responsiv

simple · effective · distinctive

Digital-SOA
The Case for Real-Time

WHITE PAPER

Digital-SOA

WHI-DSOA (Mar17) The case for real-time v0-2.docx; (02-Jun-2020)

Preface

This paper considers the merits of event (real-time) and batch oriented processing models for architects and developers who are considering using messaging as part of an integration solution. Technical understanding of basic integration concepts, including messaging, and databases is assumed.

Batch models are considered to be high throughput models for use when response time is less important. Event models are, often conceived by designers as only suitable for systems that require fast response and that have less focus on throughput.

This paper considers the distinguishing characteristics of the two models and investigates examples where each may have unexpected benefit.

The Digital-SOA style of architecture, using multi-instance, distributed services supports the use of event models in many more cases than may have been previously considered.

The objective is to design an extensible transaction processing solution with no single point of failure. Because of the time and disruption that it would cause, the solution should never require restoration to a known point. Digital-SOA is to some extent reliant on real-time to assure synchronisation between channels and to accommodate bi-model IT, Omni-channel, declarative interfaces, mobile technology, and customer-centricity at its very core.

Intended Audience

This paper is intended to support executive and leadership teams in **consumer focused companies** to understand what Digital-SOA and Digital economy really mean, and how they must be addressed.

Disclaimer

Observations and recommendations documented in this white paper are based on our opinions, experience, and research. They are as objective and representative as we can reasonably be, however Responsiv makes no representation as to accuracy or fitness for purpose. Once you have chosen a course of action, it should be thoroughly evaluated to ensure its fitness for purpose.

About the author

Richard Whyte is an accomplished IT architect with a proven ability to innovate and focus on customer requirements to deliver simple, effective solutions. He has demonstrated thought leadership based on a breadth of technical and project experience spanning Investment Banking, retail, Manufacturing, and Aerospace; delivering sustainable technology to some of the world's largest companies. Richard has a degree in statistics and computing and a Master's in Business Administration. He is a Fellow of the British Computer Society (FBCS), Chartered Engineer (CEng), Chartered IT Professional (CITP), and Fellow of the Institute of Engineering and Technology (FIET).

About Responsiv

Responsiv Solutions is a UK based company that specialises in delivering business integration across the enterprise, including API management, Business Process Automation, and Digital-SOA platforms. We work across many industries, including Retail, Financial Services, and Government.

Responsiv can provide fully commissioned solutions that include all the software and professional services needed to deliver an integration platform to support your business plan and grow with your business.

www.responsiv.co.uk

IBM Platinum Business Partner™

Table of Contents

Introduction

The finite batch window has been at the heart of a long and tiresome battle for many IT professionals. Work collected into batches during the day and processed in a batch overnight often assumes sole access to the data it manipulates; frequently work is performed using long-running transactions, or unsafe transactional semantics, which preclude sharing with on-line traffic. Because of the avalanche of data and intensity of processing during a batch, the impact on response times, and especially predictability of response, is also an intolerable side effect.

Nevertheless, time available for batch work is reducing while the workload is increasing. A solution needs to be found.

For many years the accepted way to process large volumes of transactions has been to collect them together to be processed as a *single* batch using a single transactional unit of work. This wisdom is based on the need to reduce the overall cost of processing the work and to simplify recovery operations.

"Single *instance batch"* is a simple to understand, effective model. It reduces the absolute cost of processing an individual work-item at the expense of response time and horizontal scalability.

- **Recovery** from any failure involves discarding the output, fixing the problem, and re-running the batch. Systems capable of implementing such regimes are "islands of automation"; they have limited external interaction, and no need to coordinate their recovery with that of any other system.
- **High latency and unpredictability** of individual responses is caused by the batch creation activity and transactional design of the batch model. (A batch is not finished until all the events are processed).
- **Hardware constrained** solutions need to consider the batch approach to achieve efficient (optimal) use of a single machine. The batch model often inhibits horizontal scaling across multiple pieces of hardware; this is true even when the solution naturally scales horizontally (the batch technique introduces the constraint). The model is highly dependent on the performance characteristics of the hardware.

The finite and shrinking window for batch work, increased volumes, and need for more responsive solutions mean that massive improvements for hardware performance alone cannot achieve the throughput needed by modern business.

The *event-processing* model works on the principle that events are self-sufficient and processed individually. This model continuously utilises resources instead of waiting for the batch to start. In solutions with sufficient resource this model can process work-items as they arrive, leading to reduced, and predictable response time.

This paper will document the batch and event models and compare the two approaches.

Summary

You may choose to read the supporting text before the conclusion, which is placed here for those who want the bottom line before reading on.

The conclusion of this article is to adopt the event style wherever possible. Use localised batching when there is a demonstrable need to optimise interfaces and specific parts of a solution. Localised batching is the limited and encapsulated incorporation of batching into a predominantly event system, as a performance optimisation to that system. For example, many databases support "group-commit", a batch technique to optimise commit activity. IBM MQ® uses configurable batching on its channels to optimise throughput and latency.

