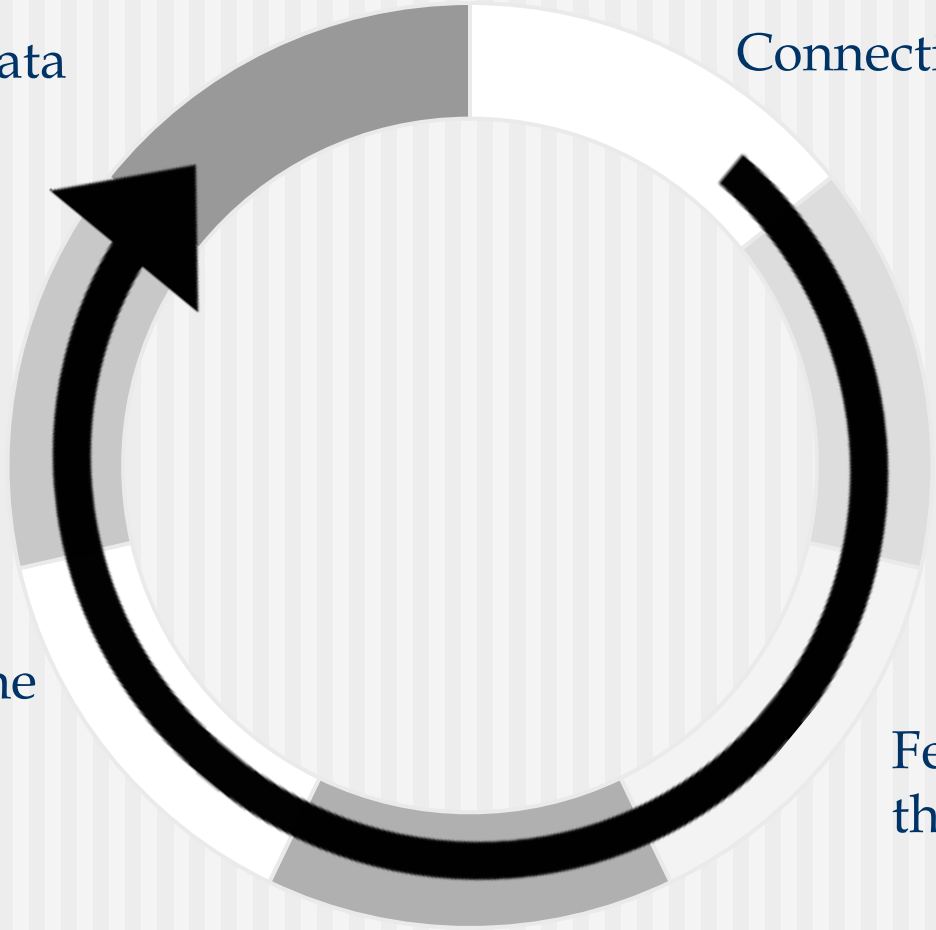
Seminar 1

ADO.NET

The Data Cycle



The Data Cycle

- connecting to data: establish a two-way communication between the application and the database server (e.g., *TableAdapter*)
- preparing the app to receive data: when using a disconnected data model there are specific objects that temporarily store data (e.g., *datasets, LINQ to SQL* objects)
- fetching data: execute queries and stored procedures (e.g., by using *TableAdapters*)

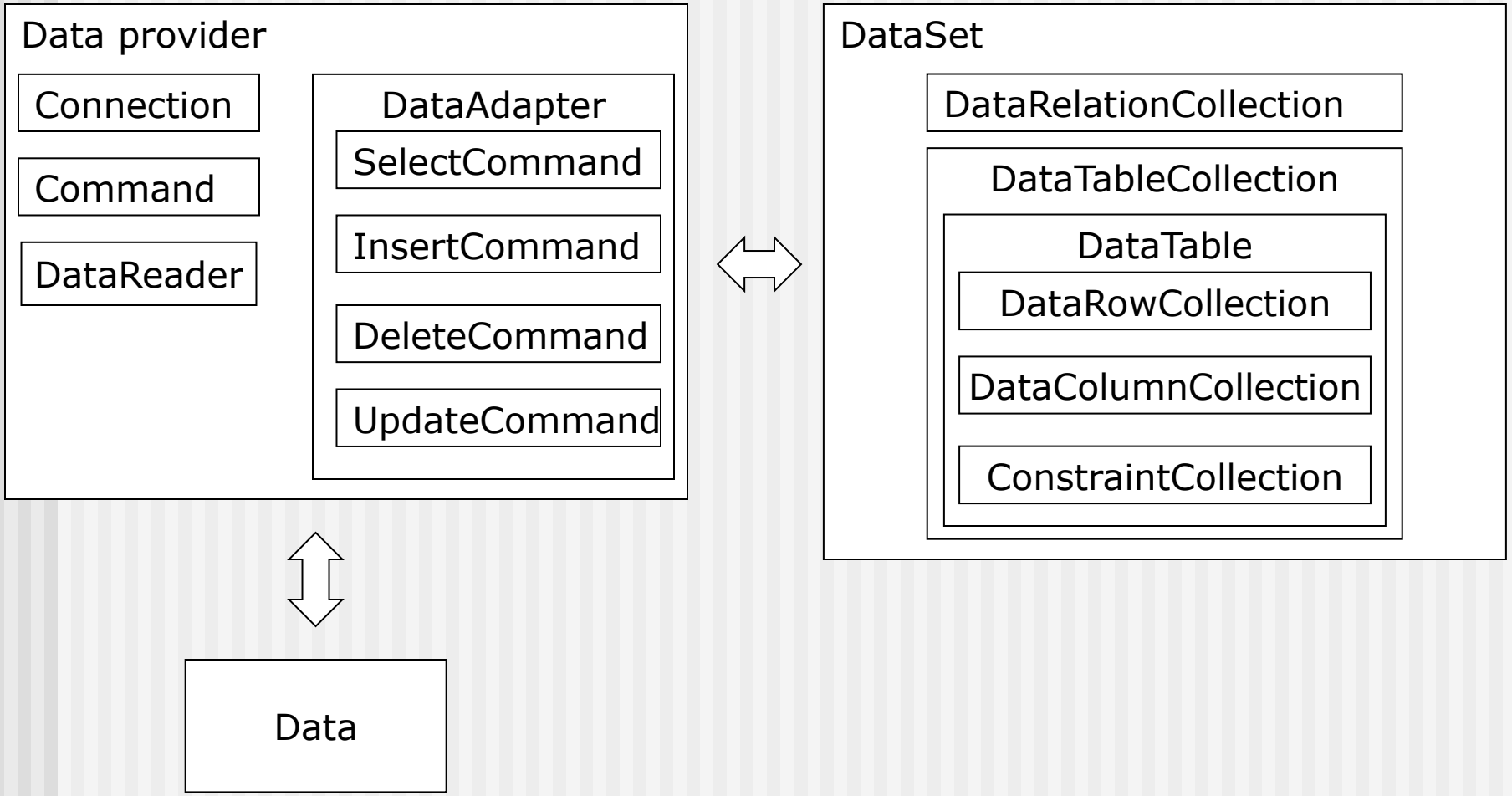
The Data Cycle

- displaying data: data-bound controls
- editing and validating data: add / modify / delete records; verify if the new values meet the requirements of the application
- saving data: send changes back to the database (e.g., *TableAdapterManager*)

Data Models

- typed / untyped datasets
- conceptual model based on the *Entity Data Model*
- *LINQ to SQL*

ADO.NET


Data provider
Connection
Command
DataReader
DataAdapter
SelectCommand
InsertCommand
DeleteCommand
UpdateCommand
DataSet
DataRelationCollection
DataTableCollection
DataTable
DataRowCollection
DataColumnCollection
ConstraintCollection
Data

Datasets

- System
  - Data
    - DataSet
    - DataTable
    - DataRelation
    - DataColumn
    - DataRow
    - Constraint

Datasets - the DataSet class

- object containing data tables that can temporarily store the data used in the application
- typed / untyped
- local in-memory cache
- also works when the application disconnects from the database
- structure similar to that of a relational database (tables, rows, columns, constraints, relationships)

Datasets - the DataSet class

- properties:
  - *Tables (DataTableCollection)* – collection of tables in the DataSet
  - *Relations (DataRelationCollection)* – collection of relations (child / parent tables)
- methods:
  - *Clear*() – clears all rows in all tables
  - *HasChanges*() – indicates whether there are new / deleted / modified rows

Datasets - the DataTable class

- properties:
  - *Rows (DataRowCollection)*
  - *Columns (DataColumnCollection)*
  - *ChildRelations* and *ParentRelations (DataRelationCollection)*

Datasets - the DataRow class

- properties:
  - *RowState* - possible values: *Added, Deleted, Modified, Unchanged*

