Updating your Codebase with FBQL

Cloud xRP Summit

Virtual Developer Conference

Kensium Solutions
Yuriy Zaletskyy

Acumatica
The Cloud ERP

FBQL is awesome, but not backward compatible

If you have hammer, everything else look like a nail.
FBQL is not backward compatible.

Fix applicable to 2019 R1 will not be applicable to previous versions
(not to 2017, 2018)

That’s why DAC Schema browser is not upgraded

Types of BQL

- Traditional BQL

- Fluent BQL
  - Similar to SQL than BQL
  - Namespaces
    - PX.Data.BQL
    - PX.Data.BQL.Fluent

How Acumatica team uses FBQL?

14 out of 2459 *.cs files has FBQL statements, which shows that Acumatica team gradually introducing FBQL (demo with TC)

It’s like you have old bridge, use it until a new bridge built

Before you Proceed

- Make sure that the application database has the database tables
- Add references to PX.Data.dll and PX.Data.BQL.Fluent.dll in the project.

- Add the following using directives to your code.
  - using PX.Data.BQL.Fluent;
  - using PX.Data.BQL;

Simpler Where2

Compare

```
return PXSelect<Vendor,
        Where2<Where<Vendor.type, Equal<BAccountType.vendorType>,
                Or<Vendor.type, Equal<BAccountType.combinedType>>>,
            And<Vendor.bAccountID, Equal<Required<Vendor.bAccountID>>>>>
    .Select(graph, bAccountID);
```

with

```
return SelectFrom<Vendor>.Where<Vendor.type.IsEqual<BAccountType.vendorType>.
        Or<Vendor.type.IsEqual<BAccountType.combinedType>.
        And<Vendor.bAccountID.IsEqual<@P.AsInt>>>.View
    .Select(graph, bAccountID);
```

but one grain of salt
if somebody used for Where2 <--> OR combination,
then he may skip OR condition during debugging

Potential pitfalls

Initially you may be tempted to write code like this:

```
public SelectFrom<SOAdjust, InnerJoin<ARPayment, On<ARPayment.docType, Equal<SOAdjust.adjgDocType>,
    And<ARPayment.refNbr, Equal<SOAdjust.adjgRefNbr>>>>> Adjustments3;
```

While correct way is this:

```
public SelectFrom<SOAdjust>.InnerJoin<ARPayment>.On<SOAdjust.adjgDocType.IsEqual<ARPayment.docType>.
    And<SOAdjust.adjgRefNbr.IsEqual<ARPayment.refNbr>>> Adjustments2;
```

Compiler will not allow you to compile BQL code in FBQL statement, but some initial learning curve may happen. This is the case when junior developers are in better position

AggregateTo<> and OrderBy<> Sections

- Accept non-empty arrays of the specific base type
- The *AggregateTo<>* section can also include an optional *Having<>* subsection
- In this subsection, you can include fields with fields with .Averaged, .Summarized, .Maximized, .Minimized, or .Grouped

.AggregateTo<Sum<field1>, GroupBy<field2>, Max<field3>, Min<field4>, Avg<field5>, Count<field6>>. Having<field5.Averaged.IsGreater<Zero>>

.OrderBy<field1.Asc, field2.Desc, field3.Asc>

Mix of BQL with FBQL classes in filtering

How to have in one BQL query old and new DAC class?

with help of Use

```
var mixSelect = SelectFrom<Contact>.Where<Contact.displayName.
    IsEqual<Use<Student1.firstName>.AsString>>.View.Select(Base);
```

Contact is new FBQL class, and Student1 as you remember is BQL.

What is more important, even after Student1 will be upgraded, usage of Use will not cause any harm!

How Having look like in code?

In Acumatica code base it is not yet presented, so you can try to pioneer in it's usage.

```
var havingTest = SelectFrom<SOOrder>.LeftJoin<SOLine>.On<SOOrder.orderNbr.IsEqual<SOLine.orderNbr>.
        And<SOOrder.orderType.IsEqual<SOLine.orderType>>>.AggregateTo<Sum<SOOrder.curyOrderTotal>>
    .Having<SOLine.baseOpenQty.Averaged.IsGreater<@P.AsDecimal>>.View.Select(Base, 35.6m);
```

Data Views in Fluent BQL

- Use the PXViewOf<> class before the fluent BQL query

```
PXViewOf<Product>.BasedOn< SelectFrom<Product>.
Where<Product.isActive.IsEqual<True>>>.ReadOnly ActiveProducts;
```

- You can omit *.BasedOn<>* if you want to declare a view that selects all records from one table
- You append *.ReadOnly* to the view definition if you need to define a read-only data view.
- Append *.View* to the fluent BQL query, as shown in the following code example

```
SelectFrom<Product>. Where<Product.isActive, Equal<True>>.View.ReadOnly ActiveProducts;
```

Samples:

Which one is more convenient:

way 1:

```
public PXViewOf<SOOrder>.BasedOn<SelectFrom<SOOrder>.
    Where<SOOrder.approved.IsEqual<True>>> ActiveProducts1;
```

way 2:

```
public SelectFrom<SOOrder>.
    Where<SOOrder.approved.IsEqual<True>>.View ActiveProducts2;
```

Try to guess what Acumatica team chosen?

