

Z/JOURNAL

TOKENIZATION

*A New
Approach to*

**ENTERPRISE
DATA SECURITY**



INSIDE **Z**

The Magic of Keeping DB2 in Order

Linux on System z:
Network Link Aggregation

Workload Manager: Understanding
Goals That Are Too Easy

The Cost of Running Event Processing
in CICS TS V4.1

A
MAINFRAME **ZONE**.COM
PUBLICATION

ARTICLE REPRINT:

**Using In-Memory Tables to Optimize Your Mainframe Environment
Sponsored by DataKinetics**

USING IN-MEMORY TABLES TO OPTIMIZE YOUR MAINFRAME ENVIRONMENT

By William Olders

As businesses try to do more with less and maximize their return on hardware and software investments, optimizing mainframe infrastructure is key. It can offer immediate benefits in performance and revenue, especially to those facing increasing transaction volumes and tight batch-processing windows. Maximizing the Millions of Instructions Per Second (MIPS) a mainframe processes can annually save companies millions of dollars. In situations where mission-critical applications process hundreds of thousands of transactions per hour, maximizing MIPS is imperative.

Companies can take several approaches to optimize their mainframe environments, increase capacity, and reduce costs. This article focuses >

on the use of in-memory tables and table-driven design to address data access inefficiencies and maintenance issues.

In-Memory Tables Overview

In-memory table management complements the features of a DBMS and adds programming power and performance to virtually any application running on z/OS. Production systems often suffer from poor performance or expensive maintenance requirements. To reduce costs and promote responsiveness to business changes, these systems are easily updated with in-memory tables. The greatest benefits occur when the transaction volumes are large or complex.

An in-memory table management system that enables externalization of rules from the applications is a powerful advantage for any application designer. Many applications can benefit when program behavior is remotely managed by the people who use the system.

Improving Performance

A daily database update process may flawlessly perform its automated task, but if it takes 25 hours to execute, it's useless. This situation is a real concern to large organizations with rapidly increasing transaction volumes. Often, appropriate use of memory-resident tables has dramatically dropped the elapsed time of an otherwise well-tuned process from 12 hours to 40 minutes or three hours to five minutes.

Implementing in-memory table access is often a straightforward task that involves replacing tabular data in an external file with corresponding main memory tables of identical design. The purpose is to minimize I/O by buffering the entire table in memory and dramatically reduce the instruction path for accesses. The same logic extends to replacing tables for any file organization or DBMS, assuming the data meets the requirements for reference data or temporary data.

This approach doesn't have functional requirements beyond those already available with standard file or DBMS processing; it doesn't require legacy-oriented application programmers to change their approach. It merely requires a simple one-to-one replacement of file or DBMS accesses with the equivalent in-memory-table access call.

Classifying Tables

To help determine whether a table belongs in a DBMS or in memory, it's useful to think in terms of process-

related data and data that's processed. Process-related data is information that tailors the process for a specific set of circumstances. Data to be processed consists of the primary input and output. With this in mind, we can group the entire data content used by applications into three classes: database data, reference data, and temporary data.

Database Data Class

DB2, IMS, and VSAM contain data that's processed and represents the bulk (80 percent) of the data that applications process. This data is typically randomly processed. Examples are customers updating their own profiles online, or an entire DBMS table processed sequentially in a batch application to create monthly statements. Because data is simultaneously accessed and updated, data hardening using I/O is unavoidable and necessary. Access performance isn't as critical as for temporary or reference data because the same data row isn't repeatedly accessed.

Often, only a fraction of the entire DBMS table is processed in any time frame. In a z/OS environment, DBMS data is served from a single source to ensure all applications have the latest version of a row in a table. The performance issues related to record locking have little impact as this class of data would likely not be simultaneously updated from multiple locations. For example, a credit card isn't typically simultaneously used in multiple locations. Usually, in-memory tables can't help improve the access performance when data updates need to be retained in permanent storage.

