Relational Databases

## Introduction

This document covers

Introduction

## Overview

An RDBMS is supposed to implement the relational model and provide the means to store, manage, enforce the integrity of, and query data.

- T-SQL Fundamentals

The relational model is based on set theory and predicate logic. SQL is a declarative language in that we describe what we want done and leave the details to the RDBMS. A relation in SQL consists of a heading that specifies the set of attributes (columns) and a body that consists of a set of tuples (rows).

### Constraints

RDBMS allows one to model data integrity by specifying constraints. A candidate key is specified on an attribute (column) to enforce uniqueness of tuples (rows). One of the candidate keys is chosen as the primary key and forms the preferred way of uniquely identifying rows. Foreign key is specified on a referencing relation. A foreign key constraint can be used to enforce referential integrity by ensuring only values that exist in the referenced relations are allowed in the referencing relations foreign key.

If we apply key constraints to a table, each element is unique, and it can be considered a set (Otherwise it is a bag or a multiset). Order is unimportant in a set. For this reason, the result of a query has no order unless we explicitly give some sorting criteria.

### Predicates

A predicate is an expression that is either true or false. Predicates can be used to

1. Enforce data integrity
2. Filter data into subsets.
3. Specify sets by their properties rather than explicit enumeration of elements

All SQL commands can be split into three buckets.

|  |  |
| --- | --- |
|  | Column Header |
| Data Definition Language | CREATE , ALTER, DROP |
| Data Manipulation Language | SELECT, INSERT, UPDATE, DELETE, TRUNCATE, MERGE |
| Data Control Language | GRANT and REVOKE |

## Select

### Cheat Sheet

#### Basic Select

The following queries work on the below Person table.

|  |  |  |
| --- | --- | --- |
| **id** | **firstName** | **secondName** |
| 1 | Kenneth | Wilson |
| 2 | John | Smith |
| 3 | Kelly | Clarkson |

|  |  |  |  |
| --- | --- | --- | --- |
| Name |  | Code |  |
| Select |  | SELECT \*  FROM Person | |  |  |  |  | | --- | --- | --- | --- | | **id** |  | **firstName** | **secondName** | | 1 |  | Kenneth | Wilson | | 2 |  | John | Smith | | 3 |  | Kelly | Clarkson | |
| Select Named Columns |  | SELECT P.firstName  FROM Person P | |  | | --- | | **firstName** | | Kenneth | | John | | Kelly | |
| Select AS |  | SELECT P.firstName AS "First"  FROM Person P | |  | | --- | | **First** | | Kenneth | | John | | Kelly | |

#### Where (Filtration)

The following queries work on the below Person table.

|  |  |  |
| --- | --- | --- |
| **id** | **firstName** | **secondName** |
| 1 | Kenneth | Wilson |
| 2 | John | Smith |
| 3 | Kelly | Clarkson |
| 3 | John | Wilson |

|  |  |  |  |
| --- | --- | --- | --- |
| Name |  | Code |  |
| Equality |  | SELECT firstName  FROM Person P  WHERE firstName = 'John' | |  | | --- | | **firstName** | | John | | John | |
| Inequality |  | SELECT \*  FROM Person  WHERE firstName <> 'John' | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 1 | Kenneth | Wilson | | 3 | Kelly | Clarkson | |
| Like |  | SELECT \*  FROM Person  WHERE firstName LIKE '%enn%' | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 1 | Kenneth | Wilson | |
| IN |  | SELECT \*  FROM Person  WHERE firstName  IN ('Kenneth', 'John') | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 1 | Kenneth | Wilson | | 2 | John | Smith | | 3 | John | Wilson | |
| NOT |  | SELECT \*  FROM Person  WHERE firstName  NOT IN ('Kenneth', 'John') | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 3 | Kelly | Clarkson | |
| OR |  | SELECT \*  FROM Person  WHERE firstName = 'John'  OR firstName = 'Kelly' | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 2 | John | Smith | | 3 | Kelly | Clarkson | | 3 | John | Wilson | |
| AND |  | SELECT \*  FROM Person  WHERE firstName = 'John'  AND secondName = 'Smith' | |  |  |  | | --- | --- | --- | | **id** | **firstName** | **secondName** | | 2 | John | Smith | |
|  |  |  |  |

The following queries work on the below Products table.

|  |  |
| --- | --- |
| **productId** | **description** |
| 1 | European Call |
| 2 | Variance Swap |

|  |  |  |  |
| --- | --- | --- | --- |
| Name |  | Code |  |
| Less than |  | SELECT \*  FROM Product  WHERE productId < 2 | |  |  | | --- | --- | | **productId** | **description** | | 1 | European Call | |
| Less than or equal to |  | SELECT \*  FROM Product  WHERE productId <= 2 | |  |  | | --- | --- | | **productId** | **description** | | 1 | European Call | | 2 | Variance Swap | |
| Greater than |  | SELECT \*  FROM Product  WHERE productId > 1 | |  |  | | --- | --- | | **productId** | **description** | | 2 | Variance Swap | |
|  |  |  |  |

### Phases

➎ SELECT empid, YEAR(orderdate) AS orderyear, COUNT(\*) AS numorders

➊ FROM Sales.Orders

➋ WHERE custid = 71

➌ GROUP BY empid, YEAR(orderdate)

➍ HAVING COUNT(\*) > 1

➏ ORDER BY empid, orderyear;

|  |  |
| --- | --- |
| ➊ FROM | Specify the table we want to query |
| ➋ WHERE | Uses a predicate to filter the rows returned. Where clauses enable the use of indices to improve performance and reduce the network traffic that would occur if we performed a table scan and filtered on the client. |
| ➌ GROUPBY | Produce a group for each unique combination of values specified in this clause. |
| ➍ HAVING | Uses a predicate to filter the groups returned. Can utilise aggregate functions in the predicate. |
| ➎ SELECT | Specify the columns we want to see in the result |
| ➏ ORDER BY | Sort the rows for presentation purposes |
|  |  |

#### Group By

If a query contains a group by phase any subsequent HAVING, SELECT, and ORDERBY clauses work on groups. As such they can only operate on expressions that return a single scalar value per group. Any fields specified in the GROUP BY phase implicitly have this process.

Any elements that do not meet this restriction can only be used as inputs to aggregation functions such as COUNT, SUM, AVG, MIN, MAX.