Search Commands

- Use the SearchFor<> class before the fluent BQL query

  SearchFor<*Product.productId*>.*In*< *SelectFrom*<*Product*>.
  Where<*Product.isActive.IsEqual*<*True*>>>

- Append .**SearchFor**<> to the fluent BQL query

  SelectFrom<*Product*>.
  Where<*Product.isActive.IsEqual*<*True*>>.***SearchFor***<*Product.productId*>

Data Access Classes in Fluent BQL

- DACs that are used in fluent BQL differ from the DACs that are used in traditional BQL
  - Not from the IBqlField interface
  - But from the specific fluent BQL classes

```
// The class used in BQL statements to refer to the AvailQty column
public abstract class availQty : PX.Data.BQL.BqlDecimal.Field<availQty> { }
// The property holding the AvailQty value in a record
[PXDBDecimal(2)]
public virtual decimal? AvailQty { get; set; }
```

- The DAC fields declared in fluent BQL style can be used in traditional BQL queries without any modifications.

Constants in Fluent BQL

Constants (such as integer Zero, datetime Now, Today, and MaxDate, string StringEmpty, and the Boolean values True and False) in fluent BQL queries without any changes

```
public class decimal_0 : PX.Data.BQL.BqlDecimal.Constant<decimal_0>
{
public decimal_0()
  : base(0m)
{
}
}
```

These constants can also be used in traditional BQL without any changes

C# Type	Fluent BQL Type
bool	BqlBool.Constant<TSelf>
byte	BqlByte.Constant<TSelf>
short	BqlShort.Constant<TSelf>
int	BqlInt.Constant<TSelf>
long	BqlLong.Constant<TSelf>
float	BqlFloat.Constant<TSelf>
double	BqlDouble.Constant<TSelf>
decimal	BqlDecimal.Constant<TSelf>
Guid	BqlGuid.Constant<TSelf>
DateTime	BqlDateTime.Constant<TSelf>
String	BqlString.Constant<TSelf>

Parameters in Fluent BQL

- Use of the Current Value of the Field from PXCache
  - –append *.FromCurrent* to the field name
  - –append *.FromCurrent.NoDefault* to the field name
- Insertion of a Specific Value into the Query
  - –Use the *@P.As[Type]* classes, where [Type] corresponds to the C# type of the parameter
  - –@P.As[Type] is the equivalent of the Required parameter in traditional BQL
- Insertion of an Optional Value into the Query
  - –append *.AsOptional* to the field name
  - –If not value passed, it takes the value from *.FromCurrent*
  - –append *.AsOptional.NoDefault*
- Insertion of a Value from the UI Control into the Query
  - –use the *Argument.As[Type]* classes, where [Type] corresponds to the C# type

Constructing F-BQL

SQL

```
SELECT Product.CategoryCD, MIN(Product.BookedQty) FROM Product
INNER JOIN SupplierProduct
   ON SupplierProduct.ProductID = Product.ProductID
INNER JOIN Supplier
   ON Supplier.AccountID = SupplierProduct.AccountID
WHERE (Product.BookedQty IS NOT NULL
      AND Product.AvailQty IS NOT NULL
      AND Product.MinAvailQty IS NOT NULL
      AND(Product.Active = 1
         OR Product.Active IS NULL)
      AND(Product.BookedQty > Product.AvailQty
         OR Product.AvailQty < Product.MinAvailQty))
   OR Product.AvailQty IS NOT NULL
GROUP BY Product.CategoryCD
ORDER BY Product.UnitPrice, Product.AvailQty DESC
```

F-BQL

```
SelectFrom<Product>.
  InnerJoin<SupplierProduct>.
    On<SupplierProduct.productID.IsEqual<Product.productID>>.
  InnerJoin<Supplier>.
    On<Supplier.accountID.IsEqual<SupplierProduct.accountID>>.
  Where<
    Brackets<Product.bookedQty.IsNotNull.
             And<Product.availQty.IsNotNull>.
             And<Product.minAvailQty.IsNotNull>.
             And<Product.active.IsEqual<True>.
              Or<Product.active.IsNull>>.
             And<Product.bookedQty.IsGreater<Product.availQty>.
             Or<Product.availQty.IsLess<Product.minAvailQty>>>>.
    Or<Product.availQty.IsNotNull>>.
  AggregateTo<GroupBy<Product.categoryCD>,
             Min<Product.bookedQty>>.
  OrderBy<Product.unitPrice.Asc, Product.availQty.Desc>
```


What is faster: BQL of FBQL?

