap)
- Key – row lock within an index (protect range of keys in an index - key-range locks)
- Page – 8 KB page (data / index page)
- HoBT – heap or B-tree
- Table
- File
- Database
- Metadata
- Application

Query Governor and DBCC LOG

- SET QUERY_GOVERNOR_COST_LIMIT *value*
  - *value*
    - the longest time in which a query can run
  - queries with an estimated cost greater than *value* are not allowed to run
  - value of 0 – all queries are allowed to run

- DBCC LOG – transaction log
  - DBCC LOG (<DBname>,<Output>)
  - Output - level of detail (0-4)

Isolation Levels in SQL Server

- **`READ UNCOMMITTED`**
  - no locks when reading data
- **`READ COMMITTED`**
  - holds S locks during the execution of the statement (default) (prevents *dirty reads*)
- **`REPEATABLE READ`**
  - holds S locks for the duration of the transaction (prevents *unrepeatable reads*)
- **`SERIALIZABLE`**
  - holds locks (including key-range locks) during the entire transaction (prevents *phantom reads*)

Isolation Levels in SQL Server

- **`SNAPSHOT`**
  - working on a snapshot of the data
- SQL syntax
  - **`SET TRANSACTION ISOLATION LEVEL`**

    …

Multiversioning

- in a DBMS with multiversioning, every write operation on a data object *O* results in a new copy (i.e., *version*) of *O*
- every time object *O* is read, the DBMS picks one of *O*'s versions
- writes do not overwrite each other, and read operations can read any version => the DBMS has more flexibility when controlling the order of read & write operations

Row-Level Versioning (RLV)

- introduced in SQL Server 2005
- useful when the user needs *committed* data, but not necessarily *the most recent version* of the data
- *Read Committed Snapshot Isolation* & *Full Snapshot Isolation*
  - the reader never blocks; instead, it obtains data that has been previously committed
- the *tempdb* database stores all the older versions of the data
  - a snapshot of the database can be assembled using these old(er) versions

Read Committed Snapshot Isolation

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

- operations see the most recent committed data as of the beginning of their execution =>
  - snapshot of the data at the command level
  - consistent reads at the command level
  - READ COMMITTED isolation level