#### ORDER BY

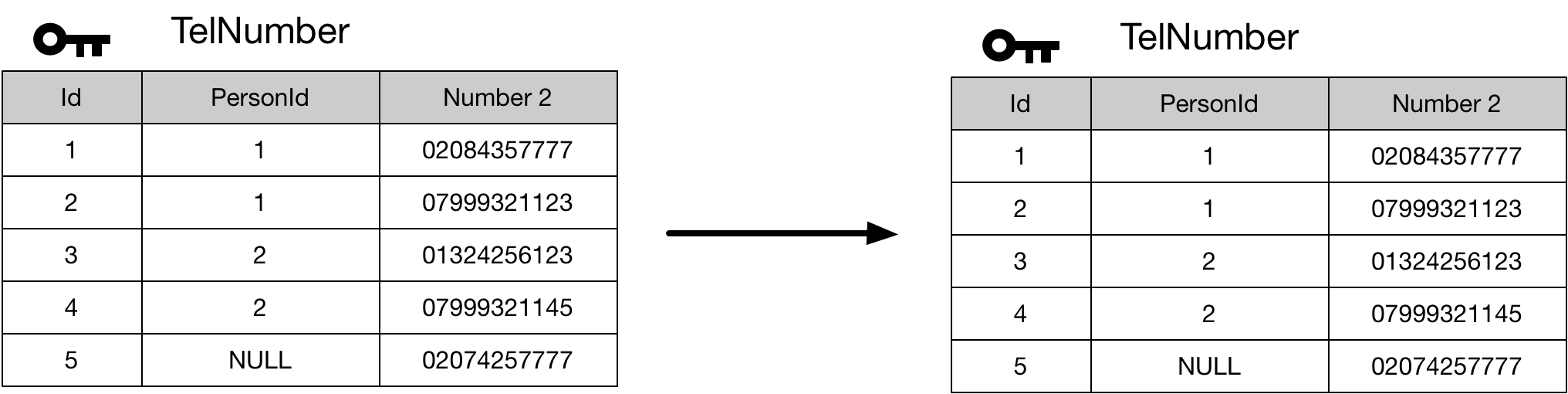
In SQL, a table has no order. By using order by the result is ordered and hence cannot be considered a table. In SQL the ordered rows are referred to as a cursor. Unlike all other phases the order by phase can utilise column alisases defined in the select phase as the order by is the only phase that follows the select phase.

### Examples

#### Simple Query

➋ SELECT \*

➊ FROM TelNumber



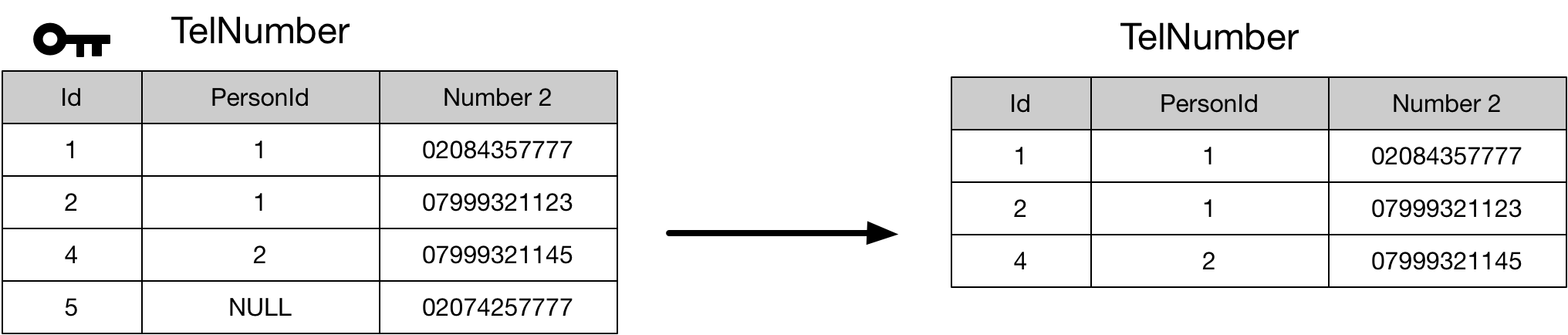
#### Where

Where add a predicate to filter the results.

➌ Select \*

➊ FROM TelNumber

➋ WHERE personId IN (1,2)



#### GROUP BY

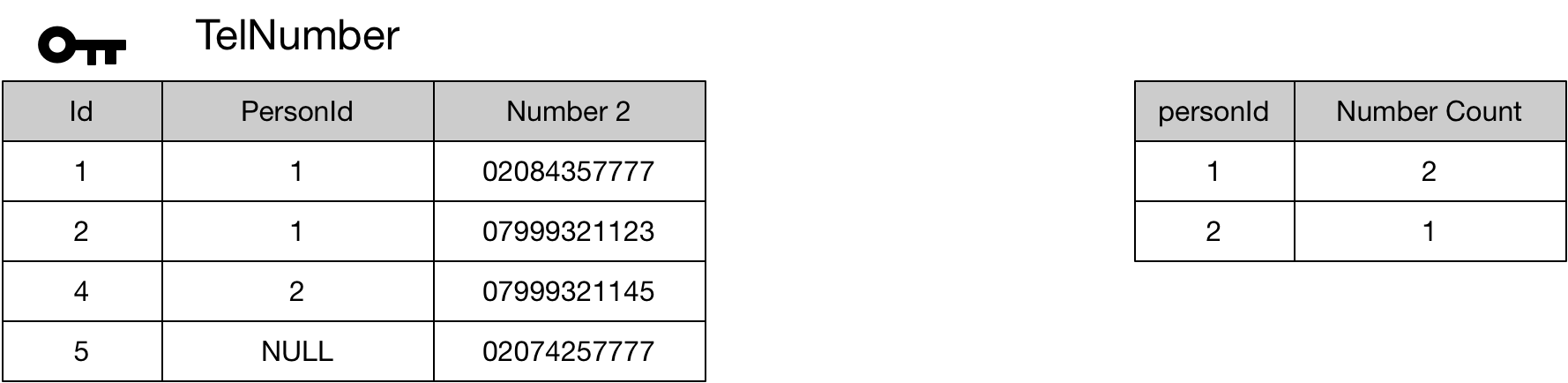
Allows grouping. The select can only work on columns appropriate to the grouping.

