LINQ

Type safe Queries

This document covers

Introduction

## Prerequisites

Before we can understand LINQ queries we need to introduce some prerequisites.

* Enumerators
* Enumerables
* Foreach statements
* Iterators

#### Enumerators

An enumerator is a read-only forward cursor over a collection of elements. We define enumerators by implementing the IEnumerator<T> interface.

Listing Simple Enumerator

public class SimpleEnumerator : IEnumerator<int>

{

public int Current => i;

object IEnumerator.Current => i;

public void Dispose() {}

public bool MoveNext() => i++ < 4;

public void Reset() => i = -1;

private int i = -1;

}

We can use our enumerator as follows

var e = new SimpleEnumerator();

while (e.MoveNext())

WriteLine(e.Current);

#### Enumerables

Enumerables produce enumerators.

Listing Enumerable

public class SimpleEnumerable : IEnumerable<int>

{

public IEnumerator<int> GetEnumerator()

{

return new SimpleEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

Enumerators are consumed by foreach statements

var e = new SimpleEnumerable();

foreach (var element in e)

WriteLine(e);

#### Foreach statements

Foreach statements consume enumerables. If the compiler sees a foreach statement like this

Listing Foreach Statements

var sequence = new List<int>( new [] {1,2,3});

foreach (var element in sequence)

{

WriteLine(element);

}

It generates something along the lines of this

using (IEnumerator<int> en = sequence.GetEnumerator())

{

while (en.MoveNext())

WriteLine(en.Current);

}

#### Iterators

Iterators provide an elegant means of creating enumerators and enumerables. The following code uses iterators to produce an enumerable whose enuerators walk over the first n items in the fibonacci sequence from 0 to n-1

Listing Iterators

IEnumerator<int> GetFibonacci(int numEntries)

{

for (int i = 0, current = 0, next = 1, nextnext = 1;

i < numEntries; i++)

{

yield return current;

nextnext = current + next;

current = next;

next = nextnext;

}

}

Questions Prerequisites

…\linqpad\Queries\InterviewQuestions\Linq\01 Prerequisites

## Basics

LINQ enables one to write type-safe queries on enumerable collections. The basic concepts are

* Sequence Any collection that implements IEnumerable<T>
* Element A single constituent of the collection
* Query operator A method that transforms one sequence into another
* Query A combination of query operators that performs a transform

The following piece of code shows all four together

Listing LINQ Basics

// 1. A sequence is any collection implementing IEnumerable<T>

IEnumerable<int> sequence = new int[] {0,1,2,3,4};

// 2. An element is a single constituent of the sequence

foreach (var element in sequence)

WriteLine(element);

// 4. Queries combine query operators

IEnumerable<int> output = sequence

.Where(s => s%2 ==0)

.Select(s => s\*s);

…

// 3. Query operators transform sequences

public static class QueryOperators

{

public static IEnumerable<T> Where<T>(this IEnumerable<T> input,

Func<T,bool> predicate)

{

foreach (var element in input)

if ( predicate(element))

yield return element;

}

public static IEnumerable<TOut> Select<TOut,TIn>(this IEnumerable<TIn> input, Func<TIn,TOut> trans)

{

foreach (var element in input)

yield return trans(element);

}

}

Query operators are implemented as extension methods that take an enumerable argument representing an input sequence and a delegate that applies some transformation to create an output sequence. As such query operators are easily composed into queries. Most query operators are not executed when they are constructed. Instead they are executed when they are enumerated. Delayed or lazy execution provides the following benefits.

* Decouples construction from execution
* Allows one to construct a query in multiple steps
* One can re-evaluate a query by enumerating it again

Exceptions to lazy execution

The following operators are exceptions which cause immediate execution

Single element or scalar values such as *First* or *Count*, *ToArray, ToList, ToDictionary, ToLookup*

Questions - Basics

**What is LINQ?**

A language feature that enables us to write type safe queries over any collection that implements IEnumerable<T>

**What inspired LINQ?**

The functional programming paradigm

**What are the basic elements**

* Sequences
* Elements
* Query operators
* Queries

**What do lambda expressions in query operators always operate on?**

Individual elements

**Do query operators alter the input sequence?**

No, they always generate a new sequence

**What does LINQ query comprise?**

A pipeline of operators that accept and return ordered sequences

**What does an SQL query comprise?**

A network of clauses working on unordered sets

## How LINQ works

We consider three aspects of the following query

1. Static structure of the decorator after query instantiation
2. Data Flow during query execution
3. Control flow during execution (Sequence diagram)

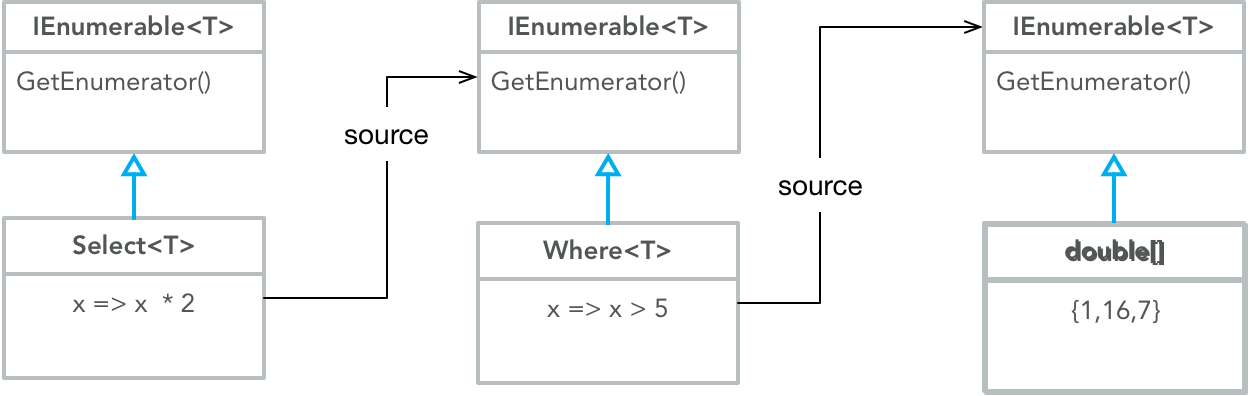
Consider the following simple LINQ query

var input = new [] {1,16,7};

var output = input.Where(i => i > 5)

.Select(i => i \* 2);

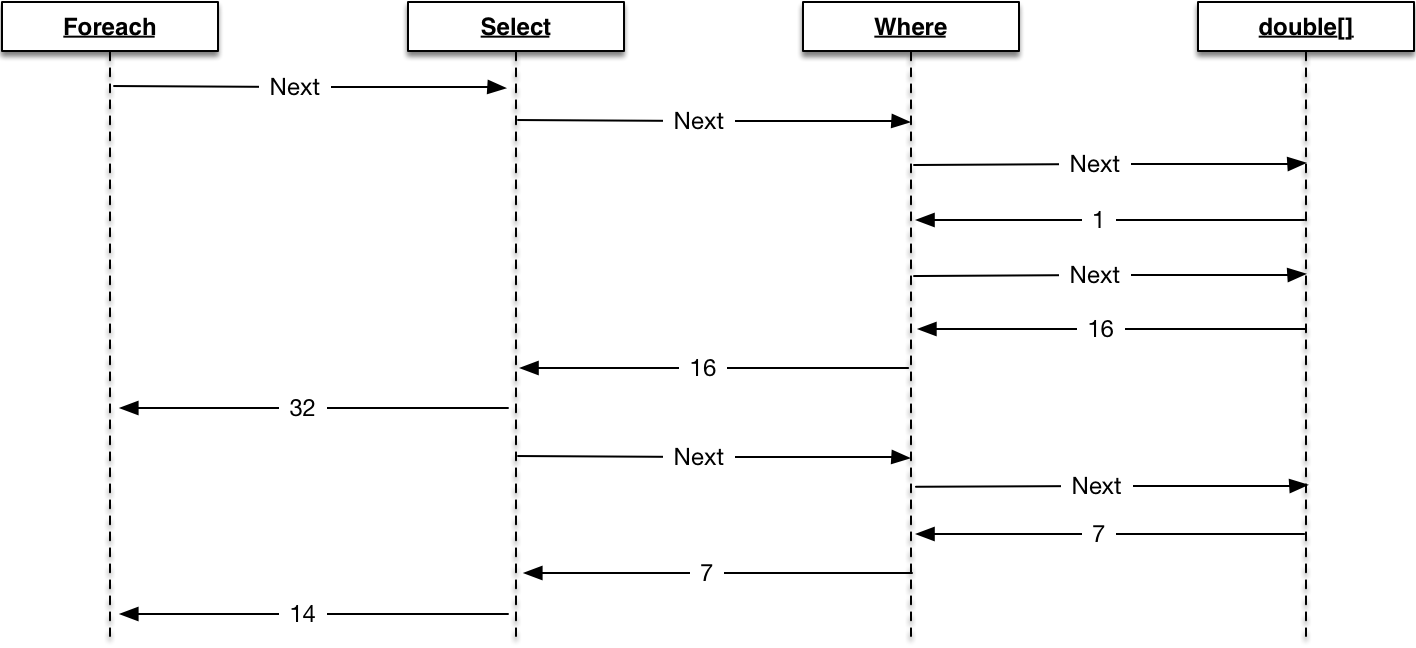
Figure 1 Static Structure On Creation



Listing 6 Data Flow On Execution



Figure 2 Control Flow On Execution



Questions – How LINQ Works

Query operators provide deferred execution by returning decorator sequences.

**What are the advantages of deferred execution?**

* Decouples construction from execution
* Allows one to construct a query in multiple steps
* You can re-evaluate a query by enumerating it again.

**What are the exceptions that return immediately?**

ToList, ToArray, ToDictionary, ToLookup

Single element or scalar operators such as First or Count

**How do decorator sequences differ from traditional collection classes?**

In general a decorator sequence has no storage of its own to store elements

**What does it have instead?**

A reference to another sequence supplied at runtime

**What happens when you request data from a decorator?**

It must in turn ask for data from its wrapped input sequence

**What happens when you chain query operators?**

A chain of decorators are created

**What happens when you enumerate a query?**

You query the original input sequence transformed through a layering chain of decorators

**What happens if you call ToList() on query?**

The whole chain is collapsed into a single list

## Operator List

### Where

Listing 7 Where Implementation

public static IEnumerable<T> Where<T>(this IEnumerable<T> source,

Func<T,bool> predicate)

{

foreach (var element in source)

if (predicate(element))

yield return element;

}

public static IEnumerable<T> Where<T>(this IEnumerable<T> source,

Func<T,int, bool> predicate)

{

int i = 0;

foreach (var element in source)

if (predicate(element,i++))

yield return element;

}

Listing 8 Where examples

IEnumerable<string> sanat = new[] { "Äiti", "Suomi", "Ranta", "Isi", "Sisu", "Edessä" };

// Example one basic filtration {"Äiti", "Suomi","Isi","Sisu""}

IEnumerable<string> os1 = sanat.Where(e => e.Contains("i"));

WriteLine(os1); //

// Example two Index Filtration {"Äiti", "Ranta","Sisu""}

IEnumerable<string> os2 = sanat.Where((e, i) => i % 2 == 0);

### ToDictionary/ToArray/ToList/ToLookup

ToArray and ToList are self-explanatory. A lookup is superficially like a dictionary, but it is not the same. It allows multiple values to share the same key. So, it generates a structure that maps from key to a grouping.

### Take, Skip, TakeWhile, SkipWhile, Distinct

Listing Take,TakeWhile,SkipWhile,Distinct examples

IEnumerable<string> sanat = new[] {"Äiti", "Suomi","Ranta","Isi","Sisu","Edessä"};

WriteLine(sanat.Take(2)); // {"Äiti", "Suomi"}

WriteLine(sanat.Skip(2)); // {"Ranta","Isi","Sisu","Edessä"}

WriteLine(sanat.Take(2)); // {"Äiti", "Suomi"}

WriteLine(sanat.Skip(2).Take(2)); // {"Ranta","Isi"}

WriteLine(sanat.Skip(4).Take(2)); // {"Sisu","Edessä"}

WriteLine(sanat.TakeWhile(e => e.Contains("i"))); // {"Äiti", "Suomi"}

WriteLine(sanat.SkipWhile(e => e.Contains("i"))); // {"Ranta","Isi","Sisu","Edessä"}

WriteLine(new[] { "Moi", "MOI", "MoI"}.Distinct(StringComparer.InvariantCultureIgnoreCase)); // {Moi}

### Select

#### Implementing SELECT

Listing 10 Select Implementation

public static IEnumerable<TResult> Select<TIn,TResult>(this IEnumerable<TIn> source, Func<TIn,TResult> project)

{

foreach (var element in source)

yield return project(element);

}

public static IEnumerable<TResult> Select<TIn,TResult>(this IEnumerable<TIn> source, Func<TIn,int, TResult> project)

{

int i = 0;

foreach (var element in source)

yield return project(element,i++);

}

#### The basic case

Select is a projection operator. It maps input sequences onto output sequences. The output sequences have the same number of elements as the input sequences.

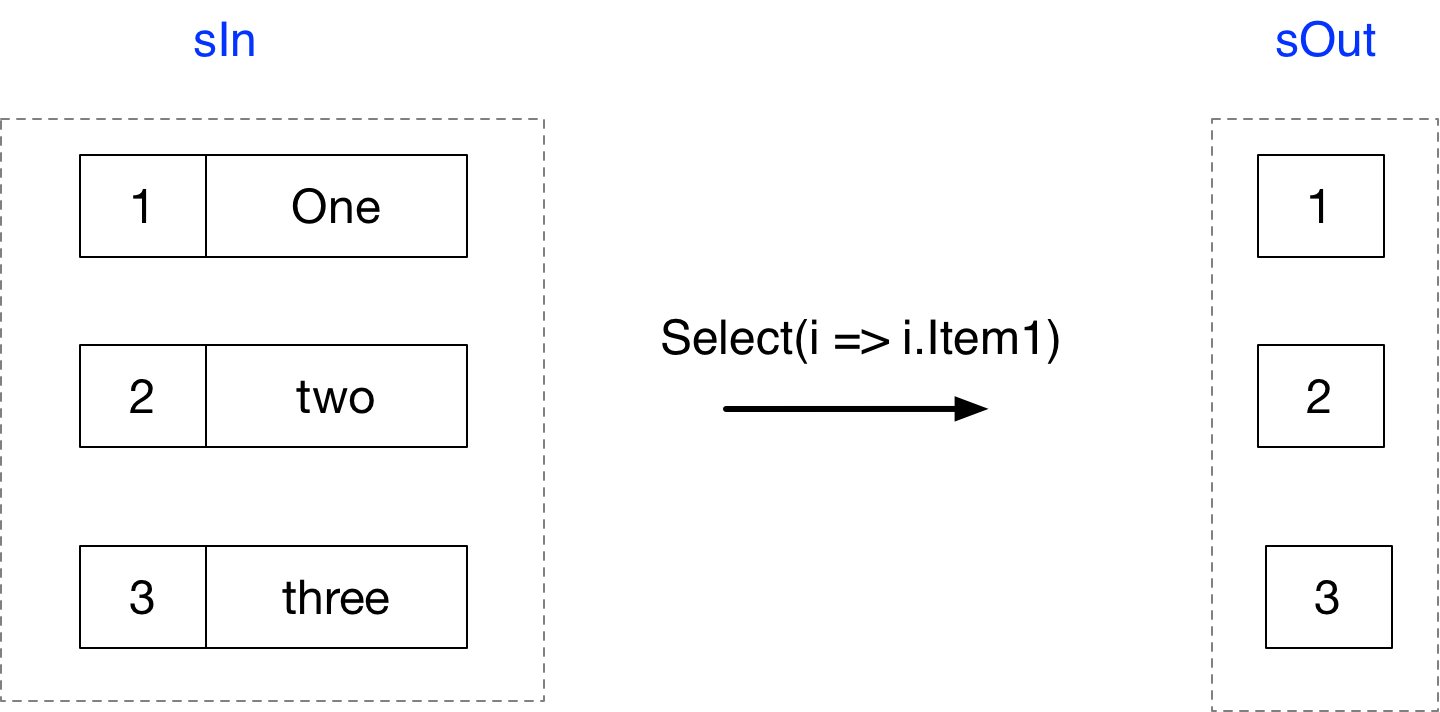
Listing Select – Basic Projection

IEnumerable<(int, string)> sIn =

new[] { (1, "one"), (2, "two"), (3, "three") };

// Projecting using fluent syntax

IEnumerable<int> sOut = sIn.Select(i => i.Item1);



#### Correlated Subqueries

We can insert subqueries inside select clauses. If the subquery references the elements of the outer query we call the query a correlated sub query.