BQL may be faster

Each FBQL statement is converted into
BQL and then converted into SQL Query

Testing methodology

```
5 references
public class FBQLStudent1 : PXGraph<FBQLStudent1, Student1>
{
    public PXSelect<Student1> Students; // Classical BQL graph
}

5 references
public class FBQLStudent2 : PXGraph<FBQLStudent2, Student2>
{
    public SelectFrom<Student2>.View Students; // FBQL graph
}
```

Classical BQL Declaration

Student1.cs

FBQLPerformanceTest | FBQLPerformanceTest.Student1.studentID

```
6 references
public class Student1 : IBqlTable
{
    0 references
    public abstract class studentID : IBqlField { }

    [PXDBIdentity]
    0 references
    public virtual int? StudentID { get; set; }

    1 reference
    public abstract class studentCD : IBqlField { }

    [PXDBString(50, IsKey = true, IsUnicode = true)]
    [PXSelector(typeof(Search<Student1.studentCD>), typeof(Student1.firstName)
    1 reference
    public virtual string StudentCD { get; set; }

    1 reference
    public abstract class firstName : IBqlField { }

    [PXDBString(50)]
    [PXUIField(DisplayName = "First name")]
    1 reference
    public virtual string FirstName { get; set; }

    1 reference
    public abstract class lastName : IBqlField { }

    [PXDBString(50)]
    [PXUIField(DisplayName = "Last Name")]
    1 reference
    public virtual string LastName { get; set; }
}
}
```

FBQL Declaration

Student2.cs

FBQLPerformanceTest | FBQLPerformanceTest.Student2 | StudentID

```
6 references
public class Student2 : IBqlTable
{
    1 reference
    public abstract class studentID : BqlInt.Field<studentID> { }

    [PXDBIdentity]
    0 references
    public virtual int? StudentID { get; set; }

    2 references
    public abstract class studentCD : BqlString.Field<studentCD> { }

    [PXDBString(IsKey = true)]
    [PXSelector(typeof(Search<Student2.studentCD>), typeof(Student2.firstName
    1 reference
    public virtual string StudentCD { get; set; }

    2 references
    public abstract class firstName : BqlString.Field<firstName> { }

    [PXDBString]
    [PXUIField(DisplayName = "First name")]
    1 reference
    public virtual string FirstName { get; set; }

    2 references
    public abstract class lastName : BqlString.Field<lastName> { }

    [PXDBString]
    [PXUIField(DisplayName = "Last Name")]
    1 reference
    public virtual string LastName { get; set; }
}
}
```