➍ Select personId, COUNT(telNumber) AS 'Number Count'

➊ FROM TelNumber

➋ WHERE personId IN (1,2)

➌ GROUP BY personId



#### Having

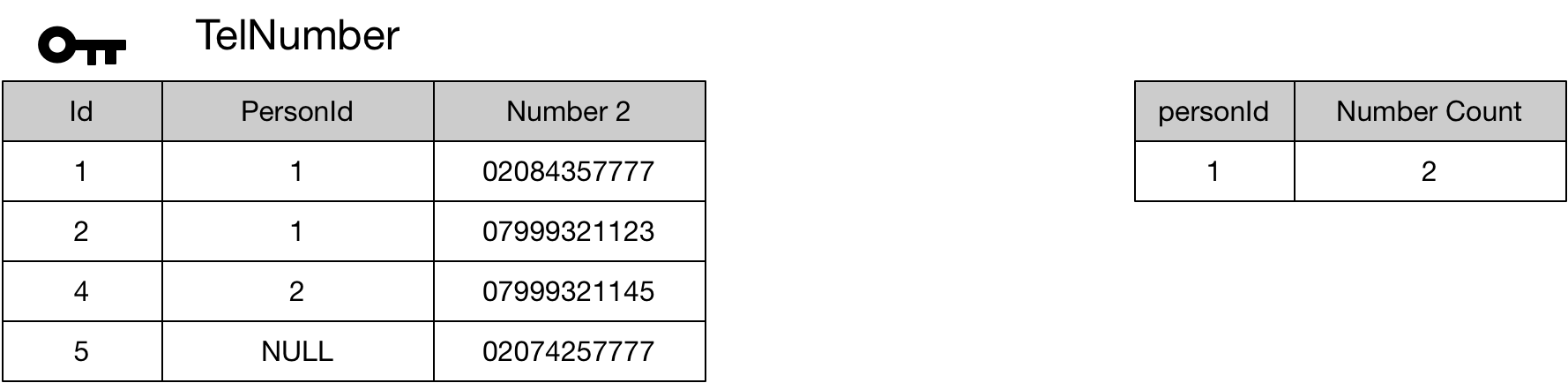
➎ Select personId, COUNT(telNumber) AS 'Number Count'

➊ FROM TelNumber

➋ WHERE personId IN (1,2)

➌ GROUP BY personId

➍ HAVING count(telNumber) > 1



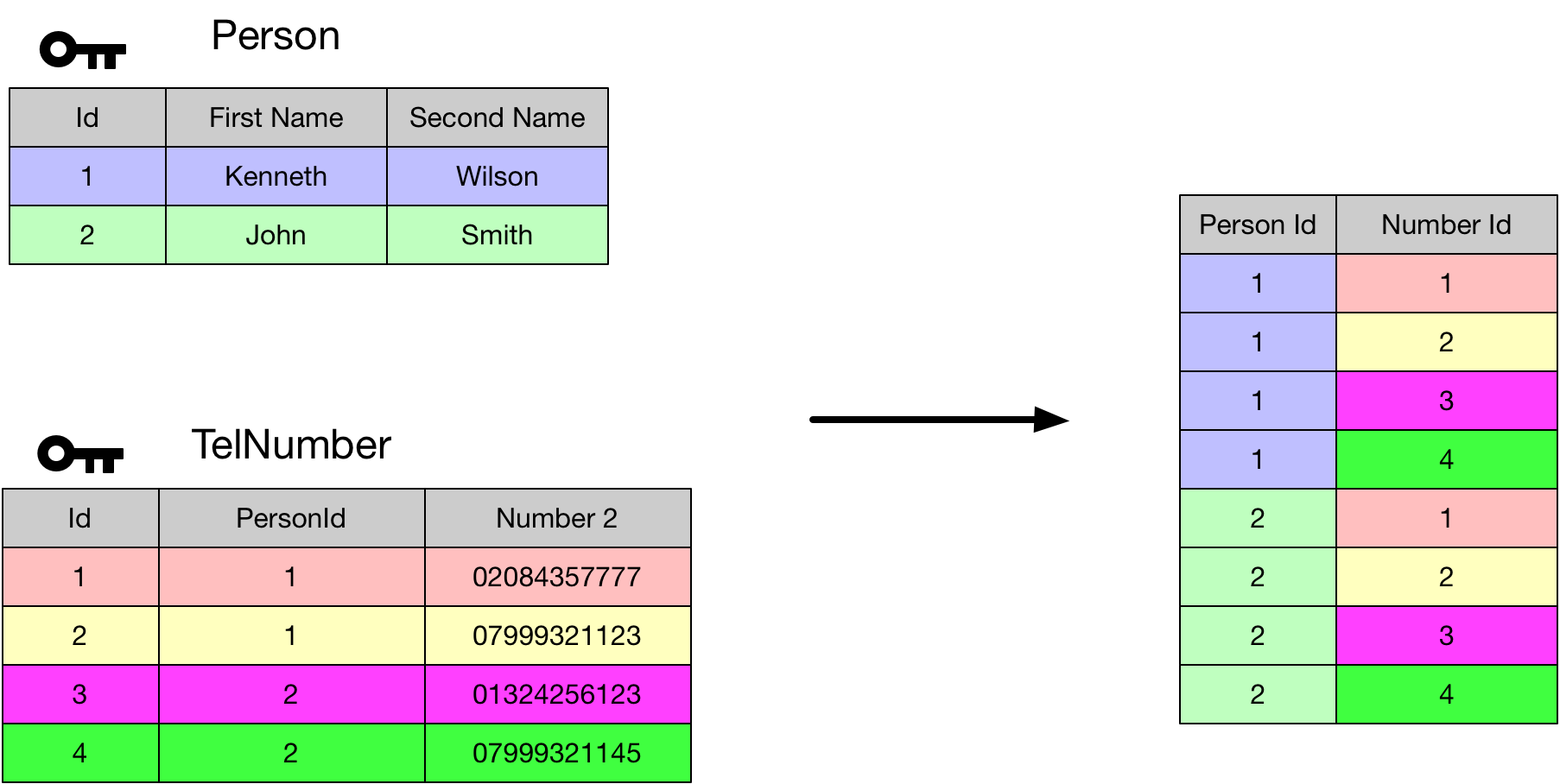
### Joins

#### Cross Join

SELECT p.id AS 'Person Id', T.id AS 'Number Id'