SqlConnection

- represents a connection to a SQL Server database
- cannot be inherited
- if the SqlConnection goes out of scope, it is not closed => must ensure the connection is always closed (e.g., *Close)*
- properties:
  - *ConnectionString* - string used to open a SQL Server database (http://www.connectionstrings.com/)
  - *ConnectionTimeout* - time to wait to establish a connection before terminating the attempt and generating an error

SqlConnection

- methods:
  - *Open*()
  - *Close*()
- if a *SqlException* is generated, the *SqlConnection* remains open when the severity level <= 19

SqlCommand

- represents a Transact-SQL statement or stored procedure to be executed on a SQL Server database
- cannot be inherited
- properties:
  - *CommandText*
  - *CommandTimeout*
- methods:
  - *ExecuteNonQuery* – returns the number of affected rows

SqlCommand

- methods:
  - *ExecuteScalar* – returns the first column of the first row in the answer set
  - *ExecuteReader* – builds a SqlDataReader

SqlDataReader

- reads a forward-only stream of rows from a SQL Server database

SqlDataAdapter

- bridge between a DataSet and SQL Server to obtain data and save changes back to the database
- a set of commands and a database connection
- properties:
  - *UpdateCommand* - statement / stored procedure used to update records in the data source
  - *InsertCommand* - statement / stored procedure used when inserting records into the data source
  - *DeleteCommand* - statement / stored procedure used to delete records

SqlDataAdapter

- methods:
  - *Fill(DataSet, String)* - adds or refreshes rows in the DataSet object to match those in the data source (2nd param – name of table)
  - *Update(DataSet, String)* – changes the values in the database by executing INSERT / UPDATE / DELETE statements for every added / modified / deleted row

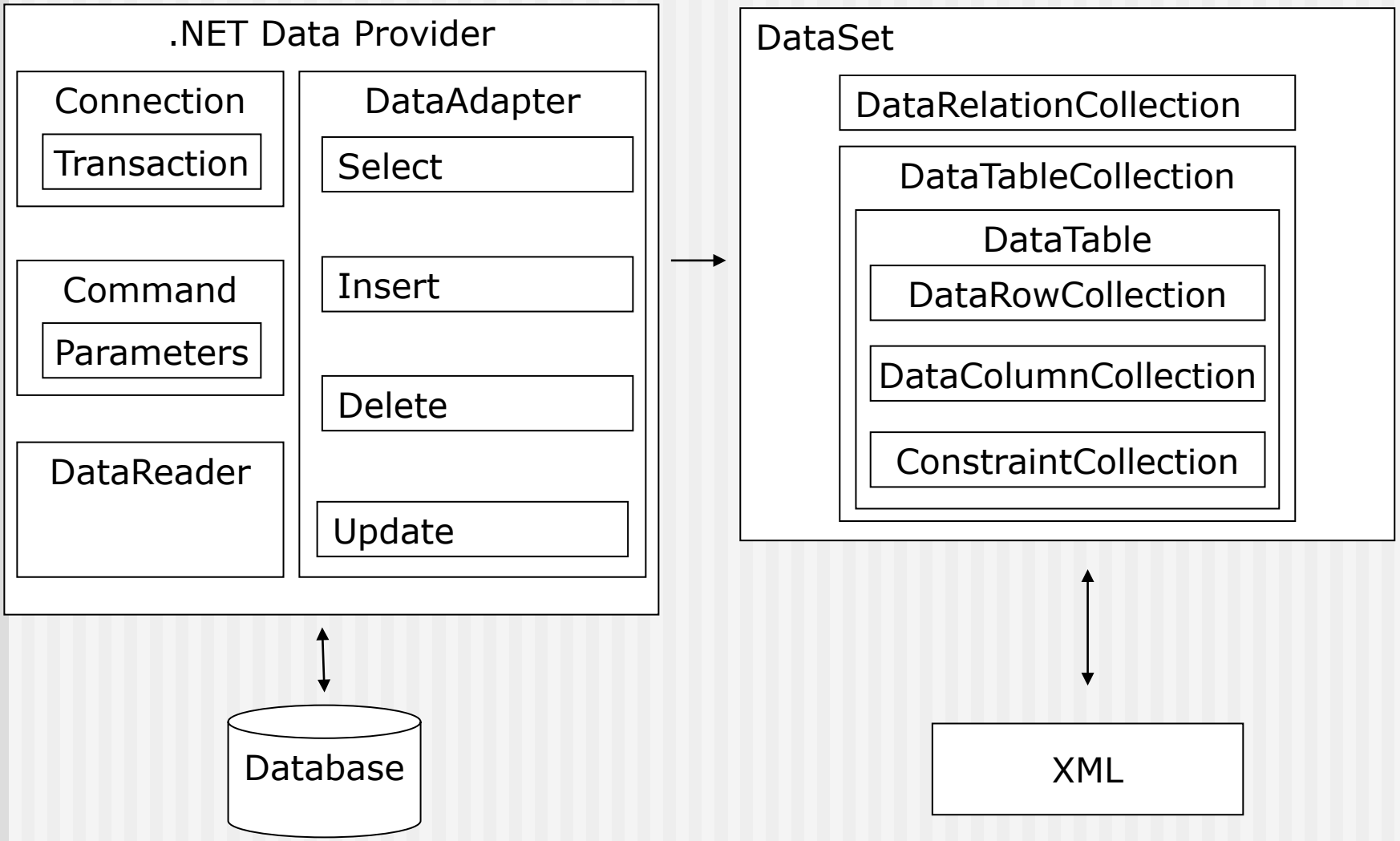
Console

- it represents the standard input, output, error streams for console applications
- properties:
  - *WindowLeft*, *WindowTop*, *WindowHeight*, *WindowWidth*, *BackgroundColor*, *Title*, etc.
- methods:
  - *Write(….), WriteLine(….)*
  - *Read(), ReadLine(), ReadKey()*

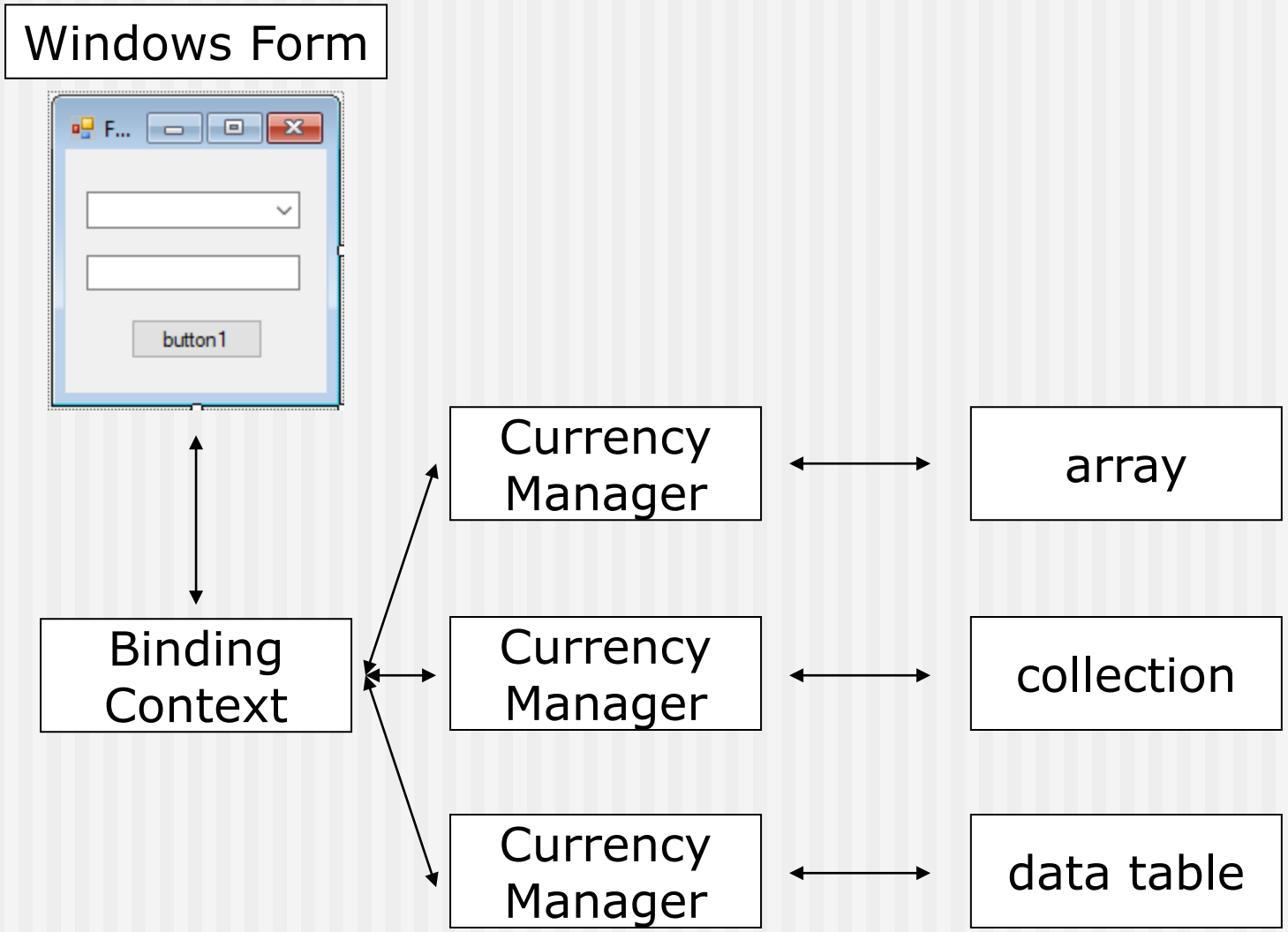
Seminar 2

ADO.NET
Data Binding

The ADO.NET Architecture



Data Binding – Windows Forms



Windows Forms – Structures to Bind To

- list-based binding – the object should support the *IList* interface
- ADO.NET provides data structures suitable for data binding
  - *DataColumn*
  - *DataTable* – with columns, rows, constraints
  - *DataView* – customized view of a single data table
  - *DataSet* – with tables, relationships
  - *DataViewManager* – customized view of a *DataSet*

