Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed

Martin Kleppmann Alastair R. Beresford Boerge Svingen

For almost half a century, ACID transactions have been the abstraction of choice for ensuring consistency in data storage systems. The well-known *atomicity* property ensures that either all or none of a transaction's writes take effect in the case of a failure; *isolation* prevents interference from concurrently running transactions; and *durability* ensures that writes made by committed transactions are not lost in the case of a failure.

While transactions work well within the scope of a single database product, transactions that span several different data storage products by distinct vendors have been very problematic: many storage systems do not support them, and those that do often perform poorly. Today, large-scale applications are often implemented by combining several distinct data storage technologies that are optimised for different access patterns. In our experience, distributed transactions have failed to gain adoption in most such settings, and most large-scale applications instead rely on ad-hoc, unreliable approaches for maintaining the consistency of their data systems.

However, in recent years we have observed the increasing use of *event logs* as a data management mechanism in large-scale applications. This trend includes the *event sourcing* approach to data modelling, the use of change data capture systems, and the increasing popularity of log-based publish/subscribe systems such as Apache Kafka. Although many databases use logs internally, e.g. as write-ahead logs or replication logs, this new generation of log-based systems is different: rather than using logs as an implementation detail, they raise them to the level of the *application programming model*.

As this approach uses application-defined events to solve problems that traditionally fall in the transaction-processing domain, we name it *online event processing* (OLEP) to contrast with OLTP. In this article, we explain the reasons for the emergence of OLEP, and show how it allows applications to guarantee strong consistency properties across heterogeneous data systems, without resorting to atomic commit protocols or distributed locking. The architecture of OLEP systems allows them to achieve consistently high performance, fault tolerance, and scalability.

Application Architecture Today: Polyglot Persistence

Different data storage systems are designed for different access patterns, and there is no single "one size fits all" storage technology that is able to efficiently serve all possible uses of data. Consequently, many applications today use a combination of several different storage technologies, an approach sometimes known as "polyglot persistence". For example:

Full-text search. When users need to perform keyword search on a dataset, e.g. a product catalog, a full-text search index is required. Although some relational databases such as PostgreSQL include a basic full-text indexing feature, more advanced uses generally require a dedicated search server such as ElasticSearch. To improve the indexing or search result ranking algorithms the search engine's indexes may need to be rebuilt from time to time.

Data warehousing. Most enterprises export operational data from their OLTP databases and load it into a data warehouse for business analytics. The storage layouts that perform well for such analytic workloads, such as column-oriented encoding, are very different from those of OLTP storage engines, necessitating the use of distinct systems.

Stream processing. Message brokers allow an application to subscribe to a stream of events as they happen (e.g. representing the actions of users on a website), and stream processors provide infrastructure for interpreting and reacting to those streams (e.g. detecting patterns of fraud or abuse).

Application-level caching. In order to improve the performance of read-only requests, applications often maintain caches of frequently-accessed objects (e.g. in memcached). When the underlying data changes, applications employ custom logic to update the affected cache entries accordingly.

Note that these storage systems are not fully independent of each other. Rather, it is common for one system to hold a copy or materialized view of data in another system. Thus, when data in one system is updated, it often also needs to be updated in another, as illustrated in Figure 1.

OLTP Transactions Are Predefined and Short

In the traditional view, as implemented by most relational database products today, a transaction is an interactive session in which a client's queries and data modification commands are interleaved with arbitrary processing and business logic on the client. Moreover, there is no time limit for the duration of a transaction, since the session traditionally may have included human interaction.

However, reality today looks different. Most OLTP database transactions are triggered by a user request made

via HTTP to a web application or web service. In the vast majority of applications, the span of a transaction extends no longer than the handling of a single HTTP request: that is, by the time the service sends its response to the user, any transactions on the underlying databases have already been committed or aborted. In a user workflow that spans several HTTP requests (for example, adding an item to a cart, going to checkout, confirming shipping address, entering payment details, giving a final confirmation), there is no transaction that spans the entire user workflow; there are only short, non-interactive transactions to handle a single step of the workflow.

Moreover, an OLTP system generally executes a fairly small set of known transaction patterns. On this basis, some database systems encapsulate the business logic of transactions as *stored procedures* that are registered ahead of time by the application. To execute a transaction, a stored procedure is invoked with certain input parameters, and the procedure then runs to completion on a single execution thread without communicating with any nodes outside of the database.

