Frozen Plans and Parallel Queries (2017)

Brendan Bannon Support Manager

First a Little Background

SQL Statement Index

SQL Statement Index

A Statement has 5 important pieces:

- Query Text
- Hash
- Plan
- Basic Runtime Stats
- Frozen State

- The Statement Index has everything you need to regenerate the same INT code for a given plan.

SQL Statement Index: How it works

When we compile an new SQL Query, we add it to an index of all the SQL statements that were compiled in that namespace.

When that query comes in (ODBC, Dynamic, Embedded), we check it against the index, If it is found in the index that Statement is used

One Statement can show up in multiple routines: xDBC Cached Query, Dynamic Cached Query, Embedded SQL.

Statement Index: Technical Details

Statements are stored in tables (eq. objects).

- INFORMATION_SCHEMA.STATEMENTS
- INFORMATION_SCHEMA.STATEMENT*
- Mapped to ^rINDEXSQL(“sqlidx”)

Statements are standardized prior to storage.

- Uppercase with space normalization & parameterization.

Frozen Statements remain after a query purge.

- Unfrozen statements do not.

Statement Index: Additional Details

Your statements can be imported and exported.

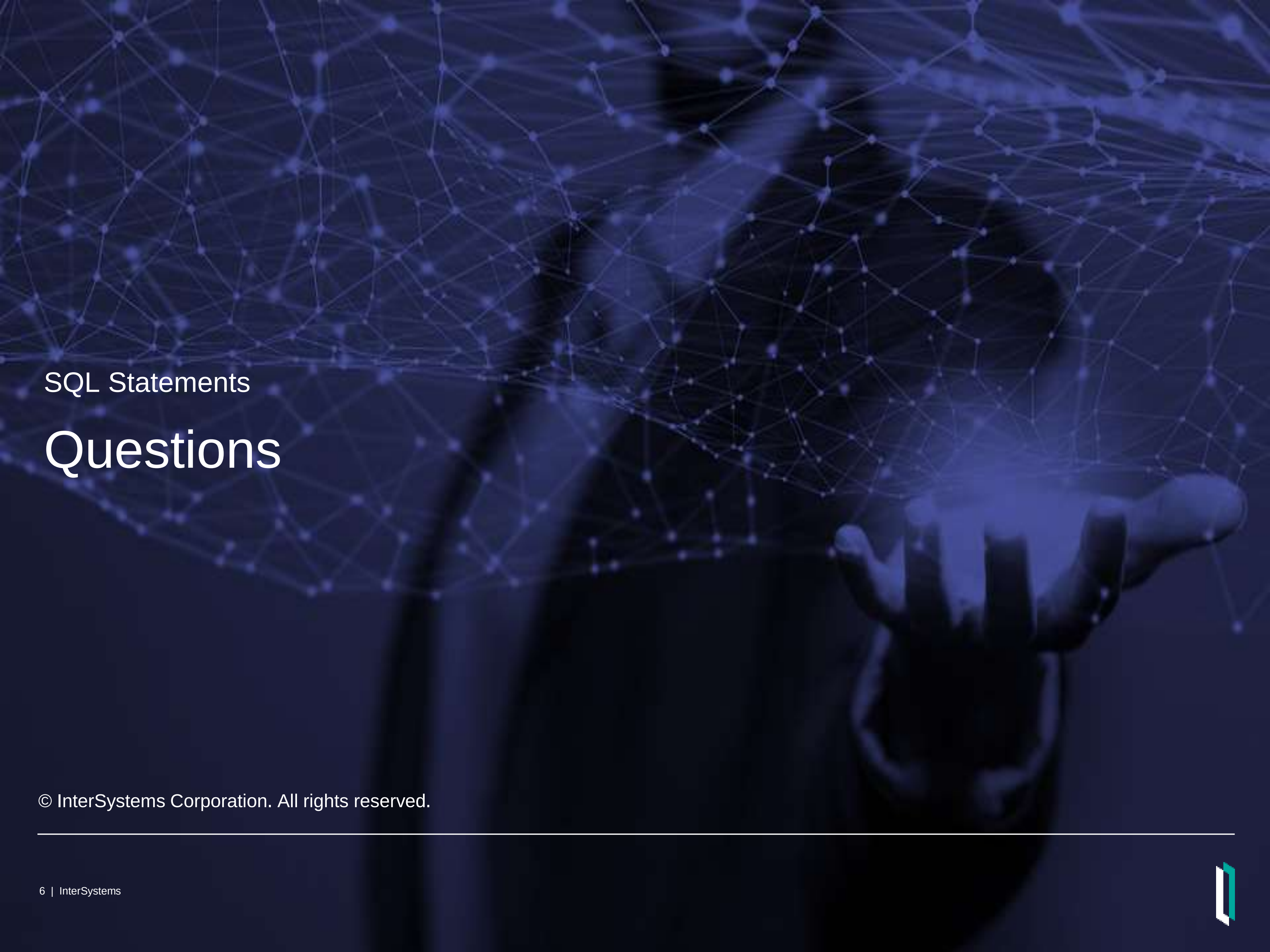
- Implication: you can move frozen plans between machines.

