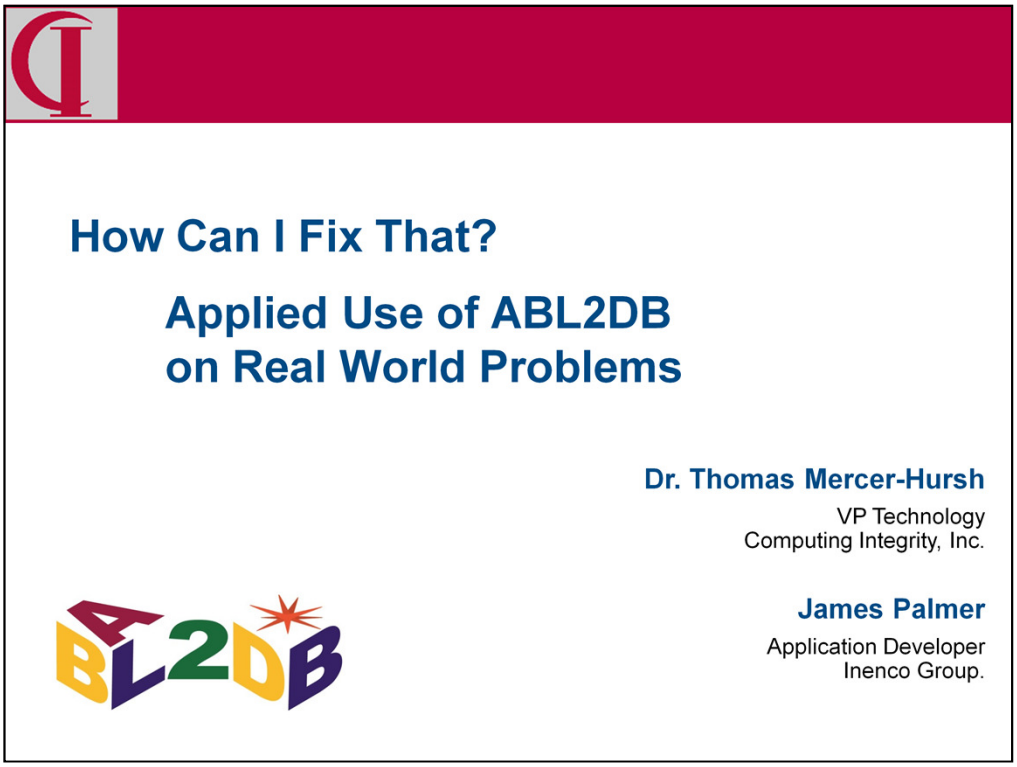
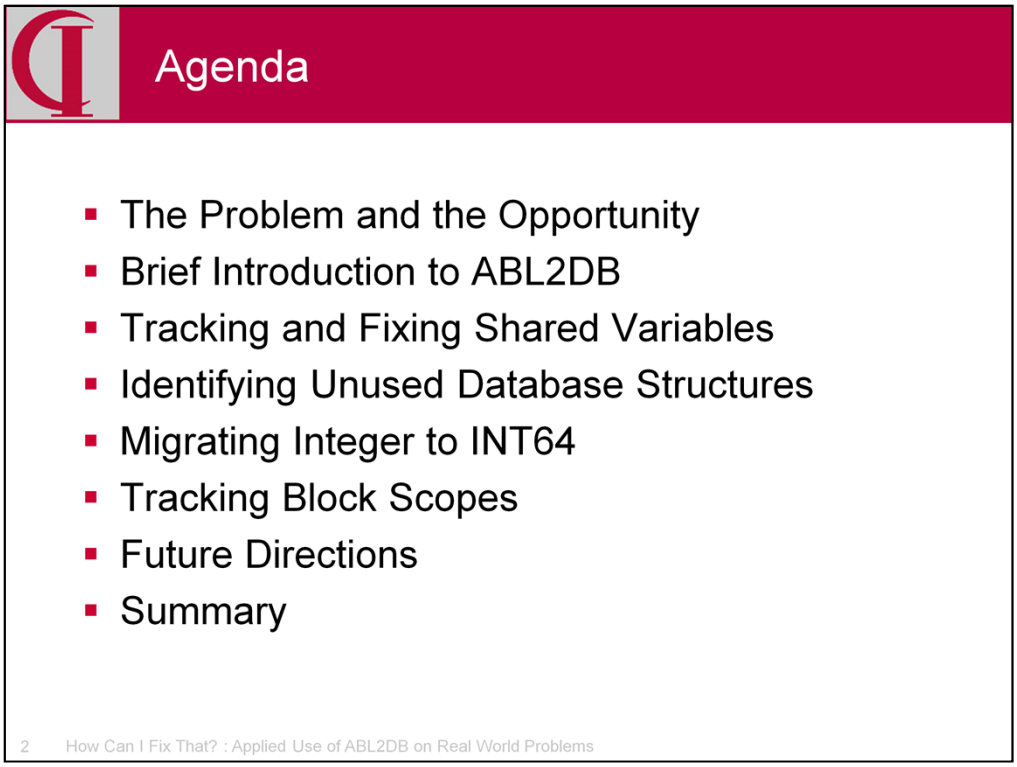


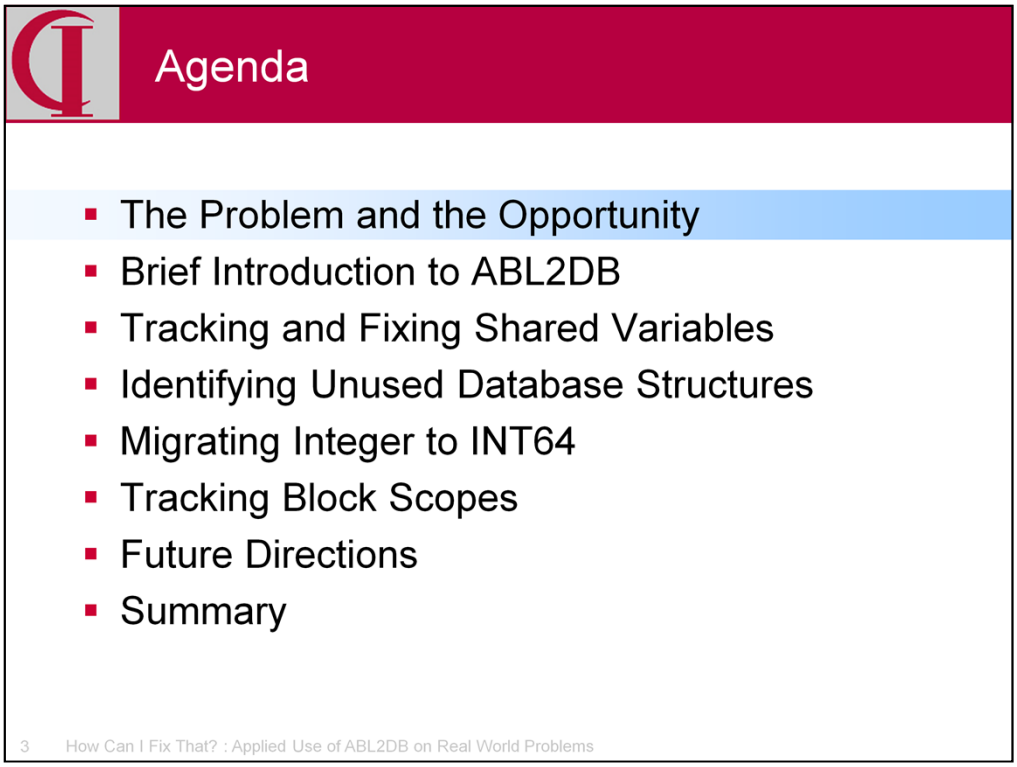
Let me begin by introducing myself. I have been a Progress Application Partner since 1986 and for many years I was the architect and chief developer for our ERP application. In recent years I have refocused on the problems of transforming and modernizing legacy ABL applications. This implies needing to understand what the existing legacy application is doing so that we can change or replace it with confidence. Today I am going to tell you about an open source tool I have been building.

Many of you will know James Palmer as Cringer from on-line forums like PSDN and ProgressTalk. James is an Application Developer and DBA for the Inenco Group, a company that assists companies with energy procurement and performance. They have a large Progress application which is one of the test beds we are using for ABL2DB.

James put in a lot of work on this project as my primary test site outside of my own code from my old Integrity/Solutions ERP package. Integrity/Solutions provided many interesting challenges and the Inenco code provided a whole new set. Happily, I was able to make changes to ABL2DB reasonably quickly to accommodate many of those differences. John Green has been very helpful in extending Proparse to handle some of the ABL syntax found in the Inenco code base and in fixing a couple of issues which have arisen. We seem to be past all those issues now.



Here is our agenda for today. First we are going to talk a little bit about the background, then look at four specific examples where ABL2DB can help us fix issues with our applications including both code and sample reports.



First, a little bit about the problem which the tool is intended to address.

The Problem and the Opportunity

- There is lots and lots of legacy ABL code out there (You are not alone!).
- Very little of it is meaningfully documented.
- Knowledge of what is in the code is very often based only on experience, which can be incomplete, unavailable, or even gone.

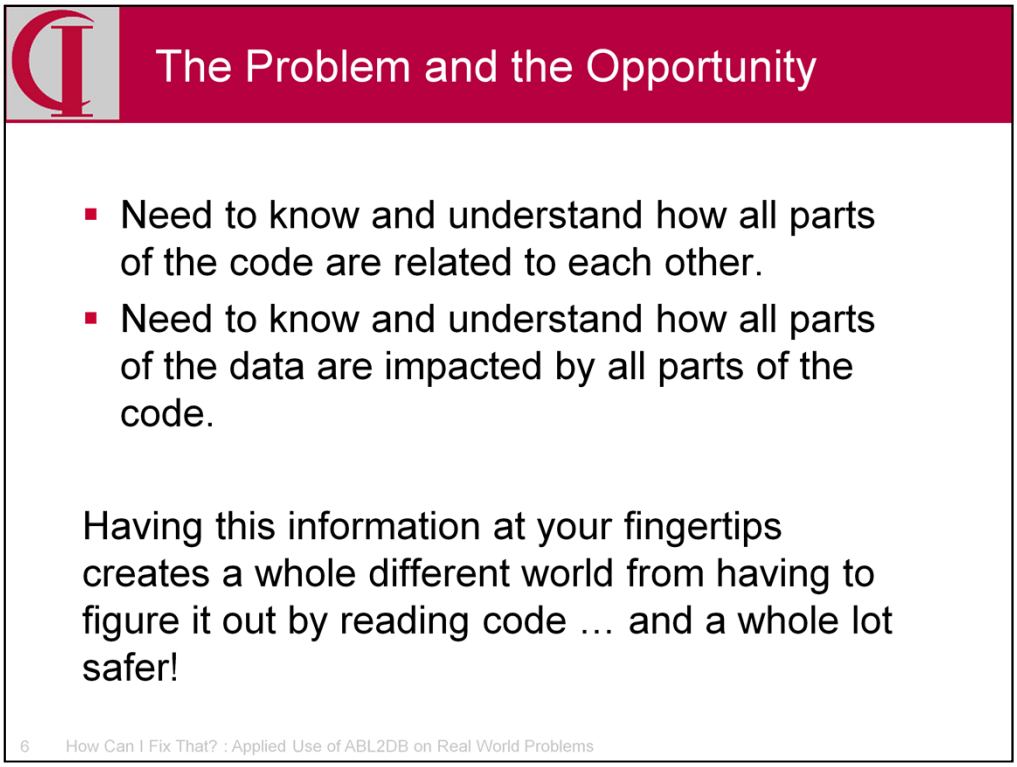
I'm sure that most of you who have been around for a while can identify the problem … large legacy ABL systems, often millions of lines of code, documentation which could be written on three napkins, and typically a reliance on some old hand who has been around for years to help everyone else figure out where things are and how they work … until he or she isn't there any more.

Moreover, often the code does not conform to what one would consider modern best practice … or even best practice at any time in the history of ABL in some cases. No matter what you think someone would never do, chances are that somewhere someone has done precisely that.

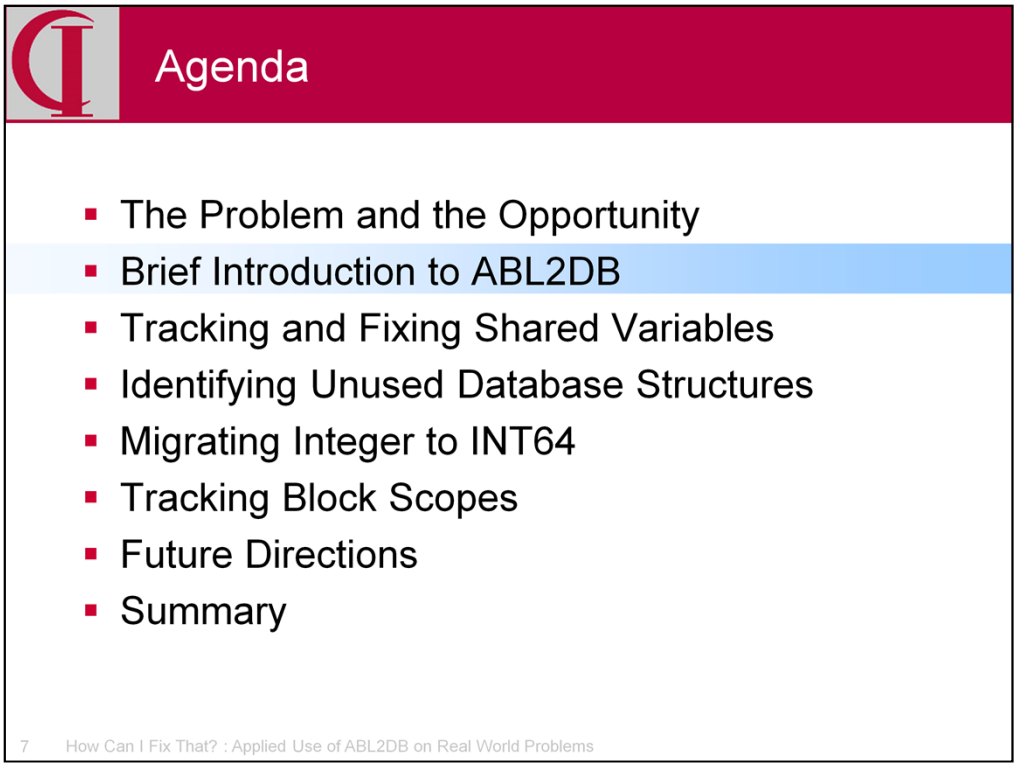
The Problem and the Opportunity

- Making changes safely requires solid knowledge of impact.
- Mistakes can be very costly and fixing them can be even more expensive than the original work.
- Worst cases can impact the integrity of the data.
- Business impact often exceeds the technical cost.