Listing Correlated Subquery

IEnumerable<string> sIn =

new[] { "Hello World", "Moi Vaimoni", "Miten menee" };

// Correlated Subquery in fluent syntax

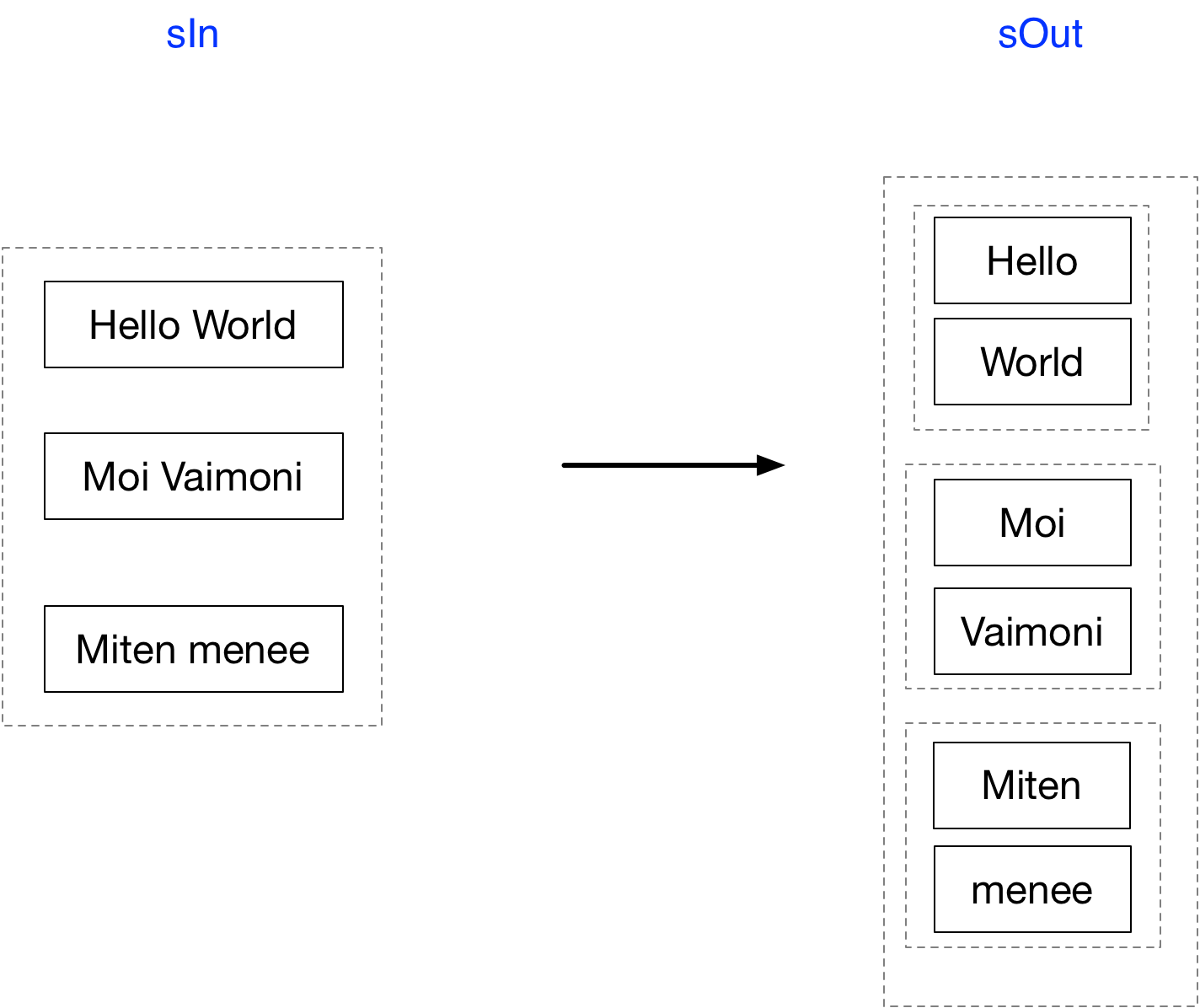
IEnumerable<IEnumerable<string>> sOut =

sIn

.Select(i => i

.Split()

.Select(j => j));



#### Cross Product (Un-flattened)

Uncorrelated queries mean when the sub query does not reference the elements of the outer query. Using select this generates a kind of hierarchical cross product type result. It is not quite a cross product as per a relational database as there is no flattening

Listing Uncorrelated Subqueries - Cross Product

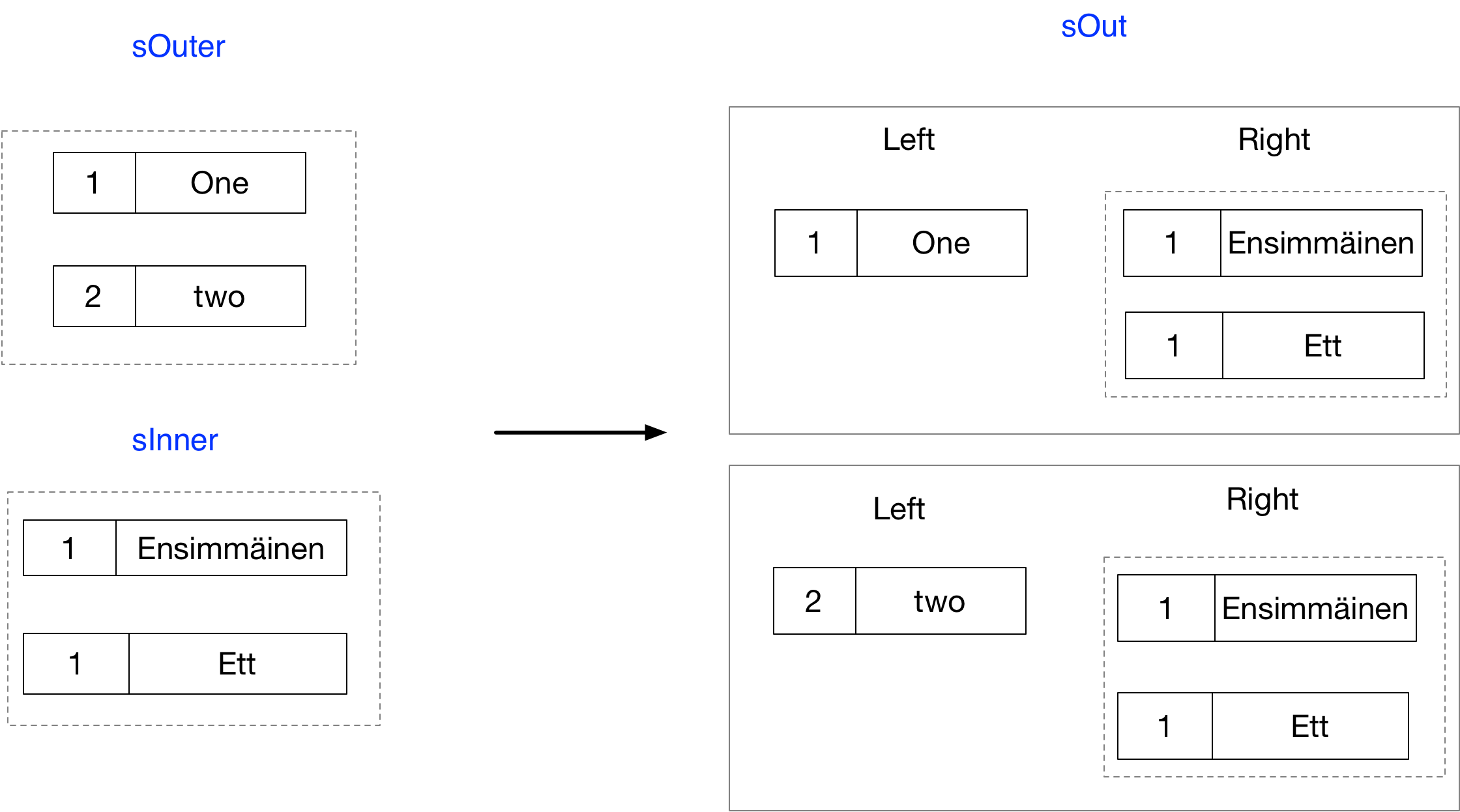
IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner

=new[] { (1, "Ensimmäinen"), (1, "Ett") };

IEnumerable<((int, string), IEnumerable<(int, string)>)> sOut2 =

sOuter.Select(i => (Left: i, Right: sInner.Select(j => j)));



#### Left Outer Join (Un-flattened)

The following code shows how to perform a left outer join style query using select. Unlike a relational database join the data in not flattened. We will show how to flatten the data in the section on SelectMany. This is an inefficient way to perform a left equi-join. In the section on GroupJoin we will show how to achieve this result in a more efficient means using GroupJoin which uses a lookup internally to prevent multiple linear traversals of the inner sequence.

Listing Left Outer Join

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

IEnumerable<((int, string) Left, IEnumerable<(int, string)> Right)> sOut

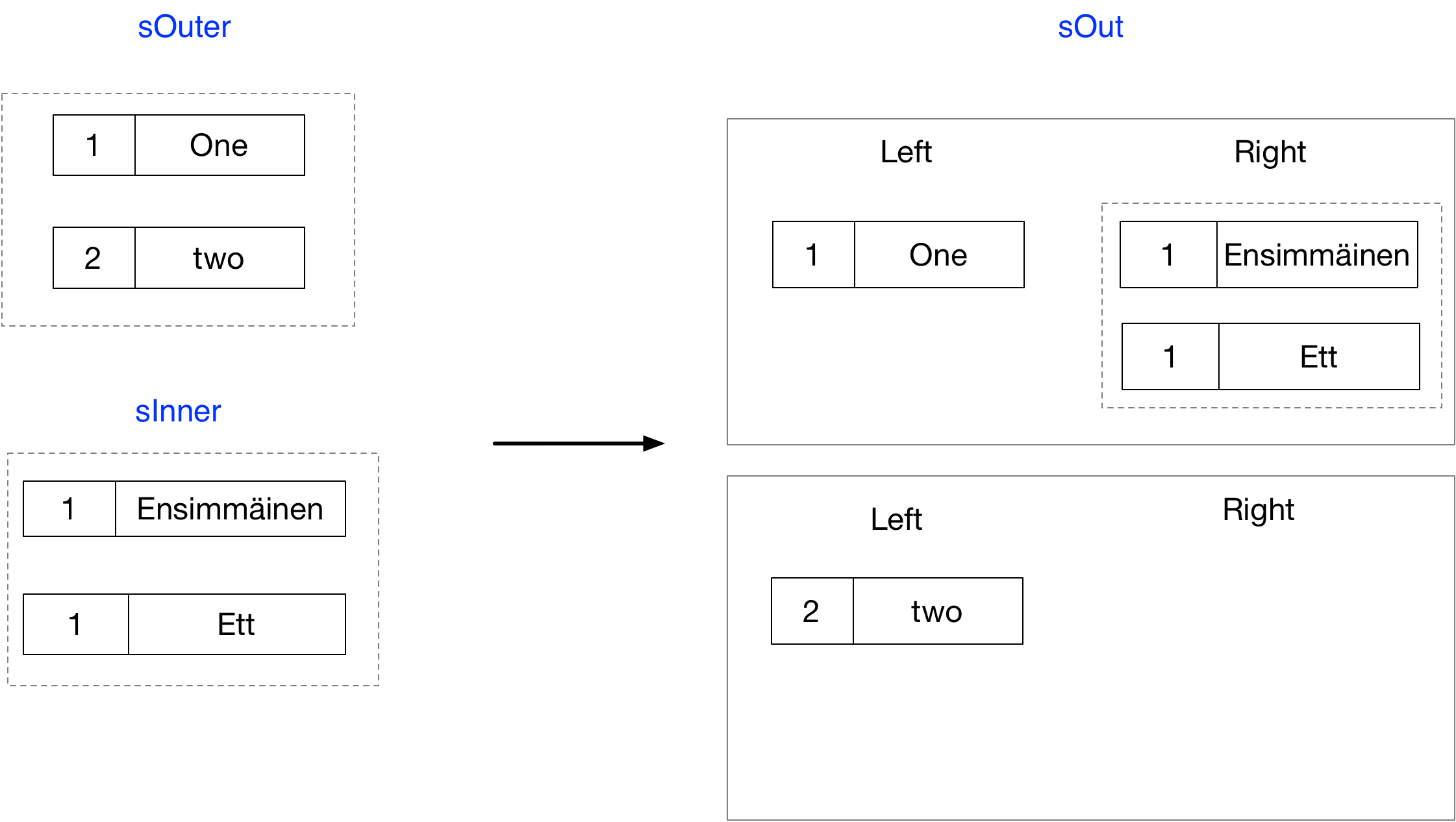
= sOuter.Select(outerEl =>

{

return (Left: outerEl,

**Right: sInner.Where(innerEl => outerEl.Item1 == innerEl.Item1)**);

});



### SelectMany

#### Implementing SelectMany

public static IEnumerable<TR> SelectMany<TS, TR>(

this IEnumerable<TS> source,

Func<TS, IEnumerable<TR>> project)

{

foreach (var element in source)

foreach (var subelement in project(element))

yield return subelement;