Statements must be the same to benefit from freezing.

- Changing the SELECT-list order changes the statement.

Statement can be managed in 3 ways:

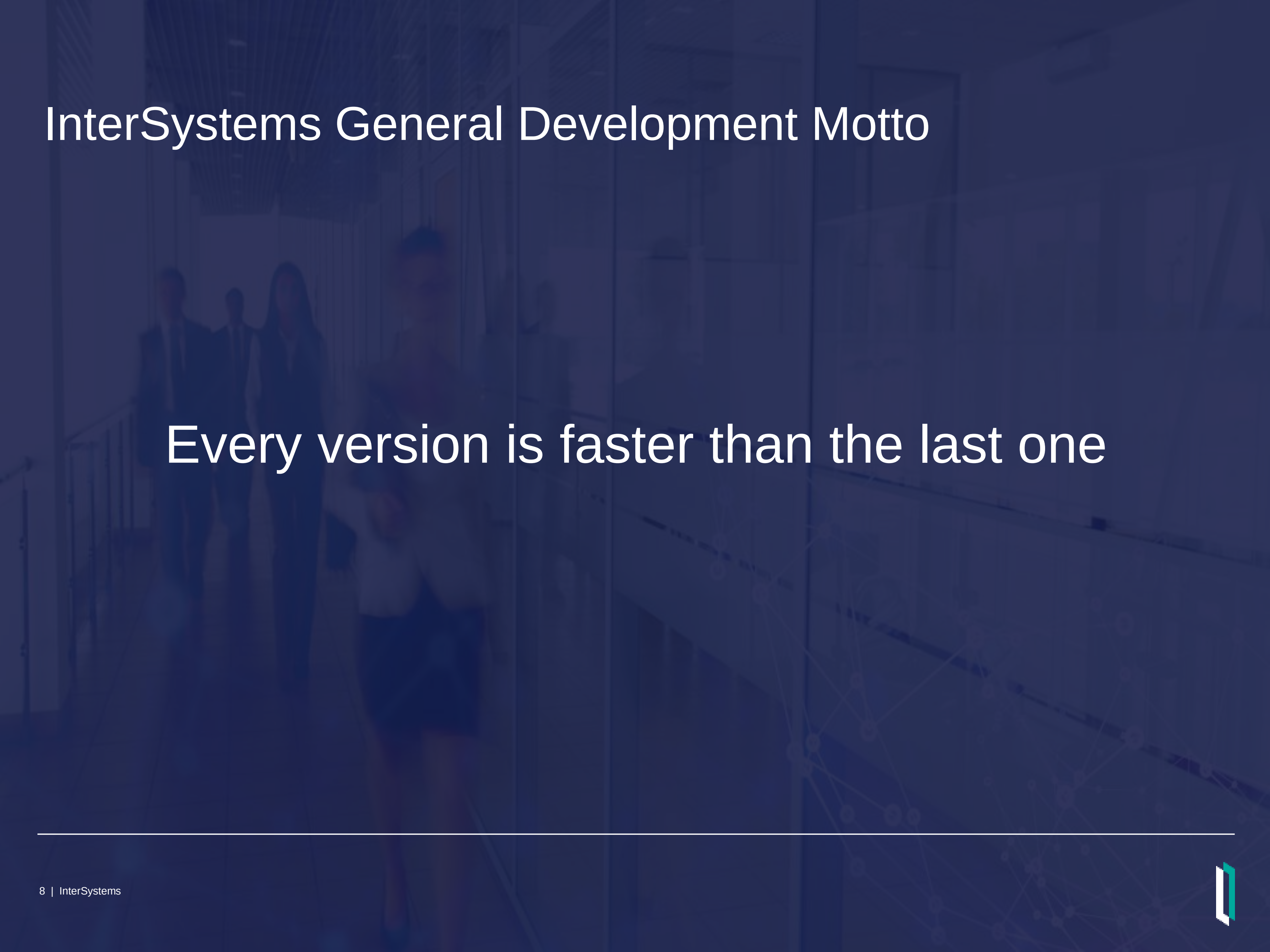
- INFORMATION_SCHEMA tables/objects.
- $SYSTEM.SQL API
- Management Portal
  - Home->System Explorer->SQL: Statements tab



