White Paper

Published: July 2006

Transaction Management Tips for Application Development

Contents

Preface

Every application program is bound to fail at some point in its life – some more than others. This might result from problems such as hardware failure, loss of electricity, programming problems, unanticipated or incorrect input data, and user errors, just to name a few. The causes are many, but the results are the same.

In a business environment, it is usually very important these programs complete their processing in a short amount of time with a minimal amount of intervention. This includes having the ability to quickly restart the program after a failure, without much manual effort or specific application knowledge. The more automated the approach the better. To quote Albert Einstein, "Things should be made as simple as possible, but no simpler."

This white paper provides a simple and proven method for managing large transactions typical to "batch" processing of data. While it does require work to retrofit existing programs with this type of logic, it is usually not too difficult to accomplish.

The approach shown can also be extended to accommodate scheduling of programs and reports. This was an easy extension of the basic restart model that we implemented for a large client. It worked out very well and will be described as well.

Before You Start

The key to success is having a solid understanding of the transactions performed by the application program. A transaction is intended to be a "logical unit of recoverable work." Consistency, data integrity, recoverability, and concurrency are all affected by transaction design – either positively or negatively. We tell our customers that performance and concurrency are *best designed into a system*, but when that is not possible it is important to truly understand the flow, logic, and dependencies of the program and system.

The only time that this is really difficult to implement is when physical transactions are inconsistent in behavior (e.g., commits are conditionally based on data value) or poorly designed (e.g., commits are too frequent). Problems like these are often introduced by inexperienced programmers, or as a quick fix / work-around to an existing problem. At Ingres sites with these problems, you will also typically find that they have also set "readlock = nolock" in an attempt to circumvent the effects of poor transaction design.

The Implementation

This approach starts with at least one new persistent schema object – a table. It is important for this information to reside after the program completes. Please note that this table can become a bottleneck if used by multiple programs simultaneously. In that scenario it is recommended to utilize several tracking tables, up to one per application program.

This table can be used to track the status of processing (just a single row of data), or it can be used to track activity and maintain history (one row per run – useful for performance and trend analysis). The following pages will outline the basic steps required to implement this approach.

Create Tracking Tables accessible by the Application Program(s):

```
-- Single table implementation to log all jobs
Batch_Job_Log (
    Job_Name      char(80) not null not default,
    Job_Step      char(32) not null not default,
    Record_Count  integer not null with default 0,
    Last_Key      char(80) not null with default,
    Last_Update   date not null with default,
    Complete_Flag char(1) with default 'N'
    ) with noduplicates, journaling;

-- Multi-table implementation
Job_Log_<Program_ID> (
    Run_Sequence  integer not null with default 0,
    Job_Step      char(32) not null not default,
    Record_Count  integer not null with default 0,
    Last_Key      char(80) not null with default,
    Last_Update   date not null with default,
    Complete_Flag char(1) with default 'N'
    ) with noduplicates, journaling;
```

Add the restart & tracking logic

Preliminary Work:

1. Identify each "batch" program that will use this restart logic. Each one will require a unique 'Job_Name' (single table approach) or unique tracking table name (multi-table approach).
2. Identify critical steps within this process. Sometimes it is easy to break a program into several logical steps (e.g., 'load and validate data', 'process transactions', 'program clean-up'), while other times there really is only one step (such as 'process transactions'). Either way is fine since it is really dependent on the current application design.
3. Determine commit thresholds, either system wide (easy to incorporate into a common function) or by program (best maintained in a look-up table). This will be used by the program during execution to incorporate many logical transactions into a single physical transaction.

Application Work:

- **[Single row approach]** When the application program starts it should read the Batch_Job_Log table, searching for entries where the 'Job_Name' data matches the programs Job_Name and the 'Complete_Flag' value is set to 'N'. If found, this indicates that a program restart is required.
- **[Multi-row approach]** When the application program starts it should read the Job_Log_<Program_ID> table, searching for max sequence number (assumes sequential assignment in an ascending order) where the 'Complete_Flag' value is set to 'N'. If found, this indicates that a program restart is required. Further granularity can be provided to track the processing of each step. While this increases the overall complexity of the model, it may provide valuable information for the performance profile.