}

public static IEnumerable<TR> SelectMany<TS,TC, TR>(

this IEnumerable<TS> source,

Func<TS, IEnumerable<TC>> collSelector,

Func<TS, TC, TR> resultSelector)

{

foreach (var sourceElement in source)

{

foreach (var subelement in collSelector(sourceElement))

{

yield return resultSelector(sourceElement, subelement);

}

}

}

public static IEnumerable<TResult> SelectMany<TIn, TResult>(

this IEnumerable<TIn> source,

Func<TIn, int, IEnumerable<TResult>> project)

{

int i = 0;

foreach (var element in source)

foreach (var subelement in project(element, i++))

yield return subelement;

}

SelectMany can be used to carry out the following

* Subsequence concatenation / flattening
* Cross Joins
* Non-equi joins
* Inner joins
* Left outer joins

We now consider each in turn

#### Flattening subsequence

Listing SelectMany - Flattening subsequence

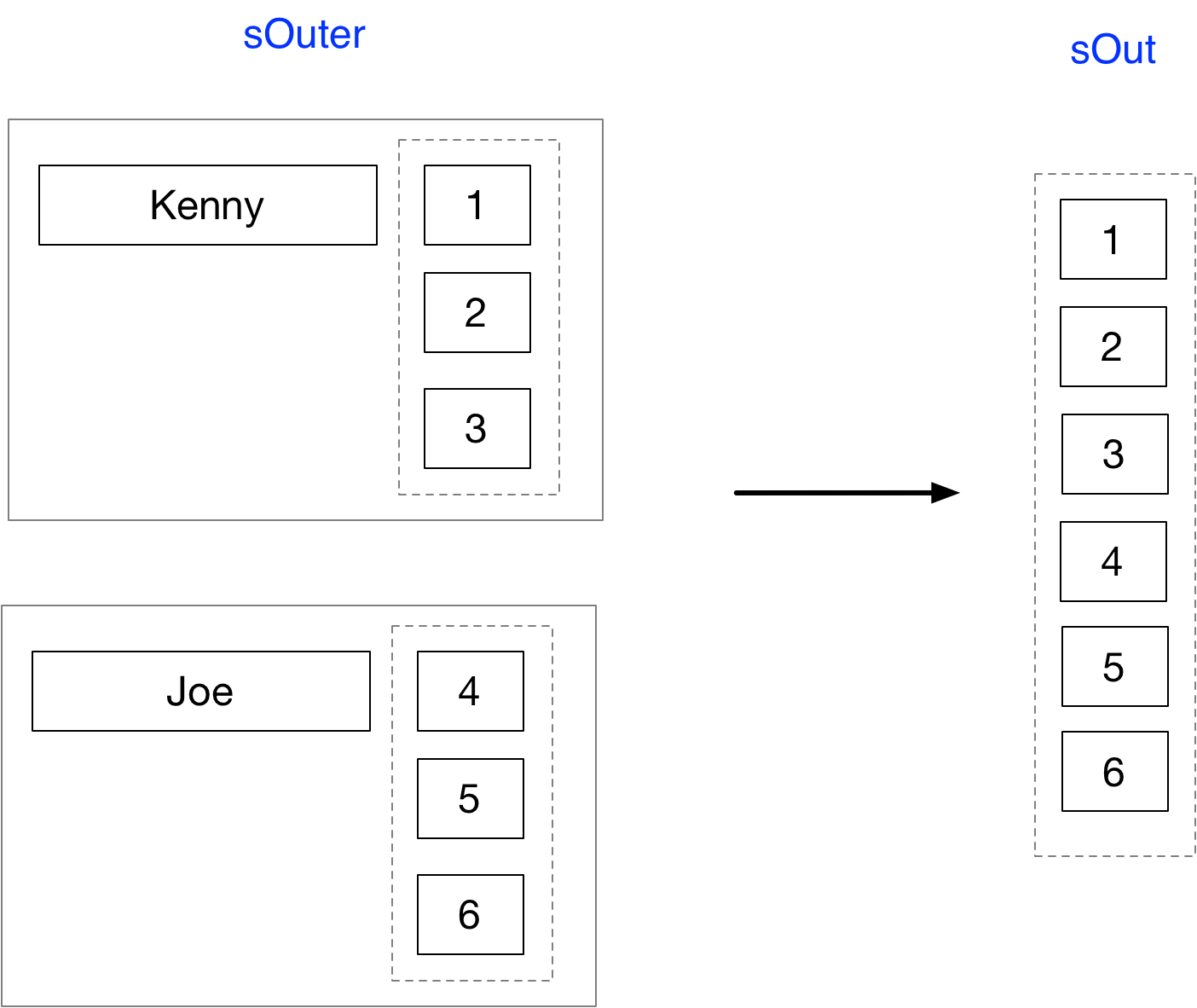
IEnumerable<(string, int[])> sIn =

new[] { ("Kenny", new[] { 1, 2, 3 }), ("Joe", new[] { 4, 5, 6 }) };

// Flattening/Concatenating subsequence using SelectMany - Fluent Syntax

IEnumerable<int> seq2 = sIn

.SelectMany(s => s.Item2);



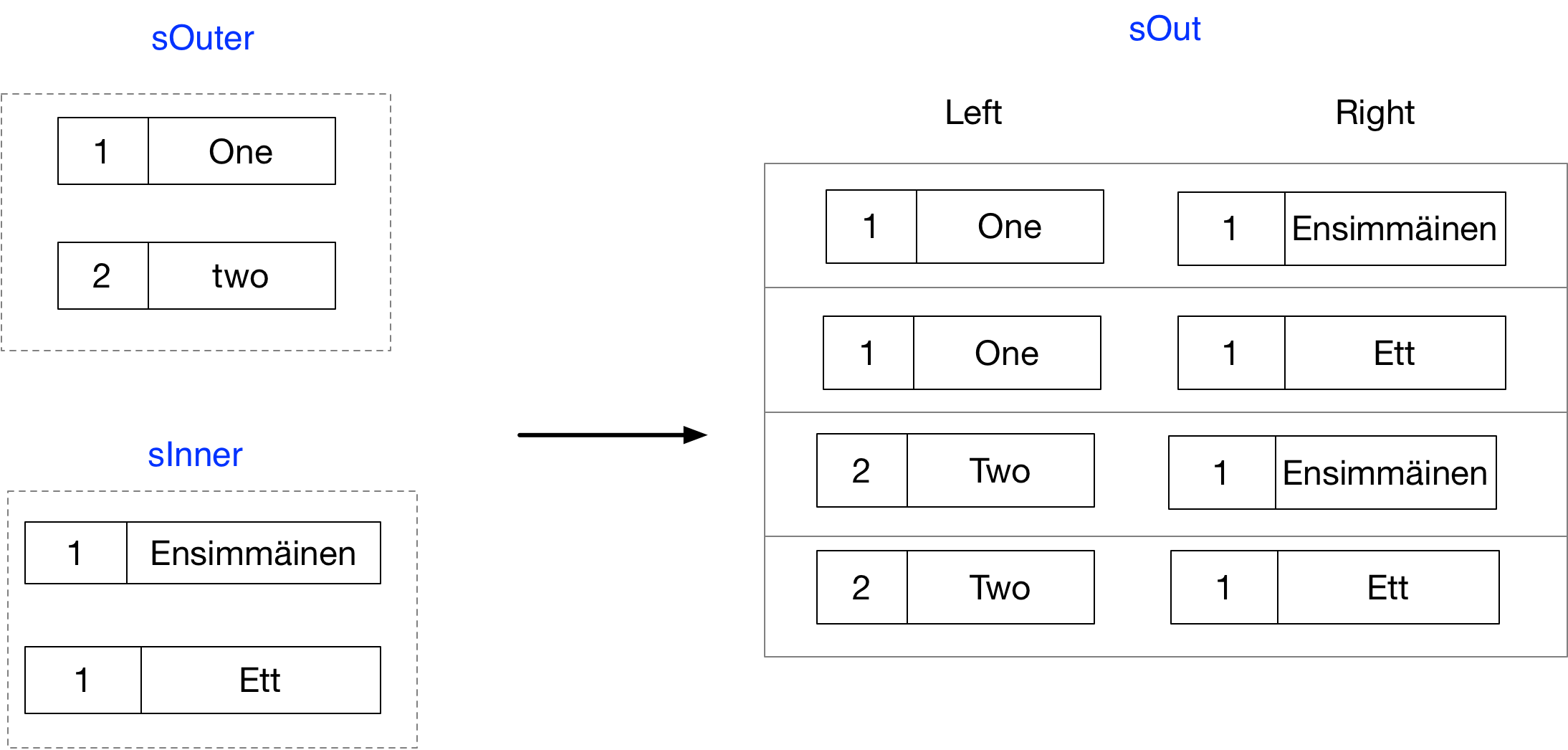
#### Cross Joins

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

IEnumerable<((int, string), (int, string))> sOut = sOuter

.SelectMany(elOut => sInner.Select(elIn => (Left:elOut, Right:elIn)));



#### I

#### Inner Join (Flattened)

The following performs an inner join using SelectMany. This code is inefficient as it requires a complete traversal of the inner sequence for each element of the outer sequence

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

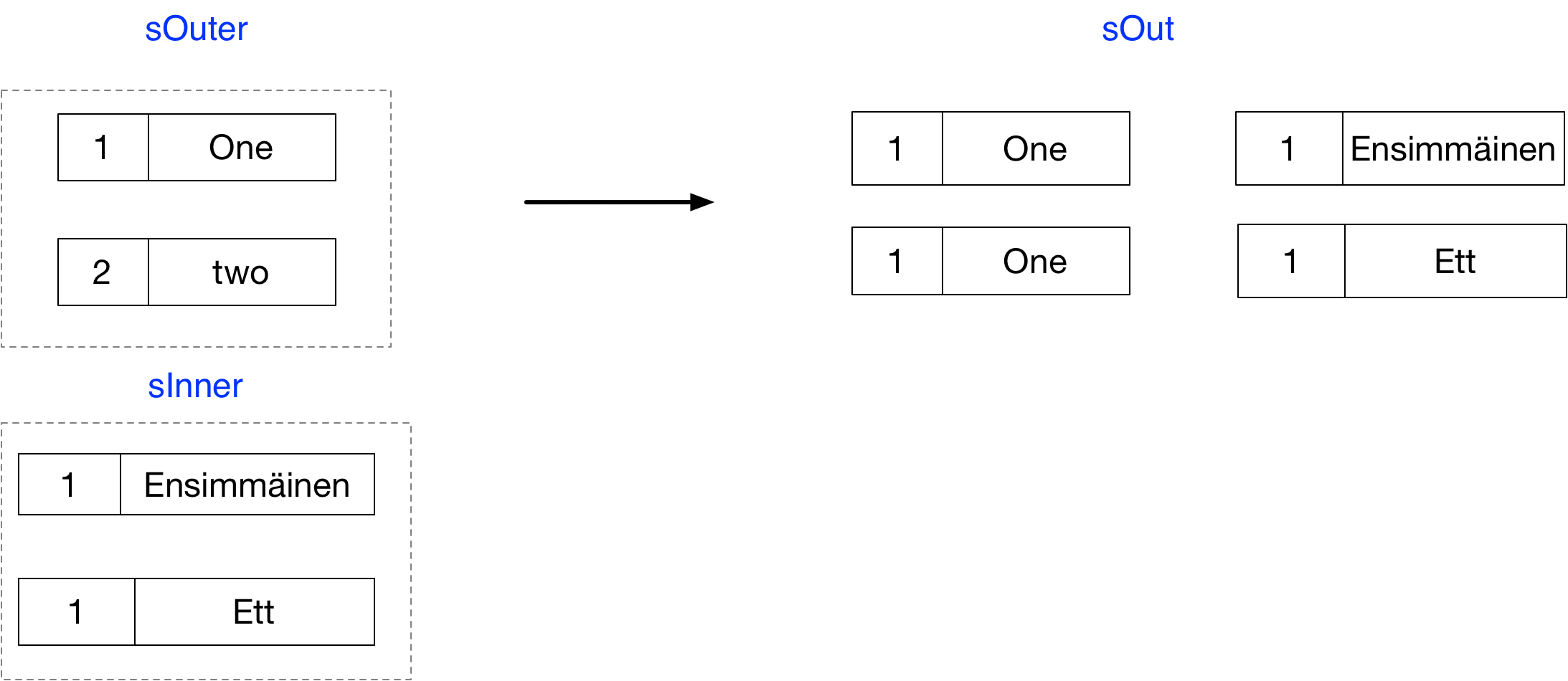
IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

var sOut = sOuter.SelectMany(outerEl =>

sInner

.Where(innerEl => outerEl.Item1 == innerEl.Item1)

.Select(inner => (outerEl, inner)));



### Join

Join carries out efficient, flattened inner joins on local object sequences

#### Implementing Join

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(

this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter,

TInner, TResult> resultSelector)

{

var lookup = inner.ToLookup(innerKeySelector);

foreach (var outerEl in outer)

foreach (var innerEl in lookup[outerKeySelector(outerEl)])

yield return resultSelector(outerEl, innerEl);

}

#### Inner Join (Flattened)

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

IEnumerable<((int, string), (int, string))> sOut =

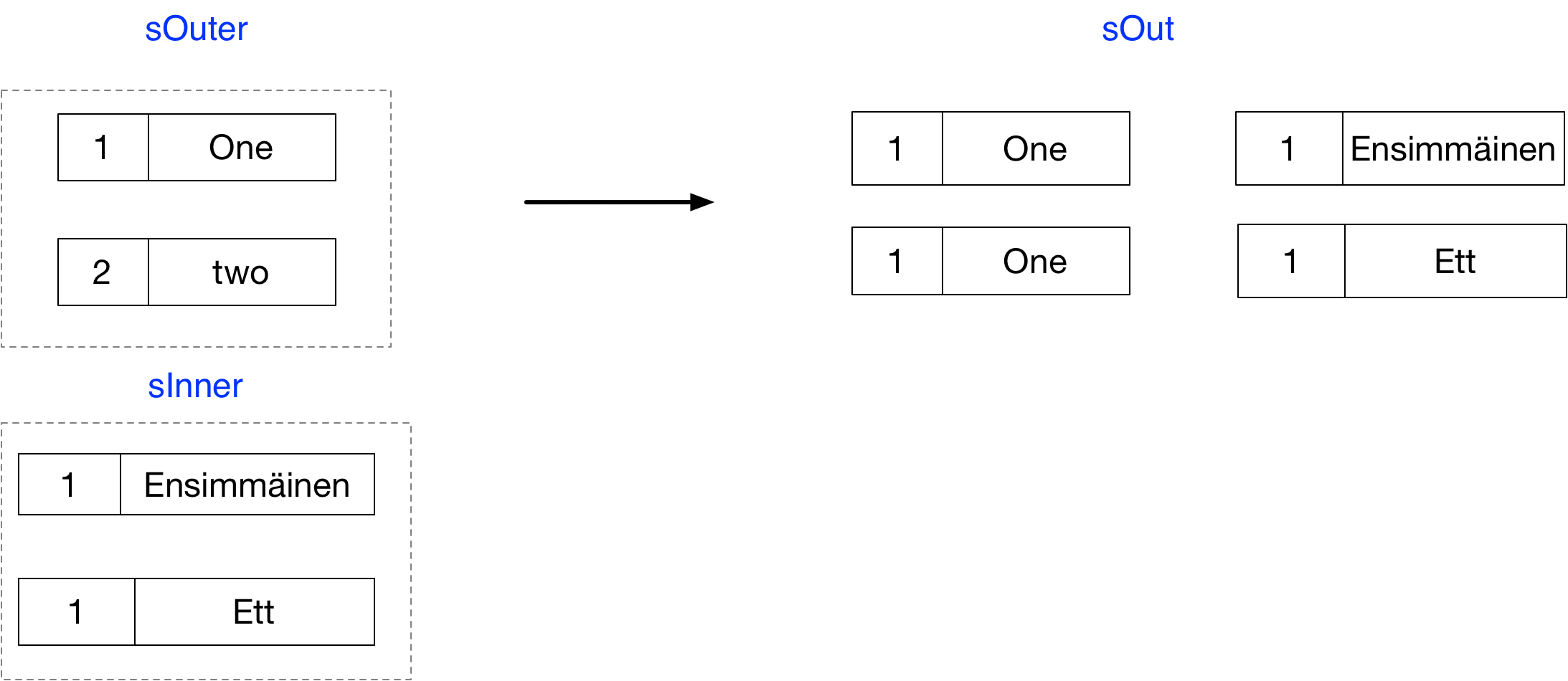
sOuter.Join(

sInner,

outerEl => outerEl.Item1,

innerEl => innerEl.Item1,

(outerEl, innerEl) => (Left:outerEl, Right:innerEl));