SQL Statements

Questions

Frozen Plans

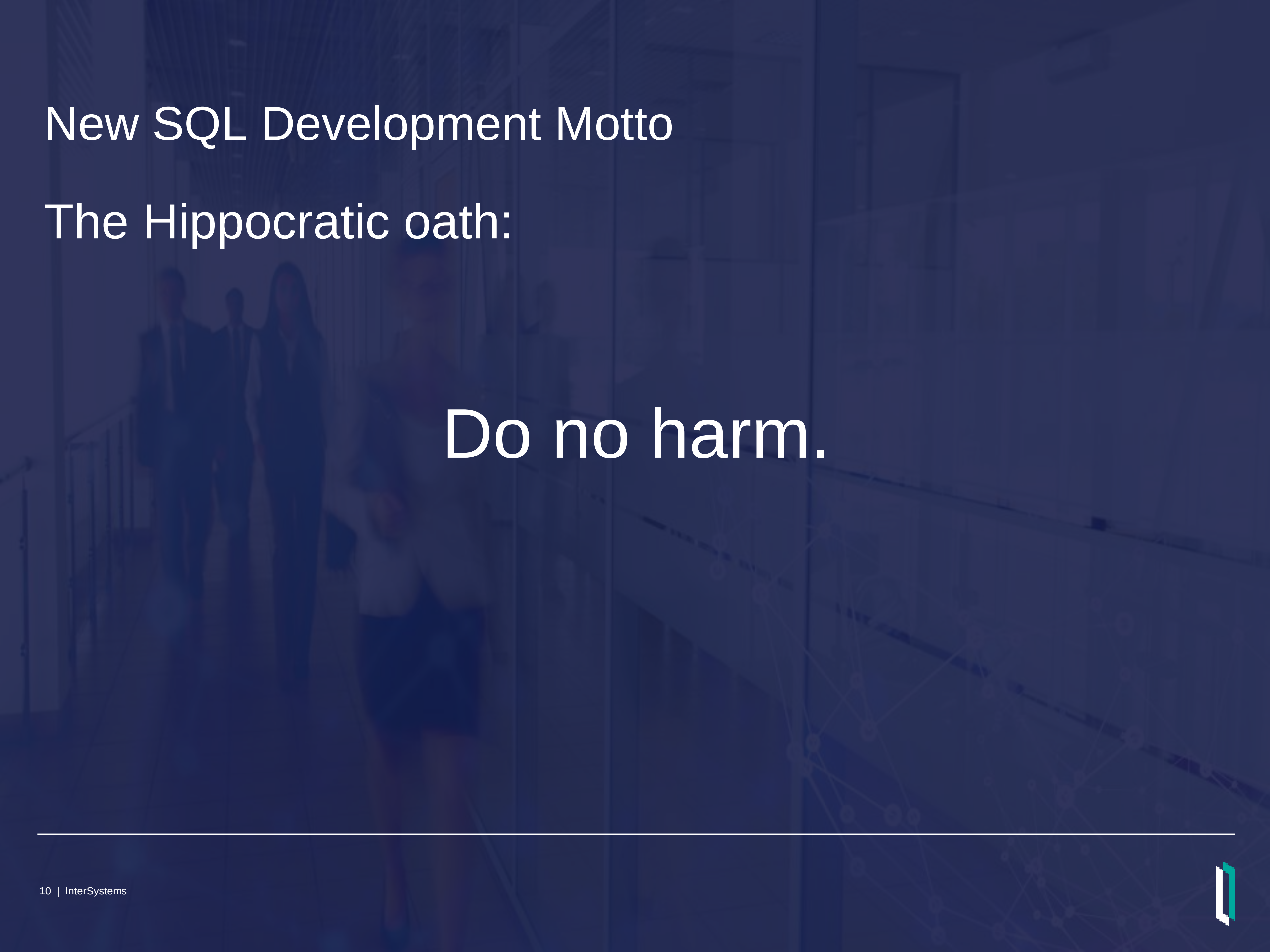


InterSystems General Development Motto

Every version is faster than the last one

Some Examples with Negative Side Effects

- Outlier Selectivity
  - If a property had one value that shows up in the data way more than any other this is an Outlier
  - TuneTable will now store the Selectivity of this Outlier
  - The Selectivity of the other values will be calculated without the Outlier value
  - Development assumed the most common Outlier would be NULL
- Union Or Optimization
  - Rewrite queries with OR conditions to be UNIONs of the different ORs
  - If indices where available for the OR conditions they would be used efficiently in the UNION legs



New SQL Development Motto

The Hippocratic oath:

Do no harm.