Heterogeneous Distributed Transactions Are Problematic

It is important to distinguish two types of distributed transaction:

Homogeneous distributed transactions are those in which the participating nodes are all running the same database software. For example, Google's Spanner and VoltDB are recent database systems that support homogeneous distributed transactions.

Heterogeneous distributed transactions span several different storage technologies by distinct vendors. For example, the XA standard defines a transaction model for performing two-phase commit (2PC) across heterogeneous systems, and the Java Transaction API (JTA) makes XA available to Java applications.

While some homogeneous transaction implementations have proved successful, heterogeneous transactions continue to be very problematic. By their nature, they can only rely on a lowest common denominator of participating systems. For example, XA transactions block execution if the application process fails during the *prepare* phase; moreover, XA provides no deadlock detection, and no support for optimistic concurrency control schemes [3].

Many of the systems listed above, such as search indexes, do not support XA or any other heterogeneous transaction model. Thus, ensuring the atomicity of writes across different storage technologies remains a challenging problem for applications.

Building upon Event Logs

Figure 1 shows an example of polyglot persistence: an application that needs to maintain records in two separate storage systems, such as an OLTP database (e.g. an RDBMS) and a full-text search server. If heterogeneous distributed transactions are available, the system can ensure atomicity of writes across the two systems. However, most search servers do not support distributed transactions, leaving the system vulnerable to potential inconsistencies:

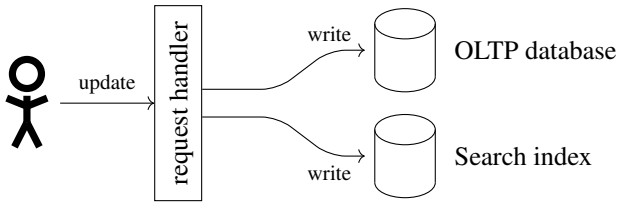

Figure 1: An updated record needs to be written both to a database and to a full-text search index, necessitating atomic commit across the two systems.

1. **Non-atomic writes.** If a failure occurs, a record may be written to one of the systems but not the other, leaving them inconsistent with each other.

2. **Different order of writes.** If there are two concurrent update requests A and B for the same record, one system may process them in the order A, B while the other system processes them in the order B, A. Thus, the systems may disagree on which write was the latest, leaving them inconsistent.

Figure 2 presents a simple solution to these problems: when the application wants to update a record, rather than performing direct writes to the two storage systems, it appends an *update event* to a log. The database and the search index each subscribe to this log, and write updates to their storage in the order they appear in the log [4]. In effect, the database and the search index are *materialized views* onto the sequence of events in the log. This approach solves both of the aforementioned problems:

1. Appending a single event to a log is atomic; thus, either both subscribers see an event, or neither does. If a subscriber fails and recovers, it resumes processing any events that it has not processed previously. Thus, if an update is written to the log, it will eventually be processed by all subscribers.

2. All subscribers of the log see its events in the same order. Thus, each of the storage systems will write records in the same serial order.

In this example, the log only serializes writes, but the application may read from the storage systems at any time. Since the log subscribers are asynchronous, reading the index may return a record that does not yet exist in the database, or vice versa; such transient inconsistencies are not a problem for many applications. For those applications that require it, reads can also be serialized through the log; we discuss an example of this later.

The Log Abstraction

There are several log implementations that can serve this role, including Apache Kafka, CORFU (from Microsoft Research), Apache Pulsar, and Facebook's LogDevice. The log abstraction we require has the following properties:

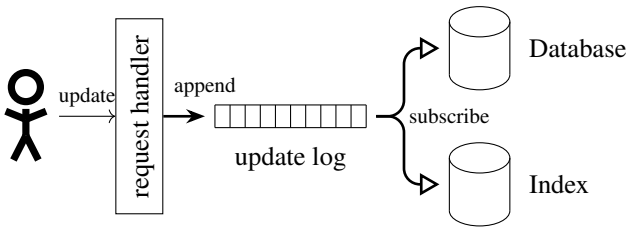

Figure 2: By sequencing updates through a log, the database and the search index apply the same set of writes in the same order, keeping them consistent with each other.

Durable. The log is written to disk and replicated to several nodes, ensuring that no events are lost in a failure.

Append-only. New events can only be added to the log by appending them at the end. Besides appending, the log may allow old events to be discarded, e.g. by truncating log segments older than some retention period, or by performing key-based log compaction.

Sequential reads. All subscribers of the log see the same events in the same order. Each event is assigned a monotonically increasing *log sequence number* (LSN). A subscriber reads the log by starting from a specified LSN, and then receiving all subsequent events in log order.