Use of localised batching techniques provides batch benefits within a predominantly event oriented architecture. This approach is a good compromise bringing benefit from both approaches.

End-user and customer demand for 24x7 availability and fast response times has clashed with batch processing schedules. For many organisations the batch window is an expensive inconvenience that decreases revenue and availability, with potential to damage customer satisfaction and loyalty.

- There are situations that *demand* a batch-oriented solution, but not as many as previously thought. Event-oriented solutions are applicable alternatives whenever events are independent and can be processed in parallel.
- "fix forward" Avoids the cost and time needed to recover to a known point. Instead the problem is fixed; processing restarts; Integrity and other problems are resolved during processing.

- Use of "backup for recovery" for data stores with data dependencies beyond their control, for example other applications or businesses, ignores the post-trauma synchronisation of operational data and state across the architecture; a problem that often makes backups an unrealistic option for timely and accurate recovery.

Consider a batch solution when:

- response time is not important;
- events need to be processed in order;
- hardware availability is severely constrained leading to a need to maximise throughput with limited resources;
- events have a high degree of data sharing
- Events are highly sensitive to data locality

The event paradigm allows a simple, extensible solution to be built to solve today's problem without compromising its ability to scale for tomorrow's challenges. Many batch solutions require radical changes to the system concept and implementation characteristics when they exceed their design capacity. These changes often affect all aspects of the system across the entire solution, including operations, development practices, design cohesion, and supporting software. The event model, once established, has a more localised effect that is less expensive to change, understand, and manage.

Consider event solutions when:

- response time is important;
- events can be processed in any order.
- Techniques can be used locally to overcome more significant constraints, for example isolated need to process events in order.

Events may be a poor choice if hardware is a constraint or the solution is unable to scale horizontally. Efficiency of Horizontal (parallel) scalability is highly influenced by the need to data share between events. Regardless of response time or additional constraints not listed above, the event model is a flexible, extensible, and scalable model.

- Throughput and response are preserved by provision of capacity to accommodate the peaks. Any workload that arrives in "batches" or at particular times will cause higher peaks.
- Data integration using real time messaging makes data available as soon as possible, and dislocates one system from another. Decisions about how to process the data and how to fill peaks and troughs in demand are given maximum latitude by this approach.
- Advances in technology make distributed workloads and event models a viable choice for high throughput systems.
- Flexibility, structure, and agility have a price. The price for using an event model is an increase in the cost of processing individual events. The event model when appropriate can scale beyond the maximum capacity of a similarly sized batch system.

Event model scalability derives from effective use of many smaller hardware platforms and ability to add or remove capacity on-demand. Processing work as soon as it arrives reduces bottlenecks and improves response times.

Batch Solutions

A batch is a collection of events. A batch solution processes each event from a batch in any order, until all items are processed.

A serial batch solution processes items in the order of the batch. In other contexts, "batch" can refer to a collection of programs that execute as a dependent stream of activity.

Batches are created by collecting events until some arbitrary completion milestone causes the batch to be processed. Typical milestones include introspective; Number of events, Delay to first event, file size; or external, time has passed, or some other external event is detected.

Database update pattern

Figure 1 shows a batch containing five events.

The batch processor processes each event into a repository. The work-stream can be represented as a file or a stream of events from a single source, for example a database table.

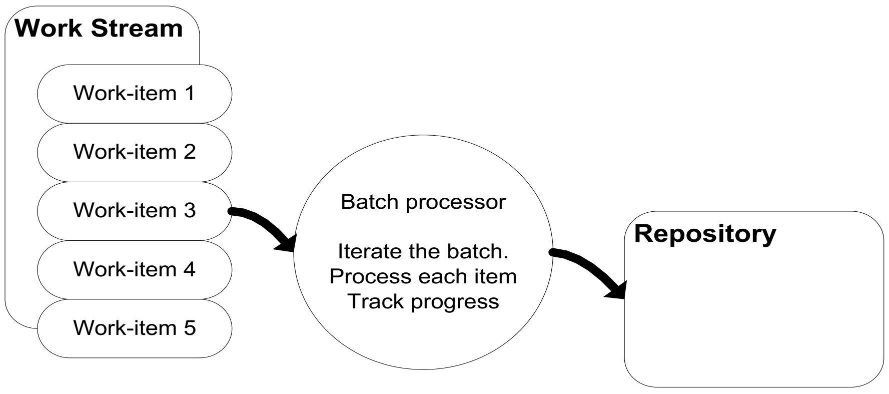


Figure 1: A simple batch schematic (database update)

To process work, the batch processor iterates through the work-items to process each one. It must understand how to handle each item in the batch; the structure of the batch; rules for batch exceptions; rules for item exceptions; and how to track its own progress through the batch.