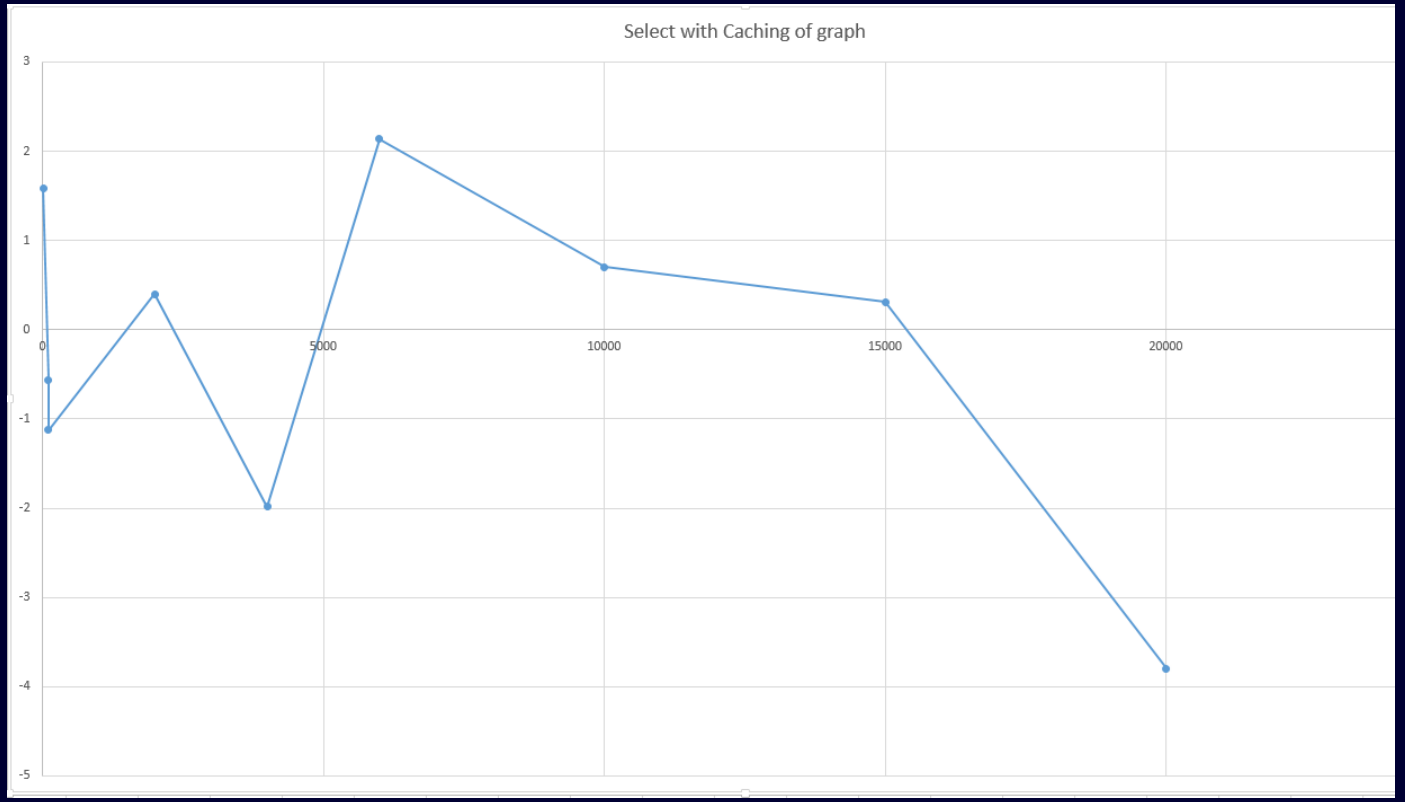
Random BQL Search

```
int numberOfNotNull = 872;

Random rand = new Random();

var sw = new Stopwatch();
sw.Start();
var graph0 = PXGraph.CreateInstance<FBQLStudent1>();
for (int i = 0; i < numberOfIterations; i++)
{
    int startIdx = rand.Next(numberOfNotNull);

    if (cachingOfGraph)
    {
        graph0 = PXGraph.CreateInstance<FBQLStudent1>();
    }

    var contact = PXSelect<Contact, Where<Contact.displayName, IsNotNull, And<Contact.displayName, Contains<Required<Contact.displayName>>>>>.SelectWindowed(graph0, startIdx, 1, ' ').First();

    var firstName = contact.GetItem<Contact>().DisplayName.Split(' ')[0];
    var secondName = contact.GetItem<Contact>().DisplayName.Split(' ')[1];
}
sw.Stop();
sb.Append($"Classical select took {sw.ElapsedMilliseconds} milliseconds on {numberOfIterations} of iterations");
```