Fault-tolerant. The log remains highly available for reads and writes in the presence of failures.

Partitioned. An individual log may have a maximum throughput it can support (e.g. bounded by the throughput of a single network interface or a single disk). However, we assume that the system can scale linearly by having many *partitions*, that is, many independent logs that can be distributed across many machines. We assume there is no ordering guarantee across different log partitions. Multiple logical logs may be multiplexed into a single physical log partition.

For subscribers of a log we make the following assumptions:

- A subscriber may maintain state (e.g. a database) that is read and updated based on the events in the log, and that survives crashes. Moreover, a subscriber may append further events to any log (including its own input).
- A subscriber periodically checkpoints the latest LSN it has processed to stable storage. When a subscriber crashes, upon recovery it resumes processing from the latest checkpointed LSN. Thus, a subscriber may process some events twice (those processed between the last checkpoint and the crash), but it never skips any events. We say that events in the log are processed *at least once* by each subscriber.
- The events in a single log partition are processed sequentially on a single thread, using deterministic logic. Thus, if a subscriber crashes and restarts, it may append duplicate events to other logs.

These assumptions are satisfied by existing log-based stream processing frameworks, such as Kafka Streams and Apache Samza. Updating state deterministically based on an ordered log corresponds to the classic *state machine replication* principle [5]. Since it is possible for an event to be processed more than once when recovering from a failure, we also require state updates to be *idempotent*.

Aside: Exactly-Once Semantics

Some log-based stream processors such as Apache Flink support so-called *exactly-once semantics*, which means that even though an event may be processed more than once, the effect of the processing will be the same as if it had been processed exactly once. This behavior is implemented by managing side-effects within the processing framework, and atomically committing these side-effects together with the checkpoint that marks a section of the log as processed. However, when a log consumer writes to external storage systems, like in Figure 2, exactly-once semantics cannot be ensured, since doing so would require a heterogeneous atomic commit protocol across the stream processor and the storage system, which is not available on many storage systems such as full-text search indexes. Thus, frameworks with exactly-once semantics still exhibit at-least-once processing when interacting with external storage, and rely on idempotence to eliminate the effects of duplicate processing.

Atomicity and Enforcing Constraints

A classic example where atomicity is required is in a banking/payments system, where a transfer of funds from one account to another account must happen atomically, even if the two accounts are stored on different nodes. Moreover, such a system typically needs to maintain consistency properties or invariants, e.g. that an account cannot be overdrawn by more than some set limit. In Figure 3 we show how such a payments application can be implemented using the OLEP approach instead of distributed transactions. It works as follows:

1. When a user wishes to transfer funds from some source account to some destination account, they first append a *payment request* event to the log of the source account. This event merely indicates the *intention* to transfer funds; it does not yet imply that the transfer has been successful. The event carries a unique ID to identify the request.

2. A single-threaded *payment executor* process subscribes to the source account log. It maintains a database containing transactions on the source account, and the current balance. This process deterministically checks whether the payment request should be allowed, based on the current balance and perhaps other factors. This log consumer is very similar to the execution of a stored procedure.

3. If the executor decides to grant the payment request, it writes that fact to its local database, and appends

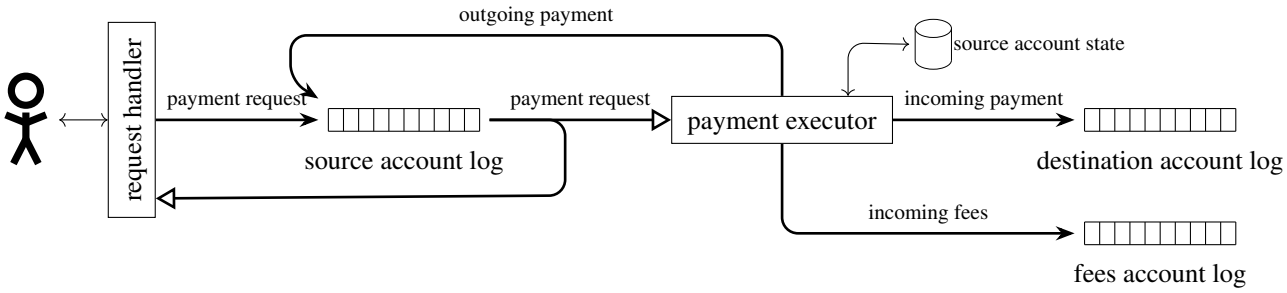

Figure 3: Flow of events in a financial payments system. Arrows with a → arrowhead denote appending an event to a log, while arrows like ⊳ denote subscribing to the events in a log.

events to several different logs: as a minimum, an outgoing payment event to the source account log, and an incoming payment event to the log for the destination account. If a fee is due for this payment (e.g. because of an overdrawn account or currency conversion), an additional outgoing payment event for the fees may be appended to the source account log, and a corresponding incoming payment event may be appended to the log of a fees account. The original event ID is included in all of these generated events, so that their origin can be traced.