### GroupJoin

GroupJoin can be used to perform efficient inner and left-outer joins on local collections

#### Implementing GroupJoin

public static IEnumerable<TRes> GroupJoin<TRes, TOuter, TInner, TKey>(

IEnumerable<TOuter> sOuter,

IEnumerable<TInner> sInner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector,

Func<TOuter, IEnumerable<TInner>, TRes> resultSector

)

{

ILookup<TKey, TInner> lookup =

sInner.ToLookup(innerEl => innerKeySelector(innerEl));

foreach (TOuter outerEl in sOuter)

{

// Convert the element from the outer sequence to its key

TKey outerElKey = outerKeySelector(outerEl);

// Use the outer element key to lookup all matching

// inner sequence elements using the lookup for efficiency

IEnumerable<TInner> matchingInnerEls = lookup[outerElKey];

// Map the current outer element, a sequence of matching

// inner elementsto a single result element.

TRes result = resultSector(outerEl, matchingInnerEls);

yield return result;

}

}

#### Left Outer Join (Un-flattened)

This is much more efficient than using select as the

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

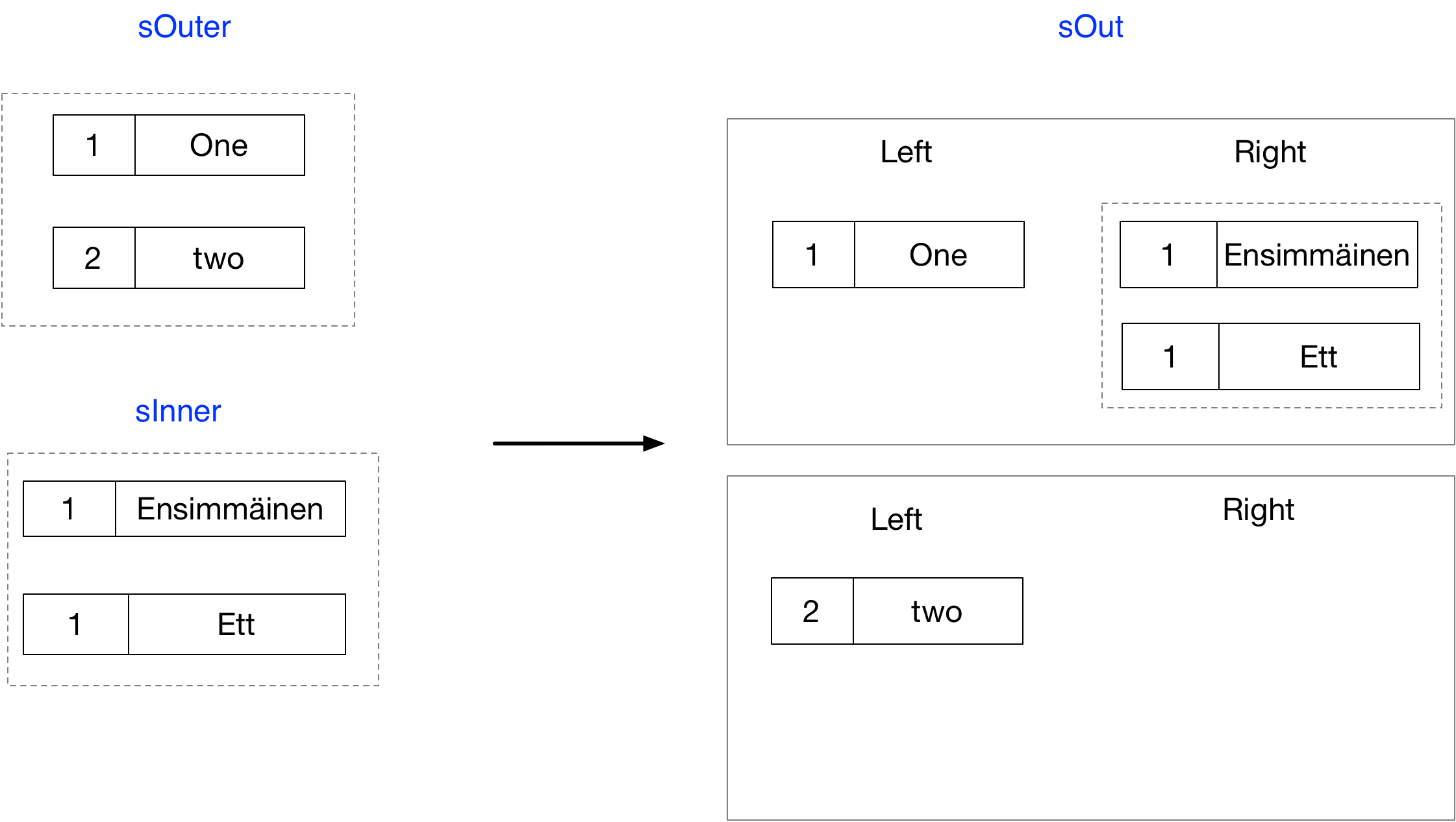
IEnumerable<((int, string), IEnumerable<(int, string)>)> sOut =

sOuter.GroupJoin(

sInner,

outerEl => outerEl.Item1,

innerEl => innerEl.Item1,

(outerEl, innerMatches) => (Left: outerEl, RightMatches: innerMatches));

#### Inner Join (Un-flattened)

To perform an inner join we filter the results of the previous section

IEnumerable<(int, string)> sOuter = new[] { (1, "one"), (2, "two"), };

IEnumerable<(int, string)> sInner = new[] { (1, "Ensimmäinen"), (1, "Ett") };

IEnumerable<((int, string), IEnumerable<(int, string)>)> sOut =

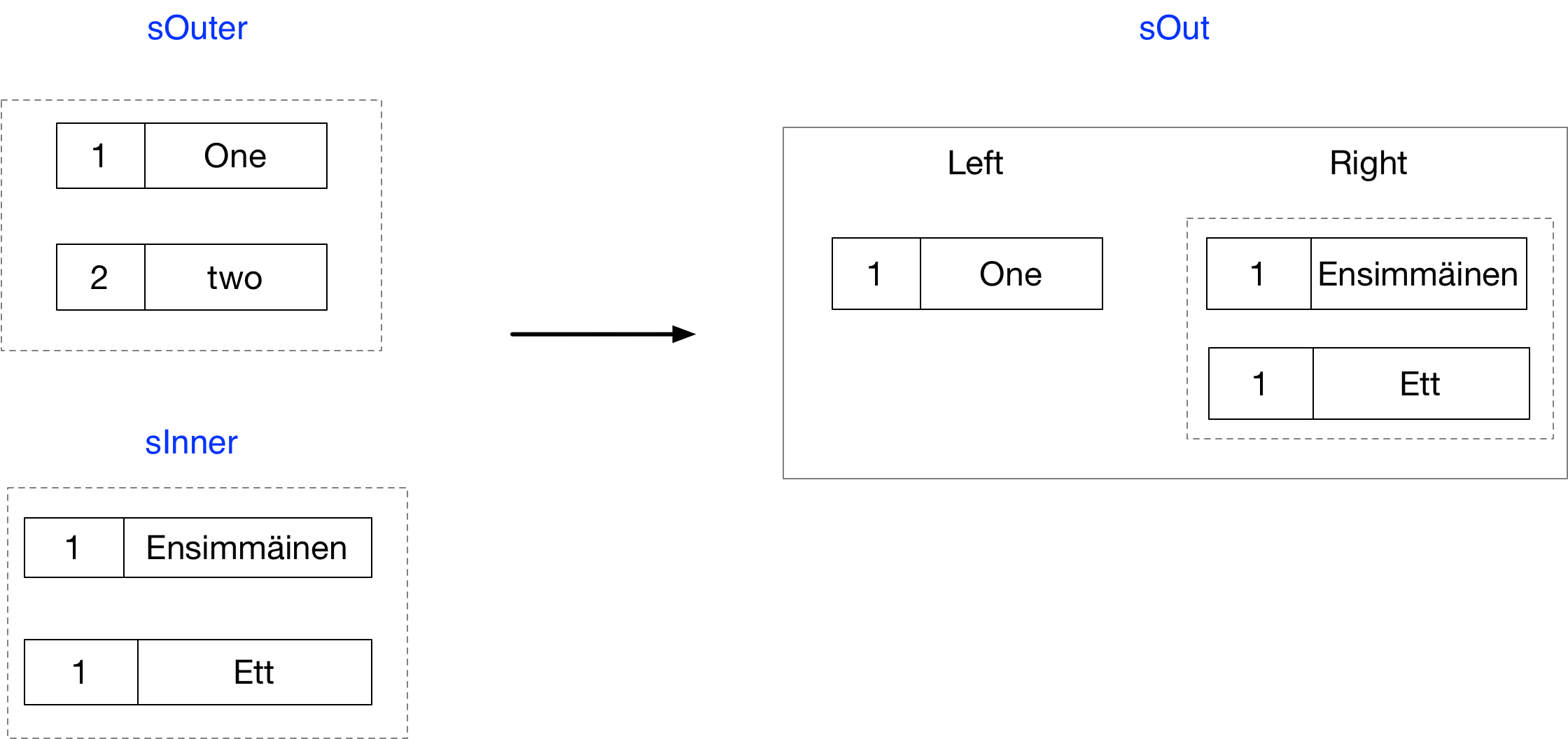
sOuter.GroupJoin(

sInner,

outerEl => outerEl.Item1,

innerEl => innerEl.Item1,

(outerEl, innerMatches) => (Left: outerEl, RightMatches: innerMatches))

**.Where(resultEl => resultEl.RightMatches.Any())**

#### Left Outer Join Flattened

Join enables us to do flattened inner joins. Group Join givens us hierarchical left outer joins. If we want to obtain flat left outer joins we need to combine GroupJoin with SelectMany as follows.

IEnumerable<(int, string)> sOuter = new[]

{ (1, "one"), (2, "two"), (3, "three"), (4, "four") };

IEnumerable<(int, string)> sInner = new[]

{ (1, "Ensimmäinen"), (1, "Ett"), (2, "Kaksi"), (2, "Tva"), (3, "Kolme") };

// Group Join gives us a hierarchical left outer join

IEnumerable<((int, string) OuterEl, IEnumerable<(int, string)> InnerEls)>

leftOuterHierarchical = sOuter.GroupJoin(

sInner,

outerEl => outerEl.Item1,

innerEl => innerEl.Item1,

(outEl, innerMatches) => (OutEl: outEl, InEls: innerMatches));

// SelectMany flattens.

IEnumerable<(string, string)> outSeq = leftOuterHierarchical

.SelectMany(hierarchicalEl => hierarchicalEl.InnerEls.DefaultIfEmpty(),

(hierarchicalEl, innerEl) =>

(hierarchicalEl.OuterEl.Item2, innerEl.Item2));

### OrderBy/ThenBy/OrderByDescencing/ThenByDescending

string[] seq = {"Four","Two", "One","Three"};

// {Two,One,Four,Three}

seq.OrderBy(s => s.Length);

// {Three,Four,Two,One}

seq.OrderByDescending(s => s.Length);

// {One,Two,Four,Three}

seq.OrderBy(s => s.Length).ThenBy(s => s);

// {Three,Four,Two,One}

seq

.OrderByDescending(s => s.Length)

.ThenByDescending(s => s)

### Aggregation

Aggregation can lead to some surprising results if we do not use a seed. Especially where we want to parallelize, we need a function that can combine sub results which is both commutative and associative

int[] s = new[] {2,3,4};

// 27 rather than 29

s.Aggregate ((x, y) => x+y\*y).Dump();

// Fix with seed

s.Aggregate (0,(x, y) => x+y\*y).Dump();

// For parallelisation we often specify a separate function

// for combining intermediate results. This function must be

// assocaite and commutative

s.AsParallel().Aggregate (()=>0,(a,e) => a+e\*e,(a1,a2) => a1+a2,a=>a).Dump();

Operator List – Questions

C:\Users\rps\AppData\Local\MyLINQPad\linqpad\Queries\InterviewQuestions\Linq

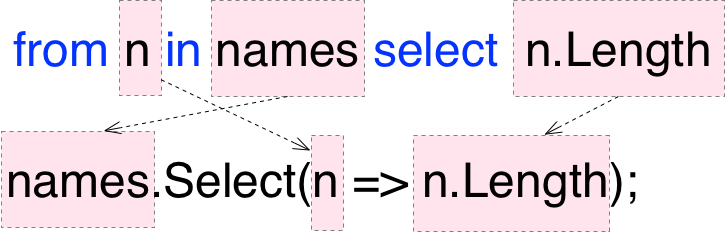
Single Element Operators

## Query Syntax

LINQ also supports an alternative syntax called query syntax. Query syntax supports only a subset of the LINQ operators.

### Range variables

The following shows how a very simple query syntax query is mapped by the compiler to a fluent query. Notice how the range variable n, defined in the query, maps to the left-hand side of the lambda expression in the generated fluent query. The expression to the right of select maps to the right-hand side of the lambda in the fluent query



### Transparent identifiers

Let allows a query syntax query to introduce a second range variable which keeping the original range variable in scope. The following query uses the let keyword to introduce a new range variable.

var names = new[] { "Wren", "Bill", "Bob", "Will" };

IEnumerable<(char,string)> s1 =

from n in names

**let c = n[0]**

orderby c

select (c,n);

Notice how the select clause can now reference two range variables. If we consider how we might translate this to fluent syntax we can see the compiler is doing some extra work for us.