The batch processor is responsible for recovery from failure; including program failure, work-item failure (validation) and batch failure. It must marshal the batch before processing and confirm that it is consistent and complete before work begins.

File to Database (variant 1)

In this example the work stream resides in a non-transactional file store and processed by a single batch-processor into a database. Programs that perform this type of work often use a single transaction for the entire file, or ignore recovery all together.

Using a **single transaction** for the entire file will incur volume dependencies caused by increased use of journal space and locks on the database and the need to roll back the entire batch if any single event fails.

To avoid volume dependencies a "tracking table" can be used to record progress in the database.

This table is committed in the same unit of work as the data and accurately records where the program should restart in the event of a failure. A tracking table allows a number of events to be committed at once. It is an efficient solution that combines the best of event and batch to reduce overheads, improve throughput and minimise cost to response time and volume dependencies.

Database to database (variant 2)

In this example the program needs to read events to process them, but also needs to update or delete the events from the input stream in the same transaction, to avoid re-processing them.

If the program reads first (shared-lock) and returns in the same transaction to update or delete the record (exclusive) it will cause deadlocks and high rates of contention as the number of instances rises. This approach will not tolerate multiple instances of a program working on the same input stream. To enhance scalability, perform the update before the read, or perform the read as an exclusive operation.

Master-Transaction merge pattern

The Master-Transaction merge (M-T) was one of the first data processing models to process high update volumes. The M-T pattern is an efficient way to process updates using serial, ordered data storage; for example, tapes, or sequential files.

High throughput is achieved at a low hardware cost; the model is applicable when best use of serial storage or hardware expense is the most important constraint. It does not support solutions that require interactive response times.

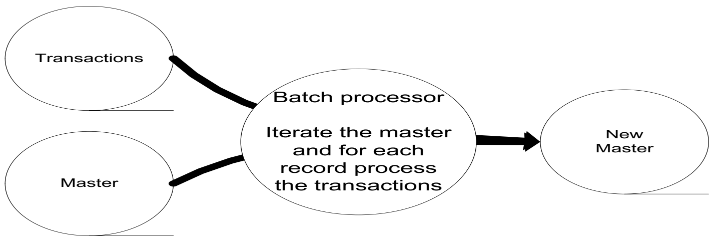

Transactions
Master
Batch processor
Iterate the master and for each record process the transactions
New Master

Figure 2: Master-Transaction Update pattern

The M-T pattern uses two separate data sources. Master records are in a known order, and the transactions sorted to the same order.

Records are read serially from the two input streams. Appropriate transaction and master records are combined to create a new master record. The resultant new master is written to the new master file. The master file will retain record order and does not require a sort before the next run of the batch.

The M-T technique is very effective. It removes multiple scans of serial media and optimises use of memory and machine resources. Recovery is achieved by discarding the new-master tape and running the entire process again.

One of the biggest bottlenecks of this style of processing is the batch processor waiting for slower devices to complete before it can continue. If the process needs to print a record directly to a line printer the process is only as fast as the printer can print.

Introduction of print-spoolers to mediate between the batch and the slower printer vastly improved overall batch performance. The job of a print spooler is to queue work for the printer and isolate the job from the printer, allowing the printer to operate as a separate, parallel operation. The use of asynchronous communication between distributed components in an event model benefits in the same way.

The pattern contains many of the fundamental characteristics of batch processing, including locality of data, resource management, synchronisation of access to resources, and Control.

Fast Batch (Moving from batch to events)

The "Fast Batch" technique splits a batch into fragments that are processed in parallel. Generally, fragmentation is arbitrarily. Fragmentation according to some partitioning key, for example account number can reduce contention and improve data locality issues already discussed.

