MEET DB2: Automated Database Migration Evaluation

Reynold S. Xin*
EECS, UC Berkeley
rxin@berkeley.edu

Patrick Dantressangle
IBM UK
dantress@uk.ibm.com

Sam Lightstone
IBM Canada
light@ca.ibm.com

William McLaren
IBM USA
wmclaren@us.ibm.com

Steve Schormann
IBM Canada
schorman@ca.ibm.com

Maria Schwenger
IBM USA
schwenge@us.ibm.com

ABSTRACT

Commercial databases compete for market share, which is composed of not only net-new sales to those purchasing a database for the first time, but also competitive "win-backs" and migrations. Database migration, or the act of moving both application code and its underlying database platform from one database to another, presents a serious administrative and application development challenge fraught with large manual costs. Migration is typically a high cost effort due to incompatibilities between database platforms. Incompatibilities are caused most often by product specific extensions to language support, procedural logic, DDL, and administrative interfaces. The migration evaluation is the first step in any competitive database migration process. Historically this has been a manual process, with the high costs and subjective results. This has led us to reexamine traditional practices and explore an automatic, innovative solution.

We have designed and implemented the Migration Evaluation and Enablement Tool for DB2 for Linux Unix and Windows, or MEET DB2, a tool for automatically evaluating database migration projects. Encapsulated in a simple one-click interface, MEET DB2 is able to provide detailed evaluation of migration complexity based on its deep analysis on the source database. In this paper, we present MEET DB2, and discuss many aspects of our design, and report measurements from real-world use cases. In particular, we show a novel way to use XML and XQuery in this domain for better extensibility and interoperability. We have evaluated MEET DB2 on 18 source code samples, covering nearly 1 million lines of code. The utility has provided benefits in several dimensions including: dramatically reduced time for evaluation, consistency, improved accuracy over human analysis, improved reporting, reduced skill requirements for migration analysis, and clear analytics for product planning.

*Work done while at IBM Canada.

Proceedings of the VLDB Endowment, Vol. 3, No. 2

1. INTRODUCTION

The relational database market is a worldwide business of over 20 billion USD. No single company dominates this space, providing a fertile competitive arena for several key companies, including IBM, Oracle, Microsoft, Teradata, and others. These companies compete for market share by attracting new business, and by encouraging companies to switch from a competitive platform to one of their own. The business term used for the conversion from a competitive platform is often called a "win-back". The process of moving an application from one database to another is called a database migration. This paper describes a prototype and its automated migration evaluation, called the Migration Evaluation and Enablement Tool for DB2, or MEET DB2.

Despite the existence of international standards for database definitions and language processing, such migrations rarely occur without development cost due to the large number of non-standard language and interfaces extensions that many database vendors provide. Therefore, when an application coded to operate against a specific databases needs to migrate to a new database platform, there are commonly a number of these extensions used by the source database that are unsupported in the target database, requiring some number of database or application changes to be made before the migration can be completed.

The costs of migration vary, and can range from 100% compatibility between disparate platforms to a significant engineering effort. Consequently, it never begins in practice without first having an evaluation and review of these costs. These evaluations have historically been performed in a manual and laborious fashion by engineers highly skilled in the database platforms of both the source and the target databases. Not only have such evaluations been time consuming, subjective and prone to variabilities of all manual processes, but the process is further compromised by the challenge of finding people deeply skilled in multiple combinations of databases. The introduction of automation into this process dramatically simplifies the analysis in several ways, as follows:

1. It significantly shortens the evaluation time.
2. It reduces the skills needed by the evaluator.
3. In many cases it will improve accuracy.
4. It adds consistency to the evaluation results.

5. It can provide a level of detail in the assessment that would be difficult for human evaluators to collect in most cases.

6. It can provide critical information (i.e. what is not supported but used extensively by customers) in future database system development's requirement planning.

MEET DB2 provides two types of interfaces: GUI and command line interface. By default, MEET DB2 presents itself in a simple GUI interface that contains a source file selection interface and a button to start the evaluation. The utility takes a file as input that contains source code of the source database's DDL and procedural logic, and produces a report in HTML file as output. For expert users and those to wish to run MEET DB2 in batch mode, it can be invoked from a command line interface. Figure 1 shows a screenshot of the generated report on a test database.

The following list summarizes the output of MEET DB2:

1. A report of the percentage of lines of code (LOC) that require modification in order to be migrated from the source system to the target system.

2. A report of the number percentage of database objects that require modification in order to be migrated from the source system to the target system.

3. Detailed list of tables, triggers, indexes, views, functions, procedures, packages requiring modification, including counts of each item and the line number in the input source code requiring attention.

4. Estimates of the human effort required to resolve each identified issue, as well as an aggregate time effort.

The rest of the paper is organized as follows: section 2 gives a background overview of the migration evaluation process; section 3 describes the approach and algorithms used to develop the MEET utility; section 4 describes experiments performed using the MEET utility against a set of customer databases in different market segments of varying complexity and evaluation of MEET's extensibility, and finally section 5 concludes the paper with a brief summary of results and future work.