Frozen Plans

Starting in 2016.2 you have the ability to Freeze a query plan.

A frozen plan will not be changed by:

Cache Upgrade,

TuneTable,

Cached Query Purge,

Class Compilation,

New Indices,

……

It is possible a frozen plan will become invalid, for example if an index is deleted.

The plan will be unfrozen.

A new plan will be generated.

An error will be recorded in the Statement index.

This is all invisible to the user.

DPV4838 – 2017.1 and up
Option for an SQL Statement

- Unfrozen

Unfrozen means the query plan will be newly determined by the query optimizer when the statement is prepared/compiled. The is the default plan state when a statement is first prepared/compiled.

- Frozen/Explicit

Frozen/Explicit means this query plan was frozen by an explicit request, not by a prepare/compile after an upgrade. Someone clicked the "Freeze Plan" button in the SQL Statement Details screen in the Management Portal, or called $SYSTEM.SQL.FreezePlan() to explicitly freeze the plan for this statement.

- Frozen/Upgrade

Frozen/Upgrade means this query plan was frozen because it was prepared/compiled under a version newer than the version plan was originally created in. For example, suppose this statement was prepared/compiled under version 2016.2. Then you upgrade to version 2017.1, and the statement is prepared/compiled again. The system will detect this is the first prepare/compile of the statement on the new version, and automatically mark the plan state as Frozen/Upgrade and use the existing plan for the new prepare/compile. This ensures the query plan used is no worse than the plan of the previous version.

MAK4642 – 2017.2 and up
Detect if a frozen plan is different to the current plan

In the SQL statement index table a new column was added to show if the frozen plan is different from the un-frozen plan, this only applies to queries where the plan is frozen.

In the SQL statement detail page also display this 'frozen plan is different from current plan' flag. Also there is a button to force a manual check of this statement.

A nightly task will review frozen plans to see if a different plan would be generated for that Statement.

Users can run the query adding the hint, %NOFPLAN to the SELECT clause of the query to see how the new plan will perform.

Cache Upgrade Strategy

Current Upgrading Techniques

Best: Full test upgrade on a system with representative data.

Good: Export Query Plans, test upgrade, Export Query Plans, diff.

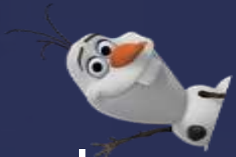
OK: Upgrade, test some queries, have snapshot to revert back to.

Bad: Upgrade and go. Call Support if there are problems.

Worse: Upgrade and try to fix all problems and never call Support. Complain loudly.

Worst: Upgrade. Set computer on fire if there are problems. Cry yourself to sleep.

DPV4838 – 2017.2 and up
Implement automatic freeze of query plans on upgrade



In version 2016.2, we added the ability to record SQL statement plans in the statement index, and the ability to freeze a statement plan.

With this change, we further enhance this capability to automatically freeze statement plans upon upgrade. The goal of this change is that the upgrade procedure will make your queries run no slower than they did in the previous version.

The version differences only take into account major.minor versions. Upgrading from 2016.2.* to 2016.3.* will be detected as a version change. Upgrading a maintenance release level, say from 2016.3.1 to 2016.2.3, is not considered a version change for a plan version.

What will Happen Now?

This means that any customer doing an upgrade where the old system is on 2016.2 or higher WILL NOT SEE ANY SQL CHANGES on upgrade.

The customer will need to manually go in and unfreeze the plans to get any new optimizations from the new version.

Good News: There should be no more calls from customer crying that they did an upgrade and the SQL is all slow now.