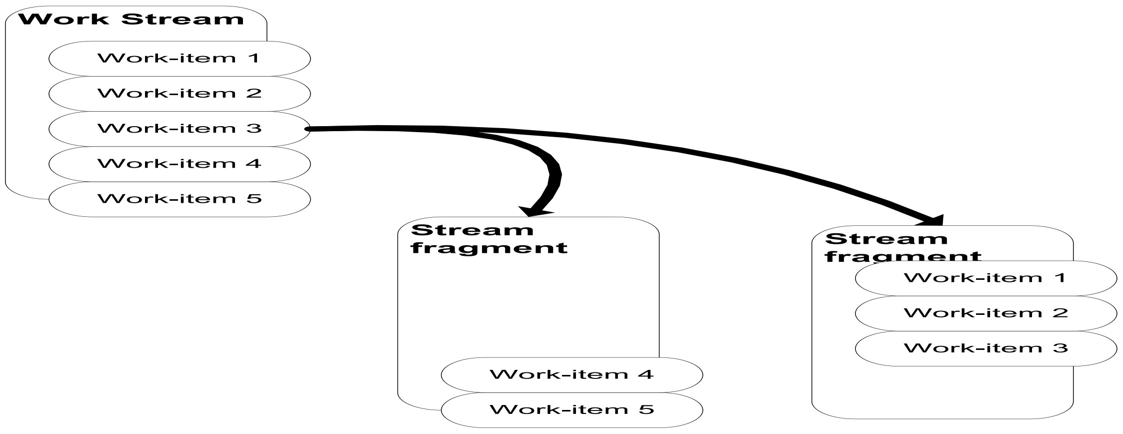


Figure 3: Fast batch, splitting a batch for parallel processing

Variant 1: Arbitrary fragmentation

Fast-batch improves throughput by creating three work streams that execute in parallel. Each work stream is still relatively large and incurs the overheads of a batch, including holding locks for the duration of the batch.

The size and content of each work-stream is arbitrary, leading to streams containing records that need access to the same resources as another stream. This means random serialisation dependencies between streams. Data locality and optimisation of processing is not addressed. In many cases benefits of a single batch are undermined with no net benefit. Specific problems include cache flushing, lock contention, unpredictable interactions, and resource contention.

Fast batch may be a cost effective way to process an existing batch in a shorter time. It is not a good pattern to adopt if there are other more effective alternatives.

Variant 2: Directed fragmentation

The purpose of splitting the batch into work-streams is to achieve increased parallelism. This approach splits the work using some attribute of the work-item, known as a **partitioning key**.

By segregating metadata and other resources across separate databases on separate machines using the partitioning key, problems of cache flushing, lock contention, unpredictable interactions, memory and resource contention are avoided between repositories but not within them. Each instance of the repository continues to single stream or suffer from the problems of variant one.

By choosing the smallest self-contained event as the batch-fragment boundary a **discrete event system** can be built. The item boundary is defined by the dependency of its fields on the primary key, which will dictate the whether the item can be processed independently of other records.

Tracking Progress

Iterative processes track their own progress using file pointers and loop indices that reflect progress. For persistent processes, for example a file to database load, progress must be persisted to support failure and restart. Persistent tracking uses one of three approaches:

1. Use a single transaction for the entire load.
2. Use multiple transactions and record progress as part of each transaction.
3. Use a staging table and check that the load has been successful before updating the main database.

Using a single transaction for the entire load means that the file is either loaded or not. It creates a volume dependency between batch size and resource consumption and suffers from scalability problems, unpredictable throughput and failure as the solution grows.

The approach is dependent on the database to scale its transaction subsystem, lock manager, and general processing to cope with large batches. All affected database tables are locked for the duration of the load and failure caused by a single item will roll back the entire batch, possibly without indicating the reason for failure. Rollback times are dependent on batch size and introduce a level of unpredictability to recovery times.[1]

Using multiple transactions and record progress as part of each transaction solves many of the problems of the last approach. Recovery is managed by checking previous progress in the tracking construct and continues from its point of failure. This is a very efficient mechanism with respect to disk IO and total work performed.

Using a staging table and check that the load has been successful before updating the main database allows the load to be done outside without transactional protection and completion is checked after the data is loaded. The approach has significant additional overheads to populate the temporary table, and may need to restart the temporary table load if loading fails.

Using ETL tools may bring benefit to the overall process and the approach avoids locking production database tables during the load process. It may benefit from using SQL to process sets of data from the staging table to the production tables.

[1] Most database systems optimise to commit successfully and are less efficient when rolling back transactions. This strategy is for fast forward performance and assuming that few transactions fail. This can cause a batch that took 2 hours to process before a failure to take over 2 hours to abort.

Conclusion

Batch processing makes good use of resources on a single machine by reduction of non-productive processing overheads; memory allocation, context switching, locking, and transaction management.

The control structure used to manage data used by a batch process being generally a file or tape, gives developers and users a high degree of comfort that they understand where the data is and that it is safe and accounted for.

As batches take longer to process they lock large parts of the database for longer and reduce concurrency. Any design that takes an arbitrary chunk of data to process in a transaction will be slower to complete as the chunk gets larger. This approach increases peak consumption of resource to process the work and makes testing difficult because what works with one volume of data may break when the volume increases beyond a threshold dictated by available resources and configuration (transaction log size, disk size, lock memory).

To avoid the problems of arbitrary transaction sizes a "batch" can be processed in fragments of a known size with predictable processing times and resource consumption.

Control

Batch processes must control their progress. If one set of processes have exclusive use of one repository the recovery from failure is a conceptually simple task. That would be to stop the batch, restore the repository to a known state, and restart the batch from that point.

This scenario is becoming increasingly unlikely because repositories are becoming too large to recover in this manner. Unlike the master-transaction update, modifications to a database require positive action to be reversed to a known point which takes time and incurs risk of further corruption. As the number of jobs increases it becomes less likely that all modifications are through one batch, or that the approach can be implemented within the constraints of business service level agreements.