2. BACKGROUND

Almost immediately after the concepts of relational database systems (RDBMS) and query language (SQL) standards were established in the late 70s (first relational model by E.F. Codd at IBM [4], first specification of SEQUEL by Donald D Chamberlin and Raymond F. Boyce [2], and others [3]), the early RDBMS vendors started adding language extensions. Initially, these extensions were related to the SQL Data Definition language (DDL) and SQL Data Manipulation Language (DML) [8] statements for leveraging storage capabilities, new data types and optimized declarations and hints for faster data retrieval. Subsequently, under the pressure of processing increasing amounts of data, procedural language extensions were added to SQL. These extensions have included control-flow structures and 3G-like languages statements. In most cases these extensions intended to encapsulate business logic closer to the data to minimize context switches between the SQL interpreter/compiler and the storage runtime, reducing many network round trips to applications.

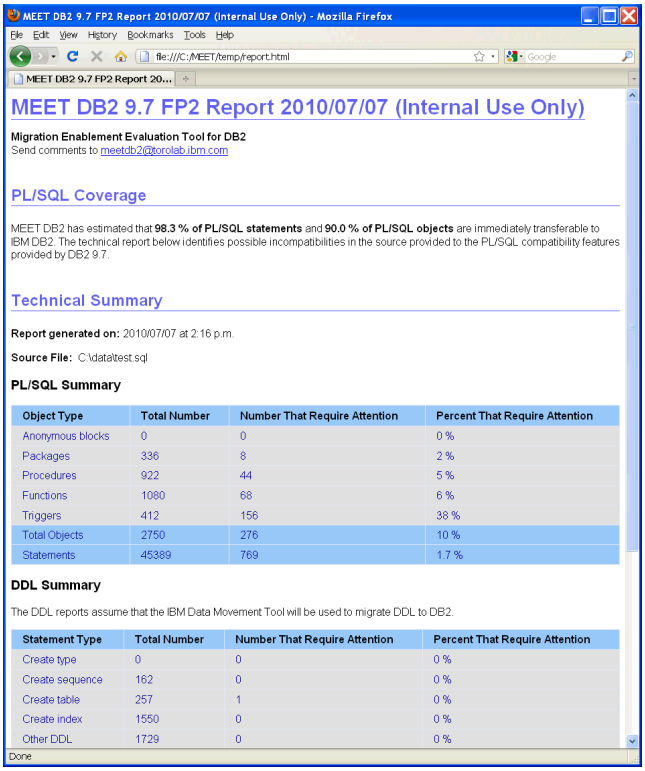

Figure 1: Sample report generated by MEET DB2

Although the SQL standardization went through many iterations and adoptions (SQL-86, SQL-89, SQL-92/SQL2, and SQL-99/SQL3 [6]) and standards for stored procedures were also established (ISO Persistent Stored modules (SQL/PSM) [10]), most RDBMS vendors continued to enhance their own (standard independent) versions of DDL, DML and procedural languages. Therefore, even with the final versions of ISO SQL standards, there are many cross-platform incompatibilities on how different RDBMS handle data, types, procedure logic and transactions, including items such as procedural and control-of-flow logic, common data types, recursive queries, triggers, XML and XQuery as SQL extensions (SQL/XML [9]), and many others extensions.

Migrating applications to a new database platform can be viewed as a multi-stage effort. Gathering and evaluating information about the cost estimation and risk assessment is the key factor in making a migration decision as well as establishing the roadmap for the future project by flagging the difficulties in mapping from the source to the target system. Almost all key steps in the migration process such as initial architecture design, development of migration strategies, testing plans, project scheduling are either based or heavily influenced by the data provided in the estimation phase.

The professionals involved in the migrations need to understand and address manually the impedance mismatch between different SQL flavors, semantics differences and different behavior of language constructs. The challenges can be categorized into one of the following:

- Data definitions (DDL) include the definition of schema and database objects (tables, indexes, views), data

types, and constraints.

- Data manipulations in SQL (DMLs) include SELECT, UPDATE, INSERT, and DELETE statements.
- Business logics that are encapsulated in procedural language, such as SQL/PL. These objects include stored procedures, functions, packages or modules, triggers.
- Application layer code written in C, C++, Java, or other languages.

Evaluating migrations in organized and detailed manner is a difficult and error prone process. In the past, most evaluations were done in manual or semi-manual manner. For example, interviews of clients' database administrators and application programmers, often subjective, are conducted to gather information about the source database. Another important step of the traditional migration evaluation process is to study long questionnaires, completed by the clients, that contain approximate or incomplete information. The incomplete and subjective nature of the manual process has been known to lead to poor evaluation results.