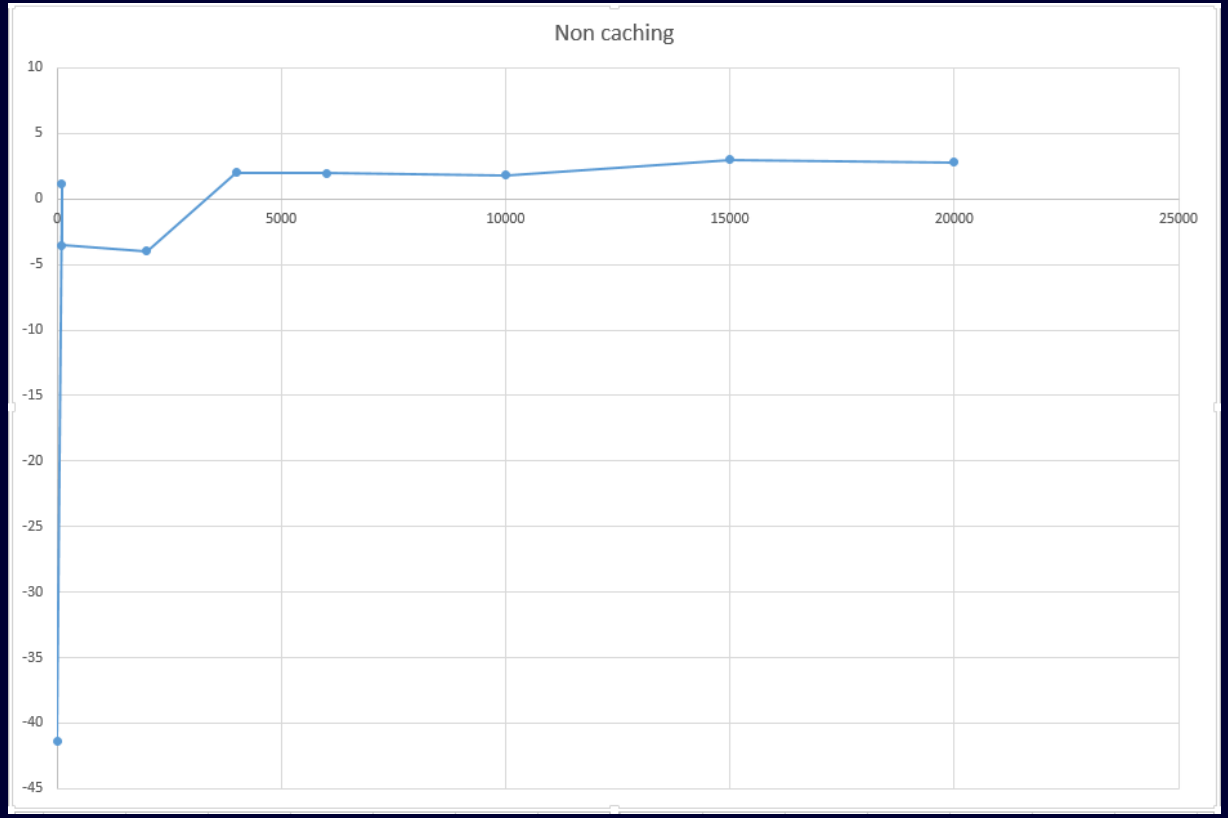
Random FBQL Search

```
var sw1 = new Stopwatch();
sw1.Start();

var graph1 = PXGraph.CreateInstance<FBQLStudent2>();
for (int i = 0; i < numberOfIterations; i++)
{
    int startIdx = rand.Next(numberOfNotNull);

    if (cachingOfGraph)
    {
        graph1 = PXGraph.CreateInstance<FBQLStudent2>();
    }


    //var contact = PXSelect<Contact, Where<Contact.displayName, IsNotNull, And<Contact.displayName, Contains<Required<Contact.displayName>>>>>.SelectWindowed(graph, startIdx, 1, ' ').First();
    var contact = SelectFrom<Contact>.Where<Contact.displayName.IsNotNull.And<Contact.displayName.Contains<@P.AsString>>>.View.SelectWindowed(graph1, startIdx, 1, ' ').First();

    var firstName = contact.GetItem<Contact>().DisplayName.Split(' ')[0];
    var secondName = contact.GetItem<Contact>().DisplayName.Split(' ')[1];


}
sw1.Stop();
sb.Append($"Performance Select");
sb.Append(Guid.NewGuid().ToString());
```

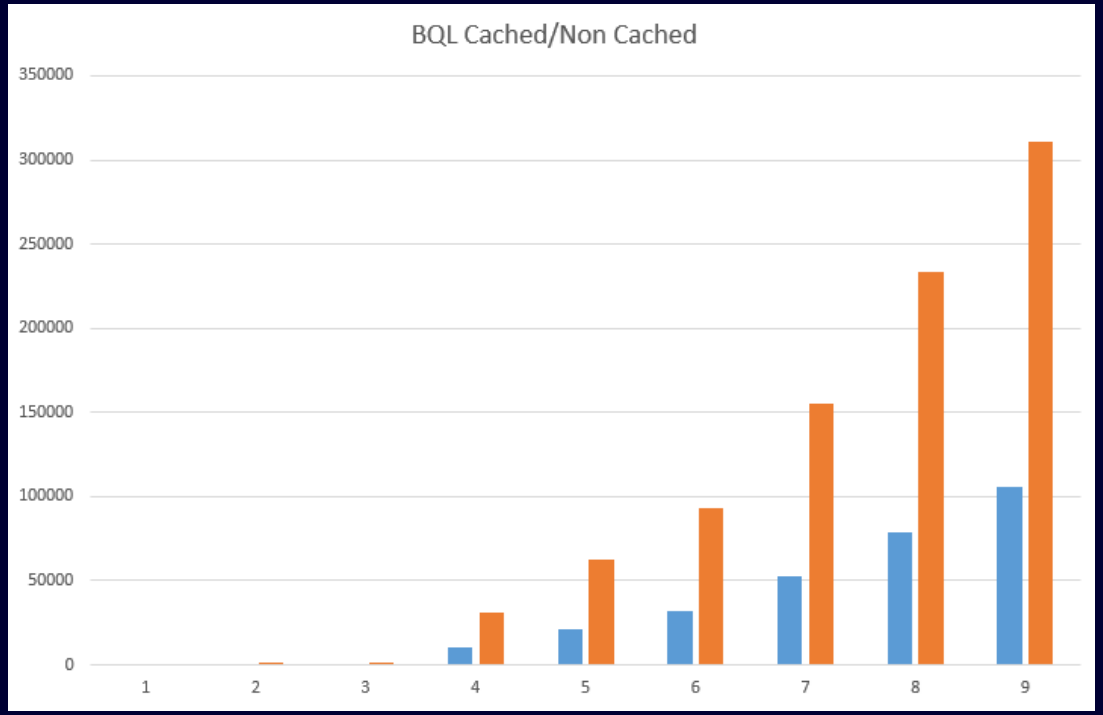
There is no clear winner in case of caching of graph