CurrencyManager

- keeps data-bound controls synchronized with each other
- for each data source associated with a Windows Form, there is one *CurrencyManager* object
- *currency* = the current position within a data structure
- the *Position* property - determines the current position of all the controls using the same *CurrencyManager* (the position within the underlying list)

BindingContext

- manages the collection of *CurrencyManager* objects for any container control / form

The *DataBindings* Property and the *Binding* Class

- *DataBindings*
  - property of a control
  - retrieves the data bindings for the control
  - one can bind any property of a control to the property of an object
- *Binding* class
  - the simple binding between the property value of an object and the property value of a control
  - *Binding (String propertyName, Object dataSource, String dataMember)*

Dataset – *Fill, Update*

- a dataset doesn't contain data by default
- data tables are filled with data when executing *TableAdapter* / data adapter (*SqlDataAdapter*) queries / commands

```
aTableAdapter.Fill(aDataSet.TableName);
```

- saving data:

```
aTableAdapter.Update(aDataSet.TableName);
```

- when the *Update* method is called, the value of the *RowState* property is examined to determine which records must be saved and which command (*InsertCommand*, *UpdateCommand*, *DeleteCommand*) must be executed

Accessing Records

- every table exposes a collection of rows
- rows can be accessed through the collection's index or by using collection-specific statements in the host programming language
- typed dataset

```
TextBox1.Text = ds1.TableName[3].aField;
```

- untyped dataset

```
string val = (string)
ds1.Tables["TableName"].Rows[0]["aField"];
```

Related Tables, *DataRelation* Objects

- the data in a dataset's tables can be interrelated
- *DataRelation* objects can be created to describe the relationships among the dataset's tables
- a *DataRelation* object can be used to locate related records:
  - the *GetChildRows* method
    - called on a *DataRow* in the parent table
    - returns an array of related child records (*DataRow* objects)

Related Tables, *DataRelation* Objects

- a *DataRelation* object can be used to locate related records:
  - the *GetParentRow* method
    - called on a *DataRow* in the child table
    - returns a single *DataRow* from the parent table

Related Tables, *DataRelation* Objects

- return the child records for a parent record

```
string custID = "VINET";
NorthwindDataSet.OrdersRow[] orders;
orders = (NorthwindDataSet.OrdersRow[])
     northwindDataSet.Customers.
     FindByCustomerID(custID).GetChildRows
          ("FK_Orders_Customers");
MessageBox.Show(orders.Length.ToString());
```

Related Tables, *DataRelation* Objects

- return the parent record for a child record

```
int orderID = 10835;
NorthwindDataSet.CustomersRow customer;
customer = (NorthwindDataSet.CustomersRow)
     northwindDataSet.Orders.
     FindByOrderID(orderID).
     GetParentRow("FK_Orders_Customers");
MessageBox.Show(customer.CompanyName);
```

Creating an Application in Visual Studio

- create a new *Windows Forms* project
  - *File* -> New -> Project
  - select *Windows Forms App,* specify a name, choose a location
  - click on *OK*
  - the project is created and added to the *Solution Explorer*

Creating an Application in Visual Studio

- create a *Data Source*
  - start the *Data Source Configuration Wizard* (*Data Sources* window)
  - choose a Data Source Type (e.g., Database)
  - choose a Database Model (Dataset)
  - choose the data connection
  - select the database objects (e.g., the required tables)

Creating an Application in Visual Studio

- drag items (e.g., a particular table) from the Data Sources window onto the form to create *data-bound controls*

=> the following components are now visible in the component tray:

- *DataSet* - typed dataset that contains tables
- *BindingSource* - binds the controls on the form to the table in the dataset
- *BindingNavigator* – allows the user to navigate through the rows in the table
- *TableAdapter* – communication between the database and the dataset

Creating an Application in Visual Studio

- drag items (e.g., a particular table) from the Data Sources window onto the form to create *data-bound controls*

=> the following components are now visible in the component tray:

- *TableAdapterManager* - controls the order of individual inserts, updates, and deletes

Constraints

- two types of constraints: unique / foreign key
- unique constraint
  - all values in a set of columns must be unique
  - class *UniqueConstraint*
- foreign key constraint
  - defines rules on how to change related child records when a parent record is updated or deleted
  - class *ForeignKeyConstraint*

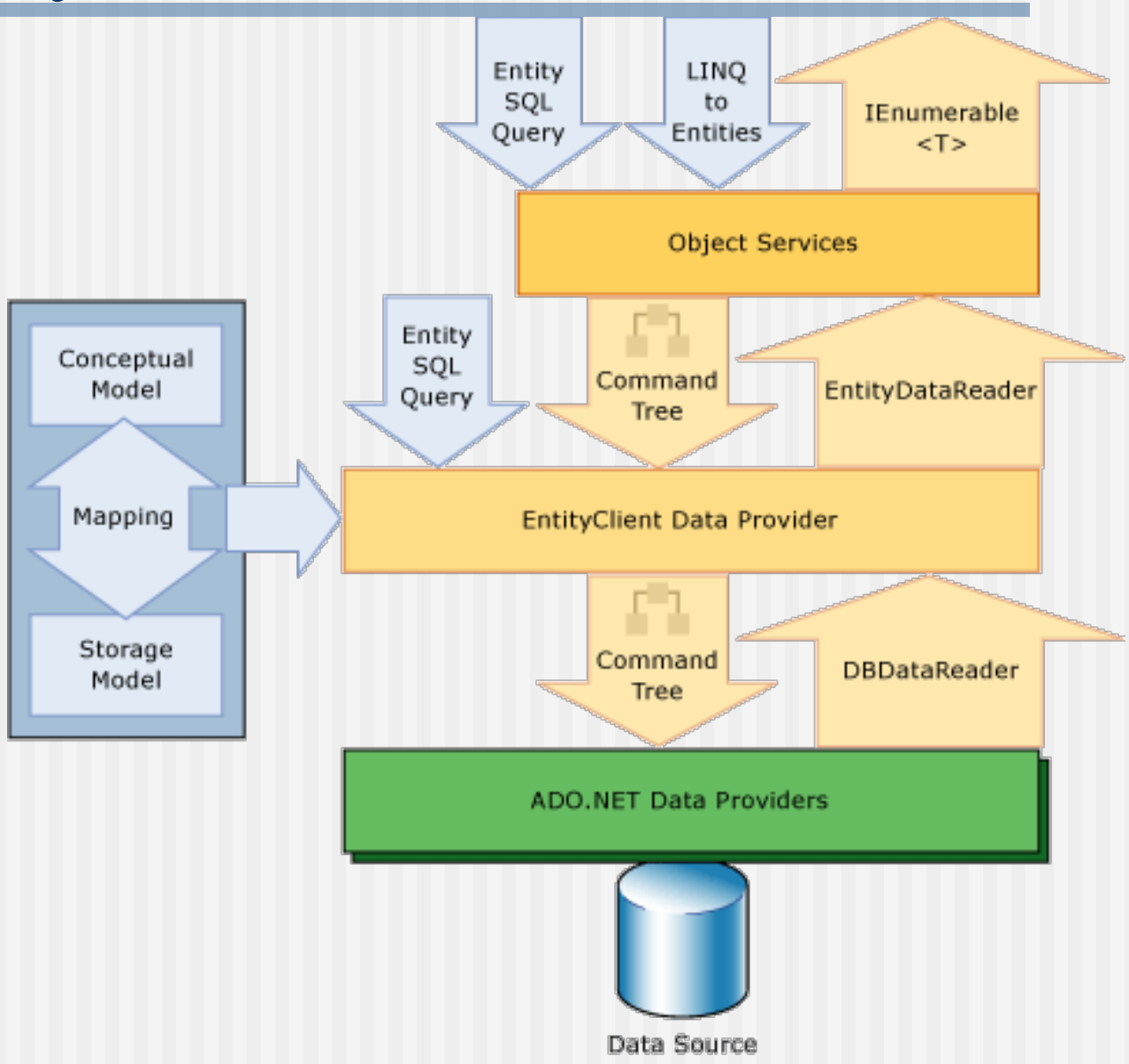
Constraints

- a foreign key constraint is automatically added when creating a *DataRelation* object in a dataset
- a table's constraints can be retrieved using the *Constraints* property
- the boolean property *EnforceConstraints* in the *Dataset* class indicates whether constraints are enforced or not (by default it's *true*)