IEnumerable<(char,string)> s2 =

names

.Select(n => **new {c=n[0], n=n}**)

.OrderBy(x => **x.c**)

.Select(x => (**x.c,x.n**));

Notice how we had to use an anonymous type to support the extra range variable **c**. This is what the compiler does, and it is known as transparent identifiers.

### Ordering

The following shows how to implement ordering using query syntax

IEnumerable<string> s1 =

from n in names

**orderby n.Length,n**

select n.ToUpper();

we can map this to fluent syntax as follows

IEnumerable<string> s2 =

names

**.OrderBy(n => n.Length)**

**.ThenBy(n => n)**

.Select(n => n.ToUpper());

If we want to order by in descending order we can use the following

IEnumerable<string> s3 =

from n in names

orderby n.Length **descending**, n **descending**

select n.ToUpper();

### SelectMany

We can generate a SelectMany in query syntax by using two from clauses.

IEnumerable<char> q1 =

**from n in names**

**from c in n**

select c;

The corresponding fluent syntax is as follows

IEnumerable<char> f1 =

**names.SelectMany(n => n)**;

If, however we have anything after the select clause in the query expression the compiler uses a transparent identifier to make both range range variables available for subsequent queries

IEnumerable<char> q2 =

from n in names

from c in n

where n == "Kenny" && c == 'n'

select Char.ToUpper(c);

The following shows how we can do this with anonymous types as the compiler might with transparent identifiers

IEnumerable<char> f2 =

names

.SelectMany(n => n, (n, c) => **new {n,c}**)

.Where(**x** => **x**.n=="Kenny" && **x**.c=='n')

.Select(**x** => Char.ToUpper(**x**.c));

### Join

We can join two sequences using the keyword join as follows

var s1 =

from ol in outerSeq

**join il in innerSeq on ol.Item1 equals il.Item1**

select (ol.Item2,il.Item2);

This can be translated into a fluent query as follows. Notice the final select (projection) is mapped directly to the projection function argument of Join

var f1 = outerSeq.Join(

innerSeq,

ol=>ol.Item1,

il=>il.Item1,

(ol,il) => (ol.Item2,il.Item2));

If, however we have anything after the select clause in the query expression the compiler uses a transparent identifier to make both range variables available for subsequent queries

var s2 =

from ol in outerSeq

join il in innerSeq on ol.Item1 equals il.Item1

where ol.Item2 == "one" && il.Item2 == "Ett"

select (ol.Item2, il.Item2);

The following shows how we can express this using anonymous types in a similar fashion to what the compiler would do with transparent identifiers.

var f2 = outerSeq

.Join(

innerSeq,

ol => ol.Item1,

il => il.Item1,

(ol, il) => **new {ol,il}**)

.Where(x=>**x**.ol.Item2 == "one" && **x**.il.Item2 == "Ett")

.Select(x => (**x**.ol.Item2, **x**.il.Item2));

### GroupJoin

We can specify a group join using query syntax as follows. We specify an into clause directly after the join clause

Into and GroupJoin or Query Continuation

If a query syntax query contains an into clause directly after a join clause it is translated as a GroupJoin. After select or group it causes query continutation which is quite different

var s1 =

from ol in outerSeq

**join il in innerSeq on ol.Item1 equals il.Item1**

**into matches**

select (ol.Item2,matches);

We can translate this into fluent syntax as follows.

var f1 = outerSeq.GroupJoin(

innerSeq,

ol=>ol.Item1,

il=>il.Item1,

(ol,il) => (ol.Item2,il));

As with join if there is only a simple select after the group join the select is implemented simply as the projection expression passed to the GroupJoin operator. If there is anything else then the compiler has to use a transparent identifier.

var q2 =

from ol in outerSeq

join il in innerSeq on ol.Item1 equals il.Item1

into matches

where ol.Item2 == "one" && matches.Count() == 2

select (ol.Item2, matches);

The following shows how we can express this using anonymous types in a similar fashion to what the compiler would do with transparent identifiers.

var f2 = outerSeq

.GroupJoin(

innerSeq,

ol => ol.Item1,

il => il.Item1,

(ol, matches) => **new {ol,matches})**

.Where(**x**=>**x**.ol.Item2 == "one" && **x**.matches.Count() == 2)

.Select(**x** => (**x**.ol.Item2, **x**.matches));

### Left Outer Join

A left outer join is a little bit tricky. A join performs flattening but gives us an inner join. A group join gives us outer join like functionality without flattening. The solution is to use groupjoin together with a select many to flatten. The query syntax is as follows

IEnumerable<(string, string)> q1 =

from outerEl in outerSeq

join innerEl in innerSeq on outerEl.Item1 equals innerEl.Item1

into matches

from e in matches.DefaultIfEmpty()

select default((int, string)).Equals(e)

? (outerEl.Item2, "")

: (outerEl.Item2, e.Item2);

And the fluent query is as follows. Note we add a function to simply the result generation and default check.

(string, string) ResGenFunc((int, string) outerEl, (int, string) innerEl) =>

default((int, string)).Equals(innerEl)

? (outerEl.Item2, "")

: (outerEl.Item2, innerEl.Item2);

IEnumerable<(string, string)> f1 =

outerSeq

.GroupJoin( innerSeq,

oel => oel.Item1,

inel => inel.Item1,

(oel, inEls) => inEls

.DefaultIfEmpty()

.Select(innerEl => ResGenFunc(oel, innerEl)))

.SelectMany(el => el);

### Joining

#### Implementing Join

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(

this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter,

TInner, TResult> resultSelector)

{

var lookup = inner.ToLookup(innerKeySelector);

foreach (var outerEl in outer)

foreach (var innerEl in lookup[outerKeySelector(outerEl)])

yield return resultSelector(outerEl, innerEl);

}

#### Inner Joins

(int, string)[] outerSeq = { (1, "one"), (2, "two"), };

(int, string)[] innerSeq = { (1, "Ensimmäinen"), (1, "Ett") };

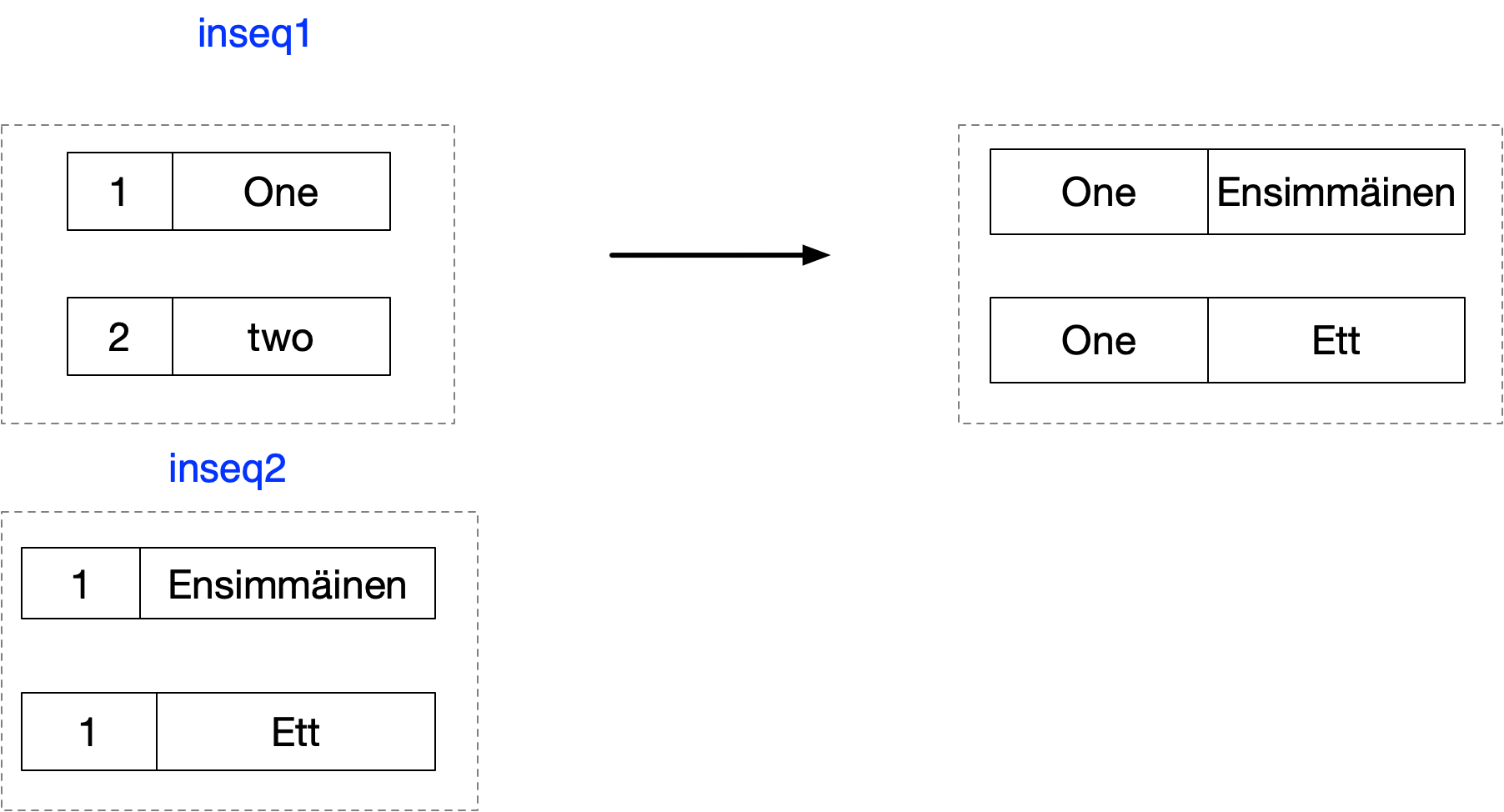
var res1 =

from outer in outerSeq

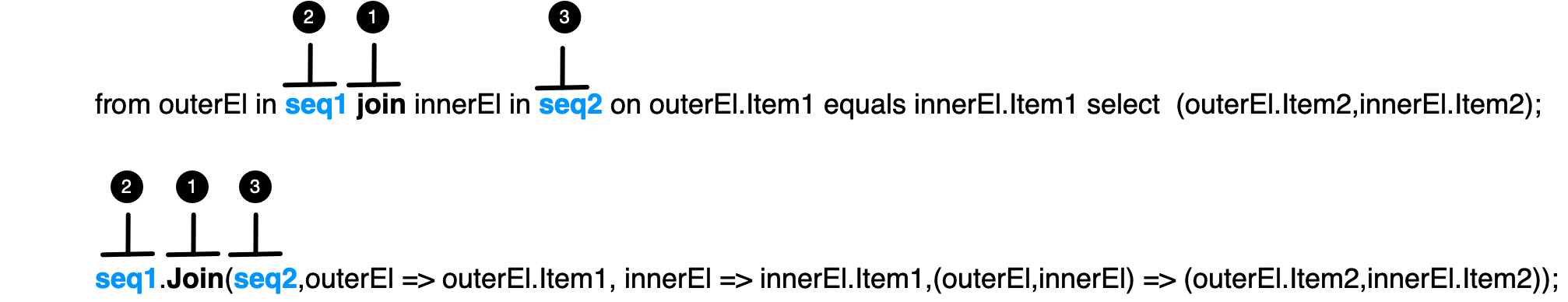
join inner in innerSeq

on outer.Item1 equals inner.Item1

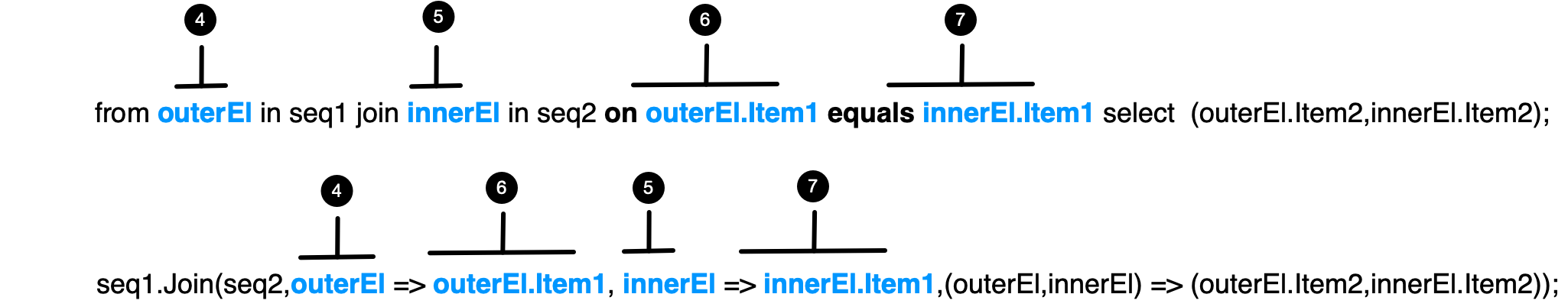
select (outer.Item2,inner.Item2);



Let us consider how the query with a join is mapped to the operator Enumerable.Join. First consider the sequences and the contextual keyword join. The keyword join instructs the compiler to invoke the Join operator on the sequence defined in the expression before the join keyword to the expresion defined after the in keyword.

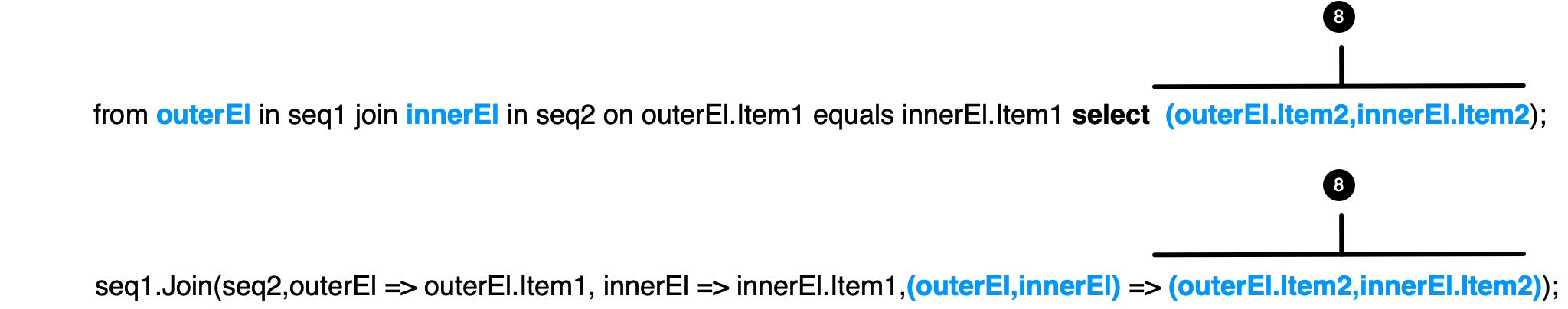


Then the keywords on and equals are used to define the keys of the outer an inner sequence respectively



The order of the key selectors is important. For the first key selector after the on keyword only the range varaible of the outer sequence is in scope. For the second key selector only the range variable of the second sequence is in scope.