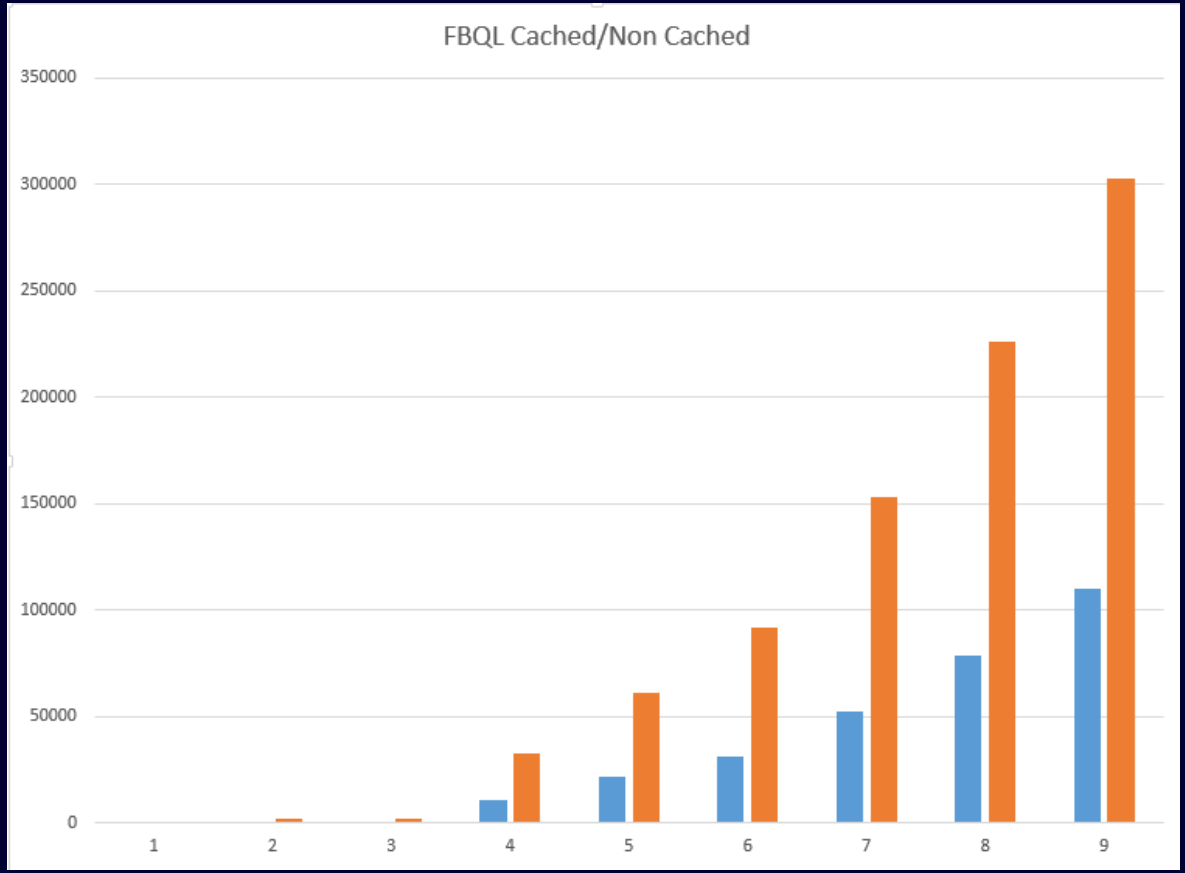
But in non Caching scenario BQL ~3% faster than FBQL