Entity Framework

- set of technologies in ADO.NET that support the development of data-oriented applications
- developers can work with domain-specific objects and properties (as opposed to dealing with tables and columns)
- developers can query entities and relationships in the domain model; the Entity Framework translates such operations to data source–specific commands

Entity Framework



Seminar 3

Transactions
Concurrency Control in SQL Server

Transactions in SQL Server

- combine multiple operations into a single unit of work
- the actions of each user are processed using a different transaction
- objective:
  - maximize throughput => transactions must be allowed to execute in parallel
- ACID properties
- serializability

Transactions in SQL Server

- transaction invocation - mechanisms:
  - unless specified otherwise, each command is a transaction
  - BEGIN TRAN, ROLLBACK TRAN, COMMIT TRAN
  - SET IMPLICIT_TRANSACTIONS ON
    - enables chained transactions
- SET XACT_ABORT ON
  - SQL errors => rollback transaction

Transactions in SQL Server

- local transactions / distributed transactions
- one can *nest* transactions (but transactions are not really nested)
- named *savepoints*
  - allow a portion of work in a transaction to be rolled back

Concurrency Problems

- transaction isolation tackles four major concurrency problems:
  - *lost updates* - two transactions (writers) modify the same piece of data
  - *dirty reads* - a transaction (reader) reads uncommitted data, i.e., data changed by another ongoing transaction
  - *unrepeatable reads* - a row read by a transaction (reader) is changed by another transaction while the reader is in progress (if the 1st transaction reads the row again it will get different row values)

Concurrency Problems

- transaction isolation tackles four major concurrency problems:
  - *phantoms* - transaction T1 reads a set of rows based on a search predicate; transaction T2 generates a new row (I/U) that matches the search predicate while T1 is ongoing; if T1 issues the same read operation, it will get an extra row

Concurrency Problems

- transaction isolation is achieved through the locking mechanism
- *write locks*
  - exclusive locks, i.e., they don't allow other readers / writers
- *read locks*
  - allow other readers
  - don't allow other writers

Concurrency Problems

- isolation levels determine:
  - whether read locks are acquired for read operations
  - the duration of read locks
  - whether key-range locks are acquired to prevent phantoms

Locking in SQL Server

- locks
  - usually managed by the Lock Manager (not via apps)
- lock granularity:
  - *Row / Key, Page, Table, Extent*, Database*
- hierarchy of related locks
  - locks can be acquired at several levels
- lock escalation
  - > 5000 locks per object (pros & cons)

* contiguous group of 8 pages

Locking in SQL Server

- lock types:
- *Shared* (S)
  - read operations
- *Update* (U)
  - deadlock avoidance mechanism
- *Exclusive* (X)
  - write operations
  - incompatible with other locks

	S	X
S	Yes	No
X	No	No

Locking in SQL Server

- lock types:
- *Exclusive* (X)
  - read operations by other transactions can be performed only when using the NOLOCK hint or the READ UNCOMMITTED isolation level
  - a transaction always acquires exclusive locks to modify data (regardless of the isolation level)
  - exclusive locks are released when the transaction completes execution

Locking in SQL Server

- lock types:
- *Intent* (IX, IS, SIX)
  - intention to lock (for performance improvement purposes)
- *Schema* (Sch-M, Sch-S)
  - schema modification, schema stability
  - Sch-M
    - prevents concurrent access to the table
  - Sch-S
    - doesn't allow DDL operations to be performed on the table

Locking in SQL Server

- lock types:
- *Bulk Update* (BU)
  - bulk load data concurrently into the same table
  - BULK INSERT statement
  - TABLOCK hint
- *Key-Range*
  - protect a range of rows implicitly included in a set of records read by a transaction (under the SERIALIZABLE isolation level)

Key-Range Locking

- lock sets of rows defined by a predicate

  …**`WHERE grade between 8 and 10`**

- lock existing data, as well as data that doesn't exist
- use predicate "**`grade between 8 and 10`**" 2 times => obtain the same rows

Transaction Workspace Locks

- every connection to a database acquires a *Shared_Transaction_Workspace* lock
- exceptions - connections to master, tempdb
- used to prevent:
  - DROP
  - RESTORE

Isolation Levels in SQL Server

- **`READ UNCOMMITTED`**
  - allows dirty reads (a transaction can see uncommitted changes made by another ongoing transaction)
  - no S locks when reading data
- **`READ COMMITTED`** (default isolation level)
  - a transaction cannot read data that has been modified by another ongoing transaction
  - allows unrepeatable reads
  - S locks - released as soon as the SELECT operation is performed

Isolation Levels in SQL Server

- **`READ COMMITTED`**
  - X locks - released at the end of the transaction
- **`REPEATABLE READ`**
  - holds S locks and X locks until the end of the transaction
  - doesn't allow dirty reads, unrepeatable reads
  - phantom reads can occur

Isolation Levels in SQL Server

- **`SERIALIZABLE`**
  - highest isolation level
  - holds locks (including key-range locks) during the entire transaction
  - doesn't allow dirty reads, unrepeatable reads, phantom reads
- **`SNAPSHOT`**
  - working on a snapshot of the data

- SQL syntax
  - **`SET TRANSACTION ISOLATION LEVEL`** …

Isolation Levels in SQL Server

concurrency probl. / isolation level	Chaos	Read Uncommitted	Read Committed	Repeatable Read	Serializable
Lost Updates?	Yes	No	No	No	No
Dirty Reads?	Yes	Yes	No	No	No
Unrepeatable Reads?	Yes	Yes	Yes	No	No
Phantoms?	Yes	Yes	Yes	Yes	No

Deadlocks

- SQL Server uses deadlock detection
- the transaction that's least expensive to roll back is terminated
- capture and handle error 1205
- SET LOCK_TIMEOUT
  - specify how long (in milliseconds) a transaction waits for a locked resource to be released
  - value 0 - immediate termination
- SET DEADLOCK_PRIORITY
  - values: {*LOW*, *NORMAL*, *HIGH*, <numeric-priority>}
  - <numeric-priority> ::= {-10, -9, …, 10}

Reduce the Likelihood of Deadlocks

- transactions - short & in a single batch
- obtain / verify input data from the user before opening a transaction
- access resources in the same order
- use a lower / a row versioning isolation level

Seminar 4

Multiversioning

Monitoring Locks

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

Resource Types

- RID – row in a heap
- Key – row in an index
- Page
- HoBT – heap or B-tree
- Object – table, view, etc.
- File
- Database
- Metadata
- Application

Isolation Levels in SQL Server

- **`READ UNCOMMITTED`**
  - allows dirty reads (a transaction can see uncommitted changes made by another ongoing transaction)
  - no S locks when reading data
- **`READ COMMITTED`** (default isolation level)
  - a transaction cannot read data that has been modified by another ongoing transaction
  - allows unrepeatable reads
  - S locks - released as soon as the SELECT operation is performed

Isolation Levels in SQL Server

- **`READ COMMITTED`**
  - X locks - released at the end of the transaction
- **`REPEATABLE READ`**
  - holds S locks and X locks until the end of the transaction
  - doesn't allow dirty reads, unrepeatable reads
  - phantom reads can occur

Isolation Levels in SQL Server

- **`SERIALIZABLE`**
  - highest isolation level
  - holds locks (including key-range locks) during the entire transaction
  - doesn't allow dirty reads, unrepeatable reads, phantom reads
- **`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

Row-Level Versioning (RLV)

- introduced in SQL Server 2005
- useful when the user needs *committed* data (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 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