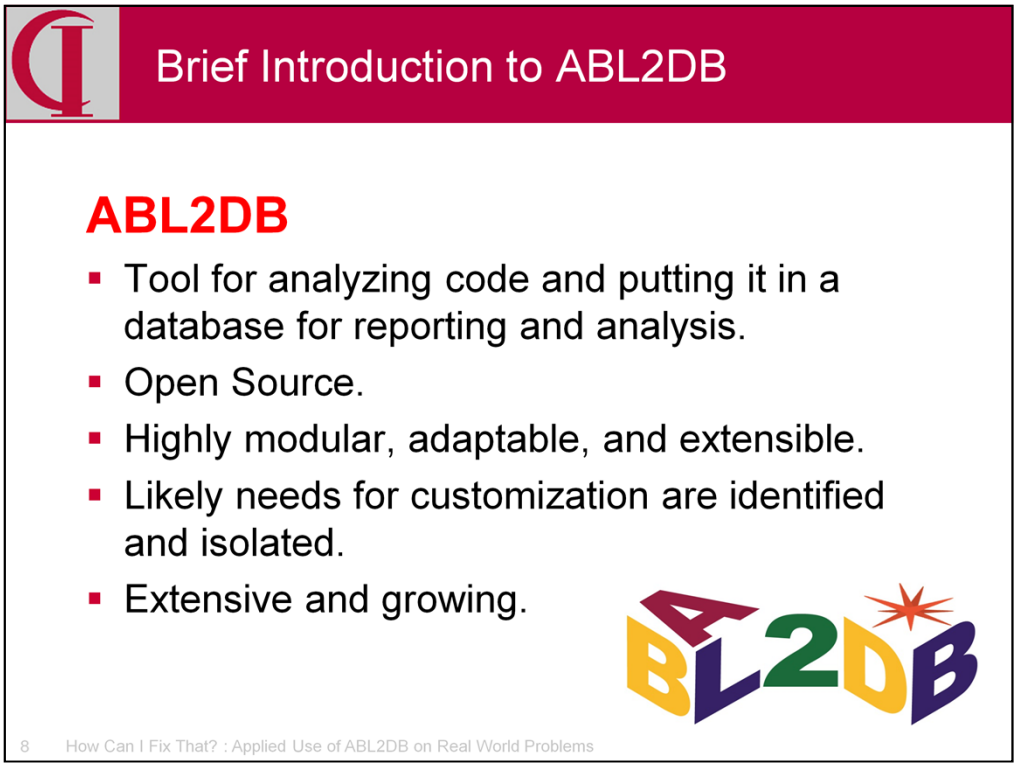
And we all know that when we make changes, we should understand what we are doing before we start … what will be impacted by any given change. But, without documentation, there isn't really any way to do that, so we make mistakes. Those mistakes often have a big impact, even on running the business. Not knowing the impact, we don't really even know what to test — so the problem can be in some area unrelated to where we did the work. In the worst cases, the cost impact to business operations exceeds all of the technical expense invested in making the original change and in repairing the damage.



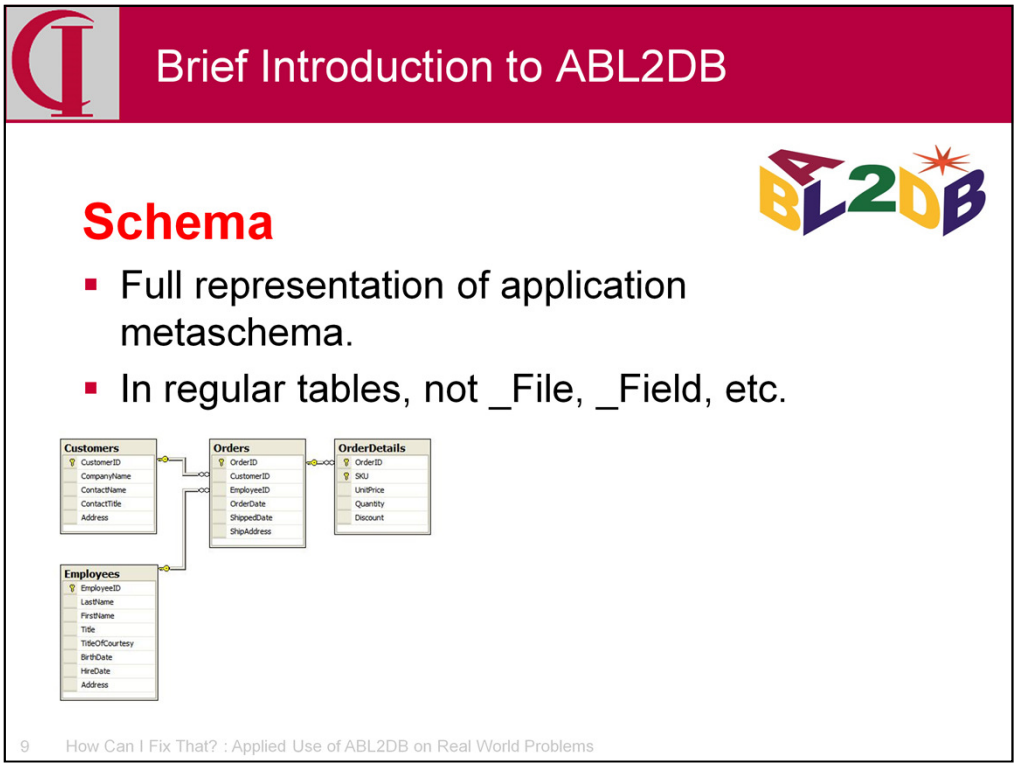
To make changes safely we need to understand how all parts of the code interact with other parts of the code and how all parts of the data are impacted by all parts of the code. We don't necessarily need to know everything all the time, but we do need to know anything relevant to the current project, so somewhere we need to have that knowledge. Having this information at your fingertips is a whole different world from having to figure it all out by reading code.



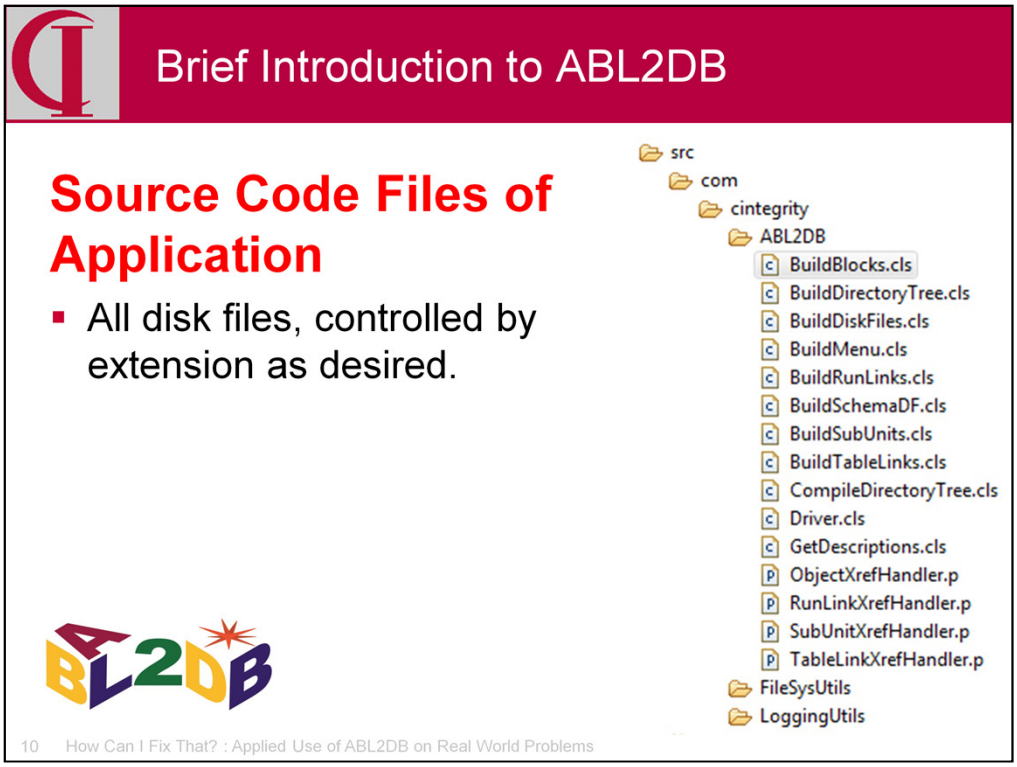
So, let's look briefly at the tool I have created to address this problem area.



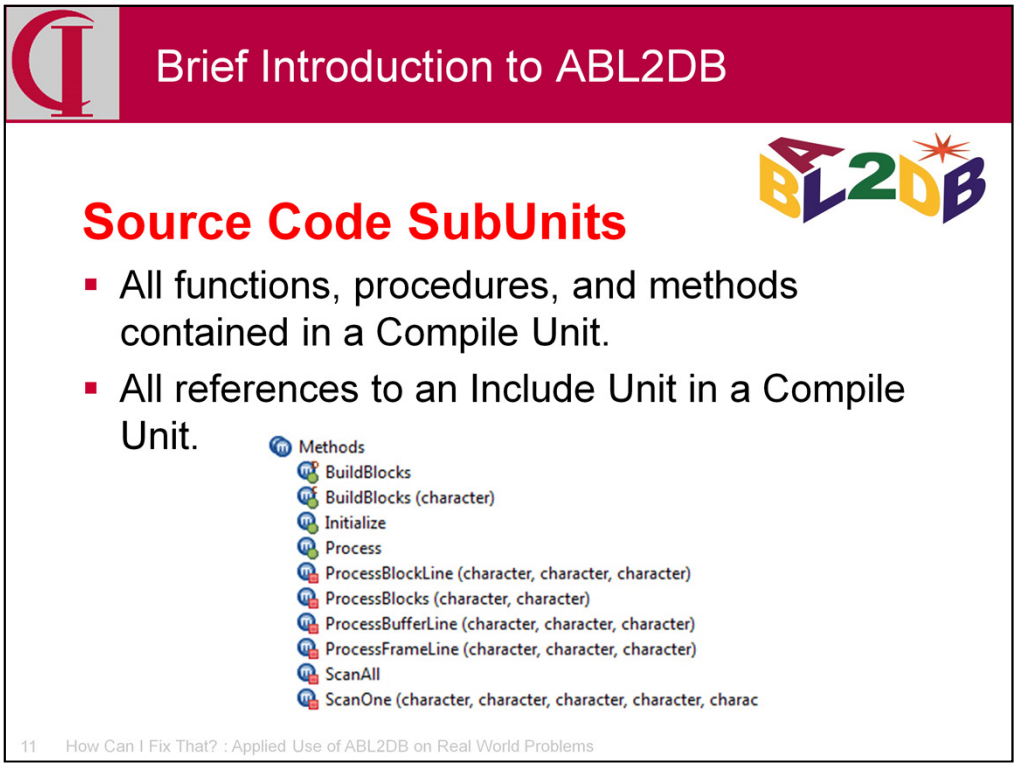
Which brings me to ABL2DB, my own offering for tackling this problem. This is a tool which analyzes existing code and schema and creates a database of information extracted from that code and schema in a form suitable for easy reporting and analysis. It is open source and has been designed to be highly modular, highly adaptable, and easily extended. There are some components which, by necessity, need to be tailored to the individual code base, but these have been isolated, identified, and packaged for easy customization. As you will see, the current version is already collecting a substantial amount of information and the roadmap is in place for extensions soon.



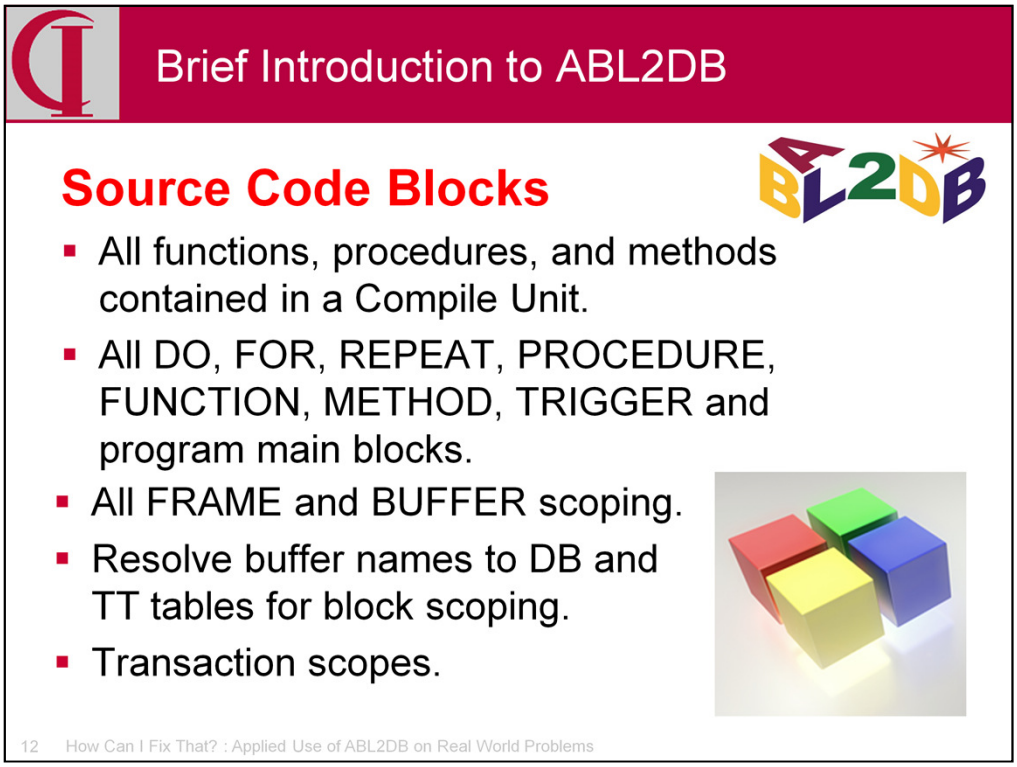
I created a set of database tables to mirror _File, _Field, etc. because those files will have the analysis schema in our target database. The fields provided are very complete and a program is provided to load these from a .df. The load program doesn't yet cover every possible option because I don't have a .df sample with everything to work from, but the structure is very easily expanded to any new dump syntax. I have already done a number of extensions as the tool is used at different sites and the changes have been quick and easy to make.