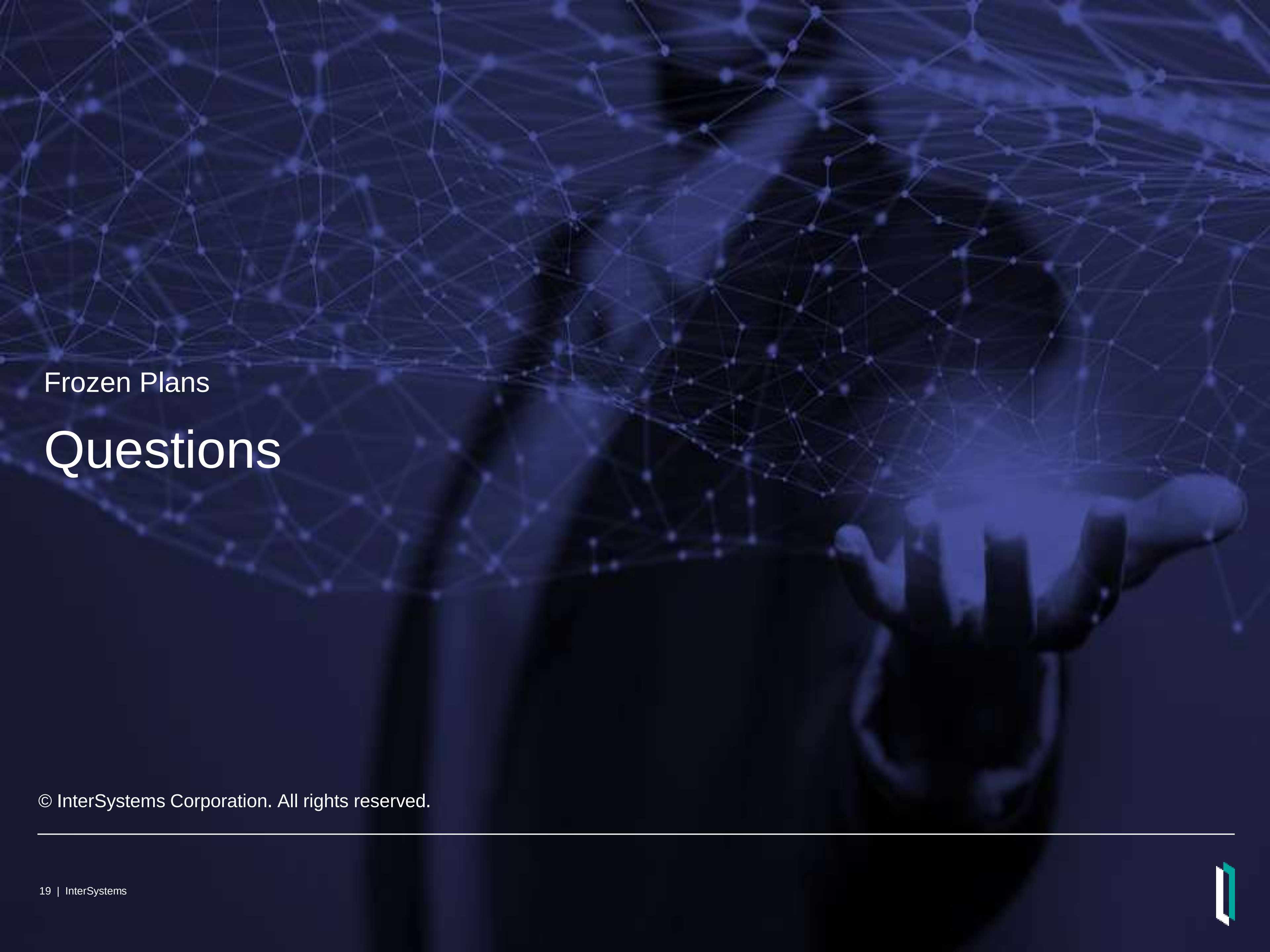
Bad News: Many customers will not know about this new behavior and will continue to run the old plans and not get the benefits of the new version.

New Support Options

Now when you are working on a performance problem and you want to reproduce what the customer is seeing you can ask them to freeze the SQL Statement and export it for you.

You will still need to get all the classes and clean them and ………. (go to Patrick's SQL Stats class for help with that)

If you make changes and want the customer to test your new version you can send them back the statement in its' new form and know that they are testing the plan you want them to.