As a simple example, consider a source database that includes support for a built-in scalar function, such as `is_greater_than`, which returns whether the first argument is greater than the second argument. If the target database doesn't support the `is_greater_than` function, this will pose a problem in migrating all of the procedural logic that references this function. Using a manual evaluation process a human expert would review the source database where the application currently runs and identify all occurrences of the function `is_greater_than`, and determine an alternate method of implementing the same capability in the target database. This requires the human expert to be able to identify all such functions like `is_greater_than` that cannot be supported by the target database as well as to clearly understand not only their semantics but also functionalities.

From these experiences different vendors began releasing a variety of migration tools and strategies for automation of the porting process and the cost based analysis, including Migration Toolkit by IBM [7], Monarca Enterprise by Endian Soft, Oracle Migration Bench [13], and Microsoft SQL Server Migration Assistant [12]. These tools encompass research fruits from schema and model management, and more importantly, program transformation [15, 14]. However, most of these tools are designed to focus on the migration of code and only partially to the evaluation of code. Such tools in general is oriented towards engineers rather than technical sales. It is difficult using these tools to take into account compatibility supports provided by a wide variety of database releases and/or third-party tools. In most cases, the evaluation process is still a manual, time consuming task that allows inaccurate interpretation and subjective criteria.

The market needs a new generation tool specialized in providing migration analysis based on evaluations of RDBMS specifics, precise diagnostic of the syntax structures, comprehensive association between syntax structures and the time needed for their new implementation according to skill sets, awareness of the supported features per release, understanding of the API support and the overall interactions with the application layer. MEET DB2 is a prototype designed to automate much of what until now has been a manual process of database design and procedural code analysis. That automation includes the parsing and analysis of DDL, DML and procedural logic to identify the database definition and logic that require modification in order for them to run on the target database. Through automation the process can be made dramatically more accurate, objective, and faster.

3. DESIGN OVERVIEW

3.1 Assumptions and Requirements

In designing MEET DB2, we have been guided by the following assumptions and requirements that have offered both challenges and opportunities.

- MEET DB2 must dramatically reduce the time required to perform a database migration evaluation analysis.
- The evaluation report generated by MEET DB2 should contain both statistics of the input source code and a detailed list of incompatibilities. The statistics can provide the reviewer with the overall complexity of the source code, whereas the list of incompatibilities will determine the migration cost itself.
- Technical sales staff, with reasonable amount of knowledge in database systems and minimal training, should be able to use MEET DB2 and interpret the output. The result should be comparable to, if not better than, in terms of accuracy, a manual evaluation process conducted by an expert in the domain.
- Database vendors roll out new releases with new features every few months and the introduction of new releases bring new incompatibilities. Third-party compatibility layer can also significantly alter the migration landscape. It must be as easy as possible for database migration experts to document these incompatibilities.
- Many tools already exist in the migration space. In fact, many migration experts have written their own utilities and scripts to aid the common migration tasks. The output produced by MEET DB2 should be highly interpolatable so it can be fed as input to existing tools and utilities.

3.2 Components Overview

MEET DB2 consists the following components: preprocessor, parser, analyzer, knowledge base, effort estimator, and the report generator. Each component is relatively independent, as they are linked together by intermediate step outputs in XML format, which vastly increase the interoperability and decrease the dependency among components. They work in the following flow, and when possible, we make links to how computer compilers work.

1. Similar to C preprocessors that handle directives and macros (`#if`, `#define`, `#include`), the preprocessor is a program that handles database commands, e.g. DB2 CLP, that are commonly found in input source code. These commands specify information such as which database to connect, but are irrelevant to the migration evaluation process. The removal of these commands simplifies the implementation of the parser.

2. The parser performs lexical and syntactic analysis on the source file (that contains both DDL and procedural logic) and generates an abstract syntax tree, or AST, of the source file. The abstract syntax tree is represented as an XML file for the analyzer to use. This stage is identical to the parsing stage in compilers.

3. The analyzer pulls incompatibility rules from the knowledge base documenting known migration problems and queries the abstract syntax tree to extract instances of these problems in the input source. The analyzer also gathers statistics about the source code. In this step, the analyzer "compiles" a preliminary report of incompatible instances.

4. The effort estimator groups incompatible instances identified by the analyzer and calculates the human effort required to migrate these instances based on metrics specified in the knowledge base. This is similar to the linking stage in compilers, as the effort estimator links the identified instances to the knowledge base.

5. The last step, the report generator, collects incompatibility information on an aggregated level and does a XSL Transformation to transform the output by effort estimator to a browser renderable report in HTML. By supplying a different XSLT style sheet, the report can be rendered in other formats, such as spreadsheets, PDF documents.

The knowledge base is a catalog of known migration issues. Although it is centrally maintained by a the tool's development team, individual users can also adjust the knowledge base according to their needs. Each entry in the knowledge base is composed of a rule in XQuery, explained in further details in next subsection, and a cost metric. The cost metrics are defined as a range in person-hours, and should cover all aspects of the migration process, including planning, design, implementation, and testing. These metrics were determined based on empirical data provided by migration experts when MEET DB2 was developed. As MEET DB2 is being used in client engagements, migration issues that are identified by migration experts and not yet exist in the knowledge base will be added accordingly. The cost metrics can also be updated when the field feedback differs from the initial estimation.