4. Since the executor subscribes to the source account log, the outgoing payment event will be delivered back to the executor again. It uses the unique event ID to determine that it has already processed this payment and recorded it in its database.

5. The payment events on other accounts, such as the incoming payment on the destination account, are similarly processed by single-threaded executors, with a separate executor per account. The event processing is made idempotent by suppressing duplicates based on the original event ID.

6. The server handling the user's request may also subscribe to the source account log, and thus be notified when the payment request has been processed. This status information can be returned to the user.

If the payment executor crashes and restarts, it may reprocess some payment requests that were partially processed before the crash. Since the executor is deterministic, upon recovery it will make the same decisions to approve or decline requests, and thus potentially append duplicate payment events to the source, destination, and fees logs. However, based on the ID in the events it is easy for downstream processes to detect and ignore such duplicates.

Multi-partition processing

In this payment example, each account has a separate log, and thus may be stored on a different node. Moreover, each payment executor need only subscribe to events from a single account, and different accounts are handled by different executors. These factors allow the system to scale linearly to arbitrary numbers of accounts.

In this example, the decision of whether to allow the payment request is conditional only on the balance of the source account; we assume that the payment into the destination account always succeeds, since its balance can only increase. For this reason, the payment executor need only serialize the payment request with respect to other events in the source account. If other log partitions need to contribute to the decision, the approval of the payment request can be performed as a multi-stage process in which each stage serializes the request with respect to a particular log.

Splitting a "transaction" into a multi-stage pipeline of stream processors allows each stage to make progress based only on local data; it ensures that one partition is never blocked waiting for communication or coordination with another partition. Unlike multi-partition transactions, which often impose a scalability bottleneck in distributed transaction implementations, this pipelined design allows OLEP systems to scale linearly.

Advantages of Event Processing

Besides this scalability advantage, developing applications in an OLEP style has several further advantages:

- Since every log can support many independent subscribers, it is easy to create new derived views or services based on an event log. For example, in the payment scenario of Figure 3, a new account log subscriber could send a push notification to a customer's smartphone if a certain spending limit on their credit card is reached. A new search index or view over an existing dataset can be built simply by consuming the event log from beginning to end [3].

- If an application bug causes bad events to be appended to a log, it is fairly easy to recover: subscribers can be programmed to ignore the incorrect events, and any views derived from the events can be recomputed. In contrast, in a database that supports arbitrary insertions, updates, and deletes, it is much harder to recover from incorrect writes, potentially requiring the database to be restored from a backup.

- Similarly, debugging is much easier with an append-only log than a mutable database, because events can be replayed in order to diagnose what happened in a particular situation.

- For data modeling purposes, an append-only event log is increasingly preferred over free-form database mutations; this approach is known in the Domain-Driven

Design community as *event sourcing* [2]. The rationale is that events capture state transitions and business processes more accurately than insert/update/delete operations on tables, and those state updates are better described as side-effects resulting from processing an event. For example, the event *"student cancelled their course enrollment"* clearly expresses intent, whereas the side-effects *"one row was deleted from the enrollments table, and one cancellation reason was added to the student feedback table"* are much less clear.

- From a data analysis point of view, an event log is more valuable than the state in a database. For example, in an e-commerce setting, it is valuable for business analysts to see not only the final state of the cart at checkout, but rather the full sequence of items added to and removed from the cart, since the removed items carry information too (e.g. that one product is a substitute for another, or that the customer may return to buy a certain item on a later occasion).

- With a distributed transaction, if any one of the participating nodes is unavailable, the whole transaction must abort, so failures are amplified. In contrast, if a log has multiple subscribers, they make progress independently from each other: if one subscriber fails, that does not impede the operation of the publisher or other subscribers, so faults are contained.

Disadvantages of OLEP approach

In the above examples, log consumers update the state in datastores (the database and search index in Figure 2; the account balances and account statements in Figure 3). While the OLEP approach ensures that every event in the log will eventually be processed by every consumer, even in the face of crashes, there is no upper bound on the time until an event is processed.