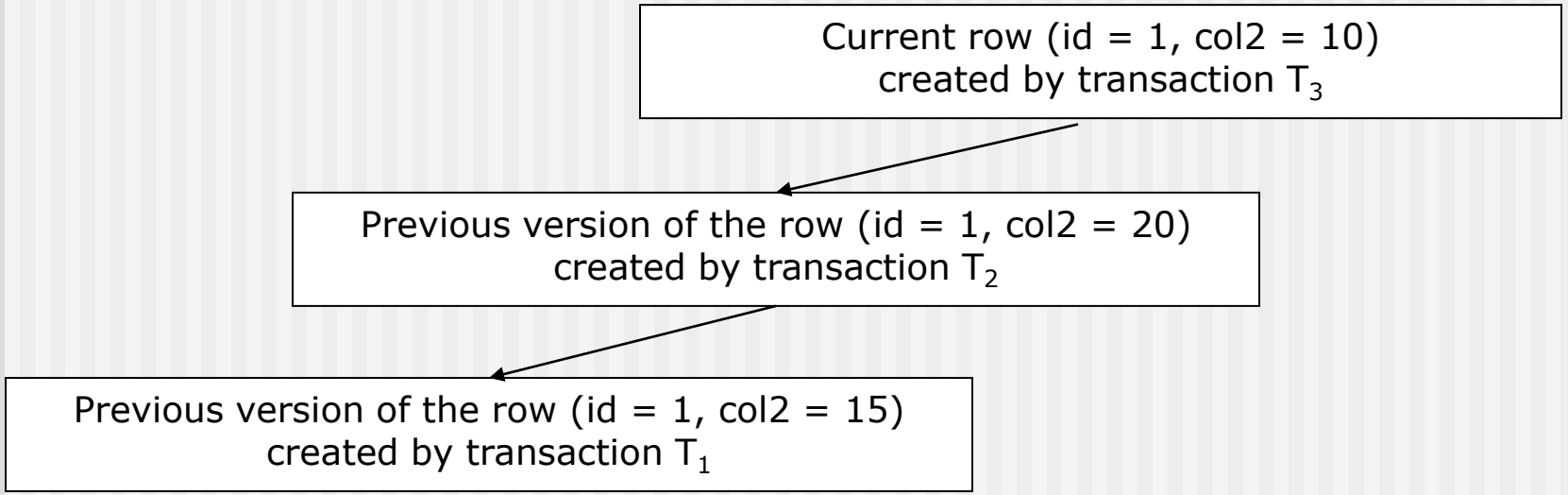
Full Snapshot Isolation

```
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

- each version of a row is marked with the *transaction sequence number* (XSN) of the transaction that changed the row
- all versions are kept in a linked list:



Row-Level Versioning

- advantages:
  - increased concurrency level
  - positive impact - 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
- row versioning is used to create the *inserted* / *deleted* tables
- before SQL Server 2005:
  - the *deleted* table was created using the transaction log – affected performance

Index Creation & RLV

- index creation / rebuilding in previous versions of SQL Server:
  - *clustered index* - table exclusively locked, data entirely inaccessible
  - *non-clustered index* - 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	snapshot	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

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 0 – all queries are allowed to run

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

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

Seminar 5

Performance Tuning in SQL Server

Query Tuning Methodology

- identify waits (bottleneck) at the server level
  - locks
  - transaction log
  - I/O
  - etc.
- correlate waits with queues
- drill down to database / file level
- drill down to process level
- tune problematic queries

* DMVs - dynamic management views

Identify Waits

- **`sys.dm_os_wait_stats`**
  - returned table
    - wait_type
      - resource waits (locks, latches, network, I/O), queue waits, external waits
    - waiting_tasks_count
    - wait_time_ms
    - max_wait_time_ms
    - signal_wait_time_ms
- reset DMV values
  - **`DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);`**

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
  - ***object_name*** - the category of the counter
  - ***counter_name*** - the name of the counter
  - ***instance_name*** - the name of the specific instance of the counter; often contains the name of the database
  - ***cntr_value*** - the current value of the counter
  - ***cntr_type*** - the type of the counter (as defined by the Windows performance architecture)

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- > 500 counters: Access Methods, User Settable, Buffer Manager, Broker Statistics, SQL Errors, Latches, Buffer Partition, SQL Statistics, Locks, Buffer Node, Plan Cache, Cursor Manager by Type, Memory Manager, General Statistics, Databases, Catalog Metadata, Broker Activation, Broker/DBM Transport, Transactions, Cursor Manager Total, Exec Statistics, Wait Statistics, etc.
- ***cntr_type*** = 65792 → ***cntr_value*** contains the actual value

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- ***cntr_type*** = 537003264 → ***cntr_value*** contains real-time results, which are divided by a "base" to obtain the actual value; by themselves, they are useless
  - to get a ratio: divide by a "base" value
  - to get a percentage: multiply the result by 100.0

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- *cntr_type* = 272696576
  - time-based, cumulative counters
  - a secondary table can be used to log intermediate values

Correlate Waits with Queues

- **sys.dm_os_performance_counters**
- ***cntr_type*** = 1073874176 and ***cntr_type*** = 1073939712 → poll both the value (1073874176) and the base value (1073939712)
- poll both values again (e.g., after 15 seconds) ☺
- to obtain the desired result, compute:

  UnitsPerSec = (cv2 – cv1) / (bv2 – bv1) / 15.0

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returns I/O information about *data files* and *log files*
- parameters
  - database_ID
    - NULL = all databases
    - useful function: DB_ID
  - file_ID
    - NULL = all files
    - useful function: FILE_IDEX

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returned table
    - database_ID
    - file_ID
    - sample_ms - # of milliseconds since the computer was started
    - num_of_reads - number of reads issued on the file
    - num_of_bytes_read - number of bytes read on the file
    - io_stall_read_ms – total time users waited for reads issued on the file

Drill Down to Database / File Level

- **`sys.dm_io_virtual_file_stats`**
  - returned table
    - num_of_writes - number of writes
    - num_of_bytes_written - total number of bytes written to the file
    - io_stall_write_ms - total time users waited for writes to be completed on the file
    - io_stall - total time users waited for the completion of I/O operations (ms)
    - file_handle

Drill Down to the Process Level

- a filter on duration / I/O only isolates individual processes (batch / proc / query)
- aggregate performance information by query pattern
  - patterns can be easily identified when using stored procedures
  - when one doesn't use stored procedures:
    - quick and dirty approach: LEFT(query string, n)
    - use a parser to identify the query pattern

Indexes

- one of the major factors influencing query performance
  - impact on: filtering, joins, sorting, grouping; blocking and deadlock avoidance, etc.
  - effect on modifications: positive effect (locating the rows); negative effect (cost of modifying the index)
- understanding indexes and their internal mechanisms
  - clustered/nonclustered, single/multicolumn, indexed views, indexes on computed columns, covering scenarios, intersection

Indexes

- one should carefully judge whether additional index maintenance costs are justified by improvements in query performance
  - take into account the environment and the ratio between SELECT queries and data modifications
- *multicolumn indexes*
  - tend to be more useful than single-column indexes
  - the query optimizer is more likely to use such indexes to cover a query

Indexes

- *indexed views* come with a higher maintenance cost than standard indexes
  - mandatory option
    - WITH SCHEMABINDING

Tools to Analyze Query Performance

- graphical execution plan
- STATISTICS IO - scan count, logical reads, physical reads, *read-ahead* reads
- STATISTICS TIME - duration and net CPU time
- SHOWPLAN_TEXT – SQL Server returns detailed information about how the statements are executed
- SHOWPLAN_ALL - SQL Server returns detailed information about how the statements are executed, provides estimates of the resource requirements

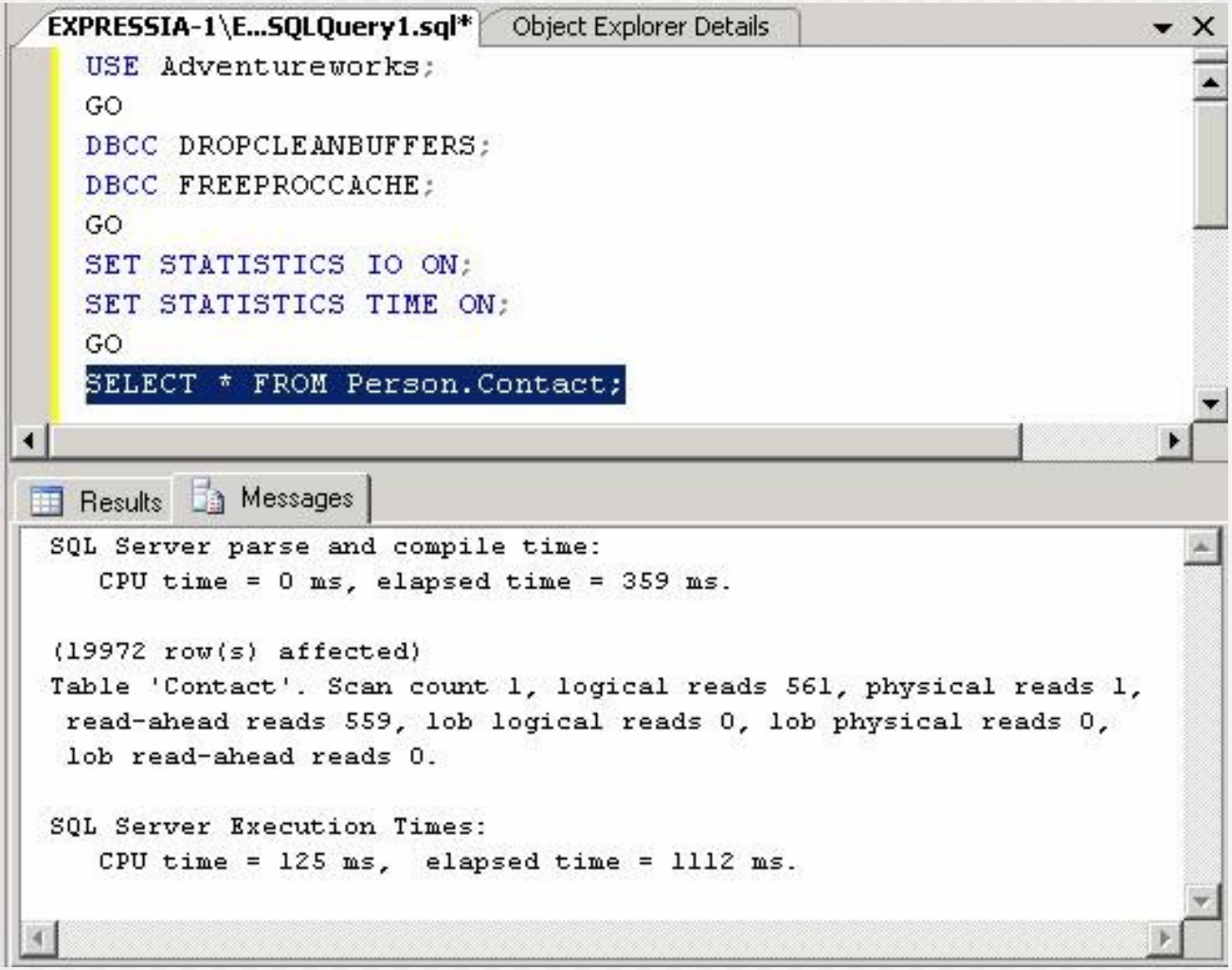
Tools to Analyze Query Performance

- STATISTICS PROFILE - a profile of the query execution
- STATISTICS XML – actual plan information in XML format
- SHOWPLAN_XML - estimated plan information in XML format

Query Optimization

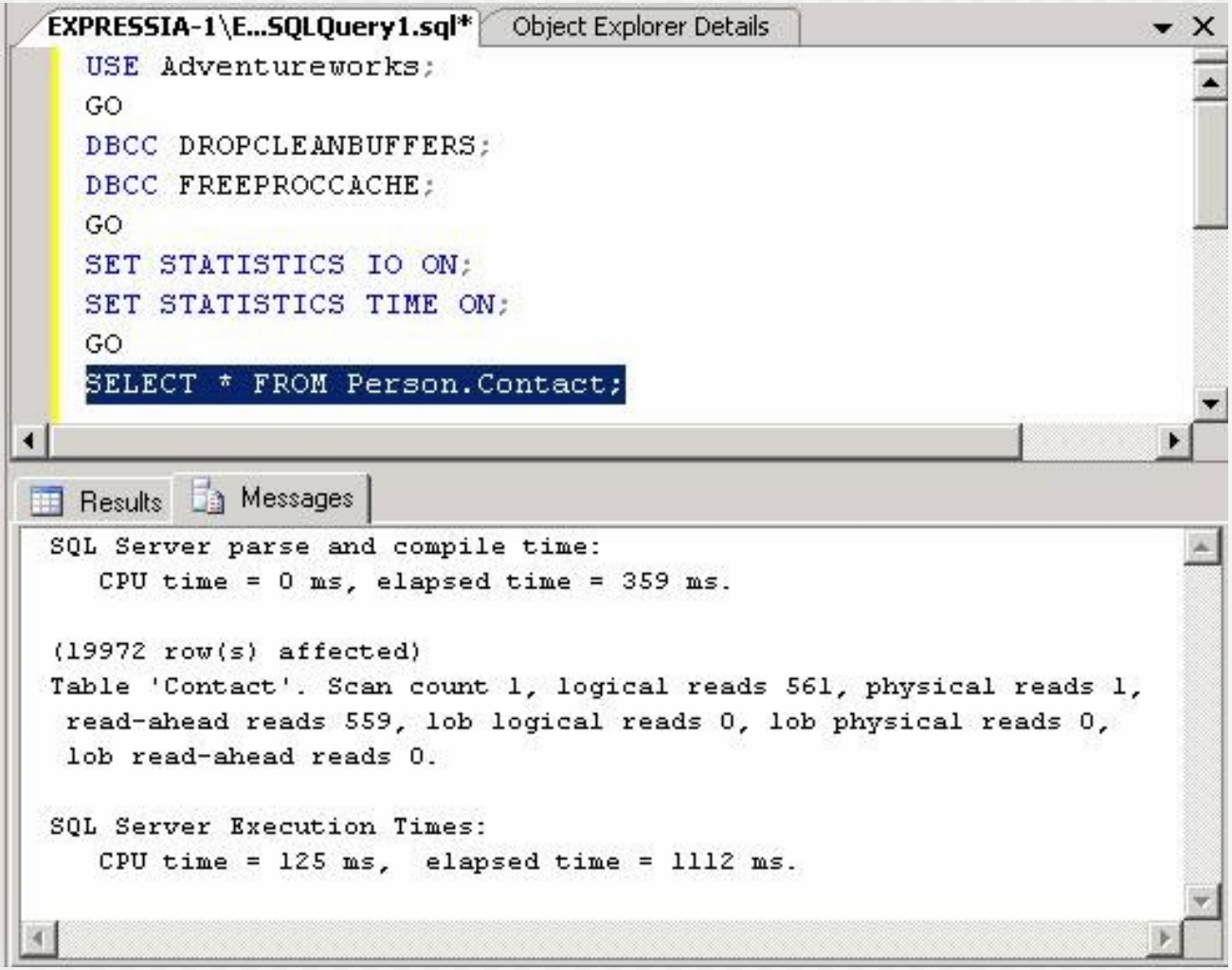
- evaluating execution plans
  - sequences of physical/logical operations
- optimization - factors
  - search predicate
  - tables involved in joins
  - join conditions
  - result set size
  - list of indexes
- goal - avoid worst query plans
- SQL Server uses a *cost-based* query optimizer

STATISTICS IO and STATISTICS TIME



- DBCC DROPCLEANBUFFERS – clear data from the cache
- DBCC FREEPROCCACHE – clear execution plans from the cache

STATISTICS IO and STATISTICS TIME



- *CPU time, elapsed time* – CPU time and elapsed time to parse and compile the query, and to execute it

STATISTICS IO and STATISTICS TIME

EXPRESSIA-1\E...SQLQuery1.sql* | Object Explorer Details