Frozen Plans

Questions

%Parallel Queries

2016.2: Parallelization Improvements

%Parallel allows for multiple jobs to work on a singular query

Good for “big” queries, aggregates, UNION/OR, memory work

- SELECT … FROM %PARALLEL …
  - Makes queries faster by using more resources
    - Sometimes all the resources
      - You can turn your query in to a monster

The Query

```
SELECT Color, COUNT(SpikeCount) As "Total Spikes"
FROM SQLUser.Widget
WHERE Color like ‘S%'
GROUP BY Color
```

The Normal Plan

```
<cost value="11188"/>
Call module B, which populates temp-file A.
Read temp-file A, looping on the hashing subscript.
For each row:
    Output the row.
<module name="B">
Read index map SQLUser.Widget.ParellelPerfection, looping on %SQLUPPER(Color) (with a
%STARTSWITH range condition), SpikeCount, and ID.
For each row:
    Check distinct values for %SQLUPPER(Color) using temp-file A,
        subscripted by a hashing of these values.
    For each distinct row:
        Add a row to temp-file A, subscripted by the hashing,
            with node data of SpikeCount and %SQLUPPER(Color).
    Update the accumulated count(SpikeCount) in temp-file A,
        subscripted by the hashing.
</module>
```

The Numbers

Row count: **3**

Performance:

- **23.701** seconds
- **1,878,068** global references
- **84,507,872** lines executed
- **0** disk read latency (ms)

Color	Total Spikes
SALMON	626280
SARCOLINE	625105
SMARGADINE	626524

The %Parallel Query

```
SELECT Color, COUNT(SpikeCount) As "Total Spikes"
FROM %PARALLEL SQLUser.Widget
WHERE Color like ‘S%’
GROUP BY Color
```

The %Parallel Plan

<cost value="11188"/>

Process query in parallel, partitioning index map SQLUser.Widget.ParellelPerfection into subranges of %SQLUPPER(T1.Color) values, piping results to temp-file A:

```
SELECT count(T1.SpikeCount),%SQLUPPER(T1.Color)
FROM  %NOPARALLEL SQLUser.Widget T1
WHERE ((%SQLUPPER(T1.Color) LIKE (%SQLUPPER(?))))
GROUP BY %SQLUPPER(T1.Color)
```

Read temp-file A, looping on a counter.

For each row:

    Output the row.

The Numbers

Row count: **3**

Performance:

- **11.528** seconds
- **1,878,384** global references
- **84,516,753** lines executed
- **0** disk read latency (ms)

Color	Total Spikes
SALMON	626280
SARCOLINE	625105
SMARGADINE	626524

The guts

```
 s %cur0188d(8)=##class(%SQL.Agent).%RunWork($name(^User.WidgetI("ParellelPerfection")),
                                                  $s(%cur0188d(6)=$c(0):"",1:%cur0188d(6)),"",
                                                  $system.Rtn.GetCurrent(),
                                                  "$$%0AmBm2l1t",$lb(),
                                                  1)



%0AmCk1 for {
                                         try { s %cur0188d(10)=%cur0188d(8).Get(20) }
                                         catch { i '$isobject($g(%cur0188d(8))) { s %cur0188d(9)=""  g %0AmCdun q } throw }
                                         i $l(%cur0188d(10)) q
                                         i %cur0188d(8).SQLDecode() { s %cur0188d(9)="" k %cur0188d(8)  g %0AmCdun q
}
                    }
                    Set %cur0188d(11)=$lg(%cur0188d(10)),%cur0188d(12)=$lg(%cur0188d(10),2)
```

Guts of the Guts – How many jobs will my query use?

##class(%SQL.Agent).%RunWork() calls: $$calcJobs^%SYS.WorkQueueMgr()

To get the number of jobs. calcJobs looks at:

- $zu(204,4) – number of CPUs
- $zu(204,5) – number of cores
- Global $g(^%SYS("Compiler","MultiCompileMax") ,32) if set for max number of process it can use
- Global ^%ISCWorkQueue("Jobs",0) to see how many active worker jobs are on the system already

Based on all that info it will return the number of jobs to start up, somewhere between 2 and 32 would be the default.

More Guts

When a Parallel Query is generated we will produce 2 SQL queries, the one the user wrote and the one the parallel processes will run. You can see the second query in the Show Plan.

```
SELECT count(T1.SpikeCount),%SQLUPPER(T1.Color)
FROM  %NOPARALLEL SQLUser.Widget T1
WHERE ((%SQLUPPER(T1.Color) LIKE (%SQLUPPER(?))))
GROUP BY %SQLUPPER(T1.Color)
```

The place to start looking for the first query will still be %0Afirst. For the second query you should start at %0Bfirst.