To try to reduce the duration of long running batch work many batches are re-configured into multiple concurrent batch streams.

Exception management

Managing exceptions and times when parts of a batch require different processing (split paths). For example, in a batch of payments, some may require currency exchange, while others fail validation or process "normally". This situation can lead to three paths the entire batch following a single (expensive), elongated (super-set) path.

Event Solutions

Event systems react to external actions and process them as they arrive.

When each event is discrete it can be processed independently of other events (avoid cross event affinity) and paths (avoid pathway affinity) through the system.

By avoiding affinities, the designer has immense flexibility for different multi-threaded processing models, disaster recovery and performance. We use the term "work-stream" to refer to a stream of actions or items that can be processed independently but arrive as a logical unit or from a given source.

Example

A business process that applies four actions to each item presented; each action completes in one second. By the time the fourth item enters the process, the first item is already past actions 1,2, and 3. The entire process takes four seconds from end to end. If work-items arrive at a rate of one per second, the process will always be busy and will complete 4 seconds after the **last** item arrives. 5-items x 4-actions = 20 seconds.

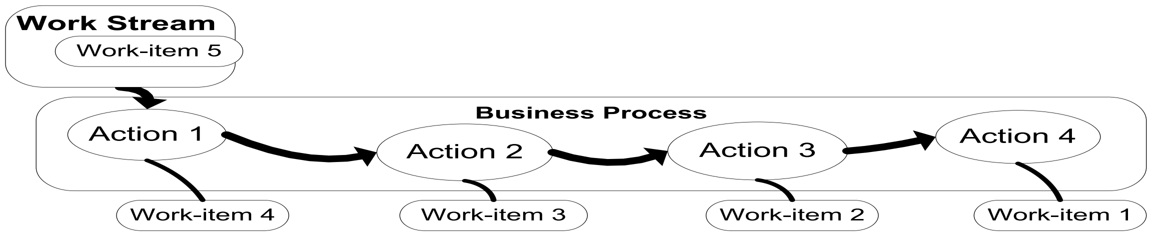


Figure 4: Event solutions

Solutions that execute actions serially are sensitive to synchronisation between components because while the first component is working the others are waiting.

If an additional action is placed at the start to create a batch of five items, then it will have to wait 5 seconds to receive the five items. Once the batch is created it must then be processed through the remaining actions (1,2,3,4), which will take around 5 seconds per action or 5-batchcreate + (4-actions x 5-items) = 25 seconds.

The end to end time being 5-batchcreate+20-processing=25 seconds.

Peak processing will hit each component in turn imposing high loads while other components are idle. The model is conceptually simple but expensive to maintain in a distributed environment where hardware and software licences are generally idle, but sized for the "wave".

Discrete event systems

A discrete event system is one where each event is processed independently of all other events. Each event contains all the information needed for it to be processed, or enriched if needed. All affinities (event order, time (temporal), application, and path affinity) between an individual event and another event or application instance are removed.

Discrete event systems have important advantages over the batch approach because they allow large numbers of events to be spread randomly across whatever processing platform is available. They allow the platform to be scaled during processing (adding and removing processing engines), and they simplify exception management by allowing items in exception to be processed out of order.

Transactional overheads for event systems are generally higher per-event than for batch; for example each event (in theory) will require a begin and end transaction and the associated processing. In batch systems the transaction can be the whole batch, which brings its own challenges but can reduce transactional overheads.

- Because each event is independent it can be processed by any suitable application. The lack of affinities allows any given load to be spread across all available resources, avoiding points of contention and creating an architecture capable of scaling horizontally as well as vertically.
- Transaction boundaries are simple to identify and communicate to developers and operations staff. Each event succeeds or fails on its own and can take whatever path is appropriate without recourse to any other event.

Variant 1: Dissimilar events

In this variant effort and duration of processing is different for each self-contained message. For example, requests for insurance quotes, or decisions from a decision engine can take seconds or minutes to complete depending on request parameters.

Dissimilar events to be load balanced, simply and effectively in a way that manages unpredictable loads caused by dissimilar events. The solution is to create a queue of events that is shared between the threads. The first engine to require work takes the next event from the queue. Required no complex feedback agents or mechanisms to determine which engine to load, and automatically manages failed engines.

Variant 2: Similar events

In this variant every event contains one self-contained record and uses the same resources as any other, for example bank transactions. This is an important variant because it is relatively easy to model and simulate, capacity and throughput are predictable, and cost-per-transaction is linear.

These characteristics allow each component of the solution after the first, to be scaled to match the output of the previous component. The system can be monitored for subtle exceptions, for example queues building up in the wrong place. Predictable lock duration and data access paths reduce the potential for deadlocks, and produce a predictable, consistent solution that makes excellent use of available resources in a parallel environment.