FROM Person AS P

CROSS JOIN TelNumber as T

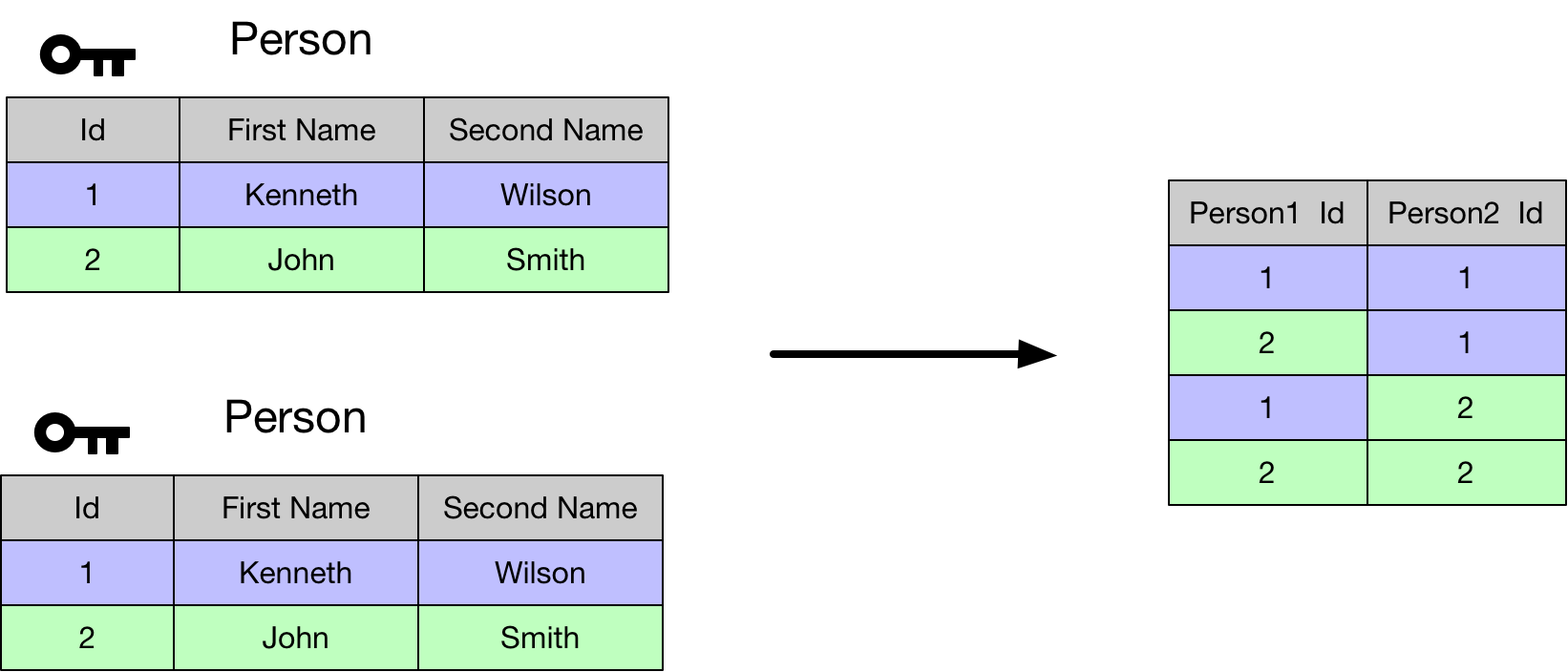


#### Self-Cross Join

SELECT P1.id AS 'Person1 Id', P2.id AS 'Person2 Id'

FROM Person AS P1

CROSS JOIN Person AS P2



#### Inner Join

SELECT

P.firstName AS 'First Name',

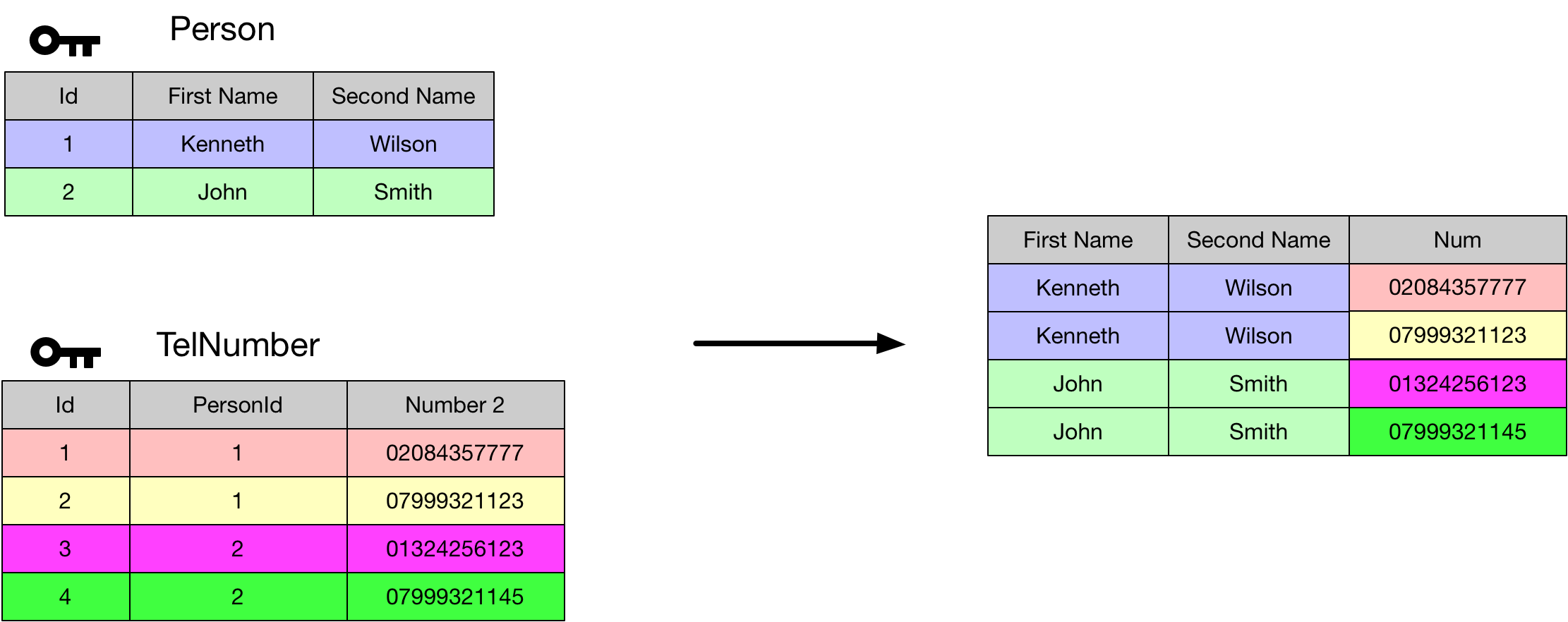
P.secondName AS 'Second Name',

T.telNumber AS 'Num'