3.3 Parser and Analyzer

The first step towards any types of static analysis on source code is to perform syntactic parsing and generate an abstract syntax tree, which is a tree representation of the syntactic structure of the source code. It should be noted that this part of MEET DB2 is indeed similar to static code analysis or program understanding tool [5]. Traditional static code analysis tools usually model the source code as a syntax tree in memory, and provide users with a set of APIs in some procedural languages, e.g. Java, JavaScript, C, to query and analyze the tree. One of the most important design decisions in MEET DB2 is to have the parser output the syntax tree to an XML file, rather than modeling and keeping it in memory, and let the migration experts define incompatibility rules in XQuery [1].

This solution has a variety of merits over traditional procedural language APIs. First of all, XML documents by design describes objects in tree structures, and XQuery is created and designed to query these tree-structured documents. While traditional procedural language APIs might be effective in complicated, multi-step analysis cases (e.g. detecting the possibility of deadlock situations), XQuery is a much simpler and natural way to query a tree structure compared with procedural languages, which are designed as general computing languages. Second, a query can be added to the knowledge base and dynamically executed, eliminating the need to re-compile the utility itself. By adopting an open standard, MEET DB2 can leverage the capacity of many existing high-performance XQuery engines. In next subsection we show how XQuery can be used to identify incompatibilities.

Note that the end user does not need to know XQuery at all. XQuery is only used when migration experts find out a new migration problem and need to document it in the knowledge base.

Program 1 AST XML Schema

```
<element name="field" type="string">
  <attribute name="name" type="string"
    use="required"/>
</element>
<element name="node">
  <complexType>
    <sequence>
      <element ref="node" maxOccurs="unbounded"/>
      <element ref="field" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="type" type="string"
      use="required"/>
    <attribute name="lineno" type="string"
      use="required"/>
  </complexType>
</element>
```

The XML schema, defined in Program 1, strictly follows how the syntax tree would be represented in memory. The full specification of the schema, however, is long and less relevant to the topic. There are two types of XML elements in the syntax tree: `node` and `field`. A `node` can contain many child `node`s and represents a grammar element node in the tree, and it is identified by a `type` attribute. Some possible types are `list`, `if`, `sqlStatement`, `table`, `function`, `trigger`, etc. The `field` XML elements describe properties of the `node`s, and the `name` of the `field` varies depending on what `type` the `node` is. For example, in a `trigger`-typed `node`, we need a `field` that specifies the trigger point (whether the trigger happens before the event or after the event). To give you a more concrete understanding of the parser output and the XML schema, Program 2 shows a simple `is_greater_than` function coded in a pseudo language and Program 3 is the corresponding XML output produced by the parser.

Once the AST XML is generated, the analyzer pulls rules from the knowledge base to run against the XML file. The output of these queries conforms to a predefined XML format and can be harvested by the report generator later. As alluded to earlier, the knowledge base consists of a list of XQuery that defines methods to generate statistics on the input source and known migration problems between the source and the target database systems. One property of

Program 2 source code of function is_greater_than

```
FUNCTION is_greater_than(
  left  NUMBER (20);
  right NUMBER (20);
)
BEGIN
  IF left > right THEN
    RETURN true;
  ELSE
    RETURN false;
  END IF;
END;
```

Program 3 The AST XML for function is_greater_than. Some information is stripped for simplicity.

```
<node type="function">
  <name>is_greater_than</name>

  <node type="list" name="arguments">
    <node type="argument">
      <field name="type">int</field>
      <field name="name">left</field>
    </node>
    <node type="argument">
      <type>int</type>
      <name>right</name>
    </node>
  </node>

  <node type="list" name="statements">
    <node type="if">
      <field name="expr">left > right</field>
      <node type="list" name="statements">
        <node type="return">
          <expression>true</expression>
        </node>
      </node>
      <node type="else">
        <node type="list" name="statements">
          <node type="return">
            <field name="expr">false</field>
          </node>
        </node>
      </node>
    </node>
  </node>
</node>
```

the abstract syntax tree is that it is constructed according to a unified structure defined by us, independent of the language production rules of different flavors of SQL. As a result, even though a code snippet can be written in different flavors of procedural logic that have the same semantics, the resulting syntax tree will be the same. This eases the definition of incompatibility rules as we only need to specify one rules for different source platforms for the same incompatible feature that DB2 does not support. It is important to note that although the incompatibility rules can be language-independent, a syntactic parser for each language is still required to generate the AST XML.