- Avoid marshalling data as batches
- Reduced complexity of processing components
- Allow the failure of one item to be managed independently of all others
- Reduce lock durations and reduced contention
- Throughput calculations more predictable
- Fully utilize the distributed model and leveraged horizontal scaling

Either variant can be made to scale in a linear way by reducing data sharing (read or write) to zero. Perfect for grid or message solutions using WebSphere MQ clustering to distribute events.

The Case for Real-Time

Batch processing is a simple model considered to have simple recovery characteristics understood by most practitioners. Adoption of process oriented architectures that incorporate distributed processing and multi-site SAN for disaster recovery reduce the benefits of batch to a point where the disadvantages are greater than the advantages.

In reality business processing is often simple to build but difficult to scale to large numbers of users with low latency requirements. The need for high concurrency, horizontal scalability, and highly integrated multi-user databases erodes many of the benefits of batch.

Controls and recovery management needed to solve event **problems** with batch **solutions** add additional complexities to control the progress and recovery needs.

Failures in batch systems cause work to be discarded.

A single "bad" item can cause the entire batch to fail. Choice of transaction boundaries can be confusing and difficult; failure of a single item means either the batch or only the item is aborted.

Batch should only be considered when the constraints mean that the overheads of events are not offset by the advantages.

Time	Affinity	Batched already
A time dependent processing window (Data warehouses, End of day reports)	A significant intra-item affinity (Item order, common data reference) Preservation of item order and the need to share data or a reference record between many threads undermines the event model to such an extent that batching may be the only way to achieve high throughput.	Data already batched (database extracts, Business controls dictate batch)

Discrete event systems have higher individual transaction cost but can utilize more hardware than an equivalent batch solution. These systems have greater scalability and avoid many of the problems associated with batch systems, including **capacity planning**, **availability** and **complex recovery operations**.

They create problems for **reconciliation** and can incur in a more obvious way, the problems of **synchronising multiple resources after a disaster**.

Reconciliation

Discrete event solutions mandate that relationships between events have no influence on where, how or when the event is processed. For example; it is important to know when all events from a particular file or external company are processed. This is not important in the context of the batch itself, but is important for business boundaries such as a business day, month, or year-end to be properly honoured.

Many business boundaries have traditionally dictated batch boundaries and this allows batch solutions to infer business information because of their processing model. Discrete event systems need to design another mechanism to enable them to properly honour the boundaries.

Geographic or other criteria, for example currency or location, is used to categorise events and it is the business categorisation that must be reconciled rather than an arbitrary batch.

Capacity planning

Discrete event systems with no sharing are linearly scalable and represent the simplest capacity planning model available. Shared resources affect solution scalability by introducing synchronisation. Exceeding the capacity of any solution will constrain the throughput and response time of that path (thread) and any synchronised threads will be constrained in unpredictable ways. Often databases are shared across threads, or applications.

Synchronisation occurring in the database, even when events are "independent" can affect the entire solution, causing applications to experience unpredictable waiting and contention, which effects the solution long before reaching the database capacity. Partitioning the data into independent databases can be used bypass this constraint to create a linearly scalable solution.

Mixing transactions that have different durations will tend to increase incidents of timeout and other contention related issues. Large transactions can benefit from fewer writes to disk, fewer coarser grained locks, and fewer program overheads for tracking batch progress. One such transaction, that takes an exclusive table lock for minutes can completely disrupt smaller transactions that require that resource. Optimal throughput is achieved when transactions are of a similar size.

Availability

Discrete event systems can receive events from many inputs and process them through many paths. This allows the model to tolerate one or more of paths becoming unavailable without preventing new work from being processed.

Batch processing models are built on a single path model and can rarely tolerate component failures while accepting new work. While a batch oriented system with many batches may be able to process some batches during a failure, it is unlikely to be able to process any part of the batch that failed until a full recovery is completed.

Event systems with one record per event can use transactional mechanisms to record their progress, and events routed around failed component instances. These abilities lead to a need for new definitions of availability to cater for reduction of capacity as well as complete failures.

Corruption of database or repository is the worst thing that can occur to a database. It involves the data becoming invalid and recovery from this type of failure is out of scope for this paper and has nothing to do with using batch or event models. Single threaded batch systems with exclusive use of an isolated repository are simple to recover but rely on hardware for their capacity and have single points of failure at every stage of processing. Multi-stream batch operations with more than one repository, or that share repositories cannot recover by scrapping all updates and recovering from a backup. The time to recover a large database and re-run a significant batch is often not available.

With the exception of pure, isolated batch systems, batch and event systems are equally difficult to recover and neither are simple operations.

Loss of repository

Database checkpoints are used to recover a single (small) database but are of little use when synchronising state across many databases. In a batch solution with one database checkpoint recovery is a viable option, this is not the case for any of the distributed options, including partitioned batches. Another approach is the fix-forward approach, which implies that processing is not rolled back beyond the current transaction boundaries. Instead the processing is completed and problems in the outcome are analysed and fixed as a new operation. 24x7 systems increasingly cannot tolerate the downtime associated with discarding a quantity of work and the associated recovery operations, including recovery from checkpoint. Use of storage area networks (SAN) implemented across multiple sites means repositories can be mirrored to a consistent, near real time, off site backup.