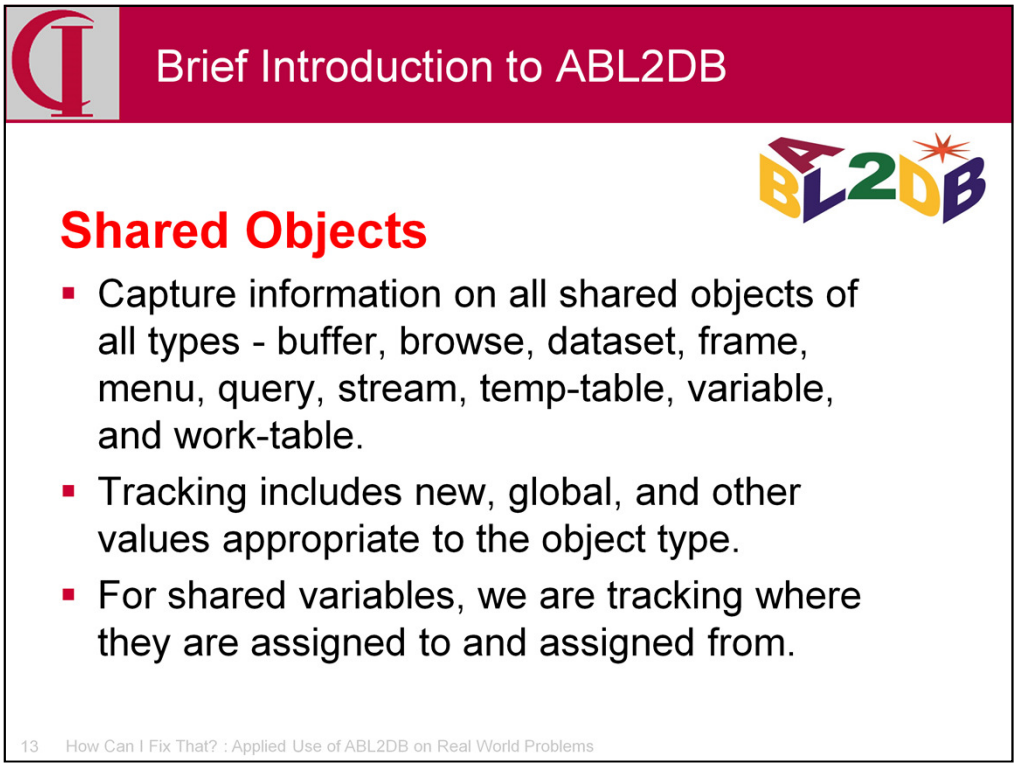
Second, we are capturing basic information about all source code files in the application. Filters are provided so that you can define what is and isn't source in the same directory tree in case you have non-source files mixed in with the source. It even handles old Varnet systems where there are compile units with no extension.



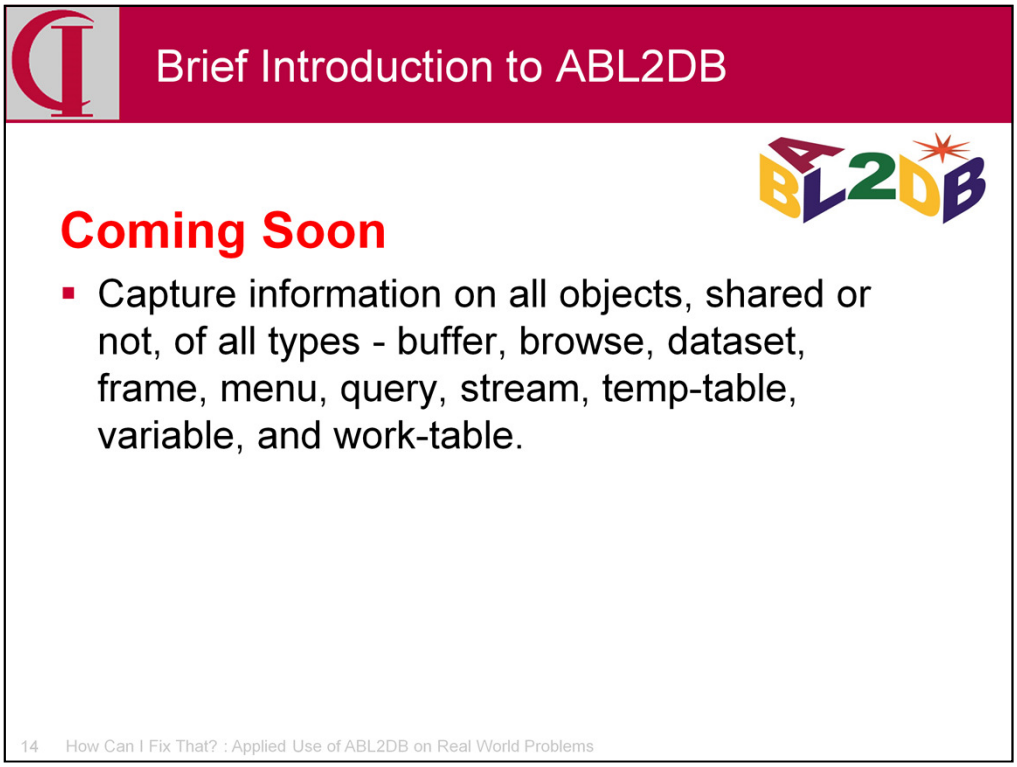
The next step breaks down Compile Units into sub components including procedures, methods and functions. It is this pass which also creates the links for empirical include files. For non-class Compile Units, a SubUnit corresponding to MainBlock is also created for code not otherwise enclosed in a SubUnit.



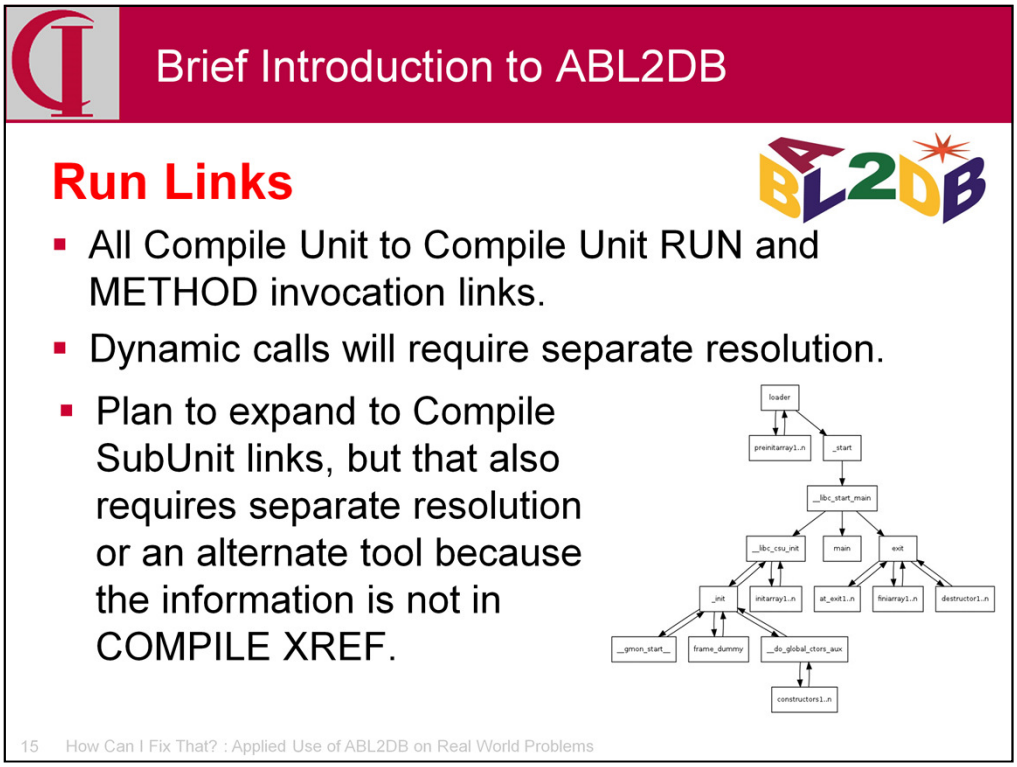
In addition to the functional decomposition of Compile SubUnits, a scan is also made to record all blocks of any type including DO, EDITING, FOR, FUNCTION, REPEAT, PROCEDURE, FUNCTION , TRIGGER and METHOD. In the process, it is recorded whether a transaction is scoped to the block and any frames and buffers scoped to the block. The full and accurate resolution of buffer names is a recent addition based on Proparse.



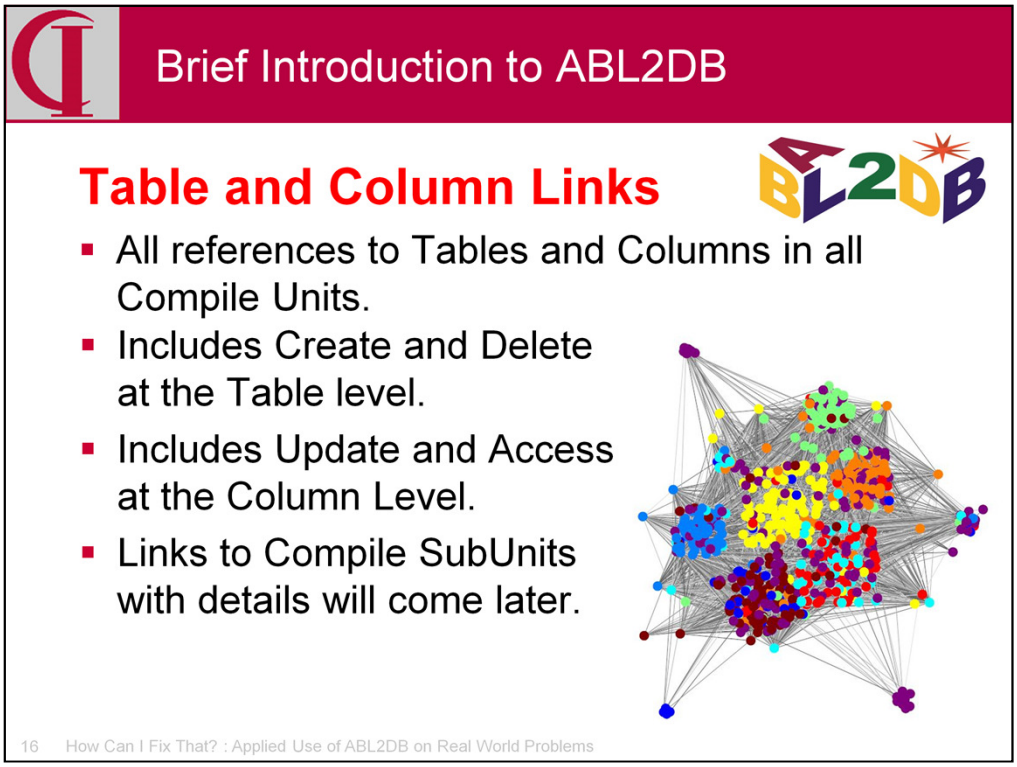
With the use of Proparse, I recently have added tracking of all forms of shared objects. This tracking includes where the definition is new, global, and a number of other parameters appropriate to each object type, including a field list for temp-tables and work-tables. This includes tracking whether shared variables are involved in an assignment on the left or right side of the equals.



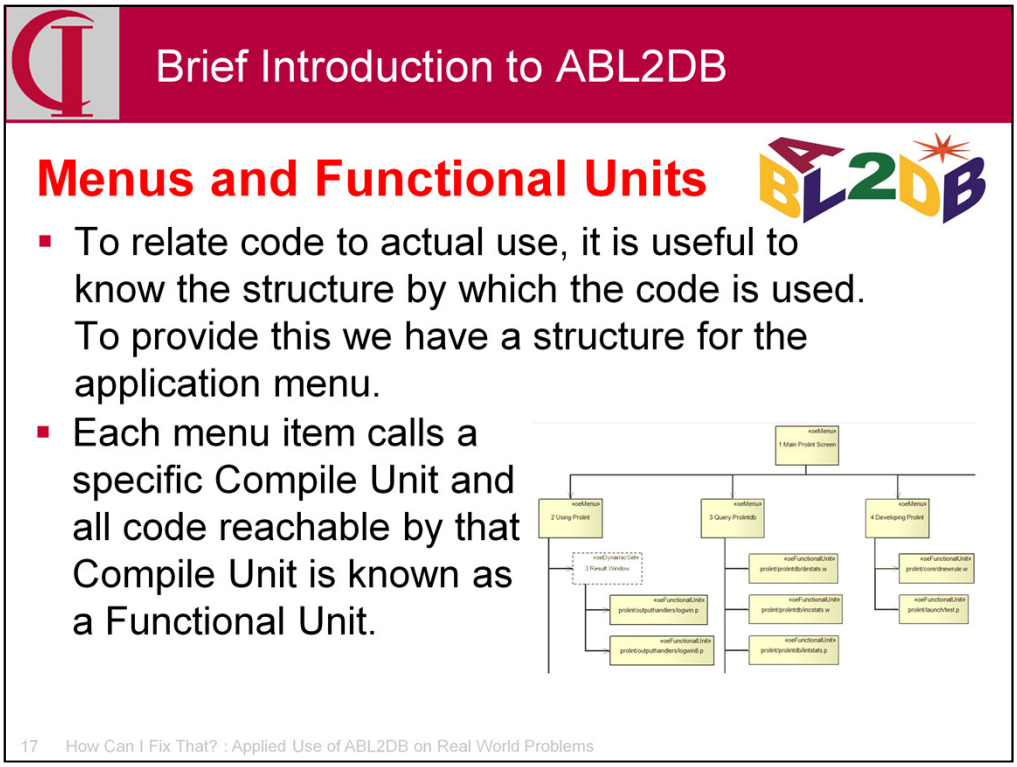
I am currently well along in extending this tracking to all of these objects, whether shared or not.