FROM Person AS P

INNER JOIN TelNumber AS T

ON P.id = T.personId



#### Non Equi-Join

SELECT

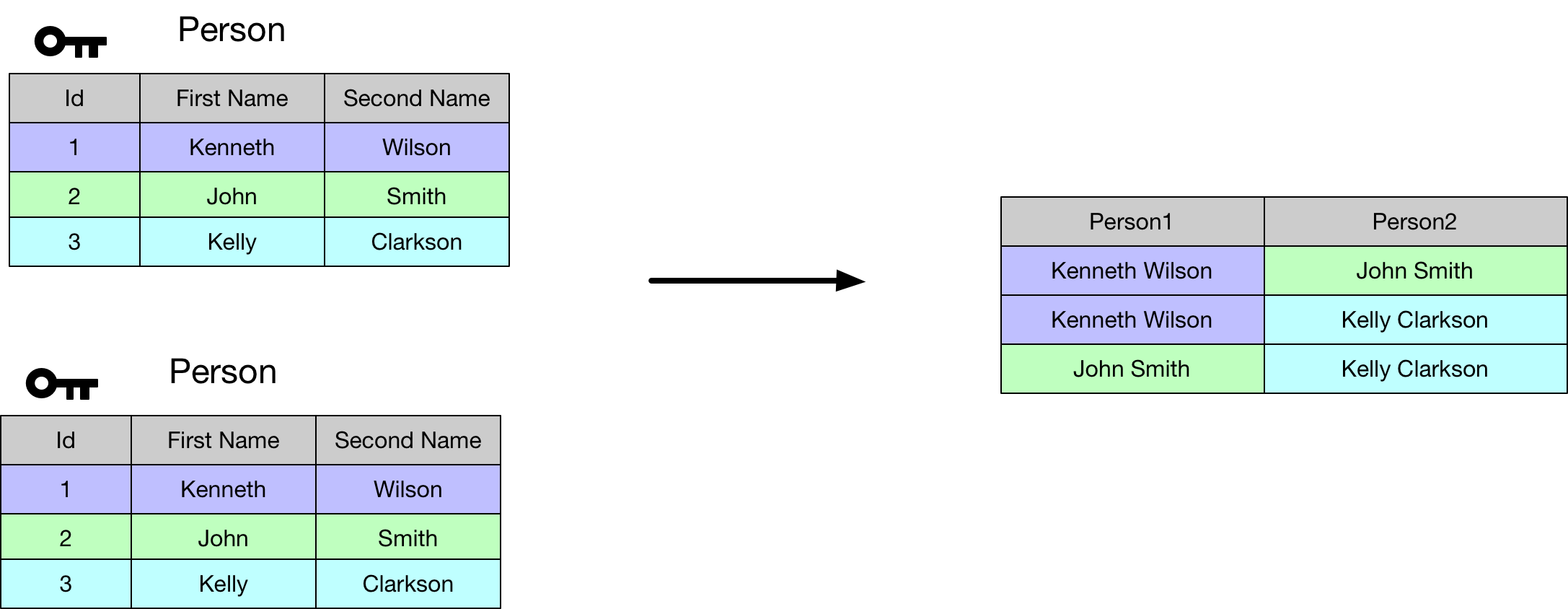
P1.firstName + ' ' + P1.secondName AS 'Person1',

P2.firstName + ' ' + P2.secondName AS 'Person2'

FROM Person AS P1

INNER JOIN Person AS P2

ON P1.id < P2.id



#### Left Outer Join

SELECT

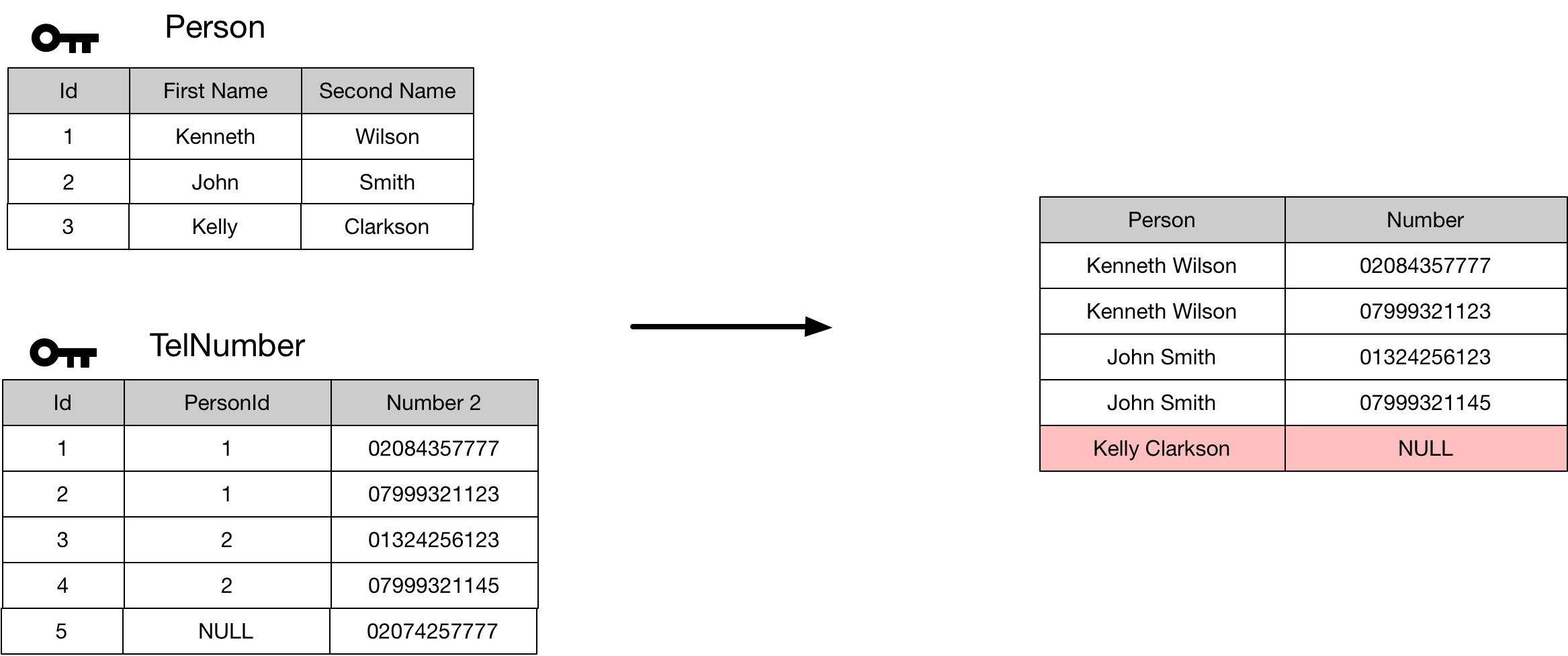
P.firstName + ' ' + P.secondName AS 'Person',

T.telNumber AS 'Number'

