Data Selection Patterns in Batch Programming

Or: Why batch programming isn't that trivial after all

Thomas Holzer
Xenium AG
Elektrastr. 6a
D-81925 Munich
+49 178 7826426

thomas.holzer@xenium.de

Why Batches ?

A batch typically is a program designed to apply operations to a multitude of objects. This might be to implement a part of the business logic of a system or to do technical mass operations like import and export of data, cleaning up, or checking data for errors. This paper focuses on batches running on a database server implementing technical necessary operations as well as business logic.
An important use of database batches is to take load off the primary transaction in business systems by processing data during times of low server load. Another important reason for implementing functionality as a batch is to connect to neighbor systems to reduce complexity in the overall system.
Some periodical tasks are of batch nature themselves, and are ideally implemented as a batch. For example, payments due to the end of a month have to be calculated to a given date.
This paper discusses several ways to implement mass data processing using a relational database system.

As running example I will use a large software system managing (among other things) social security details for millions of people using a relational database. These details are altered during the day by the users of the system. Nightly batch runs extract data and send the differences to various institutions, e.g. health insurances. All data sent has to be stored in the database so the users are able to review what data has been sent.

While speed is essential to process the load in the given time frame, there are several other forces to be considered.
A batch has to provide a means to stop and restart processing at any point. This is necessary to counter technical problem like unavailable neighboring systems or varying server loads. In times of lower loads more batches can be started to use idle resources, while in times of increasing server load batches have to be stopped to ensure low latency time for online users.

Batches have to be robust in dealing with unavailable or incomplete data. They should be able to postpone the processing of certain entries instead of exiting with an error message. This is useful in case data is temporarily locked by another process or is yet missing completely. If there is the possibility to successfully process the data at a later time, it should be skipped for now. Batches usually run without any user interaction during the run.

A batch using a database should always make full use of the database's transaction logic to prevent loss of data or inconsistent states in case of error. The business logic implemented by a batch may not prevent working in "batches" of entries.

About this paper

This paper contains a first pattern out of a pattern language for batch processing. It is based upon patterns which are only briefly described as pattlets at the end of the paper. The following figure describes the pattern discussed in this paper and some patterns connected to it.

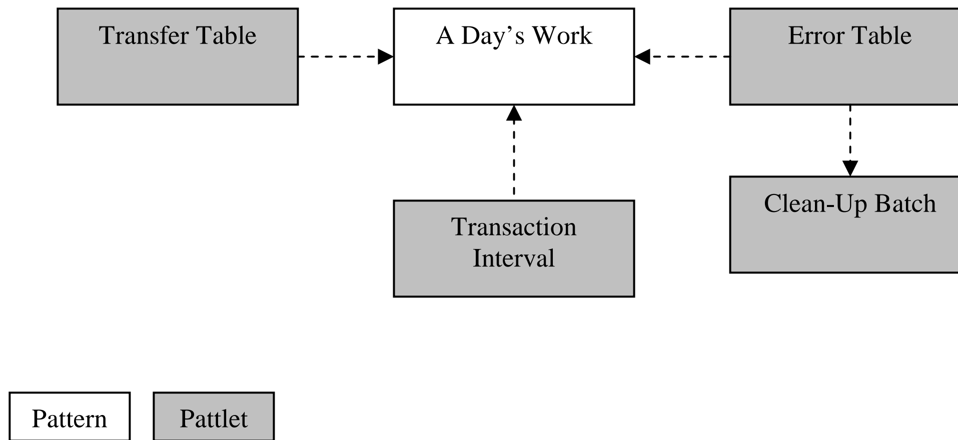

Figure 1: Data selection pattern overview

A Day's Work

Context

You're using *Nightly batches*[1]. The input data for a batch is stored in tables filled by the online system. Newer data may invalidate older data. The input data contains data in various stages, new data as well as invalidated data mixed with already processed data. The daily data input is very high. For example the batch has to determine the differences between the newest online data and the already sent data to send the differences to neighboring systems.

Problem

How do you select only the newest entries in reasonable time?

Forces

Online system load: It is necessary to minimize the load of the online system. The load increases if the application has to find and mark or delete every old entry when inserting new data.

Batch manageability: It should be possible to interrupt the batch and restart it at a later time to react to high server loads.