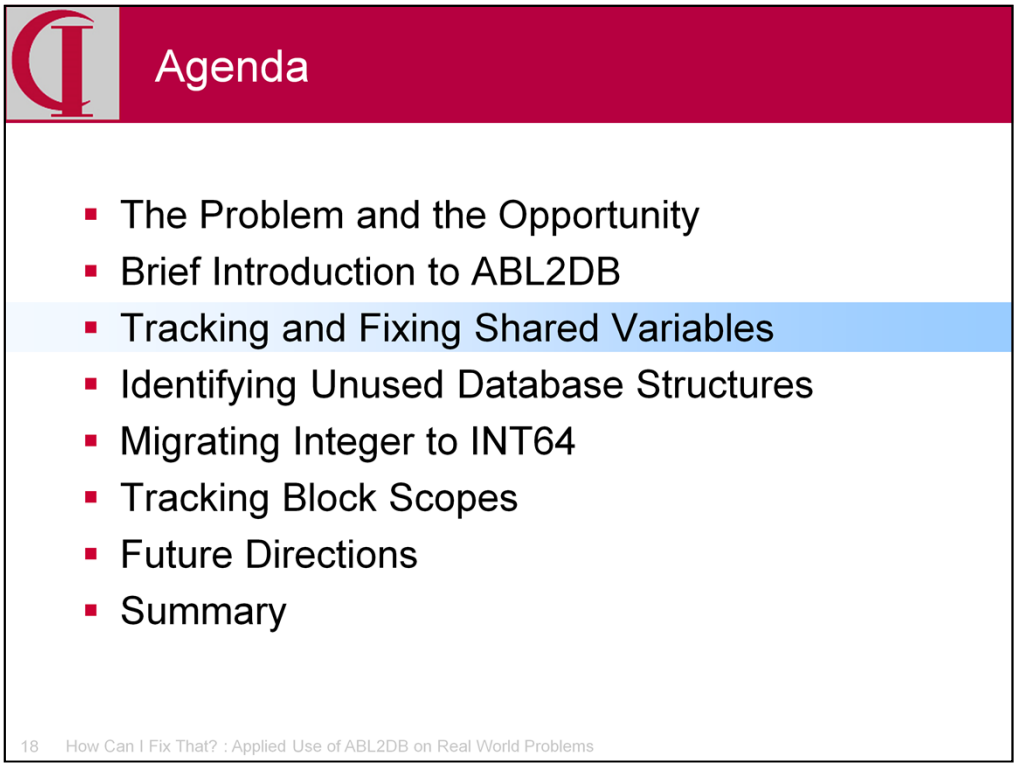
Now that we have identified all the static components, it is time to connect them. The first step is all the RUN relationships between pieces of the source code. The initial pass identifies all static Compile Unit to Compile Unit run connections. Not all can be resolved at this point because some calls are dynamic such as RUN VALUE() or RUN x IN handle. These dynamic calls will be resolved later. The data structure also provides for links between Compile SubUnits, i.e., a call located in an Internal Procedure, Method, or Function or a call to an Internal Procedure, Method, or Function from another Compile Unit, e.g., such as Method calls on classes or Internal Procedure calls to persistent procedures. Unfortunately, the information needed to make these connections is not in the COMPILE XREF data, so these connections will have to be identified by subsequent analysis or adding a new tool.



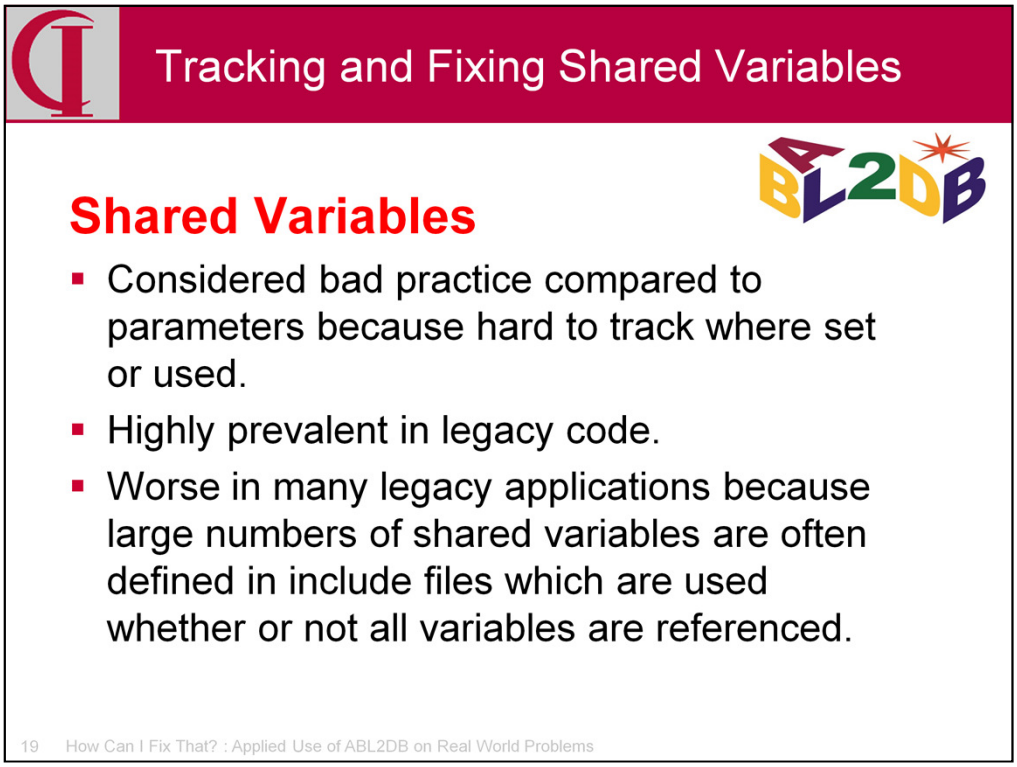
Next, we have links between the code and the data. This includes all references to tables and columns in all compile units. At the table level, it also includes flags for when the record is created or deleted. At the column level it has a flag for update, i.e., the value of the column is changed in that unit. Any column not changed is merely accessed. Only columns actually referenced are included. Links to Compile SubUnits with details will come later.



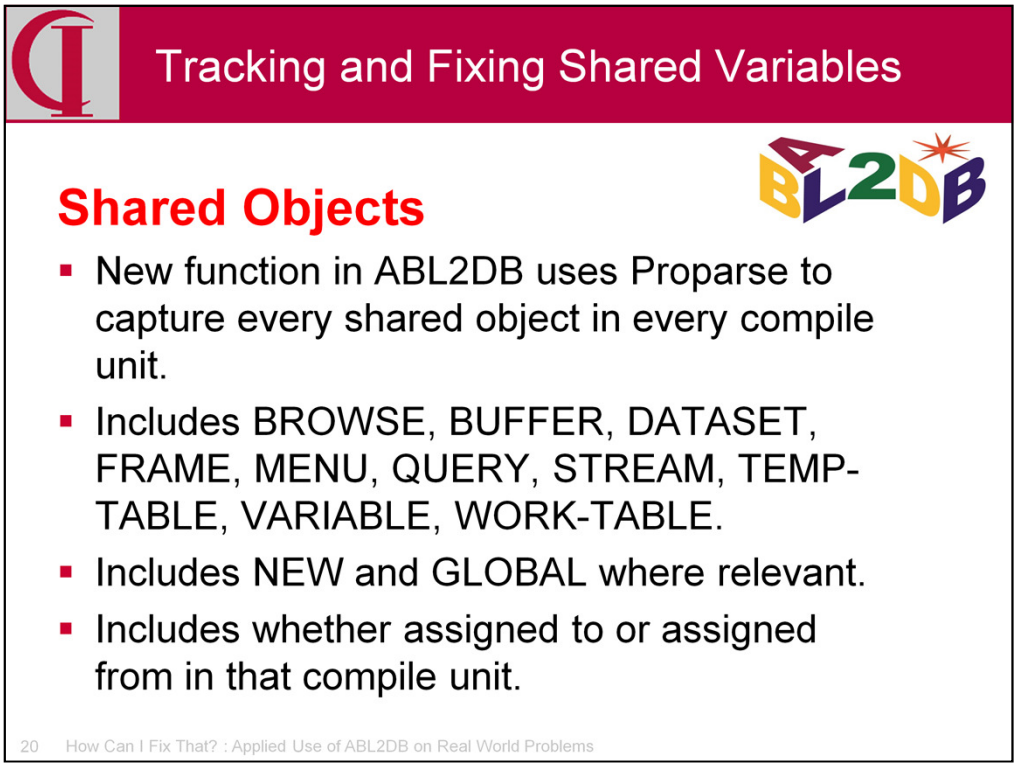
Finally, for the current version of ABL2DB, there are two additional stages. One loads descriptions for each disk file. Many code bases have some kind of standard description embedded in the source file and a tool is provided for extracting these according to the standards of the specific code base. The other is the loading of Menus and Functional Units. Menus provide us with a structure for the application as seen by the user, each selection of which runs a particular Compile Unit. From that Compile Unit, there is a body of code which can be reached including all sub programs, persistent procedures, and include files. We call this a Functional Unit. Functional Units may, of course, overlap.



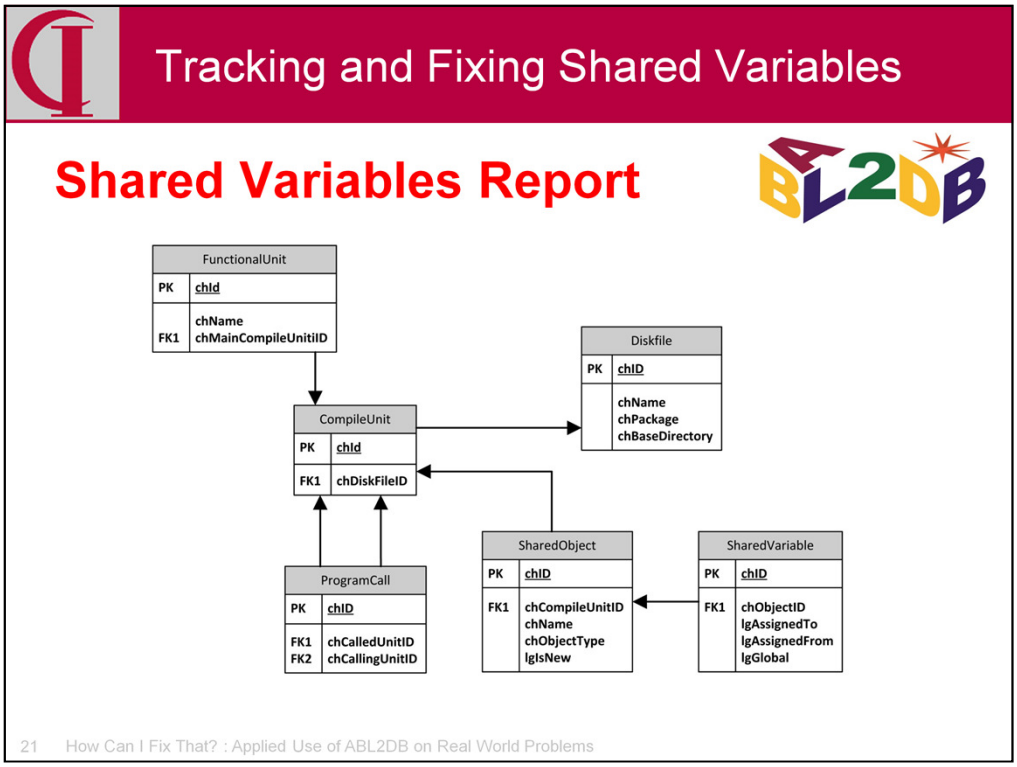
Now, on to some specific applications of the tool to real world problems. First, shared variables.



Anyone who has worked on legacy code is likely to have their own shared variable horror stories. In many applications, it was very common to define large blocks of shared variables in an include file and then to reference that include file routinely in all parts of an application or a specific function. This makes it very difficult to determine where a shared variable is set and used and clogs many files with large numbers of definitions which are not even referenced in that compile set. There are techniques for replacing shared variables with explicit parameter passing or with common reference objects, but first one has to know the landscape where the replacement will occur.