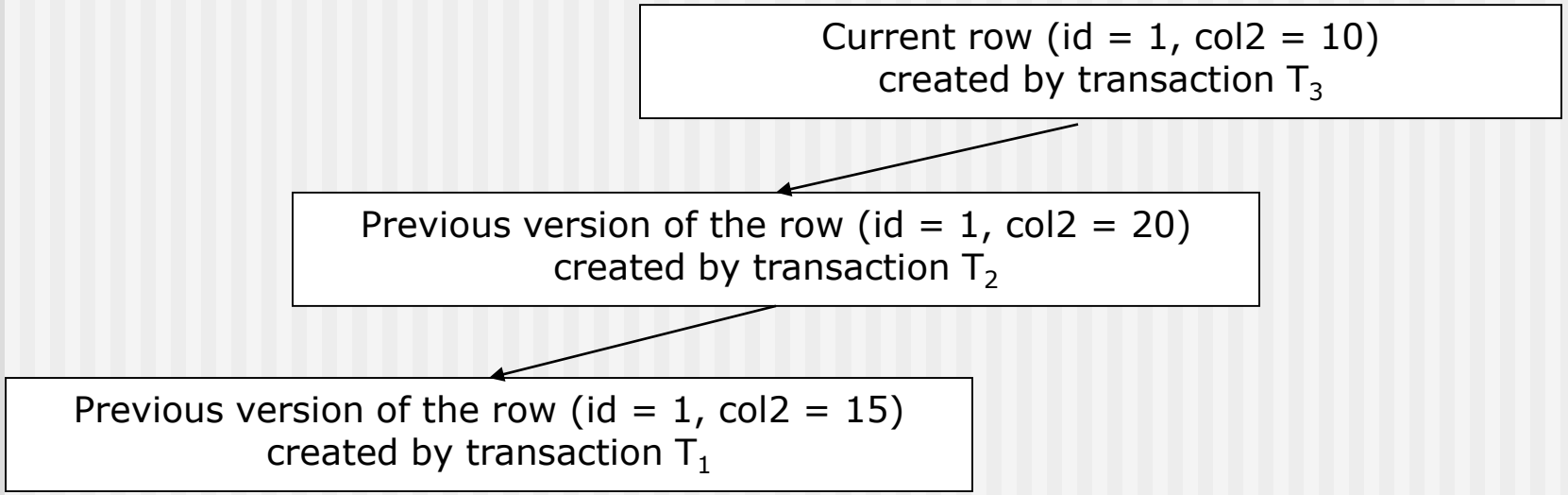
Full Snapshot Isolation

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

- operations see the most recent committed data as of the beginning of their transaction =>
  - snapshot of the data at the transaction level
  - consistent reads at the transaction level
  - SNAPHOT isolation level

Row-Level Versioning

- a record contains a *transaction sequence number* (TSN)
- all the versions are kept in a linked list



Row-Level Versioning

- advantages
  - increased concurrency level
  - positive impact on the creation of triggers / indexes
- drawbacks
  - monitoring the usage of the *tempdb* database => extra management requirements
  - update operations – slower
  - read operations – also affected (browsing the linked lists)

Row-Level Versioning

- drawbacks
  - solves the writer-reader conflict, but simultaneous writers are still not allowed

Triggers & RLV

- triggers can access 2 *pseudo-tables:*
  - the *deleted* table – contains removed rows or previous versions of updated rows
  - the *inserted* table – contains added rows or new versions of updated rows
- before SQL Server 2005:
  - the *deleted* table was created using the transaction log – affected performance
- using RLV:
  - changes are versioned for tables with relevant triggers

Index Creation & RLV

- index creation / rebuilding in previous versions of SQL Server:
  - *clustered index* - table exclusively locked, data entirely inaccessible
  - *non-clustered index* – table can only be read, index not available
- using RLV:
  - indexes are created and rebuilt online
  - all requests are processed on versioned data

Isolation Levels and Concurrency Anomalies

concurrency problem / isolation level	read uncommitted	read committed locking	read committed snapshot	repeatable read	row-level versioning	serializable
dirty reads	yes	no	no	no	no	no
unrepeatable reads	yes	yes	yes	no	no	no
phantoms	yes	yes	yes	yes	no	no
update conflicts	no	no	no	no	yes	no
concurrency model	pessimistic	pessimistic	optimistic	pessimistic	optimistic	pessimistic

PIVOT / UNPIVOT

- change a table-valued expression into another table
- PIVOT rotates a table-valued expression; it transforms the unique values in one column in the expression into multiple columns in the output; aggregations are performed where necessary on any remaining column values that are required in the output
- UNPIVOT performs the opposite operation; it rotates columns in 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 alias>
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>;
```

Recap - The OUTPUT Clause

- provides access to *inserted, updated, deleted* records
- can implement certain functionalities which can otherwise be performed only via triggers

```
UPDATE Courses
SET cname = 'Database Management Systems'
OUTPUT inserted.cid, deleted.cname, inserted.cname,
  GETDATE(), SUSER_SNAME()
INTO CourseChanges
WHERE cid = 'DB2'
```

Recap - The MERGE Statement

- a source table is compared with a target table; INSERT, UPDATE, DELETE statements can be executed based on the result of the comparison, i.e., INSERT / UPDATE / DELETE operations can be executed on the target table based on the result of a join with the source table

Recap - MERGE – General Syntax

```
MERGE TargetTable AS Target
USING SourceTable AS Source
ON (Search terms)
WHEN MATCHED THEN
  UPDATE SET
    or
  DELETE
WHEN NOT MATCHED [BY TARGET] THEN
  INSERT
WHEN NOT MATCHED BY SOURCE THEN
  UPDATE SET
    or
  DELETE
```

Recap - MERGE example

Books table

	BookID	Title	Author	ISBN	Pages
1	1	In Search of Lost Time	Marcel Proust	NULL	NULL
2	2	In Search of Lost Time	NULL	NULL	350
3	3	In Search of Lost Time	NULL	9789731246420	NULL

Recap - MERGE example

```
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;
```

Recap - MERGE example

	BookID	Title	Author	ISBN	Pages
1	3	In Search of Lost Time	Marcel Proust	9789731246420	350