There are circumstances where the number of read accesses far exceeds the occasional update access. The trading price of stocks is a good example. Many transactions would be asking the price of an equity, but there may be a thousand requests asking for the current price of the equity for every time there's a price adjustment. When the read-only accesses far exceed update accesses, in-memory tables could make a substantial performance difference if used as a write-through cache. Each time a request for the equity's price is serviced via an in-memory table, substantial performance gains are common by avoiding the trip to the DBMS, which is 30 times slower. When an update occurs, both the in-memory copy of the table's row and the identical row in the DBMS remain in sync.

Reference Data Class

Reference data is generally the most-accessed data by any application because it's process-related data. All reference data, whether used to look up prices or dynamically control the flow of a program, have one property in common: Reference data is always read-only when used in transaction processing. This property allows in-memory tables to provide up to a 30-fold performance multiple over DBMS access times and CPU usage by using pure in-memory algorithms. The greatest gain in performance of an application is achievable by taking this reference data, normally kept in a DBMS, and making it available through in-memory tables.

The 80/20 rule applies for reference data. Suppose 80 percent of your database accesses are against the 20 percent of your process-related data, the reference data. By eliminating four out of five DBMS accesses in your applications, they run in 20 percent of their original time and CPU usage. This simple change in reference table access results in serious MIPS saving.

This data can change yearly, monthly, daily, or virtually never. What distinguishes this class of data is that a table is generally treated as a set of rows that belong together. Consider a table that contains prices for the month of March. This table would be useless if it was partially updated with prices from other periods. Other examples of reference tables are process parameters data, security control tables, process control data, decision tables, rules tables, codes tables, and pull-down lists.

By their nature, some reference tables must be available in sets because data relationships are spread over several tables and entire reference table sets must be retained for transaction reversals and audit requirements. Properly designed in-memory table management supports reference tables that are updated in related sets. z/OS transactions running in a non-stop environment serving data to Web servers must be able to continue running until completion with an existing set of in-memory tables while newly started transactions continue with a new set of tables—all simultaneously and without interruption or delay.

Temporary Data Class

Temporary data—the smallest class of the total amount of data used by application programs—is the essence of process-related data. Examples include:

- In-memory arrays (defined in application programs)
- Data passed between steps of a transaction
- Online transaction work areas
- Online buffer sorting
- Online inter-transaction storage
- Batch-working storage buffers
- Dynamic alternate data views
- Batch data reduction and sorting.

Temporary tables can be re-created should a Logical Partition (LPAR) application fail. So, the highest application performance is possible if all the temporary tables are placed in the memory of an LPAR or in the memory allocated to the application. Most applications developers struggle with program arrays to accomplish searching and sorting to achieve acceptable application performance. The alternative, using a DBMS for temporary tables, hampers performance and is largely inappropriate.

An application that temporarily makes use of in-memory, sortable, and indexable table objects can often add value. Examples include program trading, billing by telecommunications companies, consolidated statement production, tax processing, and much more. An in-memory table used by an investment banker enabled parallel processing steps and reduced an application's nightly batch run-time from 10 hours to two.

An in-memory table manager has a rich set of information-building functions the application uses to dynamically define, populate, organize, and manipulate information in the memory of an application or in the memory of an LPAR shared by all applications. An in-memory table manager supports memory management when thousands of transactions are simultaneously creating and collapsing multiple table objects with multiple indexes.

Making small changes for much-improved performance is a common goal, but the primary motivation for re-engineering usually focuses on flexible application logic and reduced maintenance. High performance is a prerequisite for flexible, table-driven design. However, the most enduring benefits of in-memory tables occur when they control process flow and process decision-making. This is the discipline of table-driven design. Although DBMS tables can be used for implementing table-driven designs, processing time can be seriously compromised by using DBMS tables for this purpose.

Minimizing Maintenance

When mainframe applications are modernized, they must be able to adapt to changing conditions and be inexpensive and easy to maintain. Preferably, these improvements should be made while minimizing risk and modernization cost, something that can be accomplished using in-memory tables.

In-memory tables enable:

- The abstraction of business (if, then, and else) logic
- The decoupling of business rules from the corresponding application processing
- The externalization of business rules updated and maintained via table changes
- The creation of temporary in-memory data structures that facilitate and optimize complex algorithms.