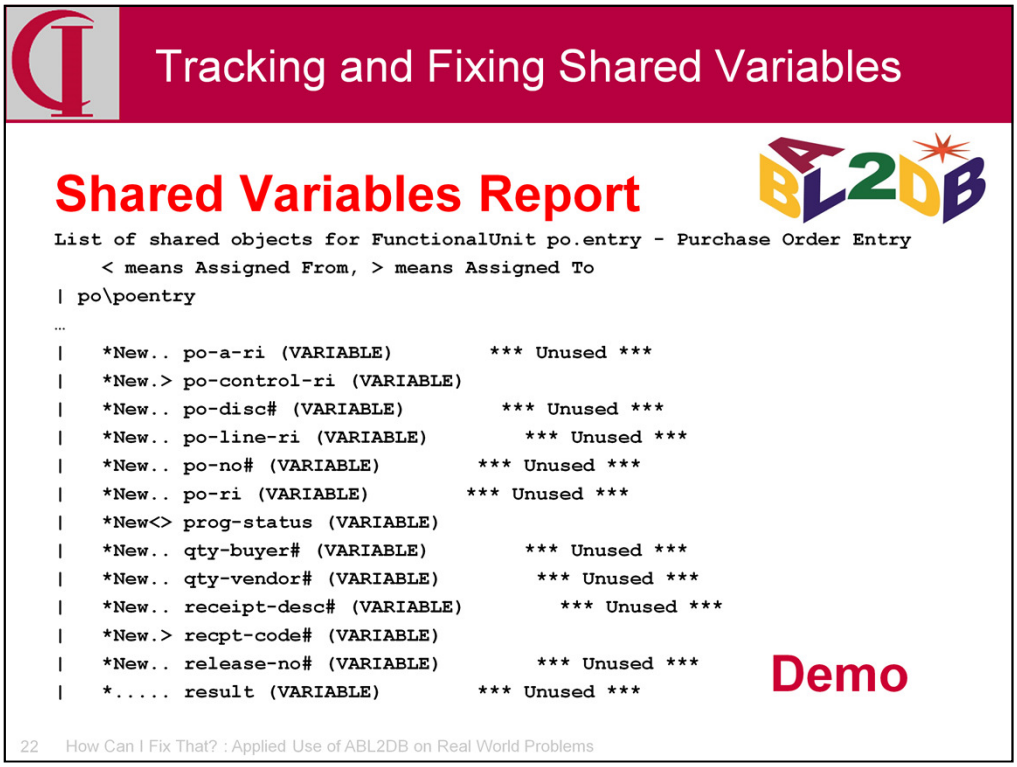


The new function captures every shared object definition in every compile unit, along with many characteristics of that definition. Includes whether or not the variable is assigned to or from in the compile unit.



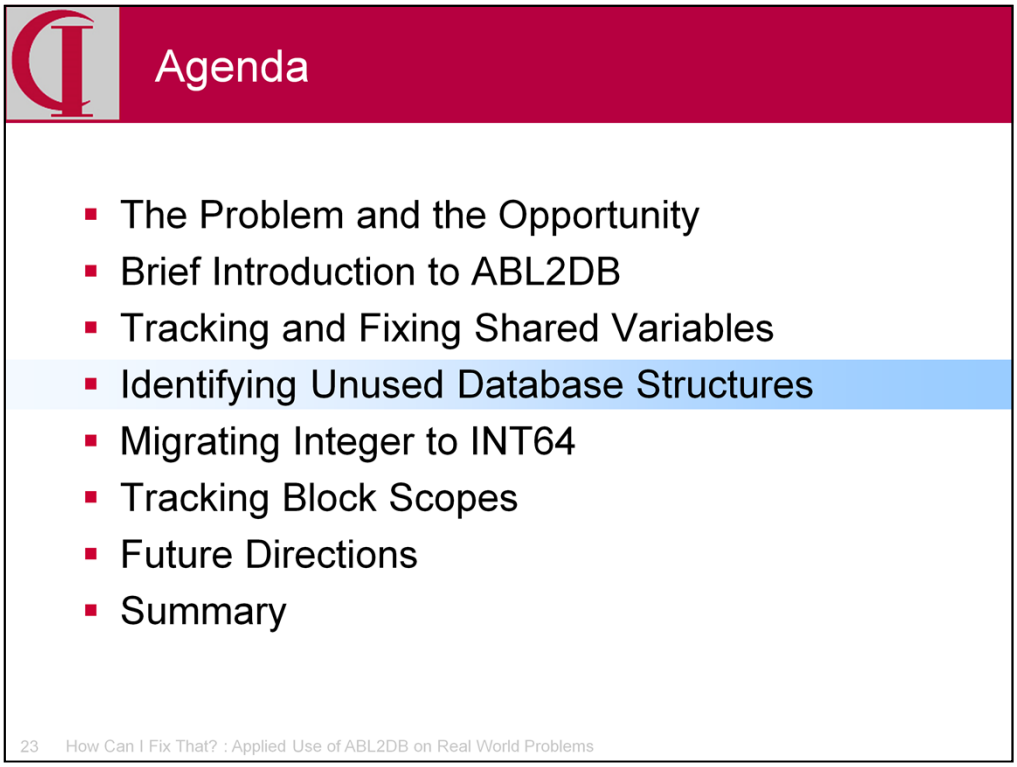
Here is the schema involved in this report. We are going to start with the Functional Unit, a term in ABL2DB for an interconnected body of code which performs some function. For example, all of the code which one might reach starting with a particular selection on a particular menu or a particular batch process. From there we will go to the main CompileUnit. One can just start with the main compile unit. For example, at Inenco, there were no database tables with menus as there are in many applications and all menus are simply embedded in programs. This makes it difficult to extract functional unit and main compile unit pairs. Finding the main compile unit for any function thus requires reading code. James is working on changing this.

ProgramCall gives us the chain of all referenced CompileUnits from that base. We link to DiskFile to get the name of the CompileUnit on Disk. Then, we link to the SharedObjects in that CompileUnit. Here, we are looking only at shared variables for now, so we link to the SharedVariable table for additional information.

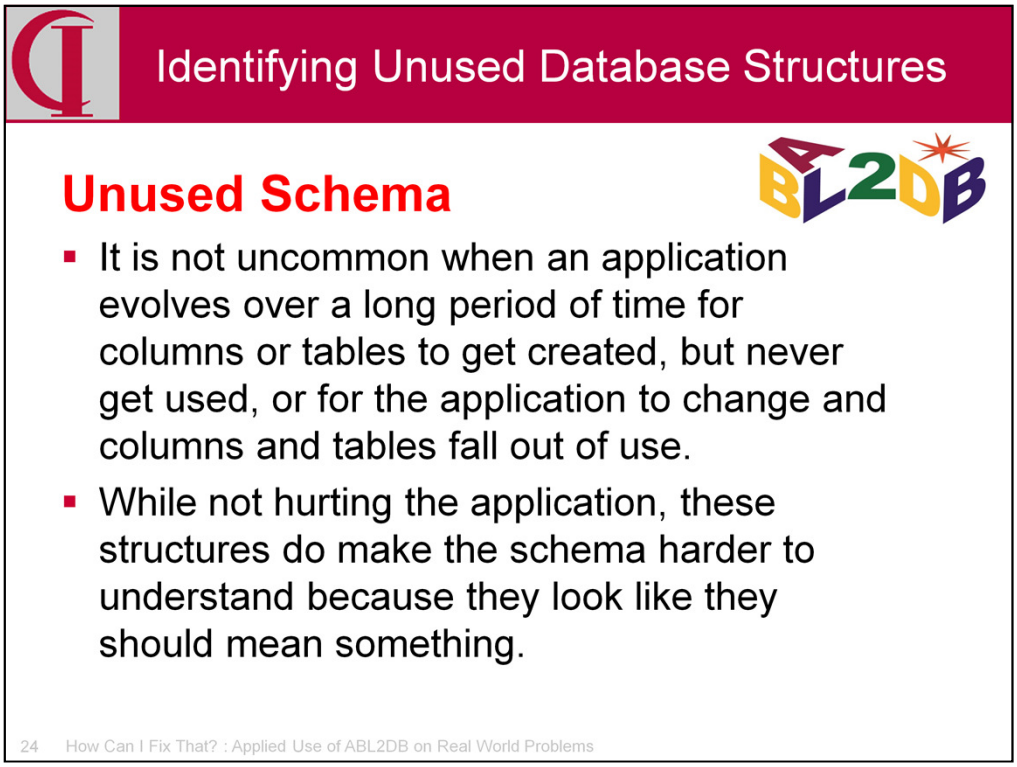


So, let's take a look at the results and the code.

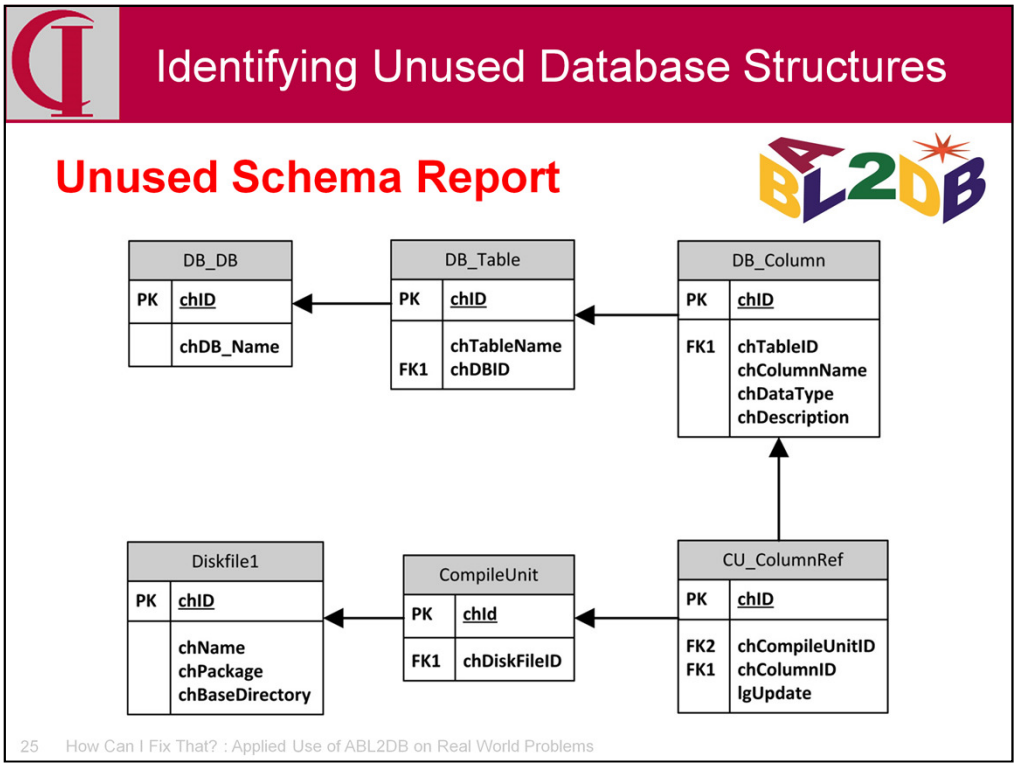
Switch to desktop to look at sample output and code. These will be published with the talk on cintegrity.com.



Now let's look at identifying unused database structures.

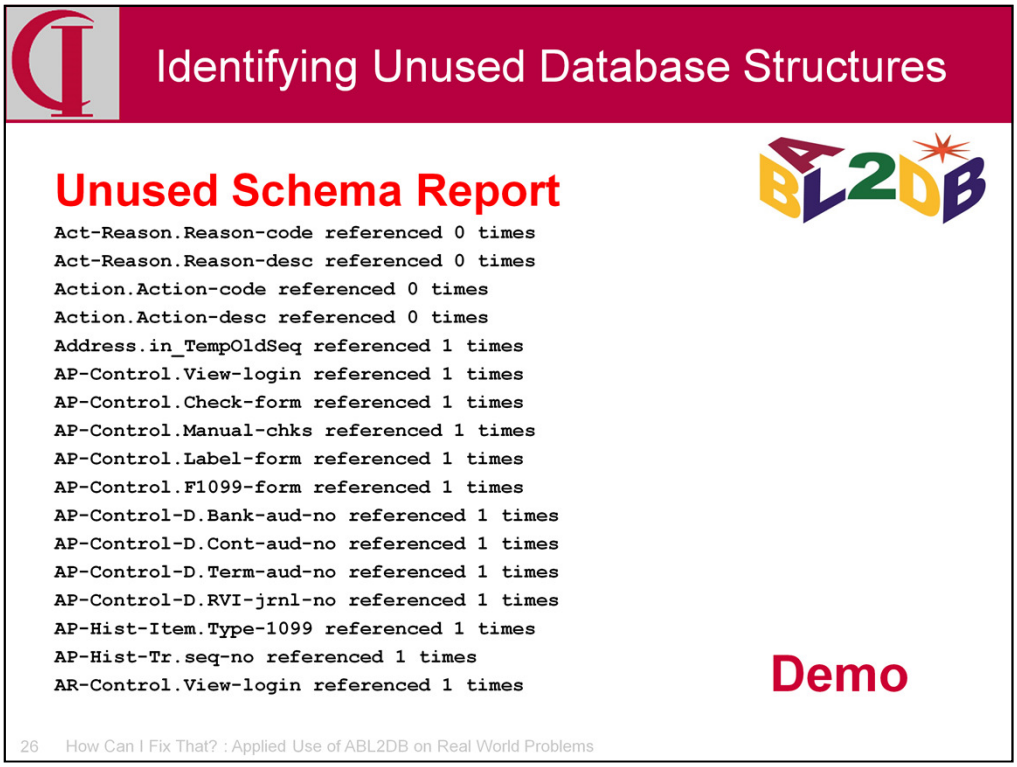


With an application that evolves over 20 years, like both of the sample code bases used here, it is common for fields and even tables to get added in anticipation of some work that is never done or for usage to change to other fields, leaving behind the old fields, or even for whole modules to fall out of use. None of this hurts the running of the application, but can make the schema harder to maintain since the field looks like it should contain meaningful data.