FROM Person AS P

LEFT OUTER JOIN TelNumber AS T

On P.id = T.personId



#### Right Outer Join

SELECT

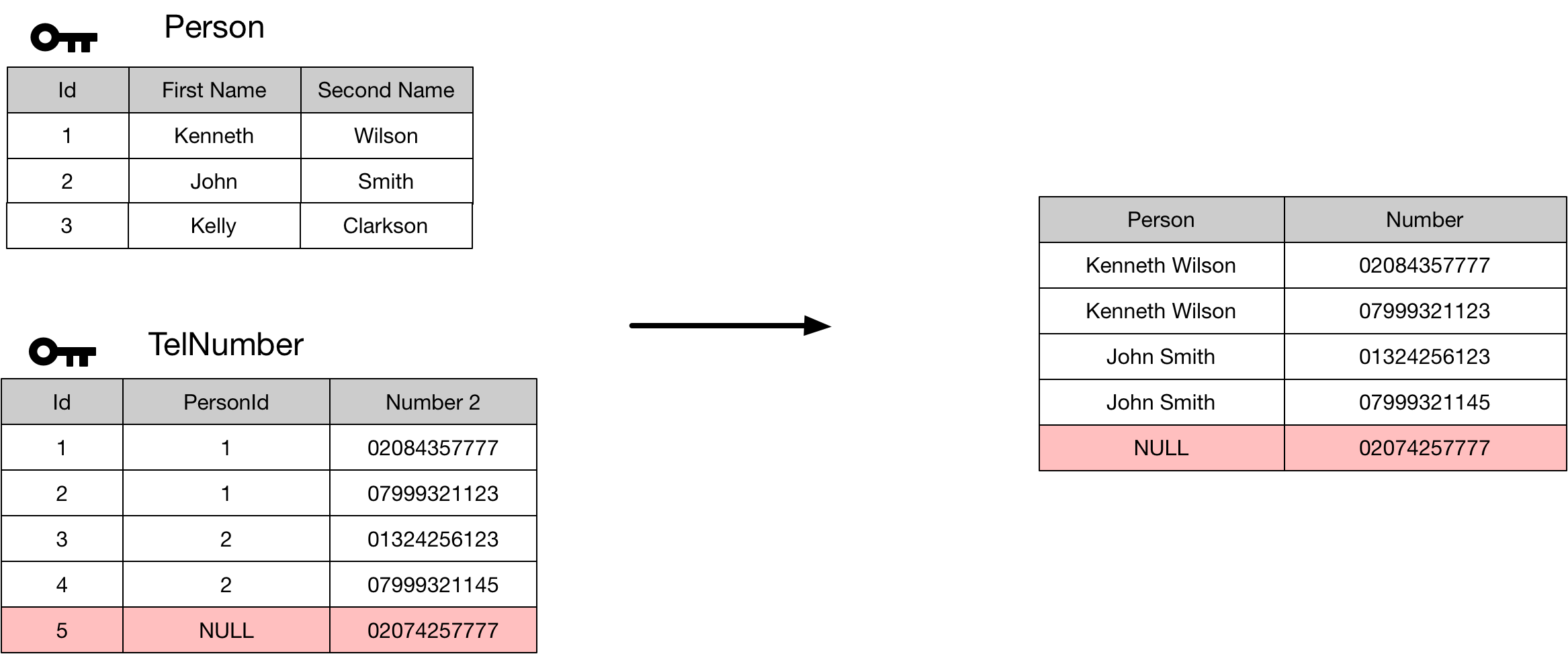
P.firstName + ' ' + P.secondName AS 'Person',

T.telNumber AS 'Number'

FROM Person AS P

RIGHT OUTER JOIN TelNumber AS T

On P.id = T.personId

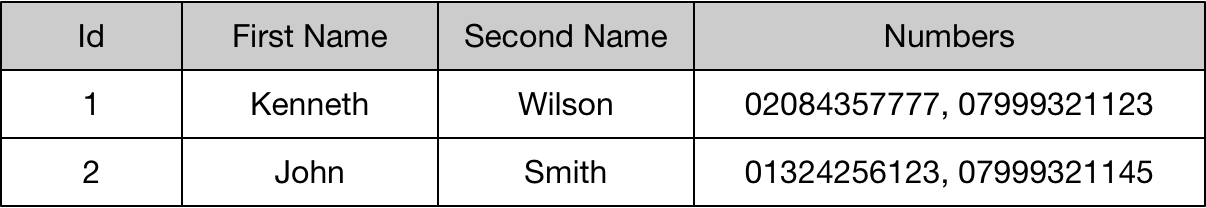


### Subqueries

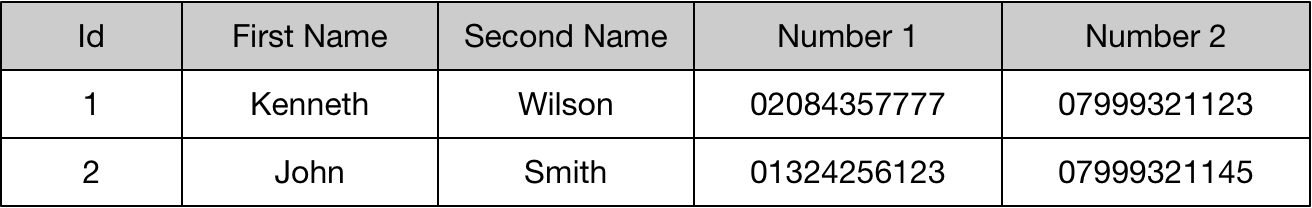
## Normalization

1st Normal Form

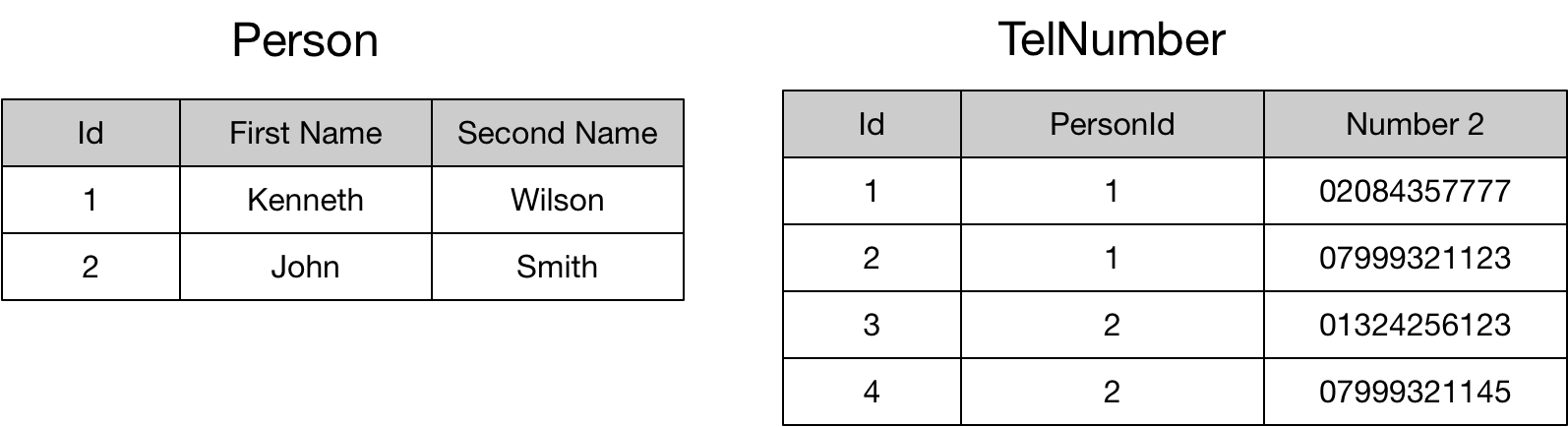
For a table to be in first normal form each cell must contain only one value from the domain. The following table violates this restriction