Terms

When designing any transaction system there are a few points that should be considered.

Serial File

A serial file, also known as a "flat file" or "heap table" is one that has no indexing structures but may have a natural order, for example time. These types of file are processed in a serial or end-to-end (top to bottom) manner. These types of file can be sorted on any key, creating a file with a physical order based on that key. Common examples are error logs and most batch files.

Locality of data

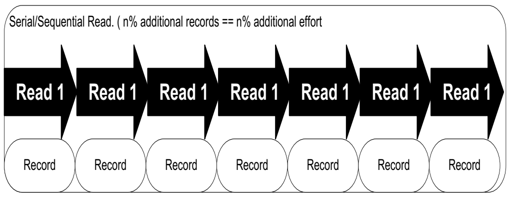
Programs do not have to wait for data that is readily accessible; work is performed faster[2] with less waiting time. A program processing a serial file in physical key sequence benefits from local data. The same program processing the same data in a different order suffers from non-locality of data and additional overheads associated with scanning the file for the next record to process.

The Master-Transaction pattern demonstrates the importance of data locality. Individual transactions are costlier to process when the transaction stream is not in the same order as the master stream and can lead to volume dependency problems.

Cache flushing

Multi-threading batches can undermine caching because they use the same cache for different parts of the batch. For example, a stream using names A-M, and a second stream using names N-Z will continually overwrite each other's' cached data. Neither benefits from a cache that always contains the wrong information. Processor use increases to accommodate cache-flush activity and low hit rates, while throughput goes down.

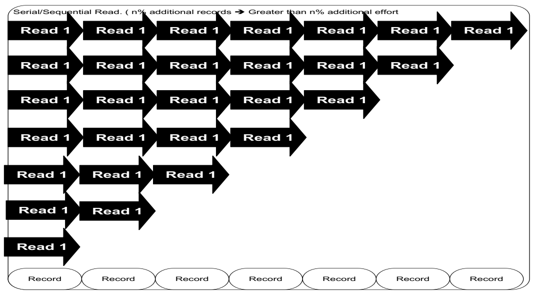
Volume Independence


Serial/Sequential Read. (n% additional records == n% additional effort
Read 1
Record

Volume dependence describes situations where small changes in the number of records in a batch to have disproportionately greater impact on the effort needed to process the batch. It is often caused by the way in which data is accessed, or saturation of a resource such as cache memory.

All designs have some volume dependency because the amount of work to perform affects the time and resource needed to perform it.

A design is "volume dependent" if its dependency is non-linear. In this example, an application reads records in sequence from a file and presents them to a screen. It takes longer to complete a larger file.


Serial/Sequential Read. (n% additional records ➔ Greater than n% additional effort
Read 1
Record

A second program presents the same records to the screen, but searches for each record from the start of the file. Additional effort is required when three new records are added. For program one the effort is linear, but for program two it is not proportional to the change in file size.

Concurrency: Big locks or small locks?

Concurrency is the ability of a repository (database) or process to support more than one thread of activity at a time. Key aspects of concurrency are the efficiency of resource sharing and efficiency of work scheduling. The

[2] Relational databases support data locality in many ways, the most obvious being the least recently used cache and the read-ahead functionality. Compilers that optimise code will expend serious efforts to ensure that data needed by an instruction is local and "cheap" to access.

greatest influence, however, is the need for threads to share data. The degree of need is dependent on the work that needs to be completed.

If a process is the sole user of a resource, it will not require serialisation overheads such as locking and context switching[3]. Deadlocks and lock waits are minimised. The approach uses very granular locks to avoid lock management overhead. Database management systems allow locking of whole tables or databases to efficiently support this style of processing.

The price of avoiding locking overheads is that only one thread of activity can be active at any given time. This translates to one thread on one machine and complete reliance on hardware capacity to provide the needed throughput.

Situations in which more than one process shares one or more resources can also benefit from a coarse grained locking scheme, but resources are likely to be unavailable when needed. This approach has a propensity to generate high lock wait times, and thus increased latency. Therefore, only situations constrained by machine capacity are likely to benefit.

The cost of avoiding locking overheads on the CPU and other resources is to reduce concurrency.

The alternative is to lock as little of a resource as possible for as little time as possible. This approach makes resources available more often and supports higher numbers of concurrent threads.

The cost of increasing concurrency is to increase lock management overhead for high numbers of fine grained locks and increase the possibility of deadlocks.

Timeouts

Timeout is a safety mechanism to prevent programs waiting forever for unavailable resources. A ***timeout is not a deadlock***; some lock managers do not recognise deadlocks and avoid the cost of deadlock searches in favour of allowing processes to timeout.