The very first prototype of MEET DB2 had a different implementation for the DDL part of the analyzer. Initially, we attempted to utilize the system catalog of the source database to identify incompatibilities. Most database systems provide a centralized repository, usually called the system catalog, of meta information about the database. These catalogs typically include the table definitions, statistics about objects in the database, relationships to other data, origin, usage, and format. All information specified in the DDL are stored as meta data in these catalogs, and it is convenient and natural to perform SQL queries on them. However, since then we have realized client engagement teams usually receive customers' database in the form of a file containing the source code for both DDL and procedural logic. To accommodate this fact, we have decided to deal with DDL analysis in the same way we implement procedural logic analysis.

Also, for a variety of reasons, it is not always possible to have a complete parser for different flavors of SQL and procedural logic. As a result, it is key that error recovery features have been built into the parser itself so that it will not stop at seeing an unrecognized grammar, but rather skip and continue to the next understandable code [11]. In addition, all unrecognized grammar elements are documented and written to the output.

3.3.1 XQuery Rules

In this subsection, we show how to construct rules in XQuery to match grammar patterns that can be used to identify migration problems or gather statistics.

MEET DB2 provides a minimal set of XQuery APIs to facilitate string matching and the construction of rules. Some of the most important ones are documented here:

- The variable `$ast` is the highest level abstract syntax tree node of the source database code.
- The function `first-node` returns the first XML node in a list.
- The function `reportNode` takes a `node` as input and packages it to generate an `instance` XML element that describes each instance of the problems.
- The function `lineOfCode` applies a heuristic algorithm to compute the number of lines of code for a specified object.

Two types of rules exist in the knowledge base: statistics and problems. Although they serve different purposes, they are indeed very similar in the construction. We introduce three types of grammar patterns that can be used for both: basic, child, and aggregate pattern.

In a basic pattern, we are looking for an AST node whose own attributes match specific patterns. For example, some vendors support procedural logic objects defined in Java. When migrating such procedural logics over to a database system that doesn't support this at all, we need to identify all function definition node that is defined in Java. Program 4 implements this rule. `$ast//node[@type="function"]` represents all function definitions and `@language="java"` qualifies that the language attribute of the function definition node has to be "`java`".

Program 4 Basic pattern: functions that are defined in Java

```
<problem name="function_in_java">
{
reportNode($ast//node[@type="function"
    and @language="java"])
}
</problem>
```

Child pattern describes an AST node that has one or more child node(s) that satisfy a condition. Many database systems disallow the use of data update statements in before-triggers for better data integrity. Another example is the pattern of sub programs, or commonly referred to as nested functions, i.e. a function defined within a function. In these cases, XQuery's FLWOR features provide us with powerful child-pattern matching. Program 5 shows the use of FLWOR to construct a rule that can match a child pattern.

Program 5 Child pattern: data update statements used in before-triggers

```
<problem name="update_in_before_trigger">
{
for $trigger in $ast//node[@type="trigger"]
    for $sql_statement in
    $trigger//node[@type="sql"]
where $trigger/point/text()="before" and
$sql_statement/keyword/text()=
("update", "insert", "delete")
return reportNode($sql_statement)
}
</problem>
```

The last one, aggregate pattern, is used when we need to obtain the aggregated attributes, such as frequency or the number of occurrences, of nodes that match specific patterns. The aggregate pattern can be used to detect incompatibilities such as functions that have too many arguments. It is also most useful in gathering statistical information (number of functions, average line of code of all functions) about the source database code, as such information is critical for human reviewers to gain a better understanding of the system. The rules for aggregate pattern commonly use XQuery's aggregate functions (`max`, `min`, `avg`, `sum`, and `count`). These functions take a sequence of nodes as argument and return a single computed value. For example, Program 6 uses `count` function to determine if a function has more than 1024 parameters and Program 7 counts the average number of lines of code for all functions.

Most incompatibilities between different languages can be reduced into one of, or a combination of, the above patterns. Consider a case where the target database does not support overloading a function with the same number of arguments. We can concatenate each function's name by the number of arguments (aggregate) to generate a function signature, and compare (basic) if any of the functions share the same signature.

Program 6 Aggregate pattern: functions with more than 1024 arguments

```
<problem name="function_num_args_gt_1024">
{
for $function in $ast//node[@type="function"]
where
count($function/node[@type="argument"]/node) > 1024
return reportNode($function)
}
</problem>
```

Program 7 Aggregate pattern: compute the average lines of code for all functions

```
<stat name="avg_loc">
{ return avg(
           lineOfCode(
             ($ast//node[@type="function"])))
}
</stat>
```

3.4 Effort Estimator

The effort estimator collects the output from the analyzer and inserts cost estimation into the XML tags. In MEET DB2's cost model, a migration consists of two parts. First is a direct translation that migrates code from the source flavor SQL or procedural logic to the target. This translation is a strict syntactic translation of the code. For example, to rewrite a for loop because the target system uses a different syntax than the source would be called a translation. The second part is the implementation of workarounds for incompatible instances that have been identified by the analyzer. A workaround is when the target database does not support certain features used in the source system, and the migration team would need to re-implement part of the code that uses these features and produce new code for the target system that is of equal semantic behavior. For example, calls to a proprietary natural language processing function would require a workaround. The costs of implementing workarounds are stored in the knowledge base.