```
USE Adventureworks;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
```

Results | Messages

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 359 ms.

(19972 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,
 read-ahead reads 559, lob logical reads 0, lob physical reads 0,
 lob read-ahead reads 0.

SQL Server Execution Times:
   CPU time = 125 ms,  elapsed time = 1112 ms.
```

- *physical reads* - number of pages read from the disk
- *read-ahead reads* - number of pages placed in the cache for the query

STATISTICS IO and STATISTICS TIME

EXPRESSIA-1\E...SQLQuery1.sql* | Object Explorer Details

```
USE Adventureworks;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.Contact;
```

Results | Messages

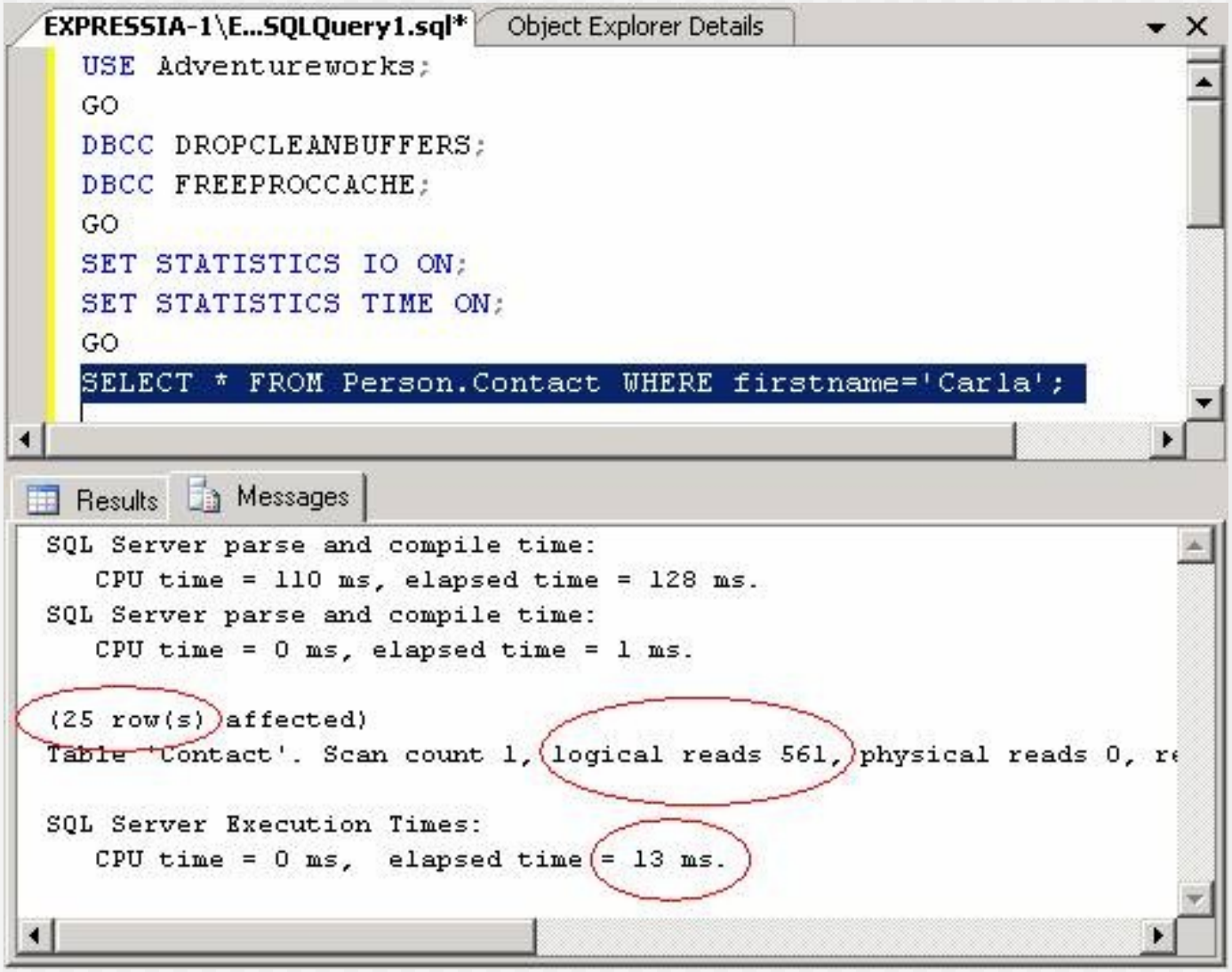
```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 359 ms.

(19972 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,
 read-ahead reads 559, lob logical reads 0, lob physical reads 0,
 lob read-ahead reads 0.

SQL Server Execution Times:
   CPU time = 125 ms,  elapsed time = 1112 ms.
```