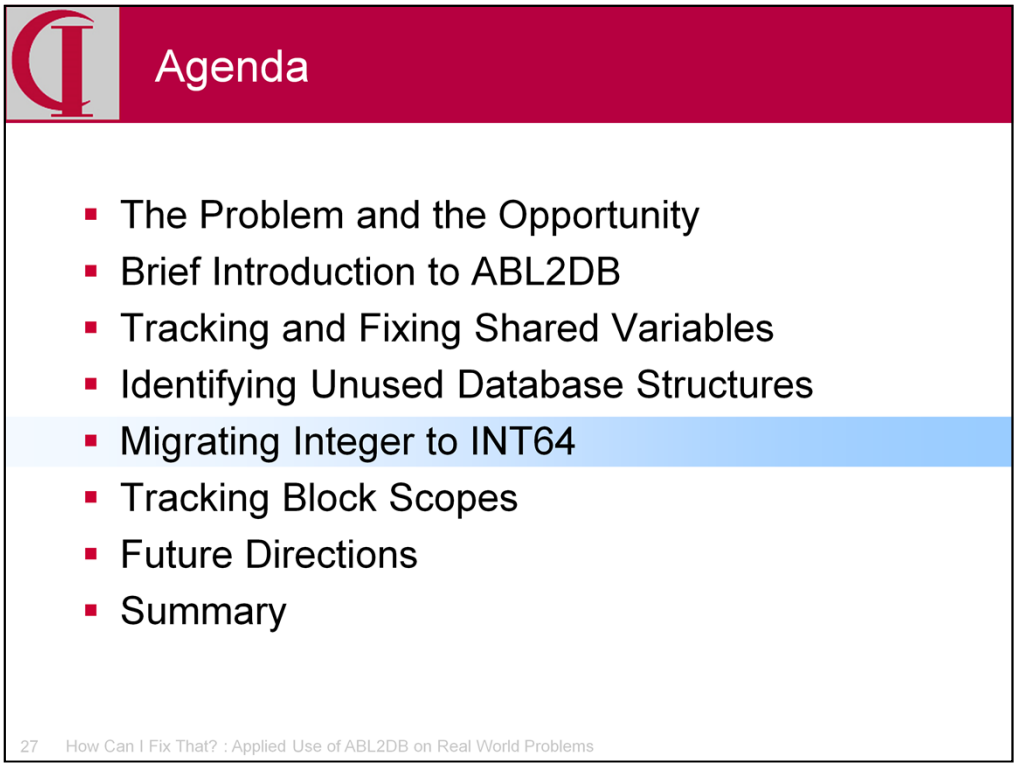
Here is the schema involved in this reporting. For all of the databases, all of the tables of each of those databases, and each of the columns in those tables, we look for column references, i.e., links between the column and a compile unit. In this case, we don't actually need to follow the link to the compile unit and diskfile, since we don't care about the names at the moment. We count these column references until we get more of them than some specified limit. If we reach the limit, we stop counting and move on to the next field. For the sample, I used a limit of 1 on the theory that a column only referenced once is probably not really being used. This will report fields with 0 or 1 references. So, any time there are 2 links, it moves on to the next column.

Note that we might want to follow the link to the compile unit name to find out where a reference occurs in order to evaluate the usage.

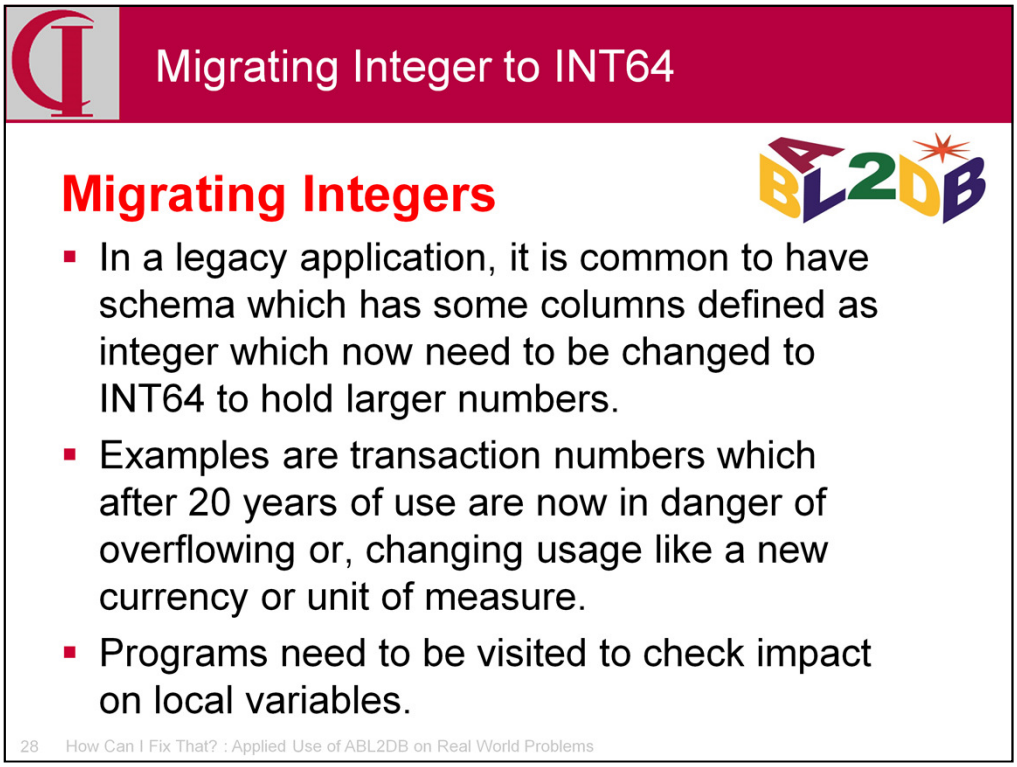


So, let's take a look at the results and the code.

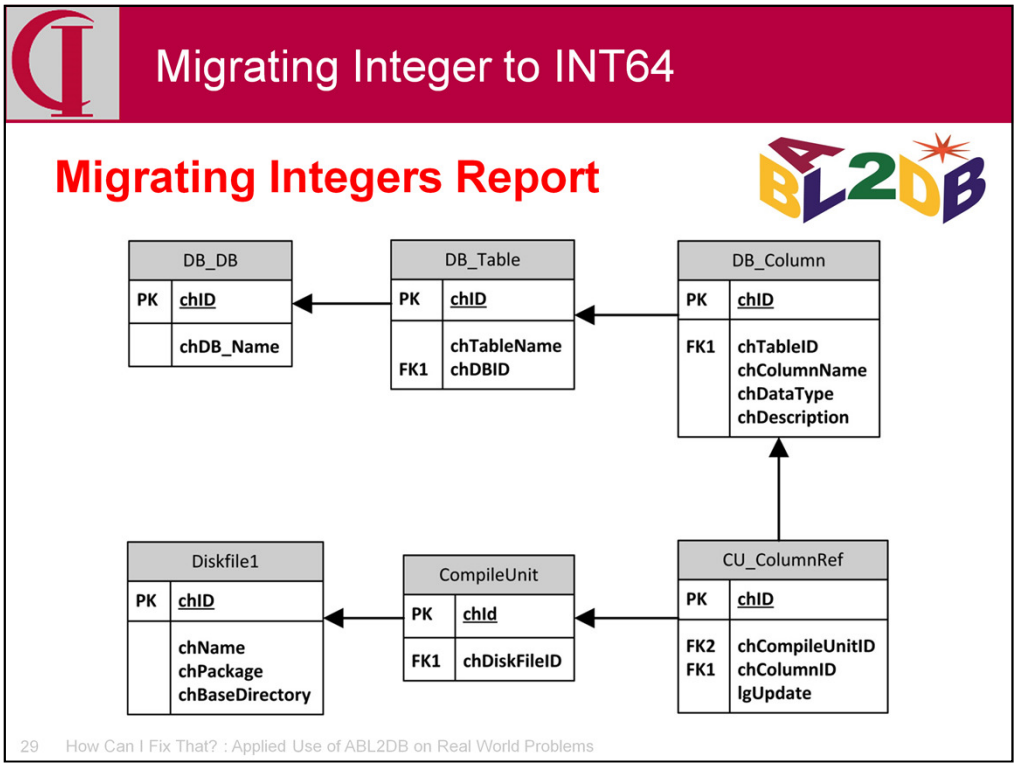
Switch to desktop to look at sample output and code. These will be published with the talk on cintegrity.com.



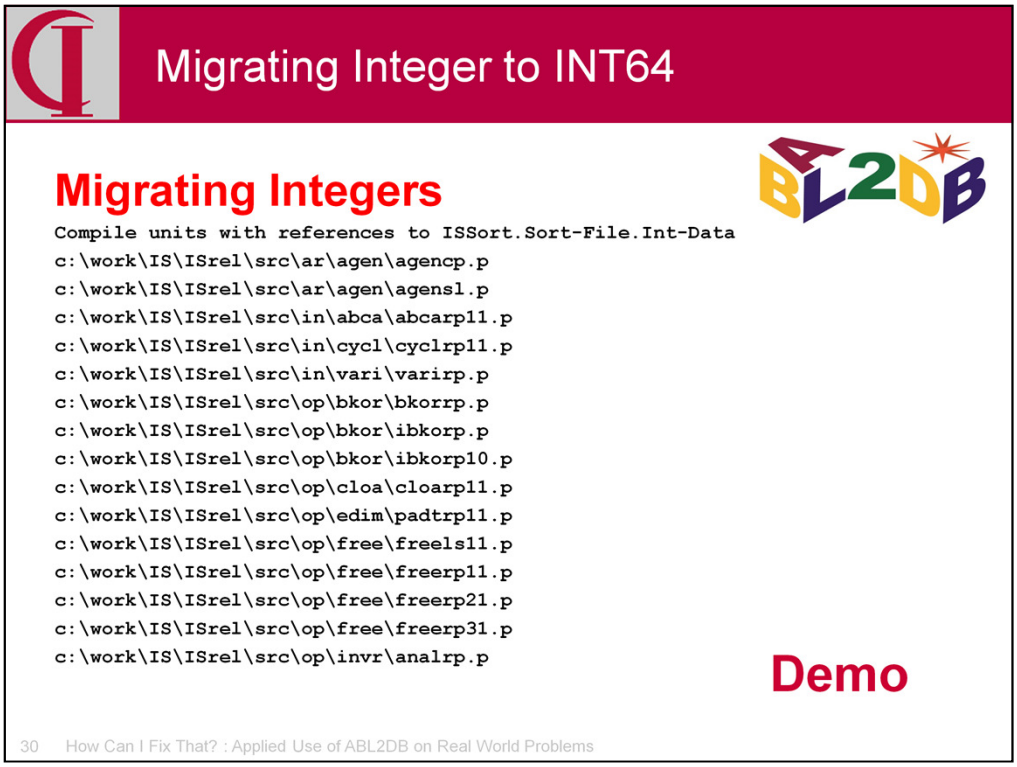
Now let's look at the problem of migrating a particular database field from integer to INT64.



Once upon a time we only had integer as a datatype, but since 10.1B we have also had INT64 to handle much larger numbers. As applications evolve, what was once satisfactory as a plain 32 bit integer may need conversion to 64 bit. For example, a counter used to sequence transactions might be in danger of overflowing or a new currency or unit of measure may lead to the need for a larger integer. While one can make this change in the dictionary easily enough, it is advisable to visit all the places the column is referenced to see if there are local variables which also need to be upsized.

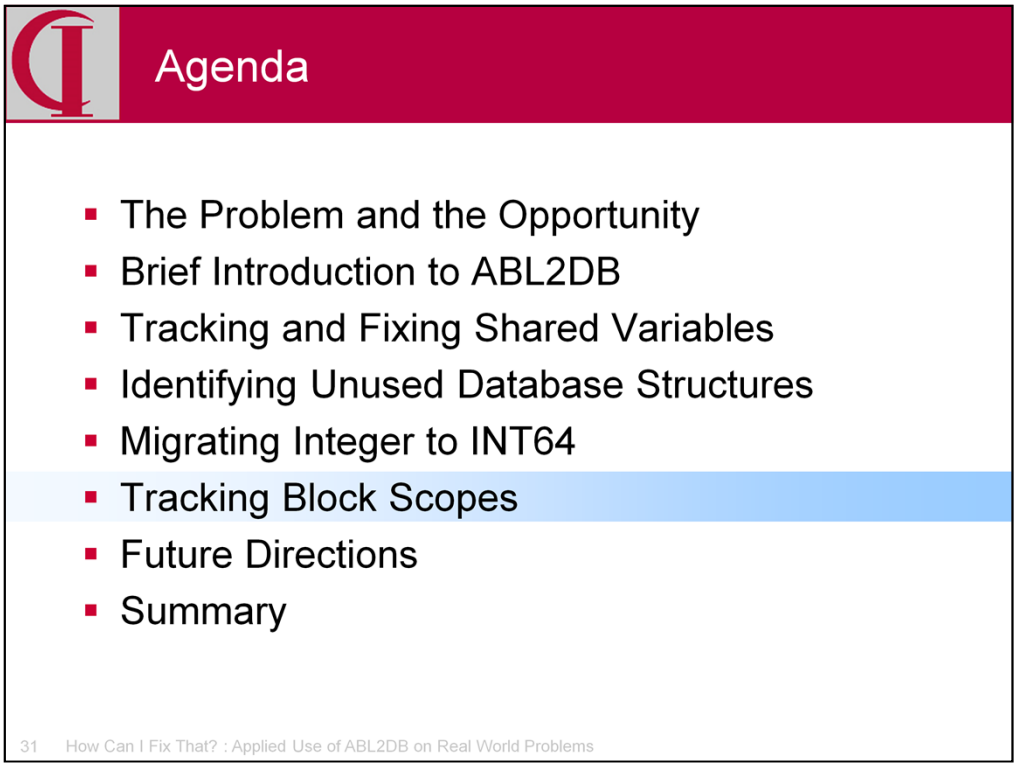


So, here is the relevant schema, which you will notice is the same tables as in the last example. This time, however, we will look for a specific database, specific table, and specific column. Then we will follow the column references for that column to the compile unit and on to the diskfile to get the disk filename of the compile unit.