Migration costs vary significantly depending on an extensive list of factors, most notably the experience level of the migration team as well as the tools used in the migration process. For example, some tools can help with the direct translation between different flavors of SQL, e.g. IBM's Migration Toolkit, and the usage of these tools can significantly cut down the cost. An extreme case is with the release of DB2 LUW 9.7, DB2 has built-in native Oracle's PL/SQL support and in such case, translation cost is virtually eliminated.

Any model for projecting the human effort required in a migration project needs to factor the large variability in productivity and expertise between individuals who maybe

working on the programming modifications. For each incompatibility issue, MEET DB2 documents in the knowledge base a range of costs that try to model the difference between nominally skilled professionals versus experts.

3.5 Report Generator

The output XML generated by the analyzer and effort estimator is grouped by the types of problems. In migration projects, information grouped by objects is usually more useful as they can be used to determine which part of the code should be migrated first. The report generator uses some XQuery to transform the XML into a report grouped both by types of problems and by objects. It also collects the `stat` tags to build a matrix visualizing the total numbers of each types of objects, the number that requires migration attention, the percentage of such cases, and the cost range, as shown in Figure 1. The report generator then uses XSLT to transform the XML report into a browser renderable report in HTML.

Although the HTML report is more human readable, the raw XML reports are invaluable as the collection of them can indicate what are not supported by the target system and the frequencies of these features used by customers. This feedback information is critical to the future requirement planning of the target system's development.

4. EXPERIMENTS

The goal of this section is twofold. First, we illustrate the extensibility of MEET DB2. Second, we study 18 database sources from external companies to evaluate the performance and effectiveness of MEET DB2.

4.1 Extensibility

Since MEET DB2 is driven by a knowledge base, adding a new incompatible feature, disabling a currently reported feature, or updating a query to run against different versions of database systems takes just a few minutes. The major steps to follow are:

- identifying the pattern of the problem: This is the first and usually the most time consuming step. The DBA needs to research the syntax construction and its variations, determining the cases that are supported (and unsupported) in the target database. The DBA then needs to study the presentation of such patterns in the AST XML file.
- implementing the query to match the pattern: The writing of the query is a streamlined process with a few options. Depending on the pattern, the DBA may inherit an existing query in the knowledge base and only change or expand certain search parameters. In the most complicated cases, the DBA has to write a new query from scratch.
- testing the query: Once the DBA finishes implementing the query, he or she can run MEET on a sample test file to ensure correct reporting. Often the query is not complete at first try, and will require re-iteration of the process.

The time it takes to design and implement new queries varies based on the skill set and experience level of the DBA. To study this, we asked one senior DBA and one junior DBA

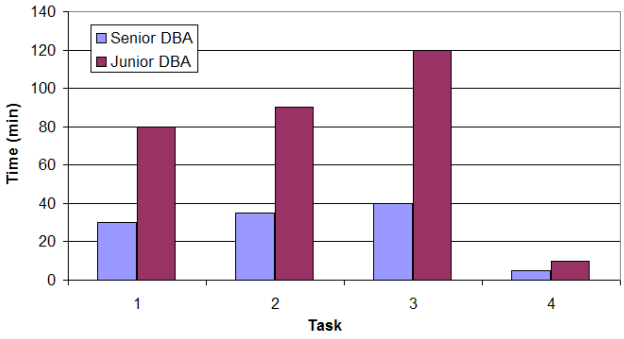

Figure 2: Total amount of time to complete the tasks. This includes all three stages: identifying patterns, implementing query, and testing.

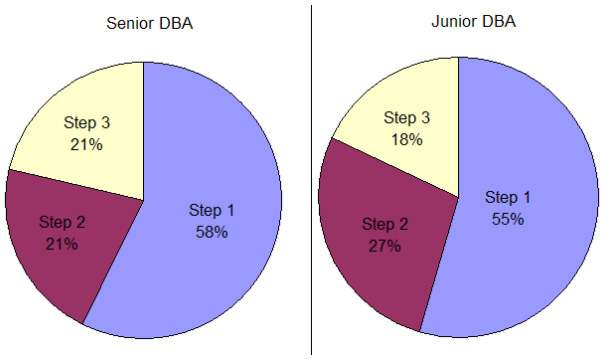

Figure 3: Breakdown of time spent in completing the tasks by steps.

to complete four tasks: 1) adding a query by inheritance; 2) adding a query by expanding an existing query; 3) designing a new query from scratch; and 4) removing a query. The senior DBA has over 15 years of experience in the source and the target database systems, while the junior DBA entered the field one year ago.