- *scan count* - number of seeks or scans after reaching the leaves
- *logical reads* - number of pages read from the data cache

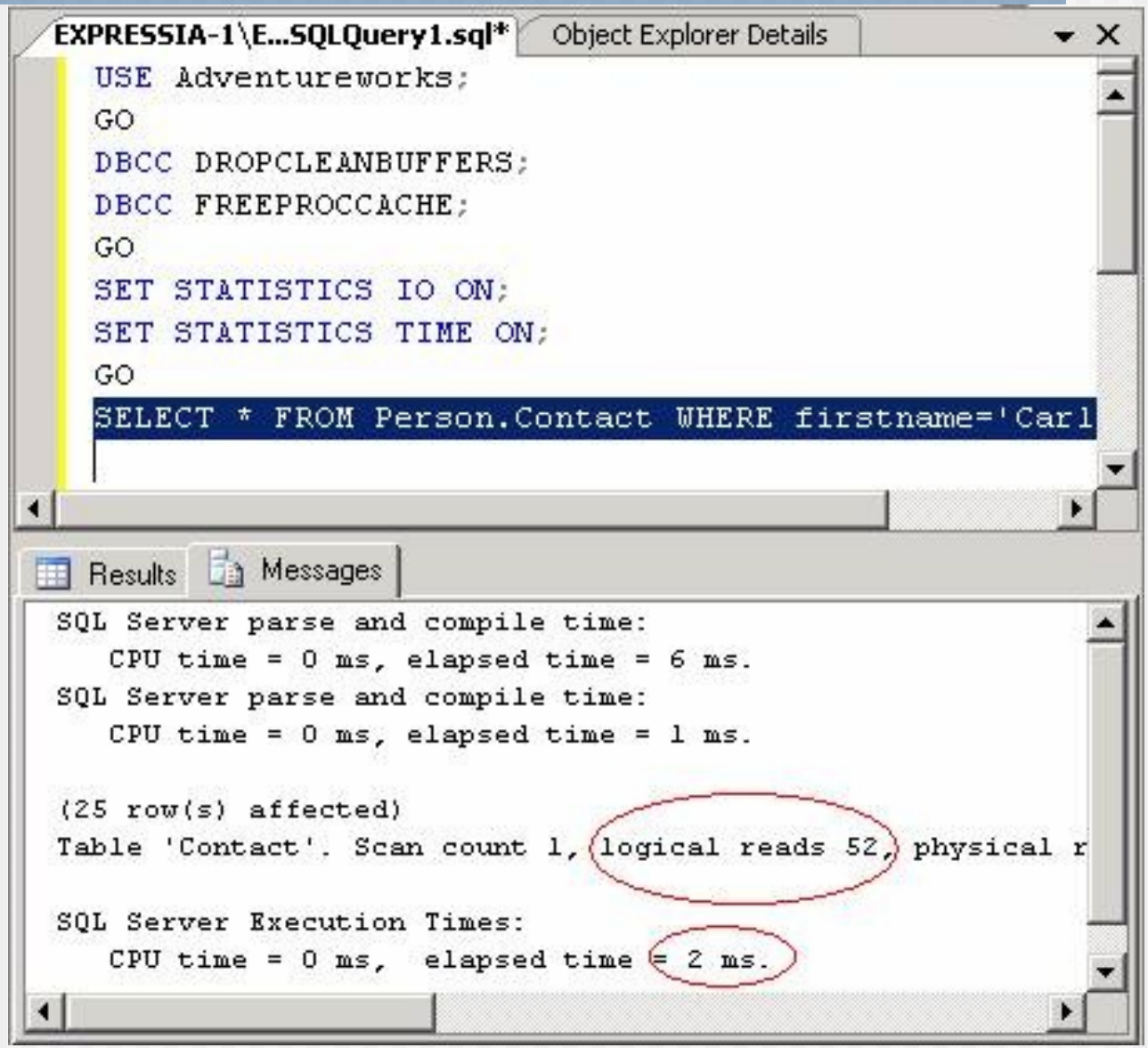
STATISTICS IO and STATISTICS TIME



STATISTICS IO and STATISTICS TIME

```
USE AdventureWorks
GO
CREATE NONCLUSTERED INDEX IDX_FirstName
  ON Person.Contact(FirstName ASC)
GO
```

STATISTICS IO and STATISTICS TIME

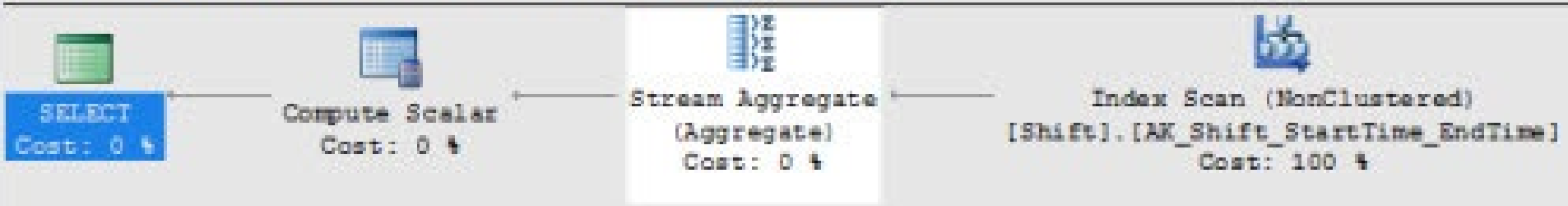


Graphical Execution Plan

```
USE AdventureWorks
GO
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
GO
```

Results | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT COUNT(*) cRows FROM HumanResources.Shift;
```



SHOWPLAN_ALL

