Article
Benjamin De Boe · Sep 13, 2022 8m read

CI/CD with IRIS SQL

In the vast and varied SQL database market, InterSystems IRIS stands out as a platform that goes way beyond just SQL, offering a seamless multimodel experience and supporting a rich set of development paradigms. Especially the advanced Object-Relational engine has helped organizations use the best-fit development approach for each facet of their data-intensive workloads, for example ingesting data through Objects and simultaneously querying it through SQL. Persistent Classes correspond to SQL tables, their properties to table columns and business logic is easily accessed using User-Defined Functions or Stored Procedures. In this article, we'll zoom in on a little bit of the magic just below the surface, and discuss how it may affect your development and deployment practices. This is an area of the product where we have plans to evolve and improve, so please don't hesitate to share your views and experiences using the comments section below.

⚠ This part of the product is changing significantly with IRIS 2025.2, please see this article for more detail. The information below applies to 2025.1 and earlier releases.

Saving the Storage Definition

Writing brand new business logic is easy, and assuming you have well-defined APIs and specifications, adapting or extending it usually is too. But when it's not just business logic, but also involves persistent data, anything you change from the initial version will need to be able to cope with data that was ingested through that earlier version.

On InterSystems IRIS, data and code coexist in a single high-performance engine, without the half dozen abstraction layers you might see in other 3GL or 4GL programming frameworks. This means there's only a very thin and transparent mapping to translate your class' properties to $list positions in a global node per row of data when using default storage. If you add or remove properties, you don't want the data from a removed property to show up under a new property. This mapping of your class' properties is what the Storage Definition takes care of, a somewhat cryptic XML block you may have noticed at the bottom of your class definition. The first time you compile a class, a new Storage Definition gets generated based on the class' properties and parameters. When you make changes to your class definition, at recompile time those changes are reconciled with the existing Storage Definition and it gets amended such that it remains compatible with existing data. So while you are going out on a limb refactoring your classes, the Storage Definition carefully considers your earlier creativity and ensures both old and new data remain accessible. We call this **schema evolution**.

In most other SQL databases the physical storage of your tables is much more opaque if visible at all, and changes can only be made through ALTER TABLE statements. Those are standard DDL (Data Definition Language) commands, but typically much less expressive than what you can achieve modifying a class definition and procedural code on IRIS directly.

At InterSystems, we strive to offer IRIS developers the ability to cleanly separate their code and data, as that is crucial to ensure smooth packaging and deployment of your applications. The Storage Definition plays a unique role in this, as it captures how the one maps to the other. That's why it's worth a closer look in the context of general development practices and CI/CD pipelines in particular.

Exporting to UDL

In the current century, source code management is file-based, so let's first take a look at IRIS' main file export format. The **Universal Description Language** is, as its name suggests, meant to be a universal file format for any and all code you write on InterSystems IRIS. It is the default export format when working with the VS Code ObjectScript plug-in and leads to easy-to-read files that resemble almost exactly what you'd see in an IDE, with an individual .cls file for each class (table) in your application. You can use $SYSTEM.OBJ.Export() to create UDL files explicitly, or just leverage the VS Code integration.

From the days of Studio, you may also remember an XML format that captured the same information as UDL and allowed grouping multiple classes into a single export. While that last part is convenient in some scenarios, it's a lot less practical to read and track differences across versions, so we'll ignore it for now.

Because UDL is meant to capture everything IRIS can express about a class, it will include all elements of a class definition, including the full Storage Definition. When importing a class definition that already includes a Storage Definition, IRIS will verify whether that Storage Definition covers all properties and indices of the class and if that is the case, just take it as-is and overwrite the previous Storage Definition for that class. This makes UDL a practical format for version management of classes and their Storage Definition, as it preserves that backwards compatibility for data ingested through prior versions of the class, wherever you deploy it to.