With embedded SQL the names might be different as one routine and have a bunch of SQL statements and the tags will use the cursor name to be unique.

Debugging What Happened

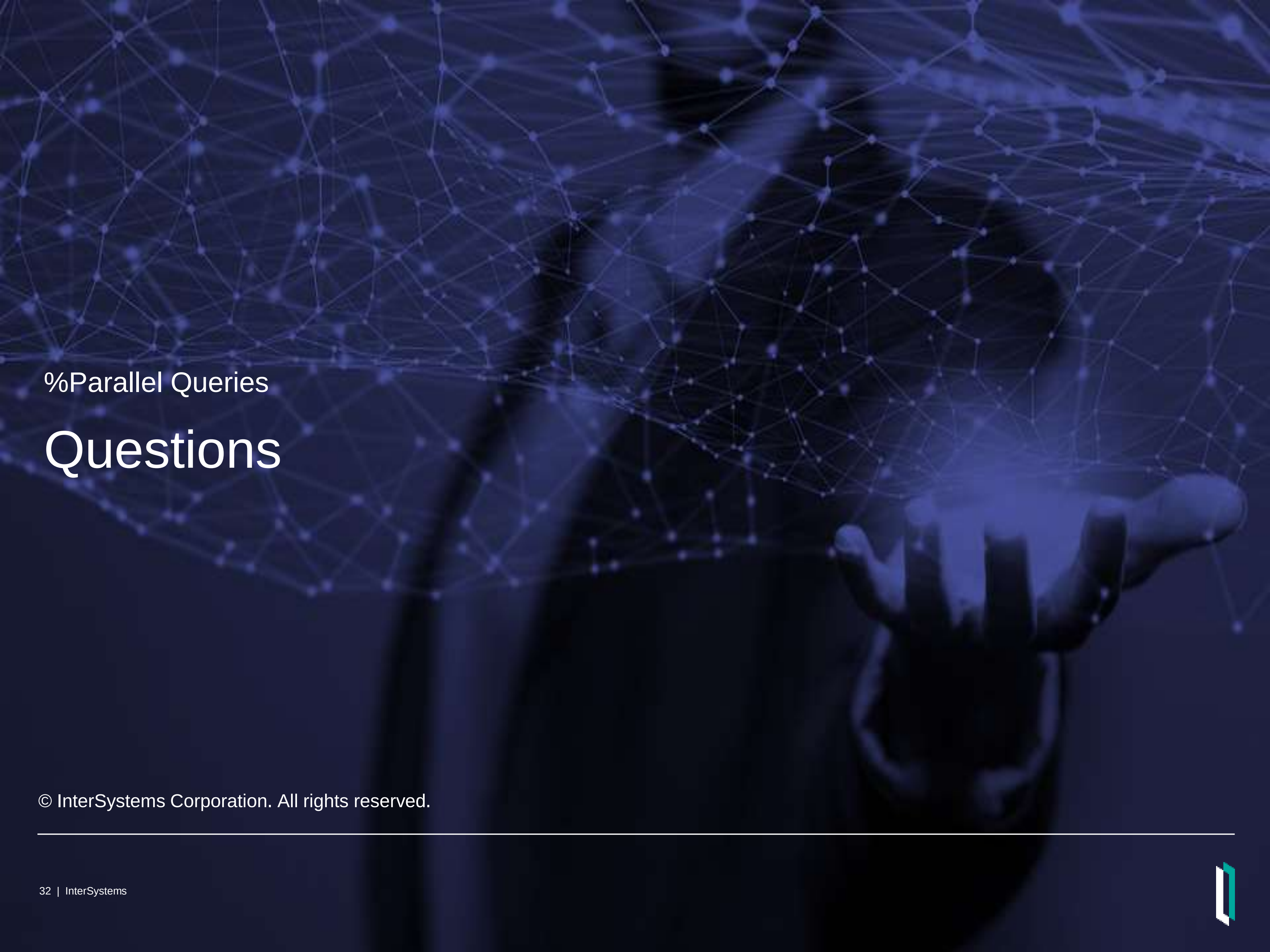
Adding a line of debugging to the generated INT code we can see what the 4 different jobs did.

```
%0Bfirst
 s ^bjb("%Parallel",$I(^bjb))=$LB($g(start),$g(end))
```