We report the measured time in Figure 2. It took the senior DBA less than 40 minutes to accomplish each task, and 120 minutes for the junior DBA. We also measured the time spent in each step. A breakdown of the total time spent in each step expressed as a percentage is illustrated in Figure 3. Using these numbers conservatively, we estimate it would cost less than one person month to implement 100 migration issues.

4.2 Performance and Accuracy

The input used for the experiment was comprised of all objects which contained procedural code which includes procedures, functions, packages and triggers. The input in aggregate included 678,269 statements, and 15,119 database objects, averaging 42,392 lines of code per case study. The lines of code in each case is illustrated in Figure 4. We compare the migration evaluations performed by human migration experts with MEET DB2.

The experiments have the following goals to validate:

- Performance - Greatly decreases the time required for evaluation;

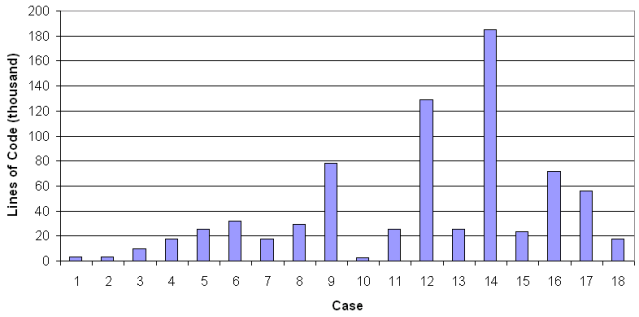

Figure 4: Lines of code for each study case.

- Accuracy - Achieve a +/-5% accuracy rate of evaluating migration code compared to the accuracy of manual evaluations.

By achieving the goals above MEET DB2 is making each evaluator more of an expert. Accuracy of evaluations will be improved overall. The tool will help the evaluators to understand the limitations/issues with migrations and make them experts on evaluations very quickly.

The accuracy of the evaluation in terms of finding instances where workarounds are required as well as the performance efficiency of MEET DB2 are validated by comparing the results of the migration expert versus MEET DB2 and then having another migration expert thoroughly review all the results as well as the database source code to identify instances which may have been overlooked.

4.2.1 *Performance Evaluation*

All experiments using MEET DB2 were performed on a Lenovo T61 laptop with Intel(R) Core(TM) 2 Duo CPU at 2.00GHz and 4GB of memory (3GB usable). The operating system was 32bit Windows XP. We use Saxon-B 9.1.0.8 as XQuery and XSLT engine and IBM JDK version 6. The same system was used by the migration experts for manual and semi-manual evaluation using tools made up of editors, system utilities like grep and sort as well as purpose-built scripts accumulated during their various migration evaluation engagements. The report produced by MEET DB2 requires a human review to ensure the accuracy of the report as well as review of unrecognized grammar elements identified. The elapsed time to perform the evaluation includes the execution time for MEET DB2 as well as the time to review the results by a human reviewer.

Since MEET DB2 reduces the evaluation time by a significant factor, it is less informative to plot the time comparison as the MEET DB2 time will be barely visible. Instead, we plot the time it takes to run MEET DB2 on these test cases as well as the time it takes for a human evaluator to go through the report generated by MEET DB2. As can be seen in in Figure 5, the MEET DB2 runtime is only a small fraction of the human review time. Using the data in Figure 5, we plot the number of factors that MEET DB2 outperforms a human expert on migration evaluations, and the result is documented in Figure 6. The MEET DB2 times used in this evaluation also included time for human evaluation of the MEET DB2 report.

As expected MEET DB2 significantly outperformed the migration expert in all evaluations, reducing the average evaluation time by 97.6%, as shown in Figure 6. In the experiment the MEET DB2 utility outperformed the migration expert by a factor of 41.4 times for the overall elapsed time to evaluate the source and report on the findings.

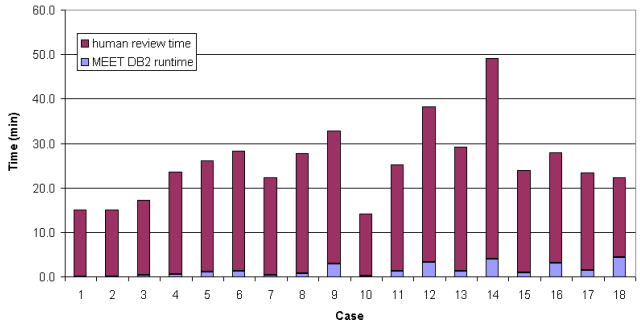

Figure 5: MEET DB2 evaluation time on test cases, including both MEET DB2 runtime and the human review time.

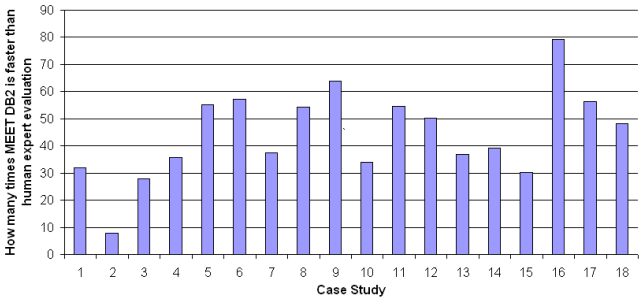

Figure 6: MEET DB2 vs human evaluation time comparison. The y-axis indicates the number of factors MEET DB2 outperforms manual evaluations.