If you are a hardcore developer, you may wonder whether these Storage Definitions keep growing and whether this "baggage" needs to be carried around indefinitely. The purpose of Storage Definitions is to preserve compatibility with pre-existing data, so if you know there isn't any of that and you want to get rid of lengthy genealogy, you can "reset" your Storage Definition by removing it from the class definition and have the class compiler generate it anew. For example, you may use this to take advantage of newish best practices such as the use of Extent Sets, which implement hashed global names and separate each index into their own global, improving low-level efficiencies. For backwards compatibility within customer applications, we cannot universally change such defaults in the %Persistent superclass (though we will apply them when creating a table from scratch using the CREATE TABLE DDL command), so a periodic review of your classes and their storage is worthwhile. It is also possible to edit the Storage Definition XML directly, but users should exercise extreme caution as this may render existing data inaccessible.

So far so good. Storage Definitions offer a smart mapping between your classes and automatically adapt as the schema evolves. What else is in there?

Static vs Stats?

As you probably know, the InterSystems IRIS SQL engine makes advanced use of table statistics to identify the optimal query plan for any given statement the user executes. Table statistics include metrics on the size of a table, how values are distributed within a column, and much more. This information helps the IRIS SQL optimizer to decide which index is most beneficial, in what order to join tables, etcetera, so intuitively the more up-to-date your statistics are, the better chances you have to optimal query plans. Unfortunately, until we introduced fast block sampling in IRIS 2021.2, collecting accurate table statistics used to be a computationally expensive operation. Therefore, when customers deploy the same application to many environments in which the data patterns were largely the same, it made sense to consider the table statistics part of the application code and include them with the table definitions.

This is why on IRIS today you'll find the table statistics embedded inside the Storage Definition. When collecting table statistics through a manual call to TUNE TABLE or implicitly by querying it (see below), the new statistics are written to the Storage Definition and existing query plans for this table are invalidated so they can take advantage of the new statistics upon the next execution. Because they are part of the Storage Definition, these statistics will be part of UDL class exports and therefore may end up in your source code repository. In the case of carefully vetted statistics for a packaged application, this is desirable, as you'll want these specific statistics to drive query plan generation for all the application deployments.

Starting with 2021.2, IRIS will automatically collect table statistics at the start of query planning when querying a table that doesn't have any statistics at all and is eligible for fast block sampling. In our testing, the benefits of working with up-to-date statistics rather than no statistics at all clearly outweighed the cost of on-the-fly statistics gathering. For some customers however, this has had the unfortunate side effect that statistics gathered automatically on the developer's instance end up in the Storage Definition in the source control system and

eventually in the packaged application. Obviously, the data in that developer environment and therefore the statistics on it may not be representative for a real customer deployment and lead to suboptimal query plans.

This scenario is easy to avoid. Table statistics can be excluded from the class definition export by using the /exportselectivity=0 qualifier when calling $SYSTEM.OBJ.Export() , or use the /importselectivity flag value of your choice to avoid importing them along with the code. The system default for these flag can be found here, and changed using $SYSTEM.OBJ.SetQualifiers(...). You can then leave it up to the automatic collection in the eventual deployment to pick up representative stats, make explicit statistics collection part of your deployment process, which will overwrite anything that might have been packaged with the application, or manage your table stats separately through their own import/export functions: $SYSTEM.SQL.Stats.Table.Export() and Import().

We're currently working on a project to move table statistics to live with the data rather than be part of the code, and differentiate more cleanly between any *fixed* statistics configured explicitly by a developer and those *collected* from the actual data. Also, we're planning more automation with respect to periodically refreshing those statistics, based on how much the table data changes over time.

Wrapping up

In this article, we've outlined the role of a Storage Definition in IRIS ObjectRelational engine, how it supports schema evolution and what it means to include it in your source control system. We've also described why table statistics are currently stored in that Storage Definition and suggested development practices for making sure your application deployments end up with statistics that are representative for the actual customer data. As mentioned earlier, we're planning to further enhance these capabilities so we look forward to your feedback on the current and planned functionality and refine our design as appropriate.

#Best Practices #Continuous Delivery #Continuous Integration #Source Control #SQL #InterSystems IRIS