```
SET SHOWPLAN_ALL ON;
GO
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
GO
SET SHOWPLAN_ALL OFF;
GO
```

Results

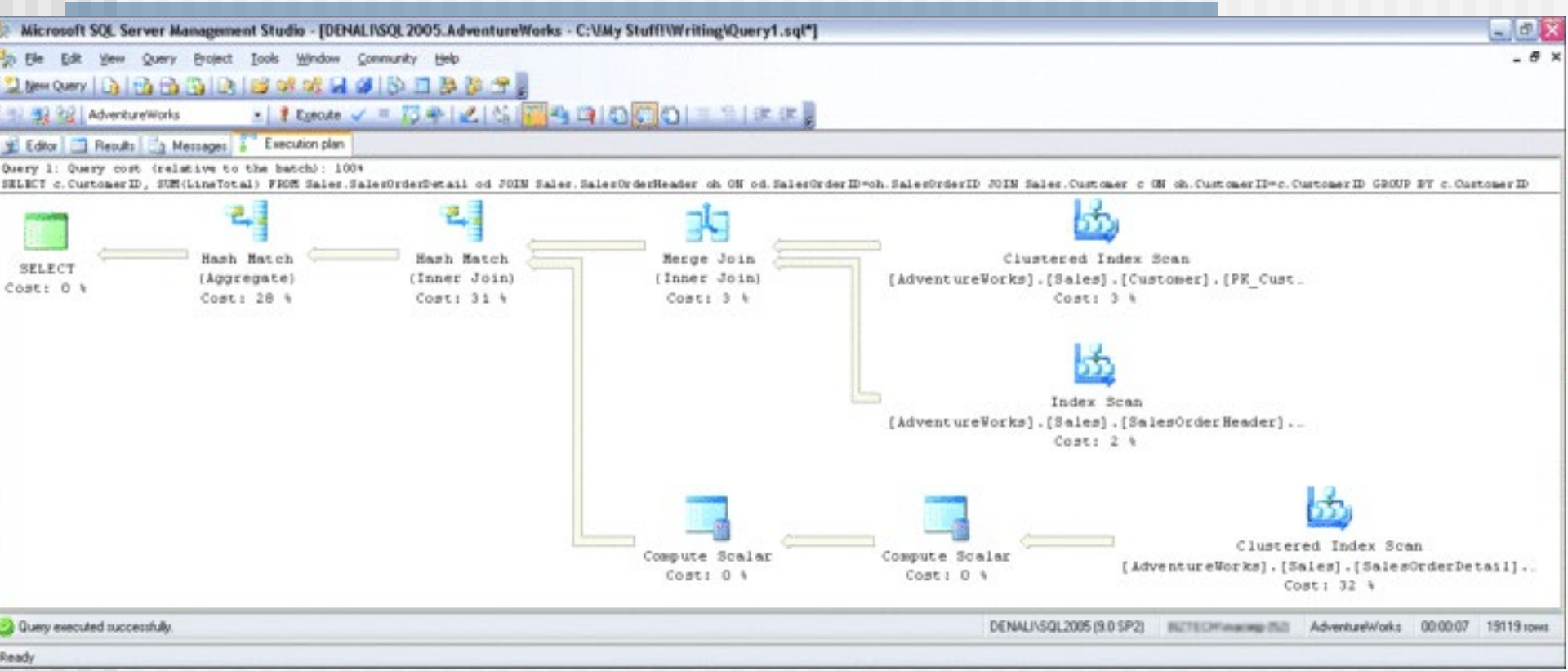
```
StmtText
-----------------------------------------------------------------------------
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
  |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))
       |--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))
            |--Index Scan(OBJECT:([master].[HumanResources].[Shift].[AK_Shif

(4 row(s) affected)
```

Graphical Execution Plan

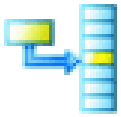
```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
  JOIN Sales.SalesOrderHeader oh ON
    od.SalesOrderID = oh.SalesOrderID
  JOIN Sales.Customer c ON
    oh.CustomerID = c.CustomerID
GROUP BY c.CustomerID
```

Graphical Execution Plan



Graphical Execution Plan

SELECT
Cost: 0 %



Hash Match
(Aggregate)

SELECT

Cached plan size	40 B
Degree of Parallelism	0
Memory Grant	812
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	3,31365
Estimated Number of Rows	19045

Statement

```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od JOIN
Sales.SalesOrderHeader oh
ON od.SalesOrderID=oh.SalesOrderID
JOIN Sales.Customer c ON
oh.CustomerID=c.CustomerID
GROUP BY c.CustomerID
```

Clustered Index Scan
[AdventureWorks].[Sales].[SalesOrderDetail].…
Cost: 32 %

Clustered Index Scan

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	121317
Estimated I/O Cost	0,915718
Estimated CPU Cost	0,133606
Estimated Operator Cost	1,04932 (32%)
Estimated Subtree Cost	1,04932
Estimated Number of Rows	121317
Estimated Row Size	29 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	8

Object

[AdventureWorks].[Sales].[SalesOrderDetail].
[PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID]
[od]

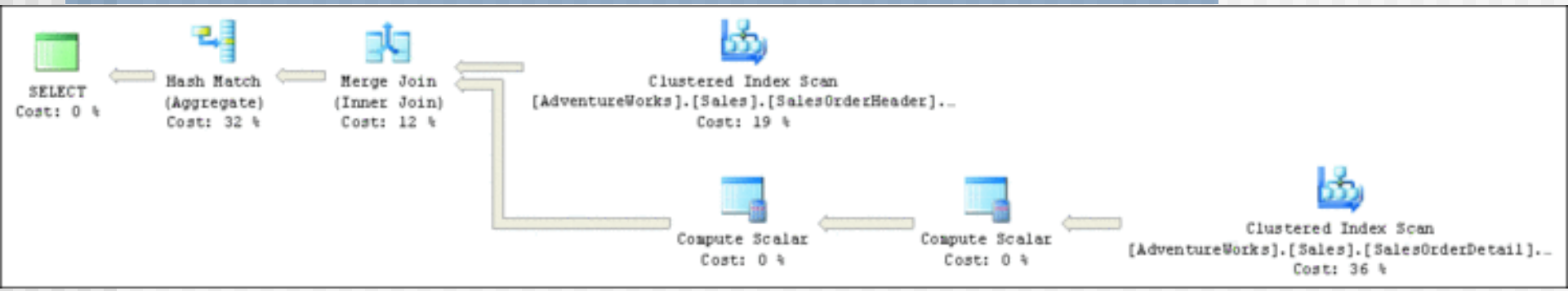
Output List

[AdventureWorks].[Sales].
[SalesOrderDetail].SalesOrderID; [AdventureWorks].
[Sales].[SalesOrderDetail].OrderQty; [AdventureWorks].
[Sales].[SalesOrderDetail].UnitPrice; [AdventureWorks].
[Sales].[SalesOrderDetail].UnitPriceDiscount

Graphical Execution Plan

```
SELECT oh.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
  JOIN Sales.SalesOrderHeader oh ON
    od.SalesOrderID=oh.SalesOrderID
GROUP BY oh.CustomerID
```

Graphical Execution Plan



Graphical Execution Plan

```
CREATE INDEX IDX_OrderDetail_OrderID_TotalLine
  ON Sales.SalesOrderDetail (SalesOrderID)
  INCLUDE (LineTotal)
```

Model for the Practical Exam

I. *Customers* is a table in a SQL Server database with schema *Customers[CustomerID, FirstName, LastName, City, DateOfBirth]*. The primary key is underlined.

CustomerID is the search key of the clustered index on *Customers.* The table doesn't have any other indexes.

Consider the interleaved execution below. There are no other concurrent transactions. The value of *City* for the customer with *CustomerID 2* is *Timisoara* when T1 begins execution.

Answer questions 1-3 (each question has at least one correct answer).

T1	**T2**
BEGIN TRAN SELECT City FROM Customers WHERE CustomerID = 2	
	BEGIN TRAN UPDATE Customers SET City = 'Cluj-Napoca' WHERE CustomerID = 2
UPDATE Customers SET City = 'Bucuresti' WHERE CustomerID = 2	
	ROLLBACK TRAN
COMMIT TRAN	

time

1. T1 and T2 run under READ UNCOMMITTED. After the *COMMIT TRAN* statement in T1, the *City* value for the customer with *CustomerID* 2 is:
a. *Timisoara*
b. *Cluj-Napoca*
c. *Bucuresti*
d. NULL
e. None of the above answers is correct.

2. T1 runs under READ COMMITTED and T2 under REPEATABLE READ. After the *COMMIT TRAN* statement in T1, the *City* value for the customer with *CustomerID* 2 is:
a. *Timisoara*
b. *Cluj-Napoca*
c. *Bucuresti*
d. NULL
e. None of the above answers is correct.

3. T1 runs under REPEATABLE READ and T2 runs under READ COMMITTED. Then:
a. T1 doesn't acquire a shared lock for its SELECT statement.
b. T1 acquires a shared lock for its SELECT statement.
c. T2 needs an exclusive lock for its UPDATE statement.
d. T1 needs an exclusive lock for its UPDATE statement.
e. None of the above answers is correct.

II. Create a database for a MiniFacebook system. The entities of interest to the problem domain are: *Users*, *Pages*, *Likes*, *Categories*, *Posts*, and *Comments.* Each user has a name, current city and date of birth. A user can like multiple pages. The system stores the date of each like. A page has a name and a category, e.g., *sports*, *movies*, *music*, etc. A category also has a category description. Users write posts and comment on existing posts. A user's post has a date, text, and number of shares. A comment is anonymous, has a text, a date, and a flag indicating whether it's a top comment for the corresponding post.

1. Write an SQL script that creates the corresponding relational data model.

2. Create a Master/Detail Form that allows one to display the posts for a given user, to carry out <insert, update, delete> operations on the posts of a given user. The form should have a *DataGridView* named *dgvUsers* to display the users, a *DataGridView* named *dgvPosts* to display all the posts of the selected user, and a button for saving added / deleted / modified posts. You must use the following classes: *DataSet, SqlDataAdapter, BindingSource.*

3. Create a scenario that reproduces the non-repeatable read concurrency issue on this database. Explain why the non-repeatable read occurs, and describe a solution to prevent this concurrency issue. Don't use stored procedures.

I.	1	1p
	2	1p
	3	1p
II.	1	2p
	2	2p
	3	2p
		1p of