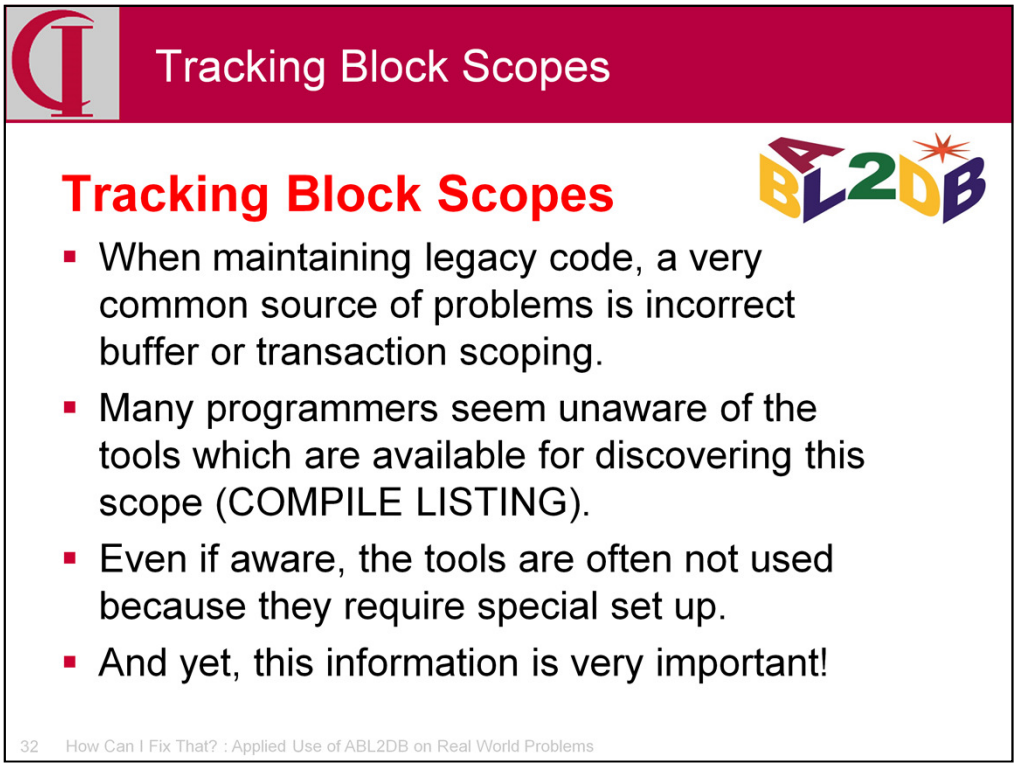


So, let's take a look at the results and the code.

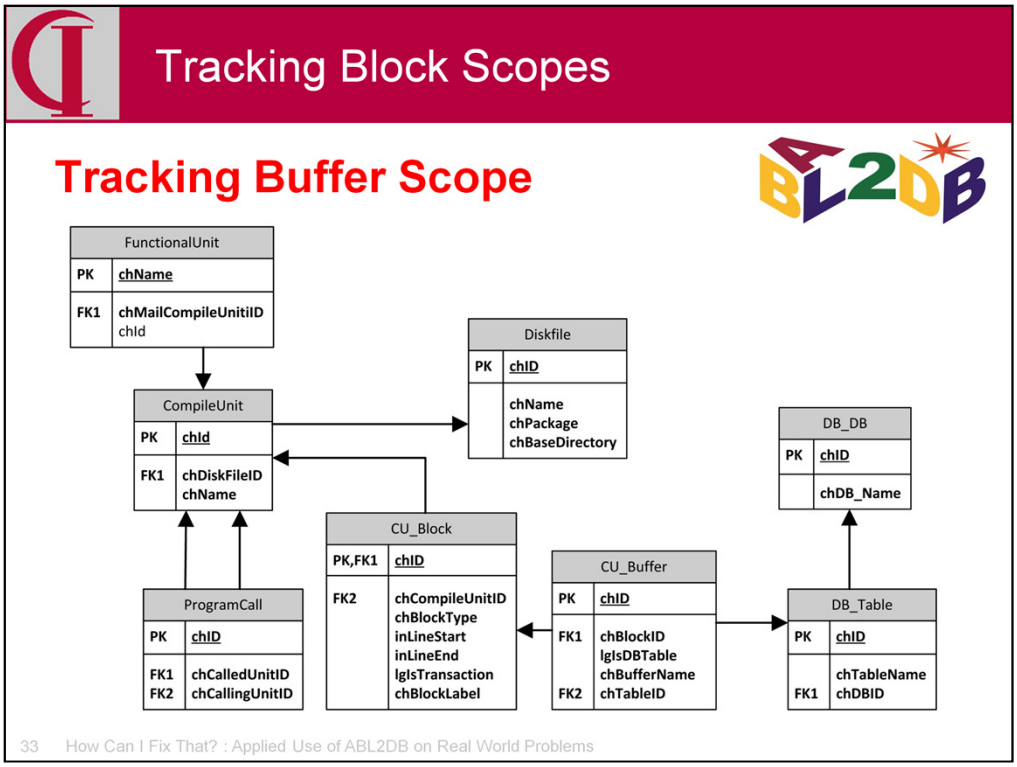
Switch to desktop to look at sample output and code. These will be published with the talk on cintegrity.com.



Finally, let's look at tracking block scopes.



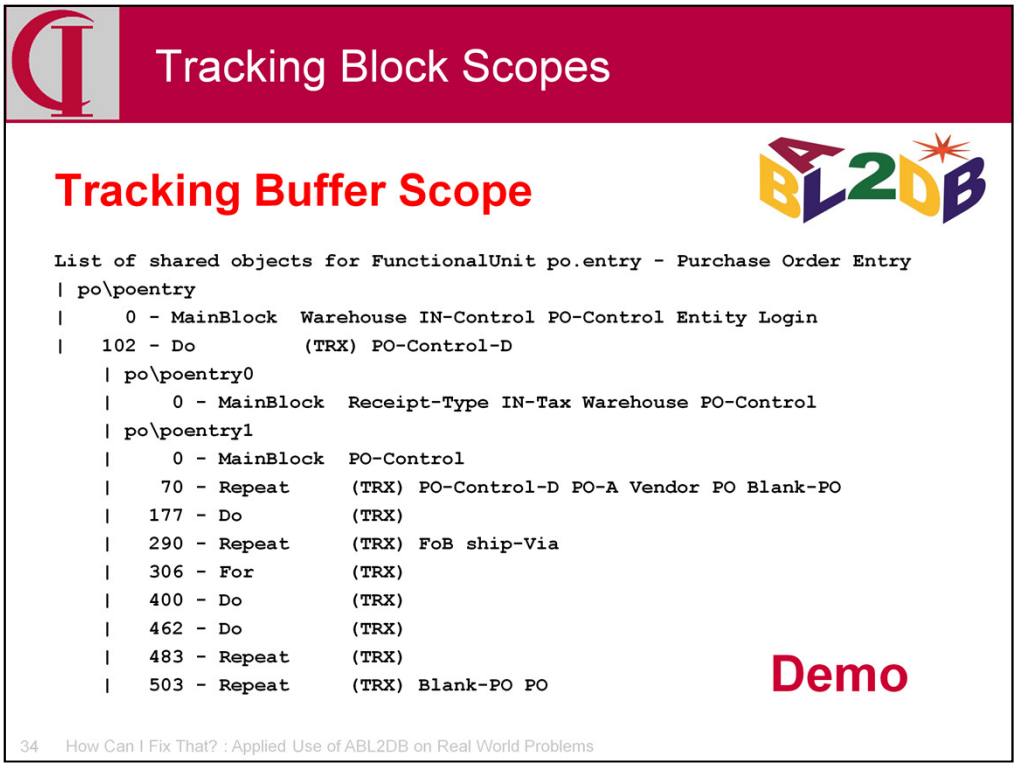
Judging by what one sees on internet forums, one of the most common problems in development is incorrect buffer or transaction scoping. We have COMPILE LISTING to show us these scopes, but many programmers are either unaware of the tool or seem to forget to use it to check their work. But, this information is critical to understanding many problems.



Here we have the schema involved in this report. We start with the Functional Unit leading to the main Compile Unit which is called to start this function as discussed before.

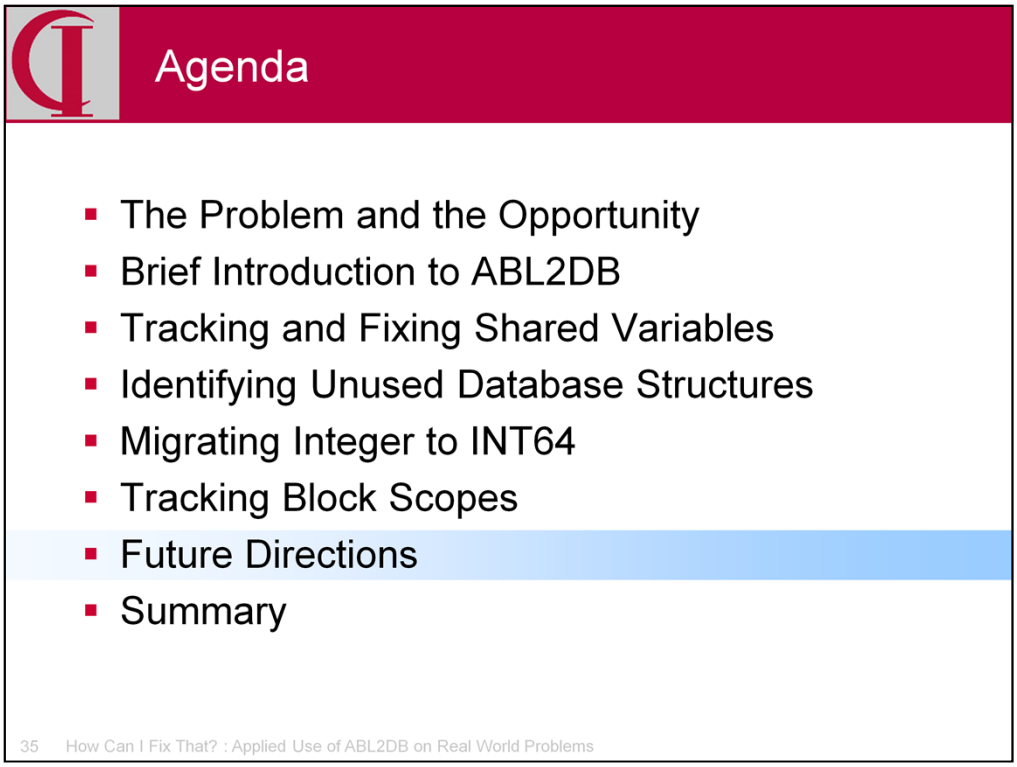
From the main Compile Unit, we walk down the Program call link which connects Compile Units with the Compile Units they call. For each Compile Unit, we link to Diskfile to get the filename.

For each Compile Unit, we then look at each Block in that Compile Unit. This tells us whether there is a transaction scoped to that block. We then look to see if there are any buffers scoped to that block and what table is associated with that buffer if the buffer is for a database table.

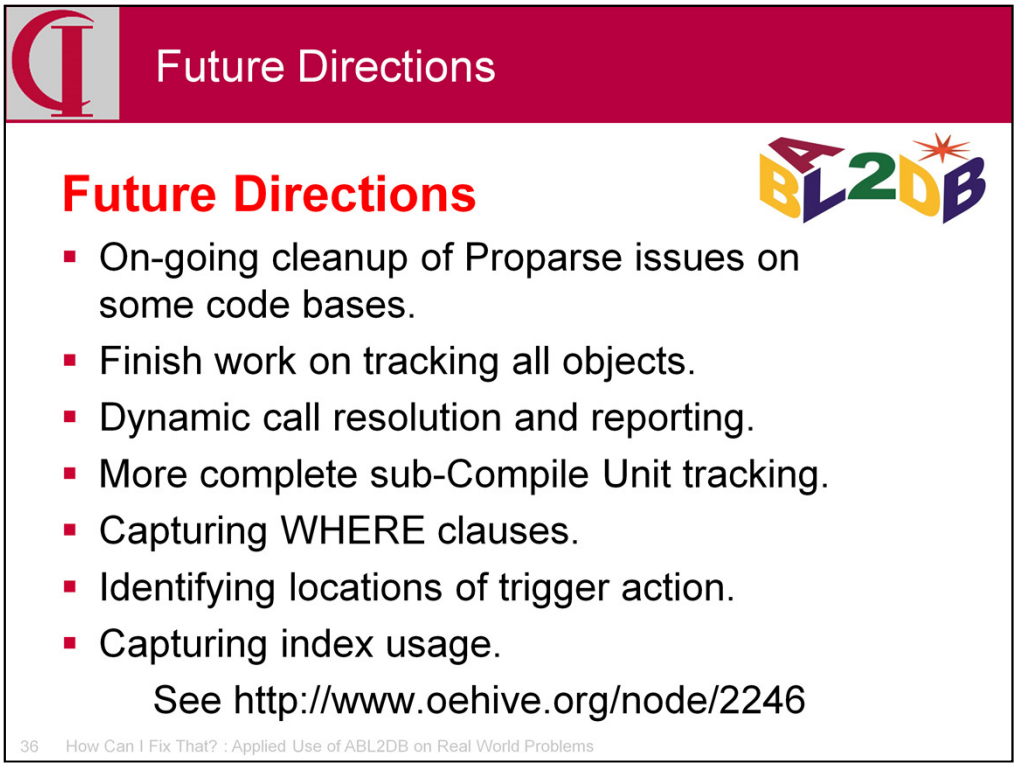


So, let's take a look at the results and the code.

Switch to desktop to look at sample output and code. These will be published with the talk on cintegrity.com.



That concludes our examples for the day. Let's look forward a bit to what is coming next.