Re-engineering for minimal maintenance is often dictated by implementing new business processing as required and if resources are available. The result is a gradual shift from a procedural to a table-driven approach to application development. Because a table-driven design approach contributes to solving business problems, the design principles that lead to procedural information systems development continue to be used. Table-driven design is an extension of traditional procedural design, not a replacement for it.

Easing Application Development

Table-driven design applies to every stage of the application lifecycle. Tables are used to define, design, develop, and enhance the application system. They provide the foundation for a smooth flow through all stages:

- In the analysis phase, tables provide concise, orderly specifications of the business challenge. The problem here is to restructure existing program code to minimize the impact of anticipated

classes of future maintenance.

- In the design and development phase, tables can be implemented directly from specifications, providing a close link between theory and practice.
- In the ongoing enhancement and maintenance phases, shared tables allow for single, centralized changes, fast turnaround and high productivity, with minimal risk to existing program code.

Abstraction of Rules

Traditional design has, for largely historical reasons, embedded tables of rules in program logic in various forms, such as IF, THEN, ELSE and other branching constructs or declarations of arrays in working storage. The abstraction of rules to separate control data from process is initially more intuitive than obvious. But it's straightforward once you understand that it consists of gathering similarly structured rules together and loading them into a table. Rules are associated with data entities or objects that are being processed.

Consider the snippet of business rules in Figure 1. The values '01', '02' and '03' of the logic are abstracted and placed as individual rows in a table. Included in each row are indicators for associated actions. Then, checking the table at execution time determines what actions should be performed for the given taxpayer type. Equivalent table-driven code would be of the form shown in Figure 2.

The restructured code allows for changes in actions performed for a particular taxpayer type simply by updating the table, without modifying the program code itself. This type of logic

```

IF TAXPAYER-TYPE = '01' OR
   TAXPAYER-TYPE = '02'
PERFORM ACTION-X.

IF TAXPAYER-TYPE = '01' OR
   TAXPAYER-TYPE = '03'
PERFORM ACTION-Y.

```

Figure 1: Business Rules Snippet

```

MOVE TAXPAYER-TYPE TO DECISION-TABLE-SEARCH-KEY.

PERFORM TABLE-LOOKUP.

IF TAXPAYER-TYPE-NOT-FOUND
PERFORM ERROR-ROUTINE.

IF ACTION-X-IS-TO-BE-PERFORMED
PERFORM ACTION-X.

IF ACTION-Y-IS-TO-BE-PERFORMED
PERFORM ACTION-Y.

```

Figure 2: Table-Driven Code Equivalent to Figure 1

construct is sometimes called “action-oriented” as opposed to the more traditional “condition-oriented” constructs so often found in procedural code. This simple change in code decouples the controlling logic from the processing actions. When conditions change in the future, table updates control the processing actions. To create new processing actions is simple in the revised table-driven code.

There are many ways of abstracting logic and the goal is always to decouple the condition from the action to create truly flexible, minimal-maintenance applications. It’s beyond the scope of this article to elaborate in detail on the various decoupling tech-

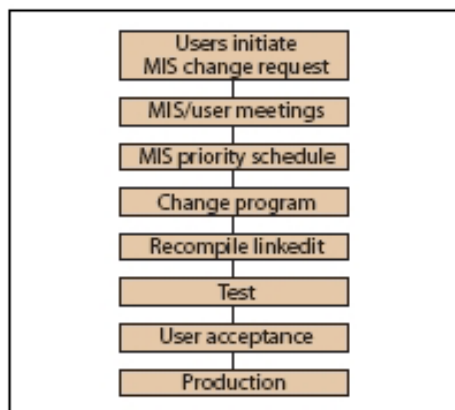


Figure 3: Typical Steps for Processing a Typical Change Request

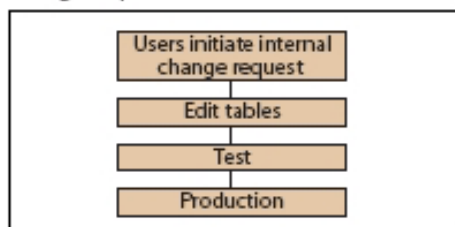


Figure 4: Steps for Processing a Change Request for an Idealized Table-Driven Application