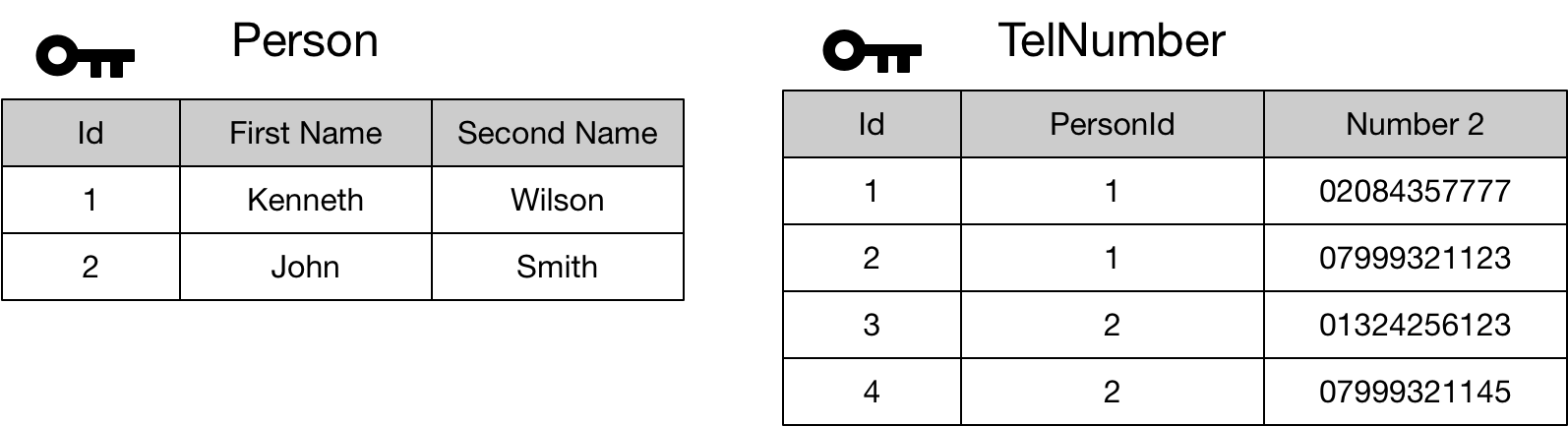
We should note the following table is still in violation of 1st normal form as repeating columns groups are also disallowed.



We can fix this using a schema something like the following. The PersonId field acts as the foreign key that indexes into the person table

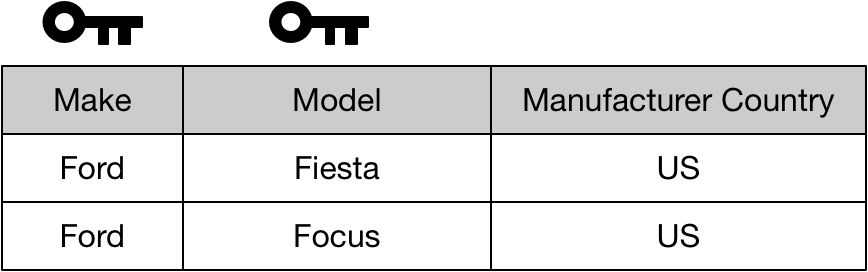


The second restriction on 1st normal form is that every row is unique. We can ensure uniqueness of rows by applying a candidate key to the row.

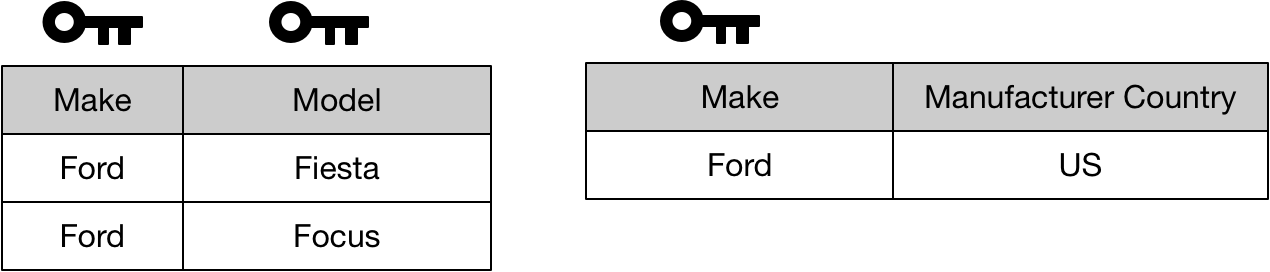


2nd Normal Form

Second normal for applies to relations with composite keys. Where there is a composite key, we should not be able to locate the value of any non-key attribute using only part of the composite key. The following table is in violation of 2nd normal form.



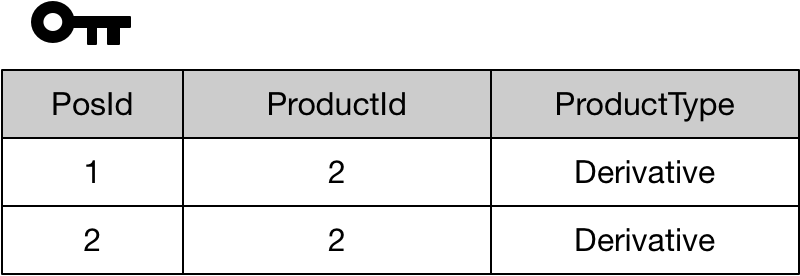
We can fix this as follows



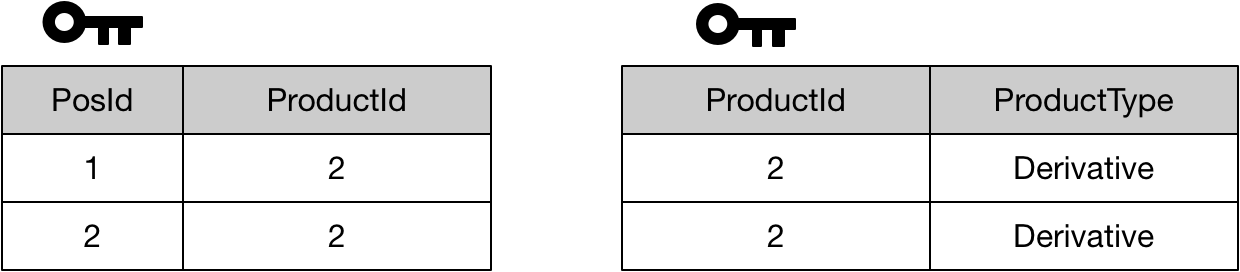
A relation with a single attribute primary key in 1st normal form is automatically in 2nd normal form.