Finally we consider how the query expression maps to the final projection function parameter of the join operator. In our simple query expression there is nothing after the join other than a simple select. In this case the selet is directly mapped into the selector function



In the case where we have something other than a simple select after the join the compiler has to work some **transparent idenfier** magic to make sure the elements from both the inner and outer sequences are available to operators after the join.

### Example One

Listing 16Query Syntax

var names = new List<String> (new[] { "Wren", "Bill", "Bob", "Will" });

IEnumerable<string> s1 =

from n in names

where n.StartsWith("W")

orderby n

select n.ToUpper();

Listing 17 Fluent Syntax

IEnumerable<string> s2 = names

.Where(n => n.StartsWith("W"))

.OrderBy(n => n)

.Select(n => n.ToUpper());

### Example Two

This example shows a situation where query syntax is more elegant than the corresponding fluent syntax query. We have two range variabes in the query syntax query which remain in scope for subsequent clauses. Note how the orderby and select access both range variables. In this example we are flattening

Listing 18Query Syntax

IEnumerable<string> parit = new[] { "Minun äiti", "Suomi on Mun", "Iso ranta", "Sanan isi", };

// A SelectManyQuery which wants to access both the outer elements and the

// flattened inner elements can be easier to write in query syntax.

from p in parit

from s in p.Split()

where s.Contains("u")

orderby p,s

select $"{p} -> {s}"

Listing 19 Fluent Syntax

parit

.SelectMany(pari => pari.Split().Select(sana => ( pari, sana)))

.Where( x=> x.sana.Contains("u"))

.OrderBy(x=>x.pari)

.OrderBy(x=>x.sana)

.Select(x => $"{x.pari} -> {x.sana}")

## Subqueries

## Projecting

## Subqueries

We can write quite inefficient queries using subqueries in LINQ

string[] names = new[] {"Kenny", "John", "Bob", "Jimmy","Rob"};

var enumerable1 = from n in names

where n.Length == names.Min(s => s.Length)

select n;

We can make this code much more efficient by restructuring as follows

var min = names.Min(s => s.Length);

IEnumerable<string> enumerable2 = from n in names

where n.Length == min

select n;

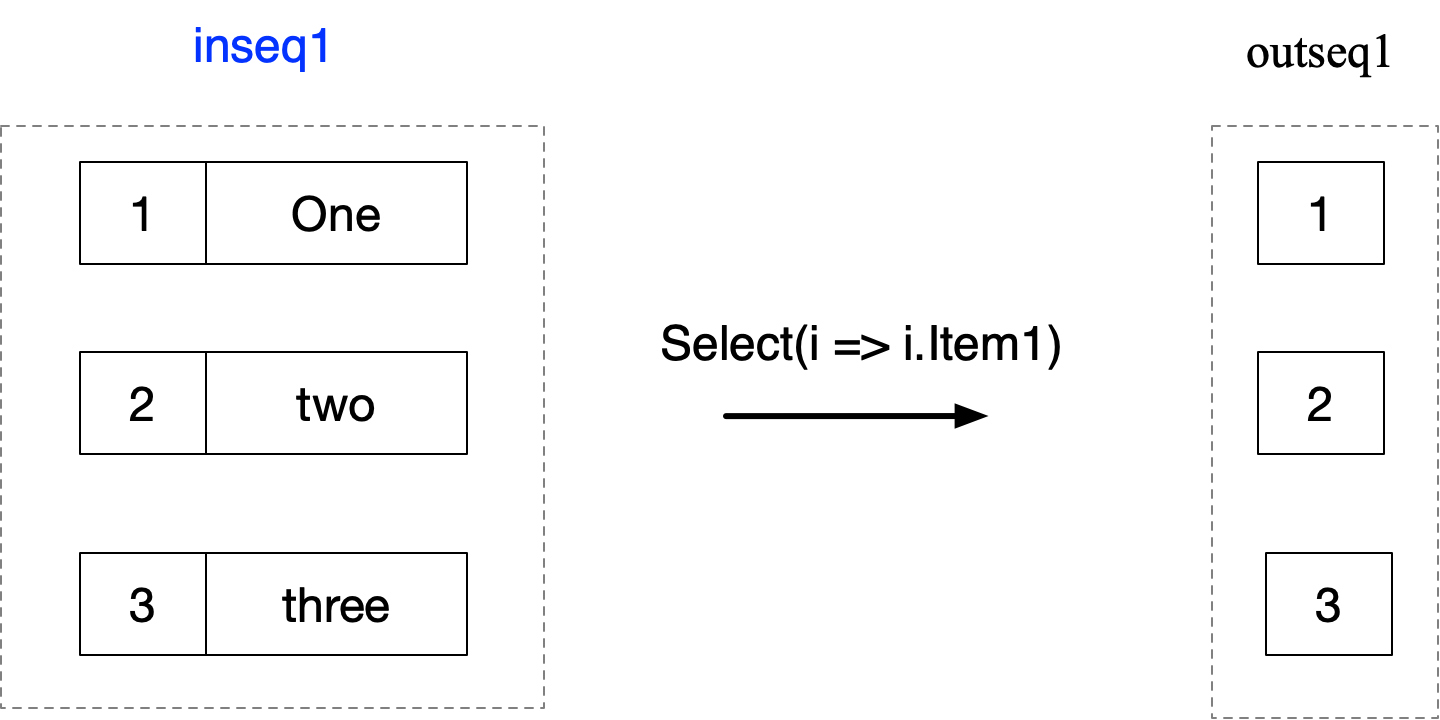
Questions – Queries

C:\Users\rps\AppData\Local\MyLINQPad\linqpad\Queries\InterviewQuestions\Linq\04 Questions and Answers

Basic Projection

Write query and fluent syntax to perform the following?

var inseq1 = new [] {(1, "one"),(2, "two"),(3, "three")};



var f = inseq1.Select(i => i.Item1);

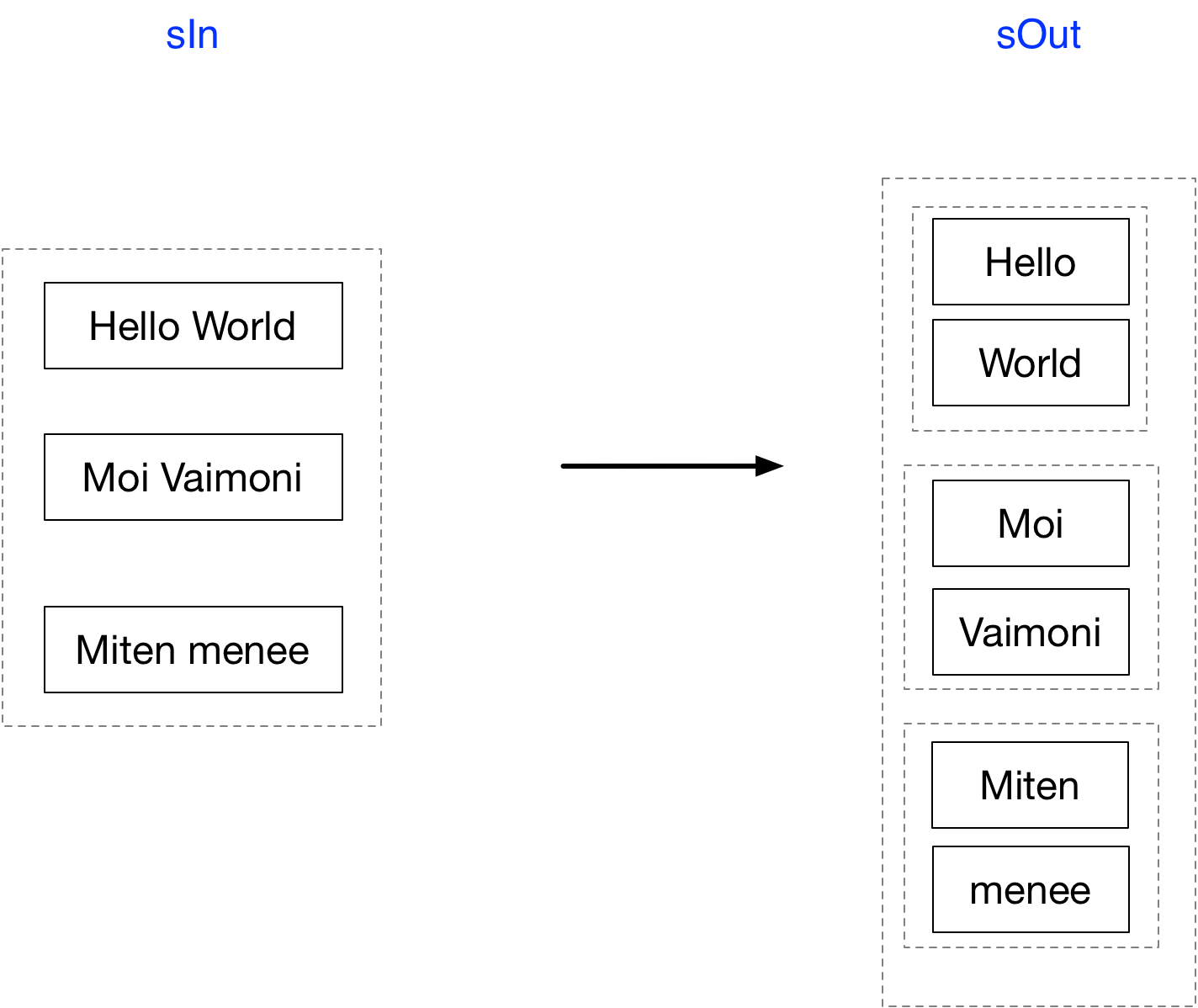
f.Dump();

var q = from n in inseq1 select n.Item1;

q.Dump();

Correlated Subquery

Write query and fluent syntax to perform the following?

var sIn = new[] { "Hello World", "Moi Vaimoni", "Miten menee"};

var f = sIn.Select(i => i.Split());

f.Dump();

var q =from n in sIn select n.Split();

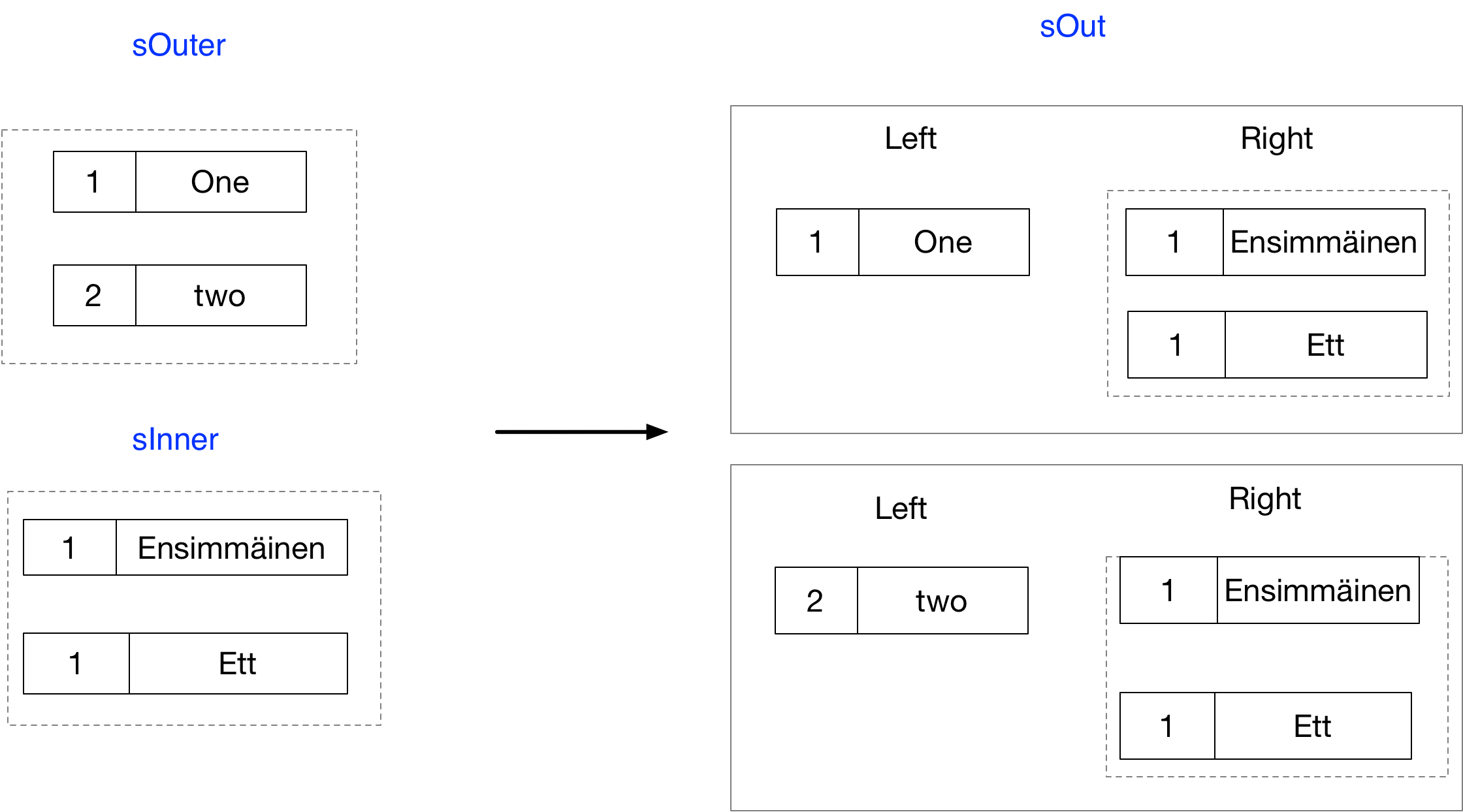
q.Dump();

Cross Product - Hierarchical

Write query and fluent syntax to perform the following?

var s1 = new[] { (1, "one"), (2, "two"), };

var s2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };



var f =s1

.Select(e1 => new {Left=e1, Right=s2});

var q = from e1 in s1

select new {Left=e1, Right=s2};

Group Join

Implement GroupJoin

public static IEnumerable<TRes> GroupJoin<TRes, TOuter, TInner, TKey>(

IEnumerable<TOuter> sOuter,

IEnumerable<TInner> sInner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector,

Func<TOuter, IEnumerable<TInner>, TRes> resultSector

)

{

ILookup<TKey, TInner> lookup =

sInner.ToLookup(innerEl => innerKeySelector(innerEl));

foreach (TOuter outerEl in sOuter)

{

// Convert the element from the outer sequence to its key

TKey outerElKey = outerKeySelector(outerEl);

// Use the outer element key to lookup all matching

// inner sequence elements using the lookup for efficiency

IEnumerable<TInner> matchingInnerEls = lookup[outerElKey];

// Map the current outer element, a sequence of matching

// inner elementsto a single result element.