After running the query I have the following global

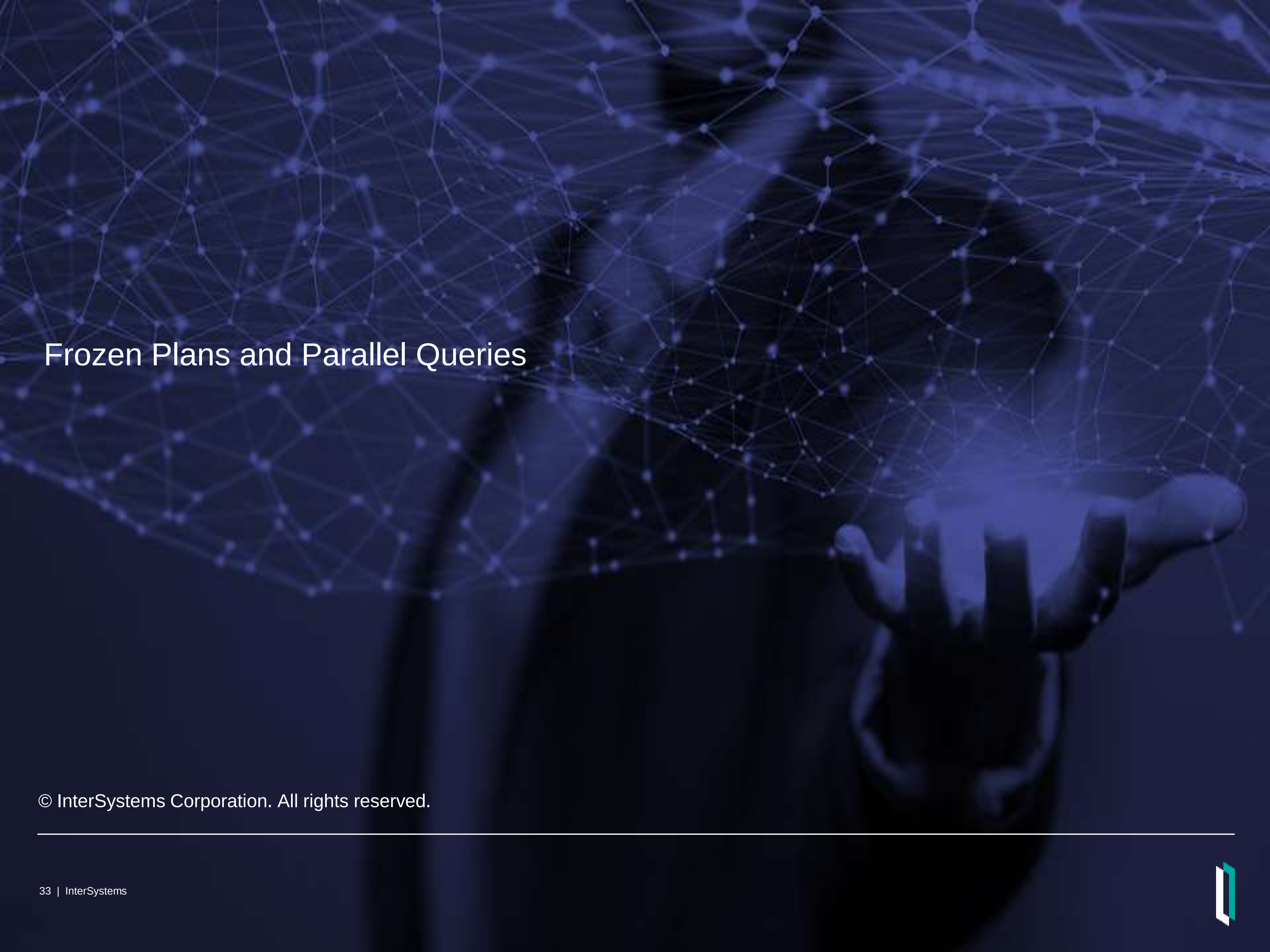
```
EUROWITS2017>zw ^bjb
^bjb=4
^bjb("%Parallel",1)=$lb(" RED"," SALMON")
^bjb("%Parallel",2)=$lb(" SALMON"," SARCOLINE")
^bjb("%Parallel",3)=$lb(" SARCOLINE"," SMARGADINE")
^bjb("%Parallel",4)=$lb(" SMARGADINE"," YELLOW")
```

We start a little early, “ RED” and we end a little late, “ YELLOW” but this is based on the disk blocks, the generated code is still testing WHERE Color LIKE ‘S%’



%Parallel Queries

Questions



Frozen Plans and Parallel Queries