4.2.2 *Accuracy Evaluation*

In this evaluation we also study the accuracy of MEET DB2 by comparing the rate of findings that a human evaluator detected that were missed by the automated utility, and conversely the rate at which MEET DB2 identified issues that were missed by the human evaluator.

Figure 7 shows the number of issues that were not found by the expert and MEET DB2. The accuracy of problem identification reported by MEET was in every case slightly higher than the migration expert. This evaluation is a relative test as it pits the findings of human evaluators against

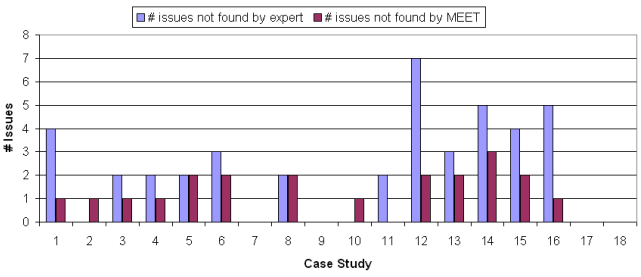

Figure 7: The number of issues not found by MEET DB2 and migration expert.

the findings of the automated utility, however, the accuracy of each is not objectively known. We find that MEET DB2 fails to detect migration problems at roughly half the rate of human evaluators. The exact difference in this study was a 48% reduction in unidentified issues across all 18 databases in the analysis. It is also natural to deduct that with factors such as fatigue, human experts' error rate would increase as the code size increases, while a computer program would not suffer from such problems. In addition, issues that are missed by MEET DB2 can be documented and added to its knowledge base, improving accuracy for future evaluations.

The 97.6% reduction in analysis time and the 48% improvement in problem identification suggest that MEET DB2 is well positioned to deliver on its goals of significant performance improvements for migration evaluation time, and improvements in the accuracy of the evaluation.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have described a prototype technology project called Migration Evaluation and Enablement Tool for DB2. The utility has been shown to be highly extensible and can significantly reduce the time required to perform evaluation for database migration from a disparate platform to DB2 by two orders of magnitude with improved accuracy when compared with manual analysis in 18 case studies. This process, a key task in win-backs and competitive migrations, was previously a manual process requiring extensive analysis by professionals with multi-platform skills. Perhaps even more importantly, through automation, the analysis can now be performed by almost anyone with basic database knowledge. In addition to speed and accuracy, the automation of the migration evaluation process provides consistency across client engagements and by different staff that would not be achievable through a manual process.

Certainly, there are rough edges where MEET DB2 can be improved. Overall, MEET DB2 has produced very good results compared with a human expert evaluation. The effectiveness of evaluation, however, is currently limited to database definitions and database procedural logic. MEET DB2 evaluations can be extended to evaluate application layer, such as those written in Java or C++. The parser component supports very limited number of vendor dialects and can be extended to support more flavors of SQL and procedural logic. In addition, we should investigate what is the best approach in cost modelling and cost estimations. For example, generating qualitative comments (e.g. easy, medium, hard) upon the migration complexity of each incompatibility might be more beneficial than giving out estimations in the form of cost ranges.

6. ACKNOWLEDGMENTS

The authors would like to acknowledge a number of colleagues who helped sponsor and/or develop the first version of the software described here. Much thanks to Andrew Lavers, Chrissie Kwan, Jamie Bennett, David Jacob, Jessica Rockwood, Berni Schiefer, and Sal Vella.

7. ADDITIONAL AUTHORS

8. REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. *W3C Recommendation*, 2007.

[2] R. F. Boyce and D. D. Chamberlin. Using a structured english query language as a data definition facility. *IBM Research Report*, RJ1318, 1973.

[3] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[5] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM System Journal*, 28(2):294–306, 1989.

[6] A. Eisenberg and J. Melton. Sql: 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1):131–138, 1999.

[7] IBM. Ibm migration toolkit user's guide and reference, 2008.

[8] ISO/IEC 9075-1:2008. *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. ISO, Geneva, Switzerland.

[9] ISO/IEC 9075-14:2008. *Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*. ISO, Geneva, Switzerland.

[10] ISO/IEC 9075-4:2008. *Information technology – Database languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM)*. ISO, Geneva, Switzerland.

[11] B. J. McKenzie, C. Yeatman, and L. de Vere. Error repair in shift-reduce parsers. *ACM Trans. Program. Lang. Syst.*, 17(4):672–689, 1995.

[12] Microsoft. Microsoft sql server migration assistant for oracle, facilitating database migration, 2005.

[13] Oracle. Oracle migration workbench white paper, 2004.

[14] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.