Batch processing time window: Sometimes batches have to process data before a certain deadline. This means they must be completed inside a predefined time window. It should be possible to calculate an approximate run time.

Batch performance: Processing of the data should be as fast as possible. This may be to minimize server costs or to generally make the system easier to use by increasing the free time for other tasks and reducing the risk of being interrupted by events outside the scope of the software system.

Batch resource usage: There has to be a predictable upper bound of used database locks, threads and memory.

[1] *Patterns* and *Pattlets* will be typed in italic and are explained at the end of the paper.

Batch stability: The batch should be able to recover quickly from fatal errors. Fatal errors may be caused by unavailability of neighbouring systems or technical errors like no free file handles or not enough free memory and may occur any time.
It might be necessary to postpone the processing of single rows of data if they are erroneous or depending on external data not available at all times.

Batch maintainability: It should be easy to understand the program code of the batch program. This also means the data selection method should be easy to implement.

Solution

Select the data by the timestamp of their creation or last update. This is easy to implement since data often has a timestamp. It is easy to add a timestamp field if needed.
All data with a timestamp newer than the last batch run is selected.
The most simple way to implement a transaction interval would then be (in pseudo SQL):

```
select <data>
from <tables>
where timestamp >= <date of real last completed run>
```

ID	Data …	Timestamp
47110815	…	2006-04-04 10:57:22
32420816	…	2006-04-04 09:44:30
10070800	…	2006-04-04 08:32:01
47110802	…	2006-04-03 23:13:49
10120808	…	2006-04-03 22:45:40
47110803	…	2006-04-03 20:02:16

Table 1: Relational data ordered by their timestamp

If the last batch run ended 2006-04-04 at 06:32:18, the first three rows of Table 1: Relational data ordered by their timestamp would be selected.

The date and time of the last batch run must be set inside the transaction implementing the batch logic to ensure a consistent state in case of a fatal error. An easy way to implement this is to store this data in the same database as the input data. This means the processing has to be completed within a single transaction.

Consequences

Online system load: Since a batch may consume a lot of database resources, it may slow down the application servers too much. This means the batches can only run during times of low online system load.

Batch manageability: It is not advisable to stop the processing in-between and restart later from the same point. If stopped, everything has to be rolled back and re-done, losing all processing already done and even adding additional effort to restore a consistent state.
This also means it is not possible to fine-grain the processing interval. This also decreases the overall system stability, since it cannot react properly increasing and decreasing server loads.

Batch processing time window: If batches have to be finished at certain times, you have to create a batch schedule to ensure the timely completion of critical tasks. You might use *Nightly Batches* whenever possible make better use of limited time windows. It is impossible to create a useful schedule if the batch run time cant be predicted. If it is critical to stop the processing at a certain time this pattern is not applicable. If a batch has to be stopped after exceeding its time window, everything is rolled back and may lead to the same problem at the next try.

Batch performance: All data is automatically marked as old by the passing of time, no entry needs to be marked. This reduces the processing overhead used to manage the data.
Invalidated entries are simply ignored by not regarding them for processing. This means all data has to be read sorted by timestamp (among other criteria). Data reading is fast because there is virtually no processing

overhead. If data is sorted by criteria other than the timestamp, additional indexes on these attributes have to be created to ensure efficient reading from the input tables.

Batch resource usage: This technique is only useful if the size of the data which has to be processed in one batch run is small enough to be processed in a single database transaction interval. If the needed processing order of the data is not the timestamp, interconnected data may otherwise end up in different technical transactions.
This solution creates an upper bound for the data size the batch is able to handle. If the size is too large, the batch cannot guarantee limits for used database resources. A way around this is to implement a custom rollback mechanism which works with several technical transactions. But this means a lot of extra developing effort and ignores one of the main advantages of using a database system.