TRes result = resultSector(outerEl, matchingInnerEls);

yield return result;

}

}

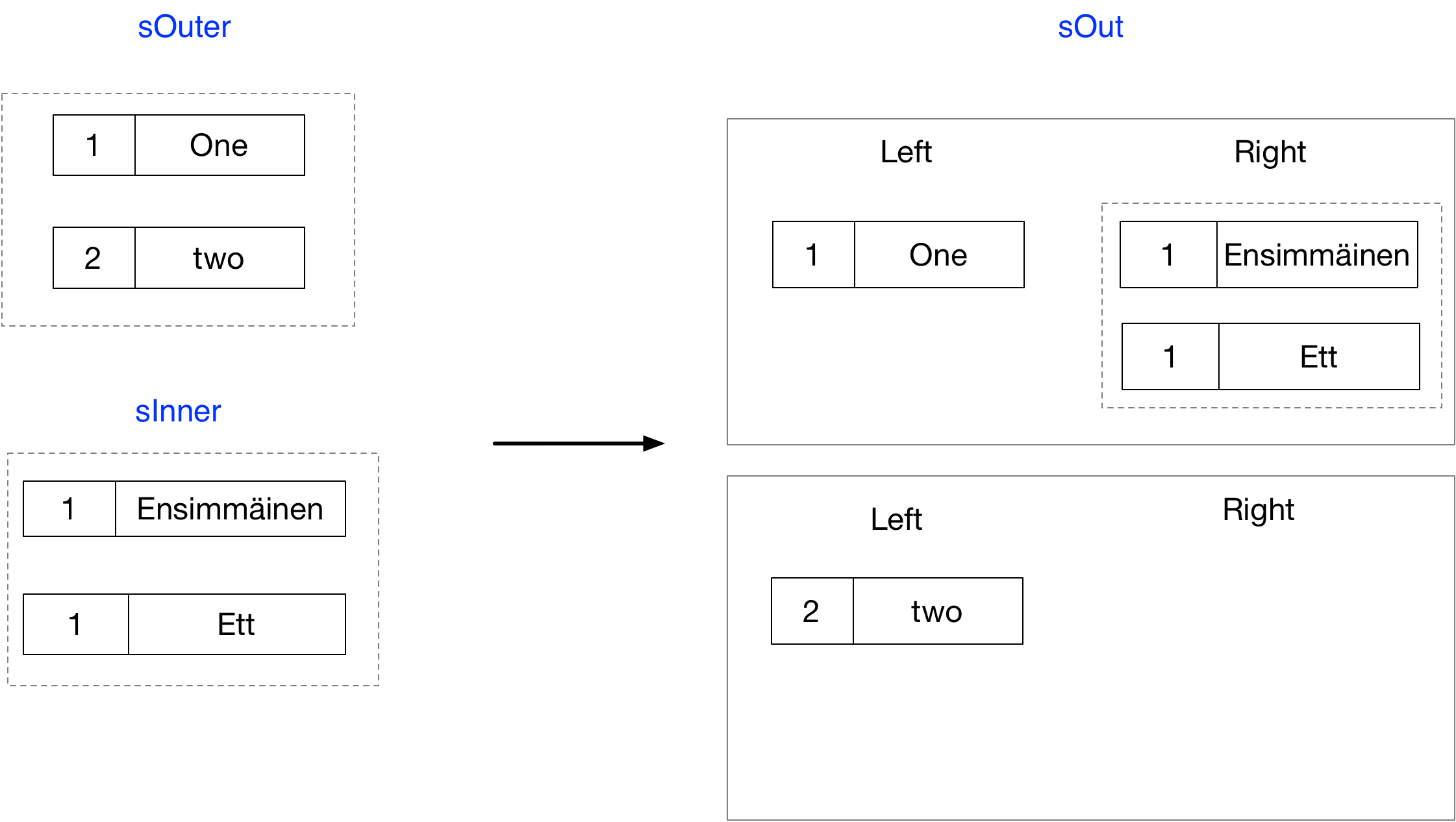
Correlated Hierarchical – Left Outer Join

Write query and fluent syntax to perform the following. Write

1. Inefficient select based queries
2. Efficient GroupJoin queries
3. Efficient select+lookup queries?

var s1 = new[] { (1, "one"), (2, "two"), };

var s2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };



// Inefficient because the inner query iterated once per outer element

var f1 = s1.Select(e1 => new { Left = e1, Right = s2.Where(e2 => e1.Item1 == e2.Item1)});

f1.Dump();

// Inefficient because the inner query iterated once per outer element

var q1 = from e1 in s1

select new {Left=e1, Right = from e2 in s2 where e1.Item1 == e2.Item1 select e2 };

q1.Dump();

// Efficient because internally Group Join uses a lookup

var f2 = s1.GroupJoin(s2,e1=>e1.Item1, e2=> e2.Item1,(e1,c) => new {Left=e1, Right =c });

f2.Dump();

// Efficient because internally Group Join uses a lookup

var q2 = from e1 in s1

join e2 in s2 on e1.Item1 equals e2.Item1

into collection

select new {Left=e1, Right =collection };

q2.Dump();

// Efficient - uses lookup

var lookup = s2.ToLookup(e2 => e2.Item1, e2=>e2);

var f3 = s1.Select(e1 => new {Left=e1, Right=lookup[e1.Item1]});

f3.Dump();

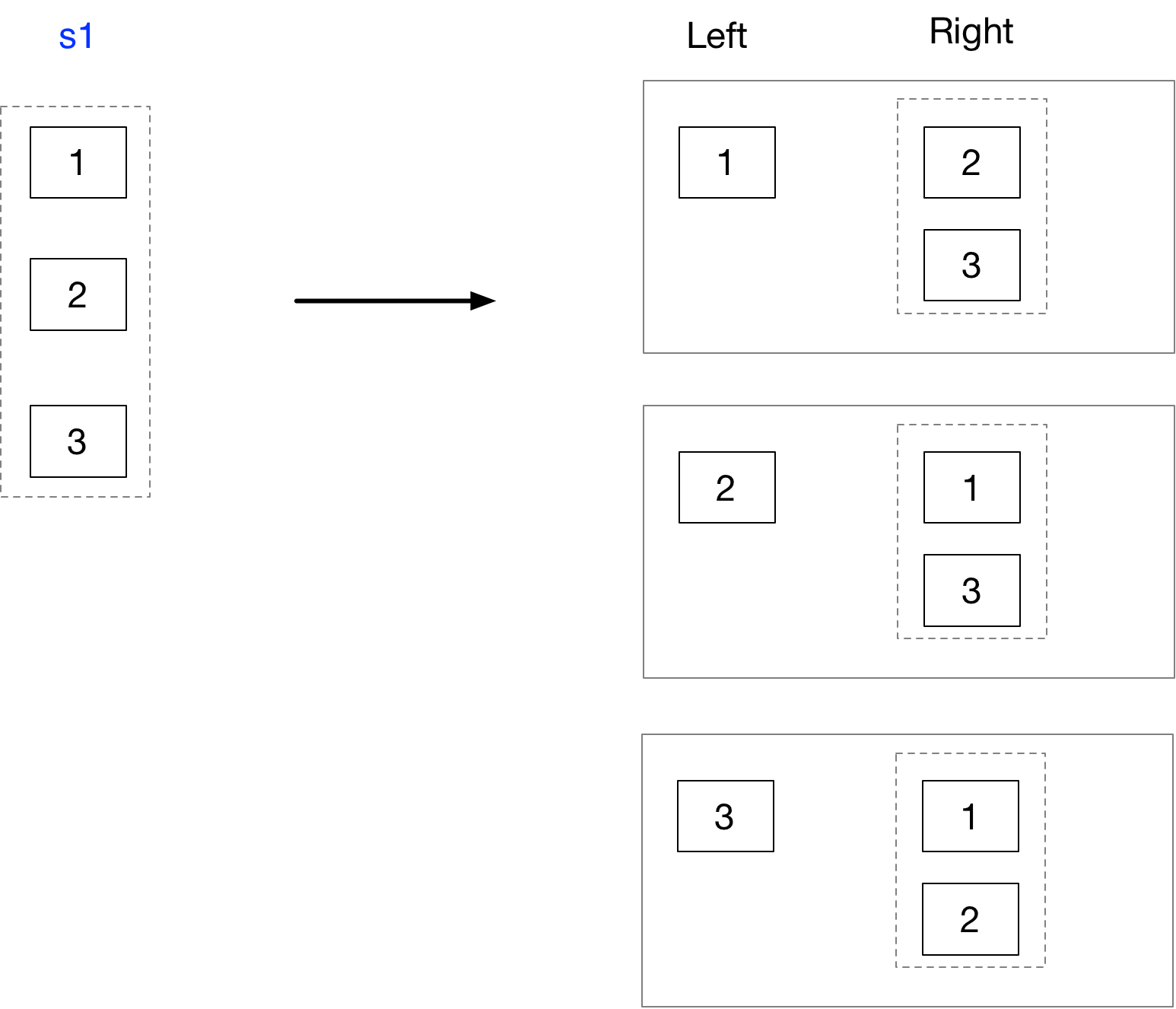
// Efficient - uses lookup

var q3 =from e1 in s1

select new {Left=e1, Right=lookup[e1.Item1]};

Non Equijoin – Hierarchical

Write query and fluent syntax to perform a non-equi join as follows?

var s1 = new[] { 1, 2, 3};

var q1 = from e1 in s1

select new {

Left=e1,

Right=from e2 in s1

where e2 != e1

select e2

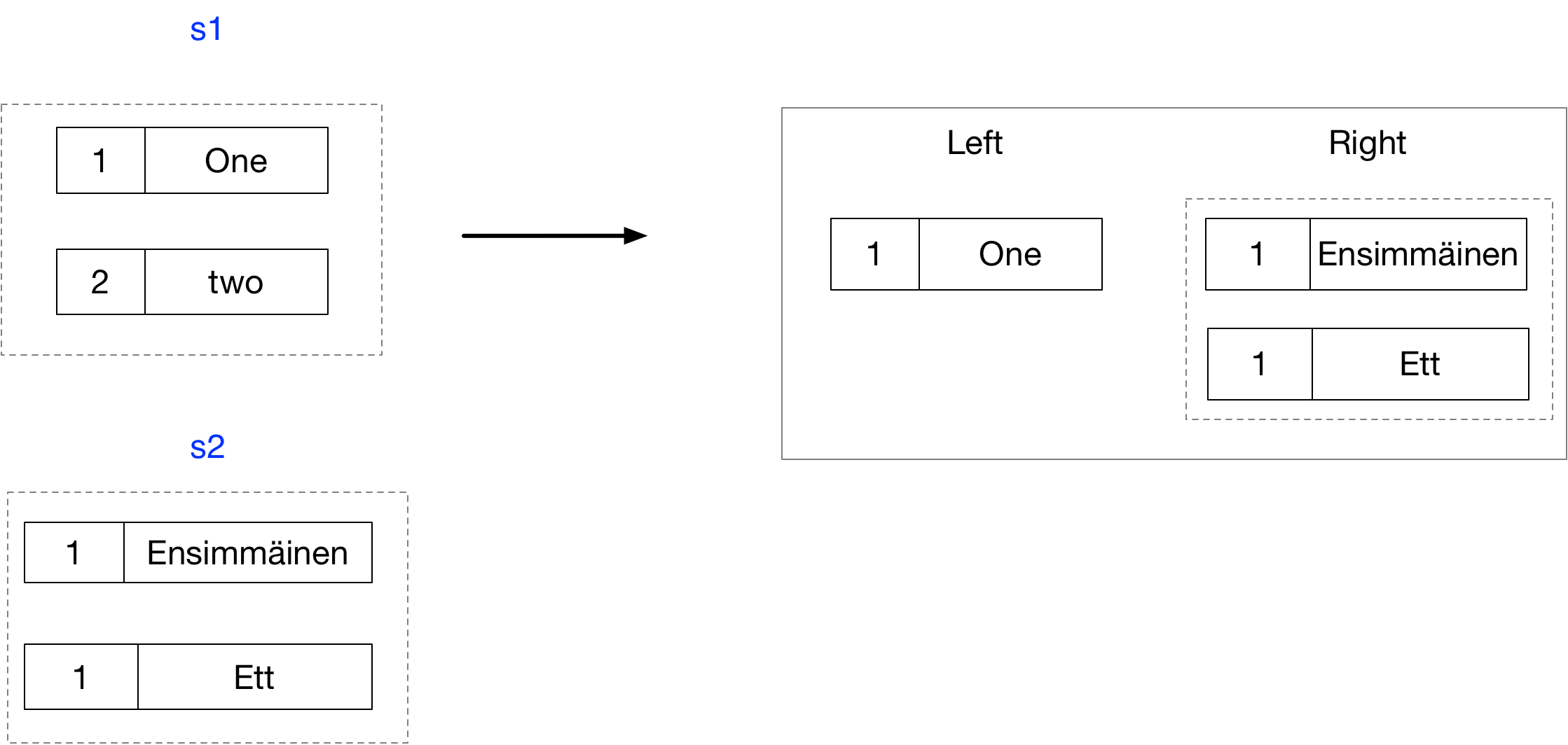
};

var f1 = s1.Select(e1 => new { Left = e1, Right = s1.Where(e2 => e1 != e2) });

Inner Join – Hierarchical

Write query and fluent syntax to perform a hierarchical inner join as follows. Write both fluent and query queries.