This means that if a client reads from two different datastores that are updated by two different consumers or log partitions, the values read by the client may be inconsistent with each other. For example, reading the source and destination account of a payment may return the source account at a time after the payment has been processed, but the destination account at a time before it has been processed. Thus, even though the accounts will eventually converge towards a consistent state, they may be inconsistent when read at one particular point in time.

Note that in an ACID context, preventing this anomaly falls under the heading of *isolation*, not *atomicity*; a system with atomicity alone does not guarantee that two accounts will be read in a consistent state.

At present, the OLEP approach does not provide isolation for read requests that are sent directly to datastores (rather than being serialized through the log). We hope that future research will enable isolation levels such as *snapshot isolation* across datastores that are updated from a log.

Case Study: The New York Times

The New York Times maintains all textual content published since the newspaper's founding in 1851 in a single log partition in Apache Kafka [6]. Image files are stored in a separate system, but URLs and captions of images are also stored as log events.

Whenever a piece of content (known as an *asset*) is published or updated, an event is appended to this log. Several systems subscribe to this log: for example, the full text of each article is written to an indexing service for full-text search; various cached pages (e.g. the list of articles with a particular tag, or all pieces by a particular author) need to be updated; and personalization systems notify readers who may be interested in a new article.

Each asset is given a unique identifier, and an event may create or update an asset with a given ID. Moreover, an event may reference the identifiers of other assets — much like a normalised schema in a relational database, where one record may reference the primary key of another record. For example, an image (with caption and other metadata) is an asset that may be referenced by one or more articles.

The order of events in the log satisfies two rules:

1. Whenever one asset references another, the event that publishes the referenced asset appears in the log before the referencing asset.

2. When an asset is updated, the latest version is the one published by the latest event in the log.

For example, an editor might publish an image, and then update an article to reference the image. Every consumer of the log then passes through three states in sequence:

1. The old version of the article (not referencing the image) exists;

2. The image also exists, but is not yet referenced by any article;

3. The article and image both exist, with the article referencing the image.

Different log consumers will pass through these three states at different times, but in the same order. The log order ensures that no consumer is ever in a state where the article references an image that does not yet exist, ensuring referential integrity.

Moreover, whenever an image or caption is updated, all articles referencing that image need to be updated in caches and search indexes. This can easily be achieved with a log consumer that uses a database to keep track of references between articles and images. This consistency model lends itself very easily to a log, and it provides most of the benefits of distributed transactions without the performance costs.

Further details on the New York Times approach appear in a blog post [6].

Conclusions

Support for distributed transactions across heterogeneous storage technologies is either non-existent, or suffers from

poor operational and performance characteristics. In contrast, Online Event Processing (OLEP) is increasingly used to provide good performance and strong consistency guarantees in such settings.

In data systems it is very common for logs (e.g. write-ahead logs) to be used as an internal implementation detail. However, the OLEP approach is different: it uses event logs, rather than transactions, as the *primary application programming model* for data management. Traditional databases are still used, but their writes come from a log rather than directly from the application. This approach has been explored by several influential figures in industry, such as Jay Kreps [4], Martin Fowler [2], and Greg Young under names such as *event sourcing* and *Command/Query Responsibility Segregation* (CQRS) [1, 7].

The use of OLEP is not simply pragmatism on the part of developers, but rather offers a number of advantages, including linear scalability, a means of effectively managing polyglot persistence, support for incremental development where new application features or storage technologies are added or removed iteratively, excellent support for debugging via direct access to the event log, and improved availability (because running nodes can continue to make progress when other nodes have failed).

Consequently we expect OLEP will increasingly be used to provide strong consistency in large-scale systems that use heterogeneous storage technologies.

Acknowledgements

This work was supported by a grant from The Boeing Company. Thank you to Pat Helland for feedback on a draft of this article.

References

[1] Dominic Betts, Julián Domínguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing*. Microsoft Patterns & Practices, 2012. ISBN 978-1-62114-016-0. URL `http://aka.ms/cqrs`.

[2] Martin Fowler. Event Sourcing, December 2005. URL `https://www.martinfowler.com/eaaDev/EventSourcing.html`.

[3] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, April 2017. ISBN 978-1-4493-7332-0.

[4] Jay Kreps. The Log: What every software engineer should know about real-time data's unifying abstraction, December 2013. URL `http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`.

[5] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.

[6] Boerge Svingen. Publishing with Apache Kafka at The New York Times, September 2017. URL `https://open.nytimes.com/publishing-with-apache-kafka-at-the-new-york-times-7f0e3b7d2077`.

[7] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, February 2013. ISBN 0321834577.