Batch stability: It is not possible to partition the input data. All data of the selected time interval has to be processed in one technical transaction.
Different rows may contain data of the same object, e.g. a row of data containing base data of an object and several additional rows containing additional information. If the business logic requires this data to be processed together it has to be done in the same technical transaction or else the database rollback mechanism won't guarantee a consistent state of all data.
While it is possible to postpone processing of erroneous entries, a fast recovery from a fatal error is not possible. All data manipulated since the date of the last completed run has to be rolled back to its state before the batch run. This is very costly.
If the time window for batch execution is very small and the batch cannot be started for some time (for reasons the batch cannot control), data piles up in the input table, increasing the time needed to process all entries. The resolution of the timestamp determines the minimum partition size. E.g. if the timestamp consists only of a date without time information, it is not possible to partition the input data on a smaller scale than a day. If the time window for running the batch becomes too small, it might even become unusable at all. So this selection method is only useful if there is a low risk of fatal errors during the daily run.
The database locking level on the used tables influences processing strongly. If page locking mode is set using more than one thread, the batch may run into a deadlock. In page locking mode, only memory pages of the database system are locked, not single rows. This is often done to reduce the number of used locks, because lock management causes additional overhead. If thread A has locked page number 100 and is waiting to lock page number 101, while thread B has locked page number 101 as is now waiting for page number 100, they are both deadlocked. This may happen since data is distributed over memory pages in any order. The only way to use this pattern with multi-threaded batches is with row locking on all tables that are both read from and written to.
If row locking is used, the usage of database locks increases. Consider that there is not only one lock per row, but also one extra lock per index on that table. This decreases the stability of the batch since the database might run out of locks, terminating the batch when requesting locks over the limit.

Batch maintainability: This solution for selecting the data is relatively simple to implement, therefore the source code can be made easier maintainable.

Resulting Context

To limit the database resources used concurrently by the batch, it is possible to introduce a *Transaction Interval* into the batch by committing the processed data after a configured number of records. But this only limits the resource usage. If one commit fails, all others have to be rolled back by the batch. It is therefore not advisable to have the logical transaction size different from the technical transaction size.
A simple solution for decoupling the online system and the batch processing is using a *Transfer Table*. The online system only writes into that table while the batch reads from it. The batch may delete entries from the transfer table to keep it small without bothering the online system with comparing data.
You might want to use a *Clean-Up Batch* to delete the processed entries from the transfer table to decouple processing and deleting of the transfer data.
Postponing processing of certain entries can be done using an *Error Table*. But you have to make sure the data doesn't get deleted before it is re-processed. When working with a *clean-up batch*, this batch also has to care for that.

Pattlets

"Nightly Batches"

How do you organize batches when the online period of your system is restricted to the working hours in a few adjacent time zones? Use the night time to run the batches according to a fixed batch schedule.

"Transfer Table"

How do you decouple producers and consumers which use data stored in the same tables ? Copy the data for the consumer into a specially designated transfer table and delete it when processed. This may be used to keep the primary transaction smaller by moving work to a secondary transaction,

"Database Transaction Interval"

How do you ensure consistent data in case of errors in a batch run? Always use the transaction logic of the database by sending begin/commit pairs at the beginning and end of the program. Try not to split up a technical transaction into smaller transactions, because this renders the rollback mechanism of the database useless. Be careful to have a consistent state at each commit or you will be forced to implement your own rollback operation manually. To limit the use of database resources, do a commit and new begin after a block of data has been processed. Be extremely careful when working with two (or more) databases.

"Error Table"

How do you postpone processing of certain entries inside a *Transfer Table* ? Use a second table to store the identifiers of the erroneous entries. Process these before the next batch run. Make sure the entries are not deleted before re-processing or invalidation.

"Sequence Number"

How do you ensure unique identifiers for database tables? Use the built-in mechanism of database state variables (often called "sequences"). These are state variables containing a number which is increased every time it is read.

"Configurable Transaction Interval"

How do you optimize the usage of database resources without knowledge of the productive load characteristics? Make the size of the transaction interval configurable. This is easily achieved by adding a variable containing the maximum transaction interval size allowed. An even more flexible solution would be to store the configuration variable in a file or the database itself and periodically read it. This allows adjusting the transaction size during run time,
The pattern adds to safety, robustness and restartability, and greatly supports running a batch while costing only little effort.

"Clean-Up Batch"

How do you split processing of data and deleting of unused input data in batches? Create a second batch concerned only with cleaning up outdated data. This supports batch stability and multi-threading since less locks are used during the normal batch run.

Acknowledgements

Special thanks go to Jens Coldewey who suggested writing down my work with batches into patterns and who was also an invaluable help in shaping the paper.

References:

Jim Gray: **Transaction Processing : Concepts and Techniques**, Morgan Kaufmann, 1993
Martin Fowler**: Patterns of Enterprise Application Architecture**, Addison Wesley, 2002