1. Inefficient select based queries
2. Efficient GroupJoin queries
3. Efficient select+lookup queries?



// Inefficient

var f1 = s1

.Select(e1 => new {Left = e1, Right=s2.Where(e2=>e1.Item1 == e2.Item1)})

.Where(s => s.Right.Any());

// Inefficient

var q1 = from e1 in s1

select new {Left = e1, Right=s2.Where(e2=>e1.Item1 == e2.Item1)}

into r

where r.Right.Any()

select r;

// GroupJoin

var f2 =s1

.GroupJoin(s2, e1=>e1.Item1, e2=>e2.Item1,(e,si)=>new {Left = e, Right=si})

.Where(s => s.Right.Any());

// Group Join

var q2 = from e1 in s1

join e2 in s2 on e1.Item1 equals e2.Item1

into c

select new {Left = e1, Right=c}

into d

where d.Right.Any()

select d;

// Efficient using select and lookup

var lookup = s2.ToLookup(s => s.Item1);

var f3 = s1

.Select(e1 => new { Left = e1, Right = lookup[e1.Item1] })

.Where(s => s.Right.Any());

f3.Dump();

// Efficient using select and lookup

var q3 = from e1 in s1

select new { Left = e1, Right = lookup[e1.Item1] }

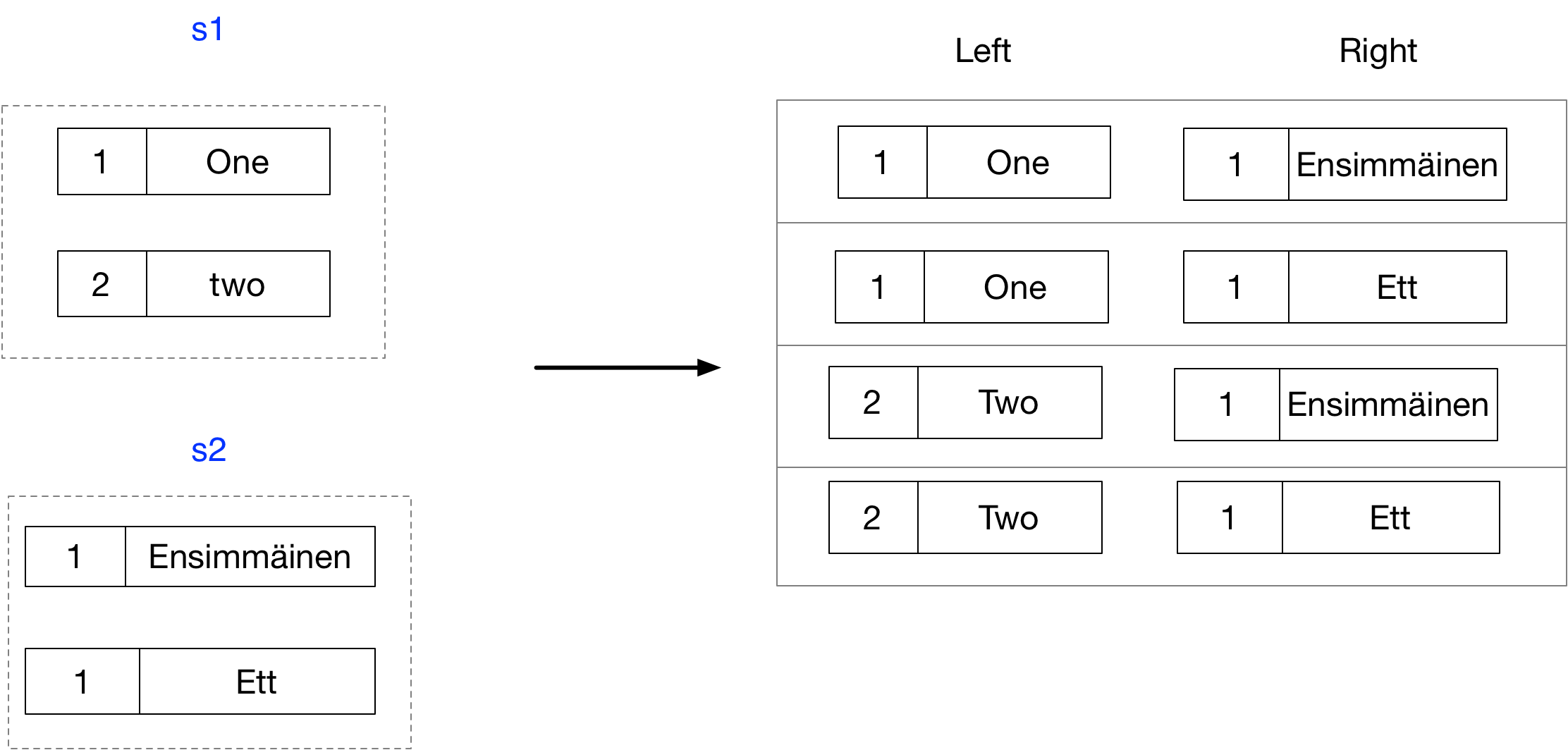
into r

where r.Right.Any()

select r;

Cross Product - Flat

Write query and fluent syntax to perform the following?



var s1 = new[] { (1, "one"), (2, "two"), };

var s2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };

var f = s1.Select(e1 => s2.Select(e2 => new { Left = e1, Right = e2 }));

var q = from e1 in s1

select (from e2 in s2 select new { Left = e1, Right = e2 });

Implementing SelectMany

Implement SelectMany

public static IEnumerable<TResult> SelectMany<TSource,TResult>(

this IEnumerable<TSource> input,

Func<TSource, IEnumerable<TResult>> f)

{

foreach (var element in input)

{

foreach(var el2 in f(element))

yield return el2;

}

}

Implementing Join

Implement Join

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(

this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter,

TInner, TResult> resultSelector)

{

var lookup = inner.ToLookup(innerKeySelector);

foreach (var outerEl in outer)

foreach (var innerEl in lookup[outerKeySelector(outerEl)])

{

var res = resultSelector(outerEl, innerEl);

yield return res;

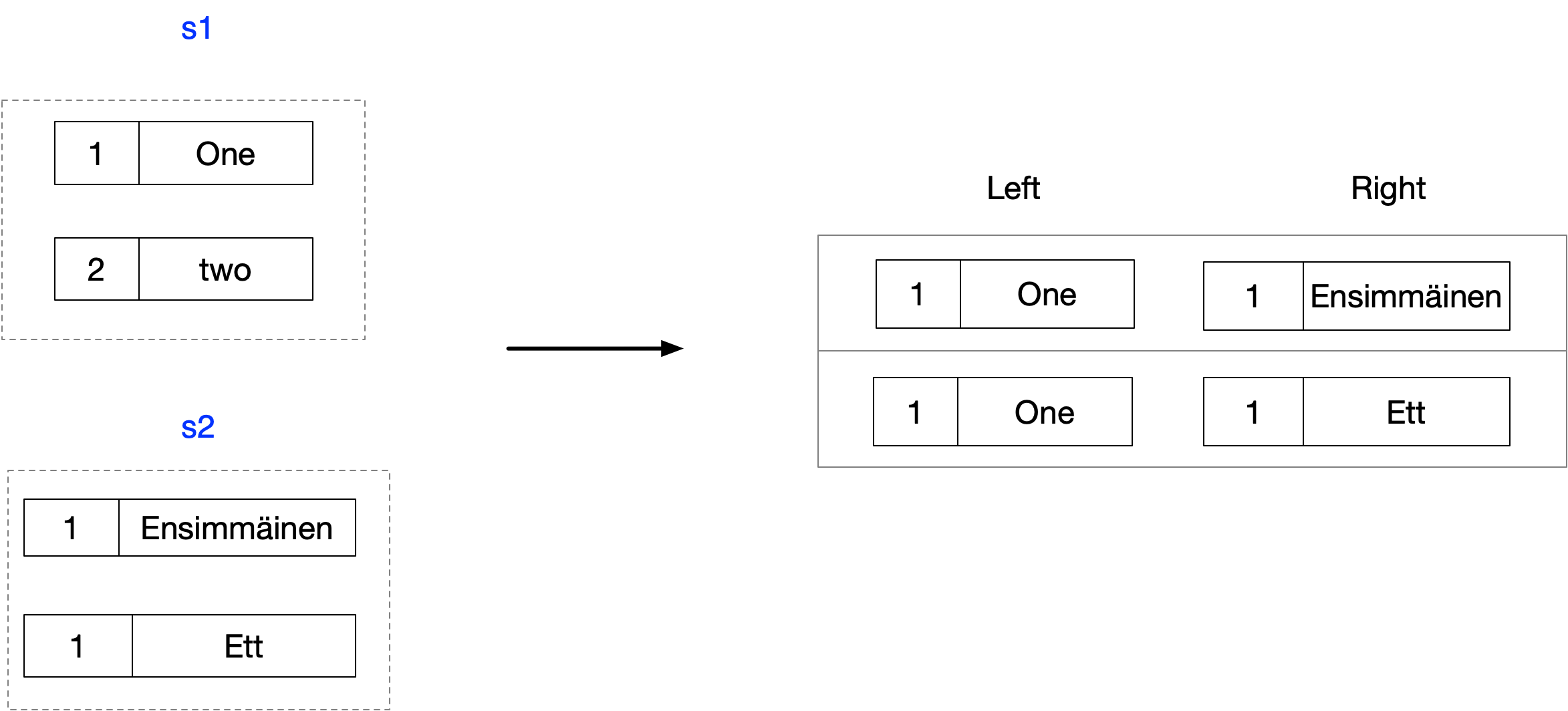
}

}

Inner Join - Flat

Write query and fluent syntax to perform the following. Foreach of the following create a fluent and query syntax query?

1. Inefficient select based queries
2. Efficient Join queries
3. Efficient select+lookup queries?



// Fluent Syntax - Inefficent Group Join

var f1 = s1

.SelectMany(e1 => s2, (e1, e2) => new { e1,e2})

.Where(r => r.e1.Item1 == r.e2.Item1)

.Select(r => new {Left=r.e1, Right=r.e2});

// Query Syntax - Inefficent Group Join

var q1 = from e1 in s1

from e2 in s2

where e1.Item1 == e2.Item1

select new {Left=e1, Right=e2};

//q1.Dump();

// Fluent - Efficient, using Join which internally uses a lookup

var f2 = s1.Join(s2, e1=>e1.Item1, e2=>e2.Item1, (e1,e2)=>new {Left=e1, Right=e2});

// Query - Efficient, using Join which internally uses a lookup

var q2 = from e1 in s1

join e2 in s2 on e1.Item1 equals e2.Item1

select new {Left=e1, Right=e2};

// Fluent - Efficient using SelectMany and Lookup

var lookup = s2.ToLookup(e1 => e1.Item1);

var f3 = s1

.SelectMany(e1 => lookup[e1.Item1], (e1, e2) => new { e1, e2 })

.Select(r => new { Left = r.e1, Right = r.e2 });

// Query Syntax - Efficient using SelectMany and lookup

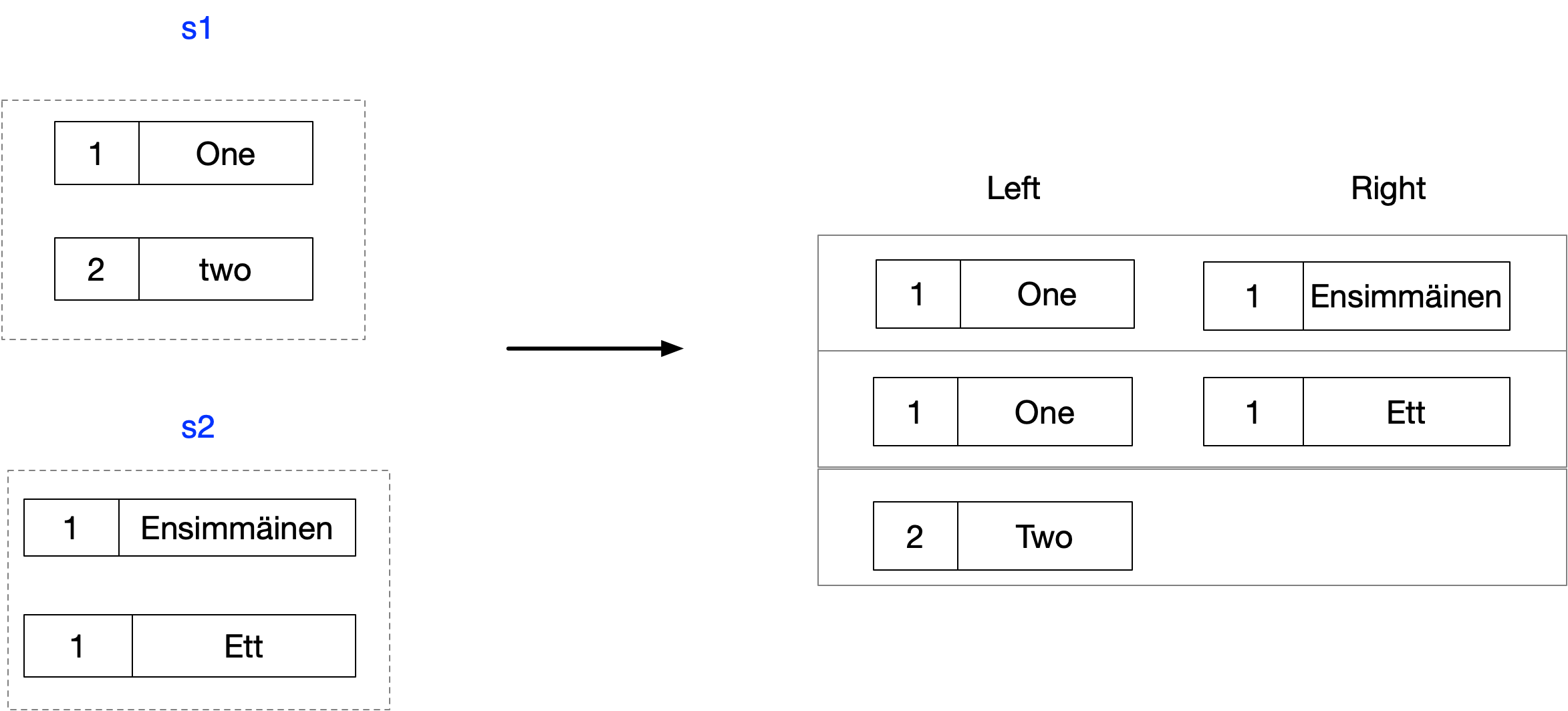
var q3 = from e1 in s1

from e2 in lookup[e1.Item1]

select new { Left = e1, Right = e2 };

Left Outer Join - Flat

Write query and fluent syntax to perform the following.?



/\* First we show how the mechanics work of first doing a group join

\* and then a SelectMany \*/

IEnumerable<((int, string) leftEl, IEnumerable<(int, string)> rightMatches)> joinResults

= Enumerable.GroupJoin(

/\* Left Sequence \*/ s1,

/\* Right Sequence \*/ s2,

/\* Left Key Selector \*/ leftEl => leftEl.Item1,

/\* Right Key Selector \*/ rightEl => rightEl.Item1,

/\* Result Selector \*/ (leftEl, rightMatches) => (leftEl, rightMatches));

IEnumerable<(string Left, string Right)> flattenedResults =

Enumerable.SelectMany(

/\* Source Sequence \*/ joinResults,

/\* Collection Selector \*/ jointElement => jointElement.rightMatches.DefaultIfEmpty(),

/\* Result Selector \*/

(tuple, valueTuple) =>

(Left: tuple.leftEl.Item2, Right: valueTuple.Item2));flattenedResults.Dump();

// Concise Fluent Syntax

IEnumerable<(string Left, string Right)> f1 = s1

.GroupJoin(s2, e1 => e1.Item1, e2 => e2.Item1, (leftEl, rightMatches)

=> (leftEl, rightMatches))

.SelectMany(jointElement

=> jointElement.rightMatches.DefaultIfEmpty(),

(tuple, valueTuple) => (Left: tuple.leftEl.Item2, Right: valueTuple.Item2));

// Concise Query Syntax

IEnumerable<(string Left, string Right)> q2 =

from leftEl in s1

join rightEl in s2 on leftEl.Item1 equals rightEl.Item1 into matches

from match in matches.DefaultIfEmpty()

select (Left: leftEl.Item2, Right: match.Item2);