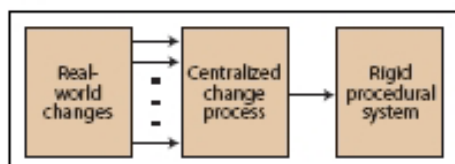


Figure 5: Overview of the Procedural Approach to System Maintenance

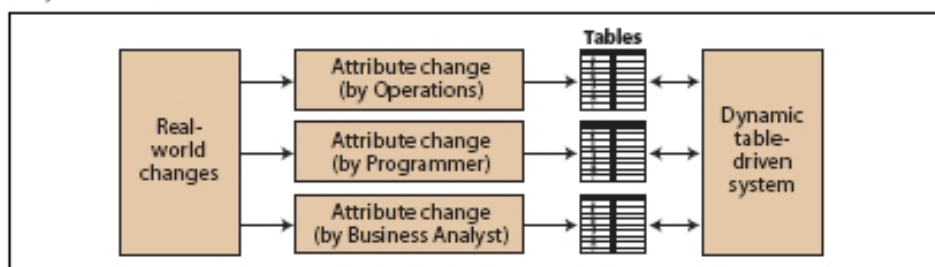


Figure 6: Table-Driven Approach

niques of table-driven architectures.

This decoupling has a dramatic effect on the steps in processing a typical change request for a traditional procedural application depicted in Figure 3. Figure 4 lists the steps involved in processing a change request for an idealized table-driven application, where the business analyst has direct update access to business rules in tables.

Decoupling for Minimal Maintenance

Within many corporate IT organizations, there’s now a clear understanding of the need to decouple process-control parameters from application code to support the development of flexible application logic. Applications may be categorized according to levels of flexibility and ease of sharing, from least desirable (least flexible or easy to share) to most desirable.

Figure 5 presents a graphical, high-level overview of the procedural approach to system maintenance. As the business environment evolves, user change requests accumulate in the IT pipeline. Multiple, unrelated change requests are queued and applied collectively to minimize overhead. Delay is part of the process.

In the table-driven approach in Figure 6, multiple change requests are distributed and independently applied, in parallel, across a series of tables. Each table describes some set of attributes for objects in the evolving business world. Business experts who request changes to business rules also are the ones most qualified to directly apply those changes to the tables that drive the application. Overhead is minimized and, often, IT involvement can be entirely eliminated.

Table-driven design is already familiar to most programmers. It’s commonly found in the traditional concept of a driver program in transaction processing. Logic paths are selected at execution time, depending on the literal value of the current transaction code. Here, though, the transaction code is usually directly associated with a given subrou-

tine in a procedural manner. There’s no way to change this association without changing the driver program itself. A truly table-driven approach would indirectly relate transaction code to subroutine via an external table.

There are many other benefits of in-memory tables, including reduction in maintenance cycle time, a shared understanding of the operation of system and a resultant system capable of user interaction at any level—operations, architecture, programming, strategic planning and tactical adaptation, and others.

The Business Case

While table-driven design is a well-understood concept to most programmers, the business case for adopting this methodology is what’s most important to the executive suite. Fortunately, this methodology offers significant benefits in performance and profitability.

More transactions performed in the same time means more revenue. The increased efficiency gained by using the shortest path to data reduces program execution time from hours to minutes. By placing rules in tables external to the application logic, time to implement is reduced from months to hours. This found time enables IT resources to focus on revenue-generating tasks or other efforts, increasing productivity and saving money. An increased return on investment occurs without the need for migration or costly upgrades.

This methodology works best in environments where mainframes are processing significant numbers of transactions, in industries such as financial, retail, insurance and telecommunications, as increased efficiency in these transaction-rich environments means substantial cost savings and increased revenue. Companies that adopt these practices also enjoy competitive advantages because they can complete processes faster than competitors, accelerate time to market, and react more easily to dynamic market conditions. **Z**

About the Author

WILLIAM OLDERS has more than 30 years of experience in computer language instruction and software product design. He co-founded DataKinetics, where he applied his knowledge of transaction processing and system architecture to design and develop the company’s flagship product, DataKinetics tableBASE. Voice: 613-523-5500 x215 Email: wolders@dkl.com; Website: www.dkl.com