As you see, cached BQL is 3 times faster



Similar stats are true for FBQL



Random search+Persist

Cached vs non Cached

BQL

```
var sw = new Stopwatch();
sw.Start();

var graph0 = PXGraph.CreateInstance<FBQLStudent1>();

for (int i = 0; i < numberOfIterations; i++)
{
    int startIdx = rand.Next(numberOfNotNull);

    if (cachingOfGraph)
    {
        graph0 = PXGraph.CreateInstance<FBQLStudent1>();
    }
    var contact = PXSelect<Contact, Where<Contact.displayName, IsNotNull, And<Contact.displayName, Contains<Required<Contact.displayName>>>>>.SelectWindowed(graph0, startIdx, 1, ' ').First();

    var firstName = contact.GetItem<Contact>().DisplayName.Split(' ')[0];
    var secondName = contact.GetItem<Contact>().DisplayName.Split(' ')[1];
    graph0.Clear();

    var student = new Student1();
    student.StudentCD = firstName.ToUpper() + secondName.ToUpper();
    student = graph0.Students.Insert(student);

    student.FirstName = firstName;
    student.LastName = secondName;
    graph0.Students.Update(student);
    graph0.Persist();
}
sw.Stop();
sb.Append($"Classical way took {sw.ElapsedMilliseconds} milliseconds on {numberOfIterations} of iterations");
```