- If a program restart is necessary:
  a) Print a message / send an e-mail indicating that the application is being restarted. Provide starting point information (from the appropriate tracking table). It is important for someone to know that this event occurred so that they can identify the root cause.
  b) The program should "fast forward" to the area that is indicated by the Job_Step. Ideally the data used to get to this point (such as a temporary table used to load the data from a flat file) still exists. If that is not the case then additional logic needs to be incorporated to reload the data. It is very important that the data remain in the same order as originally processed.
  c) The program should find the last row processed (using the 'Record_Count' and/or 'Last_Key' columns).
  d) The program should then resume "normal" processing from this point forward. Most of the time it really is this simple.

- Normal program start (not the restart previously described):
  a) Program variables should be initialized to the appropriate values and updated as needed. Data such as record count, last key and complete flag will need to be updated in an ongoing basis. **Insert a new row into the tracking table.**
  b) Commit thresholds should be retrieved (if necessary).
  c) Begin processing, incrementing the transaction counter variable for each logical transaction processed.

- Once the program is running:
  a) Check every transaction to see if the transaction threshold has been reached. If yes, update the current job tracking row, commit the physical transaction, reset the transaction counter variable to zero, and resume. **Note**: If everything works well this tracking data will be committed along with the transaction data. If there are problems this data will be rolled-back along with the transaction data. Either way this table has an accurate view of where processing was at the time of failure.
  d) If tracking changes, prior steps should have their complete flag column set to 'Y' prior to moving on to the next step.
  e) At the end of processing it is very important to remember to update the Complete_Flag value to 'Y' (if tracking changes) or to delete the row(s) used to track this job.

Enhancements for Scheduling

Extending this model to accommodate scheduling requires tracking more data, and a procedure to find and execute the schedule processes. Typically we add columns for user information (login ID, email address for sending notification), submit date / time, schedule date / time, run date / time, completion date / time, specific run command, etc. We've used this for tracking all reports run at one client (interactive and scheduled), and that information was helpful for performance tuning and capacity management.

Once scheduled, a process will be required to find and execute those items. Rules for execution should be defined and incorporated into that process. For example, "A process is a candidate for execution if the current date / time is greater than or equal to the current time (beware multiple time zones), and the schedule time is less than or equal to 24 hours prior to the current date / time." If for some reason an End of Month process did not complete at the scheduled time, it could be potentially harmful to run it several days later. It is also possible to codify this behavior, allowing multiple allowable variances.

Other Applications

We recently used this approach to consolidate a complex job stream for a Client. There were nearly identical "start" and "restart" streams that each consisted of over a dozen individual components. This approach allowed us to consolidate both job streams into a single process, incorporating the enhanced transaction management and restart ability. This simplified their production management procedures and created a more constant load on the production systems.

Summary

Consistency is almost as important as performance in a production environment. End users become frustrated when an operation takes 1 second most of the time and 30 seconds other times. Problems such as this are generally concurrency related. Implementing this type of transaction management model often improves the overall concurrency within a production system, contributing to more consistent processing times for all processes.

This is a simple yet elegant method to handle automated program restarts. It can improve performance and concurrency, and minimize problems resulting from excessive transactional logging of long transactions. It can also provide historical data for trend analysis and performance profiling.

This model requires and understanding of, and modification to, the existing transaction design within a program to be successful. Good transaction design is beneficial for recovery without compromising data integrity, so there is value to the transaction review outside the scope of this effort. This also provides an opportunity to create documentation that will be valuable later.

It is recommended that transactional analyses be performed (such as the Ingres Print Query tracing tool) to validate the transaction design (before and after) and restart logic. Failure to do this could create a condition where data is omitted after a restart, or where duplicate data is added to the system. The value of testing and quality management should never be underestimated.

About the Author

Chip Nickolett, MBA, PMP is the President of Comprehensive Solutions. He has been using Ingres since 1986, and first implemented this approach in 1987. Since then he and his team has successfully implemented it in various forms in numerous environments. The model has survived the test of time and is highly recommended.

Let Us Help You Succeed!

Call today to discuss ways that Comprehensive Solutions can help your organization save money and achieve better results with your IT projects. We provide the ***confidence*** that you want and deliver the ***results*** that you need.

Back to White Papers

Back to Services page

Comprehensive Solutions
4040 N. Calhoun Road
Suite 105
Brookfield, WI 53005
U.S.A.

Phone: (262) 544-9954
Fax: (262) 544-1236