The only way to resolve a deadlock is to abort a transaction. Lock managers that detect deadlocks perform this action on behalf of one of the parties. Those lock managers that do not detect deadlocks rely on well behaved applications to perform the abort. Because of this lazy approach to deadlock management it is worth retrying a timeout (after aborting and reworking the transaction) in case it is a deadlock in disguise.

There is no reason to repeat actions that have properly expired. Timeout settings should be set high enough that they do not occur during normal processing but be "reasonable" wait times for when something is wrong. For example, IP socket connection requests will timeout if the process listening to the port is busy or the port is full. Retrying may establish a connection but incurring a timeout may indicate capacity constraints.

[3] A **context switch** is the process of storing and restoring the state of a CPU so that multiple processes can share a single CPU resource. The context switch is an essential feature of a multi-threaded environment and is usually computationally intensive.

Avoiding deadlocks

A deadlock occurs when one context owns a resource (A) and wants resource (B) while a second context is already holding (B) and wants (A). The deadlock develops because neither process will give up its resource until it gets the resource it wants.

There are many types of deadlock and reasons for them occurring[4]. They are a form of contention that increases latency and resource consumption, and are likely to occur as increased numbers of threads perform work on the same set of resources. **Resource deadlocks** are avoided by using agreed "data access paths", for example locking resources in alphabetical order.

Process 1	Process 2	Conclusion

Begin Transaction
`Update AAA;`
`Update BBB;`
`Commit;`
	Begin Transaction
	`Update BBB;`
	`Update AAA;`
	`Commit;`
		This will suffer from high numbers of deadlocks if the two processes run together.

Begin Transaction
`Update AAA;`
`Update BBB;`
`Commit;`
	Begin Transaction
	`Update AAA;`
	`Update BBB;`
	`Commit;`
		This will never deadlock.

Isolation-escalation deadlocks are caused by increasing the isolation of an already locked resource, for example reading a record (shared lock) before updating it (exclusive lock).

```
Select number from next_numbers where series = 'MYSERIES';
Update next_numbers set number = number + 1 where series = 'MYSERIES';
```

In this example a next-number routine reads the number before updating it in the same transaction. This leads to high numbers of deadlocks as the number of concurrent processes increase. The solution is to achieve the maximum isolation required by the whole transaction immediately. In this example we achieve maximum isolation early by updating the count before reading it. This approach will never deadlock.

```
Update next_numbers set next_no = next_no + 1 where next_series = 'MYNUMBERS';
Select next_no from next_numbers where next_series = 'MYNUMBERS';
```

[4] Key forms of deadlock are resource as described above, isolation escalation where two holders of shared (read) locks attempt to convert to exclusive (update) locks, and granularity escalation when processes attempt to escalate from locking, for example, a record to locking a file.

Contention

Contention occurs processes contend for exclusive or incompatible access to a resource. Examples occur when updating, or trying to read while someone else is updating.

Synchronisation of access requires one process to wait while the other completes. When this occurs to excess the symptoms are timeout errors, reduced throughput, or increased latency. It is considered here because of its effect on the throughput and response profile of a component. Contention can be minimised or avoided by;

- Requests designed to require the same time and resources to be processed.
- Components locking fine grained resources
- Reduce the number of clients
- Deadlock contention is reduced by accessing resources in the same order, avoiding lock isolation escalation and making transactions the same size, either as quick (small) as possible for multi-client environments or large as needed for low client environments.

Performance

Computer system **performance** refers to the efficiency of the component to process work. Effective operation is a comparison of useful work against cost. Typically, the rate of work (throughput) or duration of work (response time) is used to calculate the capacity of the solution. **Throughput** is a measure of the number of work-items processed in a given time window.

$$Throughput = \left(\frac{events}{finishTm - StartTm}\right)$$

In the formula above startTm is the time the first event arrives, and finishTm is when the last departs. This formula is the traditional way to calculate "throughput". It accounts for time when the system has work to perform and is busy working on the event "batch". The approach is useful when calculating batch windows and can be used to calculate the theoretical throughput of a system. For event systems it under-estimates the ability of the system to perform work.

When used to calculate batch throughput the start time is from the first event entering the batch and the finishTm is when the last item leaves the system.

In this version utilisation of effective availability is accounted for by including the time it takes to construct the batch. This assumes that if the batching was not performed the event could start being processed. The formula is more able to properly compare batch and event system throughput. Components optimised for throughput may not give the best possible response times for a given request. It is possible to ask the IBM DB2 query engine to optimise for response rather than the default throughput optimisation.

Response time is a measure of the elapsed round-trip time between a request and receipt of a response. The time taken for a system to respond is *latency*. Throughput calculations do not consider response time; however, response time (latency) calculations are meaningless without an associated throughput because of the relationship between latency and system saturation.

$$responseTm = ResponseSentTm - RequestReceivedTm$$