Stopwatch for measuring performance

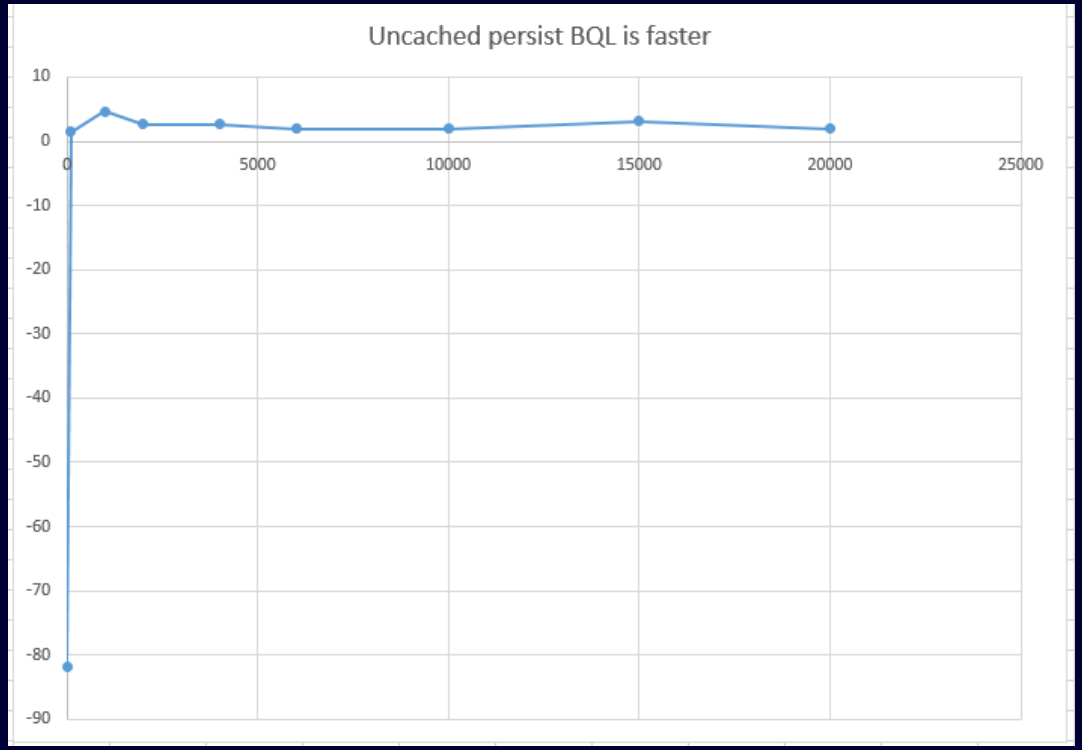
Classical search

Saving to database

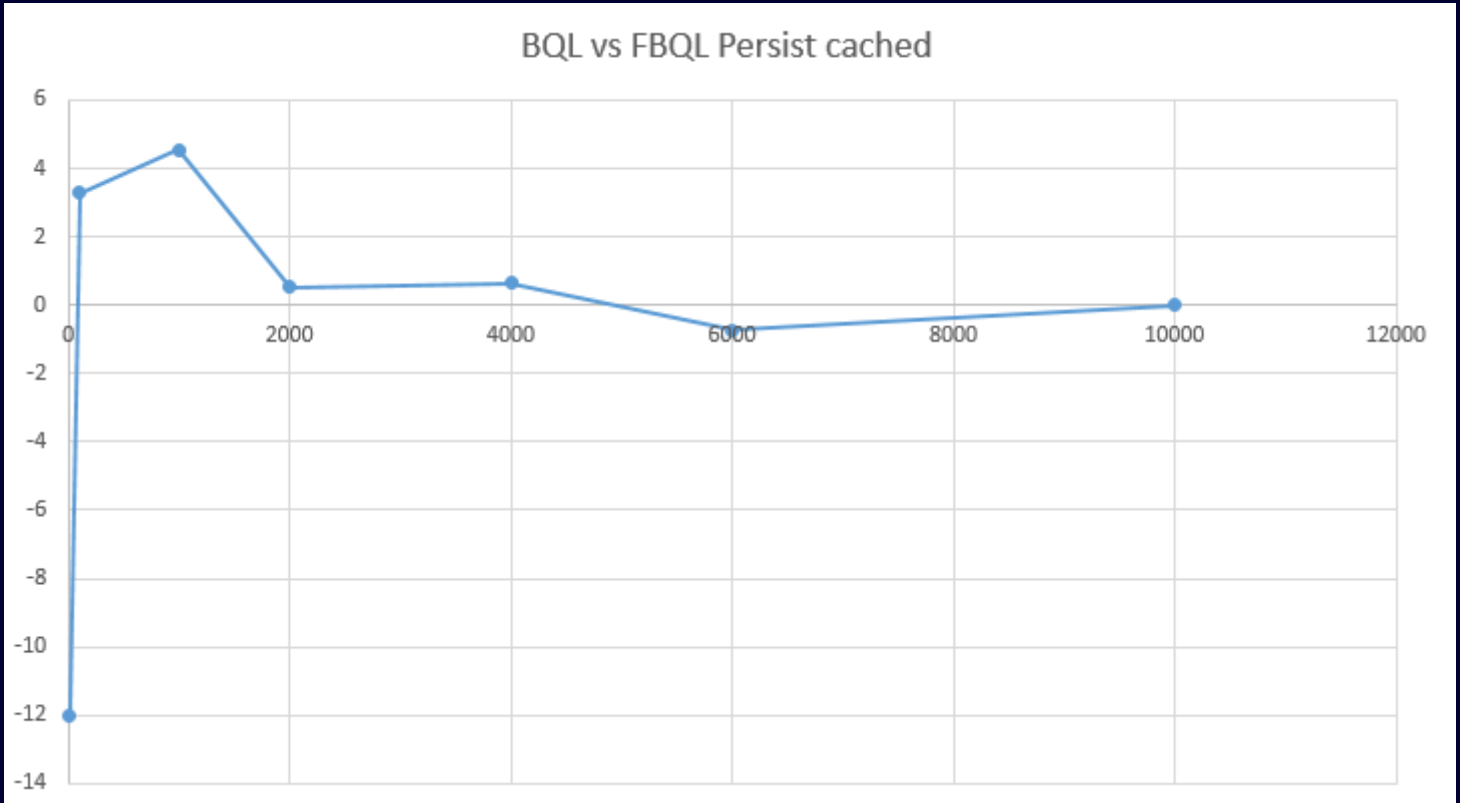
FBQL Search and Save

```
var sw1 = new Stopwatch();
sw1.Start();

var graph1 = PXGraph.CreateInstance<FBQLStudent2>();

for (int i = 0; i < numberOfIterations; i++)
{
    int startIdx = rand.Next(numberOfNotNull);

    if (cachingOfGraph)   Cache/not cache graph
    {
        graph1 = PXGraph.CreateInstance<FBQLStudent2>();
    }
       FBQL Search
    //var contact = PXSelect<Contact, Where<Contact.displayName, IsNotNull, And<Contact.displayName, Contains<Required<Contact.displayName>>>>>.SelectWindowed(graph, startIdx, 1, ' ').First();
    var contact = SelectFrom<Contact>.Where<Contact.displayName.IsNotNull.And<Contact.displayName.Contains<@P.AsString>>>.View.SelectWindowed(graph1, startIdx, 1, ' ').First();


    var firstName = contact.GetItem<Contact>().DisplayName.Split(' ')[0];
    var secondName = contact.GetItem<Contact>().DisplayName.Split(' ')[1];
    graph1.Clear();

    var student = new Student2();
    student.StudentCD = firstName.ToUpper() + secondName.ToUpper();
    student = graph1.Students.Insert(student);

    student.FirstName = firstName;
    student.LastName = secondName;
    graph1.Students.Update(student);
    graph1.Persist();   Saving to database
}
sw1.Stop();
```

Uncached persist BQL is faster. But not on small dataset.



BQL vs FBQL cached persist. FBQL may be faster!



Summary

- On big amounts of data use caching
- In our measurements time to production is 15 - 20% faster
- In scale of one week you get 1 more day in the sprint
- Easier maintainability (easier to read/modify) i.e. fixing of bugs
- Potential 3% slowdown will not be noticed, but new features which you'll implement customer will notice immediately