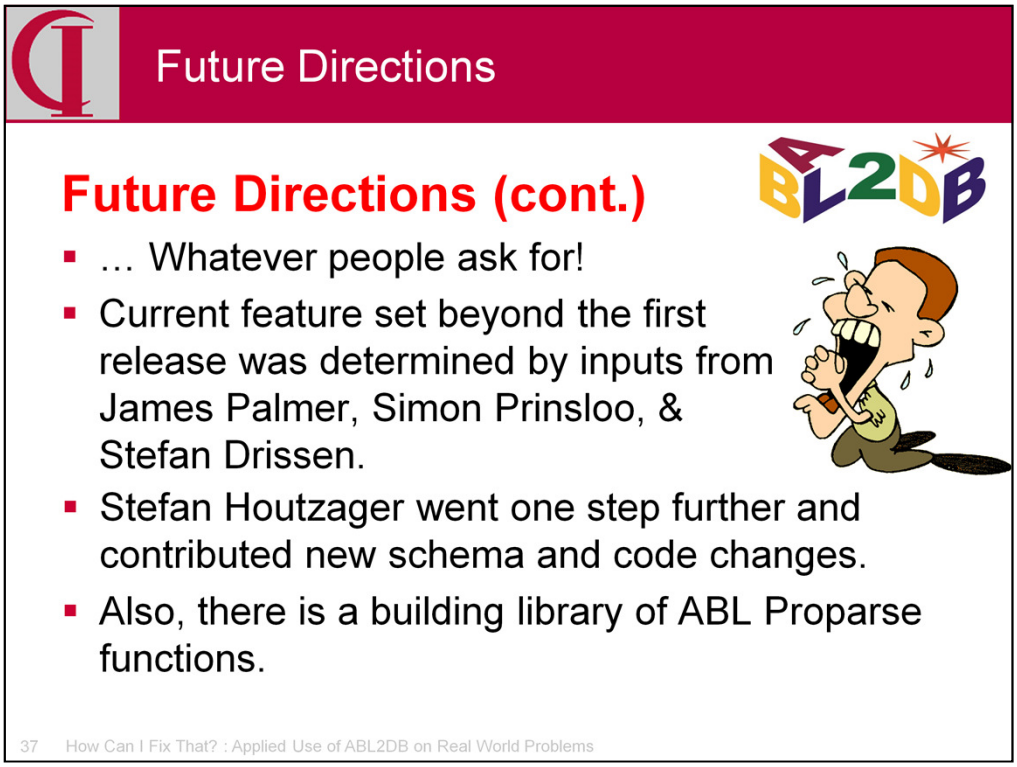


John has made extensions to Proparse to cover all of the Proparse coverage issues we have encountered and can document. No doubt we will find more as we extend the work to new code bases.

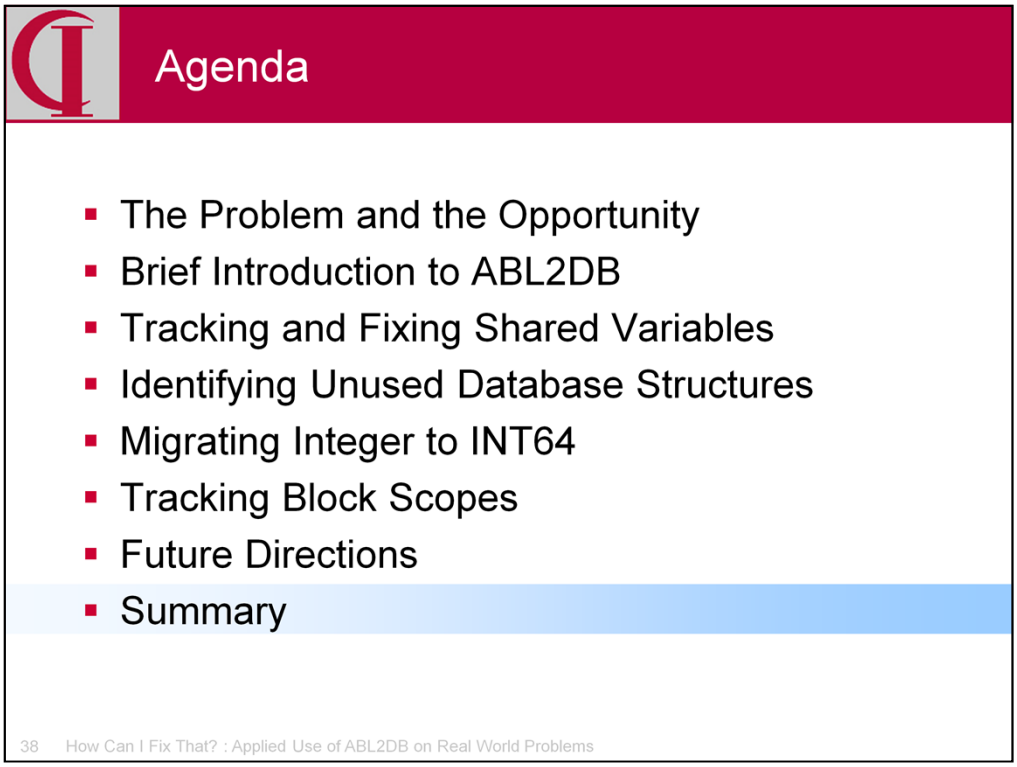
I should soon finish the work on tracking all objects, not just shared ones.

Among the other projects on the list is to improve the reporting on dynamic calls and provide a mechanism for automatic resolution of these calls in subsequent builds. We would also like to extend the tracking of calls and data links to sub-Compile Unit blocks, i.e. internal procedures and methods. We plan on capturing where clauses for all data accesses and possibly index usage. We are going to look at reporting on trigger action for when trigger maintenance is contemplated. This may simply be a report on data that is already there.

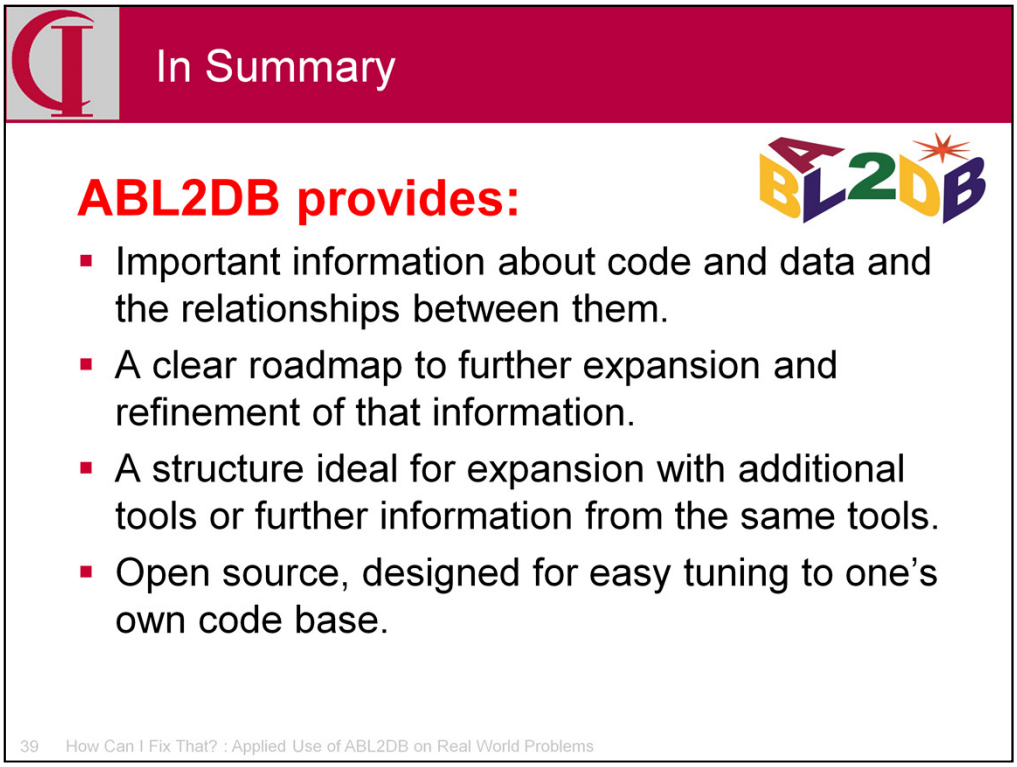
See the indicated URL for roadmap and wish list items.



But, most importantly, what gets added next will depend more than anything else on what people need and ask for. No small part of what has been added from the initial release is there because James needed it. Simon Prinsloo and Stefan Drissen have also had direct impact. Stefan Houtsager went one step farther and contributed an addition including schema and code which has been included. I would love it if more people did this. Also, there is a growing library of ABL Proparse utilities which can assist someone wanting to build their own Proparse-based tools.



To summarize…

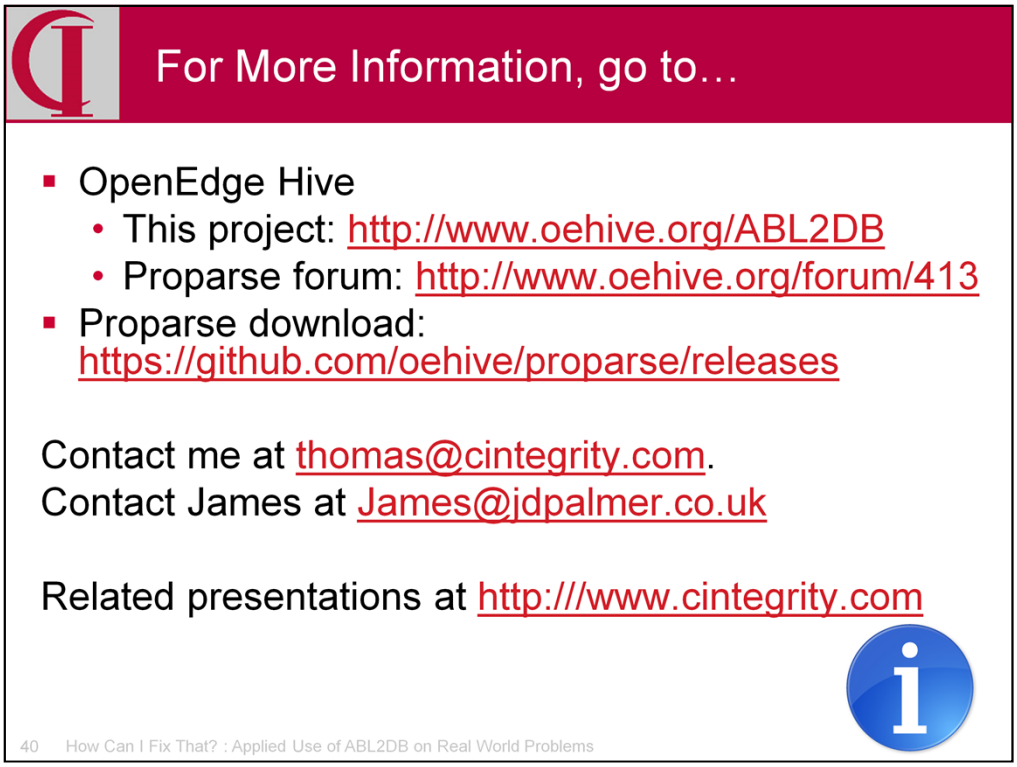


This version of ABL2DB covers a lot of information about the code and data in an application and their relationships. It is readily available and easy to use in building queries and reports.

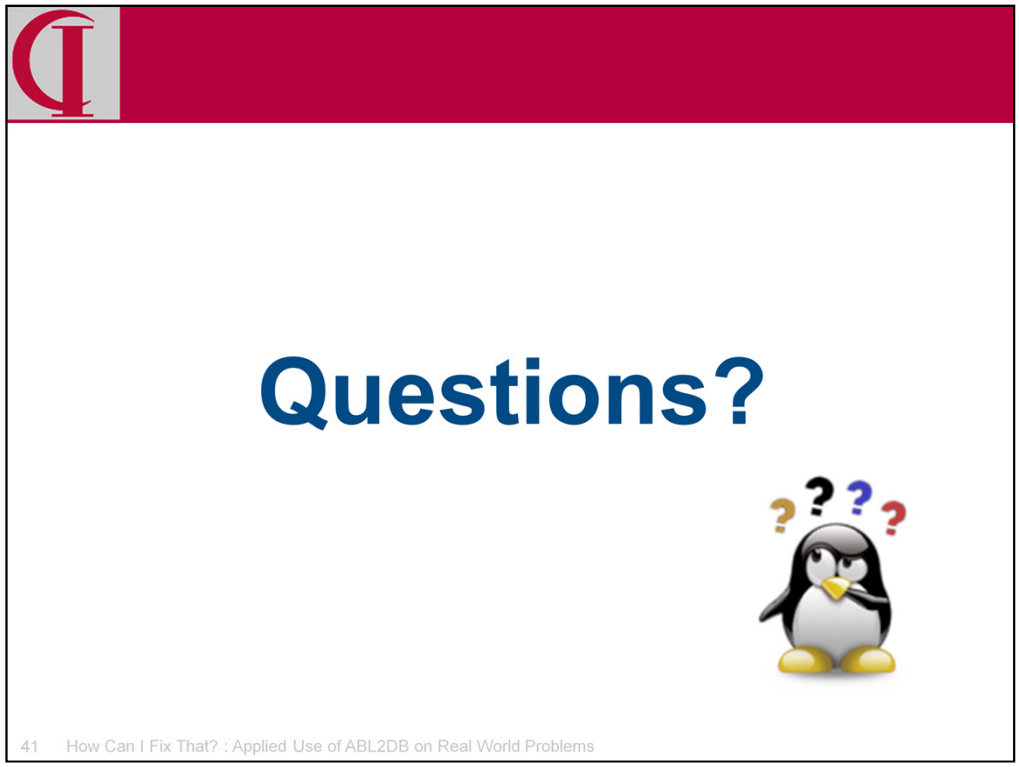
We have a clear roadmap for adding more information and functionality and refining the information we have.

The design of the system is very modular and easy to expand with new functions or to expand which information is collected.

The tool is open sourced and has been designed to make needed adjustments easily to fit it to the characteristics of the local code base.



Here are some links for more information.


Questions?
41 How Can I Fix That? : Applied Use of ABL2DB on Real World Problems

Thank You

Dr. Thomas Mercer-Hursh
thomas@cintegrity.com