3rd Normal Form

To be in third normal form the relation must first be in second normal form. The second rule is that no non-key attribute can be identified by another non-key attribute. This table is in violation of 3rd normal form



We can fix this as follows.



We can summarise the second and third forms as meaning than in order to identify the value of any non-key attribute we need to use the full primary key. Furthermore we cannot identify the value of any non-key field using another non-key field.

## Query Language

### Predicates

#### IN

Returns true if a value or scalar expression is contained in a specified set of values.

SELECT \* from Products

WHERE prodId IN (1, 2, 3)

#### BETWEEN

SELECT \* from Products

WHERE prodId BETWEEN 1 AND 3

#### LIKE

SELECT \* from Products

WHERE description LIKE 'E%'

### Operators

#### Equality and Ordinality

|  |  |
| --- | --- |
| Operator | Description |
| = | Equality |
| > |  |
| < |  |
| >= |  |
| <= |  |
| <> |  |

#### Logical

|  |  |
| --- | --- |
| Operator | Description |
| AND | and |
| OR |  |
| NOT |  |

#### Arithmetic

|  |  |
| --- | --- |
| Operator | Description |
| + |  |
| - |  |
| \* |  |
| / |  |
| % | Modulo |

If two arguments are of the same type the result is of the same type. So, 7 / 2 = 3. We might want to perform a cast in this instance as follows.

SELECT CAST(7 as Numeric(12,2))/ CAST(3 as Numeric(12,2)) as num

### Three Valued Logic

In SQL predicates can evaluate to TRUE, FALSE or UNKNOWN. If one of the arguments in a logical expression is NULL, then the result is UNKNOWN. If a logical expression is used in a query filter, then any result of UNKNOWN leads to a rejection (accept true). If a logical expression used in a check constraint returns UNKNOWN, the value is accepted (reject false)

## Type

### Characters

Characters can be either Unicode or regular

|  |  |
| --- | --- |
| Type | Description |
| CHAR | Fixed length, 1 byte per character |
| VARCHAR |  |
| NCHAR | Fixed length Unicode, 2 bytes per character |
| NVARCHAR |  |

### Date and Time

|  |  |
| --- | --- |
| Type | Description |
| DATETIME | Legacy |
| SMALLDATETIME | Legacy |
| DATE | 3 bytes representing range from 00010101 to 99991231 |
| TIME | 3 to 5 bytes giving accuracy to 100 nanoseconds |
| DATETIME2 | 6 to 8 bytes giving date and time accuracy to same level as DATE and TIME combined |
| DATETIMEOFFSET | 8 to 10 bytes. Similar to DATETIE2 but includes offset from UTC |

To create date and time values when use string literals which are then implicitly converted to the relevant type. When using literals the best practice is to use the language neutral format ‘YYYYMMDD’. (Other formats depends on the language of the session)

insert Trades values(2,'20200103')

select tradeId AS 'Trade Id', tradeDate AS 'Trade Date' from Trades

|  |  |
| --- | --- |
| **Trade Id** | **Trade Date** |
| 2 | 03/01/2020 00:00:00 |

Legacy Datetime type

When dealing with the legacy type DATETIME the convention is that we use a time of midnight if we want to only use the date part and a date of January 1st 1900 if we are only interested in the time part.

#### Inefficient of manipulation columns in filters

We need to careful when applying filters such as WHERE phases. If we manipulate the filtered column this can prevent the database server using the index in an efficient manner. The following is an inefficient query

SELECT tradeId AS 'Trade Id', tradeDate AS 'Trade Date'

FROM Trades

WHERE YEAR(tradeDate) = 2020

We can improve our query using a range filter as follows which enables the efficient use on indices.

SELECT tradeId AS 'Trade Id', tradeDate AS 'Trade Date'

FROM Trades

WHERE tradeDate >= '20200101' AND tradeDate < '20210101'

#### **Functions**

Functions

|  |  |  |
| --- | --- | --- |
| Function | Returns | Description |
| GETDATE | DATETIME | Gets current date and time |
| GETUTCDATE | DATETIME | Get current date and time in UTC |
| SYSDATETIMEOFFSET | DATETIMEOFFSET | Get current date and time with UTC offset |
| SWITCHOFFSET |  | Switches to a different UTC Offset (Timezone) |
| DATEADD |  | Add years, months or days to a |
| DATEDIFF |  | Give different between dates in some date part (year, month, day etc) |
| DATEPART |  | Get year, month, day etc |
| YEAR,MONTH,DAY |  | Abbreviations for DATEPART |

As none of the above functions return only the date or only the time we need to do a little extra to get these

SELECT

CAST(SYSDATETIME() AS DATE) AS [Date],

CAST(SYSDATETIME() AS TIME) AS [Time]

|  |  |
| --- | --- |
| **Date** | **Time** |
| 27/04/2020 00:00:00 | 20:29:54.8753964 |

## MetaData

## Architecture

A single SQL server can hold multiple user databases in addition to a set of system databases (tempdn, model etc)

## Data Definition

The following creates a table.

DROP TABLE IF EXISTS dbo.Products;

CREATE TABLE dbo.Products

(

prodId INT NOT NULL,

description VARCHAR(30) NOT NULL

);

We can setup declarative data integrity. First, we show how to add a primary key constraint.

#### Primary Key Constaint

A primary key has the following properties

* Each table can have only one primary key
* The fields making up the primary key cannot be null
* The server creates an index to efficiently enforce uniqueness and retrieval

ALTER TABLE dbo.Products

ADD CONSTRAINT PK\_Products

PRIMARY KEY(prodId);

#### Foreign Key Constraint

## SQLServer Install

Use the following link and choose the third one from the top

<https://www.hanselman.com/blog/download-sql-server-express>

Once you have installed it open a command prompt and enter the command

SqlLocalDb info

>> MSSQLLocalDB

>> ProjectsV13

We can check the status of an instance by issuing the command

SqlLocalDb info MSSQLLocalDB

>> Name: MSSQLLocalDB

>> Version: 13.1.4001.0

>> Shared name:

>> Owner: DESKTOP-6KCJGVG\rps

>> Auto-create: Yes

>> State: Running

>> Last start time: 06/01/2021 08:47:32

>> Instance pipe name: np:\\.\pipe\LOCALDB#6222EED4\tsql\query

If the instance is not running, we can start it by issuing the command.

SqlLocalDb start MSSQLLocalDB